HONEYWELL

MULTICS qedx TEXT EDITOR USER'S GUIDE

SOFTWARE

MULTICS

qedx TEXT EDITOR USER'S GUIDE

SUBJECT

Detailed Description of the Multics qedx Text Editor, Including Comprehensive Discussions of the Related Requests and Editing Techniques

SPECIAL INSTRUCTIONS

This manual presupposes some basic knowledge of the Multics system provided by the 2-volume set, *New Users' Introduction to Multics*. Some of the preliminary information covered in that set is briefly summarized here, however, so that users at any level of experience can comprehend the techniques covered in this manual.

SOFTWARE SUPPORTED

Multics Software Release 10.1

ORDER NUMBER

CG40-01

February 1983

Honeywell

PREFACE

This book is a detailed description of qedx, a Multics text editor; it provides all the necessary information to edit text and programs online. Users reading this book are expected to be familiar with the Multics concepts described in the 2-volume set, New Users' Introduction to Multics - Part I (Order No. CH24), and - Part II (Order No. CH25).

The sections of this manual fully describe the qedx editor and explain the steps required, from the most basic, elementary tasks through advanced methods for accomplishing the same tasks in a more efficient, effective and powerful way, up to the most sophisticated techniques for moving text and creating macros.

Section 1 tells how to log in (establish a connection with the computer), and how to log out (break the connection); simple methods for deleting (erasing) a mistyped character, word, or line and how to enter the qedx editor ("call" qedx) and exit from it.

Section 2 explains how to type new information in from your terminal (create text) and how to save that information for reuse (write a segment).

Section 3 extensively describes editing an existing segment; ways to locate, print, and delete lines; make substitutions on a line(s); add, replace and insert more text; explains "regular expressions" and special characters for use in editing; and shows how to print, list, and delete segments.

Section 4 shows several examples of real editing sessions and lists some hints for new users.

Section 5 details the advanced edit requests, and describes special escape sequences, use of buffers for moving text, repeated editor sequences, and macros.

Appendix A is a glossary of qedx terminology, and Appendix B is the command description itself (which also appears in the MPM Commands). Appendix C is a summary of addressing conventions; Appendix D contains the reference descriptions for all requests; and Appendix E lists qedx error messages, including what corrective actions to take.

Significant Changes in CG40-01

The manual format has been extensively revised. There are no major changes to the text in this revision.

The Multics Commands and Active Functions, Order No. AG92, is referred to in this manual as the MPM Commands; and the Multics Subroutines, Order No. AG93, is referred to here as the MPM Subroutines.

Manual Conventions

Often, user typed lines and lines typed by Multics are shown together in the same example (these examples are called interactive examples). To differentiate between these lines, an exclamation mark (!) precedes user-typed text. This is done only to distinguish user text from system-generated text; you should not actually begin your text with an exclamation mark.

Also, a "carriage return" (moving the typing mechanism to the first column of the next line, called a newline on Multics) is implied at the end of every user-typed line.

CONTENTS

		Page
Section 1	Introduction	1-1
	Getting Started	1-1
	Logging In To Multics	1-1
	Logging Out	1-3
	Correcting Typing Errors	1-3
	The qedx Editor	1-4
	Invoking qedx	1-5
	Exiting From qedx	1-5
	qedx Requests	1-5
	Operation Modes	1-6
	Input Requests vs. Edit Requests	1-6
	Input Mode	1-8
	Edit Mode	1-8
	Requests	1-8
	Input Requests	1-9
	Edit Requests	1-9
Section 2	Entering New Text	2-1
	Buffers	2-1
	Creating Text	2-2
	append (a, \f) request	2-2
	Saving Your Work	2-3
	write (w) request	2-3
	Naming the Saved Segment	2-4
	Exiting from qedx	2-6
	quit (q) request	2-6
Section 3	Editing An Existing Document	3-1
	Locating and Printing Lines	3-1
	Current Line And Pointer	3-2
	Printing The Current Line	3-2
	Addresses	3-2
	Absolute Line Number	3-3
	Printing a Single Line	3-4
	Printing More Than One Line	3-5
	Relative Line Number	3-6
	Current Line	3-6
	Print Current Line Number (=)	3-6
	Last Line ($)	3-6
	Printing The Entire Contents Of Your Buffer	3-8

		Page
	Interrupting a Lengthy Print Request	3-8
	Context Addressing (Regular Expressions)	3-9
	Special Characters in Regular Expressions	3-10
	Null Regular Expression	3-13
	Compound Addresses	3-14
	Default Addresses	3-14
	Common Mistakes In Addressing	3-15
	Substitute Request	3-15
	The Ampersand (&)	3-18
	Deleting Lines	3-18
	Adding More Text	3-19
	Replacing Lines of Text	3-20
	Inserting Text	3-22
	Printing, Listing, and Deleting Segments	3-22
Section 4	Sample Terminal Session	4-1
	Sample Invocation	4-1
	Editing Example With Both Input And Edit Requests	4-5
	Helpful Hints for New qedx Users	4-6
Section 5	Advanced Edit Requests	5-1
	Extended Edit Requests	5-2
	How To Execute a Multics Command From Inside qedx	5-2
	e (execute) request	5-2
	Global Printing, Deleting, And Printing Line Numbers	5-4
	g (global) request	5-4
	v (exclude) request	5-5
	Buffer Requests	5-7
	Creating and Changing Buffers	5-8
	b(Name) request	5-8
	Moving Blocks Of Lines (Cut And Paste)	5-8
	m(Name) request	5-8
	How To Check The Status Of Your Buffers	5-9
	x (status) request	5-9
	Repositioning The Pointer	5-10
	n (nothing) request	5-10
	Annotating (Comment) Macros	5-10
	" (comment) request	5-10
	Special Escape Sequences	5-11
	Use of Buffers for Moving Text	5-12

		Page
	Repeated Editor Sequences	5-15
	Editor Macros	5-16
	Initialization of Macros	5-19
	Additional Arguments	5-22
	Notes on Macro Use	5-23
Appendix A	Glossary	A-1
Appendix B	qedx Command	B-1
	qedx, qx	B-1
Appendix C	Summary of Addressing Conventions	C-1
Appendix D	Request Descriptions	D-1
	a (append)	D-3
	c (change)	D-6
	i (insert)	D-7
	r (read)	D-9
	p (print)	D-12
	= (print line number)	D-16
	d (delete)	D-18
	s (substitute)	D-20
	w (write)	D-23
	q (quit)	D-25
	e (execute)	D-26
	g (global)	D-27
	v (exclude)	D-29
	b (change buffer)	D-31
	m (move)	D-32
	x (buffer status)	D-33
	n (nothing)	D-34
	" (comment)	D-36
Appendix E	qedx Error Messages	E-1
Index	. . . . .	i-1

ILLUSTRATIONS

Figure 1-1.	Modes of qedx	1-7

SECTION 1

INTRODUCTION

This section briefly describes preliminary actions such as logging in, logging out, and correcting errors as you type; these are fully described in the New Users' Introduction-Part I. The procedures for entering and exiting the qedx editor are also explained here, along with a brief introduction to qedx editing requests.

GETTING STARTED

The first thing to do is establish a connection with the computer; this is called logging in. To log in you must be registered on Multics as a member of a certain project. You are given a User_id (user identification) which consists of a Person_id (name) and Project_id (project name). For example, Mary Smith, working in the sales department, may be given the following User_id:

Smith.Sales

This User_id belongs to Mary alone; no one else can use it because Mary also has a password, which along with her User_id allows her to use Multics. The password is a secret between Mary and the system ("the system" is the Multics computer system).

Logging In To Multics

The procedure for logging in is explained in depth in the New Users' Introduction-Part I. To briefly summarize the method for establishing a connection between your terminal and Multics, you turn power on for the terminal, dial the appropriate telephone number, and when you hear a beep signal, either press a button or place the telephone receiver in the modem and press the linefeed key. (This method is employed unless your terminal is directly connected to Multics, in which case you do not need to dial a phone number.)

When a connection has been established, a header like this is printed by Multics on the terminal:

```
Multics 8.0:  PCO, Phoenix, Az.
Load = 26.0 out of 100.0 units:  users = 26
```

At this point, you issue the login command and your Person_id, separated by a blank. Multics commands are words that you type to tell the system an action to take. In this case specify that you want to log in.

```
!  login Smith
   Password:
!
```

NOTE: In examples throughout this document an exclamation mark (!) precedes text typed by you, to differentiate between Multics-generated and user-typed lines. You should not actually begin your text with an exclamation mark. Also, a "carriage return"--moving the cursor to the first column of the next line (called a "newline" on Multics) is implied at the end of every user-typed line. The carriage return conceptually sends the typed line to the computer. See the glossary under newline for details.

Multics then requests your password. Depending on your terminal, the printing of the password is either suppressed (i.e., nothing prints out when you type it) or hidden in a string of cover-up characters typed by the system. If you make an error during the log-in procedure, the system informs you of it and asks you to try again:

```
!  (Mary mistyped her password here)
   Incorrect password supplied.
   Please try again or type "help" for instructions.
!  login Smith
   Password:
!  (Mary correctly typed her password here)
```

After typing your password successfully, the system responds with information regarding your last log in:

```
Smith Sales logged in 10/19/82 0937.5 mst Tue from ASCII
     terminal "234".
Last login 10/18/82 1359.8 mst Mon from ASCII terminal
"234".
```

Once the system has logged you in it prints a ready message. This may take a little time. This message is printed to indicate that Multics is at command level and ready to receive a command. The ready message consists of the letter "r" followed by the time of day and three numbers that reflect system resource usage. For more information about the ready message, refer to the description of the ready command in the MPM Commands.

```
r 937 1.314 1.332 30
```

The complete log-in sequence for Mary Smith is:

```
login Smith
Password:
(Mary types her password here)
Smith Sales logged in 10/19/82  0937.5 mst Tue
  from ASCII terminal "234".
Last login 10/18/82  1359.8 mst Mon from ASCII terminal "234".
r 937 1.314 1.332 30
```

Logging Out

To break the connection between the terminal and Multics, issue the logout command. The system responds by printing your identification, the date and time of the log out, and the total CPU time and memory units used:

```
logout
Smith Sales logged out 10/19/82 1249.4 mst Tue
CPU usage 17 sec, memory usage 103.1 units.
hangup
```

The word "hangup" is printed by Multics to remind you to hang up the telephone and to show you that the connection has been broken on purpose.

Correcting Typing Errors

There are two special symbols for correcting typing errors whether you are at Multics command level or using the editor. They are the character-delete ("erase") and the line-delete ("kill"). The number sign (#) is the character-delete symbol, and the commercial at sign (@) is the line-delete symbol.

The character-delete symbol "erases" one previously typed character when typed directly after the error. The line-delete symbol "erases" every character previously typed on the line ("kills" the whole line). Examples of both symbols are given in the login command lines below. Each line is interpreted by Multics as "login Smith":

```
login SM#mith

logen ###in Smith

logen Smit@login Smith

kigum @loge#in Smith

login    # Smith
```

Notice that several successive number signs erase an equal number of typed characters preceding them (see the second line above). However, a single number sign following "white space" (any combination of spaces and horizontal tabs) erases all the white space (see the last line above). This white space rule saves you the trouble of remembering how many spaces or tabs you just typed, and reduces the number of keystrokes necessary to remove any white space.

Note that number signs and commercial at signs do not cause characters to be erased from the screen of a video terminal, but Multics responds as though the characters have been erased when you type the newline.

THE qedx EDITOR

The Multics qedx editor is used by word processing personnel, programmers and administrators to enter data into the * computer system and make corrections or updates to that data. The qedx editor is used by people in a wide variety of jobs, for many different kinds of tasks, but they all share in common the need for (1) creation of a document (letter, memo, program), and (2) retention of that document for updating or correcting errors.

The name of the editor, qedx, is also a command (its short name is qx) that you type when you want Multics to "turn on" or invoke the editor. You invoke the editor to either (1) type new text into Multics or (2) edit some already-existing text. Typing new text into the system is often referred to in this manual as creating text.

Invoking qedx

When you type:

```
! qedx
```

the result is as if someone had placed in front of you a blank pad of paper, pencil, eraser, and file cabinet. The imaginary blank pad is referred to by qedx as the buffer. Throughout this manual there are many references to the buffer; for now, think of the buffer as that blank paper facing you. (Remember, you must type a carriage return after every line you type at your terminal, or the computer doesn't receive it.)

There is no acknowledgement (Multics prints no prompt) stating that you are in qedx; just assume that the editor is waiting for you to tell it to do something (see "qedx Requests" below).

It may seem premature to introduce the method for quitting from qedx before describing methods of editing; however, it is reassuring to know ahead of time and especially helpful when beginning to experiment with qedx.

When you type:

```
! q
```

you exit (i.e., quit) the qedx editor. After you quit, the system responds with a ready message; you are back at command level. This is the qedx quit request (typed: q).

```
! qx                              INVOKE QEDX AND
! q                               EXIT IMMEDIATELY
  r 651 0.150 0.020 4
```

qedx REQUESTS

To perform editing functions (create text, print it, change or delete it), qedx provides requests. If you think again of the buffer as a blank piece of paper, you can see what a request does. Requests make it possible for you to write on the blank pad of paper, as well as erase and rewrite. Finally, you can crumple it up and throw it away or place it in your file cabinet.

NOTE: Your work is always thrown away unless you ask for it to be filed (called "writing it to a segment").

These actions are explained in detail in Sections 2 and 3. All requests are entered as lowercase letters. You can use the erase and kill symbols mentioned above to correct typing errors on qedx requests.

Operation Modes

INPUT REQUESTS VS. EDIT REQUESTS

There are two kinds of qedx requests: input requests and edit requests. After issuing an input request, you are in input mode; when not in input mode, you are in edit mode. To create text you must be in input mode and to edit text you must be in edit mode. However, to edit you must have pre-existing text to edit. Enter qedx and use an input request to create new text or enter qedx and immediately read in existing text you intend to edit.

When entering qedx, you are automatically in edit mode. You must take explicit action to switch from edit to input and vice versa. The request that you issue determines your mode; an input request switches you from edit to input mode, and an input terminator terminates your input and switches you back to edit mode.

When working in qedx, make sure you know which of the two modes you are in. You can verify which mode you are in by typing the input terminator (\f). If you are in input mode, you will be switched to edit mode. If you are in edit mode, qedx prints an error message on your terminal, and you remain in edit mode.

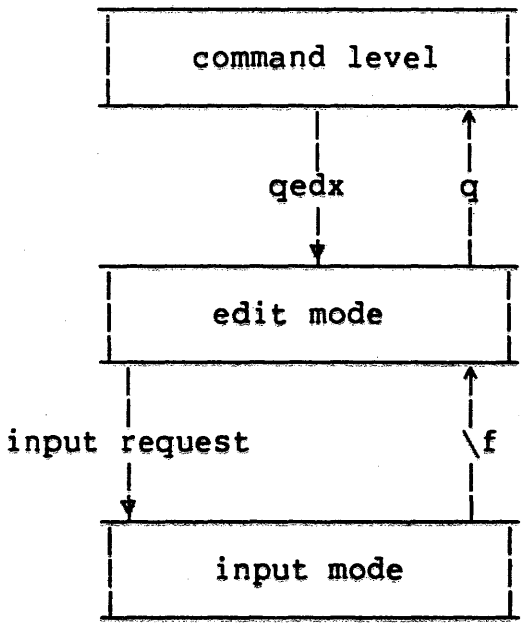

Figure 1-1. Modes of qedx

When you are in input mode, qedx does not verify that fact; that is, there is no response from qedx to you. In edit mode there is no response either; however, you can easily verify that you are in edit mode by issuing the input terminator. If you are in edit mode, and type "\f", qedx responds by printing a message indicating that the backslash is not a valid qedx request. If you are in input mode and type "\f", qedx takes you out of input mode.

Input Mode

Input mode allows you to enter new text from the terminal, until you type "\f" to leave input mode; it is analogous to writing lines of text on a page of your blank pad. Enter input mode by typing an input request, followed by a newline character. The lines that follow are all accepted as text, until you leave input mode by typing the input terminator (escaping from input mode).

Section 2 explains how to input text, as well as how to save it.

Edit Mode

Edit mode allows you to edit existing text. You can print, delete, add, and make substitutions on a line of text. Also, you can perform these operations on more than one line, or on every line of your text, with one request.

As previously mentioned, when you first enter qedx you are in edit mode, and to edit text, you must have already created it. In edit mode, you take the page of text from your pad and begin to erase and rewrite, then destroy the old copy by replacing it with the rewritten one.

Section 3 describes the edit requests used to perform the qedx equivalents of these actions.

Requests

In the qedx editor, input and edit modes operate according to the descriptions of the requests listed below.

It is important to remember that the only messages qedx prints at your terminal are error messages (see Appendix E). When most requests are performed, nothing is printed. The best way to acquire proficiency and familiarity with the modes and requests of qedx is to try them.

Following is a summary of basic qedx requests (which are fully described in Sections 2 and 3). They are listed in the form:

X (Y)
DESCRIPTION

where X is the request (what you type to perform the operation), Y is what the request stands for, and DESCRIPTION tells what the request does.

Many requests can operate on more than one line, or lines other than the current one. See the discussion of addressing in Section 3.

INPUT REQUESTS

a (append)
enters input mode and appends (adds) lines typed from the terminal into the buffer after the current line.

c (change)
enters input mode and changes (replaces) the current line with lines typed from the terminal.

i (insert)
enters input mode and inserts lines typed from the terminal before the current line.

NOTE: The input terminator (\f), leaves input mode and enters edit mode. The sequence \034 is a synonym \f. On terminals with no backslash, use ¢f instead.

EDIT REQUESTS

w PATH (write)
saves your work; names the contents of the buffer PATH and puts PATH in permanent storage.

r PATH (read)
reads (copies) the contents of the existing text named PATH into your buffer.

p (print)
prints the contents of the line.

= (print line number)
prints the current line number.

s/old/new/ (substitute)
makes a change on a line; substitutes every occurrence of "old" with "new" on a line ("old" is a regular expression--see the discussion of regular expressions in Section 3).

d (delete)
erases the current line.

q (quit)
takes you out of the editor and returns to Multics command level. Remember, your work is not saved until you explicitly save it with the write request.

There are other requests, for advanced editing and programming, described in Section 5.

SECTION 2

ENTERING NEW TEXT

This section describes how to enter new text into a buffer using an input request, how to leave input mode, how to save the contents of the buffer with the write request, and the conventions used when naming a segment.

BUFFERS

A buffer is a temporary work space--the "blank pad of paper" previously mentioned. You can create a few lines of text in it, or many pages; you can make changes on lines, delete them and add more.

However, all of this text is temporary until you do something to make it permanent. When you want to save some text that you have created in your buffer, you write the contents of the buffer into a segment. A segment can be compared to a labeled "file folder" in a "filing cabinet"; it is the basic unit of information stored on Multics. It may contain data, text, or programs, and has a name (called its entryname). (For more information on segments, entrynames, and permanent storage, see the New Users' Introduction.) When you do a write request, the temporary copy in the buffer is duplicated and given an entryname by which you can refer to it, and is put in permanent storage. You can now keep this segment for an indefinite period of time, edit it (saving changes), and manipulate it in other ways (some are listed at the end of Section 3, "Printing, Listing, and Deleting Segments").

