Technical Documentation for the Canon Cat Editor

By the Staff of
Information Appliance, Inc.

Copyright Information

Canon Cat by Jef Raskin, Dr. James Winter, Terry Holmes, Minoru Taoyama, Jonathan Sand, John Bumgarner, Paul Baker, Jim Straus, Dave Boulton, Charlie Springer, Scott Kim, Ralph Voorhees, Richard Kraus, Kouji Fukunaga, Kazuhiro Nakamura, Naohisa Suzuki, Shigeru Ishida, Susumu Takase.

Manual by David Alzofon, Lori Chavez, Jim Winter, David Caulkins, Terry Holmes, Minoru Taoyama, Jonathan Sand, John Bumgarner, Scott Kim.

About This Manual

This manual includes:
— an overview of how each part of the Canon Cat software works
— a detailed step-by-step explanation of routines and data structures
— a summary of routines and integers

To understand the Cat software, you will also need the tForth Manual.

The software was written in the programming language tForth, a version of Forth developed at Information Appliance. The software was developed on early models of the Cat itself. In the production model, the software is stored in ROM.

Important: This editor features a user interface designed for speed and ease of use. All modifications and extensions must conform to its design principles and style in order to maintain user interface integrity.

System Overview

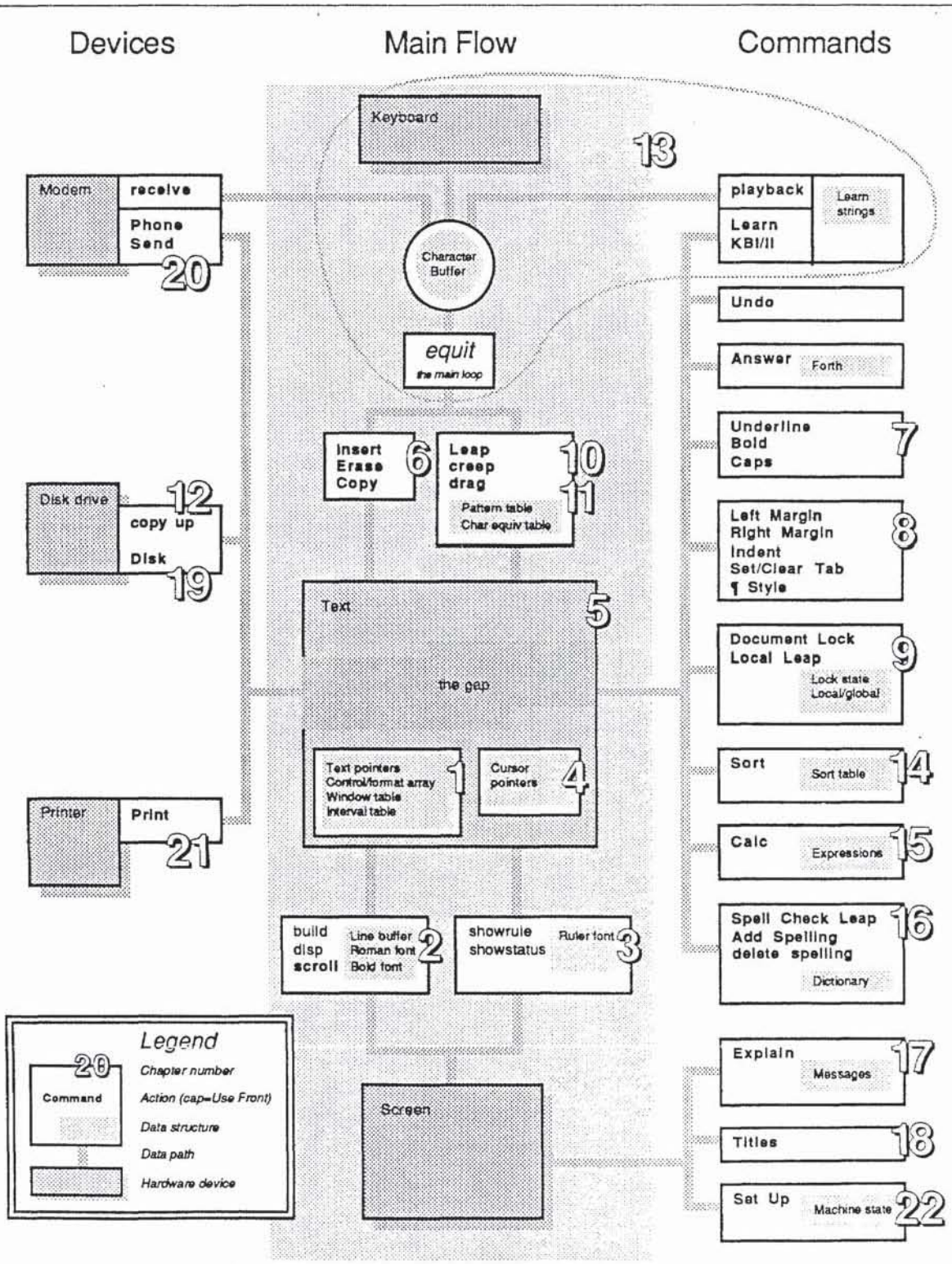


Table of Contents

COMPLETE TABLE OF CONTENTS
for
TECHNICAL DOCUMENTATION FOR THE CANON CAT EDITOR

PART II. TEXT COMMANDS (PARTS 7-12)

PART III. SPECIAL COMMANDS (PARTS 13-22)

HOW TO INTEGRATE SOFTWARE INTO THE CANON CAT

Products designed by Information Appliance Inc. (IAI), such as the Canon Cat, have a number of unique features. One of them that directly affects third-party software development is the principle of editor-based software.

In most microprocessor-based products, the user shifts between applications by returning to the operating system, indicated by a menu with a number of choices (or, equivalently, a window with a number of icons.) Then the user chooses the next application. Once having entered the application, the user gets the data on which to work.

In an IAI interface, the data stays in place at all times so that the user can concentrate on content rather than on the system. As commands are given, different "applications" come to bear on the user's text or graphics (there are graphics primitives in the Cat, although the built-in software does not use them). This is possible due to our unified data structure which is -- all at the same time -- a text, a data base, a spreadsheet, and a programming environment.

The user has a much simpler mental model with the IAI interface than with traditional products, since invoking an application looks just like another simple editor command. The user does not have to work with a number of different editors, one for each application. This is an improvement over the Macintosh, for example, in that with the Macintosh model each application must recreate (using provided routines) an interface that is similar to that of other applications.

When developing new applications for the Cat, it is easiest, both on the programmer and the user, to make your application look just like the existing built-in software. When your application needs to get information from the user, it generally asks a question. This can be done by sending the question to the screen, perhaps surrounded by a few blank lines so that it is visible. If the user finds that the question has come out in an awkward place (say, in the middle of a letter), then the user can always delete the question or move it elsewhere.

A typical question for an accounting package might be:

Name of account?

When this appears, the application should wait for a response to be sent to it by the ANSWER command (USE FRONT-ERASE). Thus the user is free to employ any and all the features of the Cat in creating the answer, for example, they might leap to their account area, or even change disks or perform a calculation to find the information they need. The idea here is to leave the full power of the Cat available at all times.

When the user has formulated the answer to the question your application has asked, they highlight it and use the ANSWER command. At this point, your application is in control again and can do what it wishes until it asks its next question.

This "loss of control" after a question has been asked will disturb some designers who are used to a forcefully directed dialog with the user. However, research has shown that users work better if they can do tasks at their own speed, and if they are in control. There is nothing more annoying than a program that demands an answer and won't let you use the system (say for looking up a phone number you need **right now**) until you are finished answering the computer's question -- a task that might take a few minutes if you have to look up something that's in a file cabinet somewhere.

One secret of the Cat's utility is that all abilities are available simultaneously and instantaneously. If your application has a number of features or areas, then allow the user to create a message which activates them when desired (the messages sent to your application via the ANSWER command, of course. One set of messages might be: "AR" to activate the accounts receivable package, "AP" to activate the accounts payable package, and "GL" to run the general ledger package. Once in any of these packages, the dialog would work as already described.

Notice that you do not have to write any I/O editing routines. You can simply send strings to the screen, and receive strings (edited by the user). Naturally, your application may need to do error checking, but when an error is detected, you can just send a string to the screen with the message, the user can edit their previous response using the Cat's built-in editor, and resend it to your application.

Following this protocol will keep the Cat feeling like a Cat, and will be least disruptive to a user's habits. It is also very easy and quick to create application interfaces this way.

Jef Raskin
13 September 1988

ENABLING FORTH IN THE CAT

Forth is normally hidden away, inaccessible in the Cat. However, with a simple incantation you can "enable Forth," making it possible to switch from the Cat's editor to a Forth programming environment, or to run Forth programs from the Cat's editor with the ANSWER command. Forth enablement is associated with a given disk and text. If you enable Forth, record the text, change to a non-enabled disk, then Forth will no longer be enabled.

Remember to exercise caution whenever Forth has been enabled. For example, a nonprogrammer may be trapped in Forth if they accidentally press the key combination SHIFT-USE FRONT-SPACE BAR while editing the text on a Forth-enabled disk. The key combination USE FRONT-SEMI-COLON will erase the disk in the drive if Forth is enabled. Other pitfalls exist. SO, PROCEED WITH CAUTION IF YOU ENABLE FORTH. READ THE DISCLAIMER AT THE BEGINNING OF THE MANUAL.

HOW TO TURN ON FORTH

We will now explain how to turn on Forth, and, equally important, how to turn it off:

1. To turn on Forth in a Cat, type the following phrase (be sure to capitalize "E", "F", and "L"):

   Enable Forth Language

2. Highlight these three words.

3. Hold down the USE FRONT key and, while holding it, tap the ANSWER key (ERASE). Then let go. This executes the ANSWER command, enabling Forth. You are not yet in Forth.

4. Now hold down the USE FRONT key AND the SHIFT key, and, while holding BOTH keys, tap the SPACE BAR. You are now in the Cat's Forth editor.

5. Type the following and press the RETURN key (the letters will automatically appear in boldface):

   -1 wheel! savesetup re

   This step allows you to enter Forth simply by pressing SHIFT-USE FRONT-SPACE BAR from now on.

   To enable easy access to Forth with Step 4 only, make some change to a Setup parameter, then use the DISK command. This will save the Forth enabling information on the disk. Whenever you play back this disk, you can then enter Forth using only the procedure of Step 4.

6. To turn off Forth, type the following and press RETURN key:

   Forth? off 0 wheel! re

   Make some change to a Setup parameter, then use the DISK command. This restores the Cat to normal operation, meaning that you will have to start over at step 1 again to invoke Forth. Normal Cat users will not be trapped in Forth in case they happen to accidentally press SHIFT-USE FRONT-SPACE BAR.

TALKING TO tFORTH

tForth is hiding in the background of every Cat system. It is very easy and convenient to communicate with tForth from within the editing environment.

SENDING COMMANDS TO tFORTH

Once Forth has been enabled (see the previous page), commands and programs can be sent to tForth from the editor by highlighting the desired command string or program listing and pressing [ERASE] while holding the [USE FRONT] key down. tForth's responses will be printed out in the editor.

All examples in this manual are expected to be typed into the editor and "sent" to tForth in this manner. All examples presented are set off from the body of the text by two blank lines and are indented:

```
3 dup . . 3 3
```

A section of the above example was underlined. In an example, the underlined sections are the sections of the text which should be highlighted and passed to tForth by pressing the [USE FRONT][RETURN] key combination. After the above example was sent to tForth, tForth responded by printing two 3's on the screen.

USING THE CALC COMMAND TO TALK TO tFORTH

Commands and programs can also be sent to tForth with the use of the [USE FRONT] [CALC] key combination. When this method is used, all command strings or program listings sent to tForth must be preceded by a "]" character:

```
]3 dup . . 3 3
```

The above example produced the same results as the [USE FRONT] [RETURN] example. The [USE FRONT][CALC] method is not used in this manual.

ERRORS

The [USE FRONT][RETURN] is used to let Forth know it should start 'processing' any highlighted words. If Forth ever has a problem processing an input, a beep will be issued. To see the error message press the [EXPLAIN] key while holding the [USE FRONT] key down. For example, if tForth is sent the following input:

  How now brown cow?

it will beep and [USE FRONT][EXPLAIN] will reveal a "can't use" message. This is the error message which occurs when tForth is sent a command it does not recognize.

CAUTION: ALWAYS RECORD YOUR EDITOR TEXT ON DISK BEFORE DIRECT EXECUTION OF tFORTH WORDS. IT IS VERY EASY TO MAKE PROGRAMMING MISTAKES WHICH COULD PERMANENTLY DAMAGE THE DOCUMENT.

A BRIEF INTRODUCTION TO FORTH

The Forth language is comprised of many "words" (commands). This collection of words is referred to as the "Forth dictionary." The tForth dictionary contains approximately 600 words. The list below shows a few Forth words and the actions they perform:

emit	Takes a number and displays the corresponding ASCII character on the screen.
+	Adds two numbers together and returns the result.
words	Produces a listing of all available words.
if then	Words used to implement the IF...THEN program control construct.
@	Fetches a 32-bit value from memory.

As the list shows, a Forth word can either have the format of a "normal" word (a sequence of letters), or it can be a punctuation mark, a sequence of punctuation marks, or a mixture of punctuation marks and characters. In a Forth program, all words must be separated from each other by at least one space, tab, or carriage return. In this document Forth commands will be shown in boldface. For example:

"The Forth word **words** will produce a listing of all available words."

Note: tForth is case-sensitive. This means that tForth thinks a capital W is different than a lowercase w. Thus tForth will think **Words** is a different command than **words**.

If the pronunciation of a Forth word is unclear, it's first usage in the text will be followed by the natural language pronunciation enclosed in quotes and parentheses. For example:

> To take a number off of the parameter stack and display it, use the word . ("dot").

EXECUTING A FORTH WORD

Most of the words in the Forth dictionary may be executed directly and immediately, from the keyboard. The example below shows how the Forth word **emit** could be used to display an asterisk character on the screen. In the example, the underlined type is used to indicate which commands should be highlighted and sent to tForth. The normal type is used to show Forth's responses.

ERRORS

The [USE FRONT][RETURN] is used to let Forth know it should start 'processing' any highlighted words. If Forth ever has a problem processing an input, a beep will be issued. To see the error message press the [EXPLAIN] key while holding the [USE FRONT] key down. For example, if tForth is sent the following input:

How now brown cow?

it will beep and [USE FRONT][EXPLAIN] will reveal a "can't use" message. This is the error message which occurs when tForth is sent a command it does not recognize.

CAUTION: ALWAYS RECORD YOUR EDITOR TEXT ON DISK BEFORE DIRECT EXECUTION OF tFORTH WORDS. IT IS VERY EASY TO MAKE PROGRAMMING MISTAKES WHICH COULD PERMANENTLY DAMAGE THE DOCUMENT.

Note: Do not confuse the underlined commands in the examples with the underlined Forth words in the text. In the examples the underlined commands are those commands which should be highlighted and sent to tForth with the ANSWER command.

```
42 emit *
```

emit , as was described above, is a Forth word which will display the character which corresponds to the ASCII value passed to it.

COMPILING FORTH WORDS

The interactive execution of **emit** in the previous example did not cause any code to compiled. The Forth word : ("colon") is used to turn the Forth compiler on:

```
: printstar 42 emit ;
```

The above example shows how a new word may be added to the Forth dictionary. The word which immediately follows : (**printstar** in the above example) is the name which will be assigned to the new word. The Forth words following the name and preceding the ; will be compiled into the new definition; these are the words which define the actions of the new word. Since the action words for **printstar** are 42 **emit**, **printstar** will print an asterisk when executed. The word ; ("semi-colon") is used to turn the compiler off and return to the interactive execution mode.

Note that in this example, sending the input to Forth did not cause the asterisk to be displayed. Since the Forth compiler was "on" when the "42 **emit**" was typed, the 42 **emit** was compiled rather than executed. Forth was able to successfully compile the new definition so no error beep was issued. Forth is an "incremental compiler"; code is compiled definition by definition; compilation is triggered by each reception of a line of input.

THE FORTH PARAMETER STACK

Forth is a stack-based language. Any Forth word which takes an input will expect to find its input parameter on the Forth parameter stack when it executes. Any Forth word which returns a value will leave the value on the parameter stack when it completes execution.

The parameter stack, and stacks in general, are functionally similar to the spring-loaded stack of plates which can be found at most institutional kitchens. Whenever a plate is taken from the stack, it is always taken from the top of the stack of plates. Whenever a plate is added to the stack, it is always added to the top of the stack of plates. A person who does not want the steaming hot plate on top of the stack must remove the top plate before the second plate can be accessed. If no plates are available, the stack is empty.

The Forth parameter stack works the same way as the stack of plates, except the Forth parameter stack is set up to hold numeric values rather than plates. Also, just as the kitchen stack was designed for a certain plate size, the Forth parameter stack is designed for a certain numeric value size (the plate size of the tForth parameter stack will be discussed later).

INTERACTING WITH THE PARAMETER STACK

To put a number on the parameter stack, send the number to Forth:

```
34
```

To take a number off the parameter stack, use the word **drop**. To take a number off the parameter stack and display it, use the word . ("dot"):

```
. 34
```

To place more than one number at a time on the stack, send the numbers, separated from each other by a space or spaces (so that Forth knows they are distinct numbers), to Forth:

```
3 6 8
```

Now there are three numbers on the stack. If . is used, it will take the top number off the stack and display it. Since the 8 was the last value placed on the stack, it will be the top value on the stack:

```
. 8
```

To place more than one number on the stack at a time, the numbers were separated by spaces and sent to Forth. This is the same way Forth commands (words) work. To take both of the remaining numbers off the stack, the word . can be used twice on the same line:

```
. . 6 3
```

Forth's response should be read left to right. The 6 is the result of the first use of . The 3 is the result of the second use of .

Note what happens if . is used again:

```
. 0
```

You should hear a beep as . tried to remove a value from an empty stack and Forth responded by displaying a zero, beeping and issuing a "stack is empty" error message.

PASSING PARAMETERS TO FORTH WORDS ON THE STACK

Many Forth words take input parameters from the stack and return results on the stack. The Forth word + ("plus") is a good example of such a word:

```
5 4 + . 9
```

+ takes two numbers from the stack (the 5 and the 4 in the above example), adds them together and returns the single number result on the stack. In the example, . was use to display the result returned by +

SUMMARY

* Forth programs are developed by creating new words out of previously existing words.

* The parameter stack is the primary means of communication among Forth words.

* The Forth language does not have many syntax requirements. This gives the experienced programmer great control over the computer but can make it difficult for beginning programmers to locate mistakes.

* The interactive abilities of Forth make it a hard-to-beat debugging environment. Each word can be tested individually and interactively.

This is the end of our brief introduction to the Forth language. For more introductory Forth reading, refer to the first chapter of Starting Forth, by Leo Brodie (Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, 1981).

CONVENTIONS

Convention	Meaning	Example	Meaning of Example
Boldface	Forth	**LBPfix**	The Forth word "LBPfix"
Underline	Variable	n choices	An as yet unspecified number of choices, to be fixed when the given operation is carried out.
$	Hex	$4E	"4E" is a hex number.

1. POINTERS AND DATA STRUCTURES

Introduction

The data maintained by the Cat editor is kept in an area called the text. The addresses of important areas or locations in the text are kept in many system integers. The basic data structure used to hold formatting information about the text is called a control/format array. The three main data structures used to maintain the text, the **#ctrl** array, the window table, and the interval table are each comprised of one or more control/format arrays.

1.0 THE TEXT

1.0.0 What It Is

The text contains all of the characters, calculations, and formatting information the user enters into the editing environment. The text contains several types of organized data:

- ASCII character codes
- character attribute information
- paragraph format information
- document format information
- calculation data

The editor's only function is to alter, maintain, and display this data. This section of the manual will discuss the lower level constructs which allow the editor to function properly and quickly.

To manage the text within its allocated area of memory, the editor relies upon many system integers which contain pointers to key locations in the text. Examples of key locations are

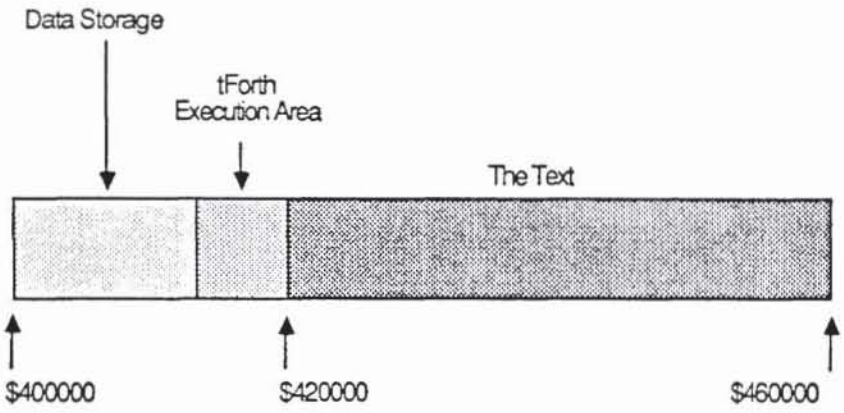
- where the text starts and ends
- which character in the text the cursor is currently over
- where new characters typed in by the user should be inserted.

The editor data structures give meaning to the text data. The fields in the data structures give the editor information on how the character data should be displayed.

1.0.1 Where It Is

The text is located in the Cat's RAM space. The current Cat system has 384K of RAM located starting at address $400,000. The following diagram (1.1) shows where in the RAM space the text is located.

1.1 Cat RAM Allocations



1.1 POINTERS USED TO MAINTAIN THE TEXT

The following diagram (1.2) shows a close-up of the text area and describes some of the pointers used to maintain the RAM text area.

1.1.0 The Beginning of the Text Area

The **text** system integer holds the address of the very first byte of text data in the first text partition. The **bot**, or beginning-of-text, system integer holds the address of the first byte of the user's text data. The memory between the two addresses is used as a buffer zone between the start of the allocated text area and the start of the actual text data. The start buffer zone is filled with eight carriage return characters.

1.1.1 The Start of the Gap Region

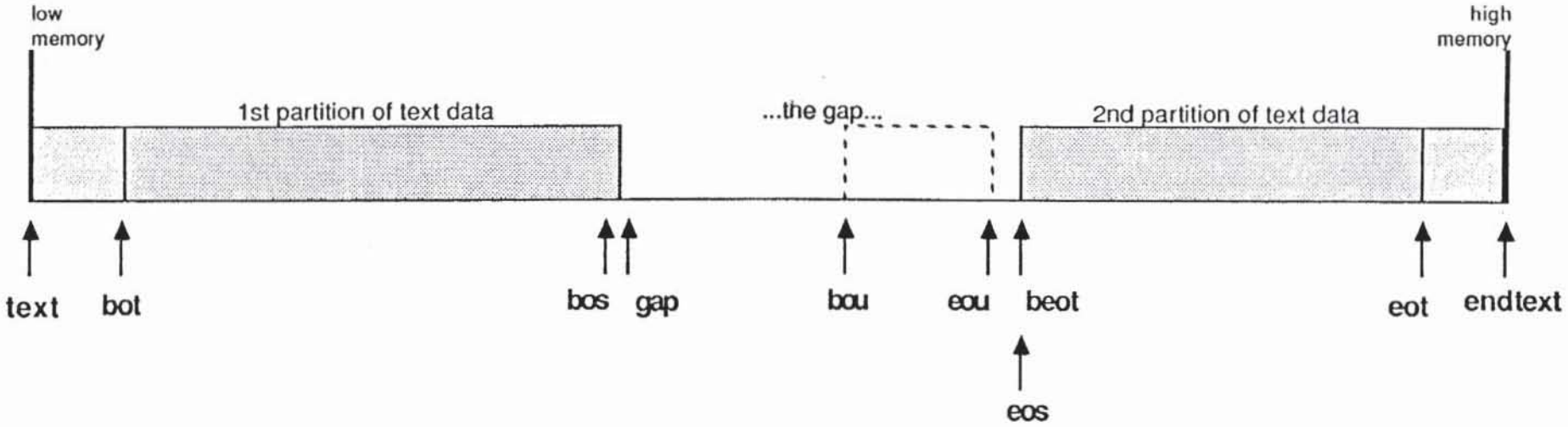
The text data is broken into two pieces. Sandwiched between them is a region of memory referred to as the gap. All unused space allocated to text is located in the gap.

The **gap** system integer holds the address of the start of the gap. The existence and location of the gap help contribute to the quick response of the editor in many situations. For example, the start of the gap is usually adjacent to the character in the text which is marked with the cursor. This means new characters typed in can be placed directly into the gap region and instantly appended to the end of the first partition of text without any movement of the rest of the text data.

The **bos**, or beginning-of-selection, system integer holds the address of the first character in the current selection. If the selection is extended, that is, if more than one character is highlighted, **bos** will hold the address of the first highlighted character in the selection. If the selection is not extended (one character highlighted), and if the cursor is wide, **bos** will hold the address of the character seen in the non-blinking half of the cursor. When the cursor is narrow, the **bos** will hold the address of the character under the narrow cursor. When the selection is not extended the **bos** will point to the address of the last character in the first partition of text data, that is, the **bos** pointer and the **gap** pointer will be right next to each other (see the following diagram, 1.3).

1.2 The Text

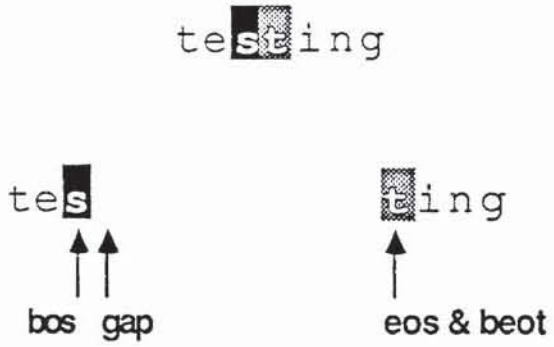
...and the pointers used to maintain it.



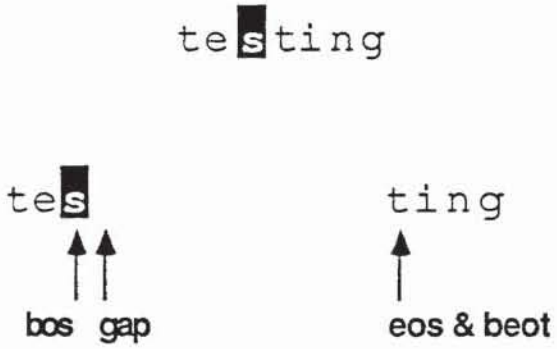
integer name		integer contents
text	:	address of the absolute start of the RAM text area.
bot	:	address of start of user text data.
bos	:	address of beginning of selection.
gap	:	address just beyond end of first partition of text.
bou	:	address of the beginning of the undo buffer.
eou	:	address of the end of the undo buffer.
eos	:	address of the end of selection.
beot	:	address of the beginning of the end (2nd partition) of text.
eot	:	address of the end of the user text data.
endtext	:	address of the absolute end of the RAM text area.

1.3 Cursor Logistics

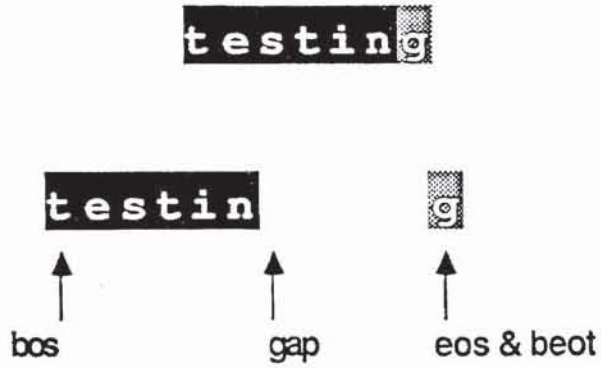
Wide cursor:



Narrow cursor:



Extended selection:



1.1.2 The Second Partition of Text Data

The start address of the second text partition is held in the **beot**, or beginning-of-end-of-text, and **eos**, or end-of-selection, system integers. If the cursor is wide, **eos** will hold the address of the character under the blinking portion of the cursor. If the cursor is narrow, **eos** will hold the address of the character which immediately follows the narrow cursor. The words **eos** and **beot** hold the same address except while a Leap key is down and a successful leap is in progress; in other words while the cursor is no longer where it was when the Leap key was first held down (see the following diagram, 1.3).

The **eot**, or end-of-text, system integer holds the address of the last byte of user text data in the second text partition. The **endtext** system integer holds the address of the last memory location in the RAM text area. The area between the **eot** and **endtext** pointers forms a buffer area between the end of the user's text and the absolute end of the RAM text area. The end text buffer holds 30 carriage return characters.

1.1.3 The Undo, or "Cut" Buffer

The last area of interest in the RAM text area is the undo buffer area. The undo buffer is located in the gap region. The start address of the undo buffer is kept in the **bou**, or beginning-of-undo-buffer, system integer. The **eou**, or end-of-undo-buffer, which is defined as

```
: eou ( -> a )  beot 4 - ;
```

is used to find the end of the undo buffer. The undo buffer is relocated whenever the **eos** position changes, that is, after a successful leap or creep. The undo buffer holds information (text, formatting info) required in order to undo an operation. Whenever information needs to be placed in the undo buffer the **bou** pointer is repositioned so that the undo buffer becomes just large enough to hold the desired information.

1.2 CONTROL/FORMAT ARRAY

The control/format array is the basic structural unit used in the Cat editor. A single control/format array holds **esize** bytes of formatting information pertaining to a particular line in the text. The three main data structures used by the editor (described below) each consist of one or more control/format arrays.

Three basic types of formatting information are kept in the control/format array:

- transient format information
- paragraph format information
- document format information

The structure of a control/format array is shown in the following diagram (1.4). The names of the fields, their hexadecimal array offsets (in bytes), and a description of their contents are listed below.

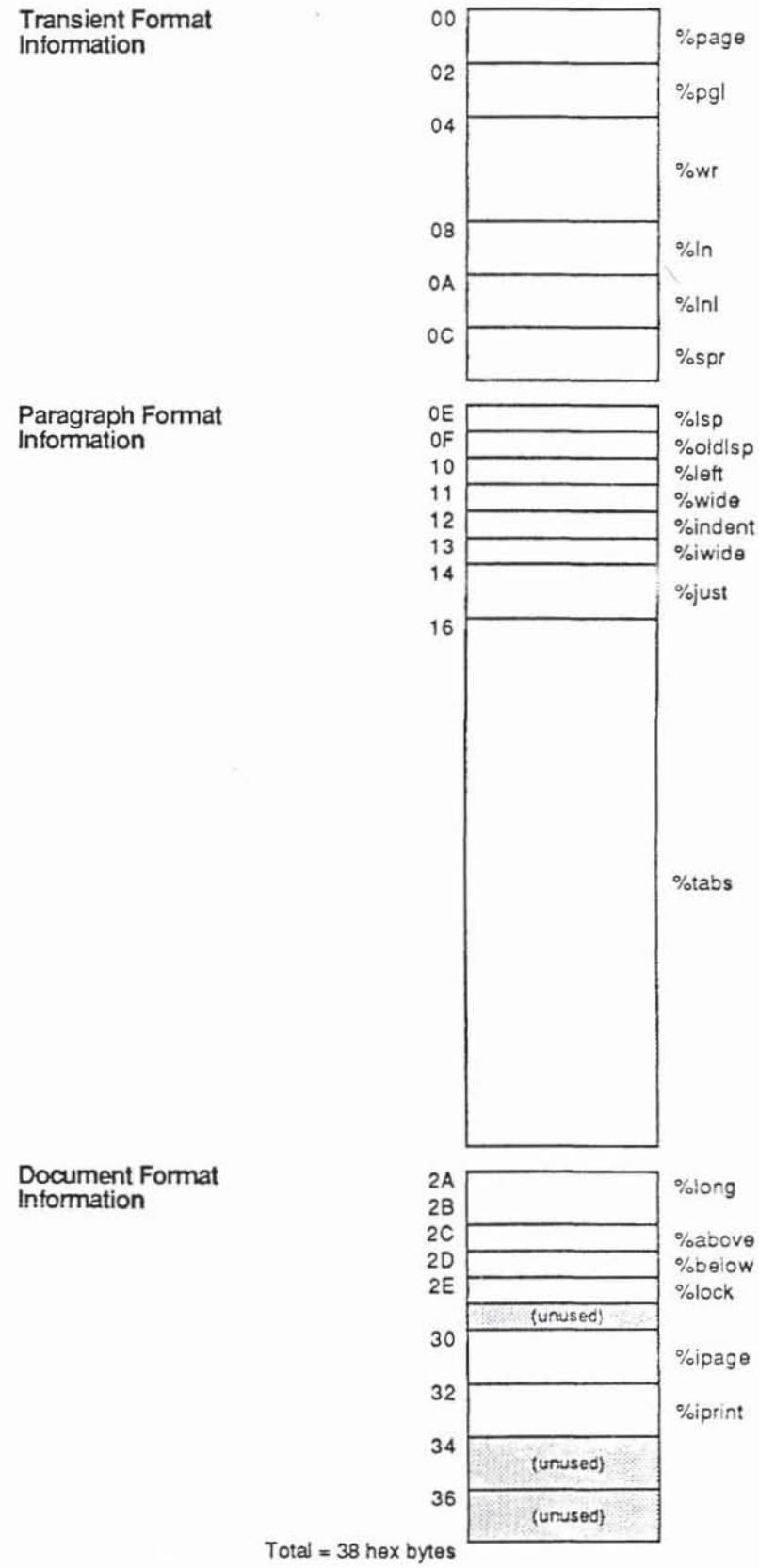
1.2.0 Transient Format Information

Transient format information is volatile format information which must be "calculated" each time it is requested. The transient information in the control/format array can become obsolete as a result of a single character insertion or deletion. If a character insertion/deletion moves the cursor to a different line or page, the line and line start information, or page information will immediately become invalid. The contents of most of the transient format information fields are described sufficiently below. The use of the **%spr** field is explained in the section on text display. "Global" means relative the the **bot**.

Name	Offset	Description
%page	00	The global page number in which this line is located
%pgl	04	The local page number (within the current document) in which this line is located
%wr	08	Address of the first character in this line
%ln	0C	The global line number for this line
%lnl	10	The local line number (within the current page) for this line
%spr	12	Can hold one of four values: 0, 1, 2, 3. Used by the words responsible for displaying lines of text

0: Display a real line of text
1: Do nothing
2: Display one blank half-line
3: Display one blank half-line

1.4 Control/Format Array



1.2.1 Paragraph Format Information

The paragraph format information fields hold values which control how the characters in the paragraph should be placed on the screen when they are displayed. A paragraph is a section of text surrounded by a break, that is, a carriage return, page break, or document break. For example, the **%left** field holds the width of the left margin, expressed in units of half spaces. The display routines will use the contents of the **%left** field when they need to determine where the first character on a line should be placed.

Name	Offset	Description
%lsp	14	Local line spacing. Can hold one of three values: 2, 3, 4. Used by the words responsible for displaying lines of text 2: Single-spaced text 3: 1½ spaced text 4: Double-spaced text
%left	15	Current left margin width, expressed in half spaces. 0 <= n <= 158
%wide	16	Width of the text area, expressed in half spaces. 2 <= n <= 160
%indent	17	Indent distance for this line, expressed in half spaces. 0 <= n <= 158
%iwide	18	The width of the text area on an indented line
%just	19	Paragraph Style 0: Normal, left-justified, ragged right 1: Right-justified, ragged left 2: Centered, ragged left and right 3: Justified left and right
%tabs	1A	Two 80-bit bit arrays. The screen is 80 full spaces wide. The state of each bit in the first bit array indicates whether the corresponding space on the screen has a tab associated with it. The second bit array indicates whether the corresponding space has a decimal tab associated with it.

1.2.2 Units Used in the Control/Format Array

1.2.2.0 Vertical Positioning Units

All vertical positioning of text is based on the unit of a half-line. Half-lines are 1/2 the thickness of a line of text. The Cat editor supports three types of line spacing: single, 1½, and double. Depending on the line spacing currently selected, zero, one, or two half-lines will be inserted between each displayed line.

