LEVEL 68

MULTICS ERROR MESSAGES: PRIMER AND REFERENCE MANUAL

SUBJECT

Description of the Multics System Error Messages, Including Troubleshooting Information

SOFTWARE SUPPORTED

Multics Software Release 8.0

ORDER NUMBER

CH26-00

September 1980

Honeywell

PREFACE

The purpose of this document is to aid you in understanding errors that may occur during use of Multics. The standard Multics error reporting mechanism is introduced, and the meaning of error messages is explained. Possible causes of many errors are discussed, and when applicable, methods for recovering from or circumventing an error are presented. The majority of this document is intended to be somewhat tutorial; the final section is included strictly as a reference.

This document is intended for use by novice users or programmers new to the Multics programming environment. As a result, the kinds of errors covered and the examples given have been tailored to this audience. However, some background knowledge of the Multics environment is assumed. (For an introduction to the Multics environment, see the New Users' Introduction to Multics--Part I (Order No. CH24) and--Part II (Order No. CH25). In this document, this book is referred to as the New Users' Intro.

The discussion of error conditions is divided into two parts that are tutorial introductions to and explanations of error messages. These sections (2 and 3) contain examples of some of the most commonly occurring errors of the particular class. Terminal scripts, along with descriptive commentaries, are used to present typical examples of the error, the methods used to determine the actual cause of the error, and means for recovering from the error.

We recommend at least one reading of Sections 1 to 3 to provide some familiarity with errors that otherwise could only be obtained by a long period of actual use. Furthermore, many errors have related causes and consequently related methods for analysis, and since, in general, a topic is explained in full detail only once, a thorough reading will help you see how a specific technique or error falls into the overall scheme. Finally, you should find a number of worthwhile hints that enable you to avoid problems in the first place.

NOTE: The listing of error messages in Section 4 is as complete as possible based on currently available information. As further information becomes available, updates will be published. updates to this manual.

The reference section (4) should be useful when you encounter specific messages and need specific information.

Section 1 is a brief introduction.

Section 2 describes errors that ANYONE might get, regardless of their level of expertise. These include command processor errors, command errors, and error messages that indicate system problems.

Section 3 discusses errors that only a PROGRAMMER is likely to get. These include some of the more unusual command processor errors, most of the fault messages, and the fatal process errors.

File No.: 1L13

CH26-00

Section 4 is a reference section that includes an alphabetical listing of the error messages (by long message and code) and what they are likely to mean.

The Multics Programmers' Manual (MPM) is the primary Multics reference document.

The MPM consists of the following individual manuals:

Reference Guide	(Order No. AG91)
Commands and Active Functions	(Order No. AG92)
Subroutines	(Order No. AG93)
Subsystem Writers' Guide	(Order No. AK92)
Peripheral Input/Output	(Order No. AX49)
Communications Input/Output	(Order No. CC92)

Throughout this manual, references are made to both the MPM Commands and Active Functions manual and the Reference Guide. For convenience, these references will be as follows:

MPM Commands
MPM Reference Guide

The Multics operating system is sometimes referred to in this manual as either "Multics" or "the system."

CONTENTS

SECTION 1

INTRODUCTION

This manual is intended to be of use to all Multics users -- specifically, the document is broken down into two tutorial sections: a nonprogrammers' section and a programmers' section. The remainder of this manual is a reference section for all users, listing the system error messages alphabetically and including diagnostic information.

Multics gives you an error message:

- When you try to do something "illegal" -- it cannot be done.
- When you type in a command line that makes no sense (to the system).

Errors are often caused by a typing error, or by your losing track of what precisely you wanted to do. Look at what you typed that caused the error, and try again.

Where do error messages come from? A certain set of standardized messages are shared by different situations and printed when appropriate for a particular error. These messages are located in a place called the error_table_, which is generally accessible to programmers.

In addition, many subsystems and other programs print error messages and comments "on their own", incorporating and/or independent of the standard system messages.

The error messages fall into four categories:

- command processor errors
- command errors
- execution errors (faults)
- fatal process errors

Command processor errors include all errors that arise in the interpretation of commands and the formation of command names and command lines. This type of error is received by all kinds of users. A common reason for receiving command processor errors is that the command line contains mistyped characters or words; essentially, in some manner the line is malformed and cannot be interpreted by the command processor.

Command errors are those that are detected by commands themselves; for example, an editor reports an error when a request is issued to read a nonexistent file. Command processor errors and command errors are _intentionally_ printed; that is, these user-caused errors are so common that their frequent occurence is anticipated and the messages are planned to inform you of the problem.

Execution errors -- hardware conditions -- arise during the execution of programs. When an error of this kind (also known as a _fault_) occurs in _your_ program, you fix the error by debugging the program. If the problem occurs because you ran out of storage space or have damaged segments, you may have to delete or change something, or ask for help from your supervisor or project administrator. If double-checking reveals no problem within your program, the cause of the error may be a _system_ bug. When the error is caused by a system program (_this_ is unintentional and shows a bug in the system program or a genuine _system_ problem), you are not to blame! You have not caused it by any action on your part, and you cannot fix it (all users are susceptible to system problems!).

Fatal process errors are usually caused by a serious bug in your program that must be found. Fatal process errors are similar in origin to execution errors, but cause your process to be aborted and a new process is automatically created for you (by the way, you did not break any machinery!). Because a fatal process error discards all the normally available debugging information, it can be very difficult to find the cause; generally, the only way to do so is to determine the last thing the program did before the error and examine the statements following this point.

All the language- and subsystem-specific manuals (e.g., the _Qedx Text Editor Users' Guide_ (Order No. CG40) and the _Fortran Users' Guide_ (Order No. CC70)) should be consulted when searching for descriptions of language- and subsystem-specific errors -- these errors are not addressed here. Multics compilers print a list of errors when they compile a source segment. For example, the list might indicate incorrect syntax in the source segment. This list of errors may be graded by severity; you may judge whether to continue compilation or halt it by issuing a quit signal. Generally, it is advisable to let the compilation finish so that all errors are reported, and fix all errors before trying to recompile. (An exception to this is an error such as a missing include file.) Severe errors automatically cause compilation to cease. The compiler prints an error message, and the system returns to command level and prints a ready message.

When an error arises, the system may handle it in one of three ways (the examples below show how the error message looks, but not its cause):

- For command errors and command processor errors --

  Prints an error message and then a standard ready message. The form of the ready message indicates that you are still at command level 1, as you were before you began, e.g.:

  ```
  Segment primt not found.
  r 9:37. 144 55
  ```

- For execution errors and faults --

  Prints an error message and then a ready message containing additional information that shows you are at a _new_ command level!

  ```
  Error:  Attempt to divide by zero at >udd>Project>Person>program|13
  (line 5)
  system handler for error returns to command level
  r 13:14 0.099 59 level 2
  ```

- For fatal process errors --

  Prints an error message and creates a new process for you (effectively logs you out and then immediately back in).

  Fatal error. Process has terminated.
  New process created.

  (Then goes through your start_up and prints a ready message when done.)

The above messages are in an abbreviated form. Frequently these messages contain additional information that is specific to the case at hand.

The descriptions of how to handle these various states are described in the sections to follow.

CONVENTIONS

The examples in this document follow certain conventions.

Technical or other unfamiliar terms are underlined when used the first time.

Quotation marks are used to indicate the exact spelling of a word, or the way a word or phrase should appear on a user-typed line. You do not type these quotation marks.

Another convention (shown within examples) is the use of an exclamation point to indicate user-typed lines. The exclamation point does NOT appear on your terminal -- you do not type it, and Multics does not type it to prompt you. Exclamation points appear ONLY in examples, and ONLY to show which lines you type.

Text within angle brackets (<...>) is also used within examples for explanatory purposes ONLY. They are not actually typed by Multics, and you should not type them.

A quit signal, issued when you press the QUIT, ATTN, or BREAK keys on the terminal, is indicated by "(quit)" appearing to the left of the normal text of the example. When a line of output is too long to be placed on a single line in an example, it is broken at an arbitrary point and continued on the next line.

In ready messages printed by the system on the completion of a command line, a series of four dots (....) replaces information, such as the time of day, irrelevant to the example being presented. For example, "r 13:23 0.067 67", and all other "real" ready messages appear in this document as "r ....".

Finally, many lines of input or output whose content should be obvious from (or is irrelevant to) the example are replaced by ellipses (...).

SECTION 2

COMMON ERROR MESSAGES FOR EVERYONE

When unusual circumstances prevent or interrupt completion of work, anyone and everyone receives error messages, regardless of how long and to what extent they use the Multics system!

The most important thing for you to remember is that no matter what the message says (or seems to imply), YOU HAVE NOT (AND CANNOT) HURT THE COMPUTER! (You may have even received a message that begins "FATAL ERROR...", which may give you a start but is not, in fact, fatal!) Generally, at the very worst, you may lose work that you had wanted to save. And if you lost a segment that you had PREVIOUSLY saved, there are ways to get it back.

This section describes three kinds of errors that anyone might get: errors that are detected by the command processor, errors detected by the commands themselves, and those that indicate system problems.

Before entering into explanations of these types of errors, here are some general suggestions to prevent common mistakes that generate some of these messages, and explanations for why you may receive an error message that seems to have no connection with an action you requested.

GENERAL PREVENTIVE SUGGESTIONS

For new users, one of the most common actions that results in an error message is an attempt to type something WHILE Multics is printing a response to your previous action.

If you type a command that causes output to be printed at your terminal, ALWAYS wait for the system to finish printing before you type anything; otherwise, the line you type may get garbled and an error results. If this happens, type an @ sign to ensure a fresh start and retype the command on a new line. (To make sure that the terminal output does not interfere with your input (and vice versa), you can use an instruction to control your terminal -- see the set_tty command in the MPM Communications Input/Output, Order No. CC92.)

There are cases where the error message you receive may seem totally inappropriate in relation to the command you MEANT to type. This is most often due to a typographical error or misspelling.

Another common cause of errors and confusion are abbreviations contained in your profile segment, created with the abbrev command (see the New Users' Intro). Once you have created abbrevs, the command processor always checks every command line for them -- if one of your abbrevs is hidden in the command line, it is found and expanded (sometimes when you've forgotten about it or are not expecting it). Keeping track of all abbreviations in your profile can help eliminate this stumper.

Errors may often be caused by a garbled telephone connection between your terminal and the computer. If there is no apparent cause for an error, reenter the same command line again.

Another point to keep in mind is that when you type a number of commands on a line (separated by semicolons) and one command fails, the rest of the line may be ignored (the action specified in any succeeding command(s) may not be taken). Whether or not this happens depends on why that command failed, and what kind of error it produced. In such an instance, however, you must find out which command failed, and reissue that command and any that followed it.

Finally, if you are at a point where every command you type causes an error, your process is damaged and you should use either the new_proc or logout command (see the MPM Commands) to obtain a new process and get yourself out of trouble. In the rare cases when your process is so damaged that these commands fail, you will have to either hang up the phone and dial in again, or, (if your terminal is hardwired) call the operator.

HOW COMMANDS CAN BE INTERRUPTED

Often it is desirable to interrupt a command before its execution is complete. You may discover while the command is executing that a mistake has been made, or it may simply not be necessary to execute the command entirely. For example, you may issue the print command (described in the MPM Commands) but not need to see the entire segment printed. So as soon as the needed information is printed, you could issue a quit signal, by pressing the button on your terminal that is labeled QUIT, INTERRUPT, BREAK, or ATTN. The quit signal causes Multics to stop whatever it is doing and print QUIT and a ready message.

The ready message printed after a quit signal is slightly different from other ready messages because it contains additional information after the standard numbers:

```
r 9:38 1.123 62 level 2
```

The character string "level 2" indicates that a new command level has been established and the interrupted work is being held on the previous level. Since the system is at command level, that is, ready to accept more commands, you can either continue the interrupted work or go on to something else.

Recovering from an Interruption

If the work interrupted by the quit signal is to be continued, you can issue either the start (sr) or the program_interrupt (pi) command (fully described in the MPM Commands). The start command resumes execution of the interrupted command from the point of interruption (but notice that if output was being printed when you signalled quit, an arbitrary amount of output may be lost, i.e., when it resumes printing it starts at a later point). The program_interrupt command resumes execution of the original command from a known, predetermined reentry point. (See the discussion of the program_interrupt command under "Handling Execution Errors" later in this section.) Usually the program_interrupt command is invoked when you are working in a subsystem like qedx or read_mail and you want to interrupt printing and remain in the subsystem. This method of resuming an interrupted command is useful for skipping over information not needed at the time. After the QUIT message is printed, typing the program_interrupt command will return you to request level in the subsystem.

If, on the other hand, you do not wish to continue the interrupted work, the interrupted command should be released before any other commands are issued. The release (rl) command (see the MPM Commands) releases the work interrupted and held by the quit signal and returns the system to the previous command level (dropping the level information from the ready message).

The first type of error described here is command processor errors.

COMMAND PROCESSOR ERRORS

Error messages from the command processor are the most basic level of error message. It is both necessary and fairly easy for you to maneuver your way past these, no matter what kind of user you are. Following is a brief explanation of how these errors occur, and then some real examples.

The command processor, as its name implies, processes commands--that is, it intercepts and interprets the lines you input at command level, and then calls the appropriate program to perform the desired operation.

If the command processor determines that the command line is improperly typed, it prints an error message on your terminal that attempts to show you where the mistake is. Then the system waits for you to type another command. Once the cause of the problem has been determined, you may retry the (corrected) command.

The following example shows how a misspelled command name is (mis)interpreted by the command processor. It searches for the program (segment) with the misspelled name ("primt," intended as "print") and then reports that the program cannot be found:

```
! primt report
  Segment primt not found.
  r ....
```

If your command line includes more than one command, for example, "print report; cwd another_dir; print new," commands that appear after the point at which the "segment...not found" error occurs are ignored by the command processor -- that is, they are not executed. Other errors that cause the rest of the command line to be aborted are all those that begin "command_processor_:" such as "Quotes do not balance," "Parenthesis do not balance," "Brackets do not balance," "Mismatched iteration sets," "Blank command name," and "Null bracket set encountered." (The other error that has the same effect is "Entry Point XX not found in Segment XX" -- see Section 3.)

When the line is retyped correctly ("print report") the command processor passes it to the print command. If all is well within the command line (e.g., the placement of arguments), which in this case, it is, the command is executed. (What happens if all is not well is discussed under "Command Errors" below.)

Following are some examples of common command processor errors, including scripts to show the method of determining the cause of the error and recovery techniques.

SEGMENT NNNN NOT FOUND

This means that name NNNN did not match any entryname within the user's search rules. The most common cause of the error is incorrectly specifying (through mistyping or a misconception) the command name.

For example, suppose that you had a program called colour, but mistakenly typed "color" when calling it:

```
! color
  Segment color not found.
  r ....

! list

  Segments = 3, Length = 3.

  re      1  colour
  r w     1  colour.pl1
  rew     1  Holmes.mbx

  r ....

! colour
  ...
```

Here you get the error, then use the list command (see the MPM Commands) to see if there is something wrong with the name. Finding the mistake, you retype the command line with the corrected name.

PARENTHESES DO NOT BALANCE

Since parentheses have a special meaning on a Multics command line (see the New Users' Intro), any parenthesis is interpreted as an attempt to employ that special usage, called iteration. So, this error means simply that a parenthesis beginning or ending an iteration set was unbalanced. For example:

```
! create >udd>Serpent>Holmes>(output1 output2
  command_processor_: Parentheses do not balance.
  r ....
```

What was intended was "(output1 output2)" to create two segments; the ending parenthesis was left off. This error is handled by reentering a command line containing a balancing parenthesis.