When you call qedx and are presented with an empty buffer, you either enter new text into this empty buffer or read an existing segment into it. The following example shows that the buffer is empty by using the "p" (print) request after entering qedx. There is nothing for the request to print:

```
! qx
! p
  Buffer empty
```

The error message "Buffer empty" is printed by qedx; the error consists of asking qedx to print when there is nothing to print. (Refer to Appendix E for other qedx error messages.)

CREATING TEXT

append (a, \f) request

Since the buffer is empty when you enter qedx, your first request to qedx should be to enter new text. You can enter new text using the append request (typed: a). The append request tells qedx to append all following lines to the contents of the buffer, until you indicate otherwise by issuing the input terminator (typed: \f). Since the buffer is empty, any input appended becomes the only contents of the buffer. The input text typed on your terminal following this request may be a single line, several lines, or an entire document. The input terminator tells qedx that you are finished entering input and wish to return to edit mode to issue other qedx requests.

The example below shows how text is put into the empty buffer using the append request, then the input terminator (\f) is typed to leave input mode and go into edit mode:

```
! qx ENTER QEDX
! a ISSUE APPEND REQUEST
! The Multics text editor, qedx, TYPE IN TEXT
! is a line-oriented editor.
! It performs editing
! functions on lines,
! using requests.
! \f LEAVE INPUT MODE
                                    (INPUT TERMINATOR)
```

Remember that the append request puts you in input mode. A common error is to enter qedx and forget to type an "a", attempting to input text immediately.

The example below shows what happens when a user enters qedx and forgets to type the append (a) request to begin input. As soon as the user types a carriage return at the end of the line of text, qedx prints an error message and the user knows that the text has not been input. (The "p" request verifies that the buffer is empty.) At this point the user recognizes the mistake and types the append request to correctly enter the text, ending the input with "\f" on the following line:

```
! qx
! This is the beginning of  new text A MISTAKE-FORGOT TO qedx:
  T not recognized as a request.  ISSUE APPEND REQUEST
! p Buffer empty.
! a ISSUE APPEND REQUEST
! This is the beginning of new text TYPE IN TEXT
! \f LEAVE INPUT MODE
```

SAVING YOUR WORK

write (w) request

When you exit from qedx, whatever is in your buffer is lost if it has not been written into permanent storage (your "file cabinet"). You can save your work by making up a name for the segment (e.g., work) and typing this name after a write request (typed: w):

```
! w work      WRITING OUT A BUFFER
                (SAVING THE CONTENTS)
```

The write request is mandatory once you create something that you want to save: if you issue the write request, your "paper" from the blank pad which now contains your work is saved in storage ("filed"). If you neglect to write your work and exit from qedx, your work is thrown away, never to be seen again.

You must be in edit mode to issue the write request or any other edit request. When you leave input mode by typing "\f", you automatically enter edit mode. If you are not in edit mode, edit requests are interpreted as input to be added to the buffer. (If this happens, you will have to delete them.) The "w" request saves the entire contents of the buffer in a segment that has the name that you specify. When you type the "w" request correctly, Multics does not respond. You can assume that the text is written in the specified segment, and continue your work.

NAMING THE SAVED SEGMENT

The entryname you give for the segment can be from one to 32 characters long. The write request is followed by a Multics pathname. (Pathnames are fully described in the New Users' Introduction.) You usually specify a segment by entryname, which is a name without any greater-than or less-than characters (><). The qedx command restricts the characters that can be used in an entryname in a write request. The allowed characters include:

letters (uppercase and lowercase)
digits (0 to 9)
`'_~^+-.:{}
*

The entryname cannot begin or end with a period (.) or contain two or more consecutive periods. These restrictions are necessary because qedx users sometimes think they are in input mode and type lines such as:

what time is it?
(contains spaces)
w/o your cooperation...
* (contains a slash, spaces, and three periods)

| In edit mode these lines would cause write operations, if the
| entryname restrictions did not prevent this. If segments were
| written, their names (e.g., "hat time is it?") would cause
| problems when you tried to use them. Characters like the asterisk (*), for instance, have special meanings; a name
| containing an asterisk invokes the Multics star convention (described in the New Users' Introduction).

It is not necessary to leave a space between the "w" (write) and the intended name; however, the space is recommended to avoid confusion and inadvertent naming. For example, if you type:

! what

while you are in edit mode, you write the contents of your buffer into a segment named "hat" because the "w" is interpreted as a
* write request rather than the initial letter of the name. If you type:

! w what

in edit mode, you write the contents of your buffer into a segment named "what".

In summary, the segment-naming restrictions are intended as protection against accidental loss of data by writing unknowingly into a strange segment that mysteriously appears in your directory (see the discussion of listing segments in Section 3).

The following example shows text input as explained previously, taken a step further with the "w" request to name and permanently store the contents of the buffer in a segment. In terms of the conceptual blank pad of paper, the example shows how you write text on that paper and then file the paper in a folder under the heading "made_up_name" in your filing cabinet.

```
! qx
! a
! The Multics text editor, qedx,          TYPING IN TEXT
! is a line-oriented editor.
! It performs editing
! functions on lines,
! using requests.
! \f
! w made_up_name                            SAVING IT
```

Now you have a segment called "made_up_name", which is a permanent copy of the buffer text.

The following example shows a case where you have entered some text, written it to a segment, and you now want to add more lines to it. Switching from input mode to edit mode (which is necessary to store the text), back to input to add more lines, and finally back to edit to save the new lines is a common sequence of operations. You don't have to exit qedx when you have finished input and enter qedx again in order to edit.

```
! qx
! a
! The Multics text editor, qedx,
! is a line-oriented editor.
! \f
! w new_name
! a
! It performs editing                  ADDING MORE TEXT
! functions on lines,
! using requests.
! \f
! w                                    DEFAULT PATHNAME (ASSUMED)
```

The last line in the previous example, a "w" alone on a line, shows that once you have named the segment ("window_name"), qedx remembers that name and assumes it for later write requests when no name is specified. The name is forgotten when you exit from qedx.

It is a good idea to issue many "w" requests during a long qedx session, in case there is trouble with the telephone line or terminal and you become accidentally cut off. If this happens, changes made after your last "w" can be lost.

Input and editing are performed not on the saved segment but on a copy of a segment in the buffer. The permanent segment is therefore protected against inadvertent destruction, since the work you do is on a "duplicate".

There are two other input requests: the change request (typed: c) and the insert request (typed: i). These requests are discussed in Section 3 since they are particularly useful for manipulating existing text (replacing and inserting lines). Use the append request when creating new text. The change and insert requests are more practical for handling existing text.

EXITING FROM qedx

quit (q) request

Instructions for exiting the editor are repeated here to remind you that if you are creating work to be saved, you must write it to a segment prior to exiting or it will not be saved. Remember that to save your latest changes, you must issue a final "w" request before quitting. After you quit, Multics responds with a ready message; you are back at command level.

```
! qx
! a
! A few lines of text
! \f
! w lines
! q
  r 651 0.150 0.020 4 COMMAND LEVEL
```

In the previous example, the text has been saved; it is contained in a segment named lines, and can be edited at a later time (see Section 3).

WARNING: qedx does not automatically save your work; to save it you must use the write request.

If you type a "q" and nothing happens, you are almost surely in input mode. The "q" has probably been entered into the buffer; preceding lines may also be part of the text--that is, entered unintentionally. This type of mistake is also common to new users. Remember, if you type any edit request, expecting a response and receiving none, you are probably in input mode. Type "\f", then "p" to print your current line. You may have to delete some edit requests that have become part of the buffer (see Section 3).

Assume in the example below that as you begin to use qedx, you make the common mistake of forgetting to end your input with "\f". You add text, and then type a "q" while in input mode. When the "q" does not take effect (i.e., you do not receive a ready message indicating that you are at command level), you recognize the fact that you have neglected to leave input mode. Type a "\f", delete the lines mistakenly added to the buffer, write the correction, and then type another quit request:

```
! a
! and more text and I'm done.
! w lines                          MISTAKE--YOU WAIT A LONG TIME
! q                                  AND RECEIVE NO READY MESSAGE
! \f                               END INPUT
! -1 p                             LOCATE AND DELETE MISTAKES
  w lines                            (DESCRIBED IN SECTION 3)
! d
! p
  q
! d
! w lines                          SAVE
! q                                QUIT
  r 850 0.377 0.426 36
```

SECTION 3

EDITING AN EXISTING DOCUMENT

When you enter qedx and issue the read request (r name_of_your_segment), you are asking for an already existing segment to be placed in your buffer. If you have previously created a segment named my_seg, you ask for qedx to read it this way:

```
! qedx
! r my_seg
```

and qedx makes a copy of my_seg, obtained from the stored segment (which has been "filed" and remains intact), and puts the copy in your buffer. When you issue the read request above, you remain in edit mode, and the copy of my_seg is at your disposal. Nothing prints out, however, until you request it. Now that the text is in your buffer, you can change it--make corrections, add lines, delete lines, edit it any way you wish. When you have finished editing the text, you must issue a final write request if you want your work to be saved. When you type a "w", the contents of your buffer destroy and replace the contents of the stored segment my_seg. For this write request, it is not necessary to specify "my_seg"; you merely type "w" and qedx remembers the name from the read request above, writing the changes into that segment.

LOCATING AND PRINTING LINES

The following discussion describes the different methods of finding the lines that are to be edited--"locating lines."

Current Line And Pointer

Two important concepts are the current line and the pointer. Briefly, the last line edited or input is the current line. If you input five lines of text, then type "\f" (reverting to edit mode), the fifth line is the current line. The pointer is an imaginary one that indicates the current line. As you edit from line to line the position of the pointer changes.

Going forward in the buffer means moving toward the last line in the buffer; going backward means going toward line 1.

In the example below, the pointer is on the fifth line of the text. If you correct a mistake on the third line, the pointer moves to that line, and the third line becomes the current line.

Printing The Current Line

To print the current line, there is a special character recognized by qedx, the period (typed: .). When you use the period as an edit request, qedx prints the line you are currently working on:

```
! qx
! a
! first line
! second line
! third line
! fourth line
! fifth line
! \f
! .                          PRINT CURRENT LINE
  fifth line
```

After you read a segment into your buffer, the current line (before you begin to edit) is the last line in the buffer. This is because that line is the one that qedx worked on last (copied into the buffer).

ADDRESSES

The qedx editor is called a line-oriented editor since its requests operate on lines of text. Most requests are preceded by an address, specifying the line or lines on which to perform the requested operation.

There are three ways to address a line or set of lines:

1. By absolute line number (e.g., 1, 3, 6)

2. By relative line number (e.g., -2, +5, +10)

3. By context (e.g., /The editor/ addresses a line containing the character string "The editor") *

For example, to print the first line of the buffer, you specify 1 as the address for the print request; to print lines one through five you specify lines 1,5 (this is a compound address, explained in detail later) as follows (assume the buffer contents are as shown above):

```
! 1p                                                          |
  first line
! 1,5p                PRINTING A RANGE                       |
  first line
  second line
  third line
  fourth line
  fifth line
```

Methods of addressing are described below; see also Appendix A, which is a comprehensive summary of qedx addressing conventions.

Absolute Line Number

Each line of text in the buffer is given an implicit line number by qedx. That is, each line is invisibly numbered; as addition and deletion of lines is done, the lines are invisibly renumbered by qedx, and kept sequentially in order. The text:

```
first line
second line
third line
```

is perceived by qedx as:

```
1 first line
2 second line
3 third line
```

Therefore, if you know the line number of a line, you can address it by its absolute line number. (These line numbers are NOT printed at your terminal alongside each line.) Bear in mind that line numbers change as lines are added or deleted. If you add or delete lines and then attempt to address using "old" line numbers you may get confused since the line numbers changed when you altered the contents of the buffer.

The following example shows a line being added, and the subsequent renumbering of the lines in the buffer by qedx:

```
! 2a
! second-and-half-line        ADDING A LINE
! \f
```

The lines of text in the buffer are now perceived as:

```
1  first line
2  second line                QEDX
3  second-and-half-line       INVISIBLY
4  third line                 RENUMBERS
```

* You can also edit the text from the "bottom up" (to ensure that changes of line numbers do not affect you). Assume that your buffer contains ten lines, and you know that you need to
| edit lines number 2, 4, and 8. You can begin editing at line 8,
| even add 20 lines after it, but when you go to line 4 to edit it,
| it is still line 4 because the alterations were made to the text
* after this line.

PRINTING A SINGLE LINE

To print a single line, type the address (absolute, relative, or by context) of the line. The pointer moves to that line and prints it. This is the special case of the print request; qedx locates the line you specify and prints it even though you did not type a "p".

The following example shows the use of an absolute line number address; as the line addressed is printed, the current line pointer moves to the specified line. (The text in this example will be assumed to be in the buffer for the rest of the examples on addressing unless otherwise noted.)

Assume the buffer contains:

```
The Multics text editor, qedx,
is a line-oriented editor.
It performs editing
functions on lines,
using requests.
```

You want qedx to print the third line of the buffer:

```
! 3                                        PRINT LINE 3
It performs editing
```

The "p" request (in the case of a one-line address) is assumed; any time you type a one-line address, qedx prints the contents of the line addressed unless you request otherwise.

PRINTING MORE THAN ONE LINE

You may also specify a range of addresses on which to perform a request. This range consists of the address of the first line, a comma, and the address of the last line. The request is then performed on the first line through the last line of the range; i.e., including the lines between.

For example, to print the first, second, and third lines from the example above, type the address range 1,3 followed by the print request:

```
! 1,3p                                PRINT LINES 1 THROUGH 3
The Multics text editor, qedx,
is a line-oriented editor.
It performs editing
```

Notice that the print request (p) is included in the example above. An entire address range is not printed by default if no print request is specified. So, to print a block of text, use a range followed by "p"; to print only one line, use the address designating a single line.

As shown in the example below, to print a block of lines you must specify the "p", or qedx ignores the first part of the range and performs the request on the second part (after the comma):

```
! 1,3
  It performs editing
```

Relative Line Number

Addressing by relative line number specifies a line by describing its position in the buffer in relation to the current line. In other words, you may not know the absolute line number of the text in the buffer, but want to address a line that you know to be two lines above the current line, so you use its relative address (-2). You can also use a relative address to specify an address range.

CURRENT LINE

As previously mentioned, the period (.) is a special address that means "the current line." When you type this address without specifying a request, it prints the current line (see the examples below).

PRINT CURRENT LINE NUMBER (=)

In qedx, the equal sign (=) is a request that prints the line number (but not the contents) of the current line:

```
!  /performs/          PRINT LINE NUMBER OF THE NEXT LINE
3                       CONTAINING "performs"
!  .2                  PRINT THE LINE THAT IS TWO LINES
using requests.         DOWN FROM IT
!  =                   PRINT LINE NUMBER OF CURRENT LINE
5
```

LAST LINE ($)

Another special address is the dollar sign ($). When used to address a line in the buffer (with no accompanying request) it means go to the last line in the buffer (you needn't know its line number) and print the contents of that line:

```
! $
  using requests.
```

In the following example, the first line you type is the period to print your current line. Then you use a relative line number address to request qedx to print the line located two lines before your current line. Next, you type the relative line number for the following third line. After you type each request, the new current line is printed:

```
! .                                   PRINT CURRENT LINE
  It performs editing
! -2                                  BACK TWO
  The Multics text editor, qedx,
! +3                                  FORWARD THREE
  functions on lines,
```

When addressing by relative line number, a minus sign (-) followed by any number tells qedx to move backward in the buffer that number of lines; a plus sign (+) followed by a number means move forward in the buffer that number of lines. Relative addresses assume the current line to be the starting point. (Appendix C contains more details about different forms of address.)

When a relative line number address is used, the current line is not counted as one of the lines in that address. In the above example, when -2 is typed, the pointer moves to the second line above the current line.

You can also specify "+3" by typing ".3"; many users prefer this method since they don't have to use the shift key for the plus sign.

```
! .3p                          PRINT THE THIRD LINE BELOW
  It performs editing               THE CURRENT LINE
```

The next example prints a range of lines with relative line numbers:

```
! .
  is a line-oriented editor.
! -1,+1p                             PRINTS PRECEDING,
  The Multics text editor, qedx,       CURRENT, AND
  is a line-oriented editor.           FOLLOWING LINE
  It performs editing
```

PRINTING THE ENTIRE CONTENTS OF YOUR BUFFER

To print the first line through the last line of the buffer, type:

```
! 1,$p
The Multics text editor, qedx,
is a line-oriented editor.
It performs editing
functions on lines,
using requests.
```

NOTE: If you have issued this request, and then decide that you do not want to see the whole buffer, you can interrupt printing with the QUIT key and then get back to qedx with the program_interrupt (pi) command (see below).

INTERRUPTING A LENGTHY PRINT REQUEST

When asking for the contents of the buffer to be printed (e.g., first through last lines), occasionally you decide that you don't want to wait for all the lines to be printed. You can interrupt processing of your request by pressing the QUIT key (sometimes labeled ATTN, BRK, or INTERRUPT). This action takes you out of qedx and back to command level; Multics prints QUIT and a ready message containing the word "level" and additional information that you needn't be concerned with. Type the Multics command "program_interrupt" (pi) to reenter qedx. Your print request has been aborted; now you are in edit mode and can issue another request. (At this point, the current line is set to the last line addressed in your print request.)

Assume the contents of the buffer to be the same as shown above.

```
! 1,$p
The Multics text editor, qedx,
is ori
QUIT                                  USER PRESSED THE INTERRUPT
r 651 0.291 0.218 24 level 2, 14        KEY HERE
! pi
! .
using requests
```

Context Addressing (Regular Expressions)

To use this form of addressing, specify the desired line by typing a set of characters, words, etc., (a character string) within a pair of slashes (/). At the line after the current line, qedx begins a forward search for the specified string. If the string is not found after searching through the last line in the buffer, qedx returns to the top of the buffer (line 1) and searches down to the current line. If the string is still not found, qedx prints a message indicating so.

The character string (regular expression) is searched for exactly as you type it. If the search fails, make sure you have typed it exactly, character-for-character as it appears in your text. Notice that you can use a context address to specify an address range. Also shown in the example below is a compound address; in this case a regular expression is combined with a relative line number to address the line immediately preceding the line that contains "lines". Compound addresses can contain any combination of different forms of address (for more information, see "Compound Addresses" below).

```
!  /sixth/                              SEARCH FOR A STRING
   Search failed.
!  /editing/
   It performs editing
!  /editor, qedx,/
   The Multics text editor, qedx,
!  /editing/,/using/p                   ADDRESS RANGE
   It performs editing
   functions on lines,
   using requests.
!  /lines/-1                            COMPOUND ADDRESS
   It performs editing


!  1,/It/p                              PRINT LINE 1 THROUGH LINE
   The Multics text editor, qedx,         CONTAINING "It"
   is a line-oriented editor.
   It performs editing
!  +1,/us/ p                            PRINT NEXT LINE THROUGH
   functions on lines,                    LINE CONTAINING "us"
   using requests.
!  /Multics/,/Multics/+1p               PRINT LINE CONTAINING
   The Multics text editor, qedx,         "Multics" THROUGH THE
   is a line-oriented editor.             LINE FOLLOWING IT
```

In its simplest form, a regular expression is one or more characters delimited by the right slant character (/). For example, all of the following are valid regular expressions:

```
/one/
/one or/
/For/
/F/
/characters,/
/ters,/
/:/
```

A regular expression search begins to search for that regular expression at the line following the current line, goes to the bottom of the buffer, then wraps to the top and goes down to the current line. After starting the search it stops at the first occurrence of the regular expression:

```
!  .
   using requests.
!  /functions/              SEARCH STARTS FROM HERE,
   functions on lines.        GOES TO TOP AND DOWN TO
                              LINE CONTAINING "functions"
```

Notice that spaces and punctuation characters can be part of a regular expression. However, certain characters have special meanings when used in regular expressions.