For example, in single-spaced text each line of text immediately follows the previous line of text; no half-lines are used in the display. In 1½-spaced text, one half-line is inserted between each line of text. In double-spaced text, each line of text is followed by two blank half-lines.

Although the text may not appear to include half-lines -- single spaced text, for example -- the code always counts in half-lines when calculating positions in text. This provides fast access to display information about any point in text, which speeds up leaping.

The Cat can display 22 lines of text at one time. Since each line of text is two half-lines wide, the screen can hold 44 half-lines.

1.2.2.1 Horizontal Positioning Units

Spaces and half-spaces determine the horizontal position of characters. Since the Cat editor uses a non-proportional font, each character in the character set has the same width, 8 pixels. Thus a half-space is 4 pixels wide. Half-spaces are often inserted into in order to fully justify text.

1.2.3 Document Format Information

The document format information fields contain information about the document in which the line is located. Several of the fields (**%above**, **%below**, **%iprint**) hold information which will determine the printed appearance of the document. The contents of the **%lock** field indicate whether or not the line is alterable. If the document which contains the line is locked, the line cannot be altered.

Name	Offset	Description
%long	2E	The value stored in **%long** indicates how long a page should be before the Cat inserts an implicit page break. The page length is expressed in half-lines.
%above	30	Holds the height of the top margin of a printed page expressed in half-lines
%below	31	Holds the height of the bottom margin expressed in half-lines
%lock	32	If this line is in a locked region, **%lock** will hold the ASCII value for the gray lock character.
%ipage	34	First page number in the document
%iprint	36	Number of the first printable page in the document

1.3 MAJOR DATA STRUCTURES

1.3.0 The Control Table

The control table (**#ctrl**) consists of one control/format array. The address of the control table is kept in the **#ctrl** system integer. The **#ctrl** is used as a scratch control/format array by routines which need a temporary location for the storage of control/format information. Control/format information which must be saved for future reference is usually moved from the **#ctrl** table to one of the other data structures described below.

A backup control table, called the previous control table (**#pctrl**), holds the previous contents of the **#ctrl** table. The system integer **#pctrl** holds the address of the control/format array for the preceding text line. This value is updated each time the word **wrap** is executed.

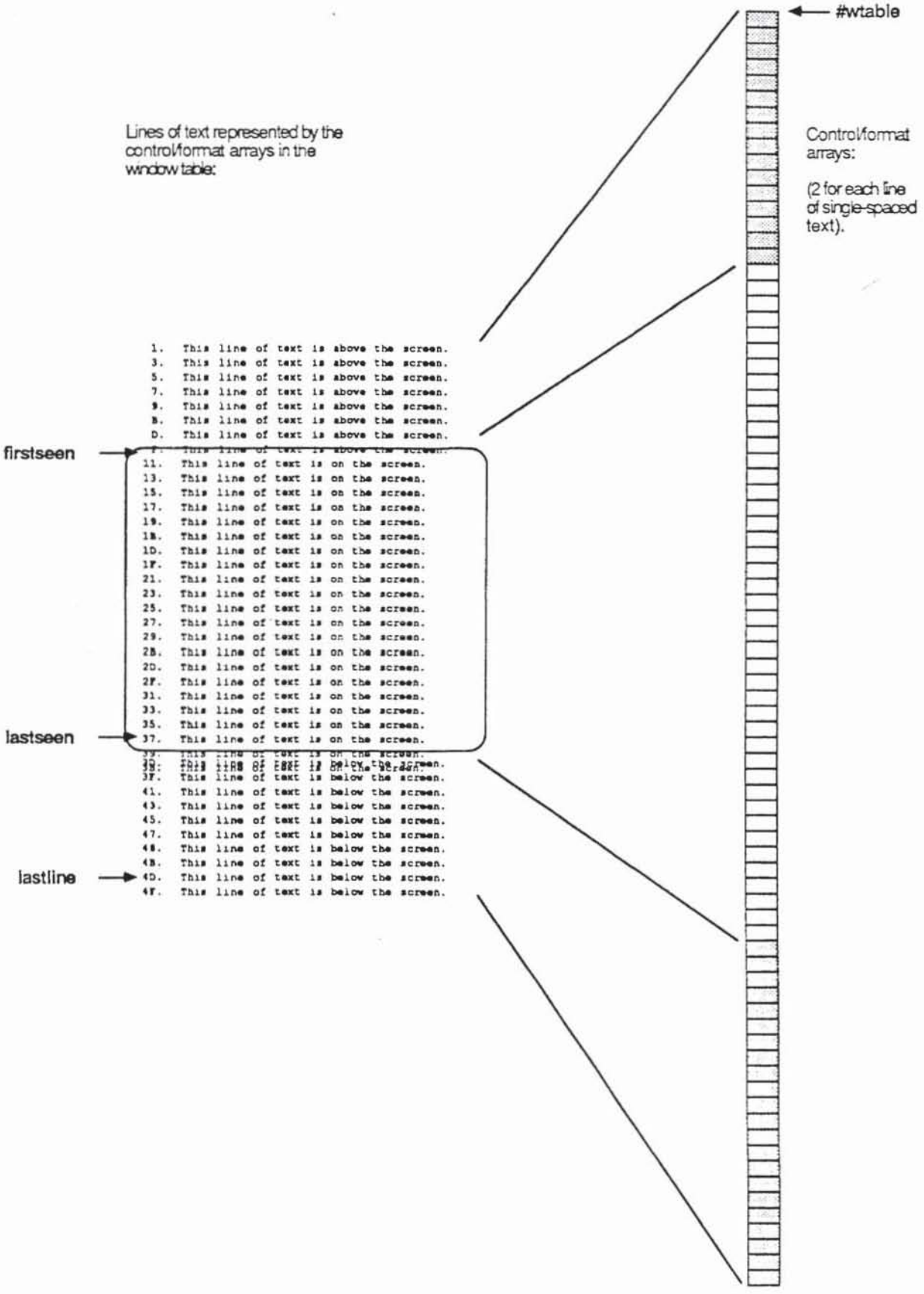
1.3.1 The Window Table

The window table (**#wtable**)consists of **lastline** ($4E) control/format arrays. Each control/format array in the window table contains formatting information about one half-line currently displayed on the screen and about the half-lines just above or just below the top or bottom lines in the display. The following diagram (1.5) illustrates the connection between the window table and the text displayed on the screen.

The 79 control/format arrays in the window table are shown on the right side of the diagram. The display text to which the control/format arrays correspond are shown on the left side of the diagram. The number to the left of a displayed line of text is the number of the window table entry corresponding to the line of text. The numbers increment by two because two window table entries are required to represent lines of single-spaced text. (The display of single-, 1½, and double-spaced text is covered in detail in the "Text Display" section of this manual.

The system integers **firstseen** ($10) and **lastseen** ($3B) hold the line numbers of the first and last visible half-lines represented in the window table. The system integer **lastline** holds the number of the last line represented in the window table. The address of the start of the window table is kept in the **#wtable** system integer. The bytes in the window table can be calculated by **lastline*esize:**.

1.5 The Window Table



1.3.2 The Update Array

The update array is closely related to the window table. The update array contains a 1-byte flag for each half-line represented in the window array. If a byte contains a non-zero value, the corresponding half-line in the display requires refreshing. If the byte contains a zero value, the corresponding half-line is properly displayed.

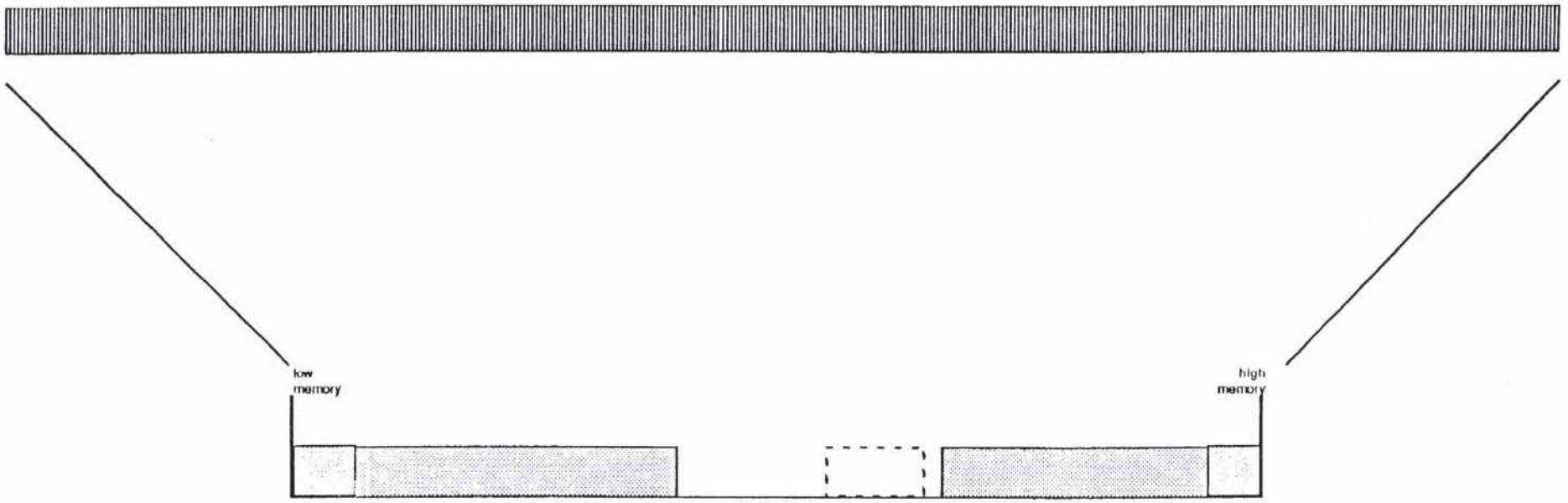
When display routines are called to redraw the screen contents, they will usually check the update array first so that only the lines which require refreshing are redrawn.

1.3.3 The Interval Table

To allow quick display of any character in the text, the text data is divided into many equal-sized text intervals. Formatting information about a line of text within each interval is kept in a table called the interval table (see diagram 1.6).

Currently the size of a single text interval is $400, or 1024 decimal bytes. Since the entire text size in the editor can be either 256K or 384K bytes, the interval table will have either 256 or 384 (decimal) entries. Each entry in the interval table is a control/format array which holds the formatting information about one line in the corresponding text interval.

1.6 Interval Table



1.3.3.0 How Control/Format Information Is Obtained

To get the formatting information which applies to a line in the text one must step through the text data which contains the characters for the line, looking for and obtaining the user-specified formatting information hidden among the characters (document and paragraph format information set by the following commands: Line Spacing, Left Margin, Right Margin, Set/Clear Tab, Indent, and Setup). This process is called wrapping through the text because the fundamental task is to determine implied word wrap.

wrap is the editor word which examines a line of text and produces a set of control/format information for the line. The word **wrap** always places its results in the **#ctrl** array. The word **wrap**'s default action is to step through one line of text only and to store the format information found into the **#ctrl** array.

The word **wrap** must always start from a location in the text where the format is known, for example, at the start of a line whose control/format information is known. The word **wrap** is supplied the information about this known line, its start address (found in the **%wr** field) and format information, to be held in the **#ctrl** array. When **wrap** finishes, the **#ctrl** array will contain a complete set of format information about the line which follows the line whose format information was passed to **wrap** in the **#ctrl** array. So, given information about a known line in the text, **wrap** will return control/format information about the following line in the text. These default actions of **wrap** were designed for stepping through the text, line by line, and producing control/format information.

The basic steps used by **wrap** are as follows:

- Use the information about the known line to quickly find the start address of the line which follows the known line. The line which follows is the line in which we are interested.

- Proceed forward from this new start position and look for text characters and format information.

- Each time a character is encountered, determine the width of the character and add it to a running total of the width of all characters on the current line.

- If format information is encountered, transfer it to the **#ctrl** array.

- If the total width ever exceeds the value of **#wide**, the end of the line has been found. The algorithm then moves back to the previous word break. The word before the last word break will become the last word on the line.

The word **wrap** looks in the system integer **wraplim** to decide when to stop wrapping through the text. Usually, **wraplim** contains a 0, which means **wrap** should only wrap the current text line. If **wraplim** contains a non-zero value, it gives the address at or beyond which wrapping should stop.

Before **wrap** overwrites the contents of the **#ctrl** with the newly found format information, it saves the analogous information for the previous half-line of text into the **#pctrl** array.

1.3.3.1 More on Intervals

Keeping the control/format information the editor data structures updated requires constant use of **wrap**. The word **wrap** will obtain information about a desired line much more quickly if it is supplied a preceding line position with known format which is very close to the target line. This is why the interval concept was developed. The interval table holds control/format information about many places in the text (see the previous interval table diagram). If the interval table is currently updated, one will never have to wrap more than one text interval in order to find information about any line of text contained in that interval.

1.3.3.2 How the Interval Table Is Used

When the editor starts, the entire text is word-wrapped to obtain control/format information about the first line of text in each text interval so that all of the entries in the interval table may be filled.

During the use of the editor, the contents of each text interval will change and the format information in the corresponding control/format entry in the interval table will become invalid. Operations which invalidate information in the interval table use the word **killivls** to mark the invalid intervals. Since updating the entire interval table after each editing operation would be prohibitively slow, the interval table is left in an incomplete state. A background task, updates information on intervals which need to be updated. The word **fixivl** is called each time a keystroke is not available. This procedure will cause all intervals to be correct within two seconds, worst case. Only certain operations, such as leaping, require the entire interval table to be updated for proper functioning.

The interval table is used to expedite the process of finding formatting information for selected locations within the text. The word **findchar** fills the **#ctrl** array with the format information for a specific character in the text. The word **findline** fills the **#ctrl** array with format information for a specific line in the text. The words **nextpage**, **prevpage**, **nextdoc**, and **prevdoc** cause the **#ctrl** array to be filled with the format information for the page or document which is before or after a specified address in the text. All of these words use the interval table to get format information about a spot close to their desired destination quickly. Then, **wrap** word wraps from the location of known format to the desired location in the text.

1.3.3.3. The Top Four Intervals

Four interval table entries have special significance to the editor:

#1	The first text interval
#2	The interval which contains the start of the gap
#3	The interval which contains the **beot**
#4	The last text interval (**?interval**, which contains **eot**)

It never changes, since the first line in the text cannot change (it always contains the starting document character). Thus the state at the start of the text is always known.

The **gapivl** system integer contains the address of the interval table entry corresponding to the text interval containing the start gap address. The **beotivl** system integer contains the address of the interval table entry corresponding to the text interval containing the **beot** address. These two intervals are complex intervals since they are separated by the gap and may or may not contain complete lines of text. The addresses of these interval table entries are saved for use by **fixivl**. If **fixivl** has problems fixing an interval table entry, and it notices the interval it is trying to fix is either **gapivl** or **beotivl**, it will skip over the interval and move on.

1.4 ROUTINES AFFECTING THE TEXT AND ITS POINTERS

1.4.0 Text Maintenance Routines

adjust (a1 a2 n ->)
(pronounced ah-just')
Adjusts all text pointers which point within the range between address a1 and a2 by the delta distance n. The pointers affected are: **gap**, **bou**, **beot**, **bos**, **eos**, **savebos**, **extbos**, **bot**, **bor**, **eot**, **eor**, **oldop**, **oldpop**, **oldbos**, **oldeos**, **oldeos2**, **oldbos2**, and **oldpop2**. Also uses **realign** to adjust the positions of the **op** and **pop** pointers and the contents of the window table.

clearundo (->)
(pronounced cleer' un-doo')
Empties the undo buffer by setting **bou** pointer equal to the **eou** pointer.

eou (-> a)
(pronounced ee' oh yu')
Initials stand for end-of-undo buffer. Calculates and returns the address of the end of the undo buffer.

realign (a1 a2 n ->)
(pronounced ree'ah-line')
Adjusts pointers and data structures which point into the range of text starting at address a1 and ending at address a2 by the offset n. The pointers affected by **realign** are **op** and **pop**. The word **realign** also affects all information in the window table entries.

selsize (-> n)
(pronounced sel' size)
Forth word composed from English words "selection-size." Calculates and returns the size in bytes n of the current selection. The equation **gap bos -** determines the selection size.

1.4.1 Window Table Routine

seenlines (-> n)
(pronounced seen' lines)
Returns the number n of half-lines visible on the display.

1.4.2 Update Table Routines

update! (n ->)
(pronounced up'date store')
Sets the update bit in the update array entry which corresponds to the specified screen line number n.

update? (n -> f)
(pronounced up'date kwes'chun)
Returns a true flag f if the update table entry corresponding to the specified line ?n in the window table indicates that the line requires updating. Also marks that line as no longer needing update.

1.4.3 Interval Routines

badivl (-> 0 | If no bad interval entry is found)
(-> a -1 | If a bad interval entry is found)
(pronounced bad' iv'il)
This word stands for "bad interval." Searches through the interval table looking for the first bad (not updated) interval. If a bad interval is found, returns the text address which corresponds to the interval and a true flag. If no bad interval is found, returns a false flag only.

fixivl (->)
(pronounced fix' iv'il)
Tries to fix one bad interval in the interval table.

goodivl (a1 -> a2)
(pronounced gud' iv'il)
Returns the address a2 of the closest up-to-date interval table entry which precedes to the specified text address a1.

hideivls (a1 a2 ->)
(pronounced hide iv'ils)
Marks all invalid intervals corresponding to text located within the specified address range (between a1 and a2) as potentially valid by clearing the high bit on the **%wr** field. A potentially valid interval is an interval whose control/format array has only incorrect page and line number information.

killivls (a1 a2 ->)
(pronounced kill' iv'ils)
Marks all intervals corresponding to text located within the specified address range (between a1 and a2) as invalid by setting **%wr** as equal to -1 in the corresponding control/format array.

knownplace (a ->)
(pronounced nown' plase)
Looks through the interval table to find the interval boundary closest to and prior to the desired text address a. Loads the control/format information which corresponds to the interval into the **#ctrl** array.

lastknownline (-> n)
(pronounced last' nown line')
Returns the line number of the last displayable line of text entered by the user.

line>ivl (n1 ->)
(pronounced line' to iv'il)
Looks through the interval table to find the interval boundary preceding the desired line number n1. Loads the control/format information corresponding to the interval into the **#ctrl** array.

nearinterval (a1 -> a2)
(pronounced neer' in'ter-vul)
Returns the address a2 of the nearest text location specified by the interval table which is closest to and prior to the address a1.

nearivl (a1 -> a2)
(pronounced neer' iv'il)
Returns the address a2 of the closest interval table entry which corresponds to the specified text address a1.

nextivl (-> a -1 | If a valid interval is found.)
(-> 0 | If a valid interval is not found.)
(pronounced nekst' iv'il)
Looks through the interval table to find the address a of the next valid interval table entry corresponding to an interval boundary address which is greater than the address found in the **%wr** field of the **#ctrl** array. If a valid interval is found, its interval table entry address and a true flag are returned. If no valid interval is found, a false flag is returned.

partknown (a ->)
(pronounced part' nown)
Uses **hideivls** to mark all text intervals between the address a and the end of text, **eot**, as partially changed.

previvl (-> a -1 | If a valid interval is found)
(-> 0 | If a valid interval is not found)
(pronounced preev' iv'il)
Looks through the interval table to find the address a of a previous valid interval table entry which corresponds to an interval boundary address preceding the address found in the %wr field of the **#ctrl** array. If a valid interval is found, its interval table entry address and a true flag are returned. If no valid interval is found, a false flag is returned.

putivl (-> f)
(pronounced put' iv'il)
Puts the control/format information found in the #ctrl array into corresponding interval table entry. The address in the **%wr** field finds the corresponding interval table entry. A true flag is returned if all interval table entries contain valid control/format information.

1.4.4 Wrap Routines

prevwrap (->)
(pronounced preev' rap)
Copies the contents of the previous control array, **#pctrl**, into the current control array, **#ctrl**.

wrap (->)
(pronounced rap')
Used to recalculate line, page, and document numbers. Performs one wrap of one half-line each time it is called unless a non-zero value is stored in **wraplimit**. Wraps the current line, that is, the line whose format information is stored in the current **#ctrl** array. If a non-zero value is stored in **wraplimit**, it is assumed to be the address at which wrapping should stop.

Since **wrap** may have to cross the **gap**, the skip characters (described in the "What's in the Text" section should be properly positioned on either side of the gap (that is, **preset** should be called before using **wrap**). **wrap** does not affect the appearance of the display, it only affects the contents of the current line's control/format array (**#ctrl**).

wrapthru (a ->)
(pronounced rap' thru)
Updates the interval table entries, starting with the entry nearest to the specified text address a, and ending at the line following the line containing a. Will fill in the interval table for all intermediate intervals in the range specified above.

1.4.5 Routines Which Get Specific Control/Format Information

findchar (a ->)
(pronounced find' kair)
Fills in the **#ctrl** array with the control/format information for the beginning of the line on which the character residing at the specified address a is located.

findline (n ->)
(pronounced find' line)
Fills in the **#ctrl** array with the control/format information for the specified line n.

nextdoc (a ->)
(pronounced nekst' dok)
Loads the **#ctrl** array with the control/format information corresponding to the first character in the document following a.

nextpage (a ->)
(pronounced nekst' page)
Loads the **#ctrl** array with the control/format information corresponding to the first character in the page following a.

prevdoc (a ->)
(pronounced preev' dok)
Loads the **#ctrl** array with the control/format information corresponding to the first character in the document preceding a.

prevpage (a ->)
(pronounced preev' page)
Loads the **#ctrl** array with the control/format information corresponding to the first character in the page preceding a.

1.5 POINTERS AND DATA STRUCTURES SUMMARY

1.5.0 Text Maintenance Integers

beot (pronounced bee'aht)
Beginning of second section of text

bos (pronounced bahs')
Beginning of selection

bot (pronounced baht')
Beginning of user-entered text

bou (pronounced bee'oh-yu)
Beginning of undo buffer

endtext (pronounced end' tekst)
Address of byte just past absolute end of text

eos (pronounced ee-oh-ess')
Address beyond end of selection

eot (pronounced ee-oh-tee')
Address beyond end of user-entered text

eou (pronounced ee-oh-yu')
End of undo buffer.

gap (pronounced gap')
Address beyond first partition of text

text (pronounced tekst')
Address of absolute start of text area

1.5.1 Integers Used to Access the Contents of the #ctrl Array

1.5.1.0 Transient Format Information Integers

%pg	**#ctrl** + integer	**#pg**	The global page number in which this line is located
%pgl	**#ctrl** + integer	**#pgl**	The local page number within the current document in which this line is located
%wr	**#ctrl** + integer	**#wr**	Address of the first character in this line
%ln	**#ctrl** + integer	**#ln**	The global line number for this line
%lnl	**#ctrl** + integer	**#lnl**	The local line number for this line within the current page
%spr	**#ctrl** + integer	**#spr**	Can hold one of four values: 0, 1, 2, or 3. Used by the words responsible for displaying lines of text 0: Display 1 half-line 1: Do nothing 2: Display 1 half-line 3: Display 1 half-line

1.5.1.1 Paragraph Format Information Integers

%lsp	**#ctrl** + integer	**#lsp**	Local line spacing. Can hold one of three values: 2, 3, or 4. Used by the words responsible for displaying lines of text. 2: Single-spaced text 3: 1½ spaced text 4: Double-spaced text
%left	**#ctrl** + integer	**#left**	Current left margin width, expressed in half spaces. 0 <= n <= 158
%wide	**#ctrl** + integer	**#wide**	Width of the text area, expressed in half spaces. 2 <= n <= 160
%indent	**#ctrl** + integer	**#indent**	Indent distance for this line, expressed in half spaces. 0 <= n <= 158

%iwide	**#ctrl +**	integer	**#iwide**	Remaining width of the text on an indented line.
%just	**#ctrl +**	integer	**#just**	Justification 0: Left-justified 1: Right-justified 2: Center-justified 3: Fully justified
%tabs	**#ctrl +**	integer	**#tabs**	Two 80-bit bit arrays. The screen is 80 full spaces wide. The state of each bit in the first bit array indicates whether the corresponding space on the screen has a tab associated with it. The second bit array indicates whether the corresponding space has a decimal tab associated with it.

1.5.1.2 Document Format Information Integers

%long	**#ctrl +**	integer	**#long**	The value in **#long** indicates how long a page can be before an implicit page break will occur. The page length is expressed in half-lines
%above	**#ctrl +**	integer	**#above**	Holds the height of the top margin on a printed page, expressed in half-lines
%below	**#ctrl +**	integer	**#below**	Holds the height of the bottom margin on a printed page, expressed in half-lines
%lock	**#ctrl +**	integer	**#lock**	If this line is locked, will hold the ASCII value for the gray lock character
%ipage	**#ctrl +**	integer	**#ipage**	First page number in the document
%iprint	**#ctrl +**	integer	**#iprint**	Number of the first printable page in the document

1.5.2 Control/Format Integer Array Offsets

1.5.2.0 Line Offsets

%page	00
%pgl	02
%wr	04
%ln	08
%lnl	0A
%spr	0C

1.5.2.1 Format Offsets

%lsp	0E
%left	10
%wide	11
%indent	12
%iwide	13
%just	14
%tabs	16

1.5.2.2 Document Offsets

%long	2A
%above	2C
%below	2D
%lock	2E
%ipage	30
%iprint	32

1.5.3 Data Structures Integers

#ctrl (pronounced sharp' cee'tee-ar-ell, or (sharp' kon-troll')
Holds the address of the start of the **#ctrl** array

#pctrl (pronounced sharp' pee'kon-troll)
Holds the address of an array which holds the previous contents of the **#ctrl** array

#wtable (pronounced sharp' duh'bl-yu tay'bl)
Holds the address of the start of the window table

#update (pronounced sharp' up'date)
Holds the address of the start of the update array

#itbl (pronounced sharp' it'a-bl)
Holds the address of the start of the interval array

1.5.4 Window Table Integers

Name	Hex Value	Description
lastline	$4E	Last line in window table.
firstseen	$10	First window table line visible on screen.
lastseen	$3B	Last window table line visible on screen.
middle	-	Offset to the middle line in the display.
eosline	-	Line in window table containing the **eos**.
topline	-	Global line number of the first line in the window table.
gapline	-	**eos** line relative to window.

1.5.5 Interval Table Integers

Name	Hex Value	Description
esize	$38	Size of a control/format array
isize	$400	Size of the text interval represented by an interval table entry
itblsize	**esize isize** *	Size of the interval table
beotivl	-	Holds the number of the interval table entry corresponding to the text interval containing the **beot** address
endtextivl	-	Holds the number of the interval table entry corresponding to the text interval containing the **endtext** address
gapivl	-	Holds the number of the interval table entry corresponding to the text interval containing the gap

1.5.6 Wrapping Integers

markpoint (pronounced mark' point)
Place beyond which to seek pb/ds in wrap

wraplim (pronounced rap' lim)
Address of stopping point for wrap

%pwrap (pronounced per-sent' pee' rap)
Previous array wrap address

#nextwr (pronounced sharp' nekst' duh'bl-yu ar')
Holds the start address of the line which immediately follows the line whose control/format information is currently stored in the **#ctrl** array. This information is meaningful only following a **loadline** from window table.

1.5.7 Unclaimed Integers

pagebase (pronounced paje' base)

disktext (pronounced disk' tekst)
Start of text area on disk.

2. TEXT DISPLAY

Introduction

Text display is both a low-level and a high-level process. The low-level text display routines must convert encoded text data to displayable character strings. They must also work within the limitations of the screen and font sizes, and must actually draw the character data on the screen. The high-level text display routines must decide which regions of text are to be displayed and must be able to locate the formatting information for that region of text so that the lower-level display routines may be called upon to display the text.

2.0 A LOW-LEVEL LOOK AT TEXT DISPLAY

The two lowest-level display words in the editor are **build** and **disp**. Both of these words are designed to display one line of text at a time. **build** prepares a line of text for display and **disp** draws the text on the screen.

2.0.0 Preparing the Text for Display

build converts one line of encoded text data into a displayable format. **build** always prepares the current line for display, that is, the line of text whose formatting information is currently stored in the **#ctrl** array. **build** analyzes the text data starting at the address found in the **%wr** field of the **#ctrl** array and stops at the address found in the **#nextwr** system integer. The **#nextwr** integer holds the address of the start of the line which follows the current line. **build** is probably the most complicated of the display words because it must understand how the format information found in the **#ctrl** array will affect the appearance of the text on the screen and must be able to describe the desired text appearance to a lower-level display routine, **~disp**, which knows nothing about format codes.

build's output is a character string (four bytes per character) which **build** stores in the line input buffer. The address of the line input buffer is kept in the **lbuff** (el-buff) system integer. Each character in the string is described with four bytes of information:

byte 1	byte 2	byte 3	byte 4

Byte 1

Extended ASCII value (8th bit is used). Value can be in the range 0 to CF. The Cat editor has only one text font, but it can display the font in two styles: normal and bold. The data which describes how each character looks on the screen is kept in a font table. Byte 1 of an **lbuff** character description contains a number which, when multiplied by 16, yields the offset into the font table.

Byte 2

Modifiers byte. Four bits in this byte are used to specify special character styles: bold, underlined, dotted-underlined, inverse-video. Another bit indicates that this character is the last character to be drawn. The bold bit in byte 2 indicates which font table -- normal or bold -- should be used. The settings of the other three style bits determine whether additional styling data should be applied to the main character data before it is drawn. The low-level drawing routine ~disp

will continue drawing characters on the screen until a modifiers byte with the stop bit in the modifiers byte set is encountered. This will be the last character drawn by ~**disp** on that line.

Byte 3

Currently unused

Byte 4

Overstrike character. This character, if any, will be OR'ed over the main character during display. Byte 4 provides a little more information on how the character should be drawn on the screen. If the character is an overstrike character its data will be merged into the screen display (with the use of an OR operation) rather than laid over the current screen contents.

2.0.1 Special Text Preparation Cases

As **build** creates its output string in the line buffer, it must look for and handle the following special cases:

1. Page breaks or document separators in the text

   Page break and document break characters are single-byte characters in memory. But their screen representations fill the entire usable width of the screen. Therefore, when **build** encounters a page break or document separator character in the text it must construct the corresponding screen representation in character form in the **lbuff**. The screen representation of a page break or document separator is composed of many small horizontal line characters, the page or document number character, and the special underline character which lies underneath the page or document number character.

2. Margins, indents, tabs, and text justification

   build is responsible for (1) discerning the margin widths, indent widths, tab placements, and text justification styles that affect the line of text, and (2) inserting spaces and half-spaces as necessary to make the text meet the desired format specifications.

3. Highlighted text

   build must check to see if the text being decoded lies within the current selection range. If it does, **build** must specify inverse video for the character being displayed. **build** must also handle the characters whose screen appearance is selection-dependent. For example, if a carriage return is selected, it is shown as a white arrow on a black background; if it is not selected it is represented by a white space.

4. Locked regions of text

   If the line to be displayed is part of a locked document, **build** is responsible for inserting the lock character (a vertical gray bar on both sides of the screen) into the front of the display string.

5. Special character styles

   If any characters in the line have associated character styles, **build** must translate the editor style information to the disp-format style information.

2.0.2 Editor Character Sets

The following two diagrams, "The Cat Character Set," and "The Cat Display Character Set," show all characters which **build** can place in the **lbuff** and also shows the decimal and hexadecimal character codes which have been assigned to the characters.

2.0.2.0 Text Character Set

Only the character codes for characters shown in the Cat text character set may appear in the text area. Of the characters shown in the Cat text character set, only those whose character codes range from $00 to $EF are actual typeable, displayable characters. The characters with codes in the $09 through $0D range are not usually visible but they can be entered from the keyboard.

Note: A universe character, which is one level above the document character, was included to allow the implementation of universes, that is, sets of documents. The universe character is not currently supported.

The character codes in the $E0 through $EF range are codes that can be allowed in the text but cannot be generated directly from the keyboard.

The skip, paragraph format, calc, and locked calc characters are special characters which mark the start of a packet of non-text data within the text area.

The characters corresponding to the character codes in the $E9 through $EF range are modifier characters which indicate that the character which precedes them in the text has a special display attribute, that is, the character is underlined, bold, dotted-underlined, or has some combination of attributes.

The backspace attribute character (character code = $E8) may be used in the future to allow any character to be used as an accent for any other character. A backspace attribute character in the text would indicate that a backspace should be emitted before the character which precedes the backspace attribute is drawn on the screen, that is, the character which precedes the backspace attribute should be laid over the previous character in the display.

The extend attribute is currently not used or understood.

2.1 "Cat" Character Set

Shaded box = unused, reserved

DECIMAL VALUE	➡	0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240
⬇	HEXA-DECIMAL VALUE	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0			BLANK (SPACE)	0	@	P	‘	p	Ç	° DEGREE	TM	´ BARE ACCENT	´ ACCENT		SKIP MARKER	
1	1			!	1	A	Q	a	q	±	æ	®	` BARE ACCENT	` ACCENT			
2	2			"	2	B	R	b	r	‖	Æ	©	^ BARE ACCENT	^ ACCENT		¶ FORM MARKER	
3	3			#	3	C	S	c	s	Ø	PERM SPACE	†	¨ BARE ACCENT	¨ ACCENT			
4	4			$	4	D	T	d	t	ø	¶	2 SUPERSCRIPT	DOUBLE UNDERLINE BARE ACCENT	DOUBLE UNDERLINE		CALC MARKER	
5	5			%	5	E	U	e	u	ß	§	3 SUPERSCRIPT	DOTS BARE ACCENT			LOCKED CALC MARKER	
6	6			&	6	F	V	f	v	å	1/8	a SUPERSCRIPT	~ BARE ACCENT	~			
7	7			'	7	G	W	g	w	ç	3/8	o SUPERSCRIPT	STRIKE OUT BARE ACCENT	STRIKE OUT		BACK SPACE ATTRIBUTE	
8	8			(	8	H	X	h	x	L·	5/8	¿	⌒ BARE ACCENT	⌒		EXTEND ATTRIBUTE	
9	9	TAB		)	9	I	Y	i	y	I·	3/4	÷				UNDERLINE MARK	
10	A	UNIVERSE CHAR		*	:	J	Z	j	z	Bx	7/8	OVER STRIKE SPACE				BOLD MARK	
11	B	DCC BREAK		+	;	K	[	k	{	Bs	¢	1/2				UNDERLINE + BOLD	
12	C	PAGE BREAK		,	<	L	\	l	|	'n	£	1/4				DOTTED MARK	
13	D	RETURN		-	=	M	]	m	}	'N	¥	¡				UNDERLINE + DOTTED	
14	E			.	>	N	^	n	~	ℓ	R	«				BOLD + DOTTED	
15	F			/	?	O	_	o	Δ	Å	f	»				UNDERLINE + BOLD + DOTTED	

BARE ACCENTS
OVERSTRIKE CHARACTERS
RESERVED FOR FUTURE USE
TEXT MARKERS
HIDDEN TEXT OF DATA

2.0.2.1 Display Character Set

The display character set includes those characters that may appear in the text display area but whose character codes, in most cases, do not appear in the data area of the text. The character codes corresponding to the display characters lie in the $02 through $1F range. The characters in the display character set are shown in the diagram on the following page.

Only the following four character codes are allowed to appear in the text as character data and in the line output buffer as character display data: tab ($09), document separator ($0B), page break (0C), and carriage return ($0D). The significance of these four character codes depends on the environment in which they are found. When these characters are encountered in the text, they affect the appearance and organization of the text (a tab indicates a break in a line of text, a carriage return signifies the start of a new line, etc.). When these characters are encountered in the line output display buffer, they cause the visual representations of the characters to be drawn in the display (a return character code in the line output buffer causes the arrow graphic associated with a carriage return to appear on the screen). The character codes for all other display characters will never be found in the text data.

The display characters can be divided into four categories: display characters used to construct implicit/explicit page breaks lines and document separator lines, display characters used to represent in-line formatting characters (tab, carriage return, blanks), the display character used to represent locked documents, and the display characters used for screen diagnostic testing.