The problem may arise when iteration of a command is not intended. For instance, the "send_message" command which transmits its arguments to another party:

```
! send_message KTWise.Doc delete the files (in Holmes>old
  command_processor_: Parentheses do not balance.
  r ....
```

Here the intent was to send a message containing a parenthetical thought. If the command line were reentered with a trailing parenthesis, two messages would be sent. That is, KTWise would receive:

```
From Holmes.Serpent: delete the files in

=: delete the files Holmes>old
```

Notice that the first message contains the first string in the "(in Holmes>old)" iteration set and the second message, the second string. You can avoid this problem by enclosing the entire message in quotes:

```
! send_message KTWise.Doc "delete the files (in Holmes>old)"
  r ....
```

It is advisable to always enclose messages in quotes to avoid unintentionally sending someone repeated messages.

BRACKETS DO NOT BALANCE

An invocation of an active function (a procedure returning a replacement string to be inserted into the command line) is enclosed in square brackets. This error simply means that the command line had an unbalanced left or right bracket:

```
! list -pathname [pd
  command_processor_: Brackets do not balance.
  r ....
```

The correct command line would have contained "[pd]" to return the name of the process directory. The error can be handled by entering a corrected command line.

Like the case of unbalanced parentheses in the message example above, an active function invocation (balanced brackets) in a send_message command line transmits a message containing the value of the active function. You can avoid this problem by enclosing the message in quotes.

QUOTES DO NOT BALANCE

Quotation marks are used in a command line to delimit a single string argument that contains special characters such as brackets, parentheses, and spaces. The error means that the command line contained an unbalanced quote and can be remedied by reentering the corrected command line.

```
! trim_list patients -select "city equal Somerville
  command_processor_: Quotes do not balance.
  r ....
```

In this example, the command line was intended to use the trim_list command (see the WORDPRO Reference Guide, Order No. AZ98) to delete from the "patients" lister file the records of all patients residing in the city of Somerville. The argument containing the name of the city must be quoted since it contains spaces, and the trailing quote was inadvertently omitted.

LINKAGE SECTION NOT FOUND

This error occurs most frequently when you have a segment in your directory that has the same name as a command and is not an object segment (i.e., an executable program).

```
! qedx
  ...  <editing occurs here>
! w who  <user creates segment "who">
! q
  r...

! who
  command_processor_:  Linkage section not found. who

! list

  Segments = 2, Lengths = 2

  r w  who
  rew  Holmes.mbx

  r...

! rename who important_people.list
  r...

! terminate_single_refname who
  r...

! who
  ...
  r...
```

In the example above, the user creates a segment named "who" with the qedx text editor. When she then tries to use the who command (see MPM Commands) to see who is logged in to the system, the command processor finds the user's segment instead of the segment containing the system's who command. She recovers from this error by first renaming her segment to avoid further occurrences of the same error, and then using the "terminate_single_refname" command (see MPM Commands) to instruct the system to forget about any segments it may know of by the name "who."

See also "Execution Errors" in Section 3.

COMMAND ERRORS

These are errors detected in the processing of a command. Command errors are not restartable; that is, after the error message indicates a problem, you must retype the entire command line (some errors can be restarted simply by typing "start" - see "Handling Execution Errors" later in this section). A message is printed, followed by a ready message, and the system resumes what it was doing (e.g., listening for commands). The cause of the error can be fixed, and the command reissued.

BAD SYNTAX IN PATHNAME

This means that a pathname (the ordered list of entrynames identifying a segment in the storage system) has been formed incorrectly. The causes of this error are typing mistakes and an incomplete understanding of what a pathname is. (In the latter case, see the MPM Reference Guide.)

```
! print >udd>Serpent>>Holmes>a.basic
  print: Bad syntax in pathname. >udd>Serpent>>Holmes>a.basic
  r ....

! print >udd>Serpent>Holmes>a.basic
  ...
```

Here the user gave a pathname with two ">"s next to each other. As this is incorrect syntax, an error message was printed. The user recovered by typing the correct pathname. This error occurs if a "<" appears out of place in a relative pathname, that is, at any place other than the beginning of the pathname. For example, the symbols preceding "Student" here are acceptable, but the one preceding "Green" causes an error:

```
<<Student<Green>old.runoff
```

INCORRECT ACCESS ON ENTRY

This means that you do not have the correct access to a segment to perform a certain operation. This error can be dealt with by using the list_acl (la) command (see the MPM Commands) to determine why you have no access and who can give you access. If you have access to do it yourself, use the set_acl command to set the appropriate access to the segment.

The error may arise when trying to read a segment or file (e.g., when reading a segment with an editor like Qedx or Emacs, or when printing a file using the print or dprint command).

In the following example, the user does not have "read" access to the segment. The following dialogue might occur for user McGinnis logged in under the Serpent project.

```
! qedx
! r color.pl1
  qedx: Incorrect access on entry. >udd>Serpent>Holmes>color.pl1
! q
  r ....

! list_acl color.pl1     <lists access to the segment>

  r w   BDLucifer.Serpent.*
  r w   *.SysDaemon.*

  r ....

! list_acl >udd>Serpent>Holmes
  sma   *.Serpent.*      <lists access to containing directory>

! set_acl color.pl1 rw   <sets access for himself>
  r ....

! qedx
! r color.pl1
  ...
```

In the above example, user McGinnis has attempted to edit segment color.pl1 by reading it into a qedx buffer. The qedx command detects that he does not have read access to the segment, and reports an error. He exits from the editor, and by using the list_acl command, finds that only one other user on the Serpent project (BDLucifer.Serpent) has access to the file. As the entire Serpent project has sma permission on the Holmes directory, McGinnis uses the set_acl command to give himself access to the file, and retries the qedx request. (The list_acl and set_acl commands are fully described in the MPM Commands.)

The error may also occur when attempting to write out a segment that you are editing. In the example below, the user does not have "write" access to the segment.

```
! qedx
! r color.pl1

  ...    <Editing changes here>

! w
  qedx: Incorrect access on entry. >udd>Serpent>Holmes>color.pl1
! e set_acl color.pl1 rw     <McGinnis gives himself access>
! w
! q
  r ....
```

Here McGinnis tries to save a program that he has been editing, but cannot do so because he does not have write access to the segment. He is faced with the problem of setting the access on the segment without losing the editing that he had done. The qedx "e" request allows him to execute a Multics command line without exiting from qedx, so he uses it to invoke the set_acl command to recover from the error. After he changes the access, he reissues the qedx write request.

If McGinnis had been unable to change the access, he could at least save what he had done by writing it out into another segment (giving it a new name) as shown below:

```
! e set_acl color.pl1 rw
  set_acl: Incorrect access to directory containing entry.
           >udd>Serpent>Holmes>color.pl1
! w color1.pl1
! q
```

INCORRECT ACCESS TO DIRECTORY CONTAINING ENTRY

This error means that your process does not have enough access on the directory in which a segment is (to be) catalogued to perform some operation on it. Again, you can rectify this error by using the list_acl and set_acl commands.

This error most commonly occurs while trying to:

- delete a segment (you lack modify access on the containing directory)
- change the access on a segment (lack modify access)
- move, create, or copy a segment (lack modify and/or append access)
- find out information about a segment (lack status permission).

```
! status <BDLucifer>souls.list
  status: Incorrect access to directory containing entry.
          >udd>Serpent>BDLucifer>souls.list
  r ....


! list_acl >udd>Serpent>BDLucifer
  sma    BDLucifer.*.*
  sma    *.SysDaemon.*
  r ....

! list_acl >udd>Serpent
  sma    *.Serpent.*
  sma    *.SysDaemon.*

  r ....

! set_acl <BDLucifer s
  r ....

! status <BDLucifer>souls.list
  ...
```

Here Anyone.Serpent attempts to find out information about the segment. The status command requires at least "s" access to the containing directory in order to return any information, and not having it, prints an error message. Anyone then checks the fact, and looks at her access to the parent of the directory containing the segment to see if she can set the appropriate access herself. She then gives herself the necessary access, and reissues the command.

SOME DIRECTORY IN PATH SPECIFIED DOES NOT EXIST

This means that a directory specified in the pathname of a segment does not actually exist. Usually, the pathname is mistyped -- one or more of the directories in the pathname may be misspelled, missing, or in the wrong order. The best way to handle this is to verify the pathname -- ask the system!

The way to determine what directory is missing and/or the entryname of the directory actually intended is to use the list command:

```
! print >udd>Serpent>SHolmes>color.pl1
  print: Some directory in path specified does not exist.
         >udd>Serpent>SHolmes>color.pl1
  r ....

! list -pn >udd>Serpent -dr     <list the directories contained
                                    in Serpent>
  Directories = 2.

  s    Holmes                   <find out proper form of name>
  sma  BDLucifer

  r ....

! print >udd>Serpent>Holmes>color.pl1
  ...
```

ENTRY NOT FOUND

This means that a segment specified was not found in the directory. (All the containing directories do exist.)

This error can be handled by using the list command to see if the segment exists under some other entryname. Use the rename or addname commands (see the MPM Commands) as desired to change the segment's entryname or give it an additional entryname.

A common cause of this error in the case of novice users is misnaming the segment. For example, a Fortran source program must have the suffix ".fortran". Thus if the segment "main" had been created containing the program, an error would ensue when you try to compile a misnamed program:

```
! qedx
! a
! ...     <Type program in here>
! \f
! w main
! q
  r ....

! fortran main
  Fortran
  fortran: Entry not found. main.fortran
  r ....

! rename main main.fortran
  r ....

! fortran main
  ...
```

In the example above, it is important to note that the program was renamed; if, instead, the name "main.fortran" was added to "main", the source segment would have been destroyed when the compiler put the object code into the segment "main".

If the name identifies a link, then another possible cause of the error is that the segment pointed to by the link does not exist. This possibility can be checked by listing the link ("list -link") and checking whether the target exists. (Note that the link target may be another link, in which case the process must be repeated.)

INSUFFICIENT ACCESS TO RETURN ANY INFORMATION

This error arises in the cases described for the above four errors when you do not even have enough access to determine <u>why</u> the operation cannot be performed. The problem is that you do not have status permission on the directory containing a segment or, in the second case, to the directory containing the directory containing the segment.

This error can be handled as described above by first setting access on the containing directory. Usually, if you receive this error, you do not have access to set the required access, and have to contact the user who controls the directory in question.

ILLEGAL ENTRYNAME

This message is generated by an editor when you try to write from an editor buffer into a segment with a malformed name. A malformed name is one which contains special characters such as blank, tab, "/", etc., or which contains missing components. Generally, this is a name that would make it difficult to access the segment because of system conventions. Examples of illegal names are:

```
a*b
ho/whose/
c..d
prog.
```

This almost always occurs when you have given an accidental write request. For example:

```
! qedx
! r second.fortran
! ...     <Editing done here>
! w = a*b
  qedx: Illegal entryname. = a*b
  ...
```

If you want to have a segment with a name containing such special characters, you can write the segment with a normal name, and use the rename command to give it the entryname containing special characters (see the MPM Commands).

SYSTEM PROBLEMS

The error messages discussed here include those that are caused by a problem with the system that, depending on the nature of the problem, may:

- go away spontaneously, or
- require an action on your part to rectify the error.

These error messages are not your fault!

Handling Execution Errors

The errors shown in the examples below are called execution errors -- that is, some program that is executing during the course of your work has encountered a problem that it cannot handle and the execution of that program is interrupted (your work stops).

When the error is encountered and the system suspends and holds whatever work you were doing, it then prints an error message. That work is held at command level one (ready to continue once the problem has been fixed); this is reported to you as the system prints a new ready message with a higher command level (level 2 or 3 -- when you are at level 1 the standard ready message omits the level number). This new level is the level at which you can rectify the circumstances that caused the error; finally you go back to the suspended work and:

- restart it, if you have successfully handled the problem that caused the error.

  This is done by typing "start" or "sr".

- "throw it away", that is, release the held work, and then begin anew or do something else.

  This is done by typing "release" or "rl". (It may sometimes be necessary to type "release" twice, if the error recurs after the first time. Also, if the level number in the ready message is greater than two, use the "-all" control argument to the release command (see the MPM Commands) to release all held work.)

There are two other system commands that you can use to recover from an error under certain circumstances: program_interrupt (pi) and new_proc.

The program_interrupt (pi) command is used when an error occurs (or quit signal is issued) while working in an interactive subsystem (e.g., the qedx editor or read_mail). When the error occurs and you are involuntarily pulled out of the subsystem, type "program_interrupt" to reenter the subsystem at its request level.

This signals the program_interrupt condition that is trapped by the subsystem. If you mistakenly issue a program_interrupt command to reenter a subsystem that does not handle the condition, or when there is no subsystem active, the condition is reported as an error at command level:

```
! program_interrupt

  Error:  program_interrupt condition by program_interrupt|71
  (>system_library_standard>bound_command_env_)

  r .... level 2
```

If there is no subsystem active, you should issue a release command to eliminate the program_interrupt condition. If you are trying to reenter a subsystem that does not handle program_interrupt, issue a release command and then a start command to reenter the subsystem. (Normally, however, a subsystem may be reentered by a start command only if it was suspended by a quit signal.)

As a last resort (other than logging out) if nothing you type works, use the new_proc command, the equivalent of logging out and immediately logging back in. A new process is created for you -- this is the only way you can continue if your process has been damaged (i.e., every command you type causes an error) or if you get inexplicable errors no matter what you do!

In summary, when you receive an error of this type, you may:

- if it is within your power to correct the circumstances causing the error, do so, and then restart the work, or

- if the problem causing the error is out of your realm, or you simply wish to start some other action, release the held work, or

- if neither of the above helps, create a new process.

Below are some real examples of common execution errors, and descriptions of how you may recover from them.

The first example shown below is a very common error that occurs for all types of users -- record_quota_overflow.

RECORD_QUOTA_OVERFLOW

When confronted with an execution error, you can see by the format of it that this error is a different type than the ones described so far. The others start with the name of the command that was interrupted by the error, followed by a colon, then the message.

This type starts with "Error:" followed by an explanation -- this error is a <u>system condition</u>. When a certain condition arises the system automatically takes over and stops execution of whatever was taking place, but <u>saves</u> what you were working on, so that if you can correct the problem you can resume what you were doing from where you left off.

In this particular case, record_quota_overflow simply means that you have run out of storage space on the system. Common cases are when you are in an editor and are attempting to write your work into a segment, or when moving segments into your directory. When the error occurs, your work is suspended (but temporarily saved) and you can usually delete some segments from your directory (thereby making room for the new segment), then restart your work and permanently save it (without having lost any).

You could also choose to move some segments to another directory where there is unused quota, or, see your project administrator to increase your quota. As long as you don't release or log out, you can restart after getting more quota; there is no time limit.

The example shown below, attempting to save new information, is a particularly dangerous problem for if it cannot be corrected, the changes made to the text or source file are lost.

Assume in the example that PJApple is attempting to edit an existing segment using the qedx editor.

```
! qedx
! r text.compin

! ... <Editing changes here>

! w   <Error occurs here>

  Error:  record_quota_overflow condition by qedx|1316
  (>system_library_standard>bound_qedx_)
  referencing >user_dir_dir>Serpent>PJApple>text.compin|4
  (offset is relative to base of segment)

  r .... level 2   <new command level>

! get_quota -wd    <check quota here>
  quota = 100; used = 100
  r .... level 2

! list  <list segments to see what can be deleted to make room>

  Segments = 69, Lengths = 100.

  re      4 xxxx
  r w     0 text.compin &lt;segment is empty because error occurred here>
  r w     1 xxxx.pl1

  r .... level 2

! delete xxxx    <delete unnecessary segment>
  r .... level 2

! get_quota -wd  <recheck quota>
  quota = 100; used = 96
  r .... level 2

! start              <reenter editor>
! w text.compin
! q
  r ....
```

The first line of the error message says that you have run out of quota, and that it happened while you were using the qedx editor. The second line tells you the complete pathname of the program (qedx) in use when the error occurred, and the third line tells you the absolute pathname of the segment being referenced by qedx.