SPECIAL CHARACTERS IN REGULAR EXPRESSIONS

You can use one or more of the special characters in a regular expression to uniquely identify a particular character string with a minimum of typing. These special characters have no distinctive meaning to qedx when you are in input mode; they are only recognized in regular expressions while in edit mode. The characters with special meanings are:

- `/` delimits a regular expression (commonly referred to as "slash")
- `*` means any number (including none) of the preceding character (asterisk, commonly referred to in Multics as "star").
- `.` matches any single character. Period is interpreted differently from the address ".", which refers to the current line.

^ as the first character in a regular expression, means the (imaginary) "character" preceding the first character on a line (circumflex, commonly referred to in Multics as "not symbol"). It is used to match lines beginning with a specified string.

$ as the last character in a regular expression, means the (imaginary) "character" following the last character on a line. Dollar sign is interpreted differently form the address "$", which refers to the last line.

Below are some examples that show how these characters can be used. Create some text in your buffer and try a few of these combinations.

/a/	Matches the letter "a" anywhere on a line.
/abc/	Matches the string "abc" anywhere on a line.
/ab*c/	Matches "ac", "abc", "abbc", "abbbc", etc. anywhere on a line.
/in..to/	Matches "in" followed by any two characters followed by "to" anywhere on a line.
/in.*to/	Matches "in" followed by any number of any characters (including none) followed by "to" anywhere on a line.
/^abc/	Matches a line beginning with "abc".
/abc$/	Matches a line ending with "abc".
/^abc.*def$/	Matches a line beginning with "abc" and ending with "def" and having anything (including nothing) in between.
/^.*$/	Matches any line.
/^$/	Matches an empty line (a line containing only a newline character).

Following are some examples that show regular expression searches with some special characters (searching the text established above).

The example below shows a search for the characters "ed," followed by any number of characters, followed by the characters "or":

```
! /ed.*or/                          FIND THE STRING STARTING "ed"
The Multics text editor, qedx,          AND ENDING "or"
```

The regular expression below defines a line that begins with an "f":

```
! /^f/                        FIND THE LINE BEGINNING WITH "f"
functions on lines,
```

The next one searches for a line beginning with "u":

```
! /^u/
using requests.
```

The following regular expression searches for the next line whose last character is a comma:

```
! /,$/
The Multics text editor, qedx,
```

The next one searches for a line containing an "M", followed by any number of any characters, followed by a comma:

```
! /M.*,/
The Multics text editor, qedx,
```

It is possible to use these characters without their special meaning; simply type the "\c" escape sequence before the special character. On terminals with no backslash (\) character, type ¢c instead.

For example, to match the string "/*" appearing anywhere on a line, the regular expression is:

```
/\c/\c*/
```

The first slash is the delimiter and the backslash c following it tells qedx to interpret the following slash literally (as part of the regular expression--not a special character). Next, another backslash c indicates the following character is to be part of the regular expression, then the asterisk, then the closing slash (delimiter). If you had typed the string "//*/" in an attempt to search for "/*", qedx would have printed back an error message.

One more special character is the ampersand (&). It only has a special meaning in a substitute request (see below) where it stands for the contents of the regular expression.

NULL REGULAR EXPRESSION

The qedx editor remembers the last regular expression used. You can reuse the last regular expression typed by using a null regular expression (i.e., //). This feature saves a lot of typing time during editing, especially if the regular expression is long or difficult to type. For example, the following expression matches a line containing "editor":

```
! /ed.*or/
   The Multics text editor, qedx,
```

Now, given a null regular expression, qedx searches for the next occurrence of the same string:

```
! //
```

Next, search using the dollar sign as a special character to indicate the end of the line:

```
! /,$/
   The Multics text editor, qedx,
```

To search for the next line ending with a comma, type:

```
! //
   functions on lines,
```

Compound Addresses

The three forms of addressing can be combined to form compound addresses, for example by combining a regular expression and a relative address:

```
! /qedx/+1
  is a line-oriented editor.
```

The above example locates the line containing "qedx", then moves the pointer to the next line and prints that line.

```
! .,+3p
  is a line-oriented editor.
  It performs editing
  functions on lines,
  using requests.
```

In the above example a period (current line) is part of the address range, requesting qedx to print the current line through the next three lines.

For more information on compound addresses, see Appendix C.

Default Addresses

For most qedx requests, if no address is specified, the request is assumed to apply to the address of the current line. For example, the print request with no address means print the current line.

When you type "1,$p" and print the first through the last lines in your buffer, the pointer stops at the last line, so the last line is the current line. If you then type an "a" to append to the buffer contents, the input lines you type are input after the last line, although you did not specify an address:

```
! 1,$p
  The Multics text editor, qedx,
  is a line-oriented editor.
  It performs editing
  functions on lines,
  using requests.
! a
! Now this becomes the last line
! \f
```

This assumed address is called a default address. When Multics assumes some information that you do not specify, that information is referred to as default information. The default address for the "a" request is the current line.

Common Mistakes In Addressing

Shown below are some common errors made in specifying addresses; the line numbers specified are higher than any in the buffer, or beyond the end of the buffer--nonexistent. (Assume that the buffer contains four lines and the current line is line 2.)

```
! -4
   Address out of buffer (negative address).
```

The address specified asks for lines to be printed that do not exist (they would be before line 1 in the buffer).

```
! -4,-9p
   Address wrap-around.
```

The print request above asks qedx to print lines from the bottom up, which qedx does not do. Lines are printed only from the top down.

```
! +1,+5p
   Address out of buffer (too big).
```

The request above asks qedx to print lines that would be beyond the last line in the buffer.

SUBSTITUTE REQUEST

The substitute request (typed: s/substitute this/with this/) substitutes for any number of characters on a line a new set of characters. You can issue more than one substitute request on a line. Specify what is to be changed, and what to change it to. For example, assume the line is:

```
just for now
```

and you want to change it to:

```
Just for Today
```

To do this, type:

```
! s/j/J/p
  Just for now
! s/now/Today/p
  Just for Today
```

or equivalently:

```
! s/j/J/s/now/Today/p
  Just for Today
```

In the line:

```
s/old/new/
```

the string "old" is called the search string, and the string "new" is called the substitution string.

The slashes are called delimiters, and can be any character as long as that character does not appear within either the search or substitution string (see example below). The example above shows that any number of substitute requests appear on one line. Also, new users are advised to combine substitution requests with print requests, to verify changes. Substitutions can be made on blocks of text, using address ranges as described above.

The substitute request performs the substitution for every occurrence of the specified string in the line (or lines), so be sure that the substitution you are making is correct:

```
! . The Multics text editor, qedx,
! s/x/X/p
  The Multics teXt editor, qedX,
! s/ed/ED/p
  The Multics teXt EDitor, qEDX,
```

As mentioned above, the delimiter symbol need not be a slash:

```
! s/the/yyy/p
  yyy Multics teXt EDitor, qEDX,
! szyyyzThez p
  The Multics teXt EDitor, qEDX,
```

You may get a message from qedx indicating that the attempted substitution has failed. Check to make sure you have typed the search string exactly as it appears in the text (as in the following example).

```
!  .
   The Multics teXt EDitor, qEDX,
!  s/QEDX/qedx/p
   Substitution failed.
!  s/ED/ed/s/X/x/p
   The Multics text editor, qedx,
```

Another reason that the substitution can fail is that you ask for a substitution on a line that does not contain the search string. You can try typing a "p", verifying the text on your current line.

```
!  s/performs/does/p
   Substitution failed.
!  p
   The Multics text editor, qedx,
!  +2
   It performs editing
!  s/performs/does/p
   It does editing
```

Substitution is always performed from left to right; if there is more than one match on a line, the first (leftmost) match is substituted, then the next one to the right and so on. For example, in the following example, the first character on the line is a match for the string; the substitution is performed on it, then the substitution is performed on the next match on the line, etc.; that is, the result of substitution is not rescanned for matches:

```
!  .
   aaa
!  s/a/aaa/p
   aaaaaaaaa
```

Match strings do not overlap, i.e., qedx takes the first, shortest matching string to perform the substitution on:

```
!  .
   aaa
!  s/aa/b/p
   ba
```

The Ampersand (&)

The & (ampersand) character, when used in the substitution string, "duplicates" the string that matched the regular expression (search string).

For example:

```
!  .
   It does editing
!  s/It/(&)/p                          REPLACE "It" WITH "(It)"
   (It) does editing
!  s/^.*$/&?&/p                          REPLACE THE WHOLE LINE
   (It) does editing?(It) does editing   WITH ITSELF, QUESTION
                                         MARK, AND ITSELF AGAIN
```

DELETING LINES

The delete request (typed: d) deletes a line or lines of text from the buffer.

For example, assume the buffer contains the three lines shown, and the current line is line one. To delete line three in the buffer, type:

```
   first line
   second line
   third line
!  3d
```

and the third line is deleted (you could also use "+2d" or "/third/d").

When you delete a line, the current line becomes the one immediately following the deleted line. In the example below, the first and second lines are deleted and then the current line is printed:

```
!  1,2d
!  p
   third line
```

To delete every line in the buffer, use the address range that specifies the first through the last line:

```
!  1,$d
!  p
   Buffer empty.
```

ADDING MORE TEXT

Lines of text are added to the buffer with the append request, as described in Section 2. When adding text to an existing segment, after reading the segment into the buffer, you type an "a":

```
! qx
! r text
! 1,$p
  The Multics text editor, qedx,
  It performs editing
  functions on lines
  using requests.
! a
! The requests are fully described in
! this manual.
! \f
! w
! 1,$p
  The Multics text editor, qedx,
  is a line-oriented editor.
  is a line-oriented editor.
  It performs editing
  functions on lines,
  using requests.
  The requests are fully described in
  this manual.
```

Most requests are preceded by an address, so qedx knows where to take the desired action as described earlier in this section. The example above shows that qedx assumes an address with the request (in this case, the append request) if you do not supply one.

When you type the append request as above, the lines following it are added after the current line (in this case, the last line in the buffer), even though you did not specify an address; qedx uses the default address. A read request automatically sets the current line to the last line in the buffer. The current line is the default of the append request. Therefore, an append with no address typed after you read something into the buffer appends text at the end.

By specifying an address, you designate exactly where the lines are to be added (to add them somewhere other than at the end):

```
/line-/
is a line-oriented editor.
=
2
.a
The three methods of
locating lines are called
addressing.
\f
w
2,6p
is a line-oriented editor.
The three methods of
locating lines are called
addressing.
It performs editing
```

Every line in the buffer following line 2 is renumbered after the append request, since qedx keeps track of the new lines of text being added (see the beginning of this section).

REPLACING LINES OF TEXT

Replace a line or lines of text use the change request (typed: c). The change request, also an input request, must be terminated with the "\f" (just like the append request). Locate the line or lines that you wish to replace, type "c", then the replacement text followed by "\f". The change request is like deleting the addressed lines and replacing them with new lines.

In the following example, one line of text is replaced with three new lines:

```
! 1,$p
  Input and editing operations are performed
  in a temporary workspace called a buffer.
  in a temporary workspace called a buffer.
! 3c
! When you edit an already existing segment,
! a copy of that segment
! is placed in the buffer.
! \f
! w
! 1,$p
  Input and editing operations are performed
  in a temporary workspace called a
  buffer.  When you edit an already existing
  segment, a copy of that segment is
  placed in the buffer.
```

Below is an example that shows three lines of text being replaced with one:

```
! 1,$p
  Input and editing operations are performed
  in a temporary workspace called a buffer.
  When you edit an already existing segment,
  a copy of that segment
  is placed in a buffer.
! 3,5c
! When you issue the write request (w),
! \f
! w
! 1,$p
  Input and editing operations are performed
  in a temporary workspace called a buffer.
  When you issue the write request (w),
```

The change request is very powerful and deletes text with no way to get it back. Verify that you are on the line you want to change before using the change request.

INSERTING TEXT

The insert request (typed: i) inserts text (any number of lines) into the buffer before the line you specify. This request is also terminated by "\f". Give the address of the line before the text which is to be inserted, followed by the insert request. If you want to insert text above the current line, you do not need to specify an address--the current line is the default address (see the example below).

In the following example, the insert request is shown with no address specified; qedx assumes the current line (in this case, the last line) as the address:

```
! 1,$p
  Input and editing operations are performed
  When you edit an already existing segment,
! i
! in a temporary workspace called a buffer.
! \f
! w
! 1,$p
  Input and editing workspace called a buffer.
  in a temporary workspace called a buffer.
  When you edit an already existing segment,
```

You can insert any number of lines in this manner; always end your insertions with the "\f" input terminator (just like the append request).

PRINTING, LISTING, AND DELETING SEGMENTS

After text segments have been created using the qedx write request, use Multics commands to list, print, delete and manipulate them. These commands are not part of qedx; they are issued at command level (after a ready message). (The list, print, dprint, and delete commands are fully described in the Commands.)

The directory in which you create your segments is usually your working directory and it contains a list of those segments. To see the list, use the list command:

```
r 1629 0.023 0.000 0

! list

Segments = 2, Lengths = 2.

r w     1  example
r w     1  memo

r 1629 0.159 0.350 17
```

NOTE: The "r w" is access information and "Lengths" refers to the combined lengths of the segments. These topics are beyond the scope of this manual; they are fully described in the New Users' Introduction.

The list command prints a list of every segment in the directory along with its length and other information.

The following example asks for information pertaining to a particular segment named memo. Multics responds with a list consisting only of that segment called memo:

```
! list memo

Segments = 1, Lengths = 1.

r w 1 memo

r 1629 0.025 0.000 0
```

Below is an example that shows the print command. The name of a segment is given with the command, telling the system what segment to print. Multics prints a banner that tells you the name of the segment, the date, and the time:

```
r 1645 0.123 0.930 30

! print example

example        03/26/79  1645.1 mst Mon

The Multics text editor, qedx,
is a line-oriented editor.
It performs editing
functions on lines,
using requests.

r 1645 0.032 0.260 4
```

To print segments too large to print at your terminal, use the dprint command. This command performs an action very similar to the print command--the only difference is that instead of printing the file out at your terminal, dprint prints it on a high-speed printer. An acknowledgement of your dprint command is printed on the terminal.

```
r 655 0.043 0.264 9

! dprint example
  1 request signalled, 1 already in printer queue 3
  r 656 0.527 4.970 70
```

The dprint command above prints the segment "example" on the high-speed printer; the dprint command puts your Person_id and Project_id as the heading on the printout. To specify a more complete destination, use the -destination and -header control arguments described for dprint in the Commands.

Below, the delete command is used to delete the segment named "example". Then the list command is used to again list the segments in that directory, verifying that the segment named "text" has been deleted:

```
! delete example
  r 1645 0.119 0.142 2

! list

  Segments = 1, Lengths = 1.

  r w 1 memo

  r 1645 0.034 0.006 2
```

SECTION 4

SAMPLE TERMINAL SESSION

This section describes a sample terminal session using regular expressions and special characters, gives some hints for new users, and shows several more editing examples.

SAMPLE INVOCATION

The sample invocation below shows a typical editing session with qedx.

You are creating new text, so invoke qedx and enter an input request immediately. Type in the lines, making use of the character and line delete symbols, then leave input mode (by typing "\f") and save the information in a segment named editing_text:

```
! qedx
! a
! Input and editing operations are performed
! inat## a temporary workspace called a buffer.
! When you edir#t an already existing segment,
! a copy of that segmenr
! us okac@is placed in a buffer.
! All of the changes you make are
! made on the copy,
! not the orginal segment.
! The edited version of the segment
! replaces the original only
! on your orders (issuing
! the write request
! \f
! w editing_text
```

At this point you have a new segment named editing_text in your working directory; the text you have created is still in your buffer. If you exit the editor now, the text is saved because you wrote it in the segment. Remember, buffers are temporary--to save text, you must write it into a segment (permanent). You are still in the qedx editor--in edit mode. To check the material you have just input, you can issue a print request:

```
!  1,$p
   Input and editing operations are performed
   in a temporary workspace called a buffer.
   When you edit an already existing segment,
   a copy of that segmenr
   is placed in a buffer.
   All of the changes you make are
   made on the copy,
   not the orginal segment.
   The edited version of the segment
   replaces the original only
   on your orders (issuing
   the write request
```

The "1,$" preceding the request is an address range--i.e., a way of telling qedx to make the request operate on all the lines. After looking at the input material, you see lines that need to be corrected. Correct the lines using qedx edit requests and overwrite the segment by issuing a "w" (write) request again.

```
!  /copy/
   a copy of that segmenr
!  s/menr/ment/p
   a copy of that segment
!  /not the/
   not the orginal segment.
!  s/org/orig/p
   not the original segment.
!  +4
   the write request
!  s/st/st)./p
   the write request).
!  w
!  q
   r 1026 0.608 5.510 261
```

Notice how combined "s" and "p" requests cause qedx to print out the edited line for verification. You can have as many separate editing requests on the same line as you want (except for "r", "w" and "q"). Also notice the editing request line "+4" to tell the editor to go four lines ahead of the current line (addressing by relative line number). This technique is highly recommended. Addressing is an important concept in a line-oriented editor like qedx; it enables you to tell qedx what lines the editing request should work on. And finally, notice the second "w" (write) request; it is not necessary to give a path argument with the second, and any succeeding, "w" requests as long as you want the name of the segment to remain the same.

Now that you are familiar with qedx, you should check the following examples. You can see that there are several different ways to edit the same material. As your familiarity with qedx increases, you tend to type less to accomplish the same tasks. The examples that follow illustrate this "type less" approach.