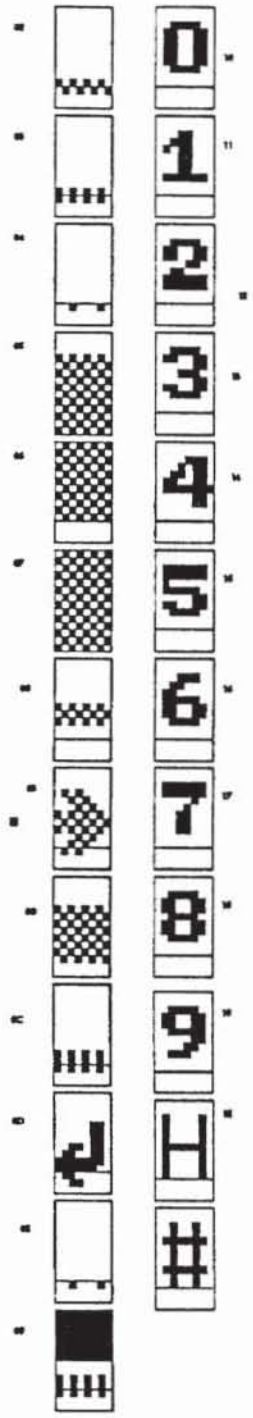
Page breaks and document separators are composed of up to four types of display characters:

1. Horizontal line segments that compose the main body of the page break or document character ($0E, $0B, $0C)

2. Special smaller and slightly elevated numbers used for page numbering ($10-$19)

3. Thin horizontal line segment that goes underneath the page number ($02, $03, $04)

4. Horizontal line segment used to represent the selected version of an explicit page break



2.2 The Display Character Set

In-line formatting uses the following display characters:

1. Selected tabs consist of two parts, the horizontal tab tail ($08), and the tab arrowhead ($09).

2. Selected carriage returns use an arrow display character which points downward and to the left ($0D).

3. Margins, and deselected carriage returns use a mark blank character ($0A).

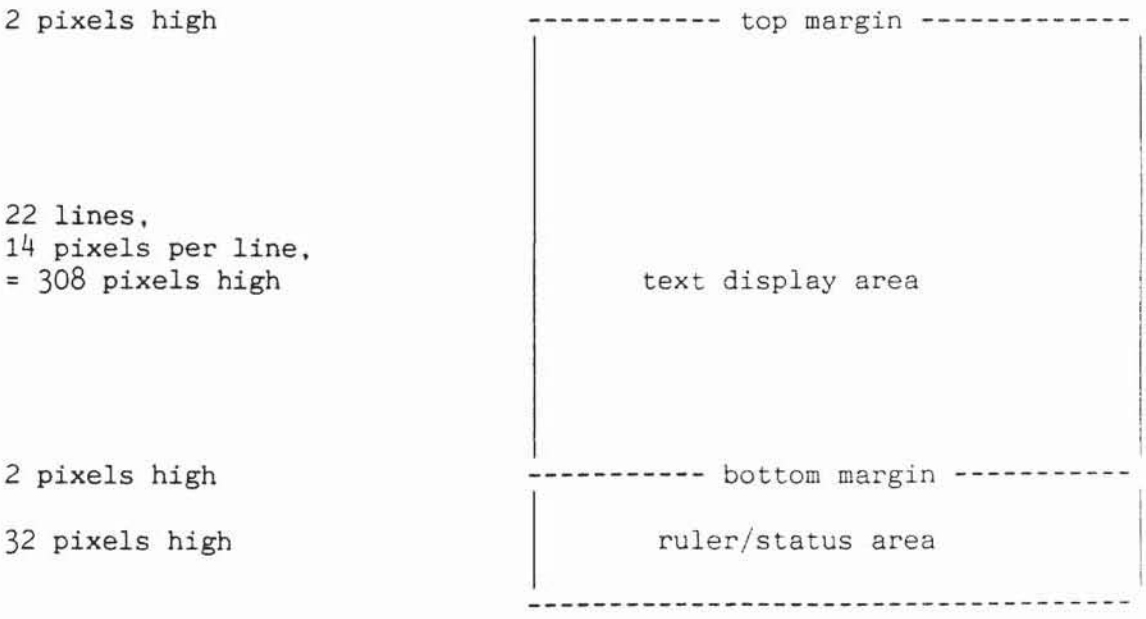
4. Deselected tabs use the blank tabspace character ($1C).

Locked documents are displayed with a series of document lock characters ($07) displayed along the edge of the locked document. The diagnostic routines use the special diagnostic display characters ($1E and $1F) for the screen test.

2.0.3 Screen and Font Dimensions

The Cat screen is 672 (decimal) pixels wide by 344 pixels high. Each character in the Cat's non-proportional font is 8 pixels wide. Therefore, 672/8 = 84 ($54) 8-bit wide characters can fit side-by-side on the Cat screen. Since a 2-character (=16 pixels) margin is always used on both sides of the text, a Cat display line supports 84 - (2 for left margin) - (2 for right margin) = 80 columns of characters.

Each character in the Cat character set fits within a 14-pixel by 8-pixel rectangle. The Cat text display area, which holds 22 lines of single-spaced text and has a 2-pixel high margin both above and below, is 22 x 14 = 308 + 2 + 2 = 312 pixels high. The ruler/status display area is 344 - 312 = 32 pixels high.



Total screen height = 2 + 308 + 2 + 32 = 344 pixels

2.0.4 Drawing Text

~disp, **disp**, and **halfdisp** are the three words that are ultimately responsible for redrawing the text portion of the display. The next section will discuss the routines which draw the ruler and status portions of the display. **~disp**, a highly optimized 7 page assembly language routine, is the word which actually draws text on the screen. **disp** sets up the registers used to pass inputs to **~disp**. **halfdisp** is a short, optimized assembly routine used to draw blank half-lines. These routines have no knowledge of the encoded format used to store the text data. Therefore, the structure of the encoded text can change without affecting the display of the text.

disp passes the following four pieces of information to **~disp**

1. the address of the normal font data table
2. the address of the bold font data table
3. the address of the line input buffer
4. the address of the location in the display memory where the text should be drawn.

The addresses of the first three parameters are fixed. **disp** uses the screen half-line number passed to it on the stack to calculate the location in screen memory where **~disp** should start drawing.

~disp traverses twice through the **lbuff**. During the first pass, **~disp** performs the following actions:

1. Checks the bold bit in the modifiers byte of the **lbuff** character information. If the bit is set, it will use the bold font table; otherwise it will use the normal font table for step 2.

2. Loads the character data for the top half of the character into the data registers. The desired character data is found by using the character code in byte 1 of the **lbuff** character information to form an offset into the font table.

3. Checks the **lbuff** character information to see if an overstrike character is required. If it is, the data for the top half of the overstrike character is obtained, and AND'ed with the data in the data registers.

4. Checks the inverse video bit. If it is set, the NOT operation complements the data in the data registers.

5. Lays the data into screen memory.

During its second pass through the **lbuff disp** processes and draws the lower halves of the characters. **~disp**'s actions during the second pass are very similar to those described above with one exception. During the second pass, after any overstrike character has been handled, **~disp** also checks the underlined and dotted underlined bits in the modifiers byte and modifies the character data accordingly before drawing it on the screen.

2.1 A HIGH-LEVEL LOOK AT TEXT DISPLAY

From a high-level point of view there are two steps required for text display: (1) the format information for the text to be displayed must be loaded into the window table, and (2) the lines represented by the entries in the window table must be displayed.

2.1.0 Obtaining Display Information

In order for any line or lines to be quickly chosen and displayed, all of the display routines expect control/format information of text to be displayed to be stored in the window table. Therefore, before the actual display routines can be called, format information must be found and placed in the window table.

loadline and **storeline** are words which can be used to access and change selected window table entries. **loadline** places the format information found in a specified window table entry in the **#ctrl** array. **storeline** stores the format information found in the **#ctrl** array into a specified window table entry.

rewindow completely fills in the contents of the window table. **rewindow** is used when the display needs to be completely recalculated and redrawn. **rewindow** assumes that the system integer **topline** holds the global line number of the first line of text to be represented in the window table. **rewindow** finds the text address of the first character in the **topline** line and wraps from that address, using **storeline** to transfer format information from the **#ctrl** array to the window table, until each entry in the window table is filled up. **rewindow** also sets the update bit for each entry in the window table as it goes along:

```
: rewindow ( -> )
      ( Install skip markers on both sides of the gap. )
      preset
      ( Load the #ctrl array with format information about the )
      ( line which should appear at the top of the screen. )
      topline findline
      ( Wrap through the text enough times to fill in each entry )
      ( in the window table.  Also update the interval table. )
      lastline 1+ 0
      do
            ( Store the information which is currently in the #ctrl )
            ( array into entry i in the window table. )
            i storeline
            putivl drop
            wrap
            ( Set the update bit for window table entry i. )
            i update!
      loop ;
```

2.1.1 Drawing the Display

refresh is the link between the low-level drawing routines **build**, **disp**, and **halfdisp** and the higher-level display routines. When called, **refresh** will check the update bits for all window table entries and will redraw, with the use of **build**, **disp**, and **halfdisp**, only those screen half-lines whose update bits are set.

If a higher-level routine wants only selected lines on the screen redrawn, it will set the update bits which correspond to those selected lines before calling **refresh**. If a higher-level routine wants to completely redraw the screen it will use **rewindow** before calling **refresh**. The definition of **refresh**, which is fairly straightforward, is included on the following page.

2.1.2 Line Spacing

rewindow uses **wrap** to get control/format information about text to be displayed and **storeline** to store that control/format information into the proper entry in the window table. **wrap** was designed for this purpose since it pays attention to line spacing as it goes through the text.

wrap will generate control/format information both for real lines of text and for display half-lines (if the text is 1½ or double-spaced). Each time **wrap** is used, it decrements the contents of the **%spr** field of the **#ctrl** array by one.

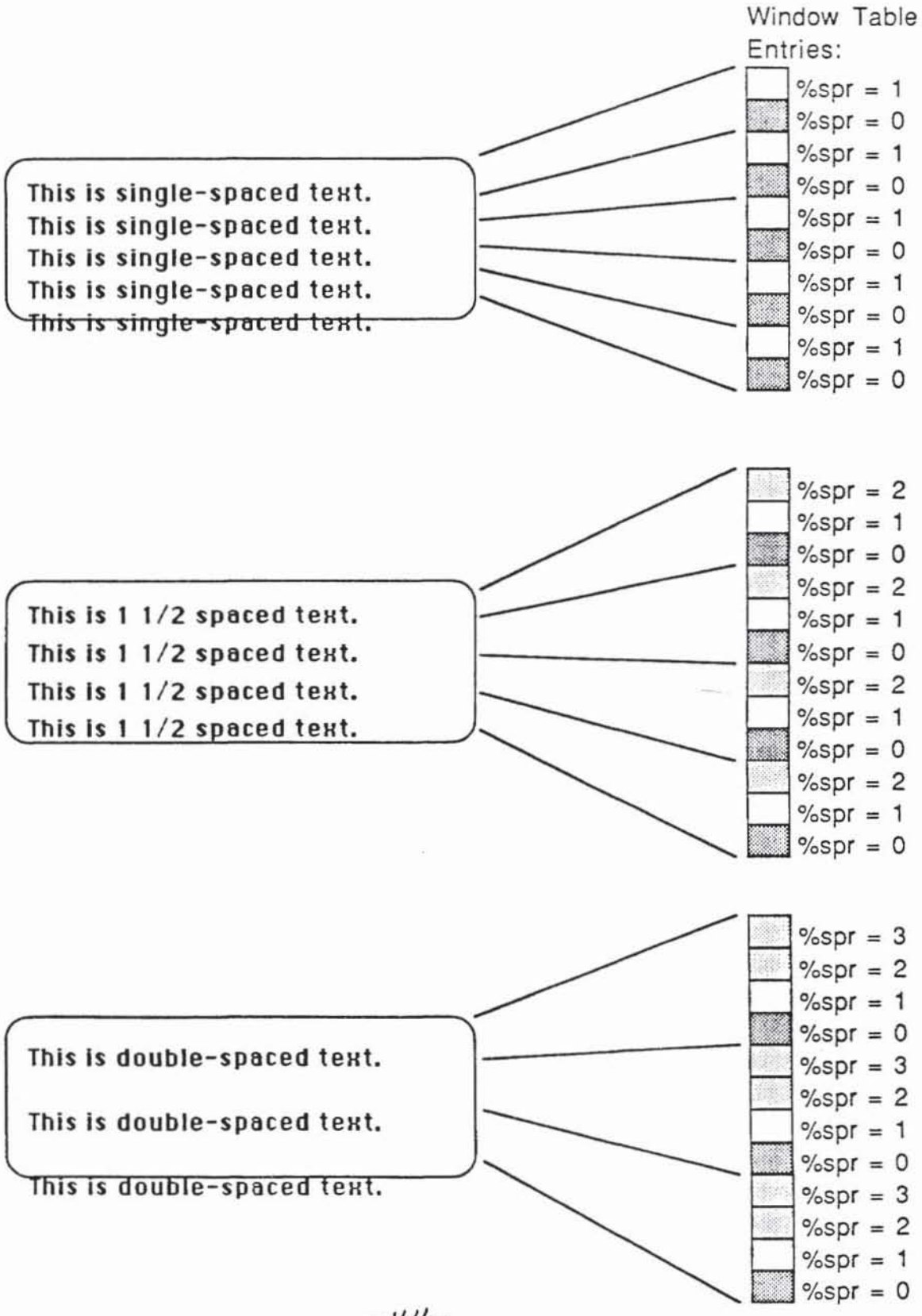
If the contents of the **%spr** are reduced to a negative number as a result of the operation (the **%spr** value will go from 0 to -1), **wrap** will reset the line spacing by replacing the contents of the **%spr** field with the value found in the **%lsp** field. If the result of the subtraction is a positive number, **wrap** will not finish wrapping the line. Instead, **wrap** will exit immediately after only having affected the contents of the **%spr** field.

refresh differentiates between the window table entries which represent real text lines and those entries which represent display half-lines by checking the contents of the **%spr** field in the window table entry. If a 0 is in **%spr**, **refresh** will display a line. If a 1 is in **%spr**, **refresh** will do nothing. If a 2 or 3 is in **%spr**, **refresh** will display a blank half-line.

Each line of single-spaced text is represented by a pair of window table entries that are identical except for the contents of the **%spr** field. The first entry in the pair will have a 1 in its **%spr** field. **refresh** will do nothing when it encounters this entry. The second entry in the pair will have a 0 in its %spr field. **refresh** will draw a line of text when it encounters this second entry.

```
: refresh ( -> )
lastseen 1+ firstseen ( Index through all half-lines in the window table.)
do ( Does this entry need to be redrawn? Exit if not.) i update?
   if
      ( If it does need redrawing, place its format)
      i loadline ( information in the #ctrl array.)
      ( Check the line spacing state, should this line be drawn?  0 = real line
          of text , 1 = do nothing, 3 & 4 = draw blank half-line.)
      #spr c@
      if ( If the first visible line, it must contain a blank half-line.)
         firstseen i =
         if 0 halfdisp
       then ( If the #spr contains a number which is greater than one then this
              text is 1½ or double-spaced. Insert blank half-lines.)
         #spr c@ 1 >
         if i 1+ firstseen - halfdisp
       then
         else
         ( A printable full line of text cannot begin on the last half-line
         position )
         lastseen i =
         if ( Draw the blank halfline at the bottom of the screen.)
            i firstseen - halfdisp
       else ( Construct display output string for line and display text.)
            build i firstseen - disp
       then
    then
 then
loop ;
```

2.3 Line Spacing



In 1½-spaced text, each line of text is represented by three almost identical window table entries. The first of the three entries will have a "2" in its **%spr** field, the second will have a "1", and the third will have a "0".

In double-spaced text, each line of text is represented with 4 window table entries. The first of the four entries has a "3" in its **%spr** field, the second has a "2", the third a "1" and the fourth a "0".

2.1.3 Drawing the Entire Display

The goal of most of the high-level display routines is to get the current selection, as delimited by the **bos** and **eos** pointers, on the screen. **new-display**, **eos-display**, **redisplay**, and **display** are all display routines which share this goal. **new-display** and **eos-display** cause the entire screen to be redrawn. **redisplay** and **display** only resort to a complete redraw if they absolutely must. If possible, **redisplay** and **display** will only redraw those screen half-lines which require updating.

fit-display is the only high-level display routine which does not care about the current selection. **fit-display**'s job is to display a specified region of text on the screen starting at a specified screen half-line number. Here is the stack notation for **fit-display**:

```
: fit-display ( halfline# start-text-rgn end-text-rgn -> )
```

The majority of **fit-display**'s work involves determining which line number should be displayed at the top of the screen. Once this number is determined, it is placed in the **topline** system integer and **rewindow** and **refresh** redraw the display. **fit-display** uses **findchar** to get format information about the two locations in the text and extracts the line numbers for the two characters from the format information. Once the start and end text range line numbers are known, **fit-display** can determine how many screen lines the text range encompasses and can position the text range accordingly.

eos-display uses **fit-display** to display the current selection with the selection start line located at a specified screen half-line position:

```
: eos-display ( halfline# -> )
        bos eos prevchar fit-display ;
```

new-display calls **eos-display** and requests that the current selection be displayed with the selection start located at the middle visible line on the screen:

```
: new-display ( -> )
        middle eos-display ;
```

2.1.4 Drawing Selected Portions of the Display

display and **redisplay** try to redraw as few lines as possible. **display** checks to see if both the **bos** and **eos** are visible in the screen and, if they are, marks only the **bos** line for updating and uses **refresh** to redraw the line. If the **bos** and **eos** are not visible in the window, **display** calls upon **new-display** to completely redraw the screen:

```
: display   ( -> )
        preset

        ( Is the character before the eos visible? )
        eos prevchar visible?
        if
                ( Is the bos line represented in the window table? )
                bos inwindow
                if
                        ( Set the update bit for the bos line. )
                        update!
                then

                ( Redraw the bos line. )
        else
                ( Redisplay the entire screen. )
                new-display
        then ;
```

redisplay is designed to be used after a text change has been made at the gap (which is usually where the cursor or selection is found). The system integer **gapline** holds the number of the screen half-line which contains the gap (actually, the character which immediately precedes the gap). **redisplay** tries to redraw only those lines which could be affected by editing activity at the gap. The lines most likely to be affected by changes at the gap are the gapline and the line which precedes the gap line.

> **Example:** Take the case of a character being inserted at the end of a word which lies at the end of a wrapped line of text. The insertion could cause the word to be too large to still fit on its current line and it would have to be pushed down to the following line (the following line would then become the **gapline**). This means that at least two lines would need to be redrawn: the new gapline (because a new word was inserted at its start), and the previous line (because the last word was removed).

If **redisplay** determines that the editing activity left the **gapline** in the window table, it will try to selectively fix the display. Otherwise, **redisplay** will call **new-display** to completely redraw the window.

To selectively fix the display, **redisplay** compares the window table format information for the **gapline**, the line which precedes the **gapline**, and for the lines which follow the **gapline**, to the current format information returned by **wrap** in the **#ctrl** array. If the format information has changed, **redisplay** places the correct format information in the window table and sets the update bit for the line and uses **refresh** to redraw all changed lines.

2.1.5 Scrolling the Display

Two words -- **scrolldown** and **scrollback** -- scroll the text downward. Two other words -- **scrollup** and **scrollfwd** -- move the displayed text upward. (Downward means that text line 1 becomes 2, 2 becomes 3, 3 becomes 4, and so forth; upward means the reverse).

Scrolling involves five steps:

1. If the text is being scrolled, rather than unscrolled, the information required for undoing the scroll must be saved away.

2. Check the selection position. If the selection is not on the top or bottom line of the display, it should be scrolled along with the text. If the selection is on the top line when the text is being scrolled up, or is on the bottom line when the text is being scrolled down, it should not scroll with the text but should stay pinned on either the top or bottom line.

3. The contents of the screen, window and update tables must be shifted downwards or upwards in memory so that one line of visible text exits the screen in the appropriate direction.

4. The lines which have been changed as a result of the scroll must be redrawn.

5. The parameters which will allow the system to undo this scroll operation must be set up.

scrollfwd and **scrollback** perform steps 1 and 3 above. If a scroll-undo operation is not occurring, they will save the selection/editor state operation into the special set of backup selection/editor state integers used only by the creeping and scrolling routines.

Next the selection position is checked. If the scroll operation would cause the selection to be scrolled out of the display, the selection will be collapsed and repositioned. If the screen contents are being scrolled up and the selection is on the top line of the display, the collapsed selection point will be moved to the first character on the following line (by altering the **bos**, **eos**, and **gap** pointers). If the screen contents are being scrolled down and the selection is on the bottom line of the display, the collapsed selection point will be moved to the first character on the preceding line (also by altering **bos**, **eos**, and **gap** pointers). If the selection is not in danger of being scrolled off the screen, it will simply be scrolled along with the line on which it resides.

Scrolling only affects the top or bottom line on the screen. The screen image and data structures associated with the lines between the top and bottom lines are shifted, but not changed. **scrolldown** and **scrollup** (used to shift the screen, window table, and update table data in memory) take advantage of this characteristic of scrolling and use block moves to shift both the screen image and the corresponding window and update table entry data up or down in memory. After **scrolldown** or **scrollup** have finished, **scrollfwd** and **scrollback** must only get new information for, and display, the top or bottom line and the line which contains the cursor (the **gapline**).

After **scrolldown** or **scrollup** has finished, the scroll operation is almost complete. Now, **scrollback** and **scrollfwd** will selectively redraw those screen lines that have been altered. If the screen contents have been scrolled down, new formatting information has been placed in the **firstseen** entry in the window table. **scrollfwd** will set the update bit which corresponds to the **firstseen** line so that it will be redrawn when the screen is refreshed. If the screen contents have been scrolled up, new formatting information has been placed in the **lastseen** entry in the window table. **scrolldown** will set the update bit for the **lastseen** line. **scrollfwd** and **scrollback** will also both set the new **gapline** line number and the update bit which corresponds to the **gapline**.

2.2 TEXT DISPLAY ROUTINES

2.2.0 Low-level Text Display Routines

build (->)
(pronounced bild')
Scans through and converts the current line of text (text with embedded formatting, and other non-printable information) to a string of printable characters in a format suitable for use by **~disp** (ASCII-value byte followed by three bytes of additional information). The string of printable characters is stored in the line output buffer **lbuff**. Any characters which lie within the current selection are marked as highlighted characters. Inserts the necessary spaces required for current justification.

The **bos** (beginning of selection) and **eos** (end of selection) pointers should be properly set up prior to the use of **build** to ensure that any highlighted characters are properly encoded.

build uses the address stored in the **#wr** field of the **#ctrl** field as the start address for its conversion process and uses the address found in the **#nextwr** integer (which should be the start address of the line which immediately follows the current line) as the end address.

The word **loadline**, which transfers the contents of the control/format array for a specified line on the screen into the **#ctrl** array (that is, to make a line on the screen a "current" line), will set up the **#nextwr** integer contents.

disp (n ->)
(pronounced disp')
Draws the line found in the line buffer, **lbuff**, on the screen at the half-line n.

~disp (code routine, parameter passed in registers)
(pronounced til'da disp')
The word which actually puts data on the screen. Draws a single line of characters on the screen each time it is called. The characters to be drawn are located a buffer whose address is passed to **~disp**. Each character to be drawn is represented by four bytes of information. **~disp** will continue taking characters from the buffer and drawing them until it reaches a character which has the end-of-buffer bit set in its modifiers byte. Does not check to see if it is printing off the edge of the screen; this check is the responsibility of the caller.

halfdisp (n ->)
(pronounced haff' disp)
Draws a blank half-line on the screen at position n on the screen.

2.2.1 Mid-level Text Display Routines

loadline (n ->)
(pronounced lode' line)
Loads the control/format information about the specified line n in the window table into the current control/format array (**#ctrl**). Also sets up the **#nextwr** integer so that it points to the start of the line which follows the current line (the line whose control/format information is stored in the **#ctrl** array). **build** requires that the **#nextwr** is properly set up.

refresh (->)
(pronounced ree' fresh)
Steps through the update array and redraws any visible half-lines which have their update bits set. If the half-line has no text, **halfdisp** will be used. If the half-line contains text, **build** converts the formatted text to character text and **disp** places the characters on the screen in the correct line position. When **refresh** has finished redrawing selected portions of the screen it will clear all positions in the update array. **refresh** can only be used to redraw lines whose formatting information is already stored in the window table.

rewindow (->)
(pronounced ree' win'doh)
Recompute the window array and mark all of lines for updating. Completely rebuilds the information in the window table. To reconstruct the window array **rewindow** makes the absolute line number found in the **topline** system integer the first line in the window array. Typically called after a situation where the entire display has been modified, for example, after an Explain message has completely overwritten the display.

storeline (n ->)
(pronounced stor' line)
Stores the current control/format information (found in the #ctrl array) into the window table field corresponding to the specified screen line number n.

2.2.2 Utility Words Used by the High-Level Text Display Routines

collapse (->)
(pronounced kah laps')
Uses **selected** to set the update bits corresponding to all lines in the display which contain the current selection, then sets the **bos** to **gap prevchar** (which reduces the selection to one character) and uses **refresh** to redraw all lines which require redisplay.

differs? (n -> f)
(pronounced dif'fers kwes'chun)
Returns a true flag if the control/format information which corresponds to screen line n in the window table is different than the control/format information in the **#ctrl** array.

inwindow (a -> n f | If flag returned is true.)
(a -> f | If the flag returned is false.)
(pronounced in-win'doh)
Returns a true flag if the character residing at the specified address a in the text belongs to one of the lines represented in the window table. If the flag returned is true, the screen line number which contains the character will also be returned.

selected (->)
(pronounced se-lek'ted)
Sets the update bits which correspond to the lines in the window table which contain the current selection so that the next time the screen is redisplayed the selection will be displayed with proper highlighting.

stepahead (n1 -> n2)
(pronounced step'a-hed')
Given the screen line number n1 of a line in the window table, returns the screen line number n2 of the next line in the window table which contains text. **stepahead** skips over blank lines and "do-nothing" lines.

stepback (n1 -> n2)
(pronounced step-bak')
Given the screen line number n1 of a line in the window table, returns the screen line number n2 of the first previous line in the window table which contains text. **stepback** skips over blank lines and "do-nothing" lines.

visible? (a -> f)
(pronounced viz'a-bl kwes'chun)
Returns a true flag if the character residing at the specified address a in the text belongs to a line which is currently visible on the screen.

2.2.3 High-Level Text Display Routines

bos-display (n ->)
(pronounced bahs' dis-play')
Causes the beginning of the current selection to be displayed at the half-line number n on the screen.

display (->)
(pronounced dis-play')
Completely redraws the display. If the format information about the selection range is already present in the window table, **refresh** redraws only those screen lines which require updating. If the lines which contain the selection range are not represented in the window table, **new-display** completely recalculates the window table and to completely redisplay the window contents.

eos-display (n ->)
(pronounced ee'ahs dis-play')
Causes the end of the current selection to be displayed at the half-line n on the screen.

fit-display (->)
(pronounced fit' dis-play')

new-display (->)
(pronounced noo' dis-play')
Causes the end of the current selection to be displayed on the middle line in the display.

redisplay (->)
(pronounced ree'dis-play')
redisplay is a less comprehensive version of **display**. **redisplay** should be used when a partial, rather than a complete display restoration, is required. **redisplay** will try to redraw only the section of the screen which has changed, but will redraw the entire display if necessary. **redisplay** wraps the text starting one line above the line which has changed. **redisplay** will continue wrapping and redrawing lines on the screen until it encounters a line which was not affected by the change. **redisplay** was designed for use after insertions and deletions at the gap have occurred (normal typing input is an example of an insertion at the gap).

scrollback (->)
(pronounced skrole' bak)
Tries to scroll the lines on the screen down by one line. If there are previous lines to be displayed, **scrollback** will first collapse the selection if it is extended.

Next, the cursor is repositioned to the start of the line which precedes the line which currently contains the cursor (the **#ctrl** array is filled with information about the previous line and the **bos** is set to point to the address in the **%wr** field of the array). **scrolldown** is then used to scroll the entire screen image down.

Finally, the update bits which correspond to the top and bottom visible lines in the display are set and **refresh** redisplays them. The top line requires redisplay because it was just scrolled in. The bottom line could be left in a half visible state after the scroll operation. If this is the case, **refresh** will detect it and erase so that it will not be shown until it completely fits in the display.

scrolldown (n ->)
(pronounced skrole' down)
Used by the higher-level scrolling word **scrollback** to scroll those lines which do not require redisplay downward on the screen. Moves the screen bit image down by n lines and moves the entries in the window table down by n entries so that each entry still corresponds to the proper half-line on the screen. Fills the invalid entries at the top of the window table with new format information.

scrollfwd (->)
(pronounced skrole' for'wurd)
Tries to scroll the lines on the screen up by one line. If there are subsequent lines to be displayed, **scrollup** will first collapse the selection if it is extended. Next, the cursor is repositioned to the start of the line following the line which currently contains the cursor (the **#ctrl** array is filled with information about the following line and the **bos** is set to point to the address in the **%wr** field of the array). **scrollup** is then used to scroll the entire screen image up.

Finally, the update bits which correspond to the top and bottom visible lines in the display are set and **refresh** redisplays them. The bottom line requires redisplay because it was just scrolled in. The top line could be left in a half-visible state after the scroll operation. If this is the case, **refresh** will detect it and erase so that it will not be shown until it completely fits in the display.

scrollup (n ->)
(pronounced skrole' up)
Used by the higher-level scrolling word **scrollfwd** to scroll those lines which do not require redisplay upward on the screen. Moves the screen bit image up by n lines and moves the entries in the window table up by n entries so that each entry still corresponds to the proper half-line on the screen. Fills the invalid entries at the bottom of the window table with new format information.

2.3 SUMMARY: INTEGERS USED FOR TEXT DISPLAY

2.3.0 Line Output Buffer Integers

A0	integer	**&horiz**	Number of horizontal half-spaces on a line.
4	integer	**lbufwide**	Width of a character entry in the line buffer.
lbuflen			Length of **lbuff**, **build** sets, print uses)
lbufwidth			The width at the last real char in **lbuff**
bosptr			Pointer into **lbuff** for **bos**
eosptr			Pointer into **lbuff** for **eos**

2.3.1 "disp" Integers

0	integer	**invbit**	Inverse video bit
1	integer	**boldbit**	
2	integer	**ulinebit**	Underline bit
3	integer	**dlinebit**	Dotted underline bit
4	integer	**stopbit**	When set, marks the end of the line output buffer contents
7	integer	**smallbit**	
01	integer	**$inv**	Mask used to check inverse video bit
02	integer	**$bold**	Mask used to check bold bit
04	integer	**$uln**	Mask used to check underline bit
08	integer	**$dln**	Mask used to check dotted underline bit
10	integer	**$end**	Mask used to check end-of-buffer data bit
80	integer	**$half**	Mask used to check half-wide character bit

2.3.2 Display-Only Characters

02			Thin, horizontal bar which goes under a document number
03			Thin, horizontal bar which goes under an explicit page number
04			Thin, horizontal bar which goes under an implicit page number
07	integer	**lok**	Gray, vertical, locked text character
08	integer	**tab0**	Flat, horizontal part of the tab arrow
0A	integer	**markbl**	A white space character
0E	integer		Horizontal component used to construct an implicit page line
0F	integer		Horizontal component used to construct selected version of an explicit page line
10			Special "0" used for page numbering
11			Special "1" used for page numbering
12			Special "2" used for page numbering
13			Special "3" used for page numbering
14			Special "4" used for page numbering
15			Special "5" used for page numbering
16			Special "6" used for page numbering
17			Special "7" used for page numbering
18			Special "8" used for page numbering

19			Special "9" used for page numbering
1C	integer	tabspace	Blank character which represents an unselected tab in the line output buffer
1E			Special "H" character used for diagnostic display testing
1F			Special "#" character used for diagnostic display testing

2.3.3 Display and Text Characters

09	integer	tab1	Arrowhead part of the tab arrow
0B	integer	ds	Text: Document separator character Display: Horizontal component used to construct a document separation line
0C	integer	pb	Text: Page break character Display: Horizontal component used to construct an explicit page break line
0D	integer	rtn	Text: Carriage return character Display: Arrow used for display of selected carriage return character

2.3.4 Screen Size Integers

50	integer	**width**	Number of bytes in one display line
54	integer	**/scan**	Bytes in a scan line
54	integer	**/lscan**	Number of "visible" bytes in a scan line
54	integer	**active/scan**	Number of active bytes per line
158	integer	**height**	Scan lines per display

10	integer	**bytes/char**	Bytes in a font table entry
4	integer	**logbytes/char**	Since we shift a lot
0E	integer	**scans/char**	Scan lines per character
/scan	integer	**scans/char***	
Bytes/line			Bytes in a text line
16	integer	**lines/screen**	Lines on screen
0E	integer	**scans/image**	Height of a character
07	integer	**tophalf**	Height/2 of a character
10	integer	**bytes/image**	**Note:** Code assumes this value!

3. RULER/STATUS AREA DISPLAY

Introduction

The editor ruler/status area is divided into two parts: the ruler bar and the status line. The status line has four separate areas:

1. Line number icon
2. Indicator lights
3. Mode icons (paragraph style, line spacing, keyboard I/II)
4. Gas gauge

The graphics used in the ruler/status area are comprised of characters from a special ruler font. The routines used to update and display the ruler/status area are discussed in this chapter.

3.0 THE RULER BAR

The ruler bar is updated each time through the main editor loop. The word **rule** is responsible for updating the ruler bar. It performs three actions:

1. Decides what the ruler should look like by examining the current tab, margin, and indent settings, and sets up a temporary ruler buffer with information about the ruler appearance. The ruler buffer is similar in function to the line buffer used for text display.

2. Uses **~showrule** to draw the ruler bar into a temporary buffer

3. Uses **~showstatus** to display the status line on the screen

The words **checkgauge** and **checkline#** are called each time through the main editor loop to update the gas gauge and line number if they have changed.

3.0.0 The Ruler Buffer

The ruler buffer is 84 decimal bytes long and is set up in the track buffer area during execution of **rule**. Each byte in the ruler buffer corresponds to one character position on the screen. The bits in each byte indicate which types of ruler items should be included in the ruler display area for each character position in the ruler buffer. The following types of items appear in the ruler:

- Left margin mark
- Right margin mark
- Indent mark
- Short tick mark
- Long tick mark
- Decimal tab mark
- Normal tab mark

When **rule** sets up the ruler buffer it first marks the positions of all of the tick marks, then it marks the left/right margin and indent marks. Finally, it marks all of the tab stops.

3.0.1 Displaying the Ruler Bar

After the ruler buffer has been set up, **rule** uses the lower-level ruler display word **~showrule** to draw the ruler into the temporary buffer. The display of the ruler requires three steps:

1. The ruler buffer information is converted to character data which is laid in bit format into an off-screen buffer.

2. Checks the contents of the system integer **blackruler**. If **blackruler** is true, the ruler should be black. The ruler image is complemented.

3. The bit-image of the ruler is transferred to the proper location in the screen memory.

A 65-byte lookup table named **rulersmarts** converts the byte information found in the ruler table to the ruler character code which should be displayed on the screen.

There are characters in the ruler font for each possible combination of ruler marks which may co-exist in one character space on the ruler. For example, there can be a left margin mark, a tick mark, and a tab mark in one character position but there should never be a left and right margin mark in the same character position.

Diagram 3.1 shows all the ruler font characters.

~showrule steps through the ruler buffer converting byte codes into character codes (with the help of the lookup table). The character code finds the data for the corresponding character in the ruler font.

Once the data is found, it is drawn into an off-screen display buffer which is also located in the track buffer area during the execution of **~showrule**.

~showrule also draws the lines which surround the ruler bar into the off-screen buffer. After the ruler has been drawn off-screen, the image is transferred to the proper location in screen memory to make it visible.