When PJApple tries to save her changes to the file "text.compin", the error occurs. To recover from the error, she takes the following steps:

1. She uses the get_quota command to give the current value of the quota and the number of records currently charged against it.

2. She uses the list command to give the lengths of the files in the directory as well as their names.

3. She deletes the segment "xxxx" to make room to write the file in the editor.

4. She uses the get_quota command, showing that four records of storage have been freed up.

5. She types the start command to reenter the editor, and

6. Types w (write) to save the segment.

Since she is finished editing, her final step is to quit out of the editor. This is a temporary solution as the next time she tries to save new information, the error will reoccur. The next step is to obtain more quota from the project administrator.

In the following example, PJApple can do nothing to gain additional storage, except delete the compin file text.compin which is unacceptable since it is the source file needed to produce the formatted text. Thus, her only option is to contact her administrator and ask for additional storage. The problem here is not as critical as in the case above, as no information would be lost by logging out.

```
compose text -of

Error:  record_quota_overflow condition by comp_write_¦4671
(>system_library_standard>bound_cg_)
referencing >udd>Serpent>PJApple>text.compout¦0

r .... level 2
```

Damaged Segments

If a device error or system crash causes part of a segment to be destroyed, the supervisor sets a special switch associated with the segment called the damaged switch. An attempt to reference the contents of a segment whose damaged switch is on causes an error with the message:

```
Entry has been damaged.  Please type "help damaged_segments.gi".
```

When a damaged segment is detected, the owner of the segment should change the ACL of the segment so that no other user can reference it, and then reset the damaged switch using the damaged_sw_off command (see the MPM Commands). The owner should then inspect the segment's contents to determine the extent of the damage. If the damage is easily correctable, you can simply fix the segment and continue. Otherwise, the segment should be retrieved from the last known good copy.

Below is an example of an attempt to edit a damaged segment:

```
qx
r  text

Error:  Segment-fault error by qedx$qx¦1250
(>system_library_standard>bound_qedx_)
referencing >udd>Project>User>text¦0
(offset is relative to base of segment)
Entry has been damaged.  Please type "help damaged_segments.gi".
r ...  level 2
```

The first four lines in the example tell what program you were executing when the error occurred and other very specific information describing at what point the error occurred; it is not necessary that you understand these lines in order to recover from the error. To recover from this error, type "release" and then follow the advice of the online help file "damaged_segments.gi."

SECTION 3

COMMON ERRORS FOR PROGRAMMERS

The errors described in this section are the sort that usually only a user writing programs will get. Included are some (more) command processor errors, execution errors (fault messages), and the fatal process error messages (where a new process is created).

Generally, for any error that you may encounter at this level, if your program has always worked before, the problem causing the error may be:

- something in the program that you recently changed (i.e., one line), or
- in another program that is called by yours.

The best method of determining whether the problem is, in fact, a change in your program is to compare a current copy with the next earliest edition (see the compare_ascii command in the MPM Commands).

If you are in a situation where you keep getting many inexplicable errors, your process may have become damaged. Errors of this type are those for which you can see no apparent reason, for example, if exactly the same thing worked for you before. In general, if you haven't seen the error before and cannot find a ready explanation, there is frequently nothing you can do to fix it. Type "new_proc" and try again.

If your process _is_ damaged, it is usually caused by a malfunctioning program of yours. Find the bug in your program (use the probe command -- see the MPM Commands) and fix it; however, it won't be possible to do so _in this process_ so use the new_proc command to create a fresh one.

There are several commands that give you control to translate status codes into messages, and regulate the length of and reprint those messages.

Two commands allow you to regulate the length of the message printed and reprint an error to a specified length and depth; these are described in the MPM Commands. To control the amount of information printed when an error occurs, use the change_error_mode (cem) command. To reprint an error that has just occurred and for which a stack history has been preserved, use the reprint_error (re) command.

One more command, the print_error_message command (see the _Multics System Programming Tools_, Order No. AZ03), prints out the standard Multics (error_table_) interpretation of a specified error code. The various entries allow you to specify the error code in either decimal or octal and have the output come out in either the short or long error_table_ form.

Entry Point XX Not Found in Segment XX

This error occurs when you call a reference name (XX) and a segment matching that name is found in your search rules, but it does not contain the entry point called XX.

For instance, in the example below, a segment matching the reference name "colour" is found, but it does not contain the entry point "colour." After receiving the error message, the programmer uses the print_link_info command (pli -- see the MPM Commands) to find out what entry points the program does contain, and then retypes the corrected line.

```
! colour
  Entry point colour not found in segment colour.
  r ....

! pli colour -entry

                colour     02/05/80 1540.4 est Thu



  3 Definitions:

  segname:  colour
  symb|0    symbol_table
  text|17   color                 Entry: text|17

  r ....

! colour$color
  ...
```

In the above example, the output printed by the print_link_info command shows the "segname" (the name by which the program was known when it was compiled); the only "entry" defined is the one called "color." So, you type the corrected line, a command name that gives both the reference name and the entry point name, separated by a dollar sign ($). You could also use the resolve_linkage_error command (see "Linkage Errors" below).

Another way to correct the error would be to rename the program (<u>both</u> the source <u>and</u> the object segments):

```
! rename colour.** color.==
  r ....

! color
  r ....
```

In this way, the segment now has the same entryname and entry point name and can therefore be called as a command by giving only its entryname. (For an explanation of the star and equals convention, see the MPM Reference Guide.)

The problem illustrated here occurs quite often when the program contains a procedure with a different name than that given to the segment containing the text of the program:

```
! qedx
! a
! color: procedure;
!
!    ...
!
! end color;
! \f
! w colour.pl1
! q
  r ....
```

There are a number of other causes for this error, for example, the entrypoint may have been deleted by the binder (see the bind command in the MPM Commands).

This error is virtually identical in cause to the "external symbol not found" case of linkage errors. See "Linkage Errors" below for additional examples. If the meaning of reference names versus entrynames and entry point names is confusing, see the MPM Reference Guide.

Improper Syntax in Command Name

This error is issued when you have specified a command name that is not in the standard form of a reference name, optionally followed by the special character "$" and an entry point name.

Examples of correctly formed command names are:

```
ref_name            ref_name$entry_point_name
```

Examples of incorrectly formatted names are:

```
name$          $name
```

More detailed information may be found in the MPM Reference Guide.

EXECUTION ERRORS (FAULTS)

This class of errors includes all hardware and software detected faults and conditions. When an error of this sort occurs, a condition is signalled. The condition can be handled by a user-supplied condition handler (a PL/I on unit), or if no on unit is found (as is normally the case), the default system on unit. The system's on unit prints an error message and invokes a new command level, suspending the execution of the program causing the error. This new command level is indicated by a ready message with a level number greater than one:

```
r 12:04 2:039 347 level 2
```

After an error has occurred, and a new command level is entered, you should eventually do one of three things:

1. Issue a release command to terminate execution of the suspended program. For example, a quit signal may be used to stop a runaway program or excessive printing:

```
!          looper
! (quit)
           QUIT

  r .... level 2

! release
  r ....
```

   The release command need not be used immediately after the error occurs. If the cause of the error is not obvious, system supplied tools (e.g., probe) can be invoked at the new command level to determine the cause. Whether or not this is possible, the release command should be issued <u>before</u> doing any additional work (e.g., changing and recompiling the program) to avoid more serious and incomprehensible errors.

2. The start command can be used to restart the program that was interrupted. This is possible if the problem is correctable, or in the case of an erroneous computation where the system's on unit performs some specified action to correct the condition upon restart. Such a correction might be to set the result of the computation, 2 ** -1000, to 0 after an underflow condition has occurred. The actions taken by the system on unit are often specified in the error message; if not, consult the MPM Reference Guide.

   Another common practice is to "quit" out of a program that appears to be looping, check the CPU time that it has used by inspecting the ready message, and if it is looping, release the suspended program (after debugging the cause of the loop); otherwise, resume the execution with "start."

NOTE: quit/starting in this way may lose output directed to the terminal. However, under certain circumstances, this may be desirable.

```
        ! count
            1
            2
! (quit)
          QUIT
          r .... level 2

        ! start
            6
            7
! (quit)
          QUIT
          r .... level 2

        ! release -all
          r ....
```

Here a program named count has been invoked. It was then stopped by a quit signal and restarted by the start command; as a result, a few lines of output were lost. The program was then stopped a second time by a quit signal and aborted by the release command.

3. Issue the new_proc command to get a new process. This reinitializes all static variables, common blocks, I/O attachments, files, etc. The use of this command is recommended when inexplicable errors occur. Once a new_proc is finished, it is advisable to retry the program with which there is a problem. Often the problem disappears. It it doesn't, it is likely that a program bug exists, and you should continue to look for some other cause. The thing to remember is that an erroneous program can cause other programs, including system programs, to go awry.

The error messages produced for most of this class of runtime errors are in a common format, for example:

```
Error: Attempt to divide by zero at >udd>Serpent>PJApple>prog|13 (line 5)
system handler for error returns to command level
```

The first line gives the type of error ("Attempt to divide by zero"), the pathname of the object segment causing the error (>udd>Serpent>PJApple>prog), the offset in the program object segment of the instruction at which the error occurred (13 octal), and, if the program was compiled with the "-table" option, the source line number. The second line gives additional information about the error. Here it states that a new command level will be created.

In general, an error that occurs in a system program can be traced to a user error. (This is not to say that there are not bugs in system programs, however it is more likely that the user did something wrong.) In the case of an error in a system program, you should verify that you have called it properly: that the correct number of arguments have been passed, that all documented requirements and restrictions have been met, and that all arguments passed as input to the system program have reasonable values.

When an error occurs in a system program, the location in the user program where the system program was called is not given in the error message. This location can be determined using the probe command (see the MPM Commands for a complete description of the probe command).

Segment Fault

This error means that the work you are doing is calling a program that has addressed a nonexistent segment. What has happened is that an address value (pointer, entry, or label) contains an invalid segment number. In the message, the line that usually shows the pathname being referenced appears as "garbage" -- groups of slashes and numbers -- representing the nonexistent segment.

There are two general causes: using an uninitialized address datum, and using an address value designating a segment after that segment has been deleted.

A deleted segment may be referenced under the follo wing circumstances:

- if the program was, at the time of the deletion, still active (its execution suspended by a quit signal or error condition):

```
! prog

  Error:  Attempt to divide by zero at >udd>Serpent>PJApple>prog|24
          (line 12).
  System handler for error returns to command level.
  r .... level 2

! ...

! delete prog.
  r .... level 2

! ...

! start

  Error:  Segment-fault error by 465|6
  referencing\000\000\000\000\000\000\000\006?\77\216
  \000\000\006\000\000\000\465\400#\000\00
  \c0\000\000\200\000\000\000   \000\264\400#\005p\000\000
  There was an attempt to use an invalid segment number.
  r .... level 2
```

- or if the segment is an input or output file that was not closed prior to deleting the segment:

```
! prog
! (quit)
  QUIT
  r .... level 2

! delete output_file
  r .... level 2

! release
  r ....

! prog

  Error:  Segment-fault error by open_uns_file$put_chars_uns_file|1036
  (>system_library_standard>bound_vfile_)
  referencing 345|0
  There was an attempt to use an invalid segment number.
  r .... level 2
```

An uninitialized address value may also be caused by forgetting to initialize the corresponding variable. (This can also cause any of the other bad address problems described under other errors. An uninitialized pointer may cause a worthless value to be displayed for a variable qualified by the pointer or for the pointer itself. Most uninitialized automatic pointers point into the stack.)

```
! prog

  Error:  Segment-fault error by >udd>Serpent>PJApple>prog|327
  (line 43) referencing 2348|27
  There was an attempt to use an invalid segment number.
  r .... level 2

! probe
  Condition segfault raised at line 43 of prog.
! source
            p -> data = 3;
  value p
     2348|27
    ...
```

<u>Fault Tag 1/Fault Tag 3</u>

This means that an addressing modification fault has occurred while attempting to indirect through a pointer. Since these modifiers never appear in PL/I pointer datums, the problem is usually uninitialized address data.

```
! prog

  Error:  fault_tag_1 by >udd>Serpent>PJApple>prog|14 (line 8)
  referencing stack_4|3320 (in process dir)
  Ascii data where pointer expected.
  r ....  level 2
```

The fact that the program was referencing some data in stack_4 at the time of the error indicates that the bad pointer was an automatic value. If the program had been referencing "!BBBJ----.area.linker" the bad pointer would be a static value. The address modifier may also be encountered when trying to execute data. In such a case, the error message indicates that the segment causing the error is a data segment such as the stack or the combined linkage section.

The error fault_tag_1 is often caused by an uninitialized pointer occupying space previously filled with ASCII data (hence the second part of the error message).

Illegal Modifier

This means that an illegal address modifier has been used. It may appear in a pointer value or in data being executed as regular instructions.

```
! prog

  Error: illegal_modifier condition by >udd>Serpent>PJApple>prog|44
  (Line 18) referencing stack_4|0 (in process dir)
  Possible illegal modifier in indirect chain or uninitialized pointer
  r .... level 2
```

The causes of this error are identical to those of a fault_tag_1/3 error. It is also not restartable. The problem must be corrected before the program can be run again.

Linkage Error

This error occurs when a program tries to reference an external symbol (for example, an external program or PL/I external data) and the specified symbol is not found. If the source of the error can be determined and the problem fixed, the program can be restarted. Note that it is possible to "fix" a linkage error in such a way as to cause another type of the same error to occur when the program is restarted. A little thought will prevent this from happening, however.

The four major subclasses of linkage errors are described below.

Segment Not Found

This means that a segment with the specified reference name was not found anywhere in your search rules. For example, assume that procedure "prog" calls another program, "zzzz," which for some reason cannot be found:

```
! prog

  Error: Linkage error by >udd>Serpent>PJApple>prog|20 (line 34)
  Referencing zzzz$zzzz
  Segment not found.
  r .... level 2
```

The basic approach for dealing with this error is to list the directories within which the program or data segment was thought to be. Then you can determine which of the following four cases apply:

- the segment referenced really did not exist.
- the segment referenced exists, but its name was given incorrectly (e.g., misspelled).
- an entry (segment or link) of the correct name exists within the search rules, but was ignored.
- the referenced segment exists in a directory not in the search rules.

The typical user who is working alone (i.e., not using programs in some "private" library) and is only using his own programs, standard system commands and subroutines usually only has to consider the first two cases. Below is a further description of all four.

- The segment may not exist.
  For example, it may never have been created. A common problem for new users is forgetting to compile the program. Continuing with the above example:

```
! list

  Segments = 3, Lengths = 3.

  re      1  prog
  r w     1  zzzz.fortran
  r w     1  prog.pl1

  r .... level 2
```

Notice that there are source and object segments for prog, but only a source segment for zzzz (zzzz.fortran). The cause of the problem then, is that there is no object segment named "zzzz" to be found. Compiling the program (as shown below) creates such a segment; restarting execution causes the search for the segment to be repeated, and this time found.

```
! fortran zzzz
  fortran
  r .... level 2,.

! list

  Segments = 4, Lengths = 4.

  re      1  zzzz
  re      1  prog
  r w     1  zzzz.fortran
  r w     1  prog.pl1

  r .... level 2

! start
  ...
```

Another way to resolve this problem is the resolve_linkage_error command (fully described in the MPM Commands). Using this command, after the new command line is typed and you receive a ready message, type "start" to resume execution of your program.