For the first example, consider the editing_text segment given in the sample invocation at the beginning of this section. After the input request, the buffer contained:

```
Input and editing operations are performed
in a temporary workspace called a buffer.
When you edit an already existing segment,
a copy of that segmenr
is placed in a buffer.
All of the changes you make are
made on the copy,
not the orginal segment.
The edited version of the segment
replaces the original only
on your orders (issuing
the write request
```

The three errors in the text were corrected with these lines:

```
! /copy/
  a copy of that segmenr
! s/menr/ment/p
  a copy of that segment
! /not the/
  not the orginal segment.
! s/org/orig/p
  not the original segment.
! +4
  the write request
! s/st/st)./p
  the write request).
```

Another way to accomplish the same corrections is:

```
!  4s/menr/ment/
!  /orgi/s//origi/
!  /request$/s//&)./
```

or:

```
!  4s/.$/t/+3s/org/orig//st$/s//&)./
```

As another example, assume the buffer contains the following lines:

```
Mondays child is far of face;
Tuesday's child is full of grace;
Wednesday's child is loving and givng;
Thursday's child works gard fir uts kuvubgl
```

These lines can be corrected in any of the following ways (with new users more apt to use a method similar to the first one):

```
!  1s/days/day's/s/far/fair/p
   Monday's child is fair of face;
!  /givng/s//giving/p
   Wednesday's child is loving and giving;
!  /work/s/works.*$/works hard for its living;/p
   Thursday's child works hard for its living;
```

Two other ways to correct the lines are:

```
!  1s/days/day's/ s/far/fair/
!  /^W/s/vng/ving/
!  +1s/works.*$/works hard for its living;/
```

and:

```
!  1s/y/y'/s/r/ir/
!  /vng/s//ving/
!  $s/works.*$/works hard for its living;/
```

EDITING EXAMPLE WITH BOTH INPUT AND EDIT REQUESTS

The most typical editing session is one where you go from edit mode, to input, and back to edit, in the process of changing, inserting and adding lines, in addition to merely fixing errors on lines. Below is an example of such a session; the most important thing to remember is to type "\f" each time you go from input to edit, and to frequently write your corrections and additions (w). Comments describing each action taken are to the right of the example.

```
! qx                                        ENTER EDITOR (EDIT MODE)
! r example                                 READ SEGMENT INTO BUFFER
! 1,$p                                      PRINT ALL LINES
  Input and editing operatons are performed
  in a temporary workspace callec by buffer.
  When you edit an already existing segment,
  is placed in a buffer.
  All of the changes you make are
  made on the copy.
  The edited version of the segment
  write request
  the write request).
! 1s/tons/tions/p                           FIX TYPO
  Input and editing operations are performed
! +1s/lec/led/p                             FIX TYPO
  in a temporary workspace called by buffer.
! s/by/a/p                                  SUBSTITUTE WORD
  in a temporary workspace called a buffer.
! /is placed/i                              INSERT LINE (INTO INPUT)
! a copy of that segment
! \f                                        LEAVE INPUT (INTO EDIT)
! w                                         SAVE CORRECTIONS
! +2                                        GO DOWN 3 LINES AND PRINT
  All of the changes you make are
  made on the copy,
! a                                         ADD A LINE (INTO INPUT)
! not the original segment.
! \f                                        LEAVE INPUT (INTO EDIT)
! w                                         SAVE ADDITION
! +1                                        GO DOWN 1 LINE AND PRINT
  The editited version of the segment
! s/iti/i/ p                                FIX TYPO
  The edited version of the segment
!  +1                                       GO DOWN 1 LINE AND PRINT
  write request
! c                                         CHANGE (REPLACE 1 LINE
! replaces the original only                WITH 2 LINES)
! on your orders (issuing
! \f                                        LEAVE INPUT (INTO EDIT)
! w                                         SAVE ADDITION
! 1,$p                                      PRINT ALL LINES
```

```
   Input and editing operations are performed
   in a temporary workspace called a buffer.
   When you edit an already existing segment,
   a copy of that segment
   is placed in a buffer.
   All of the changes you make
   are made on the copy,
   not the original segment.
   The edited version of the segment
   replaces the original only
   on your orders (issuing
   the write request).
!  q                                  LEAVE THE EDITOR
r 658 0.092 0.030 5                   AT COMMAND LEVEL
```

HELPFUL HINTS FOR NEW qedx USERS

The following list offers suggestions for users who are just beginning to work with the qedx editor.

1. Get in the habit of issuing "p" (print) requests often, to verify changes.

2. Remember the escape sequence to terminate input is "\f" (on terminals with no backslash, use ¢f instead).

   a. After issuing an input request (e.g., a for append), all lines until "\f" are considered input, including intended "w" and "q" requests. Without the "\f" sequence, any lines you type and intend as editor requests are simply more text, which you must locate and delete later.

   b. Often, users unknowingly put qedx in input mode by mistyping an editing request. If qedx does not respond to editing requests (e.g., you type "p" and nothing happens), chances are very good that qedx is in input mode. Type the "\f" sequence, print the current line and preceding lines to see if they need to be deleted, delete them if necessary and then continue editing.

3. If you have a lot of typing or editing to do, it is wise to issue the "w" request often to ensure that all the work up to that time is permanently recorded. If you are in edit mode, type "w". If you are doing a lot of input, use this sequence often:

```
! \f      (go to edit)
! w       (write)
! a       (go back to input)
```

If some problem should occur (with the system, the telephone line, or the terminal), you lose only the work done since the last "w" request. Also, if you work and then type a "q", forgetting to type a "w" first, all your work done since the last "w" is lost.

4. The qedx editor accepts more than one editing request on a single line. However, the following requests must be terminated by a newline character; therefore each one must be on a line by itself or at the end of a line containing other requests.

```
r read
w write
q quit
```

5. The qedx editor makes all changes on the buffer contents, not on the stored segment. Only when you issue a "w" (write) request does the editor overwrite the original segment with the edited buffer contents.

6. If you attempt a substitution and get back a message indicating that the substitution "failed", check to make sure that you are on the right line (p), then check to make sure that you typed the characters in the search string exactly as they appear in that line. Remember that "*", "^", ".", and "$" are special characters in a regular expression. To match these characters in text, type "\c" before them.

7. If you want to create new text, have entered qedx and begun typing text in, and get a message that says "not recognized as a request," you have forgotten to type the "a" to put yourself in input mode.

8. To print, list, or delete segments (as described in Section 3), you must be at command level. These are Multics commands, not qedx requests. If you are in qedx and need to use these commands, type "w," then "q" (quit from qedx), wait for a ready message, then proceed. (See also the "e" (execute) request in Section 5.)

9. If you have been editing your work by referring to absolute line numbers, remember that if you delete or add lines qedx renumbers the lines, so check to make sure what line you're on.

10. Generally, you should not issue a QUIT signal (press ATTN, BRK, INTERRUPT, etc.) while in the editor unless you are prepared to lose all of the work you have done since the last "w" (write) request.

    Occasionally, however, use of the QUIT signal is very handy. Suppose you have read a segment containing several hundred lines into the buffer and type "1,$p" by mistake. You should issue a QUIT signal, wait for the system to respond (stops printing lines on your terminal, then QUIT and a ready message are printed on the terminal), and then issue the program_interrupt command. The program_interrupt command (short name, pi) returns you to qedx where the editor waits for the next request, just as though no interruption has occurred, and the interrupted request is discarded.

    If you issue a QUIT signal accidentally while in the editor, you should wait for the system to respond (as above), and then issue the start command. The start command (short name, sr) returns you to qedx where the editor continues processing the interrupted request. However, if you are in the middle of a printout some of it may be discarded by Multics before it reaches your terminal (i.e., it will not start printing from the exact point that it stopped).

SECTION 5

ADVANCED EDIT REQUESTS

This section introduces some more advanced qedx editing requests: the extended edit requests and buffer requests. Some extended edit requests allow you to selectively print, delete, and print the line numbers of only lines that you specify; that is, you can perform these operations on a large number of lines with one request, but the operation need not be performed on every line within the text block. Another extended edit request enables you to execute Multics commands from within qedx.

The buffer requests include descriptions of how to use buffers for moving text ("cut and paste"), the meaning of qedx special escape sequences, and a more technical discussion of qedx macros.

LIST OF EXTENDED EDIT REQUESTS

execute (e)	Pass remainder of request line to the Multics command processor (i.e., escape to execute other Multics commands).
global (g)	Print, delete, or print line numbers of all addressed lines that match a specified regular expression.
exclude (v)	Print, delete, or print line numbers of all addressed lines that do not match a given regular expression.

LIST OF BUFFER REQUESTS

buffer (b)	Switch to a specified buffer (i.e., do subsequent editor operations to the specified buffer).
move (m)	Move specified lines into the specified buffer.
status (x)	Print a summary of the status of all buffers currently in use, and indicate in which buffer you are currently working.
nothing (n)	Do nothing (used to move the pointer to a line but perform no other action).
comment (")	Ignore the remainder of the request line.

EXTENDED EDIT REQUESTS

The editor requests discussed up to this point comprise a basic subset sufficient for most applications. The following requests offer you more advanced and, in general, more powerful capabilities.

How To Execute A Multics Command From Inside qedx

e (execute) request

The execute (e) request allows you to "escape" from the qedx environment temporarily to type Multics commands as if you were at command level. You may want to do this just as a matter of convenience, or you may be using several buffers (described later in this section) that would all be lost if you ended the session.

It is often necessary to type a Multics command to obtain some information that is pertinent to the work you are doing in qedx. You can use the execute request to enlist the aid of
* Multics commands without having to exit from qedx.

The "e" signals qedx that the rest of the line is to be sent directly to the command processor, which interprets the command line you type and executes the line. When the command is finished executing, you are still in qedx, and your current line remains the same as it was before you issued the execute request.

```
!  w                          SAVE ALL CHANGES
!  e dprint file1             EXECUTE DPRINT COMMAND FROM QEDX
   1 request signalled,...    MULTICS ACKNOWLEDGEMENT

!  1,$d                       CLEAR THE BUFFER
!  r file2                    READ ANOTHER SEGMENT
   <Editing>
        .
        .
        .
!  w                          SAVE CHANGES
!  e dprint file2             DPRINT COMMAND
   1 request signalled,...    ACKNOWLEDGEMENT
!  1,$d                       CLEAR BUFFER
!  r file3                    READ ANOTHER SEGMENT
```

One example where you might use this request is if you are editing in qedx, receive a message from another user, and wish to reply without interrupting your work.

Assume you have text in the buffer and are editing when you receive a message:

```
!  .
   as mentioned earlier
   From Smith.Sales 10/18/82 1200.6 mst Mon: where is the
      draft memo?
!  e sm Smith Sales check your mailbox
!  .
   as mentioned earlier
```

The only command that you cannot execute in this manner is qedx; if you try to execute qedx, a warning message is printed stating that the previous qedx session will be lost if you continue (see Appendix E, "Error Messages").

The most common mistake made when using this request is forgetting to precede your command with the "e". When this happens and you are in edit mode, qedx attempts to interpret the command as an edit request. When qedx fails to recognize the initial character as a request, it prints a message indicating so (see Appendix E). If you are in input mode and attempt to

execute a Multics command in this manner, the line you type is entered as input into the buffer.

When using this request, before typing the carriage return at the end of the line, check to make sure that you have preceded
* the command line with an "e" to avoid a qedx error.

Global Printing, Deleting, And Printing Line Numbers

g (global) request

The global request (g) prints, deletes, or prints the line numbers of every line in your buffer that contains the regular expression that you specify. The global request can be preceded by a line number or range of lines as the address--if you do not specify an address, "1,$" is assumed. Immediately following the "g" you specify one of three actions to be taken: "p" (print), "=" (print line number), or "d" (delete). The last part of a global request is the search string.

For example, to print every line in the buffer that contains the word "fiscal", type:

```
! gp/fiscal/
```

Since no address is specified in the above example, the range assumed is "1,$".

Suppose you have a segment of financial data and there are total lines for each division. These lines contain the division name, the word "TOTAL", and the dollar amount. You want to print just these total lines:

```
! qx                      ENTER QEDX
! r finance_data          READ THE DESIRED SEGMENT
! gp/TOTAL/               PRINT THE LINES CONTAINING TOTAL
      .
      .
(lines print here)
      .
      .
! q                       QUIT THE EDIT SESSION
```

To globally print line numbers, use "g=". This request prints the line numbers (not the content) of those lines in the address range that contain the string specified in the regular expression. If no address is specified, qedx assumes "1,$".

Suppose you want to find all references to "Figure 1" in your text, but beyond that you want to print the text that is three lines before and after the reference. A global print would print each line with "Figure 1" in it, but you would still not easily accomplish your desired goal. Below is a solution using the "g=" request in combination with the regular print request (assume there are only two references to Figure 1):

```
!  qx                ENTER QEDX
!  r text            READ THE TEXT
!  g=/Figure 1/      PRINT LINE NUMBERS OF INTEREST
   132               ASSUME THESE TWO ARE
   184                 THE RESULTS
!  129,135p          NOW PRINT THE TEXT ON
      .                EITHER SIDE OF
      .                THE IDENTIFIED LINES
!  181,187p
      .
      .
!  q                 QUIT THE EDIT SESSION
```

To delete the lines within an address range that contain a certain regular expression, use "gd" (global delete). This way only those certain lines within the range, as opposed to every line in the range (with the regular delete request) are deleted.

For example, suppose you have a data segment with entries from two fiscal years. Assume that these entries are intermingled, and that the last four characters of each entry are "FY78" or "FY79". You want to have a segment that contains only the "FY79" lines. The usual delete request won't work, except by searching for each "FY78" line and deleting each one on a case-by-case basis. The solution is:

```
!  qx                      ENTER QEDX
!  r composite_data        READ THE SEGMENT
!  gd/FY78$/               DELETE LINES ENDING WITH "FY78"
!  w fy79_Data             WRITE WHAT'S LEFT TO NEW SEGMENT
!  q                       QUIT THE EDIT SESSION
```

v (exclude) request

There are three requests comparable to the globals described above. These are the "v" (exclude) requests. They also delete, print, and print line numbers; the key difference is that the exclude requests select all the lines that do not contain strings that match the string specified in the regular expression. The format for the exclude requests is the same as for the global requests, with "v" instead of "g".

The form of address used in exclude requests is also the same as for the global requests--if no address is specified, "1,$" is assumed.

The exclude print request (vp) prints all lines in the address range that do not contain a string matching the regular expression; this request is complementary to the global print. You use it to see all lines except those which contain a particular string.

Consider a segment with a variety of programming language entries, each having a descriptive name such as fortran, pl1, or cobol; you are interested in printing all entries except for the ones related to cobol. The sequence of requests is:

```
!  qx                    ENTER QEDX
!  r languages           READ THE SEGMENT
!  vp/cobol/             PRINT ALL LINES EXCEPT
                           THOSE WITH "cobol"
        .
        .
   (lines print here)
        .
        .
!  q                     QUIT THE EDIT SESSION
```

The exclude print line numbers request (v=) prints the line numbers of those lines in the address range that do not contain the string specified.

For this example, consider a segment in which you need to locate all entries that contain "FY79" somewhere in the line. Since the position of the string in the lines may vary, a visual scan would not be a satisfactory way to verify its presence. Use the "v=" request first to see if any lines are missing the "FY79" string, then correct those lines by line number. Here is the complete sequence:

```
!  qx
!  r fy79_data              READ THE SEGMENT
!  v=/FY79/                 PRINT THE LINE NUMBERS
                              OF ALL ENTRIES WITHOUT "FY79"
        .
        .
   (line numbers here)
        .
        .
!  q
```

The exclude delete request (vd) deletes all lines in the address range except those that contain a string matching the regular expression.

In this example, consider a segment with single line entries consisting of geologic sample analysis. Each line contains the name of some element or mineral along with location information. You wish to delete all lines except those which contain the word "COAL", saving the coal information in a new segment:

```
! qx
! r sample_data
! vd/COAL/                DELETE LINES EXCEPT THOSE WITH "COAL"
! w coal_data             WRITE WHAT'S LEFT INTO NEW SEGMENT
! q
```

BUFFER REQUESTS

The discussion up to this point has assumed the existence of only a single buffer. Actually, qedx supports a virtually unlimited number of buffers. One buffer at a time can be designated as the active or current buffer; any other buffers at this time are referred to as auxiliary buffers. All of the editor requests described so far operate within the current buffer. Descriptions below show how to create buffers, switch the active status from one buffer to another, obtain some information about all the buffers, and move text from one buffer to another. Finally, you see how to enter qedx commands into a buffer and execute the commands as a group.

Each buffer is given a name of 1 to 16 characters. When the editor is invok d, a single buffer (buffer 0) is created by the editor and made the current buffer.

There are two other buffers that qedx may create on its own, depending on how it is called (see "Initialization of Macros" below) these are named "exec" and "args". Except for these three names, naming of additional buffers is completely up to you. Additional buffers can be created merely by referencing a previously undefined buffer name; in other words, you create a buffer by using it. Each buffer is implemented as a separate segment and, thus, is capable of holding any segment.

Buffer names of more than one character must be enclosed in parentheses; for example, the buffer name Fred is typed as "(Fred)". A buffer name consisting of a single character can be typed with or without the enclosing parentheses (e.g., "y" is the same as "(y)").

WARNING: Buffers exist for only the current invocation of qedx. That is, if you create several auxiliary buffers, issue the quit request and then invoke qedx again, the auxiliary buffers you created earlier are gone.

The buffer requests allow you to create auxiliary buffers, move text from one buffer to another, and check on the status of all buffers currently in use. In addition, these requests provide an interpretive programming capability when used in conjunction with certain escape sequences (see "Special Escape Sequences" below). It is important to remember when manipulating buffers that any information in any buffer that you wish to save must be written into a segment. There are three main buffer requests: the change buffer request, the move request, and the buffer status request.

Creating and Changing Buffers

b(Name) request

The change buffer request does not use an address; it is typed "b(Name)". It means that the buffer designated by (Name) becomes the current buffer--"go to buffer (Name)". If the buffer does not already exist, it is created and is empty. If it did already exist, it becomes the current buffer and its contents remain unchanged. (See the example below.)

Moving Blocks Of Lines (Cut And Paste)

m(Name) request

The move request uses an address range to designate a block of lines that you move to an auxiliary buffer--in this case, buffer (Name). This request can be preceded by a one-line address, or an address range (specifying a block of lines to be moved). The default address for this request is the current line. The lines that you move are deleted from the current buffer, and they replace the entire contents of buffer "(Name)". If buffer "(Name)" does not exist, it is created by this request. It is important to note that buffer "(Name)" does not become the current buffer. If you now want to edit the lines that you have moved, you must change buffers with the "b" request. (See the example below.)

How To Check The Status Of Your Buffers

x (status) request

You often need to check the status of your buffers as you create new buffers and manipulate buffer contents. The buffer status request (x) lists the buffers you have created, the number of lines they contain, and tells which buffer you are currently in (indicated by ->). If the contents of a buffer were read from a single segment on permanent storage, the default pathname is printed. An example of the result of an "x" request is:

```
    42     (0) >udd>Multics>JDoe>memo.compose
    16->   (temp)
```

This output indicates that there are 42 lines in buffer "0" (the one created by qedx) and its contents came from the segment called memo.compose. There is also another buffer named "temp" with 16 lines in it; "temp" is the current buffer. It cannot be determined where the contents of "temp" came from.

The example below is a "sneak preview" of "cutting and pasting" with Multics. Later, review this example some more for a better understanding of the buffer requests.