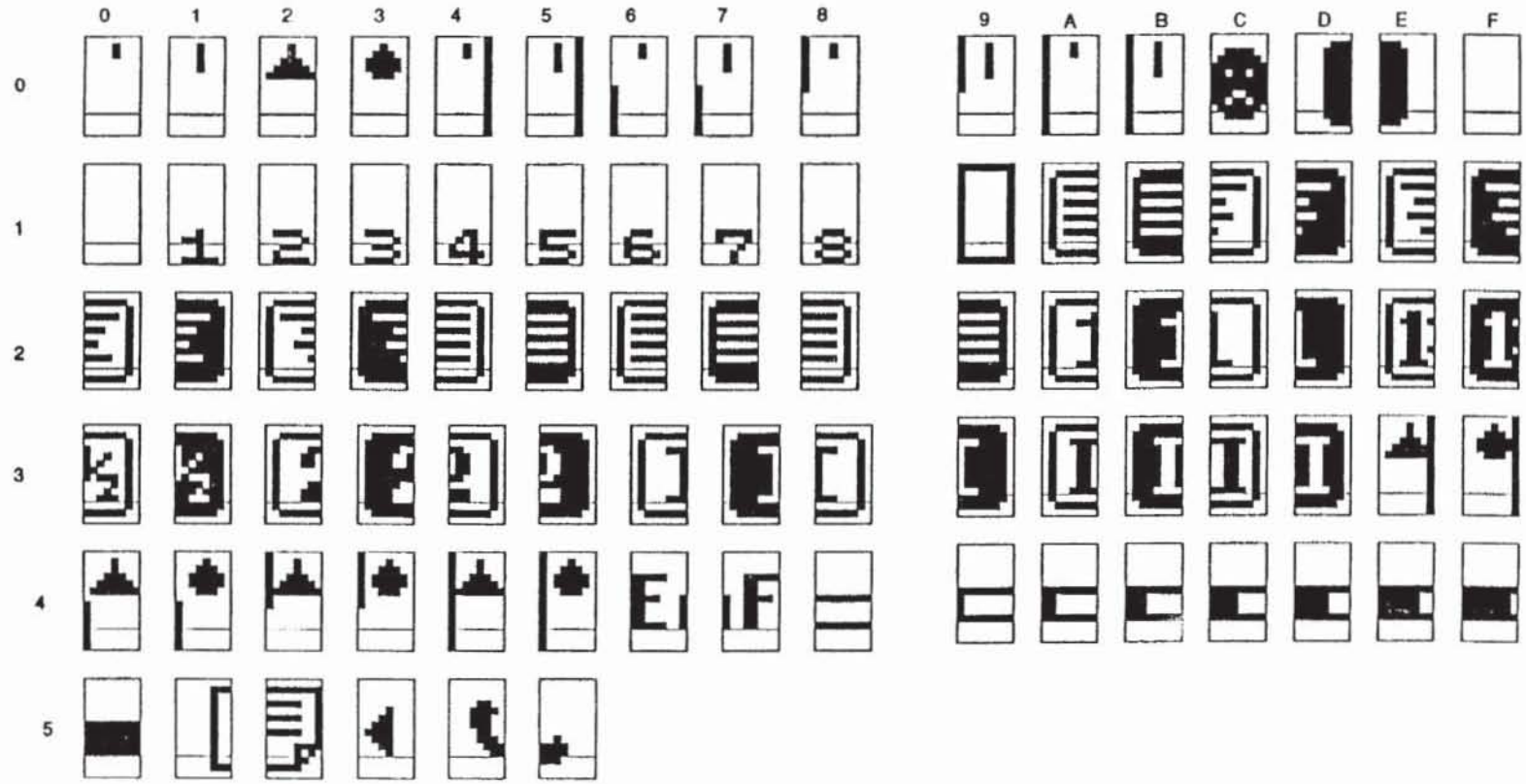
Note: The ruler is drawn off-screen first because drawing it directly into screen memory produces too much flicker in the display.

~showrule is very similar in function to **~disp**, the routine used to draw text on the screen. The display of characters in the ruler font is more straightforward than the display of text because ruler font characters can have no accents or display attributes. Because of this a separate routine, **~showrule**, which is optimized for fast display, is used instead of the more general **~disp** display routine.

3.1 THE STATUS LINE

The status line contains information about the current line number in the text, the line spacing, justification style, and keyboard currently in use, and the amount of available memory.

The status line is updated each time through the main editor loop. The word **rule**, described, uses the word **~showstatus** to update the status line display. The characters in the ruler font used for the display of the status line are shown in Diagram 3.1, page.



3.1 The Ruler Character Set

3.1.0 Display of the Status Line

The method used to display the status line is very similar to the method used to display text. In fact, the low-level screen display routine **~disp** draws the status line on the screen. The main difference between text display and status line display is that the character information for the status line is kept in a special buffer called the **statbuff** (status-buffer).

The information in the status buffer has the same format as the information in the line output buffer. Each character is described with four bytes of data (see the discussion of **disp** in the "Text Display" chapter.

The status line character information is kept in a separate buffer so that its entire contents will not have to regenerated each time through the editor loop. The only data updated by **~showstatus** each time through the loop are the paragraph style, line spacing, and keyboard settings. The new values for these items is taken from the **#ctrl** array. The other status line information, the line number, gas gauge, and indicator lights, are only updated as necessary.

3.1.1 Updating the Current Line Number

The word **checkline#** updates the current line number in the status line. The line number is updated only if required. The system integer **oldlnl** holds the line number currently displayed in the status line. **checkline#** compares the local line number found in the **#ctrl** array to the **oldlnl** value and, if the line numbers are different, will directly update the line number information in the status buffer.

3.1.2 Updating the Gas Gauge

The gas gauge indicates how much memory is available. Free memory is defined as the space between **gap** and **beot** plus the space between **here** and **applic**.

The word **checkgauge** updates the gas gauge setting. An update is done only when the change in text is significant enough to be visible on the gauge. The system integer **oldgauge** holds the free memory value used to generate the currently displayed gas gauge setting. If the gas gauge requires updating, **checkgauge** will see to it that the gas gauge information in the status buffer is altered.

Note: The gas gauge supports two memory sizes, 256K and 384K. The gas gauge gets longer when the system has 384K of RAM. The length (in pixels) to be used for the gas gauge is determined during initialization and is stored in the system integer **gaugesize**.

3.1.3 Updating the Low Battery Indicator

The low battery user informs the use when his battery is about to die. An icon is displayed in indicator light 4, whenever the hardward lowbarrery indicator is turned on.

checkbattery is called everytime through the main loop of the editor.

3.1.3 The Indicator Lights

The word **indicate** updates the indicator light data in the status buffer. The address of the string to be displayed in the light and the light position to be used is passed to **indicate**. There are five available indicator light display positions. A table called **<statuslights>**, whose address is kept in the **statuslights** system integer, holds two pieces of information for each indicator light: the maximum string length (in characters) which may be displayed in the light, and the offset into the status buffer to the display information for the indication light.

A complete list of light positions and the strings which may appear in the light positions follows:

Light 0	Light 1	Light 2	Light 3	Light 4
Phone symbol	LEARN? LEARN 1 LEARN 2 (etc.)	LOCAL	PRINT DISK BACKUP CALC SORT SEND ADDSPELL SPELLCHECK DELSPELL FORTH DISK RECOVERY	(Low Battery Icon)

Light 0 is used by the Phone command. In the future, other strings relating to the Phone command may appear in Light 1 (RINGING, BAUD300, etc.).

Light 1 is used exclusively by the Learn command. Once a number has been assigned to a Learn sequence, the "?" changes to a number from 0 to 9.

Light 2 indicates local leaping.

Light 3 is called the "thinking" light. It is used by all commands which could have a slight response delay. "FORTH" appears in the thinking light when the Answer command is used and no query is pending. Two signs, TARGET and GETFORWARD, are used only during the editor development cycle.

The following shows a typical use of **indicate**:

```
" Sort" 3 indicate rule
```

" Sort" puts the address of the string to display on the stack. 3 specifies that light 3 should be used. **indicate** changes the light 3 display information in the status buffer. **rule** causes the ruler bar and status line to be redisplayed.

Here is an example of how an indicator light is turned off:

```
0 0 3 indicate rule
```

3.1.5 The Low Battery Light

The low-battery light is not a typical indicator light. It is triggered by a hardware test rather than by a user command. When the low battery light comes on, **indicate** will be used to display the image of a battery lying on its side.

3.2 INITIALIZING THE RULER/STATUS AREA

The word **initruler** initializes the status line display. The system integer **goldbytes** holds the address of the data for the default status line display. **initruler** copies this default data into the status buffer.

3.3 RULER DISPLAY/UPDATE ROUTINES

rule (code routine, passes parameters in registers)
(pronounced rool')

Causes the ruler bar and status line to be redisplayed. Decides what the ruler bar should look like (based upon the current margin, indent, and tab settings) and creates an encoded description of the ruler bar in the status buffer. Uses **~showrule** to display the ruler bar and **~showstatus** to display the status line.

~showrule (code routine, passes parameters in registers)
(pronounced til'da sho' rool)

Displays the ruler bar. Takes byte information from the ruler buffer, converts it to ruler font character information, and draw an off-screen image of the ruler bar in the track buffer area. Then transfers the bit image directly to the screen.

3.4 STATUS LINE DISPLAY/UPDATE ROUTINES

bl# (n1 -> n2)
(pronounced bee'ell sharp')
Does one step of converting a number to a character string, suppressing zeroes. If n1 is non-zero, converts it to its corresponding ASCII value. If n1 is zero, converts it to the ASCII value for a space. Used by **checkline#**.

checkgauge (->)
(pronounced chek' gaje)
Redraws the gas gauge if necessary.

checkline# (->)
(pronounced chek' line sharp)
If necessary, updates the line number in the status line and the contents of the **oldlnl** system integer.

indicate (a n1 n2 ->)
(pronounced in'di-kate')
Places the string located at address a of length n1 into indicator light n2 and redisplays the status line.

newgauge? (-> f)
(pronounced noo'gaje kwes'chun)
Checks to see if the gas gauge needs to be redrawn. If it does, **newgauge?** returns a true flag and updates the contents of the **oldgauge** system integer.

~showstatus (code routine, parameters passed in registers)
(pronounced til'da sho' stay'tis)
Updates the paragraph style, line spacing, and keyboard information in the status buffer and then uses **~disp** to redisplay the status line.

>status (a n1 n2 n3 ->)
(pronounced too' stay'tis)
Places the string of information located at address a, which is n2 bytes in length, into the status buffer starting at an offset of n1 and uses n3 for the additional information required by **disp**.

3.5 RULER/STATUS AREA INITIALIZATION

initruler (->)
(pronounced in-it' roo'ler)
Initializes the **oldgauge**, **oldlnl**, and **gaugesize** system integers and copies the default status line data to the status buffer.

>lbuff (a1 a2 n ->)
(pronounced too ell'buf)
Moves the n bytes located starting at address a1 to memory starting at address a2. Two bytes of zeros are appended to each two bytes of data transferred so that the final data is in the 4-byte format required by **disp**.

3.6 SUMMARY

3.6.0 Ruler/Status Area Data and Data Structures

statbuff (pronounced stat' buf)
Holds address of display buffer for status line.

rulersmarts (pronounced roo'ler smarts)
Holds address of look-up table for determining ruler display characters.

statuslights (pronounced stay'tis lites)
Holds address of status lights table.

goldenbytes (pronounced gohl'den bites)
Table of status line initialization data.

goldbytes (pronounced gohld' bites)
Holds address of status line initialization data.

#goldenbytes (pronounced sharp' gohl'den bites)
Holds length of status line initialization data.

#goldenmodes (pronounced sharp' gohl'den modes')
Holds length of mode initialization data.

3.6.1 Offsets Into Status Buffer

indichars (pronounced in'di kares')
Holds offset to indicator data in the status buffer.

modechars (pronounced mode' kares)
Holds offset to mode icon data in status buffer.

gaugepos (pronounced gaje' pahs)
Holds offset to gas gauge data in status buffer.

3.6.2 Ruler/Status Screen Positioning Information

rulerstart (pronounced roo'ler start)
Scan line at which ruler area starts.

ruleredge (pronounced roo'ler edj)
Left edge position for ruler/status area.

3.6.3 Ruler/Status Area Update Information

oldlnl (pronounced ohld' ell'en-ell')
Holds current line number displayed in status line.

oldgauge (pronounced ohld' gaje)
Holds current gas gauge value displayed in status line.

blackruler (pronounced blak' roo'ler)
Holds flag used to indicate whether ruler area should be black-on-white or white-on-black

4. THE CURSOR

Introduction

The cursor has two parts, a blinking cursor and a solid highlight. Each part serves a specific purpose. The cursor always shows where the next typed character will appear; the highlight always shows what will be removed when the Erase key is pressed. In a sense, the two parts are like the two ends of a pencil (one is for writing, the other for erasing).

The cursor may appear in five different states:

1. Wide. The normal state when typing text; last typed characters appears in the highlight, blinking cursor indicates position where the next character will appear.

2. Narrow. Appears after leaping or creeping; cursor and highlight on same character, indicating that typing or erasing will take place at the location

3. Extended. Appears when both Leap keys are pressed; highlight covers more than one character

4. Split. Occurs when user leaps after extending highlight; the extended highlight remains where it is, the blinking cursor finds the new target, wherever it is in the text

5. Expanded -- occurs when the user extends the highlight (by pressing both Leap keys) during a leap; no blinking cursor in this state

The cursor routines are used by almost all of the different editor commands. They allow the cursor to be positioned relative to displayed text. They also give information about the cursor (state, size of selection, location), and blink the cursor or cause it to disappear altogether (as when it is expanded). All of the cursor routines and their associated system integers are gathered here for easy reference.

blink (->)
(pronounced blink')
If **cursorblock** holds a "0", which means the cursor is allowed to blink, **blink** will cause the cursor to flash. If the cursor is currently on, if **cursorstate** holds a "-1", <**cursoroff**> will be used to hide the cursor. If the cursor is currently off, if **cursorstate** holds a "0", <**cursoron**> will be used to show the cursor. The number of ticks until the next blink, either **ontime** (=$19), or **offtime** (=$19) is stored in the **bticks** system integer. If the text is clean (saved or just played back with no changes yet), the **ticks** value is divided by four to speed up the cursor blink rate.

cursorline (-> n)
(pronounced kur'sir lyne)
Checks the character the cursor is currently over (uses the contents of the **cpos** system integer) and the cursor state (checks the contents of the **cstate** system integer) to determine which line the cursor is on relative to the current line. Returns the relative line offset between the current line and the line holding the cursor. If the cursor is split, it will always be on the current line and a line offset of 0 will be returned. If the cursor is narrow and on a page break preceded by a break, the cursor is on the **previous** line and a line offset of -1 is returned. If the cursor is wide and on a page break, then the cursor is on the **next** line and a 1 is returned.

cursoroff (->)
(pronounced kur'sir off)
Forces the cursor off and leaves the cursor in a deactivated state. A false flag is placed in **cursorblock** to deactivate the cursor. If the cursor is currently on, <**cursoroff**> hides the cursor.

<**cursoroff**> (->)
(pronounced brak'it kur'sir off)
Turns the cursor off. Puts a false flag in **cursorstate** to set the cursor state to off. Converts the pixel position of the cursor (held in the **cx** and **cy** system integers), to a position expressed in half-characters and half-lines. If **cy** = -1, this is a signal to not display the cursor. Restores image of character where blinking cursor was put.

Next, checks **rulerblink?** to see if the ruler cursor was also being flashed. If it was, <**cursoroff**> removes the ruler cursor by filling the ruler cursor area in with the current ruler background color.

cursoron (->)
(pronounced kur'sir ahn)
Forces the cursor on and leaves the cursor in an activated state. If the cursor is currently visible, if the **cursor?** integer holds a true flag, and the cursor is currently off, **<cursoron>** will be used to show the cursor and a false flag will be placed in **cursorblock** to activate the cursor.

<cursoron> (->)
(pronounced brak'it kur'sir ahn)
Turns cursor on. Puts a true flag in **cursorstate** to set cursor state to on. If **cy** = -1, this is a signal to not display the cursor. Converts the pixel position of the cursor, held in the **cx** and **cy** system integers, to a position expressed in half-characters and half-lines.

Saves the bits which will be under the cursor in the **cursorbuf** memory buffer. Checks **cwidth** to see if the cursor is wide or narrow. Gets the proper cursor image either from **ncursorimage** (narrow-cursor-image) or **wcursorimage** (wide-cursor-image), masks out the areas of the cursor not needed (so that the character shows through the cursor), and draws the cursor image on the screen.

Next, checks **rulerblink?** to see if the ruler cursor should be flashed. If so, gets the height of the ruler cursor from **hrulercursor** and the current ruler color from **blackruler** (to determine the color for the ruler cursor) and draws the cursor in the ruler area.

?expanded (-> f)
(pronounced kwes'chun eks-pand'ed)
Checks the contents of the **cstate** system integer. If **cstate** holds a "3", if the cursor is expanded, a true flag is returned.

extend (->)
(pronounced eks-tend')
Checks to see if the cursor is currently extended. If it is not, the selection is extended. First, **extend** checks to make sure the selection start and end points are in the right order. If the **op** is located before the **beot** the selection will be extended to the left. The **bos** is set equal to the **op**, the **eos** is already properly positioned. If the **op** is located after the **beot** the selection beginning and end points must be reversed. The **eos** is set to **op nextchar**, the **op** is set equal to the **bos** and the gap is adjusted in response to the new **eos** position. Now that the pointers are properly positioned, the text is redisplayed to show the extended cursor position. **forceop** is turned on so that the **op** will follow the next character typed.

?extended (-> f)
(pronounced kwes'chun eks-tend'ed)
Checks the contents of the **cstate** system integer. If **cstate** holds a "2", if the cursor is extended, a true flag is returned.

extendedcursor (->)
(pronounced eks-tend'ed kur'sir)
Uses **widecursor** to make the cursor wide, and places a "2" in **cstate** to specify an extended cursor.

findnarrow (->)
(pronounced fynd' nair'roh)
Sets **cx** and **cy** for a narrow cursor. Uses **cursorline** to find the relative line position of the cursor. If a narrow cursor is not on the current line, then it must be on the end of the previous line. If the narrow cursor is on the previous line, **stepback** gets control/format information about the previous line.

The value of **cy** expressed in half-lines is calculated by using **inwindow** to get the text line number in which the cursor is located, and then subtracting **firstseen** from the text line number to calculate the screen half-line number on which the cursor resides. If the **cy** position indicates that the cursor is visible in the window, **loadline** and **build** builds the character display version of the line in the line output buffer, and **findwidth** calculates the width of the entire line in order to set **cx**.

The cursor width is set to the width of the last character in the line. If the narrow cursor is on the current line, or if the narrow cursor becomes positioned above the top of the window, **findsplit** sets **cx** and **cy**.

findsplit (->)
(pronounced fynd' split)
Sets **cx** and **cy** for either a split cursor or for a narrow cursor positioned in the middle of a line or above the top of the window.

Saves the screen line number which contains the cursor in **cy**. Checks to see if the cursor is on a page break character. If it is on a page break, the cursor **cx** is set so that the cursor is placed at the indent and **cwidth** is set to narrow.

Otherwise, the cursor is on a normal character. **findwidth** finds the horizontal position at which the cursor should be placed and **getwidth** determines how wide the cursor must be to cover the character which it is over.

findwide (->)
(pronounced fynd' wyde)
Sets **cx** and **cy** for a wide cursor. If the entire wide cursor is located on the current line, **inwindow** finds the line number which contains the cursor, and then **firstseen** is subtracted from the screen line number to calculate **cy** expressed in half-lines. If the end of selection is not off the end of the line (?) the value in the **eosptr** finds the width of the character under the blinking portion of the cursor (**cwidth**), and the x position of the cursor (**cx**). If the end of selection is at the end of the line, the cursor width is always set to wide and **cx** is set to the width of the entire line.

If the entire cursor is not on the current line, then it must be located at the start of the following line. If the following line is a page break, the cursor will be placed at the indent (**cx** = **#indent**) and the cursor width will be set to wide (**cwidth** = 1). Otherwise, the cursor is positioned either at the

indent (if a valid character lies at the indent), or, if there is not a valid character at the indent, over the first valid character encountered.

findwidth (a -> n)
(pronounced fynd' width)
Given the address of a character in the line output buffer, **findwidth** will calculate the widths of characters in the line output buffer up to and including the specified character and will return the result, expressed in half-character widths, on the parameter stack.

fixcursor (->)
(pronounced fiks' kur'sir)
Checks to see if the cursor has gone off the bottom of the display: **cy seenlines 2- >**. If it has, the screen is scrolled up until the line the cursor is on is at the bottom of the display and the screen display is refreshed.

getwidth (a -> n)
(pronounced get' width)
Given a pointer a to a character in the line output buffer, returns the width of the character expressed in half-characters. The result can only be "1" (one half-character) or "2" (two half-characters).

narrowcursor (->)
(pronounced nair'roh kur'sir)
Tries to force the cursor to a narrow state. If the cursor is located right after the start of a locked range of text, or if it's on the first character in the local leap range, **bor**, it is not allowed to be made narrow, so **widecursor** makes it wide. Otherwise, **cstate** is set to 0 to indicate a narrow cursor, **cpos** is positioned at **eos prevchar**, and **findnarrow** sets the cursor's **cx** and **cy** position.

narrowcursor? (-> f)
(pronounced nair'roh kur'sir kwes'chun)
Checks the contents of the **cstate** system integer. If **cstate** holds a "0", if the cursor is narrow, a true flag is returned.

real? (c -> f)
(pronounced reel' kwes'chun)
Returns a true flag if the character "c" is a character which may appear in the text ($0B<=char<=$0D or $20<=char<=$DF).

resetcursor (->)
(pronounced ree'set kur'sir)
Repositions the cursor according to the cursor state. If **cstate** is negative, **splitcursor** positions the split cursor. If **cstate** is "0" **narrowcursor** positions the narrow cursor. If **cstate** is "1" **widecursor** positions the wide cursor. If **cstate** is "2", **extendedcursor** positions the extended cursor.

?split (-> f)
(pronounced kwes'chun split)
Checks the contents of the **cstate** system integer. If **cstate** holds a "-1", if the cursor is split, a true flag is returned.

splitcursor (->)
(pronounced split' kur'sir)
Puts a "-1" in **cstate** to indicate a split cursor and uses **findsplit** to set up **cx** and **cy** for the split cursor.

widecursor (->)
(pronounced wyde' kur'sir)
Tries to force the cursor to a wide state. If the end of selection is located right at the end of a locked range of text or on the last character in a leap range **eor**, it is not allowed to be wide, so **narrowcursor** makes it narrow. Otherwise, **cstate** is set to 1 to indicate a wide cursor, **cpos** is positioned at **eos prevchar**, and **findwide** sets the cursor **cx** and cy position.

widecursor? (-> f)
(pronounced wyde' kur'sir kwes'chun)
Checks the contents of the **cstate** system integer. If **cstate** holds a "1", meaning the cursor is wide, a true flag is returned.

pushpos (-> n1 n2 n3 n4 n5 n6)
(push-position)
(pronounced push' pawz)
Push the contents of the key integers which define the editor's state onto the parameter stack. The integers pushed are: **op**, **pop**, **bos**, **cstate**, **eos**, and **gapline**.

savepos (->)
(pronounced save' pawz)
Saves the contents of the key editor state integers into a backup set of integers. The backup state integers are named: **oldop**, **oldpop**, **oldbos**, **oldcstate**, **oldeos**, and **oldtopline**.

savepos2 (->)
(pronounced save' pawz too)
Saves the contents of the current editor state integers in a special set of backup state integers used only by the creep and scroll routines. The names of these special backup state integers are: **oldop2**, **oldpop2**, **oldbos2**, **oldcstate2**, **oldeos2**, and **oldtopline2**.

swappos (->)
(pronounced swop' pawz)
Returns the screen to the way it was before all operations other than a scroll or a creep. Swaps the contents of the backup editor state integers with the contents of the current editor state integers.

swappos2 (->)
(pronounced swop' pawz too)
Returns the screen to the way it was before a scroll or a creep. Swaps the saved and current state variables for the editor.

4.2 CURSOR INTEGERS

blinktime (pronounced blink' tyme)
Holds the number of ticks until the next blink.

bosptr (pronounced boss' pee' tee arr)
Holds the offset into the line output buffer to the **bos** character

cpos (pronounced see' pawz)
Holds the text address of the character over which the cursor is currently positioned

cursor? (pronounced kur'sir kwes'chun)
Holds a flag that, if true, means the cursor is visible

cursorblock (pronounced kur'sir blok)
System integer controlling the blinking of the cursor. If **cursorblock** holds a true flag, the cursor will not blink.

cursorbuf (pronounced kur'sir buff)
System integer holding the address of the memory buffer used to hold the bit image of the screen contents currently under the cursor.

cursorstate (pronounced kur'sir stayt')
System integer which holds the flag which represents the current state of the cursor. A true flag means the cursor is on (blinking) and a false flag means the cursor is off (not blinking).

cwidth (pronounced see' width)
Holds the current width of the cursor expressed as: #half-spaces - 1. A value of 0 means the cursor is one half-space wide and a value of 1 means the cursor two half-spaces wide (full-width).

cx (pronounced see' eks)
Holds the horizontal position of the cursor expressed in pixels

cy (pronounced see' whye)
Holds the vertical position of the cursor expressed in half-lines

eosptr (pronounced ee' oh ess pee' tee arr)
Holds the offset into the line output buffer to the **eos** character, placed by **build** if character was in last built line

hrulercursor (pronounced aytch' roo'ler kur'sir)
Holds the height of the ruler cursor expressed in pixels

ncursorimage (pronounced enn' kur'sir im'ij)
Cursor image for narrow cursor

offtime (pronounced off' tyme)
How long to wait after turning cursor on (19 ticks)

ontime (pronounced on' tyme)
How long to wait after turning cursor off (19 ticks)

rulerblink? (pronounced roo'ler blink kwes'chun)
Holds a true flag if the ruler cursor should be flashed

wcursorimage (pronounced du'bl-yu kur'sir im'ij)
Cursor image for wide cursor

4.3 CURSOR PLACEMENT INTEGERS

4.3.0 Integers Which Hold the Current State of the Editor

bos (pronounced bee' oh ess)
Holds the address of the beginning of the selection (listed previously)

cstate (pronounced see' stayt)
Holds the current state of the cursor: split (negative), narrow (0), wide (1), or extended (2), or expanded (3)

eos (pronounced ee' oh ess)
Holds the address of the first character beyond the selection (listed previously)

gapline (pronounced gap' lyne)
Holds the number of the screen half-line in which the gap is located (actually, where the "gap 1-", or **gap prevchar** is located)

op (pronounced oh' pee)
Holds the address of the old cursor place

p (pronounced pee')
Means "place"

po (pronounced pee' oh)
Means "pointer"

pop (pronounced pee' oh pee)
Holds the address of the previous old cursor place

4.3.1 Integers Which Hold the Previous State of the Editor

These integers, which represent a snapshot of the text as it was when the last operation began, must be remembered in order to undo an operation. Except for the prefix **old**, they are the same as the integers defined on page.

oldop
oldpop
oldbos
oldcstate
oldeos
oldtopline

4.3.2 Integers Which Hold the Previous State of the Editor (Used by the Creeping and Scrolling Routines)

```
oldop2
oldpop2
oldbos2
oldcstate2
oldeos2
oldtopline2
```

5. WHAT'S IN THE TEXT

Introduction

This section explains what types of data are stored in the text and which editor Forth words may be used to locate and analyze the different types of data. The structure and locations of overstrike characters, character style bytes, paragraph format packets and document format packets are discussed.

5.0 STANDARD ASCII CHARACTERS AND BARE ACCENT CHARACTERS

Characters in the Cat character set which may be typed (see following diagram) have character codes ranging from $09 to $C8. In the editor, any byte data which has a value of $CF or less is assumed to be a byte of character data.

The characters with codes from $00 to **&lastasc** ("last-ASCII" $AF) are considered to be a part of the extended ASCII character set. The characters with codes from $00 to **&lastchr** ("last-character," $BF), which include all of the ASCII characters plus a few accent characters which may be typed individually (bare accents), represent all characters which may be individually typed and displayed on the keyboard.

5.0.0 Break Characters

A "break" character is any character that can cause a new paragraph to be formed in the text. Carriage returns, document separators, and page breaks are all classified as break characters are all classified as break characters.

Four words find break characters in the text: **firstbreak**, **lastbreak**, **nextbrk**, and **prevbrk**. **firstbreak** and **lastbreak** search a specified text region and return either the address of the first or last break character in the region. **nextbrk** and **prevbrk** return the address of the break character which either follows or precedes the break character located at a specified address.

There is also a word called **break?** which analyzes a character code input to determine whether or not the character code is a break character.

5.0.1 Finding Character Data

Because the text contains more than just character data, special words are included for moving from character data byte to character data byte in the text. **prevchar** takes the address of a character in the text and returns the address of the previous character in the text. **nextchar** takes the address of a character in the text and returns the address of the next character in the text. If **prevchar** or **nextchar** encounter a character with an overstrike in the text, they will always return the address of the overstruck character and never the address of the overstrike character (overstrike characters are not independent characters). Both of these words have the ability to recognize and skip over any non-character data they might encounter.

5.1 "Cat" Character Set

Shaded box = unused, reserved

DECIMAL VALUE	➡	0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240
⬇	HEXA-DECIMAL VALUE	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0			BLANK (SPACE)	0	@	P	‘	p	Ç	° DEGREE	TM	´ BARE ACCENT	´ ACCENT		SKIP MARKER	
1	1			!	1	A	Q	a	q	±	æ	®	` BARE ACCENT	` ACCENT			
2	2			"	2	B	R	b	r	‖	Æ	©	^ BARE ACCENT	^ ACCENT		¶ FORM MARKER	
3	3			#	3	C	S	c	s	Ø	PERM SPACE	†	¨ BARE ACCENT	¨ ACCENT			
4	4			$	4	D	T	d	t	ø	¶	2 SUPERSCRIPT	DOUBLE UNDERLINE BARE ACCENT	DOUBLE UNDERLINE		CALC MARKER	
5	5			%	5	E	U	e	u	ß	§	3 SUPERSCRIPT	DOTS BARE ACCENT			LOCKED CALC MARKER	
6	6			&	6	F	V	f	v	å	⅛	a SUPERSCRIPT	~ BARE ACCENT	~			
7	7			'	7	G	W	g	w	ç	⅜	o SUPERSCRIPT	STRIKE OUT BARE ACCENT	STRIKE OUT		BACK SPACE ATTRIBUTE	
8	8			(	8	H	X	h	x	L•	⅝	¿	⌒ BARE ACCENT	⌒		EXTEND ATTRIBUTE	
9	9	TAB		)	9	I	Y	i	y	I•	¾	÷				UNDERLINE MARK	
10	A	UNIVERSE CHAR		*	:	J	Z	j	z	ℬ	⅞	OVER STRIKE SPACE				BOLD MARK	
11	B	DDD BREAK		+	;	K	[	k	{	Bs	¢	½				UNDERLINE + BOLD	
12	C	PAGE BREAK		,	<	L	\	l	|	'n	£	¼				DOTTED MARK	
13	D	RETURN		-	=	M	]	m	}	'N	¥	¡				UNDERLINE + DOTTED	
14	E			.	>	N	^	n	~	ℓ	℞	«				BOLD + DOTTED	
15	F			/	?	O	_	o	Δ	Å	f	»				UNDERLINE + BOLD + DOTTED	

BARE ACCENTS
OVERSTRIKE CHARACTERS
RESERVED FOR FUTURE USE
TEXT MARKERS
HIDDEN TEXT OF DATA

5.1 OVERSTRIKE CHARACTERS

The characters with codes from **&firstacc** ("first-accent," $C0) to **&lastacc** ("last-accent," $CF) are special overstrike (or accent) characters which are treated specially both in the text and on the display.

Overstrike characters are treated specially because, although they can be typed from the keyboard, they cannot be independently displayed on the screen; they must always be displayed with (on top of) another character. Overstrike characters do not exist independently in the text either, they are always associated with the character they accent. The character code for the overstruck character is combined with the character code for the overstrike character to form a 2-byte character code in the text (the overstrike character code always follows the overstruck character code). For example, the character u with an accent grave "`" character over it, would be represented in the text with code for a "u", $75, and the code for an "`", $C0, that is, $75C0.

Note: The double underline ($C4) is a special exception to the above description of the treatment of overstrikes. The only character the double underline may combine with is the permanent space character ($93). In fact, whenever a double underline is typed, it is automatically combined with the permanent space character. This means that the double underline will always appear to be one of the standard, typeable characters to the user. In the text, however, the double underline is really viewed as a permanent space character ($93) which is overstruck with a double underline accent character.

The word **accentable?** will check a character code to determine whether the corresponding character is a character which may be accented (overstruck). A carriage return is an example of a character which cannot be overstruck. Any character with a character code between and including $20 and $AF can accept an overstrike character.

The word **accent** will take the address of a character in the text and, if the character has an overstrike character associated with it, will return the address of the overstrike character.

Note: Many of the overstrike characters have normal character counterparts known as "bare accents". A bare accent character looks like an overstrike character but can be independently typed by the user and causes only a single-byte character code to be placed in the text. The character codes for bare accent characters start at $B0 and go up to $B8.

5.2 TEXT MARKER CHARACTERS

The codes from **&firstcmd** (first-command, $E0) to **&lastcmd** (last-command, $EF) correspond to text marker characters used to mark packets of character, paragraph, or calculation data in the text. The codes from **&firsthid** ("first-hidden," $F0) to $FF are all used to represent data in the text.

Character Style Markers

A Cat character can be displayed with up to four character styles: plain, bold, underlined, and dotted underlined. Any combination of these four styles may be used for any character.

If a character has any style associated with it (other than plain), it will be followed by a byte of style information in the text. For example, if you were to look at an underlined "a" in the text you would find a byte-$61 character code value for the "a" immediately followed by a $E9, which is the style byte value used for characters that are underlined only.

The character style (attribute) markers have values in the range from $E9 to $EF. The bit representations for the style marker values are listed below:

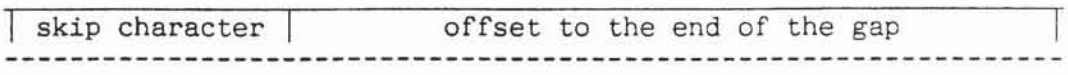
Style	Hex Value	Binary Value
Underline	E9	1110 1001
Bold	EA	1110 1010
Dotted Underline	EC	1110 1100
Underline+Bold	EB	1110 1011
Underline+Dotted	ED	1110 1101
Bold+Dotted	EE	1110 1110
Underline+Bold+Dotted	EF	1110 1111

The chart shows that bits #0, 1, and 2 are the real style bits in the style byte. Bit #0 (the leftmost bit) of the style byte is the underline bit, bit #1 is the bold bit, and bit 2 is the dotted underline bit. If a style bit is set (is a 1), the corresponding character will be displayed with that character style.

5.2.1 Gap "Skip" Markers

The gap area is a discontinuity in the text data. To let words which search through the text data know where the gap begins and ends, special information is stored in the text area on both sides of the gap. The special information is 4 bytes in length and contains the following information:

Beginning of gap:



End of gap:

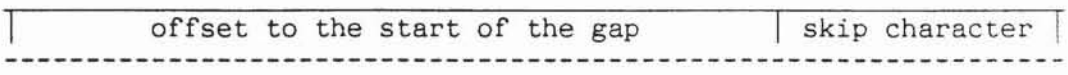


Figure 5.2: Skip Information

The skip character is a text marker character with a character code of $E0. The skip character either follows or precedes three bytes of offset information. Note that the order of the information is reversed on the different sides of the gap. The following memory dumps, executed from within the tForth environment, show how the skip information looks in memory:

```
gap 10 dump
43E804  E0 00 46 BD 69 70 20 2B 20 6F 66 66 73 65 74 0D  ..F.ip + offset. ok

beot 10 - 10 dump
442F39  66 66 73 65 74 20 2B 20 73 6B 69 70 00 46 FA E0  offset + skip.F..
```

The **beot** can be computed from the three bytes following the skip marker at the gap using the following formula:

gap + 4 + offset = **beot**

The gap can be computed from the three bytes preceding the skip marker at the **beot** by the following formula:

gap = **beot** - 1 - offset

Where offset is defined as the number represented by the three bytes shown in Figure 5.2.