The example below is a typical situation in which the program is running and a linkage error is encountered. The resolve_linkage_error command is issued, correcting the linkage error and allowing the program to continue:

```
! myprog
  Error: Linkage error by >udd>m>vv>myprog|123
  referencing subroutine$entry
  Segment not found.
  r .... level 2

! rle mysub$mysub_entry
  r .... level 2

! start
     <myprog is running>
```

- No segment of the designated name may exist.
  This can happen if you are confused about the name of the segment. For example, if a PL/I program is called "subr" (i.e., subr is the label on the procedure statement) but the program resides in a segment of another name (e.g., subroutine), calling "subr" from another program causes this error. This problem can be fixed by renaming (with the rename command) the segment (and the source segment) containing "subr."

```
! rename subroutine.** subr.==
  r .... level 2
```

- The entry does exist but was ignored.
  The cause of this problem can be a <u>link</u> of the correct name that points to a nonexistent segment or a segment to which you have no access. A nonexistent segment can be caused by the segment having been moved or deleted or the target pathname being incorrect. This might appear in a listing of the directory as follows:

```
! list -pn >udd>Serpent>PJApple
  Segs=0;Msfs=0;Dirs=0;Links=1.
  r .... level 2,.

! list -pn >udd>Serpent>PJApple -link

  Links = 1.

  zzzz              >udd>Serpent>BDLucifer>zzzz

  r .... level 2

! initiate >udd>Serpent>PJApple>zzzz
  initiate: Entry not found. zzzz
  r .... level 2
```

The first list command (listing segments) shows that there are no segments in the directory, but that there is one link. The second list command shows the link to a segment in another directory. The initiate command is used to determine the reason why the segment pointed to by the link was ignored in the search -- here it does not exist. If the target pathname is incorrect in that a directory is named incorrectly, the command error "Some directory in path specified does not exist." would be reported. If the problem is no access, the error would be "Incorrect access on entry."

- While a segment of the correct name may be known to exist, the directory containing it is not in the search rules. The current search rules may be listed with the print_search_rules command:

```
! print_search_rules
  initiated_segments
  referencing_dir
  working_dir
  >system_library_standard
  >system_library_unbundled
  >system_library_1
  >system_library_tools
  >system_library_auth_maint
  r .... level 2

! list -pn >udd>Serpent>PJApple
  ...
```

In general, when it has been determined that a segment to be referenced is outside of the search rules, one of three things can be done. The search rules can be adjusted to include the directory containing the segment; the segment may be initiated; or a link to the segment can be created. For example, assume that the segment in question is the command expand. The search rules can be corrected with the add_search_rules command. The problem could be resolved by:

```
! add_search_rules >system_library_tools
        -after >system_library_unbundled
  r .... level 2

! print_search_rules
  initiated_segments
  referencing_dir
  working_dir
  >system_library_standard
  >system_library_unbunbled
  >system_library_tools
      ....
  r .... level 2
```

Here, the print_search_rules command has been used to show the corrected search rules. In the example above, the new search rule is added after >system_library_unbundled rather than after working_dir to avoid searching >system_library_tools every time a command or subroutine is referenced for the first time in the process. This approach is useful when the missing segment is one of a collection of programs in the same directory (like a program library) whose other members are also likely to be used.

The segment may also be initiated. This is useful when there is only one program needed, and it is only good within the current process.

```
! initiate >system_library_tools>expand
  r .... level 2,.
```

A link to the program may also be created. This need only be done once, and enables the program to be referenced without issuing additional commands at any time in the future provided that the directory containing the link remains within the search rules. The simplest way to ensure this is to place the link in the directory containing the calling program itself:

```
! link >system_library_tools>expand
  r .... level 2

! where expand
  >system_library_tools>expand
  r .... level 2
```

The "where" command gives the pathname of the segment whose reference name is given. That is, the command prints the pathname of the segment that is invoked if you call a given program. It has been used here to verify that the link was successful.

External Symbol Not Found

This means that a segment matching the reference name specified was found, but that the (perhaps implicitly) specified entry point within the segment was not:

```
! prog
  Error:  Linkage error by >udd>Serpent>PJApple>prog¦34 (line 38)
  referencing xxxx$xxxx
  External symbol not found.
  r .... level 2
```

This means that the segment xxxx was found, but the external entry point (symbol) "xxxx" was not found in the segment. In addition to trivial naming and typing mistakes, one of the more frequent causes for the error is that the program resides in a segment with a name different from the one used on the procedure statement of the program. The program is then called using the segment name:

```
! qedx
! a
! tester: procedure (a);
! dcl a float binary(27);
! a = a ** 2;
! end;
! \f
! w test.pl1
! q
  r ....

! pl1 test -table
  PL/I
  r ....

! test
  Error:  Linkage error by >udd>Serpent>PJApple>call_test¦54 (line 24)
  referencing test¦test
  External symbol not found.
  r .... level 2
```

This problem can be eliminated by changing the name on the procedure statement from tester to test and recompiling the program.

Linkage Section Not Found

This means that a segment of the specified name was found, but that the segment did not have a linkage section (i.e., it is not an object segment):

```
! prog
  Error:  Linkage error by >udd>Serpent>PJApple>prog¦43 (line 42)
  referencing xxxx¦xxxx.
  Linkage section not found.
  r .... level 2
```

This may occur if the name of a data or test segment was specified instead of the name of an actual compiled program. For example, a common problem is a source segment that is given the name of its object segment:

```
! list

  Segments = 4, Lengths = 4.

  r w     1  xxxx
             xxxx.pl1
  re      1  prog
  r w     1  prog.pl1

  r .... level 2
```

The list command shows the two names on the source file "xxxx.pl1". When "xxxx" was referenced from the program, it was this segment that was found, but it was not a valid object segment.

To recover from this particular error, the name must be deleted from the segment, and the text compiled into the object program to be called. (These two steps must be taken in that order.) The program can then be restarted:

```
! delete_name xxxx
  r .... level 2

! pl1 xxxx
  PL/I
  r .... level 2

! start
   ...
```

There is No Room to Make Requested Allocation

This means that the size of a named external data area exceeded the system limit of 255K words. Examples of such areas are named common blocks in FORTRAN and external names containing a "$" in PL/I. For example:

```
! nospace: procedure;
     declare bigarea$ (300000) external static fixed binary;
     ...
     bigarea$ (1) = ... ;
     ...
  end;
```

Executing the above program would produced the following error:

```
! nospace

  Error:  Linkage error by >udd>Serpent>PJApple>nospace|11 (line 4)
  referencing bigarea|
  (with a create-if-not-found link)
  There is no room to make requested allocation.
  r .... level 2
```

When such an error occurs in a PL/I program, examine the declaration of the external symbol and calculate the size. If it is a structure containing elements each smaller than the limit, the structure can be broken up. For example:

```
declare

     1 extstruc$ external static,
       2 a (100000) fixed bin,
       2 b (100000) float bin,
       2 c (100000) float bin(63);
```

would occupy a total of 400,000 words of storage. Member a uses one word per element; b, a single precision real value, uses one word per element; and c, a double precision real value, uses two words per element. It can be broken up into two or three small structures:

```
declare

     1 extstruc1$ external static,
       2 a (100000) fixed binary,
     1 extstruc2$ external static,
       2 b (100000) float binary,
     1 extstruc3$ external static,
       2 c (100000) fixed binary;
```

If the symbol being created is one large array, then you should attempt to reduce the size of the array needed. If such a reduction is not possible, it may be possible to simulate the array as an array of pointers to cross sections of the original array.

```
declare
     array$ (3,100000) external static fixed binary;
```

would cause the error described here. This could be replaced by:

```
declare
     array (100000) fixed binary based,
     arrp (3) pointer initial
       (addr (array1$), addr (array2$), addr (array3$)),
     (array1$, array2$, array3$) (100000) external fixed binary;
```

with the program edited to replace all references to:

```
array (x, y)        by      arrp (x) -> array (y)
```

Similar problems occur in FORTRAN when very large common blocks are used. As in PL/I, there are two cases: when there are many small members of the common block, and when there is one very big member. In the first case, the problem can again be dealt with by splitting up the common block:

```
common /data/ a(100000), b(100000), c(100000)
```

becomes:

```
common /data1/ a(100000), b(100000)
common /data2/ c (100000)
```

In the second case, that of one very large member, there is no method to get around the problem that is particularly efficient. The best that can be done is to write a function that references cross sections of the array defined in different common blocks:

```
common array (3,100000)
```

becomes:

```
function array (x, y)
     common /data1/ array1 (100000)
     common /data2/ array2 (100000)
     common /data3/ array3 (100000)

     go to (1, 2, 3) x
1    return (array1 (y))
2    return (array2 (y))
3    return (array3 (y))
end
```

In FORTRAN, address data problems may occur as well. One cause is passing an array argument to a FORTRAN subroutine whose corresponding parameter is not dimensioned. When the program references this parameter with subscripts, FORTRAN treats 'the parameter as an entry value. For example, executing a program of the following form:

```
subroutine mattran (arrin, arrout)
     dimension arrout(4,4)
     ...
     arrout (i,j) = arrin (j,i)
     ...
end
```

could cause an error of the form:

```
! mattran_test

  Error: Segment-fault error by >udd>Serpent>PJApple>mattran|143
  (line 12) referencing 327|756
  There has been an attempt to use an invalid segment number.
  r .... level 2

! probe
  Condition segfault raised at line 12 of mattran.
! source
            arrout (i,j) = arrin (j,i)
! value i; value j
    1
    1
! value arrin (i,j)
  Probe (value):  Condition "seg_fault_error" occurred at
  probe_print_value_|2017.  Possible invalid pointer.

! symbol arrin
  entry variable parameter
  ...
```

Here, the subroutine mattran has been called from mattran_test. A segment fault error occurs on line 12, and probe is invoked to look for the cause of the problem. The "source" request gives the text of the statement in which the error occurred; "value" requests enable the user to determine with what variable the program is having difficulty. Probe then indicates that "arrin" is the problem. The "symbol" request is used to display the attributes of the variable. The output shows that it is an entry variable and not an array at all.

Another cause would be passing too few arguments to a subroutine. In this case, referencing a parameter for which there is no corresponding argument may cause a segment fault or other addressing error.

No Execute Permission

This means that your process is attempting to execute a segment to which it does not have execute access. Upon getting this error, you should attempt to set access (or have the access set for you by the owner of the segment) to read and execute, and if successful, restart the program:

```
! prog

  Error:  no_execute_permission condition by command_processor_|522
  (>system_library_1>bound_command_loop_)
  referencing >udd>Serpent>PJApple>prog|3
  r .... level 2

! set_acl prog re
  r .... level 2

! start
  ...
```

This can occur if the access has been set incorrectly on the segment. For instance, if a "set_acl ** rw" command has been issued in the directory, or if you had created the object segment before compiling by using the create command.

The error can also occur when an uninitialized label or entry variable is referenced. This particular case can be distinguished from the others by the identity of the segment being referenced. If it is one which could be expected to be called (e.g., in the example above, "prog" is being called), then the problem is probably a simple access error; on the other hand, if the segment is a data or text segment, then the problem is probably an uninitialized address datum.

No Read/Write Permission

These mean that the process lacks the access required to read or write the segment mentioned in the error message:

```
! prog

Error: no_write_permission condition by prog|412 (line 101)
referencing >udd>Serpent>PJApple>data_seg|2
r .... level 2
```

The simplest cause is having failed to set or obtain the necessary access. As with a no_execute_permission error above, you can attempt to set the required access and then restart the program.

This problem may also be caused by bad address data. This case may be distinguished from a simple access error as given above.

Not in Read/Execute/Write/Call Bracket

This means that an attempt has been made to reference an inner ring segment. The cause is almost always bad address data:

```
! prog3

Error: not_in_write_bracket condition by prog|26 (line 5)
referencing dseg|0

r .... level 2
```

The identity of the segment being referenced can often give a clue to the variable whose value is bad. A reference to dseg, as occurred here, usually indicates that a packed pointer (a pointer value declared unaligned) is uninitialized. A reference to the stack is strong evidence that an automatic aligned pointer, label, or entry value has not been assigned a value. A reference to the linkage section is strong evidence that a static aligned pointer, label, or entry value has not been assigned a value.

Attempt to Reference Through a Null Pointer

This means that a null pointer has been used as a locator value qualifying a reference to a based variable. It usually indicates a logical bug in the program.

```
! prog

  Error:  Attempt by >udd>Serpent>PJApple>prog|57 (line 23)
  to reference through null pointer
  r .... level 2
```

Carefully examine your program to determine how the locator (pointer or offset) value could have a null value at the location in which the error occurred. The variable may not be referenced with an explicit qualifier:

```
data        instead of        p1 -> data
```

In this case, the default qualifier e.g., based (P) is used, and you should check its value.

```
! probe
  Condition simfault_000001 raised at line 23 of prog.
! source
  result = based_num + 4;
! symbol based_num
  fixed binary(17) aligned based (p)
  Declared in prog.
! value p
     null
  ...
```

This error may also occur for controlled as well as based data, if a controlled variable is referenced before it is allocated.

Simfault_NNNNNN

This means that you have attempted to use a pointer with a segment number of -1 and an octal offset of NNNNNN. The cause is use of uninitialized address data.

NOTE: A pointer with segment number -1 and offset 000001 is a null pointer. In such a case, the error message reads "Attempt to reference through a null pointer" as described above. The condition simfault_000001 is signalled explicitly only when the pointer value is not entirely a valid null pointer (for example, it has a nonzero bit offset).

Illegal Machine Operation

This means that there has been an attempt to execute an undefined machine instruction.

```
! prog

  Error:  Illegal machine operation by prog|4
  Current instruction is:
   000004   000000000000     ....    0
  r .... level 2
```

The two most common causes of this error are: branching to a nonexistent element of a constant label array, or using an uninitialized label or entry value. The segment in which the error occurs can be used to distinguish the two cases. In the former, the segment is one of those in use (in the example above prog); in the latter, it is a data segment (stack or linkage section) or some other unexpected segment.

Storage Condition

There are two causes of this error. First, you have attempted to allocate more based or controlled storage than is available in the system area. This is accompanied by the message that system storage is full:

```
! prog

  Error:  storage condition by >udd>Serpent>PJApple>prog|154 (line 52)
  System storage for based and controlled variables is full.
  system handler for error returns to command level
  r .... level 2
```

Inspect the declaration of the variable being allocated. The system cannot allocate more than 261,120 words of storage for any one variable. If the variable being allocated has an expression for a string length or array bound, the value of those expressions should be checked. Often they may involve undefined values. If all allocations are relatively small (e.g., hundreds or low thousands of words), the problem may be that the allocation is being repeated too many times. A check should be made for an infinite loop involving the allocation.

Second, and most common, is that the stack has overflowed. This error is accompanied by the message that the stack has been extended:

```
! prog

  Error:  storage condition by >udd>Serpent>PJApple>prog|166 (line 58)
  Attempt to reference beyond end of stack. Stack has been extended.
  system handler for error returns to command level
  r .... level 2
```

This error first occurs when more than 64K words of stack space are required, or when a reference is made past the first 64K of stack. The stack is extended to the next 48K boundary. Depending on the cause for extending the stack, it may be permissible to restart the program with the start command. Subsequent storage conditions may occur if additional storage is required/referenced, and the stack is extended in 48K increments up to a maximum length of 255K. Any attempt to use more than that causes a fatal process error (see "Fatal Process Errors" below).