Assume you are in edit mode and want to "cut and paste" your text by repositioning some groups of lines. First, move the lines into another buffer, and then put them back into the original buffer in the new position:

```
!  1,$p
   qedx supports a
   virtually unlimited
   number of buffers.
   Up to now, all
   work we discussed
   has been done in
   one buffer (buffer 0).
!  1,3mA                  MOVE FIRST THREE LINES INTO BUFFER A
!  bA                     GO TO BUFFER A (CHANGE BUFFER REQUEST)
!  1,$p
   qedx supports a
   virtually unlimited
   number of buffers.
!  b0                      GO BACK TO ORIGINAL BUFFER
!  $                       GO TO LAST LINE
   one buffer (buffer 0).
!  a                       APPEND THE CONTENTS OF BUFFER A
!  \bA\f                   (SEE BELOW) AND RETURN TO EDIT MODE.
!  w                       WRITE THE NEW CHANGES
!  1,$p                    PRINT THE WHOLE BUFFER
```

```
Up to now, all
work, we discussed
has been done in
one buffer (buffer 0).
qedx supports a
virtually unlimited
number of buffers.
```

The example shows how to move lines one through three into a buffer named A (the move request creates buffer A at the same time); the lines are deleted from buffer 0 at the time they are moved. Then go to buffer A and print its contents, verifying that the lines have been moved. Then go back to buffer 0 and append the contents of buffer A after the last line, end by printing every line in the buffer. To make the change permanent, issue a write request.

The "\b" special escape sequence is described in detail below.

Repositioning The Pointer

n (nothing) request

The "n" request resets the position of the pointer to the specified address and performs no other action. This request is useful when executing a series of requests (a repeated editor sequence) where an action is performed at predetermined intervals within the sequence (see "Repeated Editor Sequences" later in this section).

Annotating (Comment) Macros

" (comment) request

The comment request annotates any type of material contained in a segment; precede your comments with quotation marks and they are ignored by qedx when executing the requests (see the examples later in this section).

SPECIAL ESCAPE SEQUENCES

The input to qedx can be viewed as a flow of characters. Depending on the context, some of these characters are interpreted as editor requests and others are interpreted as literal text (input). The following escape sequences are recognized by the editor, in either context, as directives to alter the input character flow in some fashion.

\b(X) If this sequence is used while in input mode, the contents of buffer X are considered to be literal text and are placed in the current buffer at the place that you type the "\b(X)". However, if another "\b" escape sequence is encountered while accepting input from buffer X, the newly encountered escape sequence is also replaced by the contents of the named buffer. The editor allows the nested replacement of "\b" escape sequences by the contents of named buffers to a depth of 500 nested "\b" escape sequences.

If this sequence is used while in edit mode (see "Initialization of Macros"), qedx attempts to interpret and execute the contents of X as if they were edit requests. The requests are interpreted by qedx as if they had been typed in by you, the only difference is that the request sequence is predetermined. The requests are executed sequentially and the rule for nested "\b" is the same as above.

The buffer to which the input is redirected can contain editor requests, literal text or both. If the editor is executing a request obtained from a buffer (rather than from the terminal) and the request specifies a regular expression search for which no match is found, the usual error comment is suppressed and the remaining contents of the buffer are skipped.

Reading the contents of buffer X does not change the contents of the buffer; the contents of buffer X remain constant. Thus, qedx can read input from buffer X many times (see the discussion on "Repeated Editor Sequences" below).

\r — This escape sequence temporarily redirects the input to read a single line from your terminal and is normally used when executing editor requests contained in a buffer (a macro). The "\r" is removed from the input stream and replaced with the next complete line entered from your terminal, including the newline. In the line that replaces the "\r" sequence, additional "\r" or "\b" escape sequences have no effect.

\c — This escape sequence signals qedx to ignore the special meaning of the immediately following character, which can be:

1. Any of the special characters used in a regular expression search string ("/," "&," "*," "$," "^," and "."). That character is then interpreted literally as part of the regular expression, or:

2. Any of the escape sequences used by qedx ("\f," "\b," "\r" and "\c"). These escape sequences can then be input as literal text.

\ — If you want to enter the erase or kill symbols ("#" or "@") as literal text, precede them with a backslash.

USE OF BUFFERS FOR MOVING TEXT

Perhaps the most common use of buffers in qedx is for moving text from one part of a segment to another. A typical pattern is to place the text to be moved into an auxiliary buffer with a move request. For example, the request:

```
! 1,5m(temp)
```

moves lines 1 through 5 of the current buffer into the auxiliary buffer temp and deletes them from the current buffer. Once the lines have been moved to an auxiliary buffer, they can be used as literal text in conjunction with an input request. For example, to insert the lines in buffer temp immediately before the last line in the current buffer, the following sequence might be used:

```
! $i
! \b(temp)\f
```

In this case, the pointer goes to the last line in the buffer, qedx enters input mode (the insert request), and the literal text in buffer "temp" is treated as input to the editor and inserted before the addressed line. Notice that the "\f" immediately follows the "\b" escape sequence. If the "\f" is put on a new line, a blank line will follow the last line of buffer "temp" text that was just inserted in the current buffer. The blank line is caused by the two successive newline characters: one is the last character in buffer "temp" and the other is inserted between the "\b" and the "\f" escape sequences.

For example, assume that buffer 0 contains the following text:

```
The Multics text editor, qedx,
is a line-oriented editor.
It performs editing
functions on lines,
using requests.
```

If you type:

```
! 1,5m(temp)
```

the result is:

```
Buffer 0 contents       Buffer (temp) contents

<buffer empty>          The Multics text editor, qedx,
<Yo are still           is a line-oriented editor.
in this buffer.>        It performs editing
                        using requests.
```

Suppose that you want to append the contents of "b(temp)" to the empty "b0". Type:

```
! a \b(temp)\f
```

the result is:

```
Buffer 0 contents                   Buffer (temp) contents

The Multics text editor, qedx,      The Multics text editor, qedx,
is a line-oriented editor.          is a line-oriented editor.
It performs editing                 It performs editing
functions on lines,                 functions on lines,
using requests.                     using requests.
```

The move request actually moves the lines from one buffer to another (i.e., they are deleted from the original buffer), whereas the "\b" escape sequence copies buffer contents into another buffer (i.e., the text remains in both places).

Assume that you have a segment named report in which you want to move lines 50 through 80 immediately after the line containing "SECTION 5":

```
    Request             Comments
!   qx                  "Enter the qedx editor"
!   r report            "Read the original text"
!   50,80m1             "Move lines 50 thru 80 to buffer 1"
                        "These lines are deleted from"
                        "  the current buffer and they become"
                        "  the sole contents of buffer 1"
!   /SECTION 5/a        "Locate the line with SECTION 5 and"
                        "  switch to the input mode"
                        "  with the append request"
!   \b1\f               "Transfer contents of buffer 1 to this"
                        "  point and switch back to"
                        "  the edit mode with the \f"
!   w                   "Write the revised buffer back"
                        "  to permanent storage"
!   q                   "Quit the edit session"
```

It should also be noted that whenever one buffer is transferred to another buffer with a "\b" as above, the contents of the auxiliary buffer are not changed. This means that text which might be repeated throughout a document can be stored in a buffer and transferred as required to the current buffer. This is particularly useful in programming applications.

REPEATED EDITOR SEQUENCES

Another common use for buffers is for the definition of frequently used editing sequences. It is possible to put a series of requests in a buffer and have them executed as a group. The request lines are entered in the same way as text. Switch to a buffer, use the append request to type in the requests, and then return to edit mode. For example, if you were faced with the task of adding the same text sequence in several places in a document, you might elect to type the editing sequence into a buffer only once and then invoke the contents of the buffer as many times as necessary. In the example given below, buffer NEW contains the necessary editor requests and literal text to append four lines of text at any point in the current buffer and then print them out (assume that the current buffer is buffer OLD):

```
Buffer OLD contents          Buffer NEW contents

Using the                    a
Multics Text Editor          The text editor,
The use of buffers           qedx, is a
enables you to               line-oriented
                             \f
                             .-4,.1p
```

Request:

```
2\b(NEW)
```

Resulting buffer contents:

```
Buffer OLD contents          Buffer NEW contents

Using the                    <Same as above>
Multics Text Editor
The text editor,
qedx, is a
line-oriented
text editor.
The use of buffers
enables you to
```

Resulting console output:

```
Multics Text Editor
The text editor,
qedx, is a
line-oriented
text editor.
The use of buffers
```

In this example, the buffer "NEW" is invoked in edit mode rather than in input mode. Therefore it is executed as a sequence of requests rather than appended or inserted in its entirety. The address (2, shown above) becomes the address of the append request and specifies the point at which the text is to be appended. The four lines of text in buffer NEW (lines 2-5 in "Buffer NEW contents" above) are appended to the current buffer; the "\f" terminates the append request; and the print request prints the line preceding the appended lines, the four appended lines, and the line following the appended lines.

Editor Macros

The set of requests contained in a buffer can be saved on permanent storage for use in a later session. The use of buffers in qedx allows you to place these editor request sequences (commonly called macros) into auxiliary buffers and use the editor as an interpretive programming language.

Suppose you have a file with pairs of entries, and you must place the word "coal" at the beginning of every even-numbered line (e.g., 2, 4, 6).

First invoke qedx and store the appropriate requests in a buffer:

```
!  qedx                 INVOKE THE EDITOR
!  b3                   SWITCH TO BUFFER 3
!  a                    CHANGE TO INPUT MODE
!  s/^/coal/            PLACE WORD "COAL" AT START OF LINE
!  .2n                  MOVE CURRENT LINE POINTER AHEAD 2 LINES
!  \c\b3                CAUSE BUFFER 3 TO BE EXECUTED REPEATEDLY
!  \f                   LEAVE INPUT MODE AND RETURN TO EDIT MODE
```

In "\c\b3", the initial "\c" forces the editor to treat the "\b" simply as text. Please note that buffer 3 now has the following contents:

```
s/^/coal/
.2n
\b3
```

Now read the segment into buffer 0 and execute requests in buffer 3:

```
!  b0                    SWITCH TO BUFFER 0
!  r coal_data           READ THE SEGMENT
!  2n                    START THE POINTER AT LINE NUMBER 2
!  \b3                   EXECUTE THE REQUESTS STORED IN BUFFER 3
```

The process terminates when it finishes the last even-numbered line in buffer 0 with the "Address out of buffer" message (see Appendix E). Now, you save the modified data.

```
!  w          WRITE coal_data BACK TO PERMANENT STORAGE
!  q          QUIT FROM THE EDITOR
```

In the example discussed below, a macro is implemented to read text from the terminal until an input terminating sequence, a line consisting only of ".". When you type the terminating sequence, the macro asks you for a name under which the text is filed and exits from the editor. The macro is implemented with three executable buffers named start, read, and test and is invoked by diverting the input to buffer start.

Example: Buffer start contents:

```
e fl "Input:"
\b(read)
```

Buffer read contents:

```
$a
\r\f
\b(test)
\b(read)
```

Buffer test contents:

```
s/^\c.$//
d
e fl "Give me a segment name:"
w \r
q
```

Explanation of start buffer:

1. The first request is an escape to the command processor to call the format_line command (short name fl) to print the message "Input:" on your terminal. (For more information on the format_line command, see the Subroutines.)

2. The second request executes the contents of buffer read.

Explanation of read buffer:

1. The first request ($a) places the editor in input mode to append text to the end of the current buffer (presumably buffer 0).

2. One line is read from your terminal (\r) and the append request is terminated (\f).

3. The contents of buffer test is executed.

4. If the substitute request in buffer test is unsuccessful (i.e., the line does not consist of a single period in the position), the contents of buffer read are executed again.

Explanation of test buffer:

1. The first line uses a substitute request to test the current line (the line just read in with "\r" by the above append request) for the input terminating sequence (a line consisting only of "."). If the regular expression in the substitute request fails to find the terminating sequence, the remaining requests in buffer test are ignored, and control is passed back to the point just after the last request executed in buffer read (i.e., the request "\b(read)" is the next request to be executed).

2. If the terminating sequence is found in step 1, the line that contained the terminating sequence is deleted.

3. Again, format_line is used to type a message to you. This time, the macro asks you for a segment name in which the input lines appended are to be stored.

4. The contents of the current buffer containing the input lines are written into a segment, the name of which is read from the terminal by the \r escape sequence.

5. The macro exits from the editor with a quit request. If the quit request were not included, qedx would expect further instructions from your terminal at this point.

Initialization of Macros

The editor provides a means through which a qedx macro can be invoked directly from command level.

A sequence of editing requests can be saved in permanent storage and executed as above by simply reading them into a buffer rather than typing them in. Another useful time to execute requests is when qedx is invoked. In this case the requests must be stored in a segment ending with the name ".qedx". Assume that each month you have a financial report created on the system and you edit this file each month to extract certain subsets of information. Also assume that the main report format and the editing functions are the same month to month. Store the edit requests in a segment which contains a complete set of requests from the initial read of the data file to the final quit. Suppose now these requests are stored in a segment named "financial_extract.qedx". Then each month the only command required by you is:

```
qx financial_extract
```

This command causes the segment to be read into a buffer named "exec". The requests stored in "exec" are then executed.

As a final example, let us extend the financial example above one step further. Assume that the original financial data segment you wish to edit ends with the name of a month, say of the form "JAN", "FEB", etc. It is possible to supply this unique designator on the same line with which you invoke qedx and make use of it in the commands stored in financial_extract.qedx. The qedx command now looks like this:

```
qx financial_extract JAN
```

and the first line of the commands you have stored in financial_extract.qedx looks like this:

```
r finance_report_\b(args)
```

One month your pre-stored requests can read finance_report_JUL and another month they can read finance_report_NOV. Notice also the use of the buffer named "args", created by qedx to hold the additional data (arguments) you might wish to supply to a pre-stored set of editing requests. Each additional item of data is stored in successive lines in buffer "args".

You can invoke qedx in the following fashion:

```
qedx path
```

The above command is equivalent to entering the editor with the simple command:

```
qedx
```

and immediately executing the following series of requests:

```
! b(exec)              GO TO BUFFER exec
! r path.qedx          READ THE SEGMENT NAMED path.qedx
! b0                   GO BACK TO BUFFER 0
! \b(exec)             EXECUTE THE REQUESTS CONTAINED IN
                         BUFFER exec
```

This request sequence reads the initial macro segment into buffer exec, changes the current buffer back to buffer 0 and executes the contents of buffer exec. This series of requests is sufficient to allow a multibuffer macro to be read into qedx from a segment and then executed.

For example, the macro given in the previous example can be initialized and run from a segment with the following contents (actual lines used to enter this segment are shown further below):

```
b(start)                    GO TO BUFFER start
$a                          APPEND ITS CONTENTS (SHOWN ABOVE)
e fl Input:
\c\b(read)
\f
b(read)$a                   GO TO BUFFER read
$a                          APPEND ITS CONTENTS (SHOWN ABOVE)
\c\r\c\f
\c\b(test)
\c\b(read)
\f
b(test)$a                   GO TO BUFFER test
s/^\c\c.$//                 APPEND ITS CONTENTS (SHOWN ABOVE)
e fl "Give me a segment name:"
w \c\r
q
\f
b0                          GO TO BUFFER 0 (MAIN)
\b(start)                   INVOKE THE MACRO
```

The contents of the buffers are first appended with append requests. Notice that all escape sequences placed into a buffer as literal text must be preceded by a "\c" escape sequence. Thus, the second line input to the read buffer is input as:

```
\c\r\c\f
```

and produces the following line:

```
\r\f
```

The example below shows the lines typed to enter this macro, creating the segment path.qedx:

```
qx
a
b(start)
$a
e fl Input:
\c\c\c\b(read)                          PRODUCES "\c\b"
\c\f                                    PRODUCES "\f"
b(read)$a
\c\c\c\r\c\c\c\f                        PRODUCES "\c\r\c\f"
\c\c\c\b(test)                          PRODUCES "\c\b(test)"
\c\c\c\b(read)                          PRODUCES "\c\b(read)"
\c\f                                    PRODUCES "\f"
b(test)$a
s/^\c\c\c\c.$//                         PRODUCES "\c\c"
d
e fl "Give me a segment name:"
w \c\c\c\r                              PRODUCES "\c\r"
q
\c\f                                    PRODUCES "\f"
b0
\c\b(start)                             PRODUCES "\b(start)"
\f
w path.qedx
q
r 951 0.902 0.788 61
```

Note carefully the use of escape sequences. If you print the contents of this buffer after entering it as shown here, you find the contents of the buffer to be the same as the contents of the segment shown above.

Finally, here is an actual invocation of the macro:

```
! qx path
  Input:
! begin to enter lines here
! continuing down
! until you're finished, then
! .
  Give me a segment name:
! storage
  r 1001 0.560 0.206 26
```

Here, you see the new segment, storage. Notice that the input terminator (.) has been deleted:

```
! print storage

  storage          06/13/79  1001.2 mst wed


  begin to enter lines here
  continuing down
  until you're finished, then


  r 1001 0.057 0.002 1
```

ADDITIONAL ARGUMENTS

The qedx editor can be invoked with more than one argument. Thus, the command line:

```
qedx read path
```

is the equivalent of:

```
qedx                    INVOKE QEDX
b(exec)                 GO TO BUFFER exec
r read.qedx             READ read.qedx
b(args)                 GO TO BUFFER args
a                       APPEND "path"
path
\f
b0                      GO TO BUFFER 0
\b(exec)                EXECUTE BUFFER exec
```

If the contents of read.qedx is:

```
r \b(args)
```

then the contents of the exec and args buffers become:

exec args

```
r \b(args) path
```

and the request \b(exec) reads the segment path into buffer 0. The editor then waits for further commands from you.

With the same contents of read.qedx, the invocation:

```
qedx read path 1,$s/x/y/ w q
```

enters into the exec and args buffers the following:

exec args

```
r \b(args)    path
              1,$s/x/y/
              w
              q
```

This causes the editor to read the segment path into buffer 0, substitute for every occurrence of x the character y, write out the segment path, and then quit and return to command level.

Notes on Macro Use

Since the name of the segment to be read in appears on the command line, this feature allows you to use abbreviations (see the description of the abbrev command in the Commands) in the names of segments to be edited.

There is no safeguard to keep the editor from changing a buffer from which it is also accepting editor requests. If this is attempted, havoc can be the result.