When routines encounter a skip character while looking through the text data, they need to know how to get over to the other side of the gap. Two assembly language subroutines are available for this purpose: **^sk>** (skip-to-beot) and **^sk<** (skip-to-gap). Both routines should be accessed with the 68000 JSR (jump-to-subroutine). Given the address of a skip character in the text, these routines will extract the offset from the skip information and will return the address of the other side of the gap.

5.2.2 Paragraph Format Packets

A paragraph is any sequence of non-break characters surrounded on both sides by break characters (described in 5.0.0). Paragraph format data describes the style (margins, tabs, indents, justification, and line spacing) in which the paragraph of text should be displayed. Paragraph format packets immediately following a carriage return, page break, or document separator affect the formats of all subsequent paragraphs until another format packet is encountered by the word wrap algorithm.

A format packet contains data from the paragraph format information section of the control/format array in a simple encoded form. Each nibble ($x) of data in the paragraph format section is combined with a nibble with all bits set ($F) to form a byte of encoded format packet data ($Fx). The format packet data is marked in the text with a preceding paragraph format marker whose character code value is $E2.

The character code values for the format marker character -- and for all bytes of data in the format packet -- are greater than the highest allowable value for character data, so that neither the format marker nor the packet data will be treated as character data.

The following table illustrates the structure of a paragraph format packet:

Byte	Field Use
0	$E2: paragraph format marker character
1-2	Line space setting
3-4	Unused
5-6	Left margin setting
7-8	Line width
9-10	Indent setting
11-13	Width of indented line
13-14	Justification type (0-3)
15-16	Unused
17-56	Tab settings

5.2.3 Manipulating Paragraph Format Packets

brk+ (pronounced bee' arr kay plus)
findpkt (pronounced fynd' pak'it)
fpkt? (pronounced eff' pak'it kwes'chun)
All of these words are used to find or identify format packets in the text.

copypkt (pronounced kah'pee pak'it)
movepkt (pronounced moov' pak'it)
rotatepkts (pronounced roh'tayt pak'its)
swappkt (pronounced swop' pak'it)

All of these words are used to move and insert format packets in the text.

makepkt (pronounced mayk pak'it)
Takes the paragraph formatting information from the **#ctrl** array, nibble-encodes the information, and places the new format packet at a specified location in the text. **getpkt** performs the converse action. It decodes the data in the format packet at a specified location in the text and places the information in the proper fields in the **#ctrl** array.

pktbytes (pronounced pak'it byts)
Examines a region of the text and returns the total number of bytes of format data in the region. This is usually used to determine how large the undo buffer needs to be in order to hold packet information required for any future undo operation. **savepkts** and **swappkts** transfers format packets back and forth between the undo buffer and the text.

samepkt? (pronounced saym' pak'it kwes'chun)
Compares two format packets to determine if they are the same.

5.2.4 Document Format Packets

A document is any sequence of non-document characters which is surrounded on both sides by document separator characters. Document format data describes the printed and display appearance of the pages in the document (number of lines per page, number of blank lines above the top line on the page and below the bottom line on the page) and whether the document is alterable. Document format packets are located in the text immediately after the document separator character which marks the start of the document they affect.

A document format packet contains data from the document format information section of the control/format array in a simple encoded form. Each nibble ($x) of data in the document format section is combined with a nibble with all bits set ($F) to form a byte of encoded document format packet data ($Fx). Every document separator character in the text is followed by a packet of document format data.

The following table illustrates the structure of a document format packet:

Byte	Field Use
0	$0B: document separator character
1-2	Page length in half-lines
3-4	Half-lines above first printed line on page
5-6	Half-lines below last printed line on page
7	Locked document byte
8-10	Initial page in document
11-13	First page number to print

5.2.5 Manipulating Document Format Packets

dpktbytes (pronounced dee' pak'it byts)
Examines the text region between the start address a1 and the end address a2 and returns the total number of bytes n of document format information found in the region. This is usually used to determine how large the undo buffer needs to be in order to hold the document packet information required for any future undo operation. **savedpkts** and **swapdpkts** transfers document format packets back and forth between the undo buffer and the text.

makedpkt (pronounced mayk' dee' pak'it)
Makes a new document format packet using the current document formatting information found in the **#ctrl** array and places it at a specified location in the text. **getdpkt** decodes a document format packet in the text and places the information in the relevant fields in the **#ctrl** array. **getdocpkt** is a special version of **getdpkt** which transfers document format information from the set up (user configuration) variables to the **#ctrl** array

nextdsorcalc (pronounced nekst dee' ess or kalk')
This word, which means next-document-separator-or-calc-marker, locates document format packets in the text.

5.4 ROUTINES THAT INTERACT WITH SPECIAL DATA IN THE TEXT

5.4.0 Handling Skip Data

^sk> (pronounced kair'it ess' kay gray'ter)
$a0: Address of skip character in the text. Given the address of the skip character which lies at the start of the gap region, **^sk>** will extract the offset to the other side of the gap from the skip information, add the offset to the gap start address, and return the address of the **beot**.

^sk< (pronounced kair'it ess' kay less')
a0: Address of skip character in the text. Given the address of the skip character which lies at the end of the gap region, **^sk<** will extract the offset to the start of the gap from the skip information, add the offset to the gap end address, and return the address of the start of the gap.

5.4.1 Finding ASCII Data

firstbreak (a1 a2 -> a3-or-0)
(pronounced furst' brayk)
Searches the text in the region which starts at address a1 and ends at address a2. Returns the address of the first break character encountered. Returns a "0" if no break is encountered.

lastbreak (a1 a2 -> a3-or-0)
(pronounced last' brayk)
Searches the text in the region which starts at the address a1 and ends at the address a2. Returns the address of the last break character encountered. Returns a "0" if no break is encountered.

nextbrk (a1 -> a2-or-0)
(next-break)
(pronounced nekst' brayk)
Given the address of a location in the text, a1, returns the address of the next successive break found in the text. A "0" will be returned if no successive break is found.

nextchar (a1 -> a2)
(pronounced nekst' kair)
Returns the text address a2 of the character which comes after the character at the address a1.

^nextchar (Uses the A0 and D0 registers.)
(pronounced kair'it nekst' kair)
Lower-level subroutine used by **nextchar**. Returns the text address of the character which comes after the character whose address is in the A0 register.

prevbrk (a1 -> a2-or-0)
(pronounced preev' brayk)
Given the address of a location in the text, a1, returns the address of the first previous break found in the text. A "0" will be returned if no previous break is found.

prevchar (a1 -> a2)
(pronounced preev' kair)
Returns the text address a2 of the character which comes before the character at the address a1.

^prevchar (Uses the A0 and D0 registers.)
(pronounced kair'it preev' kair)
Lower-level subroutine used by **prevchar**. Returns the text address of the character which comes before the character whose address is in the A0 register.

5.4.2 Finding Data

nextmatch (n a -> a')
(pronounced nekst' matsh)
Searches forward in the text, starting from address a, until the next occurrence of the byte value n is encountered. The address, a', which contains the first occurrence of the byte value is returned on the stack. **nextmatch** will skip over the gap if encountered.

Warning: There are no boundaries on a **nextmatch** search. It will continue forever if the specified byte data value is not found.

prevmatch (n a -> a')
(pronounced preev' matsh)
Searches backwards in the text, starting from address a, until the first previous occurrence of the byte value n is encountered. The address a' which contains the first previous occurrence of the byte value is returned on the stack. **prevmatch** will skip over the gap if encountered.

Warning: There are no boundaries on a **prevmatch** search. It will continue forever if the specified byte data value is not found.

5.4.3 Analyzing ASCII Data

accent (a -> a'-or-0)
(pronounced ak'sent)
Given the address a of a character in the text, returns either the address a' of the accent character associated with the original character, or "0" if the original character does not have an accent.

accentable? (c -> f)
(pronounced ak-sent'-a-bul)
Returns a true flag if the character code c is able to receive an accent ($20<=character code<=$AF).

break? (c -> f)
(pronounced brayk' kwes'chun)
Returns a true flag if the character c is a character which would cause a new paragraph (a document separator, page break, or carriage return).

page? (c -> f)
(pronounced payj kwes'chun)
Returns a true flag if the character c is a page break character.

5.4.4 Handling Attribute Data

attribable? (c -> f)
(pronounced at-trib'a-bl kwes'chun)
Returns a true flag if the character c can have an attribute (underlined, boldfaced, etc.). Characters with character codes in the range $20<=code<=$AF can accept attribute data.

attribute (a -> n-or-0)
(pronounced at'tri-bute)
Given the address a of a character in the text, returns either the attribute byte n for the character, or, if 0 if the character does not have an attribute associated with it.

bare? (c -> f)
(pronounced bair' kwes'chun)
Returns a true flag if the character c is a bare accent character.

5.4.5 Getting Information About Format Packets

brk+ (a1 -> a2)
(pronounced brayk' plus)
Given a1, the address of a break character, returns a2, the first byte beyond the break character where a format packet may be found.

^dfmt (pronounced kair'it dee for'mat)
a0: Address in text where a document format packet is located.
Decodes a document format packet in the text and places the format information in the **#ctrl** array.

dpktbytes (a1 a2 -> n)
(pronounced dee' pak'it byts)
Examines the text region between the start address a1 and the end address a2 and returns the total number of bytes n of document format information found in the region. This is usually used to determine how large the undo buffer needs to be in order to hold the document packet information required for any future undo operation. **savedpkts** and **swapdpkts** transfers document format packets back and forth between the undo buffer and the text.

findpkt (a1 a2 -> a3)
(pronounced fynd' pak'it)
Searches the text range starting at address a1 and ending at address a2 and returns the address a3 of the first paragraph format packet found. If no packet is found, a3 will be 0.

^fmt> (pronounced kair'it for'mat gray'ter)
a0: Address in text where format packet is located. Decodes a format packet in the text and places the format information in the **#ctrl** array. Leaves a0 pointing just beyond the packet.

fpkt? (a -> f)
(pronounced eff' pak'it kwes'chun)
Returns a true flag if a paragraph format packet follows the break character located at the address a.

getdocpkt (->)
(pronounced get' dank' pak'it)
Transfers document format information from the Setup array to the **#ctrl** array.

getdpkt (a ->)
(pronounced get' dee' pak'it)
Loads the information from the document format packet located at address a in the text into the **#ctrl** array.

getpkt (a ->)
(pronounced get' pak'it)
Decodes the format packet located at address a in the text and places the format information in the **#ctrl** array.

nextdsorcalc (a1 a2 -> a3)
(pronounced nekst' dee-ess' or kalk')
Looks through the text region starting at address a1 and ending at address a2. Returns the address of the first document separator or Calc marker encountered, if any. If no document separator or Calc marker is found, return the end address of the region.

pktbytes (a1 a2 -> n)
(pronounced pak'it byts)
Examines the text region starting at address a1 and ending at address a2 and returns the total number of bytes n of format information found in the region.

samepkt? (a1 a2 -> f)
(pronounced saym' pak'it kwes'chun)
Compares the paragraph format packets located at addresses a1 and a2 in the text and returns a true flag if they are the same.

5.4.6 Moving Format Packets Around

copypkt (a1 a2 ->)
(pronounced kah'pee pak'it)
Copies the contents of the first paragraph format packet found on or after the text source address a1 over the contents of the first paragraph format packet found on or after the text destination address a2.

makedpkt (a ->)
(pronounced mayk dee'pak'it)
Encodes the document format information found in the **#ctrl** array and places the resulting document format packet at the specified address a in the text.

makepkt (a ->)
(make-packet)
(pronounced mayk pak'it)
Encodes the format information found in the **#ctrl** array and places the resulting encoded format packet at the specified address a in the text.

makespace (a n -> a')
(pronounced mayk' spays)
Tries to create a hole in the text at address a of size n bytes. If there is not enough room, an error message is issued. If there is enough room, **makespace** moves the text around, adjusts the text pointers, and returns the address where the desired space is located (the initial address could have been altered due to text movement).

maxundo (-> n)
(pronounced maks' un'doo)
Returns the maximum capacity of the undo buffer expressed in bytes.

movepkt (a1 a2 ->)
(pronounced moov' pak'it)
Creates a paragraph format size opening in the text at the destination address a2 and moves the paragraph format packet located in the text at address a1 into the opening.

rotatepkts (a1 a2 a3 ->)
(pronounced roh'tayt pak'its)
Rotates the contents of the three paragraph format packets located in the text at addresses a1 (=packet 1), a2 (=packet 2), and a3 (=packet 3). The rotation order is: packet 1 > packet 2 , packet 2 > packet 3, packet 3 > packet 1.

savedpkts (->)
(pronounced sayv' dee'pak'its)
Move all document format packets located between the address found in the system integer **prepkt** and the start of the gap into the undo buffer.

savepkts (->)
(pronounced sayv' pak'its)
Move all format packets located between the address found in the system integer **prepkt** and the start of the gap into the undo buffer.

swapdpkts (->)
(pronounced swop' dee' pak'its)
Swap all document format packets located in the text between the address found in the system integer **prepkt** and the start of the gap with the document packets in the undo buffer.

swappkt (a1 a2 ->)
(pronounced swop' pak'it)
Checks to see if there are paragraph format packets at the text addresses a1 and a2. If there are paragraph format packets at both locations, swaps the contents of the packets. If there is only a paragraph format packet at one of the locations, inserts a copy of the packet which does exist into the text at the location which did not contain a format packet.

<swappkt> (a1 a2 ->)
(pronounced brak'it swop' pak'it)
Swaps the contents of the paragraph format packet lying after the break located at address a1 with the contents of the paragraph format packet lying after the break located at address a2.

swappkts (->)
(pronounced swop' pak'its)
Swap all format packets located between the address found in the system integer **prepkt** and the start of the gap with the corresponding format packets in the undo buffer.

5.5 SUMMARY

5.5.0 Break Characters

$0B	integer	ds	Document separator character code.
$0C	integer	pb	Explicit page break character code.
$0D	integer	rtn	Return character code

5.5.1 Text Markers

$E0	integer	**&skip**	Skip the gap
$E2	integer	**&fmt**	Format packet code
$E4	integer	**&calc**	Calculation packet code
$E5	integer	**&lockedcalc**	Locked calculation packet code
$E8	integer	**&attr**	Character attribute code. Used in arithmetic code words
$EC	integer	**&dln**	Dotted underline code used in arithmetic code words

5.5.2 Character Code Limit Values

$AF	integer	**&lastasc**
$BF	integer	**&lastchr**
$C0	integer	**&firstacc**
$CF	integer	**&lastacc**
$E0	integer	**&firstcmd**
$EF	integer	**&lastcmd**
$F0	integer	**&firsthid**

5.5.3 Format Packet Values

$39 integer pktsize
Size of a paragraph format packet in the text, including format character

$0E integer dpktsize
Size of a document format packet in the text, including document separator character

6. INSERTING, ERASING, AND COPYING TEXT

Introduction

Inserting (typing), erasing, and copying are the three most basic Cat editing operations. The inserting routines must decide what styles, if any, should be given to new characters being entered into the text. The insert routines will gather characters in the gap area until they get a chance to insert the block of characters into the text. The copy routines copy the current selection and insert the copied characters into the text. The copying process is very similar to the text insertion part of the typing process. The erase routines will either erase forward, or backward or will erase an extended selection from the text depending upon the cursor state when Erase is used.

6.0 INSERTING TEXT

Characters typed at the Cat keyboard do not go directly to the editor. The interrupt routine responsible for scanning the keyboard array places key/character information in a low-level key event queue each time a key or keys is detected going up or down. When the editor is ready to receive characters it uses Forth keyboard I/O words to obtain key/character data from the queue. The Cat keyboard interface and the Forth words used to handle keyboard input are discussed in Chapter 13.

Insert (with a capital "I") is the editor word which takes typed character input and implaces it in the text. The three main functions of **Insert** -- gathering characters, inserting characters into the text, and redisplaying the text -- are discussed below.

6.0.0 Checking the Attribute State

Before **Insert** actually places characters in text, it must be determine whether special character modifiers are needed. If there are any attributes common to the nearest printing characters on both sides of the insertion point, all inserted characters will inherit those attributes. Printing characters include all characters excepting break characters and tabs. If any such attribute is on only one side of the insertion point, the inserted material will not inherit that attribute.

The attributes associated with the characters on both sides of the insertion point are AND'ed together to obtain the attribute value for the new character. The result of ANDing together two completely different attribute values -- bold and underline, for example -- is 0 or $E8. If **Insert** finds either of these results, the new character will not be assigned an attribute. If the resulting value is other than 0 or $E8, it will be used as the attribute value for the new character.

As an example, imagine that the cursor is wide and a new character is to be inserted between an underlined character and an underlined, boldface character. When the underline attribute value, $E9, and the underlined-bold attribute value, $EB are AND'ed together, the resulting value is $E9, the underlined attribute value. Since this value is not 0 or $E8, the new character, and all subsequent characters placed between the original attribute-determining characters, will be given the underline attribute.

The following code excerpt from **Insert** shows the attribute assignment decision process:

```
...

beot narrowcursor? ( look for attributes beginning at insertion point )
if prevchar then dup ( if cursor is narrow, step back one character )
begin dup c@ pb rtn ( skipping all break characters )
inrange while nextchar again ( forward to printable character )
attribute swap ( find attribute here and swap for insertion point again )
begin prevchar dup c@ pb rtn ( skip all break characters )
inrange 0= until ( scan backwards for first printable character )
attribute and ( find attribute here, then AND with other attribute )
dup e8 = ( if no common attribute bits )
if drop 0 then attrib to ( mark for "no attribute" )

...
```

6.0.1 Gathering Characters

Insert uses **continueinsert?** (which uses the lower-level Forth word <?k>) to obtain characters from the keyboard event queue. **Insert** will not terminate execution until the event queue is completely empty. This ensures that all input characters will enter text as quickly as the previously entered characters can be displayed. As the characters are received (and assigned attributes if necessary), they are placed temporarily in the gap, immediately after the skip information. A local variable pointer named "place" keeps track of the offset into the gap to the position where the next received character should be placed.

Before each character is placed in the gap area, **Insert** uses the word **enoughtext** to check the amount of available gap space. There must be at least enough room for current temporary insertion string and for a document format packet (in case the next character received is a document separator character) for **Insert** to proceed:

```
...
place gap -             ( length of the current insertion string )
dpktsize +              ( the most gap space which could be )
                        ( required by the next character )
enoughtext not          ( if there is not enough text, error )
if
        2drop
        noerror error
        leave
then
...
```

If there is enough room, **Insert** will analyze the character just received to determine how to handle it. There are six possible types of characters **Insert** will have to handle:

1. Accented characters (two bytes)
2. Accented character with an attribute (three bytes)
3. Document separator characters (**dpktsize** +1 bytes)
4. Normal character preceded by a bare accent character (two bytes)
5. Normal character with attribute (two bytes)
6. Normal character (one byte)

Accented characters are easy to recognize because their character value is greater than $FF, the maximum value which may be expressed with 1 byte. When an accented (2-byte) character is encountered, it is stored (using **w!**) at the location currently pointed to by the **place** pointer. If the accented character is to receive an attribute, the accent part of the accent character is bumped over by one byte and the attribute value is inserted between the main character and its accent character.

Whenever a document separator character is encountered, a document format packet must be created and inserted. **makedpkt** constructs a document format packet in the gap using the document formatting information currently found in the **#ctrl** array. A page break character is also inserted in the temporary gap string immediately after the document format packet.

If the character was not an accented character or document separator character, it must be a normal character represented by as a single byte. It is inserted, using **c!**, in the gap at the location pointed to by **place**. If the normal character is a character that can receive an accent and if the character which precedes the normal character in the gap is a bare accent character, the normal character and the bare accent character will be swapped and changed into a standard, accented (2-byte) character:

```
...
swap c@ accentable?      ( Can this character receive an accent? )
place prevchar c@ bare?  ( Is previous character a bare accent? )
and                      ( If both of these cases is true... )
if
    place prevchar dup   ( Duplicate address of bare accent. )
    c@                   ( Fetch the bare accent. )
    place c@             ( Fetch the character to be accented. )
    rot c!               ( Store the character to be accented in )
                         ( the location where the bare accent was. )
    0F and C0 or         ( Turn bare accent into a normal accent. )
    place c!             ( Store it immediately after the main )
                         ( character. )
else
        ....
```

If the normal character is not turned into an accent character, **Insert** will check to see if the character should receive an attribute. If the character can and should receive an attribute, the attribute value will be stored in the gap area immediately behind the character. If the character does not receive an attribute, **Insert**'s character handling process is completed. The **place** pointer will be properly incremented and **continueinsert?** will be used to check for the availability of more characters.

continueinsert? will return a true flag and a character code if a valid insertable character is available. If no valid character is available, a false flag will be returned, and the **Insert** character handling loop will be terminated, and the process of inserting the temporary character string into the text will be started.

6.0.2 Inserting Characters Into the Text

The word **insertblock** inserts the characters in the gap into the text. Before inserting the string into the text, **insertblock** checks for the following cases:

- Locked text
- Empty text
- Not enough room for insertion string
- Extended selection

If the text is locked, or if there is not enough room for the insertion, an error message is issued and no text is inserted. If the text is empty, the editor is initialized using **initedde** before the insertion. If the cursor is extended, it will be collapsed before the insertion.

If the string is to be inserted, it will be placed either before or after the **bos** character. Since any extended selections were collapsed, the **bos** character will be the character located immediately before the start of the gap. If the cursor is narrow, or if the **eos** character is the last character in a range of locked text, AND if the **bos** character is not a bare accent character, the string will be inserted before the **bos** character. This means that the **bos** character must become the **eos** character and be moved to the other side of the gap:

```
...
narrowcursor?
eos eor = or
gap prevchar c@ bare? not and
if
        ( Make bos character the eos character )
        ( and move it to the other side of the gap. )
then
...
```

After the above test has been made, and the gap moved if necessary, the string is ready to be inserted. The selection is reset if necessary and the string is inserted starting at the gap location. If the character immediately before the gap was a bare accent, and if the first character in the inserted string is a character which may accept an accent, the two characters are swapped and turned into a real accented character pair. The gap position is incremented to just beyond the end of the inserted string and the text is marked as dirty.

6.0.3 Redisplaying the Text

After the text has been inserted, the **bos** pointer is reset:

gap prevchar bos to

the screen contents are redrawn as necessary:

redisplay

and the cursor is set to the wide cursor state.

6.1 ERASING TEXT

The **Erase** command is associated with the Erase key. **Erase** always removes whatever character or characters are in the highlight and no others, with one exceptional case noted below.

Backward erase: This is a form of character-by-character erasure which resembles backspacing on a typewriter. When the cursor is wide before an erasure, it will be wide on the character preceding the erased selection afterward. Thus the wide cursor removes text to its left or backward a character at a time.

Forward erase or "gobble": This, too, is a form of character-by-character erasure, useful in removing text to the cursor's right. If the cursor is narrow, **Erase** removes the character immediately underneath it, then replaces the missing character with the next character to the right. The cursor doesn't change or move, so, with repeated applications, it appears to be standing still while "gobbling" the text to its right.

Extended erase: When the highlight is extended, all the highlighted characters will be erased. The cursor will be left wide on the character preceding the erased selection.

Erase turnaround: The first or last document break in the text or local leap region are exceptional cases. If **Erase** is invoked when the cursor is on the first document character in the text (or local leap region), that character will not be erased, but the cursor will become narrow on the next character forward. If the cursor is on the last document break in the text (or leap region), that character is not erased, but the cursor becomes wide on the character immediately preceding it. This is called erase turnaround because it switches a forward-erasing cursor to backward-erase, and vice versa.

6.1.0 Preparing for Text Removal

Before text can be removed, three tests must be made:

1. Is the selection within a valid text range?
2. Is the selection within a locked range of text?
3. Does **Erase** undo preparation need to be performed?

trimselection makes sure the selection lies within a valid text range. If the user has used the Local Leap keys to reduce their working text area to a subset of the entire text, erasures can only occur within the local leap region of text. A local leap region of text will always be bounded on both sides by document characters. **trimselection** checks to see if the current selection flows over or includes either of the bounding document separator characters for the current local leap region and "trims" away any part of the selection which does not lie inside of the local leap region. If the entire selection lies outside of the local leap region, **trimselection** will return a false flag to indicate that the **Erase** operation cannot continue.

Next, **Erase** checks to see if any part of the selection lies in a locked region of text. If any part of the selection is locked, the **Erase** operation is aborted, a warning beep is sounded, and an Explain message is made available.

If the first two tests above were passed, the **Erase** operation will occur. First though, **Erase** checks to see if any undo preparation is required. If the previous operation was not an **Erase** operation, undo preparation must be performed. This involves clearing out the undo buffer (with **clearundo**), saving the current cursor state (with **savepos**), and clearing the contents of the format packet scratch area (by filling **workpkt** with zeros). If the previous operation was **Erase**, these undo preparations will have already been performed.

Now we're ready to **Erase**. The text is marked as dirty (**dirtytext? on**) and the state of the cursor is checked. If the cursor is narrow, **gobble** will be used. Otherwise, **removeselection** will be used to remove text.

6.1.1 Gobbling Text

6.1.1.0 Checking the Selection Length

gobble is the word used to remove the selection under the narrow cursor. The table below shows the possible contents and lengths of a selection under a narrow cursor:

Selection	Selection Contents (Size)
Normal Character	ASCII code (1 byte)
Accented Character	ASCII code for character plus ASCII code for accent (2 bytes)
Attributed Character	ASCII code for character plus code for attribute (2 bytes)
Accented attributed Character	ASCII code for character, plus code for attribute, followed by ASCII code for accent

Carriage return	ASCII code for CR plus paragraph format packet (1 + paragraph format packet length)
Document Separator	ASCII code for document separator plus document format packet plus paragraph format packet. (1 + document format packet length + paragraph format packet length)

gobble checks for a selection length of less than 1 byte. The only time a narrow cursor's selection length can be less than 1 byte is when the narrow cursor is positioned over the last document character in the text. Since the last document character cannot be erased, **gobble** responds to this situation by moving the **bos** back by one character and resetting the cursor to a wide state. That way, if the ERASE key is pressed again, the erasing will proceed backwards in the text, away from the end of the text document character. After the **bos** has been repositioned, **redisplay** redraws the necessary parts of the text:

```
.
.
selsize 1 <                    ( Is the selection length less than 1? )
if
    eos prevchar bos to        ( Move the bos back by one character. )
    redisplay                  ( Redraw the screen contents. )
    widecursor                 ( Make the cursor wide instead of narrow. )
    exit                       ( Exit gobble immediately. )
then
.
.
```

6.1.1.1 The Relationship Between Break Characters, Paragraph Format Packets, and the Text

As you may recall from the section on "What's In the Text," paragraph format packets can only reside next to break characters (carriage return, page break or document separator) in the text. A paragraph format packet controls the appearance of all text following it up to the next occurrence of a break character followed by a paragraph format packet. A paragraph format packet, and the break character immediately following the text which the paragraph format packet controls, are invisibly linked to each other. The diagram on the following page illustrates this relationship between paragraph format packets, break characters, and the text. As you can see, the paragraph format packet and its associate break character lie on opposite ends of the text they control.

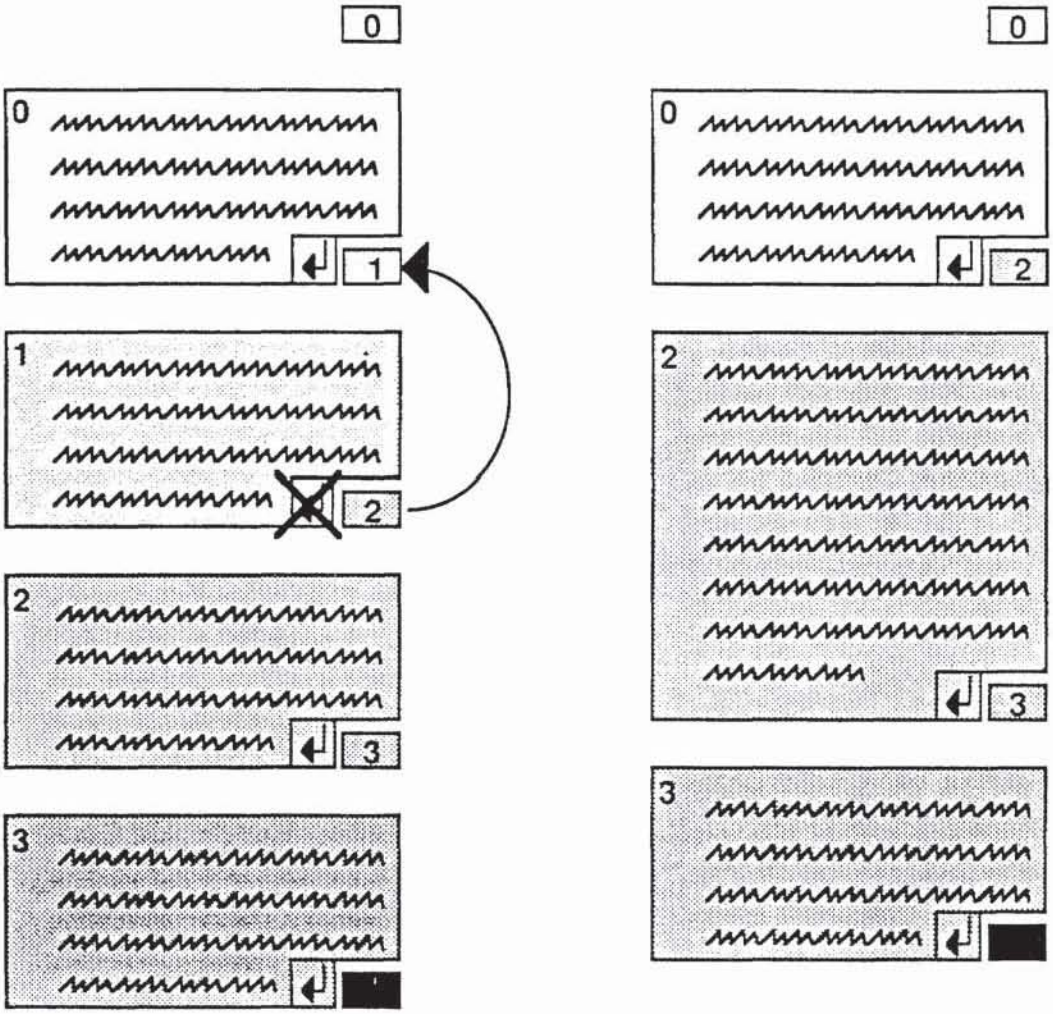
The confusing part about this arrangement is that when a break character is selected, any paragraph format information following it is also included in the selection. But the paragraph format information following a break character is not the paragraph

format information with which the break character is associated, it is the paragraph format information which controls the following paragraph!

As an example, refer to the left portion of diagram 6.1 on the following page. The carriage return following paragraph "1" has been selected for erasure. This means that both carriage return "1" and paragraph format packet "2" are included in the selection. We cannot simply dispose of the selection (paragraph format packet "2" is not associated with carriage return "1"), nor can we remove only the carriage return (paragraph format packets must always be located after a break character). The solution is to remove carriage return "1" and to move paragraph format packet "2" immediately after the previous break character in the text, carriage return "0".

In this example, since the carriage return being removed (#1) was also associated with a paragraph format packet (#1), paragraph format packet "2" will be moved into the location previously occupied by paragraph format packet "1" (which becomes invalid once its corresponding carriage return is erased). If carriage return 1 had not been associated with a paragraph format packet, paragraph format packet "2" would still have been moved after carriage return "0". The difference is that in the second case a space would have to made in the text after carriage return "0" for the insertion of paragraph format packet "2".

6.1 Gobbling a Carriage Return (and replacing its associated format packet)



Carriage return 1 is tied to paragraph format packet 1 which controls the appearance of paragraph 1. If carriage return 1 is removed, paragraph format packet 1 must be replaced with the next successive paragraph format packet, packet 2.

After carriage return 1 and paragraph format packet 1 have been removed, the previous paragraph 1 text is merged with the paragraph 2 text. This larger paragraph 2 is controlled by paragraph format packet 2.

6.1.1.2 Checking for Format Packets in the Selection

Once **gobble** has checked the selection length, it uses **pktbytes** to check for paragraph format packets in the selection. The value returned by **pktbytes**, which is the number of paragraph format packet data bytes found in the specified region, is stored in the system integer **fmtchrs**. If the selection does contain paragraph format information, **gobble** is about to erase a break character. This means the break character, and its associated paragraph format packet, if any, must be removed and the paragraph format packet contained in the selection must be moved to a location immediately after the previous break character. This is the code that handles paragraph format packets in selections to be gobbled:

```
.
.
bos gap pktbytes dup fmtchrs to ( Any format packets in selection? )
if
    workpkt @ 0=
    if
        bos findchar              ( Save the current paragraph format )
        workpkt makepkt           ( packet state in the workpkt.)
    then

    ( Does the break being removed have an associated format packet? )
    ( Remember, the packet associated with the break being removed will )
    ( be located after the previous break in the text. )
    bos prevbrk dup fpkt? 0=
    if
        brk+ pktsize makespace 1-   ( If not, make room after the )
                                    ( previous break for the insertion )
                                    ( of a format packet.)
    then

    ( Place a copy the packet contained in the selection after the )
    ( previous break in the text. )
    bos swap copypkt
then
.
.
```

Note that if the scratch **workpkt** contains no data, the state of the paragraph which the break character follows is calculated and stored in the **workpkt**.

6.1.1.3 Finishing Up the Gobble

Once any format packets have been handled, **gobble** is almost finished. **findcalc** and **linkcalc** find any Calc packets in the selection and to append them to a linked list (see the Calc discussion). **killivls** marks all intervals which correspond to the selection for updating. **partknown** marks the second text partition -- all text after the selection -- as partially valid. **movetext** appends the selection to the end of the current undo buffer contents. The first character after the selection, the

character at the **beot** location, is moved to the front of the gap and is made the **bos** character -- it becomes the next character under the narrow cursor (**gobble** is marching forward in the text). **aftererase** redraws the display. **narrowcursor** keeps the cursor narrow and **ungobble** is set as the undo operation for **gobble**.

6.1.1.4 Undoing a Gobble

ungobble is the undo operation for **gobble**. **ungobble** takes text out of the undo buffer and places format packets back in the text.

If the **workpkt** contains paragraph format information, and the previous break in the text contains a format packet, the format packet in the **workpkt** is copied over the format packet at the previous break. If the previous break does not contain a format packet, no action is taken. Next, **findcalc** and **unlinkcalc** unlink all of the Calc packets which were linked by **Erase** (or **regobble**, see below). **killivls** marks all intervals which correspond to the selection for updating. **partknown** marks the second text partition, all text after the selection, as partially valid.

Now the contents of the undo buffer must be moved back into the text at the **bos** position. This is a three step process. First, the current **bos** character must be moved to the other side of the gap. Second, the first character in the undo buffer must be moved from the undo buffer to a location right before the front of the gap. Finally, the rest of the characters in the undo buffer must be moved to the far side of the gap. After these movements occur, the character before the gap is made the **bos** character and the first character after the gap is made the **beot** character.

The remaining operations of **ungobble** involve using **clearundo** to clear the undo buffer, using **aftererase** to redraw the display, using **narrowcursor** to set the cursor to a narrow state, and setting **regobble** as the undo operation.

6.1.1.5 Undoing an Ungobble

regobble is the undo operation for **ungobble**. The first time the Undo key is pressed after an ERASE operation, an **ungobble** occurs. **ungobble** and **regobble** will alternate with each subsequent press of the Undo key. **ungobble** will always place everything in the undo buffer back into place in the text and **regobble** will always place the entire selection into the undo buffer.

6.1.1.6 Removing a Selection

If the cursor is wide or extended when **Erase** is used, all text within the selection will be removed. The cursor moves backwards in the text when **Erase** is used in this manner. The two words used to implement this type of erasing are **<removeselection>** and **removeselection**. See the definitions of these words for further information.

6.2 COPYING TEXT

The Copy command is very similar to the part of **Insert** which actually places a small section of text into the larger text area. The words used to implement the Copy command are **Copy**, **insertcopy**, **<insertcopy>**, and **Uncopy**.