One cause of this error is that the program is recursing too deeply (or infinitely). This case can be verified by using probe:

```
! probe
  Condition storage raised at line 17 of file_x.  (Level 143)
```

A level number in the hundreds is a certain sign of trouble. (In fact, a value in excess of 30 to 40 is uncommon, and can generally be regarded as a sign of problems.) The cumulative automatic storage requirements for a moderately recursive program (or set of programs) may also be too great. The required storage can be determined from a compilation listing produced with the "-map" option (under the heading "storage requirements for the program"). If the storage requirements will not exceed the maximum of 208K, it is safe to restart the program.

An excessively large stack frame size can also arise if there are automatic variables declared with expression length or array bounds, and the expressions reference uninitialized values. A common mistake is to make use of another automatic variable in such an expression whether or not that variable has an initial value specified. For example, a program containing the declaration:

```
  declare
       array (array_dim) fixed binary,
       array_dim fixed binary automatic;
```

could cause the error message appearing above. A debugging session might continue as follows:

```
! probe
  Condition storage raised at line 58 of prog.   (Level 11)
! source
       call subr (...);
```

Here probe has been used to determine where the error occurred. The source request shows that the error occurred while trying to call another subroutine. The reason that the error occurs at this point is that until the subroutine is called (creating a new frame for the subroutine) the stack is not actually extended. So you examine the program for abnormally sized variables:

```
! symbol array
  fixed binary(17,0) aligned automatic dimension(71902)
  Declared in prog.
```

The symbol request gives the evaluated dimensions for the array, showing it to be extremely large. (The error could appear in the same fashion if the large bounds were intended.)

Another cause is subscripting an automatic array with a value far out of bounds. This can be detected in PL/I programs by putting a subscriptrange prefix on the procedure statement:

```
(subscriptrange):
prog: procedure;

     .....

end;
```

In Fortran this can be accomplished by compiling the program with the "-subscriptrange" control argument:

```
! fortran zzzz -table -subscriptrange
```

A similar cause is a string range error; that is, the use of the substr builtin function with out-of-range arguments. In general, this is an initial position (the second argument) that is negative or far past the end of the string, or a length (the third or assumed argument) that is negative or far greater that the actual length of the rest of the string. This error can be trapped by recompiling the program with a "stringrange" prefix on the procedure statement.

A final cause is the invocation of a function that returns a value with star (expression) extents. If the bounds of an array developed as the return argument are bad, or if a bad substr expression or uninitialized character varying string is returned, a storage condition can be raised after the called function has returned, but before the calling program has resumed execution. This is indicated by a storage condition occurring in a system segment. If this is the case, there will be no other information as to what user program was executing at the time of the error.

It should be noted that a storage condition indicating a stack overflow ("stack has been extended") does not always indicate that an error has occurred. It is entirely possible for a recursive program or a Fortran program with many automatic variables to require more than 64K words of stack storage. If this is known to be the case, type "start" to continue the program's execution.

Out of Bounds Fault

This means that a nonexistent portion of a segment has been referenced by the program. A storage condition due to a stack overflow is really an out of bounds fault on the stack; as a result, the causes and recovery methods are similar (see above). The most common causes include an out-of-range array subscript or substring reference. The error is particularly common when the data in question is a Fortran variable, either in common (occurring in a segment in the process directory) or in a SAVE statement (occurring in the linkage section), or a PL/I internal static variable (occurring in the linkage section), or an external static variable (occurring in a segment in the process directory). If the segment is the program itself, it is likely that the program is referencing outside of the bounds of a label array or an internal static array that has an initial value specified but has never been modified.

Illegal Procedure

This occurs when the hardware is requested to perform an illegal operation. The most usual cause is uninitialized decimal data.

```
! baddec

  Error:  illegal_procedure condition by >udd>Serpent>PJApple>baddec|6
  (line 5) referencing stack_4|3320 (in process dir)

  r ... level 2

! probe
  Condition illegal_procedure raised at line 5 of baddec.
! source
            dv = dv + 1;
! symbol dv
  fixed decimal(7,0) aligned automatic
  Declared in baddec.
! value dv
  probe (value):  Illegal decimal data. dv
```

Here probe has been used to show the source of the line at which the error occurred. It contains a reference to a decimal variable. This is sufficient evidence to believe that the problem is uninitialized decimal data.

Other, less likely, causes of the same error are transferring to an element of a label array outside of the bounds of the label array, and referencing uninitialized label or entry variables. In the former case, the location of the error is often listed as the first line of the program; the line from which the condition is signalled is not available. In the latter case, the location of the error is usually in some unexpected segment.

Conversion

This means that an error has occurred in the conversion of a character string to some other data type. This condition occurs in conversion to an arithmetic value if the string is not a correctly formed number. It occurs in conversion to a bit string if the source character string contains characters other than "1" or "0":

```
! badconv

  Error:  conversion condition by >udd>Serpent>PJApple>badconv|22
  (line 6 onsource = "one", onchar = "o")
  Illegal character follows a numeric field.
  system handler for error returns to command level
  r .... level 2
```

The error message gives, in addition to the location at which the error occurred, the values of the PL/I builtin functions, onsource and onchar. Onsource represents the character string being converted; onchar is the (first) character in the string that is invalid for the conversion.

This error can arise during implicit or explicit conversions among variables (or the results of expressions) in the program, or during execution of a get statement when the input is converted to an arithmetic or bit value.

Size

This condition has three causes. It occurs when the value assigned to a fixed point datum exceeds the precision of the target -- for example, assigning the value 9999 to a fixed binary(3) datum. The error occurs in this way only if size checking was enabled for the statement in which the assignment was performed by a size prefix on the statement or the procedure statement. Second, it occurs during picture-controlled conversion, if the target field is too small to hold the value being converted. Again size checking must be enabled. Third, it occurs during a put list or put data statement, when the value stored exceeds the precision declared for the variable, or during a put edit statement, if the output field cannot hold the value being output. Size checking is always enabled for put statements.

```
! size_err

  Error:  size condition by >udd>Serpent>PJApple>size_err¦136 (line 14)
  Precision of target is insufficient for number of integral
  digits assigned to it.
  System handler for error returns to command level
  r .... level 2
```

You should be aware of a side effect of a size condition raised while executing a put statement. A common debugging technique is to include an error on unit within the program that dumps all the variables:

```
on error put data;
end;
```

If a size condition occurred invoking the on unit, the put data statement within the on unit causes another size condition to be signalled when formatting the variable for which the condition was originally signalled. The on unit is invoked a second time, and the size condition signalled yet another time, and so on, ad infinitum, eventually leading to a storage condition or fatal process error.

Error Condition

An error condition is reported when an erroneous state arises in the program, and there is no specific condition for that state. For example, this includes use of mathematical builtin functions with arguments that are out of range.

The following program illustrates a typical situation in which the error condition is raised:

```
bigexp: procedure;
     dcl sysprint file;
     put list (exp (2345));  put skip;
end;
```

Executing the program causes the condition to be signalled. The system on unit gives the reason for the specific cause of the problem, and states a fixup to be taken if the program is restarted.

```
! bigexp

  Error:  error condition by >udd>Serpent>PJApple>bigexp|53 (line 3)
   exp(x), x > 88.028, not allowed
  Type "start" to set result =  .17014118e+39
  r .... level 2

! start
   1.701e+038
  r ....
```

After receiving the error, you may decide that the standard fixup is acceptable, and restart the program as has been shown above. Notice that the program proceeds normally to output the result as set by the action of the system on unit.

Subscriptrange

This means that a subscript specified in an array reference is outside of the bounds of the array. The condition is normally raised only when you have specified that subscript range checking be performed (by placing a subscriptrange condition prefix on a PL/I procedure statement, or compiling a Fortran program with the -subscriptrange control argument). Such checking is useful when there are unexplainable storage, out of bounds, or fatal process errors.

```
! subrange

  Error:  subscriptrange condition by >udd>Serpent>PJApple>subrange|17
  (line 7).
  A subscript value has exceeded array bounds.
  system handler for  error returns to command level
  r .... level 2

! probe
  Condition subscriptrange raised at line 7 of subrange (Level 12)
! source
            array (i) = i;
  ! value i
        5
! symbol array
  fixed binary(17,0) aligned automatic dimension(4)
  Declared in subrange.
  ...
```

Above is an example of a subscriptrange condition. Upon receiving the error, enter probe to determine the cause of the problem. The source request gives the text of the line on which the error occurred (line 7). Then display the value of i and compare it with the dimensions for the array as given by the symbol request. Here the subscript, i, is only a little bit out of range. This indicates a logical bug, specifically, that the program is not constraining the value of the subscript properly. Alternatively, if the value of the subscript were grossly out of range (for example, -72301292), this would be an indication that the problem was that the subscript was uninitialized or assigned the value of some (function of an) uninitialized variable.

This condition may also arise when a function that returns a dimension (*) array is used, and the bounds of the array returned do not match the bounds of the array to which it is assigned. For example, assume that data has dimension (4) and that array_fun returns an array with dimension (5). Then:

```
data = array_fun (...);
```

causes a subscriptrange condition to be signalled.

Stringrange

This means that a substring of a character or bit string value as specified by the substr builtin function is not completely contained within the string value. Given the reference:

```
substr (s, i, j)
```

the error implies that one of two conditions is true: that i, specifying the starting position of the substring, is less than one or greater than the current length of the string, or that j, specifying the length of the substring, is less than zero or greater than the number of positions included in that portion of the string from position i to the end.

The stringrange condition is only raised if you have compiled the program with a stringrange condition prefix on the procedure statement or on the statement that uses the substr built-in function.

```
! stringrange

  Error:  stringrange condition by >udd>Serpent>PJApple>stringrange|17
  (line 7).  A substring specified by substr is not completely
  contained in the first argument.  System handler for condition
  returns to command level.
  r .... level 2

! probe
  Condition stringrange raised at line 7 of stringrange (Level 11)
! source
            substr (str, 1, i) = "a";
! value i
      -1
  ...
```

Fixedoverflow, Overflow, Underflow

These errors indicate that the result of a computation has exceeded the precision or range of the machine. Fixedoverflow applies to fixed point computations and indicates that the result is too large. It should not be restarted.

```
! folf

  Fixed point overflow by >udd>Serpent>PJApple>folf|143 (line 27)
  System handler for error returns to command level
  r .... level 2
```

Overflow applies to floating point computations, and indicates that the result is too large. Under certain conditions it may be restarted; however, generally, it should not be restarted.

```
! olf

  Error:  Exponent overflow by >udd>Serpent>PJApple>olf|160 (line 33)
  System handler for condition returns to command level
  r .... level 2
```

Underflow applies to floating point computations, and indicates that the result is too small. The program is automatically restarted with the result of the computation set to 0.

```
! unfl

  Error:  Exponent underflow by >udd>Serpent>PJApple>unfl|167 (line 39)
  r ....
```

Notice that after an underflow condition the system does not enter a new command level, but instead continues with the program. Here it has terminated normally, returning to command level 1.

The exponent_control command may be used to change the system default action for exponent overflow/underflow.

Fatal Process Errors

In general, a fatal process error occurs when the system detects a condition such that the process is not able to continue running. (In particular, the system default on unit cannot be executed to interpret the cause of the error.) The action taken by Multics in this case is to terminate the process in which the error occurred and to create a new process for you. Because it is a new process, there is no information available about the programs that were running when the error occurred, the value of program variables, etc. The only clue as to the cause of the error is the error message.

The single most common form of a fatal process error is an out of bounds error on the stack. The causes are the same as for a storage condition (see above) arising on the stack. The message that is generated by the system designates that a fatal error has occurred, and then gives an error message indicating a more specific problem.

```
Fatal error. Process has terminated. Out of bounds fault on user's
stack.
New process created.
```

In the event of this kind of fatal process error, it is advisable to recompile your program with subscriptrange and stringrange checking enabled and try the program again. If a stringrange or subscriptrange condition then occurs instead of the fatal process error, it is likely that the new error is the source of the problems.

If the fatal process error recurs despite having the checks enabled, then the cause of the problem can be just about anything. Check your access to all programs and files that you are using to insure proper access. Also check for the possible causes of a segment fault error. Finally, calls to system programs should be checked to see if they conform to all documented conventions. Should these checks fail to turn up a clue, use the probe command to set breakpoints at various strategic points in your program to isolate the point at which the fatal process error is occurring. Often the process has to be repeated with additional breaks set until the location is narrowed down to a single statement.

Any of these fatal process errors can be caused by a program overwriting critical information in the stack or linkage section; most frequently by referencing through an invalid pointer.

You may see several other kinds of fatal process errors. They include:

No unclaimed signal handler specified for this process.
: This means that no default on unit could be found. The possible causes include subscript and stringrange errors, and the use of uninitialized address data (see above).

Fault in signaller by user's process.
This indicates the presence of a very complex error condition and probably involves more than one cause. Apply the methods of debugging described for the other errors.

Unable to perform critical I/O.
This means that your process was unable to perform an input or output operation at a crucial point, for example, writing out an error message. This indicates that the I/O attachments for the user_input, user_output, error_output, and/or user_i/o I/O switches are causing a problem. Consider the kinds of operations that you performed prior to the fatal error, and determine if they conform to the documented operating procedures.

Process terminated because of system defined error condition.
This is a catch-all message. Again, try the methods described above.

You should recall the comments about errors that vanish after a new process is created; they apply to a fatal process error as well.

SECTION 4

ALPHABETICAL REFERENCE TO ERRORS

This section contains explanations (including cause, result, and how to recover) of the Multics system error messages. An alphabetical listing of the messages sorted by the long form of the printed message includes the description of the error and the categories showing frequency of occurrence and originating area of the system.

To find an explanation of a particular message, locate the long message in the following alphabetical list--the description accompanies it.

Errors are broken down into three categories designating frequency of occurrence:

- COMMON
  Common errors that are encountered by all types of users.
- PROGRAMMER
  Less common, but frequently encountered programming errors.
- RARE
  Infrequent, specialized errors.

When possible, the errors are further classified into categories that indicate from what area of the Multics system they originate:

●Absentee	●General	●Network	●Storage System
●Access	●Hardware	●PL/I	●Subroutines
●Archive	●I/O	●Process Environment	●Supervisor
●Binder	●ipc_	●RCP	●trace
●Command Processor	●Linker	●Search Facility	

A RFNM is pending on this IMP link.

Class: rare Type: network

This only occurs at sites running the ARPANet, and probably indicates a logic error in the network software or a hardware failure in the network. The failing operation will probably succeed if retried.

absentee: Attempt to reenter user environment via a call to cu_$cl. Job terminated.

Class: common Type: absentee

This message is only seen in messages from the Initializer, where it indicates that an absentee job has terminated abnormally, most likely by taking a fault and trying to establish a new command level, which cannot be done in an absentee process.

absentee: CPU time limit exceeded. Job terminated.

Class: common Type: absentee

This message also appears only in messages from the Initializer, and indicates that an absentee job has been terminated because it has run over its CPU time limit.

A fatal error has occurred.

Class: rare, programming

This indicates that some operation could not be performed. It is generally used only for communication between subroutines, when it is necessary to distinguish only between complete success and complete failure. If it ever appears in a printed message, it should always be accompanied by explanatory text.

A first reference trap was found on the link target segment.

Class: rare Type: linker

This is issued by the binder or linker (as the code for a linkage error), and generally indicates a damaged linkage section or object segment.

A logical error has occurred in initial connection.

Class: rare Type: ARPANet

This indicates that an internal error has occurred in the ARPANet software.