APPENDIX A

GLOSSARY

absolute address
: The imaginary line number qedx gives to each line of text in the buffer.

address (ADR)
: A means of telling qedx what line to locate. An address can be:
(1) absolute line number (e.g., 1),
(2) relative line number (e.g., +5),
(3) context address (matches regular expressions to strings in the line (e.g., /locate this/).

ampersand (&)
: A special character used in the replacement part of a substitute request (e.g., s/this/is replaced by this/) where it duplicates the string that is to be replaced (e.g., s/this/is replaced by &/ means the same as the substitution above).

asterisk (*)
: A duplicating character in a regular expression; means any number (including none) of the preceding character.

backslash c (\c)
: A qedx escape sequence that causes the special character that follows to be interpreted as literal.

backslash f (\f)
: A qedx escape sequence to end input; it puts you in edit mode.

buffer
A temporary workspace ("scratch pad") created by qedx; all input and editing is done in a buffer.

carriage return
A term meaning that the typing mechanism moves to the first column of the next line. See newline below.

character delete
See erase.

character string (also s ring)
A group of characters (letters, words, symbols), including spaces, used in a regular expression.

circumflex (^)
A special character that matches the beginning of a line when used in a regular expression; that is, an imaginary character that precedes the first character on a line.

command
A program which is called when you type its name; you type a command instructing Multics to perform some action for you.

command level
A term used to indicate that lines input from your terminal are interpreted by Multics as a command (i.e., the line is sent to the command processor). You are at command level when you log in, when a command completes or encounters an error, or when you stop command execution by issuing a quit signal. Command level is indicated by a ready message.

compound address
A way to locate a line (or more than one line) using some combination of absolute, relative, or context addressing (e.g., /power/+1).

context address
A means of locating a line which contains the character string that you specify in a regular expression (e.g., /FIND/ locates the next line that contains the string "FIND").

current line
The line of text in the buffer that you are currently or have just finished working on.

default
An action taken by Multics unless specifically instructed otherwise.

delete (dl)
A Multics command to delete a segment.

delimiter
A character placed in front of and at the end of a regular expression; used for context addressing and substitution.

directory
A segment that contains information describing segments and/or other directories.

directory hierarchy
The tree-structural organization of the contents of the Multics storage system.

directory (home)
The directory under which the user logs in. Usually this directory is named:

>udd>Project_id>Person_id

This directory is also known as the initial working directory.

directory (working)
The directory under which the user is doing work. Often the working directory is also the home directory. (This is always true at login time.) The user can redefine the working directory by use of the change_wdir command.

dollar sign ($)
A special character used as:
(1) an address in a qedx request (specifies the last line in the buffer), or:
(2) the imaginary character that follows the last character on a line, when used in a regular expression.

dprint (dp)
A Multics command to print the contents of a segment on a high-speed printer.

edit mode
One of two modes of operation in qedx; it allows you to:
(1) perform editing functions (e.g., substitution, deletion, printing) on data,
(2) read in the contents of existing data, and:
(3) save your editing work: If the data is new, it is placed in storage; if it is taken from storage and changed (existing), you can overwrite the stored version with the newest version.

entry
An item catalogued in a directory, such as a segment.

entryname
The name by which a segment is catalogued in a directory.

erase (#)
The symbol used on Multics to "erase" the character immediately preceding it (e.g., "tha#e" is seen by Multics as "the"). To input an erase character as text, type "\#".

error message
A message from Multics indicating that some action you called for was not carried out.

exit
See quit request.

file
A term sometimes used to mean segment (see segment).

home directory
See directory (home) above.

input mode
One of the two modes of operation in qedx; it allows you to enter new data from your terminal until you signal the end of input.

kill (@)
The symbol used on Multics to erase anything (and everything) that precedes it on a line. To input a kill character as text, type "\@".

line delete
See kill.

linefeed
See newline.

list (ls)
A Multics command to list segments.

log in
To establish a connection between your terminal and Multics.

log out
To break the connection between your terminal and Multics.

macro
An elaborate editor request sequence.

mode
In qedx, you work in one of two modes: input mode or edit mode.

Multics Programmer's Reference Manual
The primary reference manual for Multics (see the preface of this document).

newline
A term used to indicate that the typing mechanism moves to the leftmost column of the next line. The terminal type determines which key(s) you press to perform the equivalent action (e.g., RETURN, LINE SPACE, or NL).

null regular expression (//)
When a regular expression is used, // can be used repeatedly to search for the previous regular expression (see "regular expression" below).

password
A character string supplied by you and known only to you and Multics. You use it when you log in to validate your identity.

pathname
A character string that specifies a segment by its position in the storage system hierarchy (sometimes referred to as "path").

period (.)
A special character used as:
(1) an address in a qedx request (specifies the current line), or:
(2) a character in a regular expression (can be used in place of any single character in a character string).

Person_id
A unique name assigned to each user of the system. It is usually some form of your name and contains both uppercase and lowercase characters. It cannot contain blank characters. Associated with your Person_id is a single password.

pointer
A conceptual indicator which moves from line to line as you specify lines that you wish to work on; it always indicates the current line.

print (pr)
A Multics command to print a segment.

project
An arbitrary set of users grouped together for accounting and access control purposes.

project administrator
A person who specifies spending limits and other attributes for all of the users on a particular project.

Project_id
A name under which a particular project is registered on the system.

quit request
A qedx request to exit the editor; not the same as QUIT (see below).

QUIT signal
The means by which you may interrupt Multics from processing a lengthy qedx print request, a program or command lines. The QUIT signal is invoked by pressing the ATTN, INTERRUPT, BRK, or QUIT key on the terminal; Multics responds with a ready message and a new command level.

range
A two-part address given in the form 1,5 which includes the first through the fifth line.

ready message
A message that is printed each time you are at command level, indicating the system is "ready" to accept another command.

regular expression (/REGEXP/)
One or more characters delimited by a slash; it matches a string of characters and is used to search for a line and make substitutions on a line.

relative address
The location of a line in relation to the current line in the buffer.

request
Within qedx, the means by which you signal to create, add to, delete, change, and save your work.

segment
The basic unit of information within the Multics storage system. Each segment can contain data, programs, or text.

string (STRING)
See character string.

user_dir_dir (udd)
The user directory directory, which contains all project directories. Its pathname is >udd, and all user segments and directories are subordinate to it.

User_id (user identification)
Used to refer to a Person_id.Project_id pair.

APPENDIX B

qedx Command

Following is a copy of the qedx command description that appears in the Commands. This description is intended only as a summary of qedx features. For a detailed discussion of addressing, see Appendix C; for an in-depth description of each request, see Appendix D.

qedx, qx

qedx, qx

QEDX, QX

The qedx editor can be used to create and edit segments in Multics. The qedx editor cannot be called recursively. This description of the qedx editor summarizes the editing requests and addressing features provided by qedx.

Standard Usage

qedx

This invocation puts you in the editor in edit mode, where the editor waits for you to type a qedx request. To create a new segment, you might perform the following steps:

1. Invoke qedx and enter input mode by typing one of the input requests (e.g., append) as the first qedx request.

a. Enter text lines into the buffer from the terminal.

b. Leave input mode by typing the escape request sequence as the first characters of a new line.

2. Inspect the contents of the buffer and make any necessary corrections using edit or input requests.

3. Write the contents of the buffer into a new segment using the write request.

4. Exit from the editor using the quit request.

To edit an existing segment, you might perform the following steps:

1. Invoke qedx and read the segment into the buffer by giving a read request as the first qedx request.

2. Edit the contents of the buffer using edit and input requests as necessary. (The editor makes all changes on a copy of the segment, not on the original. Only when you issue a write request does the editor overwrite the original segment with the edited version.)

3. Using the write request, write the contents of the modified buffer either back into the original segment or, perhaps, into a segment of a different name.

4. Exit from the editor using the quit request.

You can create and edit any number of segments with a single invocation of the editor as long as the contents of the buffer are deleted before work is started on each new segment.

ADDRESSING

Most editing requests are preceded by an address specifying the line or lines in the buffer on which the request is to operate. Lines in the buffer can be addressed by absolute line number; relative line number, i.e., relative to the "current" line (+2 means the line that is two lines ahead of the current line, -2 means the line that is two lines behind); and context (locate the line containing /any string between these slashes/).

Current line is denoted by period (.); last line of buffer, by dollar sign ($).

REGULAR EXPRESSION

The following characters have specialized meanings when used in a regular expression. A regular expression is the character string between delimiters, such as in a substitute request, or a search string. You can reinvoke the last used regular expression by giving a null regular expression (//).

* signifies any number (or none) of the preceding character.

^ when used as the first character of a regular expression, signifies the (imaginary) character preceding the first character on a line.

$ when used as the last character of a regular expression, signifies the (imaginary) character following the last character on a line.

. matches any character on a line.

ESCAPE SEQUENCE

\f exit from input mode and terminate the input request; puts you in edit mode. It is used constantly when editing a document, and is the key to understanding the difference between input mode and edit mode. The sequence \034 is a synonym for \f.

\c suppress the meaning of the escape sequence or special character following it.

\b(X) redirect editor stream to read subsequent input from buffer X. The sequence \030 is a synonym for \b.

\r temporarily redirect the input stream to read a single line from your terminal.

NOTE: On terminals with no backslash (\), use cent-sign (¢) instead.

REQUESTS

In the list given below, editor requests are divided into four categories: input requests, basic edit requests, extended edit requests, and buffer requests. The input requests and basic edit requests are sufficient to allow a user to create and edit segments. The extended requests give the user the ability to execute Multics commands without leaving the editor and also to effect global changes. Because the requests are, in general, more difficult to use properly, they should be learned only after mastering the input and basic edit requests. The buffer requests require a knowledge of auxiliary buffers. (Since the nothing and comment requests are generally used in macros, they are included with the buffer requests.) The buffer requests, used with any of the other requests, and special escape sequences allow the user to make qedx function as an interpretive programming language through the use of macros.

The character given in parentheses is the actual character used to invoke the request in qedx and does not always bear a relation to the name of the request. The second part of each entry shows the format, default in parentheses, and brief description. For the value of ADR, see "Addressing" above; for the value of regexp, see "Regular Expression" above.

Input Requests

These requests enter input mode and must be terminated with \f.

append (a)
Enter input mode, append lines typed from the terminal <u>after</u> a specified line.

ADRa (.a) append lines after specified line.

change (c)
Enter input mode, replace the specified line or lines with lines typed from the terminal.

ADR1,ADR2c (.,.c) change existing line(s); delete and replace.

insert (i)
Enter input mode, insert lines typed from the terminal before a specified line.

ADRi (.i) insert lines before the specified line.

Basic Edit Requests

delete (d)
Delete specified line or lines from the buffer.

ADR1,ADR2d (.,.d) delete line(s).

print (p)
Print specified line or lines on the terminal; special case print needs address only.

ADR1,ADR2p (.,.p) print line(s).

print line number (=)
Print line number of specified line.

ADR= (.=) print line number.

quit (q or Q)
Exit from the editor.

*

read (r)
Read specified segment into the buffer.

ADRr path ($r path) append contents of path after specified line.

substitute (s)
Replace specific character strings in specified line or lines.

ADR1,ADR2s/regexp/string/ (.,.s/regexp/string/) substitute every string matching regexp in the line(s) with string. If string contains &, & is replaced by the characters which matched regexp. First character after s is delimiter; it can be any character not in either regexp or string. Strings matching regexp do not overlap and the result of substitution is not rescanned.

write (w)
Write current buffer into specified segment.

ADR1,ADR2w {path} (1,$w path) write lines into segment named path. If path omitted, a default pathname used if possible, otherwise error message printed.

Extended Edit Requests

execute (e)
Pass remainder of request line to the Multics command processor (i.e., escape to execute other Multics commands).

e <command line> execute command line without leaving editor.

global (g)
Print, delete, or print line number of all addressed lines that contain a match for a specified character string.

ADR1,ADR2gX/regexp/ (1,$gX/regexp/) perform operation on lines that contain a match for regexp; X must be d for delete, p for print, or = for print line numbers.

exclude (v)
Print, delete, or print line number of all addressed lines that do not contain a specified character string.

ADR1,ADR2vX/regexp/ (1,$vX/regexp/) perform operation on lines that do not contain a match for regexp; X must be d for delete, p for print, or = for print line numbers.

Buffer Requests

buffer (b)
Switch to specified buffer (i.e., switch all subsequent editor operations to the specified buffer).

b(X) go to buffer named X; destroy old contents of buffer X.

move (m)
Move specified line or lines into the specified buffer.

ADR1,ADR2m(X) (.,.m(X)) move line(s) from current buffer into buffer named X; destroy old contents of buffer X.

status (x)
Print a summary of the status of all buffers currently in use.

x give the status of all buffers in use.

nothing (n)
Do nothing (used to address a line with no other action).

ADRn (.n) set value of "." to line addressed.

comment (")
Ignore the remainder of this request line.

ADR" (.") ignore rest of line; used for comments.

Spacing

The following rules govern the use of spaces in editor requests.

1. Spaces are taken as literal text when appearing inside of regular expressions. Thus, /the n/ is not the same as /then/.

2. Spaces cannot appear in numbers, e.g., if 13 is written as 1 3, it is interpreted as 1+3 or 4.

3. Spaces within addresses except as indicated above are ignored.

4. The treatment of spaces in the body of an editor request depends on the nature of the request.

Responses From the Editor

In general, the editor does not respond with output on the terminal unless explicitly requested to do so (e.g., with a print or print line number request). The editor does not comment when you enter or exit from the editor or change to and from input and edit modes. The use of frequent print requests is recommended for new users of the qedx editor.

Stopping qedx Execution

If you inadvertently request a large amount of terminal output from the editor and wish to abort the output without abandoning all previous editing, you can issue the quit signal (by pressing the proper key on your terminal, e.g., BRK, ATTN, INTERRUPT), and, after the quit response, you can reenter the editor by invoking the program_interrupt (pi) command (fully described in the Commands). This action causes the editor to abandon its printout, but leaves the value of "." as if the printout had gone to completion.

qedx, qx

If an error is encountered by the editor, an error message is printed on your terminal and any editor requests already input (i.e., read ahead from the terminal) are discarded.

If you exit from qedx by issuing the quit signal, and subsequently invoke qedx in the same process, the message "qedx: Pending work in previous invocation will be lost if you proceed; do you wish to proceed?" is printed on the terminal. You must type a "yes" or "no" answer.

Macro Usage

You can place elaborate editor request sequences (called macros) into auxiliary buffers and then use the editor as an interpretive language. This use of qedx requires a fairly detailed understanding of the editor. To invoke a qedx macro from command level, you merely place your macro in a segment that has the letters qedx as the last component of its name, then type:

```
qedx path optional_args
```

where:

1. path
   specifies the pathname of a segment from which the editor is to take its initial instructions. Such a set of instructions is commonly referred to as a macro. The editor automatically concatenates the suffix qedx to path to obtain the complete pathname of the segment containing the qedx instructions.

2. optional_args
   are optional arguments that are appended, each as a separate line, to the buffer named args (the first optional argument becomes the first line in the buffer and the last optional argument becomes the last line). Arguments are used in conjunction with a macro specified by the path argument.

The editor executes the qedx requests contained in the specified segment and then waits for you to type further requests. If path is omitted, the editor waits for you to type a qedx request.

Notes

While most users interact with the qedx editor through a terminal, the editor is designed to accept input through the user_input I/O switch and transmit output through the user_output I/O switch. These switches can be controlled (using the iox_ subroutine described in the Subroutines) to interface with other devices/files in addition to the user's terminal. For convenience, the qedx editor description assumes that the user's input/output device is a terminal.

APPENDIX C

SUMMARY OF ADDRESSING CONVENTIONS

There are three basic means by which lines in the buffer can be addressed:

1. Addressing by absolute line number
2. Addressing by relative line number, i.e., relative to the "current" line
3. Addressing by context

In addition, a line address can be formed using a combination of the above techniques.

Addressing by Absolute Line Number

Each line in the buffer can be addressed by a decimal integer indicating the current position of the line within the buffer. The first line in the buffer is line 1, the second line 2, etc. The last line in the buffer can be addressed either by line number or by using the $ character, which is interpreted to mean "the last line currently in the buffer." In certain cases it is possible to address the (fictitious) line preceding line 1 in the buffer by addressing line 0.

As lines are added to or deleted from the buffer, the line numbers of all lines that follow the added or deleted lines are changed accordingly. For example, if line 15 is deleted from the buffer, line 16 becomes line 15, 17 becomes 16, and so on.

If an attempt is made to address a line not contained in the buffer, an error message is printed by the editor. If the buffer is currently empty, as it is when the editor is first entered, only the line numbers 0 and $ are considered valid.

Addressing by Relative line Number

The qedx editor maintains the notion of a "current" line that is specified by using the character "." (period). Normally, the current line is the last line addressed by an edit request or the last line entered from the terminal by an input request. The value of "." after each editor request is documented in the description of the request.

Lines can be addressed relative to the current line number by using an address consisting of "." followed by a signed decimal integer specifying the position of the desired line relative to the current line. For example, the address .+1 specifies the line immediately following the current line and the address .-1 specifies the line immediately preceding the current line.

When specifying an increment to the current line number, the + sign can be omitted (e.g., .5 is interpreted as .+5). In addition, when specifying a decrement to the current line number, the "." itself can be omitted (e.g., -3)is interpreted as .-3). It is also possible to follow the "." with a series of signed decimal integers (e.g., .5+5-3 is interpreted as .+7).

Addressing by Context

Lines can be addressed by context by using a "regular expression" to match a string of characters on a line. When used as an address, a regular expression specifies the first line encountered that contains a string of characters that matches the regular expression. In its simplest form, a regular expression is a character or a string of characters delimited by the right slant character (/). For example, in the following text, the regular expression /abc/ matches line 2.

```
a:  procedure;
    abc=def;
    x=y;
    end a;
```

To use a regular expression as an address, the user types /regexp/, where regexp is any valid regular expression as described below. The search for a regular expression begins on the line following the current line (i.e., .+1) and continues through the entire buffer, if necessary, until it again reaches the current line. In other words, the search proceeds from .+1 to $ and then from line 1 to the current line. If the search is successful, /regexp/ specifies the first line encountered during the search in which a match was found.

A regular expression can consist of any character in the ASCII set except the newline character. However, the following characters have specialized meanings in regular expressions.

/	Delimits a regular expression used as an address.
*	Signifies "any number (or none) of the preceding character".
^	When used as the first character of a regular expression, the ^ character signifies the character preceding the first character on a line.
$	When used as the last character of a regular expression, the $ character signifies the character following the last character on a line.
.	Matches any character on a line.

Some examples follow:

/a/	Matches the letter "a" anywhere on a line.
/abc/	Matches the string "abc" anywhere on a line.
/ab*c/	Matches "ac", "abc", "abbc", "abbbc", etc. anywhere on a line.
/in..to/	Matches "in" followed by any two characters followed by "to" anywhere on a line.
/in.*to/	Matches "in" followed by any number of any characters (including none) followed by "to" anywhere on a line.

/^abc/	Matches a line beginning with "abc".
/abc$/	Matches a line ending with "abc".
/^abc.*def$/	Matches a line beginning with "abc" and ending with "def".
/^.*$/	Matches any line.
/^$/	Matches an empty line (a line containing only a newline character).

The special meanings of "/", "*", "$", "^", and "." within a regular expression can be removed by preceding the special character with the escape sequence \c.

/\c/\c*/	Matches the string "/*" anywhere on a line.

The editor remembers the last regular expression used in any context. The user can reinvoke the last used regular expression by using a null regular expression (i.e., //). In addition, a regular expression can be followed by a signed decimal integer in the same manner as when addressing relative to the current line number. For example, the addresses /abc/+5-3, /abc/+2 or /abc/2 all address the second line following a line containing "abc".

The two uses of "." and "$" (as line numbers and as special characters in regular expressions) are distinguished by context.

Compound Addresses

An address can be formed using a combination of the techniques described above. The following rules are intended as a general guide in the formation of these compound addresses.

1. If an absolute line number is to appear in an address, it must be the first component of the address.

2. A relative line number can appear anywhere in a compound address.

3. A regular expression can appear anywhere in a compound address.

   a. An absolute line number can be followed by a regular expression. This construct is used to begin the regular expression search after a specific line number. For example, the address 10/abc/ starts the search for /abc/ immediately after line 10.

   b. A regular expression can follow or be followed by an address specified by a relative line number. For example, the address .-8/abc/ starts the search eight lines before the current line, while /abc/.8 addresses the line eight lines after the first occurrence of /abc/.

   c. A regular expression can be followed by another regular expression. For example, the address /abc//def/ matches the first line containing "def" appearing after the first line containing "abc". As mentioned earlier, a regular expression can be followed by a decimal integer. For example, the address /abc/-10/def/.5 starts the search for /def/ 10 lines before the first line to match /abc/ and if /def/ is matched, the value of the compound address is the fifth line following the line containing the match for /def/.