Copy makes a copy of the current selection and tries to insert the copy into the text immediately after the end of the selection. **Copy** first checks the beginning and end of the selection to handle selections which may start or end on Calc packets or which lie at the beginning or end of a Leap range. After the selection endpoints have been adjusted, **Copy** checks to see if a valid selection is still left and exits if not. **selected** redisplays the adjusted selection. **Copy** also checks to make sure there is enough room for the selection and exits if not. At this point, the procedure used to copy the selection depends upon whether the selection to be copied lies within a locked document. If the selection is not locked, **insertcopy** places a copy of the selection in the gap, immediately after the skip information. **insertcopy** is then used to insert the copy into the text, immediately before the gap.

If the selection is fully or partially locked, the copy must be placed immediately after the last document in the locked region. **Copy** checks to see where the last locked document ends and checks to see if there is any unlocked text following the last document.

Once the destination for the copied text is determined, **insertcopy** places a copy of the selection in the gap. **killivls** marks all intervals in the insertion area as altered. **insertblock** actually inserts the copy into the text.

6.3 ROUTINES SUMMARY

6.3.0 Insert Routines

continueinsert? (-> c -1) if key is available
(-> 0) if key is not available
(pronounced kahn-tin'yu in'surt kwes'chun)
Fetches a key from the keyboard queue. If the key is not a special key, a true flag and the key information are returned on the stack.

enoughtext (n1 -> f | if result is true)
(n1 -> n2 f | if result is false)
(pronounced e-nuff' tekst')
enoughtext checks the amount of memory available for text insertion (that is, the amount of room in the gap) to see if there is enough room for n1 bytes to be inserted into the text. If there is enough room, a true (non-zero) flag is returned. If there is not enough room, the number of available bytes (n2) and a false flag (0) are returned. **enoughtext** uses this basic equation to determine the amount of available gap memory:

available gap memory = **bou gap - 4 -**

Insert (c ->)
(pronounced in-sert')
Inserts characters into the text until there are no characters left to insert.

insertblock (a n ->)
(pronounced in-sert' blok')
Inserts the string at address a of length n into the text. The text is inserted starting at the gap location. Before inserting the text, **insertblock** checks for a locked text or an empty text. If the text is locked, an error message will be issued and no characters will be inserted. If the text is empty, the editor will be initialized before the insertion. If the string is inserted, the text will be marked as changed.

resetselection? (-> f)
(pronounced ree'set sa-lek'shun kwes'chun)
Returns the flag held in the **forceop** integer and then places a 0 in **forceop**. If the flag is true, typing should force movement of the **op**.

6.3.1 Erase Routines

aftererase (->)
(pronounced af'tur ee-rays')

Checks to see if the line which contains the first break before the **bos** is visible in the window. If it is, **aftererase** selectively updates the window table entries for the lines between and including the previous break line and the **gapline**. Later when **redisplay** is used, only these selectively updated lines will be redrawn. Otherwise, if the previous break line is not in the window, **gapline** is set to zero to cause **redisplay** to completely recalculate and redraw the window contents.

Erase (->)
(pronounced ee-rays')

Erase is the word executed when the ERASE key is pressed. Erase puts the **%erase** value in the **curop** system integer to identify itself as the current operation. **trimselection** ensures a valid, erasable selection exists and **lockedtext?** makes sure the selection is not part of a locked portion of text. Next, **Erase** checks to see if the undo buffer has already been initialized for an erase operation. If the value in **curop** (**%erase**) is equal to the value in **lastop** then the last operation was an erasure and the undo buffer does not require initialization. Otherwise, **clearundo** clears the undo buffer, **savepos** saves the current cursor state, and the work packet, **workpkt**, is filled with zeroes. This is a signal that the format state preceding the selection has not been calculated yet. After **Erase** has checked and handled any undo initialization required, the text is marked as dirty. Now **Erase** must determine which type of erase operation should be performed. If the cursor is narrow, **gobble** erases forward in the text. If the cursor is not narrow, **removeselection** removes whatever characters are in the current selection. After the correct text has been erased, the locations of the **op** and **pop** are checked. If either of them point into the gap area, they are changed to point to the current **bos** location.

gobble (->)
(pronounced gah'bl)

Used by **Erase** when the cursor is narrow. If there is no text in the selection when **gobble** is called, the cursor must be at the end of text. The **bos** is set to **eos prevchar**, the text is redisplayed, the cursor is set to wide, and **gobble** is exited. If there is valid text, **gobble** next checks for format packets in the selection. If there are format packets, and the **workpkt** is empty, the state at the **bos** creates a format packet in the **workpkt**. This occurs only the first time in a series of erase operations to optimize for speed. Then, the break prior to the selection start is checked for a format packet. If the prior break has no format packet, a space is created for the insertion of a format packet. Then, the format packet in the selection is copied over the format packet or format packet space at the previous break location. **gobble** then uses **killivls** to mark all intervals corresponding to the text range between **bos** and **gap** as invalid, uses **partknown** to mark all intervals corresponding to

all text after **beot** as partially known, moves the selection to the undo buffer, sets up the next character in the text as the new selection, uses **aftererase** to redraw the display, leaves the cursor narrow, and sets **ungobble** as the undo operation.

movetext (a1 a2 n ->)
(pronounced moov' tekst)
In general, **movetext** removes the text region which starts at address a1 and is n bytes in length from the text and insert it into the text starting at address a2. **movetext** is primarily used for moving selections, which start at the **bos** location, to and from the undo buffer, which starts at the **bou** location.

If text is not being moved between the **bos** and **bou** locations, **movetext** will first use **enoughtext** to make sure there is enough room for the move operation. If the destination address is in the second text partition, **movetext** will move enough text to create a hole which is n bytes in size, just before the destination address and then will update all pointers affected by the text movement. If the destination is in the first text partition, room for the new text is created by moving the **gap** pointer position ahead by n bytes.

Now that space has been created for the text, **move** moves the text into place. Next, **movetext** takes care of closing up the hole created in the text when the source text was removed. If the source text was located in the second text partition **move** closes up the hole and all pointers affected are updated. If the source text was in the first text partition the hole is closed up by simply reducing the **gap** pointer by n bytes. **preset** resets the gap skip markers.

regobble (->)
(pronounced ree'gah'bl)
The undo operation for **ungobble**.

removeselection (->)
(pronounced ree-moov' sa-lek'shun)
First uses **<removeselection>** to check the selection, link the calcs in the selection, and to handle any format packets in the selection. If **<removeselection>** returns the false flag that indicates that the erase process should continue, **removeselection** will move the selection to the undo buffer, set **bos** and **cpos** equal to **gap prevchar** to reset the selection start, use **aftererase** to redraw the display, set the cursor to a widecursor, and make **restoreselection** the undo operation.

<removeselection> (-> f)
(pronounced brak'it ree-moov' sa-lek'shun)
Lower-level word called by **removeselection**. Returns a true flag if there is no valid text to erase or a false flag if the erase process should continue. **<removeselection>** first checks to see if there is any text in the selection. If there is text, but no text in the selection, then the cursor must be at the beginning of text. The cursor is made narrow and placed just after the **bot**, the text is redisplayed, and **<removeselection>** is exited. If the text and selection are both empty, the cursor is made

wide, placed at the beginning of the text, the text is redisplayed, and **<removeselection>** is exited. If the text or selection are not empty, **<removeselection>** links all Calc packets in the selection and checks to see if the selection contains any format packets. If there are no format packets in the selection, **killivls** marks the intervals corresponding to the **bos** through **gap** region for updating and **partknown** marks all of the text past the **beot** as partially known and **<removeselection>** is exited. If there is a format packet in the selection, **<removeselection>** exchanges the last format packet in the selection with the packet at the first break before the selection. The format packet at the first break before the selection is saved in the **workpkt** scratch area. All intervals corresponding to the **bos prevbrk** through **gap** region are marked for updating and all text past the **beot** is marked as partially known.

restoreselection (->)
(pronounced ree-[illegible]r' sa-lek'shun)
This is the undo operation for **removeselection**. **restoreselection** moves the text in the undo buffer back into the text starting at the **gap** location and increments the **gap** and **bou** pointers by **ubufsize** bytes. If **<removeselection>** saved a packet in **workpkt**, **restoreselection** will place it back in the text after the break which immediately precedes the selection start. **clearundo** clears the undo buffer, the **workpkt** area is filled with zeros, all Calc packets in the selection are unlinked, all intervals between the break before the selection start and the selection end are marked for updating, the text after the **beot** is marked as partially known, the selection moved back into the text is reselected, the text is redisplayed, the cursor is made extended, and **removeselection** is set as the undo operation.

trimselection (-> f)
(pronounced trim' sa-lek'shun)
If the selection begins on the document character that lies at either the start or end of the current local leap range, **trimselection** will "trim" the selection by bumping the start and/or end of selection forward/backward by one character. This is because the document characters which start and end the local leap range are not allowed to be erased or copied. Used by **Erase**. Returns a true flag if there is still a valid selection after trimming and a false flag if the selection contains no valid characters. If the only two characters in the selection region are the two document characters which mark the beginning and end of the current leap range, **trimselection** will return a false flag to indicate that there are no valid characters in the selection.

ungobble (->)
(pronounced un'gah'bl)

ungobble is the undo operation for **gobble**. **ungobble** first looks in **workpkt** to see if **gobble** removed a format packet. If **workpkt** holds a non-zero value, there is a format packet which needs to be put back into the text. If the first break before the **bos** has a format packet, the packet saved in **workpkt** will be copied over the current packet. If the prior break has no format packet, no action is taken. Next, **unlinkcalc** unlinks all selected Calc packets that must be placed back in the text (which currently resides in the undo buffer).

6.3.2 Copy Routines

Copy (->)
(pronounced kah'pee)

Makes a copy of the current selection and tries to insert the copy into the text immediately after the end of the selection. **Copy** first checks the beginning and end of the selection to handle selections which may start or end on Calc packets or which lie at the beginning or end of a Leap range. After the selection endpoints have been adjusted, **Copy** checks to see if a valid selection is still left and exits if not.

selected redisplays the adjusted selection. **Copy** also checks to make sure there is enough room for the selection, and exits if not. At this point the procedure used to copy the selection depends on whether the selection to be copied lies within a locked document. If the selection is not locked, **insertcopy** places a copy of the selection in the gap, immediately after the skip information.

insertcopy is then used to insert the copy into the text, immediately before the gap. If the copy contains any paragraph formatting information, the format at the start of the copy must be preserved. If the break just before the copy contains a paragraph format packet, its contents are changed to match the formatting of the original. If the previous break has no format packet, a packet is inserted. If the selection is fully or partially locked, the copy must be placed immediately after the last document in the locked region.

Copy checks to see where the last locked document ends and whether there is any unlocked text following the last document. If there is no unlocked text after the locked range, **Copy** issues an error and exits. If there is an unlocked area after the locked range, **Copy** proceeds with the copy process.

insertcopy places a copy of the selection in the gap. **killivls** marks all intervals in the insertion area as altered. **insertblock** inserts the copy into the text, immediately before the gap, which has been repositioned to right after the locked text region.

Now, **Copy** takes care of preserving formats. A format packet which reflects the original format of the text following the locked text region is created and either copied over a format packet that is before the copy or is inserted after the first break before the copy.

Likewise, a format packet which reflects the format at the beginning of the original selection is either copied over the first format packet after the copy or is inserted after the first break after the copy. **<swappkt>** is then used to swap the two packets so that both the copy and the text after the copy have the correct format.

After the copy has been inserted into the text, the text and cursor are redisplayed, and **Uncopy** is set as the undo operation. Copy of unpocketed definitions (calc pocket) omits the expression starting at the colon to avoid duplicate definitions. Copy of pocketed calc pockets copies only surface text result and removes the dotted underline.

<insertcopy> (a1 a2 a3 -> a1' a2')
(pronounced brak'it in-sert' kah'pee)

Tries to copy the text bytes in the range from address a1 to address a3 to memory starting at address a2. The copy process will stop if either (1) a document separator character is encountered, (2) a Calc packet is encountered, (3) the end address a3 is reached. The current locations of the source address a1' and destination address a2' are returned when **<insertcopy>** terminates.

insertcopy (a1 a2 n -> a2')
(pronounced in-sert' kah'pee)

Copies the n bytes located starting at address a1 to address a2. Uses the lower level word **<insertcopy>** repeatedly to actually move the data. If **<insertcopy>** returns without having moved all of the data, then either a document separator or Calc token was encountered. If a document separator character was encountered, **getdpkt** reads the contents of the packet into the **#ctrl** array.

The **markbl** value is stored in the **#lock** field to specifically mark the document as unlocked. **makedpkt** is then used to insert the modified document format packet into the copy and **<insertcopy>** is called again to continue moving the data after the document separator. If a Calc token was encountered, **copypocket** copies the Calc packet and **<insertcopy>** is called to continue moving the data after the Calc packet. **insertcopy** will continue calling **<insertcopy>** until all data has been moved. The address of the end of the copy is returned.

Uncopy (->)
(pronounced un'kah-pee)

Uncopy uses **<removeselection>** to remove the copy, fixes the gap pointer, marks the text as dirty, makes **Copy** the undo operation, and redraw the display.

7. CHARACTER STYLE COMMANDS

Introduction

Three character style commands are available on the Cat: Bold, Underline, and Caps. All three affect the characters in the current selection. Bold and Underline affect the style/attribute byte which may or may not be associated with any character in the text (see "What's in the Text"). The Caps command affects only the ASCII value for the character.

7.0 PREPARING TO CHANGE THE CHARACTER STYLE

Before any actions affecting the text are taken, the character style commands (**Bold**, **Under**, **Caps**) use **extend** to extend the selection and **lockedsel** to check whether the selection lies in a locked document. If any part of the selection lies in a locked document, the operation is aborted.

After extending the selection and checking for locked text, the character style commands use these texts to see if undo preparation actions need to be taken. If any test is true, the character style commands will not initialize the undo buffer:

1. Is **Uncformat** the current undo operation?

   If **Uncformat** is the current undo operation, it means the user is trying to undo the affect of a previous character style command execution. The original selection affected by the character style command is already in the undo buffer. Undo buffer initialization is not required.

2. Does the undo buffer contain text or data?

   If the undo buffer contains text or data, if **ubufsize** returns a undo buffer length which is greater than zero, the undo buffer should not be initialized.

3. Is the Use Front key currently up?

   If the Use Front key is up, then the Undo key -- rather than the [Use Front]-[Character Style Command] key combination -- was used to execute this character style command. The original selection affected by the character style command is already in the undo buffer. Undo buffer initialization is not required. When a series of character style changes have been made, Undo will undo all of them, returning the text to the state it was in prior to any changes.

If undo preparations are required, **clearundo** will be used to clear out the undo buffer, **selsize needtext** will be used to check for memory availability, and **cformat1** will be used to copy the current, unstyled selection to the undo buffer.

7.1 TO STYLE OR NOT TO STYLE

All of the character style commands are toggle commands. Each time a character style command is executed, it must either style, or unstyle the current selection. The action taken depends on the answers to the following questions:

Should the selection be styled?

> Each of the character format commands analyzes the contents of the selection to determine if the selection should be styled or unstyled.

The Bold and Underline commands check the following cases:

1. Can any of the characters in the selection receive an attribute?

2. Are there any characters in the selection which do not already have the chosen character style?

If all printable characters in the selection are already bold or underlined when the command is given, the bold or underline attribute is removed from all of them. If any of the printable characters in the selection are not bold or underlined, they will become so when the command is given, that is, the attribute will be added to those characters and the rest will remain unchanged. Break characters and tabs do not take character attributes, and so are unaffected. Spaces are considered printing characters and do take character attributes.

The Caps command performs one test: Are there any lowercase characters in the selection?

If so, the entire selection should be capitalized. If all alphabetical characters are capitalized, they will be made lowercase.

This logic does not apply in the case of a Learn command. If a Learn replay is in progress, the selection will always be styled on the first execution of the command, regardless of the foregoing set of rules.

To understand why, imagine the following scenario: During the recording of a Learn sequence you press [Use Front]-[Bold]. The Cat assumes that your intention was to make the selected text bold, no matter what. By suspending the rule about making completely bold text plain, the Cat prevents you from inadvertently changing some previously bolded text into the opposite of what you intended when you designed the Learn sequence.
The only way to cause a character format command to unstyle the text in a Learn sequence is to press the command key twice when you are recording the Learn sequence.

7.2 CHANGING THE CHARACTER STYLE

Adding a character style attribute to each character in the selection is a four-step process:

1. Determine how much extra memory will be required for the new attribute bytes.

   extramods looks through the selection and returns a count of how many new attribute bytes will be required.

2. Check memory availability.

   There must be enough room in the gap for both the characters in the current selection and for all of the new attribute data.

3. Move the selection into the gap area.

4. Move the selection, with new attributes installed, back into the text. The phrase:

   gap extrasize + 5 + bos &uln movewith

   will move the selection characters, one-by-one, back into the text and will add the underline attribute to each character which requires it.

Removing a character style attribute from each character in the selection is a three-step process:

1. Check memory availability. There must be enough room in the gap for a copy of the selection.

2. Move the selection into the gap area.

3. Move the selection (with the attribute removed) back into the text.

The phrase **gap 5 + bos selsize $uln movenotwith** will move the selection characters, one-by-one, back into the text and will remove the underline attribute from each character which possesses it. If the only attribute the character has is the underline attribute, **movenotwith** will remove the entire attribute byte.

The words **uppercase** and **lowercase** are used by **Caps** to capitalize or lowercase the characters in a selection.

7.3 UNDOING A CHARACTER STYLE COMMAND

All of the character style commands set **Uncformat** as their undo operation.

Uncformat uses **cformat3** to swap the current selection with the contents of the undo buffer, to set the new **bou** and **bos** values, to mark the intervals corresponding to the selection are for updating, and to redraw the display with the help of **cformat2**. **Uncformat** sets itself as the undo operation.

7.4 ROUTINES SUMMARY

7.4.0 Bold and Underline Command Routines

attribregion (a n1 n2 -> f)
(pronounced at'trib ree'jin)
Returns a true flag if any attributable character within the region of text of length n1 which starts at address a does not yet have the attribute n2.

Bold (->)
(pronounced bohld')
Checks the current selection. The bold attribute will be removed from all characters in the selection if
a. None of the characters in the selection can receive an attribute, OR
b. If all of the characters in the selection already have the bold attribute, AND
c. The current operation is the same as the last operation, OR
d. A Learn activity is not occurring
The bold attribute will be added to all characters in the selection if
a. Any characters in the selection can receive an attribute, AND
b. If there are characters in the selection that aren't bold, OR
c. The current operation is not the same as the last operation, AND
d. A Learn activity is occurring.

Bold copies the current selection to the gap, then copies the selection back to the text, and inserts the bold character attribute at the same time. If the selection lies within a locked area of text or if there is not enough memory to create a copy of the selection, the operation will be aborted. If the selection is not already extended, it will be extended before the operation continues.

Under (->)
(pronounced un'der)
Checks the current selection. The underline attribute will be removed from all characters in the selection if:

a. None of the characters in the selection can receive an attribute, OR
b. If all of the characters in the selection already have the underline attribute, AND
c. The current operation is the same as the last operation, OR
d. A Learn activity is not occurring

The underline attribute will be added to all characters in the selection if:

a. Any characters in the selection can receive an attribute, AND
b. If there are not-underlined characters in the selection, OR
c. The current operation is not the same as the last operation, AND
d. A Learn activity is occurring.

Under copies the current selection to the gap, then copies the selection back to the text and inserts the underline character attribute at the same time. If the selection lies within a locked area of text or if there is not enough memory to create a copy of the selection, the operation will be aborted. If the selection is not already extended, it will be extended before the operation continues.

7.4.1 Caps Command Routines

capregion (a n -> f)
(pronounced kap' ree'jin)
Returns a true flag if a lowercase character is found in the region of text of length n which starts at address a in the text.

lowercase (a n ->)
(pronounced lo'er kays)
Changes the n characters located in memory starting at address a to lowercase.

Upper (->)
(pronounced up'per)
Checks the current selection. All characters in the selection will be capitalized if:

a. The selection contains one or more lowercase letters, OR
b. The current operation is not the same as the last operation, AND
c. A Learn activity is occurring

All characters in the selection will be changed to lowercase if:

a. The selection contains only capital letters, OR
b. The current operation is the same as the last operation, AND
c. A Learn activity is not occurring

The routines **uppercase** and **lowercase** capitalize or lowercase the selection.

uppercase (a n ->)
(pronounced up'per kays)
Capitalizes the n characters located in memory starting at address a.

7.4.2 Words That Alter the Character Data

extramods (a n1 -> n2)
(pronounced eks'tra mahds')
Examines the n1 bytes of text starting at address a and returns a count n2 of how many characters within the range can be modified (can be underlined or boldfaced).

movewith (a1 a2 n1 n2 -> a3)
(pronounced moov with')
Moves the n1 bytes of text starting at address a1 to memory starting at address a2 and inserts the desired modifier information n2 after each character which can accept a modifier. Returns the address of the end of the newly modified selection.

movenotwith (a1 a2 n1 n2 -> a3)
(pronounced moov naht' with)
Moves the n1 bytes of text starting at address a1 to memory starting at address a2. If a character has the modifier n2, the modifier is removed. If n2 was the only modifier appended to a character, the entire modifier byte is removed. Returns the address of the end of the newly modified selection.

cformat1 (->)
(pronounced see for'mat wuhn)
"Character-format-one" prepares a selection for character modification. By repositioning the **bou** pointer, expands the undo buffer so that it is just large enough to hold the current selection. Moves the selection to the undo buffer.

cformat2 (a ->)
(pronounced see for'mat too)
Refreshes the selection after the characters in the selection have been modified. Updates the **gap** system integer with the address a which lies just after the last modified character. Recalculates the control/format information for those lines in the window record (which were affected by the character modification). Uses **refresh** to redraw the changed lines. Sets the cursor to "wide."

cformat3 (->)
(pronounced see for'mat three)
Places the modified selection text found in the undo buffer back in its previous spot in the text and uses **cformat2** to update the display area containing the new text.

Uncformat (->)
(pronounced un' see for'mat)
Calls **cformat3** and makes itself the undo operation.

8. PARAGRAPH FORMAT COMMANDS

Introduction

There are six paragraph formatting operations:

Left Margin
Right Margin
Indent
Paragraph Style
Line Spacing
Set/Clear Tabs

The shifted versions of these six, which restore the default setting for the corresponding format operation, make a total of 12. Each operation changes either all paragraph format packets in the current selection or, if there is no selection, just the paragraph format packet which controls the paragraph which contains the cursor. The text is redisplayed to show the affects of the new paragraph format.

8.0 GENERAL DISCUSSION

In order to change a particular aspect of a paragraph's format, the corresponding field in the paragraph format packet which controls the paragraph must be changed and the display updated. The table below shows which paragraph format information fields in a control/format array are affected by each paragraph formatting command:

Paragraph Format Operation	Field in Paragraph Format Packet
Line Spacing	**%lsp**
Left Margin	**%left, &wide**
Right Margin	**%wide, %iwide**
Indent	**%indent, %iwide**
Paragraph Style	**%just**
Set/Clear Tabs	**%tabs**

The six paragraph formatting commands are linked together so that as long as the Use Front key is held down several paragraph format commands may be used in succession without forcing the user to rehighlight the selection after each individual operation. When several commands are used in this manner, a single press of the Undo key will undo the effects of all of the commands.

8.0.0 Paragraphs and Paragraph Format Packets

A paragraph is defined as a group of characters which starts with the first character after a break character and ends with the following break character. The paragraph format packet which defines the format for a paragraph of text is located before the first character in the paragraph, immediately after the break character located at the end of the previous paragraph. Thus, when the format of a paragraph is changed, the previous paragraph format packet must be found and updated. If a paragraph does not have its own paragraph format packet, it is controlled by the first previous paragraph format in the text.

8.0.1 The Paragraph Formatting Routines

There are two versions of each of the six main paragraph formatting functions. The main version sets a format to the user's specifications; the default version restores the default settings for the particular format operation.

Main Version	Default Version
Indent	**Defindent**
Justify	**Defjustify**
Left	**Defleft**
Right	**Defright**
Spacing	**Defspacing**
Tabs	**Deftabs**

A table listing the default paragraph format settings for each country is included at the end of this section.

8.1 FOUR STEPS TO A NEW PARAGRAPH FORMAT

The definition for the main version of the paragraph style command provides a general outline of the four steps required to change a paragraph format.

Step 1: Preparing the selection

All of the paragraph formatting operations use **preform** for selection preparation purposes. **preform** checks for a locked selection and loads the current control/format information into the **#ctrl** array.

Step 2: Getting the new format setting

There are two methods used to determine new settings. The paragraph style and line spacing operations each allow only a small, fixed set of possible choices. Each time one of these operations is invoked it presents the user with the next available choice in its set.

A new left/right margin, indent, or tab setting is chosen by the user with the aid of a graphical positioning tool. The listing below shows how the **Justify** command, a command with a fixed set of possible choices, performs step 2.

Step 3: Saving the selection information

The new format setting(s) was determined in Step 2. In Step 3, the new setting information is stored into the appropriate field in the **##ctrl** array.

Step 4: Updating the Text and the Display

In Step 4, the formatting operations use **reform** to complete the formatting operation. In the example below, the token of **fixjustify**, which transfers the value previously stored in the **%just** field of the **##ctrl** array to the **%just** field of the **#ctrl** array, is passed to **reform**. **reform** executes **fixjustify**, creates a paragraph format packet using the new paragraph format information in the **#ctrl** array, replaces all paragraph format packets in the selected area with the new format packet, and causes the screen to be redisplayed.

The words **fixspacing**, **fixindent**, **fixleft**, **fixright**, and **fixtabs** perform functions similar to **fixjustify** for the other formatting operations.

```
: Justify  (  -> )
        preform                 ( 1. Prepare for formatting operation )
        #just c@ 2+             ( 2. Calculate new format setting )
        dup 4=                  ( Determine what the new value for the )
        if                      ( paragraph style should be. )
                drop  1
        then
        dup 5 =
        if
                drop 0
        then
        ##ctrl %just + c!       ( 3. Save new setting(s) info. )
                                ( Store the new value in the %just )
                                ( field of the ##ctrl array. )
        ['] fixjustify reform ; ( 4. Update the text and selection )
```

8.2 FOUR STEPS TO A DEFAULT PARAGRAPH FORMAT

The definition for the default version of the paragraph style command provides a general outline of the four steps required to change a paragraph format.

Defjustify, the default version of **Justify**, stores the default paragraph style setting found in an array of default paragraph format settings, in the **##ctrl**, and then uses **reform** to finish the reformatting job. The default paragraph format settings are found in the **#defaults** array.

```
: Defjustify  (  -> )
        preform                  ( 1. Prepare for formatting operation. )
        #defaults %just + c@     ( 2. Calculate new format setting. )
        ##ctrl %just + c!        ( 3. Save new setting(s) information. )
        ['] fixjustify reform ;  ( 4. Redisplay the selection. )
```

8.3 OBTAINING NEW FORMAT SETTINGS

The methods each paragraph format command uses to determine the user's new desired format setting are described below.

8.3.0 Obtaining a New Line Spacing Setting

The editor can handle three different line spacing settings:

Line Spacing	Field Value
Single-spaced	2
1½-spaced	3
Double-spaced	4

Spacing calculates a new line-space setting by (1) obtaining the current line spacing value from the **%lsp** field in the **#ctrl** array, (2) adding 1 to the value, (3) subtracting 3 from the result if it is equal to 4.

8.3.1 Obtaining a New Paragraph Style Setting

The editor supports four paragraph styles:

Paragraph Style	Description	Field Value
Left-justified	Justified left margin, ragged right	0
Right-justified	Justified right margin, ragged left	1
Centered	Ragged left and right margin	2
Fully justified	Justified left and right margin	3

The paragraph style icons in the ruler display area are arranged from left to right as follows: Left-justified, Centered, Right-justified, Fully justified.

When multiple selections of the paragraph style command are used, the icon highlight should flow smoothly from left to right, without skipping any icons. This means the **%just** field must be fed the following sequence of values: 0, 2, 1, 3 (refer to the table above).

So, to calculate a new paragraph style value, **Justify** (1) obtains the current paragraph style value from the **%just** field of the **#ctrl** array, (2) adds 2 to the value, and (3) if the result is 4 (invalid) sets the result to 1, or, if the result is 5 (invalid), sets the result to 0.

8.3.2 The Vertical Formatting Bar

The left/right margin, indent, and tab commands use a graphical aid -- a vertical formatting bar -- to help the user select new settings. After the user chooses one of these commands, the vertical bar appears on the screen in a command-specific location.

At this point the command will wait in a loop, moving the vertical bar according to the user's inputs, until either an invalid key is pressed or until the Use Front key is released. An invalid key is any other paragraph formatting command key aside from the command key which caused the loop to be entered. So, as long as the Use Front key is depressed and an invalid key is not entered, the vertical bar will remain on the screen and the command word will be watching for positioning directions.

8.3.3 Obtaining a New Left/Right Margin or Indent Setting

If the left/right margin or indent commands are chosen, the word **marginloop** will be used for vertical bar positioning. While **marginloop** runs, it watches for presses of either Leap key. When the left or right Leap key is pressed, the vertical bar is moved one ruler increment to the left or right, selected fields in the **#ctrl** array are updated, and the ruler is redrawn to reflect the new left/right margin or indent position.

When the Left Margin command is used, the **%left**, **%wide**, **%indent**, and **%iwide** fields must be updated after each movement of the vertical bar. When the Right Margin command is used, the **%wide** and **%iwide** fields must be updated after each vertical bar movement. When the Indent command is used, the **%indent** and **%iwide** fields must be updated with each movement.

8.3.4 Obtaining New Tab Settings

When the tab command is chosen, the word **tabloop** is used for vertical bar positioning. While **tabloop** runs, it watches for six keys, and begaves as follows when they are pressed:

Key	Action
Left Leap	Vertical bar moves one ruler increment to the left.
Right Leap	Vertical bar moves one ruler increment to the right.
Tab	May either set, change, or remove a tab at the current **Shift-Tab** vertical bar location. The %tabs field in the **#ctrl** array is updated and the ruler redrawn.

Space — The vertical bar is positioned at the next tab stop to the right of its current position. If there are no tabs to the right, the vertical bar will wrap around to the leftmost tab stop. If there are no tabs at all, the vertical bar will not be moved.

Erase — All of the tab stops are removed.

8.3.5 Example of a Command That Uses the Vertical Bar

The definition of **Left** shows the construction of a paragraph format command which uses the vertical format bar for new format selection:

```
: Left  (  -> )
        preform                  ( 1. PREPARE FOR FORMATTING OPERATION. )
        %setl curop to           ( Set the current operation. )
                                 ( uses curop to check for valid keys )
                                 ( received. )
            2                    ( leftmost position for vertical bar )
        #left c@ #wide c@ + 2/   ( rightmost position for vertical bar )
        #indent c@ 2/ 2+         ( initial position for vertical bar )
        initset                  ( Initialize integers used by )
                                 ( marginloop.)

        marginloop               ( Watch while the user chooses their )
                                 ( settings. )

        #indent c@ ##ctrl %indent + c!  ( Place new format info )
        #left c@ ##ctrl %left + c!      ( in ##ctrl array. )

        ['] fixindent reform ;   ( Update the text and display. )
```

Refer to the routines summary, for more information on words used in the above listing.

8.4 PARAGRAPH FORMAT COMMANDS ROUTINES SUMMARY

8.4.0 Words Used by Paragraph Format Commands

Defindent (->)
(pronounced deff' in-dent)
Moves the default indent data from the **#defaults** table to the **##ctrl** array and uses **reform** to reformat the entire selection according to the default indent setting.

Defjustify (->)
(pronounced deff' jus'ti-fy)
Moves the default paragraph style data from the **#defaults** table to the **##ctrl** array and uses **reform** to reformat the entire selection according to the default paragraph style setting.

Defleft (->)
(pronounced deff' left)
Moves the default left margin data from the **#defaults** table to the **##ctrl** array and uses **reform** to reformat the entire selection according to the default left margin setting.

Defright (->)
(pronounced deff' ryt)
Moves the default right margin data from the **#defaults** table to the **##ctrl** array and uses **reform** to reformat the entire selection according to the default right margin setting.

Defspacing (->)
(pronounced deff' spay'sing)
Moves the default line spacing data from the **#defaults** table to the **##ctrl** array and uses reform to reformat the entire selection according to the default line spacing setting.

Deftabs (->)
(pronounced deff' tabs)
Moves the default tab setting data from the **#defaults** table to the **##ctrl** array and uses **reform** to reformat the entire selection according to the default tab settings.

fixindent (->)
(pronounced fiks' in-dent)
Transfers the contents of the **%indent** field in the **##ctrl** field to the **%indent** field in the **#ctrl** array. Defindent and Indent pass the token of **fixindent** to **reform** which executes it once at each format packet location in the effectively modified range of text.

fixjustify (->)
(pronounced fiks' jus-ti-fy)
Transfers the contents of the **%just** field in the **##ctrl** field to the **%just** field in the **#ctrl** array. Defjustify and Justify pass the token of **fixjustify** to **reform** which executes it once at each format packet location in the effectively modified range of text.

fixleft (->)
(pronounced fiks' left)
Transfers the contents of the **%left** field in the **##ctrl** field to the **%left** field in the **#ctrl** array. **Defleft** and **Left** pass the token of **fixleft** to **reform** which executes it once at each format packet location in the effectively modified range of text.

fixright (->)
(pronounced fiks' ryt)
Transfers the contents of the **%right** field in the **##ctrl** field to the **%right** field in the **#ctrl** array. **Defright** and **Right** pass the token of **fixright** to **reform** which executes it once at each format packet location in the effectively modified range of text.

fixspacing (->)
(pronounced fiks' spay-sing)
Transfers the contents of the **%lsp** field in the **##ctrl** field to the **%lsp** field in the **#ctrl** array. **Defspacing** and **Spacing** pass the token of **fixspacing** to **reform** which executes it once at each format packet location in the effectively modified range of text.

fixtabs (->)
(pronounced fiks' tabs)
Transfers the contents of the **%tabs** field in the **##ctrl** field to the **%tabs** field in the **#ctrl** array. **Deftabs** and **Tabs** pass the token of **fixtabs** to **reform** which executes it once at each format packet location in the effectively modified range of text.

Indent (->)
(pronounced in'dent)
Sets the current operation to **%seti**. Uses **initset** to set the **iposit** integer to the current indent location (#indent half-spaces), the **rbound** integer to the current right margin location (#left+#wide half-spaces), the **lbound** integer to two half-spaces, and then sits in a loop, **marginloop**, and moves the indent line around while the user selects a new indent position. When the user has finished setting the indent position, **Indent** moves the new user indent data to the **##ctrl** array and uses **reform** to reformat the entire selection according to the new indent setting.

Justify (->)
(pronounced jus'ti-fy)
The paragraph style field in a control/format array, **%just**, can accept four values: 0 for Left-Justified, 1 for Right-Justified, 2 for Centered, and 3 for Fully Justified. This field cycles between values in the following order:

Before...		After...
0	->	2
2	->	1
1	->	3
3	->	0

The new value is placed in the **%just** field of the **##ctrl** field. **reform** reformats the entire selection according to the new paragraph style setting.

Left (->)
(pronounced left')

Sets the current operation to **%setl**. Uses **initset** to set the **iposit** integer to the current left margin location (#left half-spaces), the **rbound** integer to the current right margin location (#left+#wide half-spaces), the **lbound** integer to two half-spaces and then sits in a loop, **marginloop**, and moves the left margin line around while the user selects a new left margin position. When the user has finished setting the left margin, **Left** moves the new user left margin data to the **##ctrl** array and uses **reform** to reformat the entire selection according to the new left margin setting.

Right (->)
(pronounced ryt')

Sets the current operation to **%setr**. Uses **initset** to set the **iposit** integer to the current right margin location (#left+#wide half-spaces), the **rbound** integer to **&horiz** half-spaces, the **lbound** integer to either the left margin location (#left half-spaces) or the indent location (#indent half-spaces), whichever is greater, and then sits in a loop, **marginloop**, and moves the right margin line around while the user selects a new right margin position. When the user has finished setting the right margin, **Right** moves the new user right margin data to the **##ctrl** array and uses **reform** to reformat the entire selection according to the new right margin setting.