A new search list was created.

Class: common Type: commands

This indicates that a search list has been created (using the set_search_paths or add_search_paths commands). This is only a warning, since the creation may be intentional; it may also indicate a misspelled search list name.

A pointer that must be eight word aligned was not so aligned.

Class: rare, programming Type: subroutines

This indicates that a pointer was not aligned as expected; pointers used in spri and lpri instructions must be eight-word aligned.

A previously referenced item has been changed by another opening.

Class: rare Type: I/O system

This indicates that the described situation has occurred while manipulating a file.

ACL is empty.

Class: common Type: access control

This indicates that an attempt was made to examine an empty Access Control List (ACL). It is not strictly an error, merely a more informative way of indicating an empty ACL than simply printing nothing.

Active process table is full. Could not create process.

Class: rare Type: Supervisor

This indicates that a new process could not be created; it only happens at login or new_proc time. If this error occurs, site maintenance personnel should be informed. Rare.

An improper attempt was made to terminate the process.

Class: rare, programming Type: fatal process error

This indicates that, due to a programming error, an improper attempt was made to terminate the user process; the process is terminated anyway.

An interprocess signal has occurred.

Class: rare, programming Type: ipc_

This indicates that an interprocess signal (IPS) has been sent. You get it when calling the hardcore ipc_ entries.

Archive component pathname not permitted.

Class: common Type: any command

This indicates that a command that cannot deal with archive components was given an archive component pathname (using the "::" syntax) as an argument. Most commands cannot deal with archive components.

Argument ignored.

Class: common Type: any command

This indicates that an extra argument has been ignored, and that the requested operation continues.

Argument is not an ITS pointer.

Class: common Type: command

This indicates that a command or subroutine that was expecting a pointer argument did not receive one as expected.

Argument too long.

Class: common

A command or subroutine argument is too large. Should be self-explanatory in the context where it is issued.

Attach and open are incompatible.

Class: common Type: I/O system

This indicates that the attach and open modes for a file or I/O device are incompatible; for instance, an attempt made to open a sequential file in stream input mode.

Attachment loop.

Class: rare Type: I/O system

This indicates that an attempt was made to create a circular set of I/O switches.

Attempt to access beyond end of segment.

Class: common

Should be self-explanatory in the context where it is issued.

Attempt to attach to an invalid device.

Class: common Type: I/O system

This indicates an attempt to use an invalidly specified I/O device. Check the attachment or resource description and try again.

Attempt to create a stack which exists or which is known to process.

Class: rare, programming Type: fatal process error

This indicates an attempt to invalidly create a stack. It's really too complicated to explain exactly what this means, but if you aren't expecting it, then your program has done something wrong and mangled your address space.

Attempt to indirect through word pair containing a fault tag 2 in the odd word.

Class: rare, programming Type: environment, linker

This indicates that an invalid indirection was attempted, due either to an uninitialized pointer or a damaged linkage section.

Attempt to manipulate last or bound pointers for device that was not attached as writable.

Class: rare Type: I/O system

This indicates an I/O error of the sort described. It is only produced by I/O modules in the obsolete (ios_) I/O system.

Attempt to modify a valid dump.

Class: rare

This only occurs when using the copy_fdump command; it indicates that the DUMP partition cannot be copied. Actually, it should never happen, and is, in any case, only going to happen to systems programmers.

Attempt to read or move read pointer on device which was not attached as readable.

Class: common Type: I/O system

This should be self-explanatory in the context where it appears. It may be the result of attempting an I/O operation before the device or connection is ready.

Attempt to re-copy an invalid dump.

Class: rare

This only occurs when using the copy_fdump command; it indicates that the FDUMP presently in the DUMP partition has already been copied out, and cannot be copied again. The copy_fdump command can only be used once with each FDUMP.

Attempt to set delimiters for device while element 1 size is too large to support search.

Class: rare Type: I/O system

This indicates an I/O error of the sort described. It is only produced by I/O modules in the obsolete (ios_) I/O system.

Attempt to set max length of a segment less than its current length.

Class: rare Type: storage system

This indicates an attempt to set the max length of a segment to a lower value than the current length. The segment must be truncated to the shorter length before this can be done.

Attempt to unlock a lock that was not locked.

Class: common, programming Type: subroutines

This indicates that an attempt was made to unlock a lock that was not locked. This almost always indicates a logic error in the calling program, such as a cleanup handler that unlocks a lock without regard to whether it was previously locked. The lock remains unlocked.

Attempt to unlock a lock which was locked by another process.

Class: common, programming Type: subroutines

This indicates that the lock being unlocked is locked by another process. The lock remains locked to the other process. Like lock_not_locked, it usually indicates a logic error in the calling program. The lock remains locked.

Attempt to write improperly formated bisync block.

Class: rare Type: I/O system

This indicates that an error has occurred while doing I/O on a bisync communications line.

Attempt to write or move write pointer on device which was not attached as writeable.

Class: common Type: I/O system

This should be self-explanatory in the context where it appears. It may be the result of attempting an I/O operation before the device or connection is ready.

Bad class code in definition.

Class: rare Type: linker

This is issued by the binder or linker (as the code for a linkage error) and generally indicates a damaged object segment.

Bad definitions pointer in linkage.

Class: rare Type: linker

This indicates that the linkage section has been mangled.

Bad mode specification for ACL.

Class: common Type: access control

This indicates that an invalid ACL mode was supplied, such as using a directory mode for a segment or MSF, or an invalid character in an access mode.

Bad part dump card in config deck.

Class: rare

This only occurs when using the copy_fdump command; it indicates that the DUMP partition cannot be copied.

Bad socket gender involved in this request.

Class: rare Type: ARPANet

This indicates that an internal error has occurred in the ARPANet software.

Bad status received from IMP.

Class: rare Type: network

This only occurs at sites running the ARPANet, and probably indicates a logic error in the network software or a hardware failure in the network. The failing operation will probably succeed if retried.

Bad syntax in pathname.

Class: common Type: commands

This indicates that a pathname argument had invalid syntax, such as misplaced "<" characters, or doubled ">>." It can also be printed by the command processor itself, to indicate an error with a command being invoked by explicit pathname.

Bisync line did not respond to line bid sequence.

Class: rare   Type: I/O system

This indicates that an error has occurred while doing I/O on a bisync communications line.

Communications with this foreign host not enabled.

Class: common   Type: ARPANet

This indicates that the requested network connection could not be opened because the Multics host table does not permit connection to the foreign host. This usually indicates that the host table has not been updated yet to reflect the existence of a new host, rather than an administrative restriction.

Component not found in archive.

Class: common   Type: commands

This indicates that the specified archive component could not be found.

Condition requiring manual intervention with handler.

Class: common   Type: I/O system

This indicates that some condition that requires intervention by the operator or some other person has occurred on an I/O device, such as a tape drive dropping out of ready state or undergoing some other sort of hardware failure. Retrying the operation may succeed.

Connection not completed within specified time interval.

Class: common   Type: ARPANet

This indicates that the requested network connection could not be opened because the foreign host did not respond in time (15 seconds). This may indicate either a problem with the foreign host, or that the Multics NCP must be reinitialized by the system administrator.

Current process_id does not match stored value.

Class: rare, programming

This indicates that a process_id in some data base does not match the current one; it is generally indicative of a program malfunction. It is used mostly by the ARPANet software (only at sites that run it), where it definitely indicates a malfunction in the network software.

Cyclic synonyms.

Class: rare   Type: I/O system

This indicates an attempt to create a circular I/O attachment loop using the syn_ I/O module.

Data has been gained.

Class: rare, programming

This should always be accompanied by an explanatory message describing the situation in more detail.

Data has been lost.

Class: rare, programming

This should always be accompanied by an explanatory message describing the situation in more detail.

Data not in expected format.

Class: rare

This indicates that a program received data in a format it could not handle. This message should always be accompanied by some explanatory text to describe the problem in detail.

Data sequence error.

Class: rare, programming

This should always be accompanied by an explanatory message describing the situation in more detail.

Defective file section deleted from file set.

Class: common Type: I/O system

This indicates that, due perhaps to an error on the I/O medium, part of the file was deleted. This happens, for instance, when a bad spot is found while reading a tape.

Device attention condition during eof record write.

Class: rare Type: I/O system

This indicates just what it says; operator intervention is probably required. Retrying the operation may succeed.

Device is not currently usable.

Class: common Type: I/O system

This indicates that a particular device is no longer usable, most likely due to a hardware failure.

Device type is inappropriate for this request.

Class: rare Type: RCP

This indicates that the device type specified in a resource request is not compatible with other parameters in the resource specification.

Device type unknown to the system.

Class: common Type: I/O system

This indicates that an attempt was made to assign an I/O device of a type that is not defined at this site, or that is logically inconsistent (such as attempting to assign a disk pack to the wrong sort of drive).

Directory irreparably damaged.

Class: rare Type: storage system

This indicates that a directory has been irreparably damaged, and cannot be salvaged. This should never occur; if it does, contact system maintenance personnel.

Directory or link found in multisegment file.

Class: rare Type: I/O system

This indicates that a multisegment file (MSF) appears to be inconsistent. Most likely, the MSF is not really supposed to be an MSF, but rather, a directory, and its bitcount has accidentally become set.

Directory pathname too long.

Class: common Type: commands

This indicates that the directory portion of a pathname argument was more than 168 characters long (after processing of "<" characters), and could not be used.

Duplicate entry name in bound segment.

Class: rare Type: linker

This is issued by the binder or linker (as the code for a linkage error), and generally indicates a damaged object segment. Specifically, it indicates that the definitions section contains more than one definition with the same name.

Encountered end-of-volume on write.

Class: common Type: I/O system

This indicates that an attempt has been made to write past the end of the I/O medium (e.g., an attempt to write past the EOF mark on a tape).

End of information reached.

Class: common

This indicates that no more information is available to the calling program. This message should always be accompanied by explanatory text to describe the situation better, unless the meaning is very obvious from context.

End-of-file record encountered.

Class: common Type: I/O system

This indicates that a read operation has encountered an end-of-file record, and that no more data is available.

Entry is for a begin block.

Class: rare — Type: subroutines

This is returned from the symbol table utility (stu_) to indicate that a particular block is a begin block rather than a procedure.

Entry name too long.

Class: common — Type: commands

This indicates that the entryname portion of a pathname argument was more than 32 characters long and could not be used.

Equals convention makes entry name too long.

Class: common — Type: commands

This indicates that an equal name argument expanded into too long an entryname (more than 32 characters), by adding components to the name.

Error in conversion.

Class: common — Type: any command

This indicates that a conversion could not be performed; for instance, a command argument could not be converted to an integer because it was not all digits.

Executive access to logical volume required to perform operation.

Class: rare — Type: storage system

An operation which requires "e" access to the logical volume access control segment (>lv>LV_NAME.mdcs), such as creating a quota account, could not be performed.

Expanded command line is too large.

Class: rare — Type: command processor

This message is obsolete, and should never appear.

External symbol not found.

Class: common — Type: linker

This indicates that an external reference (of the form seg$entrypoint) specified an entrypoint that does not exist in the segment found as segname. The print_link_info command can be used to list the entrypoints in a segment.

External variable or common block is not the same size as other uses of the same name.

Class: rare — Type: linker

This is issued by the binder or linker (as the code for a linkage error), and generally indicates that two different subroutines in a program declare the block or variable differently.

File already busy for other I/O activity.

Class: common Type: I/O system

This indicates that the specified file is in use either by another process or by another program in the current process, and hence may not be accessed. This may be the result of a program malfunction, in which case issuing the new_proc command resolves the problem.

File expiration date exceeds that of previous file.

Class: common Type: I/O system

Explanation unavailable.

File is already opened.

Class: common Type: I/O system

This indicates that the specified file is already opened, and hence may not be opened again.

File is empty.

Class: rare Type: I/O system

This indicates that a file, which was not expected to be empty, was.

File is not a structured file or is inconsistent.

Class: rare Type: I/O system

This indicates that a file is being incorrectly used or has become damaged.

File set contains invalid labels.

Class: common Type: I/O system

This indicates that some volume or volumes of a tape file set contains an invalid or otherwise unacceptable label.

File set structure is invalid.

Class: common Type: I/O system

This indicates that the structure of a tape file set is not valid.

Foreign IMP is down.

Class: common Type: ARPANet

This indicates that the requested network connection could not be opened because the foreign host is currently not operating.

Foreign host is down.

Class: common Type: ARPANet

This indicates that the requested network connection could not be opened because the foreign host is currently not operating.

Format error encountered in archive segment.

Class: rare Type: any command

This indicates that a command was unable to complete processing of an archive because it either has been damaged in some fashion (and must be retrieved or repaired) or because it is not an archive at all. Rare.

Format of IMP message was incorrect.

Class: rare Type: network

This only occurs at sites running the ARPANet, and probably indicates a logic error in the network software or a hardware failure in the network. The failing operation will probably succeed if retried.

I/O in progress on device.

Class: common Type: I/O system

This indicates that an attempt was made to do I/O on a device that appears to already be in use for some previously requested operation. This condition is not necessarily detected by the I/O system itself, and may not be a correct diagnosis of the problem, but rather an indication of confusion on the part of the I/O module.

IO device not currently assigned.

Class: rare Type: I/O system

This indicates an attempt to use an I/O device that is not now assigned to the process.

Illegal command or subroutine argument.

Class: rare, programming Type: subroutines

This indicates that an invalid argument has been supplied to a subroutine, and that no more specific error code is available to describe the problem.

Illegal entry name.

Class: common Type: commands

This indicates that the entryname portion of a pathname does not conform to the standards for constructing entrynames and starnames.

Illegal format of quota account name.

Class: rare Type: storage system

This indicates that an attempt was made to create or use a master directory quota account on a logical volume that had an improperly formatted name (not one of Person.Project, Person.*, or *.Project).

Illegal initialization info passed with create-if-not-found link.

Class: rare Type: linker

This is issued by the binder or linker (as the code for a linkage error), and generally indicates a damaged object segment.

Illegal procedure fault in FIM by user's process.

Class: common, programming Type: fatal process error

This message usually only appears to accompany a fatal process error message; it indicates that the system has encountered an error condition too serious to attempt to recover the process. Usually caused by a malfunctioning program either overwriting or deleting the stack.

Illegal self reference type.

Class: rare Type: linker

This is issued by the binder or linker (as the code for a linkage error), and generally indicates a damaged object segment.

Illegal structure provided for trap at first reference.

Class: rare Type: linker

This is issued by the binder or linker (as the code for a linkage error), and generally indicates a damaged object segment.

Illegal syntax in equal name.

Class: common Type: any command

This indicates that an invalid equal name was supplied as an argument, perhaps because it was used in the wrong argument position.

Illegal type code in type pair block.

Class: rare Type: linker

This is issued by the binder or linker (as the code for a linkage error), and generally indicates a damaged object segment.

Illegal use of equals convention.

Class: common Type: commands

This indicates that the command is not equipped to deal with equal names.

Improper access class/authorization to perform operation.

Class: common Type: access control

This indicates that an operation could not be performed because the process did not have sufficient access to perform it. It can only occur at sites that use the Access Isolation Mechanism (AIM) for access control. Common, but at AIM sites only.

Improper access on handler for this signal.

Class: rare, programming Type: subroutines

This indicates that an signal handler could not be accessed. Its original meaning is no longer relevant, and it should never be used in new programs.

Improper access on user's linkage segment.

Class: rare Type: environment

This indicates that the linkage section could not be accessed. Its original meaning is no longer relevant, and it should never be used in new programs.

Improper access on user's stack.