Addressing a Series of Lines

Several of the editor requests can be used to operate on a series of lines in the buffer. To specify a series of lines, two addresses must be given in the following general form:

```
ADR1,ADR2
```

The pair of addresses specifies the series of lines starting with the line addressed by the address ADR1 through the line addressed by ADR2, inclusive.

Examples:

1,5	specifies line 1 through line 5.
1,$	specifies the entire contents of the buffer.
.1,/abc/	specifies the line following the current line through the first line (after the current line) containing "abc".

When a comma is used to separate addresses, the address computation of the second address is unaffected by the computation of the first address (i.e., the value of "." is not changed by the evaluation of the first address). For example, the address pair:

```
.1,.2
```

specifies a series of two lines, the line immediately after the current line through the second line after the current line.

However, if a semicolon is used to separate addresses instead of a comma, the value of "." is set to the line addressed by ADR1 before the evaluation of ADR2 begins. In contrast to the example given immediately above, the address pair:

```
.1;.2
```

specifies a series of three lines, the line immediately following the original current line through the second line following the line specified by ADR1. As a further example, the address pair:

```
/abc/10
```

is equivalent to the address pair:

```
/abc/,/abc/+10
```

Addressing Errors

The following list describes the various errors that can occur when the editor is attempting to evaluate an address.

1. "Buffer empty" -- An attempt has been made to reference a specific line when the buffer is empty. (Only "$", "." and "0" are legal addresses within an empty buffer and only if used with a read, append, or insert request.)

2. "Address out of buffer" -- An attempt has been made to refer to a nonexistent line (e.g., an address of 20 when there are fewer than 20 lines in the buffer or an address of .+5 when the current line is fewer than 5 lines from the last line in the buffer).

3. "Address wrap around" -- An attempt has been made to address a series of lines in which the line number of the second line addressed is less than the line number of the first (e.g., $,1).

4. "Search fai ed" -- A regular expression search initiated from the user's terminal has failed to find a match.

5. "Syntax error in regular expression" -- A regular expression used as an address has not been properly delimited, or successive asterisks have been encountered without an escape sequence character (e.g., /ab**c/).

6. "// undefined" -- A null regular expression has been used and no previously typed regular expression is available.

APPENDIX D

REQUEST DESCRIPTIONS

This appendix describes each qedx request in detail; the format is described below. Described first are the basic requests (in the order listed below), then the extended edit requests, and finally the buffer requests.

REQUEST DESCRIPTIONS

A request to qedx can generally take any one of the following three general formats:

```
<request>
ADR<request>
ADR1,ADR2<request>
```

where ADR is a one-line address, ADR1 and ADR2 are the first and second components of an address range, and <request> is a qedx request. When addressing a series of lines (ADR1,ADR2<request>), any one of the three types of addressing (absolute, relative, or context) can be used for either ADR1 or ADR2. For example, if line 1 is the current line and the buffer contains the following:

```
b:procedure;
a=r;
c=s;
k=t;
end b;
```

You could print lines 2 through 4 by typing a p (print) request preceded by any one of several address combinations; a few of the possible print requests are given below:

```
2,4p
2,+3p
2,/^k/p
+1,/^k/p
/a=/,+3p
/a=/,/^k/p
```

In each of the request descriptions that follow, several "standard" headings are used:

Name	gives the invocation character followed by the request name
Format	shows the proper format to use when invoking the request
Default	explains what action qedx takes if you do not specify an address in the request
"."->	identifies the position of the current line after the request operation is completed
Example	shows correct usage of the request, including buffer contents before and after the request is given, and shows line(s) printed at the terminal as a result of the request (console output)

Basic Requests

The basic requests are presented in a functional order, i.e., input requests before edit requests. Within the edit requests, read is first and quit is last. The exact order is:

Invocation Character		Name
a	\f	append
c	\f	change
i	\f	insert
r		read
p		print
=		print line number
d		delete
s		substitute
w		write
q		quit

NOTE: You should remember when entering text that you must terminate an input request with the \f escape sequence. The qedx editor cannot respond to another request until it is in edit mode. All lines, including ones you intend as requests, are regarded as input until the \f escape sequence is given.

Advanced Requests

The second group of descriptions consists of the advanced qedx editing requests: extended edit requests and buffer requests.

EXTENDED EDIT REQUESTS

Invocation Character	Name
e	execute
g	global include
v	global exclude

BUFFER REQUESTS

Invocation Character	Name
b	change buffer
m	move
x	buffer status
n	nothing
"	comment

a (append) a (append)

A (APPEND)

The append request enters input lines from the terminal to create a new segment, or, appends these lines after the line addressed by the append request.

Format: ADRa or a
TEXT TEXT
\f \f

Default: a means append after current line.

Value of ".": set to last line appended.

Example 1:

Buffer contents:

```
<buffer empty>
```

Request sequence:

```
a
This can be
a letter,
memo, report,
or user manual.
\f
```

Resulting buffer contents:

```
This can be
a letter,
memo, report,
"."->  or user manual.
```

Resulting console output:

None

Example 2:

Buffer contents:

```
This can be
a letter,
or user manual.
```

Request sequence:

```
2a                              /letter/a
memo, report,        or         memo, report,
\f                              \f
```

Resulting buffer contents:

```
     This can be
     a letter,
"."-> memo, report,
     or user manual.
```

Resulting console output:

None

Example 3:

Buffer contents:

```
the text editor,
qedx, is a
text editor
```

Request sequence:

```
2a                          /qedx/a
line-oriented      or       line-oriented
\f                          \f
```

Resulting buffer contents:

```
     the text editor,
     qedx, is a
"."-> line-oriented
     text editor.
```

Resulting console output:

None

Note: The request 0a can be used to insert text before line 1 of the buffer.

C (CHANGE)

The change request enters input lines from the terminal and inserts the new text in place of the addressed line or lines.

Format:

```
ADR1,ADR2c or ADRc or c
TEXT                  TEXT          TEXT
\f                    \f            \f
```

Default: c means change the current line.

Value of ".": set to last line input.

Example 1:

Buffer contents:

```
This can be
a letter,          or     This can be
memo,                     a letter,
report                    memo
or user manual.           or user manual.
```

Request sequence:

```
3,4c                          3c
memo, report,        or       memo, report,
\f                            \f
```

Resulting buffer contents:

```
     This can be
     a letter,
"."-> memo, report,
     or user manual.
```

Resulting console output:

None

Example 2:

Buffer contents:

```
The text editor,
qedx, is a
line-
oriented
text editor.
```

Request sequence:

```
3,4c
line-oriented
\f
```

Resulting buffer contents:

```
The text editor,
qedx, is a
line-oriented
text editor.
```

Resulting console output:

None

I (INSERT)

The insert request enters input lines from the terminal and inserts the new text immediately before the addressed line.

Format:

```
ADRi or i
TEXT          TEXT
\f            \f
```

Default: i means insert before current line.

Value of ".": set to last line inserted.

Example 1:

Buffer contents:

```
This can be
memo, report,
or user manual.
```

Request sequence:

```
2i                          /memo/i
a letter,        or         a letter,
\f                          \f
```

Resulting buffer contents:

```
      This can be
"."-> a letter,
      memo, report,
      or user manual.
```

Resulting console output:

None

Example 2:

Buffer contents:

```
The text editor
qedx, is a
text editor.
```

Request sequence:

```
3i                          $i
line-oriented    or         line-oriented
\f                          \f
```

Resulting buffer contents:

```
     The text editor,
     qedx, is a
"."-> line-oriented
     text editor.
```

Resulting console output:

None:

R (READ)

The read request puts the contents of an already existing segment into the buffer. This request appends the contents of a specified segment to the buffer after the addressed line.

Format: r path or ADRr path

where path is the pathname of the segment to be read into the buffer. The pathname can be preceded with any number of spaces and must be followed immediately by a newline character.

Default: r path is taken to mean $r path.

Value of ".": set to the last line read from the segment.

Example 1:

Buffer contents:

```
     You can input:
     text,
     programs
```

Request:

```
     2r extra    or    /text/r extra
```

where extra is a segment containing the following text:

```
     (such as
     letters, memos,
     lengthy reports)
```

Resulting buffer contents:

```
     You can input:
     text,
     (such as
     letters, memos,
"."-> lengthy reports)
     programs
```

Resulting console output:

```
     None
```

Example 2:

Buffer contents:

```
     <buffer empty>
```

Request:

```
r b.pl1
```

where b.pl1 is the following:

```
b:  procedure;
    c=d
    end b;
```

Resulting buffer contents:

```
b:  procedure;
    c=d
"."->   end b;
```

Resulting console output:

None

Note: The request "0r path" is used to insert the contents of a segment before line 1 of the buffer.

The read request sets the default pathname for the buffer to the pathname given in the request, if the buffer was empty before the request. (See also the write request below.) For example:

Buffer contents:

```
<buffer empty>
```

Request:

```
r file1
```

reads the contents of file1 into the buffer; if you edit file1 and then issue a write request:

Request:

```
w
```

your edited version is written to the default pathname, file1.

However, the read request issued in a nonempty buffer resets the default pathname for the buffer to null. Thus, you can read an unlimited number of segments into the buffer, but when you attempt to issue a write, you must specify a pathname, i.e., those segments are "protected" from being destroyed.

For example:

Buffer contents:

```
<buffer empty>
```

Request:

```
r file1
r file2
w
```

qedx prints:

```
No pathname given.

w newfile
```


P (PRINT)

The print request prints the addressed line or set of lines on your terminal.

Format: ADR1,ADR2p or ADRp or p

Default: p means print the current line.

Value of ".": set to last line addressed by the print request (i.e., the last line to be printed).

Example 1:

Buffer contents:

```
     a:  procedure;
         x=y;
         q=r;
         s=t;
         end a;
```

Request:

```
     2,4p      or      /x=/,/s=/p
```

Resulting buffer contents:

Same as above.

Resulting console output:

```
         x=y;
         q=r;
"."->    s=t;
```

Example 2:

Request:

```
     1,$p
```

Resulting buffer contents:

Same as above

Resulting console output:

```
     a:  procedure;
         x=y;
         q=r;
         s=t;
"."->    end a;
```

Example 3:

Buffer contents:

```
The text editor,
qedx, is a
line-oriented
editor.
```

Request sequence:

```
1,3p
```

Resulting buffer contents:

Same as above

Resulting console output:

```
The text editor,
qedx, is a
"."-> line-oriented
```

There is a special case of the print request that sets the value of "." to a specific line and prints the line. This usage needs no letter to tell qedx what operation to perform; you merely type a valid address (generally a context address although any type is permitted) followed by a newline character. In context addressing, a search is done through the file starting with the first line after the current line to the last line in the file, and then from line number one to the current line (see the discussion of context addressing in Appendix C).

Format: ADR

Default: typing a period (.) prints the current line.

Value of ".": set to line addressed by request.

Example 1:

Buffer contents:

```
        aardvark
"."->   emu
        gnu
        kiwi
        rhea
```

Request:

```
/^k/    or    +2    or    4
```

Result:

```
kiwi
```

Example 2:

Buffer contents:

```
        aardvark
"."->   emu
        gnu
        kiwi
        rhea
```

Request:

```
/^a/    or    -1    or    1
```

Result:

```
aardvark
```

= (print line number)

= (PRINT LINE NUMBER)

This request prints the line number of the addressed line.

Format: ADR= or =

Default: = means print the line number of the current line.

Value of ".": set to line addressed by request.

Example 1:

Buffer contents:

```
a:  procedure;
    x=y;
    p=q;
    end a;
```

Request:

```
/q;/=
```

Resulting buffer contents:

Same as above

Resulting console output:

```
3
```

= (print line number)	= (print line number)

Example 2:

Buffer contents:

Same as above

Request:

$=

Resulting buffer contents:

Same as above

Resulting console output:

4

Example 3:

Buffer contents:

```
The text editor,
qedx, is a
line-oriented
text editor.
```

Request sequence:

/line/=

Resulting buffer contents:

Same as above

Resulting console output:

3

= (print line number)

= (print line number)

Example 4

Buffer contents:

```
     The text editor,
     qedx, is a
"."-> line-oriented
     text editor.
```

Request:

```
=
```

Resulting buffer contents:

Same as above

Resulting console output:

```
3
```

d (delete)

d (delete)

D (DELETE)

The delete request deletes the addressed line or set of lines from the buffer.

Format: ADR1,ADR2d or ADRd or d

Default: d means delete the current line.

Value of ".": set to line immediately following the last line deleted.

d (delete)

Example 1:

Buffer contents:

```
a:  procedure;
    x=y;
    q=r;
    s=t;
    end a;
```

Request sequence:

```
3,4d      or      /q=/,/s=/d
```

Resulting buffer contents:

```
a:  procedure;
    x=y;
"."->   end a;
```

Resulting console output:

None

Example 2:

Buffer contents:

```
The text editor,
qedx, is a
line-oriented
text editor.
```

Request sequence:

```
3d
```

d (delete)

Resulting buffer contents:

```
     The text editor,
     qedx, is a
"."-> text editor.
```

Resulting console output:

None

s (substitute)

S (SUBSTITUTE)

The substitute request modifies the contents of the addressed lines, by replacing all strings that match a given regular expression with a specified character string.

Format:

```
ADR1,ADR2s/REGEXP/STRING/
     or
ADRs/REGEXP/STRING/
     or
s/REGEXP/STRING/
```

(The first character after the "s" is taken to be the request delimiter and can be any character not appearing in either REGEXP or STRING. It must be the same in all three instances.)

Default: s/REGEXP/STRING/ means substitute STRING for REGEXP in the current line.

Value of ".": set to last line addressed by request.

s (substitute)

Operation: Each character string in the addressed line or lines that matches REGEXP (see Section 3 for a description of regular expressions) is replaced with the character string STRING. If STRING contains the special character &, each & is replaced by the string matching REGEXP. The special meaning of & can be suppressed by preceding the & with the \c escape sequence. However, when used in REGEXP the ampersand has no special meaning.

Example 1:

Buffer contents:

```
The quick brown sox
```

Request:

```
s/sox/fox/
```

Resulting buffer contents:

```
The quick brown fox
```

Resulting console output:

None

Example 2:

Buffer contents:

```
The qedx text editor
```

Request:

```
s/qedx/(&)/
```

Resulting buffer contents:

```
The (qedx) text editor
```

Resulting console output:

None

Example 3:

Buffer contents:

```
a=b
c=d
x=y
```

Request:

```
1,$s/$/;/
```

Resulting buffer contents:

```
a=b;
c=d;
".">  x=y;
```

Resulting console output:

None

Example 4:

Buffer contents:

```
The text editor,
qedx, is a
line-oriented
text editor.
```

s (substitute)

Request sequence:

```
1,$ s/^/?/
```

Resulting buffer contents:

```
?The text editor,
?qedx, is a
?line-oriented
?text editor.
```

Resulting console output:

None

w (write)

W (WRITE)

The write request writes the addressed line or set of lines from the buffer into a specified segment. The buffer and current line are unchanged.

Format: w path or ADR1,ADR2w path

where path is the pathname of the segment whose contents are to be replaced by the addressed lines in the buffer. If the segment does not already exist, a new segment is created with the specified name. If the segment does already exist, the old contents are replaced by the addressed lines. The old segment contents are destroyed.

w (write)

The pathname can be preceded by any number of spaces and must be followed immediately by a newline character. If path is omitted, the default pathname for the buffer is used. If path is omitted and the buffer has no default pathname, the message "No pathname given" is printed and no action is taken.

Default: w path is taken to mean 1,$w path.

Value of ".": unchanged.

Example 1:

Buffer contents:

```
A regular expression
searches for a
certain character
string in the buffer
```

Request:

```
w reg_exp
```

Resulting buffer contents:

Either the buffer contents replaces the contents of the segment reg_exp in your working directory (if the segment named reg_exp already exists) or reg_exp is created in the working directory and contains the contents of the buffer. The buffer contents are unchanged.

Resulting console output:

None

Example 2:

Buffer contents:

```
The text editor,
qedx, is a
line-oriented
text editor.
```

Request sequence:

```
w editor
```

Resulting buffer contents:

Same as above; now a segment named editor has been created containing same contents as the buffer.

Resulting console output:

None

Note: The write request must be the last request on a line.

Q (QUIT)

The quit request is used to exit from the editor and does not itself to save the results of any editing that might have been done. If you wish to save the modified contents of the buffer, you must explicitly issue a write request.

Format: q

Default: the quit request cannot have an address.

Note: The quit request must be the last request on a line.

E (EXECUTE)

The execute request invokes the Multics command system without exiting from the editor. Whenever an execute request is recognized, the remaining characters in the request line are received as if you were out of qedx and at command level. The execute request can be followed by any legal Multics command line. However, you should not invoke qedx again.

Format: e <command line>

Value of ".": Unchanged.

Example: The request line:

```
e print report
```

can be used to print the segment in your working directory named report. After the segment is printed on the your terminal, you can continue your work in qedx as though you had not issued the execute request. (You are still "in" qedx.)

The request line:

```
e list; print_mail
```

lists the contents of the your working directory and prints the contents of your mailbox (if any).

Note: If you wish to abort a command line invoked with the execute request, you can issue the quit signal and then invoke the program_interrupt (pi) command (described in the Commands) to abort the command line and restore control to qedx.

G (GLOBAL)

The global request is used in conjunction with one of the following requests: print, delete, or print line number. The d, p, or = request operates only on those lines addressed by the global request that contain a match for a specified regular expression.

Format: ADR1,ADR2gX/REGEXP/ or gX/REGEXP/

where X must be one of the following requests:

d to delete lines containing REGEXP
p to print lines containing REGEXP
= to print the line numbers of lines containing REGEXP

The character immediately following the request X is taken to be the regular expression delimiter and can be any character not appearing in REGEXP.

Default: gX/REGEXP/ is taken to mean 1,$gX/REGEXP/

Value of ".": Set to ADR2 of request, if an address range is given, or to the last line in the buffer if no address is given.

Example 1:

Buffer contents:

```
eagle
whale
wolf
baby sea lion
star
```

Request:

```
gd/w/
```

Resulting buffer contents:

```
eagle
baby sea lion
"."-> star
```

Resulting console output:

None

Example 2:

Buffer contents:

```
The text editor,
qedx, is a
line-oriented
text editor.
```

Request sequence:

```
gpzxz
```

Resulting buffer contents:

Same as above

Resulting console output:

```
The text editor,
qedx, is a
text editor.
```


V (EXCLUDE)

The exclude request is also used in conjunction with one of the following requests: print, delete, or print line number. The d, p, or = request operates only on those lines addressed by the exclude request that do not contain a match for a specified regular expression.

Format: ADR1,ADR2vX/REGEXP/ or vX/REGEXP/

where X must be one of the following requests:

- d to delete lines not containing REGEXP
- p to print lines not containing REGEXP
- = to print the line numbers of lines not containing REGEXP

The character immediately following the request X is taken to be the regular expression delimiter and can be any character not appearing in REGEXP.