Spacing (->)
(pronounced spay'sing)

The line-spacing field in a control/format array, **%lsp**, can accept three values: "2" for single-spaced text, "3" for 1½-spaced text, and "4" for double-spaced text. The table below shows before and after values:

Before...		After...
2	->	3
3	->	4
4	->	2

Each time **Spacing** is executed, it takes the contents of the **%lsp** field in the **#ctrl** array and transforms it according to the pattern shown in the table above. The new value is stored in the **%lsp** field of the **##ctrl** field. **reform** then reformats the entire selection according to the new line spacing setting.

Tabs (->)
(pronounced tabs')

Sets the current operation to **%sett**. Uses **initset** to set the **iposit** integer to the ruler increment location closest to the current cursor position, the **rbound** integer to 0 half-spaces, the **lbound** integer to 0 half-spaces. It then sits in a loop,

tabloop, and moves the vertical tab line around as the user sets up tab stops. When the user has finished setting tab stops, Tabs moves the new user tab data to the **##ctrl** array and uses **reform** to reformat the entire selection according to the new user tab settings.

8.4.1 Low-Level Paragraph Formatting Words

#defaults (-> a)
(pronounced sharp' dee'falts)
Pushes the address a of a table of default paragraph format settings on the stack.

pformat1 (-> f>)
(pronounced pee' for'mat wun')
Prepares the selection for a paragraph formatting operation by implacing paragraph format packets at the start and end of the selection, if necessary. A false flag means the operation was successful; a true flag means an error occurred. The steps in the execution of this word are as follows:

1. Checks to see if there is enough room in the undo buffer to hold at least two paragraph format packets which the Cat needs to change the format state before and after the affected region. If there isn't enough room, **pformat1** exits immediately and returns a true flag.
2. The format state before and after the selection must be found and saved so that it may be restored after the text in the selection has been formatted. The current state after the selection (which was previously found by **preform**) is saved into the **workpkt** scratch area.

   To find the previous format state, **prevbrk** and **brk+** find the address of the first break (or first paragraph format packet) before the **bos** position, or before the **bos nextchar** position if the cursor is wide. The address is saved in the **prepkt** system integer.

   To find the format state after the selection, **nextbrk** and **brk+** find the address of the first break (or paragraph format packet) after the **beot prevchar** position, or after the **beot prevchar prevchar** position if the cursor is is narrow. The address is stored in the **postpkt** system integer.
3. Now **pformat1** checks both **prepkt** and **postpkt** to see whether they contain addresses of breaks or format packets. The address belongs to a paragraph format packet if the first byte at the address contains the paragraph format packet identification marker, **&fmt**.

   If the preceding break is not followed by a format packet, **prepkt prevchar findchar** gets the previous formatting state information and **prepkt pktsize makespace makepkt** encodes the information and insert the resulting format packet after the break.

   If the succeeding break is not followed by a format packet, **postpkt prevchar findchar** gets the formatting state

information for the text which follows the selection and **postpkt pktsize makespace postpkt to postpkt makepkt** encodes the formatting information and to insert the resulting format packet after the break.

pformat2 (n ->)
(pronounced pee' for'mat too')

Executes the token n to modify the paragraph format data structure fields affected by the current formatting operation, and causes the text to be redisplayed.

Uses **killivls** to mark all intervals between **prepkt** and **postpkt beot max** for updating. Uses **partknown** to mark all intervals between **postpkt beot max** and the end of text as intervals whose interval table data is mostly correct.

Next, **pformat2** spins in a loop, finding paragraph format packets in the selection and updating specific fields in the format packet by executing the token provided.

After the format packets have been updated, **pformat2** finds the best way to redisplay the newly formatted text, trying to keep the end of the selection on the screen, if not on the same line on the screen.

If the entire selection fits in the window, **refresh** selectively redraws the window contents. Otherwise, **new-display** completely redraws the window contents. The text is marked as dirty and the selection is reduced to the single character at the beginning of the selection.

preform (->)
(pronounced pree' form)

Performs preparation tasks before the start of a paragraph formatting operation. Uses **lockedsel** to see if the current selection lies within a locked region of text. If it does, the operation is aborted. Loads control/format information which corresponds to the **beot** address, if the cursor is wide, or the **beot prevchar** address if the cursor is not wide, into the **#ctrl** array.

reform (n ->)
(pronounced ree-form')

The main word used by the paragraph formatting operations. Uses **pformat1** to prepare the selection for formatting. If the Use Front key is not currently pressed or if the last operation was not a paragraph formatting operation (**%pfmt**, **%setl**, **%seti**, **%setr**, or **%sett**), **reform** performs these undo preparations:

1. Checks for enough memory for the undo buffer
2. Saves copies of all format packets in the selection in the undo buffer, and uses **savepos** to save the current current state of the editor.
3. **reform** turns the **ufpressed?** system integer off, uses pformat2 to update the text and redraw the display, sets **unformat** as the undo operation, sets **%pfmt** as the current operation, and uses **rule** to update all of the paragraph format indicators in the ruler display.
4. Now **reform** waits in a loop to see what operation, if any, the

user will perform next.

5. As soon as the Use Front key is released or a non-special key is pressed, **reform** exits the loop and examines the key press scan code data.
6. If the keyboard information indicates that another paragraph format operation is to be performed next, **reform** will leave the selection extended so that the next operation acts upon the same selection. Otherwise, **reform** collapses the selection before returning.

unformat (->)
(pronounced un' for'mat)
This is the undo operation for **reform**. Uses **swappkts** to exchange the format packets saved by **reform** in the undo buffer with the packets in the text which were changed by **reform**. Marks all interval entries which correspond to the changed region of text for updating and all intervals beyond **postpkt beot max** as partially known intervals (same as **pformat2**). Swaps the contents of the **oldbos** and **bos**, and the **oldcstate** and **cstate**, then uses **eos-display** to redisplay the text, and **resetcursor** to fix up the cursor.

8.4.2 Tab Routines

addtab (n f ->)
(pronounced add' tab)
Adds a tab of type f, where f = -1 means decimal tab, and f = 1 means normal tab to the **#ctrl** tab array at position n.

deltab (n ->)
(pronounced dell' tab)
Deletes the tab at position n in the tab array from the **#ctrl** tab array.

getkey (-> f)
(pronounced get' kee)
Used by **tabloop** and **marginloop** to get the keys used for user specification of new tab, margin, or indent settings (to move the left/right margin line>, indent line, and tab line around). Returns a true flag if a valid key was pressed. A valid key is any Use Front key combination which is (a) not a paragraph format command key combination, or (b) is equal to the paragraph format command key which caused **getkey** to be called.

initkey (->)
(pronounced in-it' kee)
Stores a $FF in the **lastkey** system integer.

initset (n1 n2 n3 ->)
(pronounced in-it' set)
Initializes the system integers used during the setting of tabs, left margin, right margin, or indent. Sets the value of **iposit** to n3, **rbound** to n2, and **lbound** to n3. Stores a 0 in the first four bytes of the **vtbuff**. Uses **repos** to position the vertical

tab line at the location specified by **iposit** and uses **initkey** to set the initial **lastkey** value to $FF.

marginloop (->)
(pronounced mar'jin loop)

This is the loop used to help the user specify a new left margin, right margin, or indent position. While the Use Front key is held down and while **getkey** returns a true flag (indicating that a valid margin/indent positioning key has been pressed), **marginloop** checks for the occurrence of two keys: the left and right Leap keys. If the left Leap key is pressed, **repos** positions the vertical tab line one ruler increment to the left. If the right Leap key is pressed, **repos** positions the vertical tab line one ruler increment to the right.

nextab (n1 -> n2 f | If the next tab is found.)
(n -> 0 | If no next tab is found.)
(pronounced neks' tab)

Looks through the **%tabs** field in the **#ctrl** array for a tab stop that is to the right of the specified ruler position n. If there are no tab stops to the right of the specified position, **nextab** starts looking again starting from the left margin. If no tab is found, a "0" is returned. If a decimal tab is found, a "-1" is returned. If a normal tab is found, a "1" is returned.

repos (n ->)
(pronounced ree'pohs)

Repositions the vertical tab line at position n (specified in units of whole spaces) and updates the necessary fields in the **#ctrl** array. If **left** is the current operation, **repos** will update the contents of the **#left**, **#wide**, **#indent**, and **#iwide** fields. If **Indent** is the current operation **repos** will update the contents of the **#indent** and **#iwide** fields. If **Right** is the current operation **repos** will update the contents of the **#wide** and **#iwide** fields.

tabloop (->)
(pronounced tab'loop)

This is the loop used to help the user specify new tab settings. While the Use Front key is held down and while **getkey** returns a true flag (indicating that a valid tab positioning key has been pressed), **tabloop** checks for the occurrence of five keys: the left Leap key, the right Leap key, the Tab key, the Space Bar, and the Erase key. If a left Leap key is pressed, **repos** positions the vertical tab line one ruler increment to the left. If a right Leap key is pressed, **repos** positions the vertical tab line one ruler increment to the right. If the Tab key is pressed, **tabloop** will either add a tab (**addtab**), or delete (**deltab**) or change (**nextab**) the tab at the current vertical tab line position. If the Space Bar is pressed, **repos** will be used to position the tab line at the next tab stop location. If the Erase key is pressed, all tabs are removed.

unvtline (->)
(pronounced un' vee tee lyne)
Restores the bytes under the vertical screen line used for tab setting, left margin, right margin, and indent positioning. The saved copies of the image underneath the vertical line are stored in the **vtbuff**. The first 4-byte location in the **vtbuff** holds the byte position where the first saved byte image should be placed.

vtline (n ->)
(pronounced vee' tee lyne)
Places a full-screen vertical line at byte position n on the screen. Saves copies of the image bytes under the vertical line in the **vtbuff** memory buffer. The byte position n is saved in the first byte of the **vtbuff**.

8.5 PARAGRAPH FORMATTING INTEGERS

lastkey (pronounced last' kee)
Holds the previous key processed in formatting loop

lbound (pronounced ell' bownd)
Holds the left boundary for the vertical format line

posit (pronounced pahz-it)
Holds the instantaneous position, in pixels, of the vertical line

rbound (pronounced arr' bownd)
Holds the right boundary for the vertical format line

thiskey (pronounced this' kee)
Holds the key most recently processed in the formatting loop

vbheight (pronounced vee' bee hyte)
Holds the height of the vertical tab line expressed in pixels.
vbheight is defined as: **scans/image lines/screen ***

vtbuff (pronounced vee' tee buff)
Holds the address of the buffer used to hold the bits behind the vertical tab line. The size of the buffer is **vbheight** 6 +.

8.6 SCAN CODES FOR THE PARAGRAPH FORMAT KEYS

Paragraph Format Function	Scan Code	Character
Paragraph Style	$02	t
Line Spacing	$0B	u
Indent	$29	-
Right Margin	$31	=
Set/Clear Tab	$32	tab
Left Margin	$38	\

8.7 DEFAULT PARAGRAPH FORMAT SETTINGS

Default Paragraph Settings

Left Margin	7
Right Margin	73
Paragraph Style	Left Justified
Line Spacing	Single-spaced
Indent	0

Default Tab Settings

Country Code	Country	Tab Stops
00	USA	13,18,28,38,48,58
01	Canada	13,18,28,38,48,58
02	United Kingdom	13,22,32,42,52,58
03	Norway	13,18,28,38,48,58
04	France	12,22,32,42,52,62
05	Denmark	13,18,28,38,48,58
06	Sweden	13,18,28,38,48,58
07	Japan	13,18,28,38,48,58
08	West Germany	11,17,27,37,47,57
09	Netherlands	13,18,28,38,48,58
0A	Spain	17,27,37,47,57,67
0B	Italy	13,18,28,38,48,58
0C	Latin America	13,18,28,38,48,58
0D	South Africa	13,18,28,38,48,58
0E	Switzerland	13,18,28,38,48,58
0F	ASCII	13,18,28,38,48,58

9. DOCUMENT COMMANDS

Introduction

The Cat has two commands which operate on whole documents only: Document Lock and Local Leap. The words used to implement these commands are discussed here.

9.0 THE DOCUMENT LOCK COMMAND

The Document Lock command locks and unlocks the text of a document or a contiguous set of documents. When a document is locked, no changes can be made to it. A one-character wide gray bar is displayed along both edges of a locked document.

DocLock

DocLock is the word used to implement the Document Lock command. The actions of the Document Lock command are based on the current selection. Since attribute information about a locked document is kept in the document format packet, only whole documents can be locked.

The first action of **DocLock** is to expand the region defined by the current selection to the smallest region which contains both the entire selection and a whole number of documents. The documents in this expanded region are the documents that will be affected by the Document Lock command.

If the new region contains an unlocked document, or a combination of locked and unlocked documents, **DocLock** will lock the entire region. If all documents in the new region are unlocked, **DocLock** will lock them all. Similarly, if all documents in the new region are locked, **DocLock** will unlock them.

9.0.0 How Document Lock Affects Document Format and Calc Packets

Two items in a document are affected by the lock state of the document: document format packets and Calc packets.

If a document is locked, its **#lock** field will contain the value **lok**. If a document is unlocked, its **#lock** field will contain the value **markbl**. **lok** and **markbl** are two Cat display characters. **lok** corresponds to a gray box character which is the width and height of one character. **markbl** is a white space character which is the width and height of one character. When text is displayed, the value in the **#lock** field is always placed in the first and last character position of each line of text. This explains why a locked document always has a one-character wide gray bar along the left and right edges of the screen display area.

If a Calc packet is locked, it is marked by a **&lockedcalc** token in the text. If a Calc packet is unlocked, it is marked by a **&calc** token in the text. This indicates to the Calc command that the results should not be changed in the text.

9.0.1 Undoing the Document Lock Command

Before changing a document's lock state, **DocLock** saves the document's current lock state, as found in the **#lock** field of the document's document format packet, in the undo buffer. When **undoclock** undoes a **DocLock** operation, it uses the saved state to determine whether a particular document, and any Calc packets in the document, should be locked or unlocked.

9.0.2 Words That Check the Lock State

The words **lockedtext?**, **lockedsel**, and **lockedrange?** check the lock state at a particular location in the text. **lockedsel** checks for a locked selection range, **lockedtext?** checks for a locked character, and **lockedrange?** checks for a locked range of characters.

9.1 THE LOCAL LEAP COMMAND

Local Leap, a toggle command, defines the range over which the Leap command can operate. The word **local/global** implements Local Leap.

If Local Leap is used when the current range of leap is the entire text, **local/global** will shrink the Leap range to the smallest whole document range which entirely contains the current selection. This means that a partially highlighted document will be included in the new local leap region.

If Local leap is used when the current Leap range is restricted, that is, when it does not include the entire text, **local/global** will expand the leap range to include the entire text.

bor (beginning-of-range) and **eor** (end-of-range) are the two system integers used to hold the start and end addresses of the current leap range. Whenever **local/global** expands or reduces the leap range, it also adjusts the **op** and **pop** pointers to fit within the newly defined leap range.

The undo operation for **local/global** is **undolocal/global**.

9.2 UPDATING DOCUMENT FORMAT PACKETS

The word **redoc** can be used to integrate document format information specified by the Setup command into all document format packets contained within the current selection range.

9.3 ROUTINES SUMMARY

9.3.0 Locked Document Routines

Doclock (->)
(pronounced dahk' lahk)
Locks or unlocks all documents contained within the current selection, or which contain the current selection. Currently only whole document(s) may be locked/unlocked. **DocLock**'s first action is to define the range of text to be locked/unlocked. The first document separator located before the start of the selection will be used as the start of the locked/unlocked range, and the first document separator located after the end of the selection will be used as the end of the locked/unlocked range.

Next, **DocLock** must determine whether the range of text should be locked or unlocked. If any of the documents within the range are currently locked, **DocLock** will unlock all documents in the range. Otherwise, all documents in the range will be locked. The exception to this rule occurs when **DocLock** is used for the first time during a prerecorded Learn sequence. If this is the case, the documents in the range will always be forced to the locked state.

Now **DocLock** may begin the locking/unlocking process. It steps through the range looking for all occurrences of Calc packets and document format packets. If a Calc packet is to be locked, a **&lockedcalc** token will be placed in the packet. Otherwise, a **&calc** token will be placed in the packet. If a document format packet is encountered, **getdpkt** reads its contents into the **#ctrl** array and the current value of its **#lock** field is stored in the undo buffer. Then, the **#lock** value is changed to either **lok** (if the document is being locked) or **markbl** (if the document is being unlocked), and **makedpkt** writes out the changed document format packet.

After all Calc and document format packets have been properly updated, the undo operation is set to **undoclock**, all intervals which correspond to the affected range are marked as changed, the display is redrawn to show or hide the lock bars, and the text is marked as dirty.

lockedrange? (a1 a2 -> f)
(pronounced lahkt' raynj kwes'chun)
Checks to see if any characters in the range specified by the addresses a1 and a2 lie in a locked region of text. If they do, a true flag is returned.

lockedsel (-> f)
(pronounced lahkt' sell)
Checks to see if the selection lies within locked range of text, or contains a locked range of text. An abort message is issued if it does.

lockedtext? (a -> f)
(pronounced lahkt' tekst kwes'chun)
Uses **findchar** to read the control/format information for the character at address a into the **#ctrl** array. Then checks the contents of the **%lock** field to see if the character lies in a region of locked text. If the text is locked, a true flag is returned.

undoclock (->)
(pronounced un' dahk lahk)
The undo operation for **DocLock. undoclock**'s first action is to define the range of text to be locked/unlocked. The first document separator located before the start of the selection will be used as the start of the locked/unlocked range and the first document separator located after the end of the selection will be used as the end of the locked/unlocked range.

Now **undoclock** may begin the locking/unlocking process. **undoclock** steps through the range looking for all occurrences of Calc packets and document format packets. If a document format packet is encountered, the previous value of the packet's **#lock** field is obtained from the undo buffer and sets a local lock/unlock flag.

getdpkt reads the packet contents into the **#ctrl** array and the current value of its **#lock** field is stored in the undo buffer. Then, the **#lock** value is changed to either **lok** (if the document is being locked) or **markbl** (if the document is being unlocked), and **makedpkt** writes out the changed document format packet.

If a Calc packet is to be locked, a **&lockedcalc** token will be placed in the packet. Otherwise, a **&calc** token will be placed in the packet.

After all Calc and document format packets have been properly updated, the undo operation is set to **undoclock**, all intervals which correspond to the affected range are marked as changed, the display is redrawn to show or hide the lock bars, and the text is marked as dirty.

adjustleaprange (->)
(pronounced a-just' leep raynj)
Shrinks the allowable leap range according to the current selection. The start of the leap range is set to the first document break before the start of the selection; the end of the leap range is set to the first document break after the end of the selection.

checklocallight (->)
(pronounced chek lo'kil lyte)
Checks the current leap range and sets the LOCAL indicator light accordingly. If leaping over the entire text range is currently possible (**bor bot = eor eot = and**), the LOCAL indicator is turned off. Otherwise, the "LOCAL" string is displayed in the indicator.

local/global (->)
(pronounced lo'kil slash glo'bil)
Toggles between local and global leaping. The current **bor** and **eor** values are saved in the undo buffer. The undo operation is set to **undolocal/global**. If the leap range is currently fully expanded, or if this is the first use of **local/global** in a prerecorded Learn sequence, then **adjustleaprange** shrinks the leap area to the smallest allowable leap range which contains the current selection. Otherwise, the leap range is fully expanded by setting **bor bot =** and **eor eot =**. The **op** and **pop** values are adjusted to fall within the new local leap area, **checklocallight** handles the LOCAL indicator light, and the text is marked as dirty.

undolocal/global (->)
(pronounced un-doo' lo'kil slash glo'bil)
Undo operation for **local/global**. The current **bor** and **eor** values are swapped with the **bor** and **eor** values saved in the undo buffer, the **op** and **pop** values are adjusted to fit within the new leap range, **checklocallight** handles the LOCAL indicator light, and the text is marked as dirty.

9.3.1 Document Format Packet Update Routines

findds (a1 a2 -> a3-or-0)
(pronounced fynd' dee ess)
Searches through the range of text which starts at address a1 and ends at address a2 for the first occurrence of a document separator character. If a document separator character is found in the specified text range, its address, a3, is returned. Otherwise, a 0 is returned. Used by **redoc**.

redoc (->)

(pronounced ree' dahk)

Updates all document format packets in the current selection. For each document format packet encountered, **getdpkt** places its current document state information into the **#ctrl** array and then **getdocpkt** updates the document state information with values from the set-up version of the document format information. **makedpkt** writes the updated document information in the **#ctrl** array back over the original format packet. After all document format packets have been updated, the affected intervals are marked as changed and the text is redisplayed.

10. LEAP

Introduction

The leap mechanism, which makes use of the two Leap keys (Leap Forward and Leap Backward), performs six basic operations:

1. Cursor placement (leaping from place to place in the text)
2. Display scrolling (using Shift-Leap)
3. Creeping (tapping a Leap key to move the cursor forward or backward character-by-character)
4. Spell Check Leap (pressing Undo while holding a Leap key down)
5. Text selection (press both Leap keys to select)
6. Text movement (dragging)

Three of these operations are repeatable: Leap (Leap Again appears on the fronts of the Leap keys), Scroll, and Spell Check Leap.

The routines used to implement the leap mechanism are primarily text search routines. In order to place the cursor at the string-specified location in the text, the Leap routines must be able to find the strings in the text which match the user's leap string. An optimized text string search algorithm called the Boyer-Moore algorithm is used by the Leap search routines to achieve maximum Leap performance.

10.0 THE BOYER-MOORE FAST STRING SEARCH ALGORITHM

The Boyer-Moore algorithm, by eliminating the need to look at each successive character in order to find an occurrence of a particular character sequence in the text, provides an optimal method for performing fast string searches.

When presented with a character from the text, the algorithm determines the distance to the next significant character position. The characters between the current character and the next significant character need not be analyzed and are skipped. To make decisions about significant character positions, the algorithm depends upon information from a preconstructed pattern table, discussed next.

10.0.0 The Pattern Table

The pattern table is a 256-byte table. There is one byte-length entry in the table for each of the 256 possible ASCII characters. The pattern table is constructed prior to the start of a search with **buildtable**. The data required to build the pattern table are

1. the address of the character string (pattern) being searched for
2. the length of the pattern
3. the direction of the search

The address of the pattern table is kept in the **ptable** integer. The address of the pattern string is kept in the **pattern** integer, and the length of the pattern is kept in the **patlen** integer. If a forward search is being used, the **direction** integer will hold a true flag.

The data to be used for this discussion are shown below. We are searching for the 3 character string "hij". The text we are searching through contains the first 16 characters of the alphabet:

Pattern:	**hij**
Pattern Length:	3
Text Being Searched:	abcdefg**hij**klmnop
Search Direction:	Forward (start-of-text to end-of-text)

Given this information, **buildtable** will construct the pattern table shown below. Note that the entries for all characters not included in the pattern contain either a 3 (the length of the pattern) or a "1". In a forward search, if a character is the first character in the pattern, its entry will hold the value length-1. In this case, h is the first character in the pattern so its entry contains "3-1=2". The second character in the pattern, i, receives length-2 or 3-2=1. The last character in the pattern, which turns out to be the most important character in a forward Boyer-Moore search, receives the value FF.

Pattern Table for a Forward Search for the String "hij":

Second hex digit	First Hex Digit -> 0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
1	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
2	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
4	3	3	3	3	3	3	3	3	**2**	**1**	**FF**	3	3	3	3	3
5	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
6	3	3	3	3	3	3	3	3	**2**	**1**	**FF**	3	3	3	3	3
7	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
8	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
9	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
A	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
B	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
C	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
D	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
E	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
F	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

10.0.1 The Character Equivalence Table (**maptable**)

Note that "h" and "H" receive the value 2 in the pattern table. This is because a leap to a lowercase "h" should find both upper and lowercase "h". When **buildtable** creates the pattern table it refers to a table of character equivalents (shown below). The address of the character equivalence table is kept in the **maptable** integer. This is another 256-byte table. Each entry in this table contains the ASCII code for the character, if any, that is considered to be equivalent to the character to which the entry corresponds. In general, the uppercase version of any character is considered to be equivalent to its lowercase version. The converse is not true. The search used by Leap is not case-sensitive unless the Shift key is pressed along with a character. If the Shift key is pressed and a character key is struck, the character will be considered to be strictly uppercase. If a character has no equivalent, its entry in the **maptable** will contain a 0.

Character Equivalence Table:

Second hex digit	First Hex Digit -> 0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	0	93	0	0	70	0	0	87	0	0	0	0	0	0	0
1	0	0	0	0	61	71	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	62	72	0	0	0	91	0	0	0	0	0	0
3	0	0	0	0	63	73	0	0	84	0	0	0	0	0	0	0
4	0	0	0	0	64	74	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	65	75	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	66	76	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	67	77	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	68	78	0	0	89	0	0	0	0	0	0	0
9	0	0	0	0	69	79	0	0	0	0	0	0	0	0	0	0
A	0	0	0	0	6A	7A	0	0	0	0	0	0	0	0	0	0
B	0	0	0	0	6B	0	0	0	0	0	0	0	0	0	0	0
C	0B	0	0	0	6C	0	0	0	0	0	0	0	0	0	0	0
D	0	0	0	0	6D	0	0	0	8C	0	0	0	0	0	0	0
E	0	0	0	0	6E	0	0	0	0	0	0	0	0	0	0	0
F	0	0	0	0	6F	0	0	0	86	0	0	0	0	0	0	0

10.0.2 A Step-by-Step Explanation of the Algorithm

Pattern: hij
Text Being Searched: abcdefghijklmnop

1. The search starts, more or less, at the current cursor position.

   abcdefghijklmnop
   ^

2. Since our string is three characters long, the first character to be examined by the search routines will be the third character in the text, the c.

   abcdefghijklmnop
     ^

3. The **ptable** data for the character c determines the location of the next character to check. The entry for c in the **ptable** contains a 3.

What we know at this point:

We know that c is not the last character in the pattern because its **ptable** entry does not contain a $FF.

Since we are on the third character in our text and it is not the last character in the pattern, there is no way any of the characters we skipped over could contain the pattern.

Because we do know that we could not have skipped over a possible match, we can skip ahead another full pattern length number of characters (3).

4. The next character we encounter is an f.

   abcdefghijklmnop
      ^

   Its **ptable** entry contains a 3 also. For the same reasons described in Step 3 above, we will skip ahead another three character positions.

5. The next character we encounter is an i.

   abcdefghijklmnop
         ^

   The entry for i in the **ptable** contains a 1. Whenever the search routines encounter an i, which is the next to last character in the pattern, they must be sure to check the character which follows the i.

6. After advancing by one character, we encounter a j.

   abcdefghijklmnop
         ^

   j is the last character in the pattern because its **ptable** entry contains a $FF. Now the search routine knows it has a possible match. Only at this point will it take the time to explicitly compare each character in the pattern with the possible match in the text.

7. The text string matches the pattern so the search is finished. The cursor is placed over the h.

If a normal search -- a "compare-each-character-in-the-text-to-the-pattern" search -- had been used, eight character comparisons would have been performed before the match was located. With the Boyer-Moore method, only three numeric comparisons and two character comparisons were required. On the negative side, the Boyer-Moore search does require extra time to create the **ptable**. As the length of the search pattern increases, the speed of the Boyer-Moore search surpasses the speed of the conventional string search, even when the table-building time is taken into account.

10.0.3 Handling Accent Characters

In the **ptable** you will notice that all of the entries from $B0 up contain a 1. In this range the only entries which correspond to characters found in the text are the entries $B0 -> $B8 and $C0 -> $C8. These are the entries for the accent characters. As you may recall from previous explanations of accents and accented characters, an accented character such as ñ is stored as a 2-byte value in the text. The first byte holds the character code for the main character, the n, and the second byte holds the code for the accent.

The search routines will only pay attention to the main characters in the text unless an accent is specifically included in the pattern. If the search routine happens to land on the data for ~ part of ñ, the 1 in the **ptable** entry will cause the search to be advanced by one, effectively skipping over the accent character. This means that both the word "Canada" and "Cañada" will be found with the pattern "can."

If the more specific pattern "cañ" is used, only "Cañada" will be found by the search routines.

10.1 THE LEAP MECHANISM

The word called when either Leap key is pressed or another key is pressed while a Leap key is down is **do-lex**. Since many operations are supported by the Leap mechanism, **do-lex** has the responsibility of deciding which of them it should perform.

10.1.0 Initializing a Leap Operation

The first action of **do-lex** is to check the contents of the **lexxing** integer to see if a leaping operation is already in progress. If leaping is not already in progress, **init-lex** performs general leaping preparations.

During leaping, all characters typed are considered to be lowercase unless they are deliberately typed while the Shift key is down. The Lock key has no effect, and the Lock light is turned off during leaping. The previous state of the Lock key is saved in the **oldshiftlock** integer. **savepos** saves the state of the current cursor-state integers. **newlex** and **lexxing** are turned on to indicate that a new Leap is being initiated and that leaping is occurring. **matched** is turned off to indicate that a match has not yet been found. **leftlex?** is turned on if the left Leap key, Leap Backward, is pressed. The **p** (place) integer is set to point at the end of the current selection. The **extbos** and **savebos** integers are prepared in case a drag operation occurs.

10.1.1 Leaping Around the Text

Whenever a character is pressed while a Leap key is down, **do-lex** is called with the character on the stack. If the character is not a special key character, the word **searching** will be used to try to build up the pattern and start the search.

If the character passed to **searching** corresponds to the Undo key, and **pattern** holds a -2 value or a new leap operation has just started, **searching** is being asked to pass control to the Spell Check Leap command. Therefore **searching** discards the character and calls **spellcheckleap**.

First, **searching** must use the character to update the pattern. If the character corresponds to the Erase key, **pattdel** will be used to remove a character from the end of the pattern. If the character is any other text character, **pattadd** will be used to append the character to the current pattern. **pattadd** is smart enough to handle accent-character merging (like **Insert**). **pattadd** contains a **begin...again** loop and, once called, will continue calling **<?k>** and adding or deleting characters from the pattern until no valid characters are available. This means that if the pattern is typed quickly, some intermediate character patterns may not be searched.

Once **pattadd** completes, the **ptable** is built and the search is started. If a single-character pattern occurs, **searching** will resort to the use of a standard string comparison search. The search for a repeated single character pattern is the worst case search on the Cat.

When the search has completed, **end-search** updates the cursor position and display if necessary.

10.1.1.0 Low-Level Search Routines

Text searches on the Cat will wrap around from the starting search position to the end of the text, then to the start of the text and back to the start position, if necessary.

The low-level routines used to perform the searching are **<search<>**, **<search>>**, **search<**, and **search>**. The word **<search<>** searches in a forward direction through a specified range of text. The word **<search>>** searches in a backward direction through a specified range of text. None of these four routines are aware of the **gap** area and are able to skip over it. The words **search<** and **search>** are slightly higher level search words which directly use **<search<>** and **<search>>**.

10.1.1.1 Searching for Single Page Breaks

Page-to-page leaping using a pattern consisting of a single page break is a special case because the cursor stops on both explicit and implicit page breaks. This is unusual because implicit page breaks have no corresponding character in the text and would not normally be found using the leap search algorithm described previously.

This special leap case is handled within **search>** and **search<**. If these routines find that the pattern contains only a single page break character, they will use the page routines **page?** and **prevpage** to find the next implicit or explicit page break.

Although these page break searches are not as optimized as the Boyer-Moore search, the largest area they will be asked to search is approximately 2K bytes, the maximum amount of data which can be held on a single page.

10.1.2 Scroll Again

If **do-lex** detects that the key passed to it corresponds to a Use Front key (which indicates that a repeat operation is being requested), it will first check to see if **pattern** holds a -1. If **pattern** does hold a -1, **lex-scroll** will be used to scroll the screen once and **do-lex** will then terminate execution. As described in **finish-lex**, whenever the display contents are scrolled up or down by holding down a Shift key and pressing a Leap key, a 4-byte -1 value will be placed in the **pattern** integer. If **do-lex** receives a Use Front key press, it will know

that it should repeat the scrolling operation.

Before **do-lex** terminates execution, autorepeating will be turned on for the [Use Front]-[Leap key] combination so that the screen will scroll continually as long as Use Front and Leap are pressed.

10.1.3 Spell Check Leap Again

If **pattern** holds a -2 value when a Use Front key press is passed to **do-lex**, it means a Spell Check Leap command is being repeated. The first time Spell Check Leap is used, it stores a -2 into **pattern**. **do-lex** will turn on the SPELLCHECK indicator light and call **spellcheckleapagain**. After autorepeating is turned on for the [Use Front]-[Spell Check Leap] key combination, **do-lex** will be exited.

10.1.4 Search Again

If a Use Front key press is received, and **pattern** contains none of the special flag values described above, **do-lex** knows that it is being asked to repeat a search for the current leap pattern. The **ptable** will be constructed for the current pattern and **research** performs the searching.

10.1.5 Finishing a Leap Operation

After checking for the start of a new leap operation, **do-lex** performs an opposite test and checks for the termination of a leap operation. If neither the left nor the right Leap key, nor the left nor the right Use Front keys are down, leaping is terminating and **finish-lex** is executed.

finish-lex will restore the previous Shift Lock key state, re-enable the cursor, and set the **lexxing** integer to false in order to indicate that leaping is no longer occurring. Next, **finish-lex** must determine which type of leap operation is being completed.

10.1.6 Shift-Leap Scrolling

If **newlex** is true (new leap operation), **matched** is false (no pattern was successfully matched during this leap operation), and a Shift key is down, **finish-lex** is being executed because Shift and Leap were pressed and released. The termination of Leap under these conditions will cause **lex-scroll** to be executed and a -1 to be placed in **pattern**. This makes **lex-scroll** the current Leap-repeat command.

10.1.7 Creeping

Creeping, or character-by-character cursor movement, takes place when a Leap key is pressed and released without a pattern being entered. Internally, if **newlex** is true (a new leap operation begun), **matched** is false (no pattern was successfully matched), and a Shift key is not down, **finish-lex** must be being executed. This will cause **lex-tap** to be executed. No flag will be placed in **pattern** because creeping is not a repeatable leap operation. **lex-tap** will cause the cursor to creep (move) one character position to the right or left.

10.1.8 Other Leap Terminations

If **newlex** is false (not a new leap operation) or **matched** is true (successful search), then **finish-lex** was called because either:

1. The Leap keys were used to highlight a selection and now they are being released.

2. The Leap keys were used to choose a new location for a section of text and are now being released so the drag operation can be performed.

3. A successful Leap or Spell Check Leap is finished, or...

4. An unsuccessful Leap or Spell Check Leap is finished.

10.1.8.0 Highlighting a Selection

If **finish-lex** is called when the selection is expanded, **leave-extended** ensures that the newly highlighted selection is left highlighted, and then **finish-lex** is exited.

10.1.8.1 Dragging a Selection

If **finish-lex** is called when the cursor is either split or extended, dragging is desired. If the drag destination location is still within the highlighted text to be moved, the text cannot be dragged and the selection is collapsed and a narrow cursor is placed at the drag destination. If the drag destination is valid, **drag** moves the text and **finish-lex** is exited.

10.1.8.2 Successful Spell Check Leap

If **finish-lex** is executed when **pattern** contains a -2, a successful Spell Check Leap has finished. **p** is saved in **pop** and the **op** is set to point to the end of the misspelled word so that if the user presses both Leap keys, the misspelled word will be selected and could be conveniently added to the user dictionary with Add Spelling.

10.1.8.3 Successful Leap

If **finish-lex** is executed when **pattern** did not contain -2, a successful Leap has finished. The cursor state integers are updated.

10.2 LEAP ROUTINES SUMMARY

advanceptr (a n -> a')
(pronounced add-vans' poynt'er)
Uses **nextchar** to advance the address a by n text character positions. n must be a positive value. The new address a' is returned on the stack. This word adjusts the starting point of searches.

buildtable (a n f ->)
(pronounced bild' tay'bl)
Builds the 256-byte Boyer-Moore search table used during leap searches. a is the address of the string to be searched for. n is the length of the string and f is a flag which indicates the search direction. If the flag is true, the search will proceed forward in the text.