Class: rare, programming Type: subroutines

This indicates that a stack segment could not be accessed. Its original meaning is no longer relevant, and it should never be used in new programs.

Improper access to given argument.

Class: rare, programming Type: subroutines

This indicates that an argument could not be accessed. Its original meaning is no longer relevant, and it should never be used in new programs.

Improper mode specification for this device.

Class: common Type: I/O system

This indicates that an attempt was made to set an I/O device to a mode that does not exist, such as an invalid tty mode in the use of the set_tty command.

Improper syntax in command name.

Class: rare Type: command process

This indicates a malformed command name. See description in Section 2.

Incompatible character encoding mode.

Class: common Type: I/O system

This indicates that an attempt was made to read or write a tape with a character encoding mode that is incompatible with the label standard or with the hardware itself.

Inconsistent combination of control arguments.

Class: common Type: commands

This indicates that the control arguments were used to specify a contradictory or invalid combination of operations. The precise meaning of this error is entirely dependent on the command itself; refer to the documentation of that command for details.

Inconsistent multiplexer bootload data supplied.

Class: rare, programming   Type: I/O system

This indicates that a tty channel multiplexor could not be loaded because its data is inconsistent; it will most likely occur in the Initializer process, in response to the load_mpx operator command.

Incorrect I/O channel specification.

Class: rare   Type: I/O system

This indicates that an invalid I/O channel name was supplied to a command or subroutine.

Incorrect access on entry.

Class: common   Type: commands

This message is usually followed by a pathname and indicates that the process has insufficient access to the object in order to perform the requested operation.

Incorrect access to directory containing entry.

Class: common   Type: storage system

This indicates that an operation, such as deletion or renaming, cannot be performed by the user because he has insufficient access to the directory containing the object. Use the list_acl command to find out who has access, and, if there is sufficient access, the set_acl command to change access on the directory.

Incorrect detachable medium label.

Class: rare   Type: I/O system

This indicates that the label on a disk or tape either could not be read at all, or does not agree with the expected value.

Incorrect recording media density.

Class: rare   Type: I/O system

This, generally returned from an I/O module, indicates that an attempt was made to read a tape at the wrong density. Try another density.

Indicated device assigned to another process.

Class: rare   Type: RCP

This indicates that an I/O device could not be assigned to the current process.

Initial connection socket is in an improper state.

Class: rare   Type: ARPANet

This indicates that an internal error has occurred in the ARPANet software.

Input ring number invalid.

Class: rare

Should be self-explanatory.

Insufficient access to return any information.

Class: common Type: commands

This indicates that the process has insufficient access to determine anything at all about the specified entry (even whether it exists or not).

Insufficient access to use specified block size.

Class: common Type: I/O system

This indicates that the user is attempting to use a large block size for tape or disk I/O and does not have access to the Access Control Segment (ACS) that allows this. The user should either reduce the block size, or contact the system administrator to request access. The ACS is >sc1>rcp>workspace.acs.

Insufficient information to open file.

Class: common Type: I/O system

This indicates that the I/O module requires more information than it has already been given to open the file; some parameter is missing.

Insufficient quota on logical volume.

Class: rare Type: storage system

This indicates that there is not enough quota remaining on the logical volume to create the specified directory or change its quota. The set_volume_quota command can be used to increase the available quota.

Internal inconsistency in control segment.

Class: rare Type: I/O system

This indicates that an inconsistency has been detected in the control segment for an I/O switch, probably due to a system program logic error. The operation may succeed if the switch is closed and reopened, and will almost certainly succeed after a new_proc.

Internal index out of bounds.

Class: rare Type: network

This indicates that an internal error has occurred in the ARPANet software. It only occurs at sites running the ARPANet. It is generally indicative of an inconsistency in the system, and must be corrected by system maintenance personnel. The operation that failed may work if tried again.

Invalid backspace_read order call.

Class: common Type: I/O system

Explanation unavailable.

Invalid delay value specified.

Class: common Type: I/O system

This indicates an attempt to set invalid values for tty carriage motion delays; some value or values is too large or negative.

Invalid element size.

Class: common Type: I/O system

Explanation unavailable.

Invalid logical record length.

Class: common Type: I/O system

This indicates that the record length is not compatible with the blocksize. For instance, many I/O record formats require that the record length be no larger than the blocksize.

Invalid mode specified for ACL.

Class: common Type: access control

This indicates an attempt to set an invalid ACL mode, such as one containing invalid characters.

Invalid move of quota would change terminal quota to non terminal.

Class: common Type: access control

This indicates an attempt to move quota from a directory that has other directories with quotas underneath it. It is required that all directories superior to a directory with a quota limit (nonzero quota) also have nonzero quotas, except for master directories. See the Reference Guide for details.

Invalid multiplexer type specified.

Class: rare

This indicates an attempt to use an invalid tty multiplexor type. It should only occur in response to operator commands, or for systems programmers.

Invalid physical block length.

Class: common Type: I/O system

This indicates that an attempt was made to do I/O with an invalid block length; for example, Multics cannot write tape blocks with lengths that are not multiples of 4.

Invalid project for gate access control list.

Class: common Type: access control

This indicates an attempt to set the Access Control List (ACL) on a gate segment such that users on more than one project (not including SysDaemon) have access to it. Only system administrators can create gate segments that are accessible to multiple user projects. This can occur when setting either the ring brackets or ACL on a segment.

Invalid variable-length record descriptor.

Class: common Type: I/O system

Explanation unavailable.

Invalid volume identifier.

Class: rare Type: RCP

Explanation unavailable.

Invalid volume name.

Class: rare Type: RCP

Explanation unavailable.

Ioname already attached and active.

Class: rare, programming Type: I/O system

This indicates that the specified I/O stream is already attached, and hence may not be attached again.

Ioname not active.

Class: rare, programming Type: I/O system

This indicates an attempt to do I/O with a stream name that specifies a stream that is not presently in use. This error only occurs in calls to the obsolete (ios_) I/O system.

Ioname not found.

Class: rare, programming Type: I/O system

This indicates an attempt to do I/O with a stream name that specifies a nonexistent stream. This error only occurs in calls to the obsolete (ios_) I/O system.

Key out of order.

Class: rare Type: I/O system

This is a vfile_ error that indicates that the file was opened as keyed sequential output (which requires that all keys be entered in ascending order), and a key was detected to be not in ascending sequence. Open the file with keyed sequential update or reorder the keys appropriately.

Line type number exceeds maximum permitted value.

Class: rare Type: I/O system

This indicates an attempt to use a tty line number that is too large.

Looping searching definitions.

Class: rare　　　　　　　　Type: linker

This is issued by the binder or linker (as the code for a linkage error), and generally indicates a damaged object segment. Specifically, it indicates that the definitions section contains more than one definition that would satisfy the search--for instance, a reference to a bound_seg$name might match both segname1$name and segname2$name, where segname1 and segname2 are two separate segnames defined in the bound segment.

Master directory missing from MDCS.

Class: rare　　　　　　　　Type: storage system

This indicates that an inconsistency has been found in the master directory control segment (MDCS).

Master directory quota must be greater than 0.

Class: rare　　　　　　　　Type: storage system

This indicates that an attempt was made to set the quota of a master directory to zero or less.

Mismatched iteration sets.

Class: common　　　　　　　Type: commands

This indicates that a command line containing parentheses, to specify iteration, contained differing numbers of tokens in different sets of parentheses; for example: rename (file1 file2 file3) (name1 name2). The command line is not executed.

Mount request could not be honored.

Class: common　　　　　　　Type: RCP

This indicates that a specified tape or disk could not be mounted, for one of several reasons: all the tape or disk drives might be in use, or offline during unattended service; the volume might not exist, or the operator might be unable to locate it; it might not be permitted for the current process to access the volume.

Mount request pending.

Class: rare

Explanation unavailable.

Multics IMP is down.

Class: common　　　　　　　Type: network

This only occurs at sites running the ARPANet. It indicates that a network connection cannot be established because the Multics IMP hardware is not operating.

Multisegment file is inconsistent.

Class: common | Type: I/O system, commands

This indicates that a multisegment file (MSF) is inconsistent, and cannot be used. A possible reason is missing or damaged components.

Name duplication.

Class: common | Type: commands

This indicates that an attempt was made to re-use a name that already exists in the directory. The default response for this is to either remove the name from the entry it already refers to, if the entry has multiple names, or to ask the user whether to delete the entry, if it has but one name.

Network Control Program encountered a software error.

Class: common | Type: ARPANet

This indicates that the ARPANet software encountered an internal error. This condition may clear itself automatically, or it may require that the system administrator re-initialize the network software. The operation that got the error should be retried at least once.

Network Control Program not in operation.

Class: common | Type: ARPANet

This indicates that the requested network connection could not be opened because the Multics Network Control Program is not presently operating. This may indicate a need for it to be reinitialized; contact the system administrator.

Network connection closed by foreign host.

Class: common | Type: ARPANet

This indicates that the network connection has been closed by the foreign host; most frequently, this indicates that the foreign host has crashed.

New offset for pointer computed by seek entry is negative.

Class: rare | Type: ARPANet

This indicates that an internal error has occurred in the ARPANet software.

No I/O switch.

Class: common | Type: I/O system

An attempt was made to perform an operation on a named I/O switch that does not exist.

No device currently available for attachment.

Class: common | Type: I/O system

This indicates that the requested device assignment could not be made because all such devices are already in use.

No execute permission on entry.

Class: common Type: commands

This indicates that the specified operation could not be performed because it requires that the process have execute access to the segment.

No initial string defined for terminal type.

Class: common Type: I/O system

The initial string cannot be sent to the terminal because none is defined.

No quota account for the logical volume.

Class: rare Type: storage system

This indicates that there is no quota account for this process on the specified logical volume.

No unclaimed signal handler specified for this process.

Class: rare Type: fatal process error

This probably indicates that the stack header for the process has been mangled by a malfuctioning program.

Number of blocks read does not agree with recorded block count.

Class: rare Type: I/O system

Explantion unavailable.

One or more of the paths given are in error.

Class: rare Type: storage system

This indicates that an inconsistency has been found in the master directory control segment (MDCS).

Operation not performed because of outstanding line_status information.

Class: rare, programming Type: I/O system

This indicates that a control order cannot be performed on a communications line because there is already a line_status control order pending that has not returned results yet.

Path violates volume or account pathname restriction.

Class: rare Type: storage system

This indicates that an inconsistency has been found in the master directory control segment (MDCS).

Pathname already listed.

Class: rare Type: storage system

This indicates that an inconsistency has been found in the master directory control segment (MDCS).

Pathname appears more than once in the list.

Class: rare Type: storage system

This indicates that an inconsistency has been found in the master directory control segment (MDCS).

Pathname not found.

Class: rare Type: storage system

This indicates that an inconsistency has been found in the master directory control segment (MDCS).

Physical end of device encountered.

Class: common Type: I/O system

This indicates that an attempt was made to read past the end of the device medium, such as an attempt to space past the EOF marker on a tape.

Procedure called improperly.

Class: common, programming Type: subroutines

This indicates that a subroutine or command has been called improperly, and that there is no more specific code to describe the error. This code should alway be accompanied by some explanatory text message to describe the problem in more detail.

Process lacks permission to alter device status.

Class: common Type: I/O system

This indicates an attempt to perform an I/O operation on a device that the process did not have sufficient access to; for instance, performing privileged control orders on a tty channel.

Process lacks permission to initiate Network connections.

Class: common Type: ARPANet

This indicates that the requested network connection could not be opened because requesting process does not have proper access to use the network. Contact the system administrator for information about this.

Process lacks sufficient access to perform this operation.

Class: common Type: access control, storage system

An operation was attempted for which the process did not have sufficient access. The operation for which access is required should be evident from the source of the message.

Process lacks sufficient access to perform this operation.

Class: rare Type: storage system

This indicates that the process does not have appropriate access to the control segments for the logical volume to perform the operation.

Quota account has master directories charged against it.

Class: rare Type: storage system

A logical volume quota account may not be deleted if there are existing master directories in the hierarchy that charge against it. This can occur spuriously if the master directory control segment (MDCS) for the volume has been damaged; this damage can be repaired with the privileged check_mdcs command.

Record is too long.

Class: common, programming Type: storage system

This indicates that a call was made to read a record that could not be fit into the the supplied buffer. For stream I/O, as much of the record as would fit is read into the buffer, and a subsequent read call will continue reading from the first byte following the last byte read.

Record located by seek_key has been deleted by another opening.

Class: rare Type: I/O system

This indicates that the described situation has occurred while manipulating a file.

Record size must be positive and smaller than a segment.

Class: rare Type: I/O system

A call to the I/O system failed, for the reason described.

Record with key for insertion has been added by another opening.

Class: rare Type: I/O system

This indicates that the described situation has occurred, while manipulating a file.

Relevant data terminated improperly.

Class: rare Type: I/O system

This probably indicates that a final delimiter or data end marker was missing from an I/O record.

Request for connection refused by foreign host.

Class: common Type: ARPANet

This indicates that the requested network connection could not be opened because the foreign host refused to connect.

Request is inconsistent with current state of device.

Class: common Type: I/O system

This indicates an attempt to perform an I/O operation that cannot be performed due to the present state of the I/O device; for instance, hanging up an already hung-up tty line.

Request is inconsistent with state of socket.

Class: rare Type: ARPANet

This indicates that an internal error has occurred in the ARPANet software.

Requested volume not yet mounted.

Class: rare Type: I/O system

This indicates that the volume will probably be mounted in the near future.

Resource specification is invalid.

Class: common Type: RCP

This indicates that an RCP resource specification, such as an argument to an RCP command, does not have valid syntax.

Reverse interrupt detected on bisync line.

Class: rare Type: I/O system

This indicates that an error has occurred while doing I/O on a bisync communications line.

Search list is empty.

Class: common Type: environment

This indicates that an attempt was made to use or examine a search list with no components. The search list must be reinitialized and the operation retried.

Segment contains characters after final delimiter.

Class: rare

This indicates that an ascii segment could not be parsed because it was not in the expected format.

Some directory in path specified does not exist.

Class: common Type: commands

This indicates that one or more of the directories in the specified pathname does not exist as supplied (e.g., misspelled).

Specified access class/authorization is greater than allowed maximum.

Class: rare Type: access control

This indicates that an access class or range specification was unacceptable for the reason described. It only occurs at sites that use the Access Isolation Mechanism (AIM) for access control and occurs in contexts such as Resource Control and Logical Volume (LV) management. Rare, even at AIM sites.

Specified attribute incompatible with file structure.

Class: common　　　　Type: I/O system

This is a vfile_ error that indicates that a vfile_ attach control argument was inconsistent with the file type.

Specified buffer size too large.

Class: rare　　　　Type: I/O system

This indicates that the buffer size is too big for the maximum workspace allowed.

Specified control argument is not implemented by this command.

Class: common　　　　Type: commands

This indicates that the control argument printed in the message is not acceptable to the command.

Specified offset out of bounds for this device.

Class: rare　　　　Type: I/O system

This indicates that the channel program or status queue is not fully contained in the I/O INTERFACER (IOI).

Specified quota account not found

Class: rare　　　　Type: storage system

This indicates that an attempt was made to charge a master directory against a nonexistent (misspelled, for instance) quota account. Use the list_mdirs command with -all to list all the quota accounts.

Specified socket not found in network data base.

Class: rare　　　　Type: ARPANet

This indicates that an internal error has occurred in the ARPANet software.

Specified volumes do not comprise a valid volume set.

Class: common　　　　Type: I/O system

An attempt was made to use a multivolume tape volume set that is invalid.

Specified work class is not currently defined.