Default: vX/REGEXP/ is taken to mean 1,$vX/REGEXP/

Value of ",": Set to ADR2 of request, if an address range is given, or to the last line in the buffer if no address is given.

Example 1:

Buffer contents:

```
eagle
whale
baby sea lion
wolf
star
```

Request:

```
v=/w/
```

Resulting buffer contents:

Same as above

Resulting console output:

```
1
3
5
```

Example 2:

Buffer contents:

```
The text editor,
qedx, is a
line-oriented
text editor.
```

Request sequence:

```
vpyxy
```

v (exclude)	v (exclude)

Resulting buffer contents:

Same as above

Resulting console output:

line-oriented

b (change buffer)	b (change buffer)

B (CHANGE BUFFER)

The change buffer request designates an auxiliary buffer as the current buffer. The previously designated current buffer becomes an auxiliary buffer.

Format: bN or b(STR)

where N can be a single digit or character, and STR is more than one number or character string; N or (STR) is the name of the buffer that is to become the current buffer.

Value of ".": Restored to the value of "." when this buffer was last used as the current buffer (i.e., the value of "." is maintained separately for each buffer and saved as part of the buffer status). If a new buffer is created, then "." is undefined.

M (MOVE)

The move request moves one or more lines from the current buffer to a specified auxiliary buffer. The "moved" lines are deleted from the current buffer. The addressed lines replace the previous contents (if any) of the auxiliary buffer. The entire old contents of the auxiliary buffer are lost.

Format: ADR1,ADR2m(X) or ADRm(X) or m(X)

where X is the name of the auxiliary buffer to which the lines are to be moved.

Default: m(X) means move the current line

Value of ".": Set to the line after the last line moved in the current buffer.

Example:

Contents of:

Current Buffer	Buffer B
eagle	a letter
whale	a memo
baby sea lion	
wolf	
star	

Request:

3,4mB or /bab/,/wo/mB

Resulting buffer contents:

```
                Current Buffer              Buffer B
                                      "."->
                eagle                       baby sea lion
                whale                       wolf
          "."-> star
```

Resulting console output:

None

X (BUFFER STATUS)

The buffer status request prints a summary of the status of all buffers currently in use. The name and length (in lines) of each buffer is listed; the current buffer is marked with a right arrow "->" to the left of the buffer name. Finally, each buffer's default pathname, if any, is listed.

Format: x

Value of ".": Unchanged.

Example: If you have created the additional buffers text1 and program1 and have designated text1 as your current buffer, the output from the buffer status request might be as follows.

```
157      (0)      demo
 32    ->(text1)
 53      (program1)
```

This output indicates 157 lines in buffer 0 (the initial buffer), 32 lines in text1 (the current buffer) and 53 lines in program1. It also indicates that the default pathname for buffer 0 is demo (in your working directory) and that buffers text1 and program1 have no default pathnames.

N (NOTHING)

The nothing request addresses a particular line in the buffer (i.e., sets the value of "." to a particular line). No other action is taken; i.e., the line is not printed.

Format: ADRn or n

Default: n is taken to mean .n

Value of ".": Set to line addressed by request.

Example:

Buffer contents:

```
      read
"."-> write
      substitute
      change
      delete
```

Request:

```
/ch/n
```

Resulting buffer contents:

```
      read
      write
      substitute
"."-> change
      delete
```

Resulting console output:

None

Note: This request is normally only used in macros (see Section 5).

" (comment)

" (COMMENT)

The comment request is generally used to annotate qedx macros and also can be used to annotate online work. The editor ignores the remainder of the request line.

Format: ADR" or "

Default: " is taken to mean ."

Value of ".": Set to line addressed by request.

Example:

Buffer contents:

```
"This macro enters special headers
"for subroutine descriptions
b(heads)
a
<This is the macro>
```

Resulting console output:

None

APPENDIX E

qedx ERROR MESSAGES

This section lists the error messages that may be printed on your terminal during a qedx session, the meaning of each message, and describes what action you should take to remedy the error.

The error messages described here are listed in four categories: addressing errors, syntax errors, regular expression errors, and miscellaneous errors. Within each category, the messages are shown with the most frequently-received messages first to the rarely-received messages last.

There are about 20 error messages that may be printed at your terminal which are not described here. They are Multics error messages (not qedx error messages) which may, however, occur during a qedx session. They appear in the form:

qedx: <Multics error message>

The Multics error messages are fully described in the Multics Error Messages manual, Order No. CH26-00.

Below are some general suggestions to prevent common mistakes that generate these messages, and explanations for why you may receive an error message that seems to have no connection to an action you requested.

GENERAL PREVENTIVE SUGGESTIONS

One of the most common actions that results in an error message is when you attempt to type something while Multics is printing a response to your previous action.

If you type a qedx request (or any Multics command) that causes output to be printed at your terminal, always wait for Multics to finish printing before you type anything; otherwise the line you type gets garbled and a syntax error results. If this happens, type a few @ signs and retype your request on a new line.

Also important to keep in mind is that when you type a number of requests on a line and one request fails, the rest of the line is ignored (the action specified in any succeeding request(s) is not taken). Therefore, you must find out which request failed, and reissue that request and any that followed it.

There are cases where the error message you receive may seem totally inappropriate in relation to the request you meant to type. This is most often due to either a typographical error, or thinking you are in input mode when you are really in edit mode.

For instance, often syntax errors in edit requests occur when you think you are in input mode. Depending on the first letter of the line you type as input, qedx may interpret that line as an incorrectly formatted edit request (when it really was not intended as a request). In this case, you should type the appropriate input request, and then proceed.

Another common error is when you try to type a regular expression, either to locate a line or as part of a substitute request, and forget one of the delimiters. Again, depending on the first letter of your intended regular expression, qedx may interpret that line as an incorrectly formatted request of some kind, so the error message you receive may seem unrelated to your intended action. Usually you can trace the problem back to a typographical error.

ADDRESSING ERRORS

Search failed.

This means that a regular expression search has failed, the request was aborted, and your current line remains the same as before you made the search. No action is taken; your current line remains the same.

You should check to see if you have typed the regular expression correctly.

Address out of buffer (too big).

This means that the address you specified is a line that is not within the addressing range (e.g., if you address line 20 and the buffer only contains 15 lines). No action is taken; your current line is the same as before you made the request.

You should:

1. Check to make sure that you have typed the address correctly, that it does not exceed the length of your buffer, and retype the address on a new line.

2. Check to make sure that you have addressed the correct line; another way to address the line you want is to use a regular expression to locate the line containing a certain character string.

Address out of buffer (negative address).

This means that you have specified a line that is not within the addressing range, e.g., if you address -5 (meaning go backwards in the buffer five lines) when you are on line one. No action is taken; your current line is the same as before you made the request.

You should:

1. Type a period to print the current line or an equal sign to print the current line number.

2. You can avoid this error message by locating the line with a regular expression instead of line number (find the line by something you know to be on it).

Address wrap-around

This means that you typed an address range that is not valid. The pointer only moves in one direction, which is toward the last line in the buffer. Thus the second address in a range must be greater than the first. For example, the print request 20,15p produces this error message since the second line number specified is less than the first. No action is taken and your current line remains the same.

You should:

1. Check the addresses you typed in the range to make sure that the second address is greater than the first.

2. Try addressing the lines using a different form of address; it may make more sense.

Address syntax error.

This means that whatever you typed has been interpreted as a mistyped address. No action is taken; your current line is the same as before you made the request.

You should check to make sure that the request you typed is a legal form of address, and type it again on a new line.

SYNTAX ERRORS

Syntax error in substitute request.

This means that whatever you typed has been interpreted as a mistyped substitute request. No action is taken and your current line remains the same. The usual cause of this message is leaving out the last delimiter, e.g. s/a/b.

You should:

1. If you meant the line to be input, type an input request and then proceed.

2. If you meant to specify a regular expression, surround it with delimiters.

3. For a substitute request, make sure that the delimiter (character immediately following the "s") is the same in all three instances, and that you have remembered to type a closing delimiter.

Syntax error in regular expression.

This means that whatever you typed has been interpreted as a mistyped regular expression. No action is taken and your current line is the same as before you made the request.

You should:

1. Check to make sure that the line you typed is a legal regular expression.

2. If there are special characters (for instance, slashes) within your regular expression, make sure that you have preceded them by the "\c" escape sequence.

Syntax error in quit request.

This means that whatever you typed has been interpreted as a mistyped quit request (q). No action is taken and your current line remains the same.

You should:

1. If you meant the line to be input, type an input request and then proceed.

2. If you meant to specify a regular expression, surround it with delimiters.

3. If you really are attempting to quit, you cannot precede the "q" with any form of address, and it must be followed immediately by a newline character.

Syntax error in global request.

This means that whatever you typed has been interpreted as mistyped global request. No action is taken and your current line remains the same.

You should:

1. Make sure that you have included the request that accompanies the global request, indicating what action is to be taken globally.

2. Make sure that the regular expression has been closed with a delimiter.

REGULAR EXPRESSION ERRORS

// undefined in regular expression.
This means that you attempted to locate a line using a null regular expression without first defining a regular expression. (A null regular expression (//) means repeat the search for the last regular expression typed.) No action is taken, and your current line remains the same.

You should type the regular expression in its entirety.

Invalid use of * in regular expression.
This means that you have incorrectly (or inadvertently) used the special character asterisk within a regular expression.

You should:

1. Check the line you typed--examine your usage of the asterisk (see the discussion of special characters).

2. If you want the asterisk to be a literal part of the expression, precede it with the "\c" escape sequence.

Regular expression is too long.
This means that whatever you typed was interpreted as a regular expression that is too long. No action taken; current line remains the same. (This message is very rare.)

You should:

1. Make sure that you are attempting to define a regular expression and not make a substitution.

2. If it is a regular expression, shorten it.

Regular expression is too complex.
This means that whatever you typed has been interpreted as a regular expression that is too complex. No action is taken; current line remains the same. (This message is very rare.)

You should:

1. Make sure that you are attempting to define a regular expression and not make a substitution.

2. If it is a regular expression, redefine it more simply.

MISCELLANEOUS ERRORS

Substitution failed.
This means that the substitute request you attempted did not work. No action is taken and your current line remains the same.

You should:

1. Make sure that you have typed the sequence of characters to be replaced exactly as they appear on the line (this is the most common mistake).

2. Make sure that you are on (or have specified) the current line on which to make the substitution.

3. If you have used special characters (. * ^ $) in any part of the request, make sure you have used them correctly or preceded them (for literal translation) with the "\c" escape sequence.

4. Make sure the delimiter that the character immediately following the "s" is the delimiter that is used throughout the request.

qedx: X not recognized as a request.

This means that qedx was expecting a request or an address as the first letter of the line you typed, and instead it encountered X, where X is anything other than a request or address. Since X is not recognized as a request, the rest of the line you typed is ignored; your current line remains the same.

You should:

1. If you meant the line to be input, type an input request and then proceed.

2. If you meant to specify a regular expression, surround it with delimiters.

Buffer empty.

This is self-explanatory; you have issued a request and qedx cannot carry it out because the buffer is empty.

You should:

1. Check to see if you forgot to put something in the buffer with a read or input request.

2. See if you have inadvertently moved lines into another buffer by typing either an "m" or "b" request as the first letter on a line while in edit mode, or type "x" to see if you have more than one buffer.

3. If you are working with more than one buffer, you may be in a different buffer than you think. Type an "x", which lists your buffers and shows which buffer you are currently working in.

4. Check to see if you typed a "d" or a line beginning with a "d", in which case you have deleted a line or lines of text.

No pathname given.

This means that you have issued a read request without specifying a pathname, or a write request in an instance where an accompanying pathname is required. No action is taken, and your current line remains the same.

You should:

1. Type an "x" to list buffers--you may be in the wrong buffer.

2. Issue the request again, this time with a pathname.

3. Remember, if you write the buffer contents to an existing pathname (that is, the pathname of an existing segment), you overwrite the contents of that segment.

Buffer (Name) not found.

This means that you have specified a request to act on a buffer named (Name), which you have not created. You are still on the same line in the same buffer as before you made the request.

You should:

1. If the buffer you specified has a name of more than one character, make sure you type the name within parentheses.

2. Type an "x" to display a list of all buffers, their names, number of lines in each, and show which buffer you are currently working in.

3. If the buffer you specified does not exist, you can create it with the change buffer (b) request.

qedx: Command line too long.

This means that the command line that you used to call qedx is too long (e.g., attempting to invoke qedx with a command line of the form "qedx macro first_argument second_argument..." where the combined characters in the arguments is over 512 characters).

You should issue a shorter command line.

```
qedx:  Pending  work in previous invocation  will be lost if
     you proceed;
do you wish to proceed?
```

This means that you have invoked qedx again without first issuing a quit request to exit from your previous invocation. This is not allowed in qedx. (You have probably pressed the QUIT key and are attempting to reenter qedx.)

You should type "no" when asked if you wish to proceed, which will take you back into qedx to the same line you were on. Your work is not lost this way, and you can continue as if you had not been interrupted.

If you type "yes" in answer to the query above, you enter qedx anew--the buffer is empty and all previous work is lost.

```
qedx:  Entry not found.  start.compin
Error in buffer args at level 2.

Current buffer is 0 at level 0.

Buffer args not found.
Error in buffer exec at level 1.
Unexecuted lines in buffer:

b0
Current buffer is exec at level 0.
```

These are two separate occurrences of the same buffer error message. The first line in each case is different because this is the line that shows what problem caused this buffer error.

This message means that there was an error while executing in a buffer and only occurs when you are working with macros in more than one buffer.

You should type an "x" for buffer status, and see which buffer you are in and which buffer you want to go to.

INDEX

MISCELLANEOUS

! (precedes user-typed lines) 1-2

" (comment) 5-10

#
see character deletion

$
(in regular expression) 3-11
(print last line) 3-6

& (ampersand) 3-18

* 3-10

. (print current line) 2-2

. in regular expression 3-10

/ 3-9, 3-10

// 3-13

= (print line number) 3-6, D-16

@
see line deletion

\c 3-12

^ 3-11

A

a (append) 2-2, D-3

absolute line number 3-3

adding text 3-19

address 3-2
absolute 3-3, C-1
compound 3-14, C-4
context 3-9, C-2
default 3-14
range 3-5, C-5
relative 3-6, C-2
see also locating lines

addressing conventions
summary C-1

addressing mistakes, common 3-15

ADR (address) 3-2, D-1

advanced edit requests 5-1, D-3
buffer requests 5-7
(comment) D-36
b (change buffer) D-31
m (move) D-32
n (nothing) D-34
x (buffer status) D-33
extended edit requests 5-2
e (execute) D-26
g (global) D-27

advanced edit requests (cont)
  extended edit requests
    v (exclude) D-29

B

b (buffer change) 5-8

basic request descriptions
  = (print line number) D-16
  a (append) D-3
  c (change) D-6
  d (delete) D-18
  i (insert) D-7
  p (print) D-12
  q (quit) D-25
  r (read) D-9
  s (substitute) D-20
  w (write) D-23

buffer 1-5, 2-1
  see also moving text with buffers

buffers, creating and changing 5-8

C

c (change) 3-20, D-6

carriage return 1-2

character deletion 1-3

character string 3-9

characters
  correcting mistyped 1-3

command level 1-3

context addressing 3-9

correcting typing errors 1-3

creating text
  a (append) 2-2

current line 3-2

D

d (delete) 3-18, D-18

default, in addressing 3-19

delete command 3-23

deleting lines
  d (delete) 3-18

deleting segments
  delete command 3-23

dprint command 3-24

E

e (execute) 5-2

edit existing document 3-1
  r (read) 3-1

edit mode
  see operation modes

editing examples 4-3

entering text 1-11

entering the editor 1-5

error messages
  descriptions of E-1

errors
  addressing C-7
  typographical 1-3

example with input and edit requests 4-5

exiting qedx 1-5
  q (quit) 1-5, 2-6

G

g (global) 5-4

g= (global print line number) 5-4

gd (global delete) 5-5

gp (global print) 5-4

H

helpful hints for new qedx users 4-6

I

i (insert) 3-22, D-7

implicit print request 3-4, D-14

input mode
  a (append) 1-6
  see operation modes

input requests vs. edit requests 1-6

input terminator 1-6, 2-2

inserting text
  i (insert) 3-22

interrupting print request 3-8

invoking qedx 1-5

L

line deletion 1-3

line number
  absolute 3-3
  relative 3-6

list command 3-22

listing segments
  list command 3-22

locate and print
  see implicit print request

locating and printing lines 3-1

locating lines
  addressing 3-2
    absolute line number 3-3, C-1
      range 3-5
    compound 3-14, C-4
    context 3-9, C-2
    implicit print request 3-4
    relative line number 3-6, C-2
    series of lines C-5
  current line 3-2
    pointer 3-2

logging in 1-1
  login command 1-2

logging out
  logout command 1-3

M

m (move) 5-8

macros
  see use of editor macros

moving lines (cut and paste) 5-8

moving text with buffers 5-12

N

n (nothing) 5-10

naming conventions 2-4

newline 1-2

null regular expression 3-13

O

operation modes 1-6
  input vs. edit 1-6
    edit 1-8
    input 1-7

P

p (print) D-12

password 1-2

pointer 3-2

print command 3-22

print entire buffer 3-8

printing certain lines
  $ (dollar sign) 3-6
  = (equal) 3-6

printing segments
  dprint command 3-24
  print command 3-22

Q

q (quit) 2-6, D-25

qedx command description B-1

R

r (read) D-9

reading segments
  r (read) 3-1

ready message 1-3

regular expression
  null 3-13
  special characters in 3-10

regular expression (REGEXP) 3-9

relative line number 3-6

repeated editor sequences 5-15

replacing text
  c (change) 3-20

request descriptions D-1

requests 1-5

response from qedx 1-7

S

s (substitute) 3-15, D-20
  delimiters 3-16

sample invocation 4-1

sample terminal session 4-1

saving your work
  w (write) 2-3

search string 3-16

special characters 3-10
  in user input
    # 1-3
    @ 1-3

special escape sequences 5-11

string
  search 3-16
  substitution 3-16

substitute request 3-15

substitution string 3-16

T

typing errors, correcting 1-3

U

use of editor macros 5-16
  macro initialization 5-19
  notes on macro use 5-23

V

v (exclude) 5-5

v= (global print line number)
5-6

vd (exclude delete) 5-6

vp (exclude print) 5-6

W

w (write) 2-3, D-23

white space 1-4

X

x (buffer status) 5-9

CUT ALONG LINE

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form

TITLE: MULTICS
qedx TEXT EDITOR
USER'S GUIDE

ORDER NO. CG40-01

DATED FEBRUARY 1983

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

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

Honeywell

CUT ALONG LINE

FOLD ALONG LINE

FOLD ALONG LINE

Honeywell

Honeywell Information Systems
In the U.S.A.: 200 Smith Street, MS 486, Waltham, Massachusetts 02154
In Canada: 155 Gordon Baker Road, Willowdale, Ontario M2H 3N7
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.

36283, 183, Printed in U.S.A.

CG40-01