Class: rare　　　　Type: tools

This occurs when an attempt is made to use a nonexistent workclass, generally as a result of using the set_work_class command.

Supplied area too small for this request.

Class: rare, programming　　　　Type: subroutine

This indicates that a subroutine was unable to return a value or perform some operation because the area provided was too small.

Supplied identifier already exists in data base.

Class: rare, programming

This should be self-explanatory in the context where it is issued.

Supplied identifier not found in data base.

Class: rare, programming

This should be self-explanatory in the context where it is issued.

Syntax error in ascii segment.

Class: rare

Should be self-explanatory in the context where it is issued. Should also always be accompanied by some explanatory text to describe the specific problem.

The FNP is not running.

Class: rare, programming

This indicates that an operation, such as debugging the FNP, or an operator FNP command, could not be performed because the FNP is not running.

The NCP could not find a free table entry for this request.

Class: rare   Type: ARPANet

This indicates that an internal error has occurred in the ARPANet software.

The Operator refused to honor the mount request.

Class: common   Type: I/O system

This indicates that the operator declined to mount a tape or disk for the user, possibly because it did not exist, could not be found, or was not accessible to the user. Phone the operator for details.

The access name specified has an illegal syntax.

Class: common   Type: access control

This indicates that an access name with invalid syntax (such as Foo.Bar.Baz.Quux) was supplied to an ACL manipulation command or subroutine.

The day-of-the-week is incorrect.

Class: common   Type: environment

This indicates that a date/time string has specified both a date and weekday name (such as "4/26/80 Wednesday"), and that the two are incompatible (that is, that 4/26/80 is not a Wednesday).

The event channel specified is not a valid channel.

Class: rare, programming   Type: ipc_

This indicates that an invalid event channel name was passed to an ipc_ subroutine; probably, the variable containing the channel name is uninitialized.

The event channel table was full.

Class: rare   Type: ipc_

This indicates that an attempt was made to perform an ipc_ operation for which no room could be found in the ECT. This error generally results in process termination.

The event channel table was in an inconsistent state.

Class: rare   Type: fatal process error

This indicates that, due either to a user program malfunction or a system logic error, the event channel table has become damaged. This generally indicates that the process's linkage area has become damaged.

The initial connection has not yet been completed.

Class: rare   Type: ARPANet

This indicates that an internal error has occurred in the ARPANet software.

The item specified is over the legal size.

Class: rare   Type: general

This means that a data item is too large. Supplementary data should always be included in the error message to identify the precise cause of the problem.

There is already a record with the same key.

Class: common   Type: I/O system

This indicates an attempt to add a record to a file that already contains a record with the same key.

The lock could not be set in the given time.

Class: common, programming   Type: subroutines

This indicates that the lock word is already locked, by an existing process, and could not be locked in the time allowed. Possibly the other process was running a program that malfunctioned, and failed to unlock the lock. The lock is not locked to this process, but remains locked to the process that already holds it.

The lock does not belong to an existing process.

Class: common, programming Type: subroutines

This indicates that a lock word has been found to be locked by a process that no longer exists. In general, such a lock should be reset, and the data base it protects should be salvaged if necessary. The lock remains locked to the dead process.

The lock was already locked by this process.

Class: common, programming Type: subroutines

This indicates that an attempt was made to lock a lock that this process already has locked. This may indicate a programming error in the calling program, unless it expects that the lock may already be locked, and will not cause it to be spuriously unlocked. The lock remains locked.

The lock was locked by a process that no longer exists. Therefore the lock was reset.

Class: common, programming Type: environment

This indicates that a lock was found locked by a process that terminated before unlocking it, either by terminating abnormally while performing some operation involving the lock, or by simply failing, due to a logic error in the program, to unlock the lock, and terminating normally. This is normally not a serious condition, and the program that detects it simply continues operating. It may indicate that the data base protected by the lock is inconsistent in some fashion, but not necessarily. When this condition occurs, the lock word is locked for the calling process.

The lock was set on behalf of an operation which must be adjusted.

Class: common Type: I/O system

This is a vfile_ error that indicates that an operation was interrupted while it had part of a vfile_ file locked, and that the file must be adjusted (via vfile_adjust) before anything else can be done with it.

The logical volume is already attached.

Class: rare Type: storage system

This indicates that an attempt was made to attach a private logical volume that is already attached to the requesting process. It remains attached.

The logical volume is already defined.

Class: rare Type: storage system

This indicates that an attempt was made to redefine an existing logical volume.

The logical volume is full.

Class: common Type: storage system

This indicates that there is no room left on a logical volume to allocate another page (at page reference time) or another VTOC entry (at segment creation time). This generally occurs as a seg_fault_error condition, indicating that a new page could not be allocated, or when a call is made to create a segment.

The logical volume is not attached.

Class: common Type: storage system

This indicates that an attempt was made to reference a segment residing on a private logical volume that is not attached to the requesting process.

The logical volume is not defined.

Class: common Type: storage system

This indicates that an attempt was made to reference a segment residing on a logical volume that either does not exist, or has not been defined for the bootload. This can happen when a public logical volume has been taken offline because of disk problems or similar difficulties.

The logical volume table is full.

Class: rare Type: storage system

This indicates that there is no room in the system logical volume table to add a new logical volume.

The maximum depth in the storage system hierarchy has been exceeded.

Class: common Type: commands

This indicates that an attempt was made to create a segment or directory deeper than is permitted in the directory hierarchy. A maximum depth of sixteen directories is permitted.

The name specified contains non-ascii characters.

Class: rare

This indicates that an attempt was made to use a name containing non-ascii characters, which are not allowed in many contexts.

The name was not found.

Class: rare, programming Type: subroutines

This indicates that the specified name could not be found where expected. This should always be accompanied by some further explanatory information.

The process's limit for this device type is exceeded.

Class: common Type: I/O system

This indicates that the process has reached the limit on the number of devices of a particular type that it may be assigned at once; the unassign_resource command may be used to free up unneeded ones. This limit is a site-determined parameter; contact your system administrator for details.

The reference name table is in an inconsistent state.

Class: rare, programming

This indicates, most likely, that the process's linkage section has been damaged by a malfunctioning program. A new_proc should be issued, and the program fixed.

The request is inconsistent with the current state of the resource(s).

Class: common Type: RCP

This indicates that the requested operation cannot be performed because it is inconsistent with the resource state; for instance, freeing a resource that is already free.

The requested action was not performed.

Class: rare, programming Type: subroutines

This message indicates general failure on the part of a command or subroutine. It does not indicate any specific error condition, but rather that a condition was encountered that could not be described more exactly by a more specific error code.

The requested device is not available.

Class: common Type: I/O system

This indicates that the process could not be assigned an I/O device, possibly because there are no devices of that type at the site, because they are all in use or offline, or because the user does not have access to use them.

The resource is presently in use by a system dumper.

Class: rare Type: storage system

This indicates that an operation may not be performed on a segment or directory because the system dumper process is presently using it. This condition should go away quickly, so that the operation will succeed if retried.

The rest of the tape is blank.

Class: common Type: I/O system

This indicates that an attempt has been made to read past the last record or file mark on a tape, and that the rest of the tape has nothing written on it.

The ring brackets specified are invalid.

Class: common Type: access control

This indicates an attempt to set ring brackets to an invalid value, such as one of the three values outside the range of zero to seven, or that the three values do not satisfy the relation $R1 \leq R2 \leq R3$.

The signaller could not use the saved sp in the stack base for bar mode.

Class: rare, programming   Type: environment

This indicates that the stack has been overwritten in such a way as to damage it, and is usually caused by user program malfunction.

The specified access class/authorization is not within the permitted range.

The specified access classes/authorizations are not a valid range.

Class: rare   Type: access control

These two messages indicate that an access class or range specification was unacceptable, for the reasons described. They only occur at sites that use the Access Isolation Mechanism (AIM) for access control. They occur in contexts such as Resource Control and Logical Volume (LV) management. Rare, even at AIM sites.

The specified subsystem either does not exist or is inconsistent.

Class: rare

This indicates an attempt to use a prelinked subsystem (specified by -subsystem at login time) that is either inconsistent or nonexistent. The subsystem is considered inconsistent if the hardcore supervisor version is different from what it was when the subsystem was prelinked; if this happens, the subsystem must be re-prelinked by a systems programmer.

The specified terminal type is incompatible with the line type.

Class: rare   Type: I/O system

This indicates that the specified terminal type cannot be used because it is not compatible with the type of communications line the terminal is attached to.

The specified volume cannot be unloaded from its device.

Class: common   Type: I/O system

Certain types of devices may not be unloaded, such as nondemountable disk packs.

The time is incorrect.

Class: common   Type: environment

This indicates that the time specified in a date/time string is invalid.

The year is not part of the 20th Century (1901 through 1999).

Class: common   Type: environment

Multics standard date/times must be part of the 20th century, and the supplied year was not.

There is already a record with the same key.

Class: common   Type: I/O system

This indicates an attempt to add a record to a file that already contains a record with the same key.

There is an inconsistency in arguments to the storage system.

Class: common, programming   Type: storage system, commands

This indicates that a storage system subroutine was given an invalid argument. It is used in any situation where there is no more specific error code to describe the problem, and even (by older routines) when there is.

There is an inconsistency in this directory.

Class: rare   Type: storage system

This indicates that a directory has become inconsistent and cannot be salvaged. It should never happen; if it does, system maintenance personnel should be informed.

There is an internal inconsistency in the segment.

Class: common

This indicates that the segment is not an object segment, or is otherwise unacceptable to the command.

There is no initial connection in progress from this socket.

Class: rare   Type: ARPANet

This indicates that an internal error has occurred in the ARPANet software.

There is no more room in the file.

Class: common   Type: I/O system

This indicates that the specified file is full, and that no more data may be added to it.

There was an attempt to create a copy without correct access.

Class: rare   Type: storage system

This indicates that an attempt to copy a segment with its copy switch set failed because of incorrect access.

There was an attempt to delete a non-empty directory.

Class: rare, programming   Type: storage system

This indicates that the specified directory cannot be deleted because it still contains branches; this code is only returned by the storage system primitive for deletion.

There was an attempt to delete a segment whose copy switch was set.

Class: rare — Type: storage system

A segment's copy switch must be turned off before it may be deleted.

There was an attempt to make a directory unknown that has inferior segments.

Class: rare — Type: storage system

This indicates that, due to a logic error in the supervisor, a directory cannot be made unknown. A systems programmer should be contacted if this message occurs frequently; in general, the failing operation will succeed if retried.

There was an attempt to move segment to non-zero length entry.

Class: rare — Type: storage system

This indicates an attempt to use the storage system primitives to move a segment into another segment that already contained data.

There was an attempt to terminate a segment which was known in other rings.

Class: common — Type: storage system

This indicates an attempt to terminate an inner ring segment (e.g., a mailbox). The operation cannot be performed because it violates Multics ring validation security.

There was an attempt to use an invalid segment number.

Class: common — Type: storage system

This indicates an attempt to call a storage system primitive with a pointer that does not correspond to any segment in the process's address space, and therefore cannot be manipulated.

There was an illegal attempt to delete an AST entry.

Class: rare, programming — Type: storage system

This indicates that, due to a logic error in the supervisor, a segment cannot be deactivated. A systems programmer should be contacted if this message occurs frequently; in general, the failing operation will succeed if retried, or, failing that, if retried after a new_proc.

This entry cannot be traced.

Class: programming — Type: trace

This indicates that an attempt was made to trace an entrypoint that cannot be traced; for instance, a gate entrypoint.

This operation allowed only on master directories.

Class: rare — Type: storage system

This indicates that master directory operations (e.g., set_mdir_quota) may only be performed on master directories.

This operation is not allowed for a directory.

Class: common Type: commands

This indicates that an attempt was made to perform an operation that cannot be done on a directory, such as printing or editing.

This operation is not allowed for a link entry.

Class: common Type: commands

This indicates that a procedure was used to attempt to perform an operation on a link that may only be performed on a segment or a directory.

This operation is not allowed for a master directory.

Class: rare Type: commands

This indicates that the requested operation cannot be performed for a master directory, such as moving quota to it, rather than using the set_mdir_quota command.

This operation is not allowed for a multisegment file.

Class: common Type: commands

This indicates that the requested operation may not be performed on a multisegment file.

This operation would cause a reference count to vanish.

Class: rare, programming

The meaning of this error should be clear in context.

Too many ""<"" 's in pathname.

Class: common Type: commands

This indicates that a pathname argument contains too many "<" characters. See the MPM Reference Guide for details.

Trap-before-link procedure was unable to snap link.

Class: rare Type: linker

This is issued by the binder or linker (as the code for a linkage error), and generally indicates a damaged object segment.

UID path cannot be converted to a pathname.

Class: rare Type: storage system

This indicates that a UID pathname was invalid. This occurs mostly when manipulating logical volume accounts, and indicates a damaged logical volume control segment, which can be rebuilt using the register_mdir command. It can also be given by various system analysis tools.

Unable to complete connection to external device.

Class: rare Type: I/O system

This indicates that a communications channel connection could not be established (e.g., a dial_out call could not be completed).

Unable to convert access class/authorization to binary.

Unable to convert binary access class/authorization to string.

Class: rare Type: access control

These two messages indicate that an access class or range specification was unacceptable for the reason described. They only occur at sites that use the Access Isolation Mechanism (AIM) for access control. They occur in contexts such as Resource Control and Logical Volume (LV) management. Rare, even at AIM sites.

Unable to convert character date/time to binary.

Class: common Type: environment

This indicates that a date/time string could not be converted successfully because it did not conform to the syntax described in date_time_strings.gi.info.

Unable to process a search rule string.

Class: common Type: commands

This indicates that a search rule name, as supplied to the set_search_rules command, is not acceptable (e.g., a bad keyword or nonexistent directory).

Undefined preaccess command.

Class: common

This indicates an attempt to use an undefined preaccess command before logging in (probably a misspelling).

Unrecoverable data-transmission error on physical device.

Class: common Type: I/O system

This indicates that a parity error has occurred while doing I/O to a device, and that the operation was not successful. The operation may succeed if it is retried.

User has deferred messages.

Class: common Type: mail system

This indicates that an interactive message could not be delivered immediately because the recipient has temporarily deferred messages. The message will be delivered as soon as the recipient accepts messages again.

INDEX

CUT ALONG LINE

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form

TITLE	LEVEL 68 MULTICS ERROR MESSAGES: PRIMER AND REFERENCE MANUAL	**ORDER NO.**	CH26-00
		DATED	SEPTEMBER 1980

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be investigated by appropriate technical personnel and action will be taken as required. Receipt of all forms will be acknowledged; however, if you require a detailed reply, check here. ☐

FROM: NAME ______________________ **DATE** ____________

TITLE ______________________

COMPANY ______________________

ADDRESS ______________________

____

PLEASE FOLD AND TAPE–
NOTE: U. S. Postal Service will not deliver stapled forms

FOLD ALONG LINE

CUT ALONG LINE

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 39531 WALTHAM, MA02154

POSTAGE WILL BE PAID BY ADDRESSEE

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTN: PUBLICATIONS, MS486

FOLD ALONG LINE

Honeywell

Honeywell

Honeywell Information Systems
In the U.S.A.: 200 Smith Street, MS 486, Waltham, Massachusetts 02154
In Canada: 2025 Sheppard Avenue East, Willowdale, Ontario M2J 1W5
In the U.K.: Great West Road, Brentford, Middlesex TW8 9DH
In Australia: 124 Walker Street, North Sydney, N.S.W. 2060
In Mexico: Avenida Nuevo Leon 250, Mexico 11, D.F.

29370, 7.5C1080, Printed in U.S.A.