First, **buildtable** will fill all table entries between offset $00 and $B0 with the length n of the string. All entries from offset $B0 to $FF will be filled with $01. Next, **buildtable** will selectively alter the entries corresponding to characters found in the search string. If the search is a forward search, the first character in the string, and all equivalent characters (as determined by the **maptable**) will be given the value n-1. The second character will be given the value n-2 and so on. The last character in the string will be given the special value $FF.

If the search is a backward search, the last character in the string, and all equivalent characters (as determined by the **maptable** will be give the value n-1. The second to last character will be given the value n-2 and so on. For information on the use of this table, refer to the discussion of the Leap search algorithm in this chapter.

do-lex (c ->)
(pronounced doo' leks)
The word called when a Leap key is pressed. Since the Leap keys are used for several types of operations, **do-lex** must coordinate all these operations. The cases **do-lex** must handle are

1. The first detection of a Leap key
2. The release of all Leap keys
3. The pressing of both Leap keys (highlights a selection)
4. When a new character has been added to the search pattern
5. [Use Front]-[Leap] scrolling
6. Creeping
7. Spell Check Leaping
8. Re-searching for a previous pattern (Leap Again while the Leap key is down or when it has been released and then pressed again)

If (1) occurs, **init-lex** will be used to initiate a new leap operation. If neither the left nor the right Leap key nor a Use Front key is down, (2) has occurred. The character c will be discarded, **finish-lex** will be used to terminate the leap, scroll, or creep, and **do-lex** will be exited. If (2) didn't occur, the **%lex** value is placed in the **curop** integer to indicate that Leap is the current operation. If (3) occurs, both Leap keys are down

and the cursor is not split, the character c will be discarded, the selection will be expanded, and **do-lex** will be exited. If (3) didn't occur, **direction** is set to -1 for forward leap, or 0 for backward leap. If c is a break or printable ASCII character, (4) **searching** will add the character to the leap pattern and search through the text. If the character is a Use Front key downstroke, then **do-lex** is being asked to repeat a previous action. If **do-lex** is being asked to repeat an action for the first time (**newlex** holds a true flag), **do-lex** will set **newlex** to false. Now, **do-lex** will check to see if (5) is occurring. If the first four bytes of **pattern** hold a -1, this is a signal that scrolling was the last operation, so **do-lex** will execute **lex-scroll**, then terminate. A -2 in **pattern** means case (6) is occurring. **do-lex** will turn on the SPELLCHECK indicator light, call **spellcheckagain** to handle the request, check for a panic key and set autorepeat accordingly, and then will terminate execution. If none of the other cases are occurring, **do-lex** is being asked to repeat the search of a normal leap string. The selection, if any, will be collapsed, **start-search** will be used to prepare for the repeat search, autorepeat is properly set up for the search, **buildtable** creates the Boyer-Moore table for the search, and **research** performs the search.

end-search (a -> f)
(pronounced end' sertch)

Terminates a Leap or drag operation. The address a on the stack is the result returned by the leap search routines. A copy of this value is placed in the **matched** integer. If this address is non-zero, a leap string match was found in the text. **end-search** will reposition the cursor position **cpos** to a if it is non-zero or to **gap prevchar** (the original leap start position) if a is zero. Then, if the cursor state was not extended or split, as it would be if a drag operation were being performed, the **bos** will be updated with the new cursor position and **eos** will be updated with the **bos nextchar** address. The screen display is updated and the cursor is set to a narrow state. Finally, if **matched** indicates that the pattern was not matched by the Leap search routines and a Learning operation is occurring (either recording or playing back), **end-search** will abort the Learn operation. **end-search** performs other actions related to the dragging of text. These actions will be discussed in the following section on dragging.

finish-lex (->)
(pronounced fin'ish leks)

This routine is called when a leap operation is completed, that is, when all the Leap and Use Front keys are released. Shift Lock is restored to the state it was in before a Leap key was pressed, the cursor will be reenabled, and the **lexxing** integer will be set to false to indicate that leaping is no longer occurring.

Next, **finish-lex** determines which type of leap operation is being completed. If **newlex** is true and **matched** is false, then **finish-lex** is executing a scroll if a Shift key is down, or creep if a Shift key is not down. If **newlex** is false or **matched** is

true, then **finish-lex** was called because either (1) the Leap keys were used to highlight a selection and now they are being released, (2) the Leap keys were used to choose a new location for a section of text and are now being released so the drag operation can be performed, (3) a successful Leap or Spell Check Leap is finished, or (4) an unsuccessful Leap or Spell Check Leap is finished.

If (1) has occurred, **leave-extended** ensures that the newly highlighted selection is left highlighted and then **finish-lex** is exited.

If (2) has occurred, the drag destination location is still within the highlighted text to be moved, the selection is collapsed and a narrow cursor is placed at the drag destination. If the drag destination is valid, **drag** moves the text and **finish-lex** is exited.

If (3) has occurred, and the search pattern contains a -2, a successful Spell Check Leap has finished. **p** is saved in **pop** and the **op** is set to point to the end of the misspelled word so that if the user presses both Leap keys, the misspelled word will be selected and could be conveniently added to the user dictionary with ADD SPELLING.

If (3) occurred and the search pattern did not contain -2, a successful Leap has finished. The previous old selection range, marked by the **op** and **pop** integers, is updated. The current **op** value is placed in **pop** and the current **p** value is placed in **op**. The **forceop** integer is set to true and **unmove** is set as the undo operation.

If the **bos** lies within the current leap range, the cursor is set to narrow. If the **bos** is at the start of the leap range, the cursor is set to wide.

If (4) occurred, **clearlearn** aborts any Learn activity and the cursor is reset to its pre-Leap state.

init-lex (->)
(pronounced in-it' leks)
Performs initialization at the start of a new leap operation. The state of the shiftlock key is saved and then the shiftlock is turned off. The undo buffer is cleared if the previous operation was not an **uncreep** or **unscroll**. **savepos** saves the state of the current cursor state integers. **newlex** and **lexxing** are turned on to indicate that a new Leap is being initiated and that leaping is occurring. **matched** is turned off to indicate that a match has not yet been found. **leftlex?** is turned on if the left Leap key (Leap backwards) is pressed. The current **beot** **prevchar** address is saved in the **p** and **savebos** integers. The current **bos** address is saved in the **extbos** integer.

leave-extended (->)
(pronounced leev' eks-tend'ed)
Leaves a leap operation with the cursor extended.

lex-scroll (->)
(pronounced leks' skroll)
This word is called when the [Shift]-[Leap key] combination is used. Depending on which Leap key is pressed, **lex-scroll** will

scroll 1, 1½, or 2 lines of text off the top or bottom of the screen (depending on the Line Space setting). **lex-scroll** will store a -1 in **pattern** to indicate to **do-lex** that the last operation was a scroll. **%scroll** will be placed in the **curop** integer and **forceop** will be turned on to indicate that the op should be advanced when the next key is typed.

lex-tap (->)
(pronounced leks' tap)
Tries to advance the cursor forward or backward by one character. If the current undo operation is not **uncreep** (if we are not undoing a previous **lex-tap**), the undo buffer is cleared and cursor state information is saved. Next, **lex-tap** checks the current cursor state and advances the cursor accordingly. If the cursor is extended and the left Leap key is pressed, **tapmove** will be used to collapse the cursor on the first character of the selection. The cursor will be narrow. The previous end of the extended selection (**eos prevchar**) will be saved in the **op** integer. If the cursor is extended and the right Leap key is pressed, the cursor will be collapsed on the last character in the selection. The cursor will be wide. The previous start of the extended selection (**bos**) will be saved in the **op** integer. Otherwise, if the cursor is not extended, **lex-tap** will make the cursor narrow if necessary and then will use **tapmove** to move the narrow cursor one character to the left or right.

pattadd (c ->)
(pronounced pat' add)
Adds the character c to the existing pattern pointed to by the address in the **pattern** system integer. Before adding the character to the pattern, **pattadd** checks to make sure the addition of the character will not cause the string to become longer than the maximum allowable leap string length (**patternsize** = 256 bytes). If there is enough room, and the character value is greater than $ff, **w!** places the character into the string. If the character value is less than $ff, **c!** places the character in the string. If the character is a single byte value, **pattadd** will also check to see if the current character is an accentable character which is preceded in the string by a bare accent character. If so, the 2 characters will be swapped to form a single accented character. After the character is added, the pattern length, kept in **patlen**, is incremented by one. Before terminating execution, **pattadd** will use **<?k>** to see if another key has become available. If a key is available and a Leap key is down, and the key is not a special key or Undo or Erase, **pattadd** will immediately add the character to the leap string pattern. **pattadd** will continue adding characters to the leap string until **<?k>** indicates that no more characters are available.

pattdel (->)
(pronounced pat'dell)
Removes a character from the leap string pattern by decrementing the contents of the **patlen** integer by one. **pattdel** will only decrement **patlen** if **patlen** holds a non-zero value.

pbpat (-> f)
(pronounced pee-bee' pat)
Returns a true flag if the leap string pointed to by pattern contains only pagebreak characters. Returns a false flag if any character in the leap string is not a page break character. This special pattern leaps to the end or start of the leap region.

research (->)
(pronounced ree'serch)
This word is executed when the Leap Again command ([Use Front]-[Leap])repeatedly searches for a previously specified pattern. If a forward search is being used, **research** must double-check its current position before starting the search. If the cursor is currently sitting on the pattern and a forward search for the pattern is started, the search routines will endlessly find the pattern at the start position. In this case, the start position must be advanced by **patlen** before the search begins. The text is then searched back to the start position if necessary. If a backward search is used, **research** does not have to worry about the start position. The text is searched towards the start of text and back to the start position if necessary.

search< (a1 a2 -> a3-or-0)
(pronounced serch bak'wurd)
High-level leap search word. Searches backward through the range of text which starts at address a1 and ends at address a2 looking for a string in the text which matches the leap string whose address is in the **pattern** system integer. If the search range is invalid (if the end address is greater than the start address) or if there is no leap string to match (**patlen** = 0), will place a zero on the stack and exit immediately. If the pattern contains a single page break character, the words **page?** and **prevpage** find the next page break in the text. Otherwise, **<search>>** performs the search. If **<search>>** cannot match the pattern, **pbpat** sees if the pattern contains any page break characters. If it does, leap matches the last or first document break in the current leap region.

search> (a1 a2 -> a3-or0)
(pronounced serch for'wurd)
High-level leap search word. Searches forward through the range of text which starts at address a1 and ends at address a2 looking for a string in the text which matches the leap string whose address is in the **pattern** system integer. If the search range is invalid (if the end address is less than the start address) or if there is no leap string to match (**patlen** = 0), will place a zero on the stack and exit immediately. If the pattern contains a single page break character, the words **page?** and **prevpage** find the next page break in the text. This is how implicit page breaks can be matched. Otherwise, **<search><** performs the search. If **<search><** cannot match the pattern, **pbpat** is used to see if the pattern contains any page break characters. If it does, leap matches the last or first document breaks in the current leap range

<search<> (a1 a2 -> a3-or-0)
(pronounced brak'it serch bak-wurd)
Searches backward through the range of text which starts at address a1 and ends at address a2 looking for a string in the text which matches the leap string whose address is in the **pattern** system integer, and the length is in **patlen**. If a matching text string is found, the address a3 of the string is returned on the stack. Otherwise, zero is returned. **<search<>** uses the Boyer-Moore table pointed to by the **ptable** system integer to locate potential matches and then performs a string comparison to explicitly validate the match.

<search>> (a1 a2 -> a3-or-0)
(pronounced brak'it serch for'wurd)
Searches forward through the range of text which starts at address a1 and ends at address a2 looking for a string in the text which matches the leap string whose address is in the **pattern** system integer, and the length is in **patlen**. If a matching text string is found, the address a3 of the string is returned on the stack. Otherwise, zero is returned. **<search<>** uses the Boyer-Moore table pointed to by the **ptable** system integer to locate potential matches and then performs a string comparison to explicitly validate the match.

searching (c ->)
(pronounced serch'ing)
Searches for the current leap pattern in the text. If the key is the Undo key and **pattern** holds a -2 (Spell Check Leap occurring) or **newlex** is true, the character will be dropped, only **spellcheckleap** will be executed. If the key is the Erase key and **newlex** is true, the character is dropped and **searching** is exited since it cannot search for an erase character. If **newlex** is true (new leap occurring) or **pattern** holds a -1 (previous operation was a [Shift]-[Leap] scroll), **newlex** is set to false and the old pattern is omitted (starting fresh).

Now **searching** is almost ready to start the search. **unexpand** collapses the selection and **start-search** prepares for the search. If the character was an erase character, the last character is removed from the search pattern and the search start point is set to just after the current cursor position. If the character was not an erase character it is appended to the search pattern and the search start point is set to just after the current cursor position.

Now that the pattern has been checked and adjusted, **buildtable** builds the pattern-specific Boyer-Moore search table. Now the search occurs. If a forward search is used, the second half of the text is searched first, then, if necessary, the first half of the text is searched.

start-search (->)
(pronounced start' serch)
Prepares for the start of a search operation. If the cursor is extended or split, the contents of **savebos** are moved to **bos** and **bos nextchar** is moved into **eos**. Even if the selection is

extended, the **eos** and **bos** must temporarily point to the last character before the gap for the search algorithms to work correctly. Otherwise, **start-search** performs no actions.

tapmove (a ->)
(pronounced tap'moov)
Tries to move the cursor to text position a. If the text position is not within the current leap range, **tapmove** is exited. Otherwise, the **eos** is set to a, **bos** is set to **eos prevchar**, the **gap** is adjusted, the display is fixed, the cursor is set to narrow at the new position, and **forceop** is set to true.

uncreep (->)
(pronounced un'creep)
Executes **<uncreep>**. Used to undo a series of creep operations.

<uncreep> (->)
(pronounced brak'it un-creep)
Toggles the effect of a series of **lex-tap** operations. Uses **swappos2** to swap the saved contents of the cursor state integers with the current cursor state integers. The next time the cursor is drawn, the previous cursor state (as represented by the saved cursor state integer values) will be reflected. Adjusts the **gapline** if necessary and sets itself as the undo operation.

unmove (-> n1 n2 n3 n4 n5 n6)
(pronounced un'moov)
Uses **pushpos** to push the contents of the current cursor state integers onto the stack, and then **swappos** to swap the contents of the current cursor state integers (on the stack) with the previous cursor state values. Tne undo buffer is cleared an **unmove** sets itself as the undo operation.

unscroll (->)
(pronounced un'skroll)
Executes **<uncreep>**. Used to undo a series of scroll operations.

10.3 LEAP INTEGERS

direction (pronounced dy-rek'shun)
Equal to -1 for forward leap; equal to 0 for backward leap

leftlex? (pronounced left'leks)
Remembers the value of direction during a series of scroll-again or creep-again operations

lexxing (pronounced lek'sing)
Holds a true flag if leaping is occurring

maptable (pronounced map' tay'bl)
Holds a pointer to a table which maps invalid text characters to their valid text equivalents

matched (pronounced match't)
Holds either the address of a leap string match in the text or 0 if a leap string was not matched

newlex (pronounced noo'leks)
Holds a true flag if this a new leap operation; set by **init-lex**

oldshiftlock (pronounced ohld' shift-lahk)
Holds the saved state of the Shift Lock key during a leap operation

patlen (pronounced pat' len)
Holds the length of the current leap string

pattern (pronounced pat'turn)
Holds a pointer to the current leap string characters

patternsize (pronounced pat'turn syze)
Holds the maximum length of a leap string pattern (256)

ptable (pronounced pee' tay'bl)
Holds a pointer to the 256-byte Boyer-Moore search table

savebos (pronounced sayv' boss)
Holds a saved copy of the **bos** pointer during a leap operation

11. DRAG

Introduction

The drag routines move sections of text to different locations within the text. To move a section of text, the user (a) selects the section of text to be moved, and (b) uses the leap mechanism to move the cursor to the desired destination location for the text. When the user releases the Use Front and Leap keys, the text will be moved from the old to new location. Be sure to read the Chapter on leaping before reading about drag.

drag (->)
(pronounced drag')

drag is the word called by the Leap routine **finish-lex** when leaping is terminated and an extended selection exists. **drag** will first check to see if the drag destination location, found in the **savebos** system integer, lies within a locked region of text. If so, the original, highlighted selection is redisplayed if necessary and the operation is aborted. Otherwise, **start-drag** is executed.

start-drag prepares the selection and destination location for a drag operation. The flag returned by **save-drag**, which indicates whether format packets must be adjusted, is stored in a local variable. If the destination location is currently represented in the window table, the screen line number in which the destination is located is saved in another local variable. The purpose of this is to maintain the screen position of the destination point, if possible. If the destination is not represented in the window table, this local variable will be set to 0.

Next, **drag** checks to see where the destination location is relative to the selection location (the **gap**). If the destination is before the gap, **drag-forward** implements the drag. If the destination lies after the gap, **drag-backward** is used. Both **drag-forward** and **drag-backward** are passed the size of the piece of text they will have to move in order to implement the drag operation.

After the text has been moved and the text pointers readjusted, **drag** uses **preset** to fix the gap skip markers and then passes the local packet and text-in-window flag to **end-drag** (see the individual descriptions of **drag-backward** and **drag-forward** for more information).

Finally, **undrag** is set as the undo operation and the **dirtytext?** integer is turned on.

drag-backward (n ->)
(pronounced drag bak'wurd)

Used to drag a selection to a destination in the second text partition. The value passed to **drag-backward** is the size of the text between the **beot** and the drag destination location, which is the amount of text which will have to be moved to insert the selection at the destination. All intervals between the **bos** and destination are marked for updating and all intervals after the destination are marked as partially known.

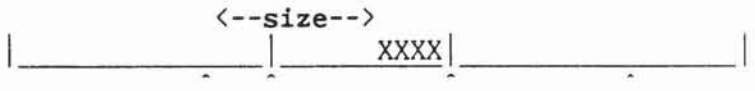
Next, the selection text is moved into place. If the system has enough room, the selection is moved forward **size** bytes; then the text between the **beot** and destination is moved into place, right before the new selection location. If the system does not have enough room to perform this straightforward text movement, a series of **reverse** operations will be used to reposition the text.

drag will reposition the text pointers after **drag-backward** completes. The old **bos** location will be saved in the **pop** integer. **bos size +** will be saved in the **op** integer. The gap,

beot, and **extbos** integer contents will all be decremented by "size" bytes.

Before:

After:

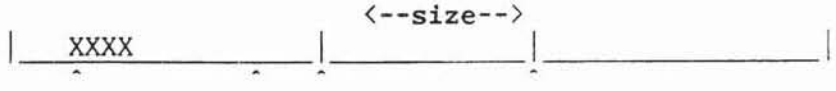


drag-forward (n ->)
(pronounced drag for'word)

Drags a selection to a destination in the first text partition. The value passed to **drag-forward** is the size of the text between the destination and the **bos** location, which is the amount of text which will have to be moved to insert the selection at the destination. All intervals between the destination and **beot** are marked for updating and all intervals after the **beot** are marked as partially known. Next, the selection text is moved into place. If the system has enough room, the "size" bytes between the destination location and the selection start are moved right before the **beot**. Then the selection text is moved back to the destination location. If the system does not have enough room to perform this straightforward text movement, a series of **reverse** operations will be used to reposition the text. **drag** will reposition the text pointers after **drag-forward** completes. The old **beot** address will be saved in the **pop** integer. The old **savebos** (destination) address will be saved in the **op** integer. The **gap**, **beot**, and **extbos** integer contents will all be incremented by **size** bytes.

Before:

After:



end-drag (f n ->)
(pronounced end' drag)

Fixes up format packets if necessary and redisplays the text in the window if necessary. The selection which was dragged is left highlighted.

start-drag (-> f)
(pronounced start' drag)

Prepares the selection and the destination location for a drag operation. If necessary, **start-drag** adjusts the destination location, trims the selection to be dragged, and adjusts selection format packets.

A selection cannot be dragged to the backward side of the first document separator (**savebos bor** = cannot be true). If this case exists, the **savebos** location is incremented by one so that it will be located just past the first document separator character.

Nor can the first document separator be dragged with a selection. If the first document separator character is included in a selection, **bos bor** =, the **bos** will be incremented by one so that the first document separator is not included in the selection.

Similarly, the last document separator cannot be dragged with a selection. If the last document separator character is included in a selection to be dragged, **eos eor** =, the **eos** will be decremented by one so that the last document separator character is not included in the selection.

Next, **start-drag** deals with format packets. If there are no format packets or breaks in either the selection to be dragged or the piece of text which will be moved to execute the drag, format packets do not need to be adjusted and **start-drag** will complete execution.

If all of the above cases are not true, **start-drag** will check the break immediately before the selection and the last break in the selection. If either of these breaks does not have an associated format packet, **start-drag** will make one and implace it in the text.

start-drag will also save the formatting state at the destination location in the **workpkt** if the formatting cases were not all true. The flag returned by **start-drag** is true if the format packets had to be checked.

undrag (->)
(pronounced un'drag)

Uses **drag** to redo the drag operation in the reverse order. Restores the screen to the way it looked before the drag began, that is, the way it looked before the user pressed the Leap key. The **pop** contents are moved into **savebos** before the new drag operation is performed. The window is redisplayed as necessary.

12. COPYUP

Introduction

The copy-up routines allow the user to transfer text from one disk to another. Copyup is not an explicit command executed by the user. To transfer text, the user selects the section of text to be transferred, places the destination disk in the disk drive, and uses [Use Front]-[Disk] to load the contents of the destination disk into memory.

During the loading process, the disk code will notice that the previous text contained selected text. The selected text will be put in safe place during loading of the new text and will be inserted into the new text at the current cursor location once the new text has been loaded.

12.0 COPYUP STEP-BY-STEP

The user has saved the current text and left a portion or all of it highlighted. They put a new disk in the drive and use the Disk command, indicating that they want to copy up the highlighted text into the new universe that is loading.

First, the size of the selection and the ID block of the destination are checked to make sure the selection will fit into the new universe. If there will not be enough room, an error is indicated.

If there is enough room to accommodate the new text, the selection needs to be split out of the current text. This is done by attaching a format packet to the beginning and the selection moved to a safe place. Since the new text will be loaded in sequentially and then unpacked to give us our complete text, the end of the text area is used as a safe place.

Communicating text from one universe to another is a bit tricky (a universe is all the text associated with one disk). It uses a special area of memory which is not part of the saved image and not overwritten during the loading of a new universe. An indicator is kept in this area that tells the Cat whether a copyup was performed, and, if so, where the old text is loaded.

After the new universe is loaded in and running, the Cat checks the special indicator. If it says there is extra text, a special unpack routine opens the text while swapping the saved text at the end of memory into the new gap. When that is done, the new selection is allowed to merge into the new universe.

12.1 COPYUP ROUTINES

copyup (->)
(pronounced kah'pee up)

If the user saves the text on the screen, leaving all or a portion of it highlighted, then puts a new disk (with its own universe) in the drive, and uses the Disk command, **copyup** will be executed. **copyup** transfers a copy of the currently highlighted text into the text about to be loaded into memory. If the entire on-screen text is highlighted, it can be dirty (not saved) and **copyup** will still take place.

The text copied up will be inserted into the new text, starting at the current cursor location in the new text. **copyup** will save a copy of the highlighted text in a safe space (at **ramend**) before the process of moving the new text into memory begins. **copyup** performs seven actions:

1. If the disk in the drive (which contains the new text to be loaded in) is locked, a **copyuplock** error will be issued.
2. The size of the highlighted selection (**gap bos -> size to**) is determined.
3. If the selection contains a break, the selection size is incremented (**pktsize 2* size +to**) to account for packets which will be inserted into the selection to preserve the format of the selection during the move.
4. If there will not be enough room to hold the selection during the loading of the new text, a **nocopyuproom** error will be issued.
5. If the selection contains a break, format packets are inserted into the selection as necessary.
6. The selection is moved up to a safe position near the end of RAM memory.
7. The location of the safe selection is stored in the **copyuptr** system integer.

move&adjusttext (a1 a2 ->)
(pronounced moov' and a-just' tekst)

Moves and compresses/expands the current text area to a new region of memory which starts at address a1 and ends at address a2. After the text has been moved, **text** will be located at address a1 and **endtext** will be located at address a2. All of the affected text pointers are adjusted accordingly and the interval table is updated. Used by the disk code as it prepares to move text to and from disk.

unpackcopiedup (a1 a2 a3 ->)
(pronounced un-pak' kah'peed up)

Used to merge text saved by **copyup** into the text of the new disk. **unpackcopiedup** first checks to make sure there is enough room in the new text for the copied-up selection. If so, the copied-up text will be moved temporarily to the end of the text. Then the copied-up text is moved into the undo buffer and the text is moved around to make room for the copied up text.

If any of format packets need to be adjusted/inserted in the copied up text, the adjusting/inserting is performed while the copied up text is in the undo buffer.

After the copied-up text is ready, **insertblock** inserts the copied-up text into the new text at the current cursor location. Any Calc packets in the copied-up text are adjusted, the text is redisplayed, the undo buffer is cleared, and **removeselection** is set as the undo operation.

13. THE KEYBOARD INTERFACE AND THE LEARN COMMAND

Introduction

There are two possible sources of key events in the Cat system. Real key events generated directly from the keyboard are spotted by the interrupt service routine, which is responsible for scanning the keyboard and reporting keyboard state changes as key events which are added to the key event queue. Recorded key events are generated when a recorded Learn sequence is played back. Only the lowest-level Forth keyboard I/O words know the difference between real and recorded key events. This section discusses the terminology and data structures associated with the Cat keyboard interface, the Forth words involved with key press handling, and the close tie between the keyboard interface and the Learn command.

13.0 KEYBOARD INTERFACE TERMINOLOGY AND DATA STRUCTURES

13.0.0 Scanning the Keyboard

At the lowest level of the keyboard interface is the keyboard interrupt service routine. Every time the timer interrupt goes off, the keyboard is scanned to see if it has changed since the last timer interrupt. Part of the code fragment used to service a Level 1 interrupt is responsible for polling the keyboard.

During the keyboard polling process, eight bytes of information are received. These eight bytes of data are placed in an 8-byte buffer whose start address is stored in the **tempkey** integer. This is the current keyboard scan information. The information received during the previous polling process is stored in another 8-byte buffer whose start address is stored in the **newkey** integer.

Each time the keyboard service routine is called, it polls the keyboard and compares the information received to the information received during the last execution of the service routine. If the information in the **tempkey** and **newkey** buffers is the same, the service routine will take no action. Only when the **tempkey** and **newkey** buffers hold different information, which indicates that a change in the state of the keyboard has occurred, does the keyboard service routine report a keypress to the rest of the system.

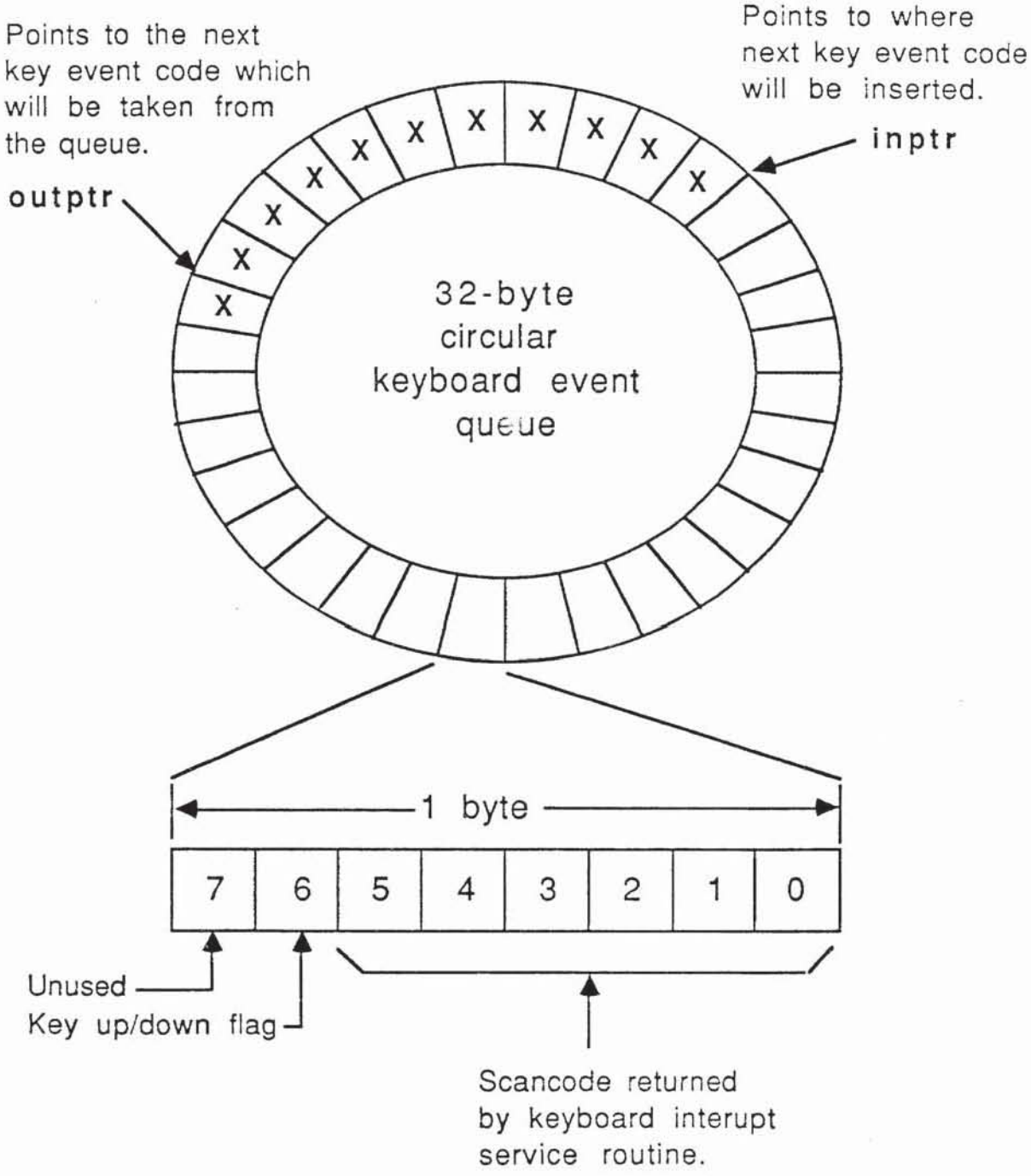
When a change in keyboard state occurs, the eight bytes of information in **tempkey** are transferred to the **newkey** buffer and one byte of scan information, condensed down from the eight bytes of information actually received, is placed into the keyboard event queue.

13.0.1 The Keyboard Event Queue

The keyboard event queue is a $20-hex-byte circular queue (see diagram 13-1). The start address of the queue memory is located at an offset in the **system.status** vector. The two pointers used to maintain the circular queue are kept in the **inptr** and **outptr** system integers. When the keyboard service routine adds an event byte to the queue, the byte is placed in the address pointed to by **inptr** and then the **inptr** address is incremented by one. When an event byte is removed from the queue, the byte is taken from the address pointed to by **outptr** and then the **outptr** address is incremented by one.

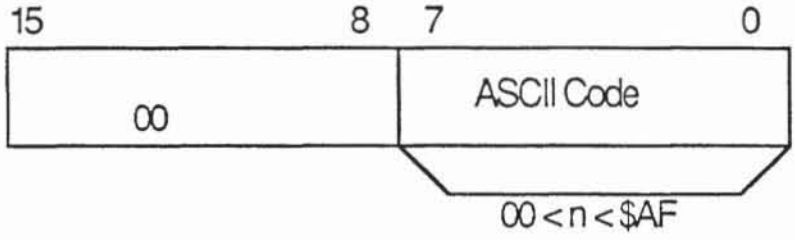
The word **inptr** always points to the next available location in the queue and **outptr** always points to the next available event byte in the queue. If **inptr** and **outptr** hold the same address, the queue is empty. The information is the number of the scanned key in the range of 0 to 63 and a bit (128) saying if the key was going up or down. Each entry in the keyboard event queue is one byte long.

13.1 The Keyboard Event Queue

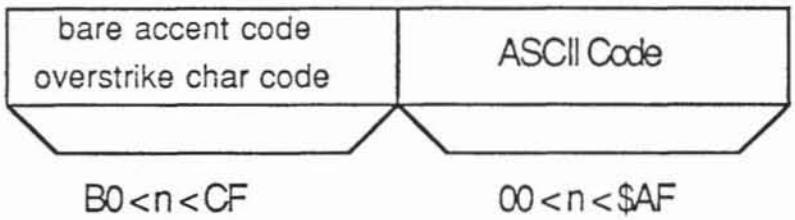


13.2 Keyboard Translation Table Entries

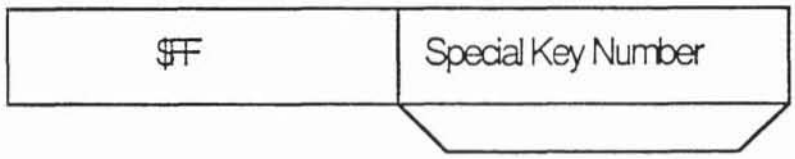
Normal ASCII Character:



Character with a Special Accent:



"Special"



Special Key Numbers:
0 KB1/2
1 Left Shift Key
2 Right Shift Key
3 Caps Lock Key
4 Left Use-Front Key
5 Right Use-Front Key
6 Left Leap Key
7 Right Leap Key

13.0.2 Special Keys

A special key is one that does not generate a character code. A special key will either affect the way subsequent key presses will be interpreted or will cause an editing command to be executed. The special keys on the Cat keyboard are listed below:

KB-I/II (this is really an imaginary key)
Left Shift
Right Shift
Shift Lock
Left Use Front
Right Use Front
Left Leap (Leap Backward)
Right Leap (Leap Forward)

Up/down state information about the special keys is kept in two 8-bit bit arrays. One bit array is held in the **shiftstate** system integer and the other is kept in the **modifiers** system integer.

Note: In reality, **shiftstate** should be called specialstate because its contents actually represent the current states of all of the special keys, not just the state of the Shift keys.

Each bit in the bit array represents one of the eight special keys. These are the bit assignments:

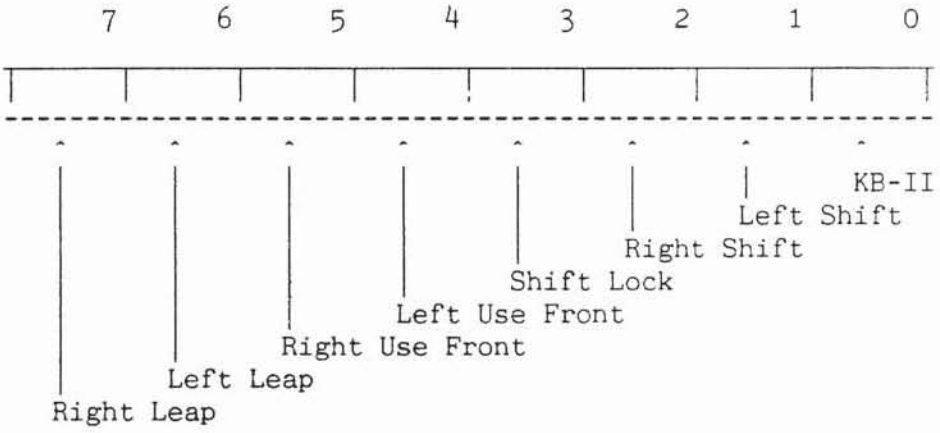


Figure 13.2: Bit Assignments in a Special Key Bit Array

The **shiftstate** bit array always represents the actual physical state of the special keys, with one exception. The Shift Lock bit is not cleared until a key-up event for either the left or right Shift key is received. Normally, a special key bit is cleared as soon as a key-up event for that special key is received. Also, KB-I/II is not set or cleared by the low-level representation, but is lit by the KB-I/II command.

The **modifiers** bit array holds the state of the special keys as viewed by the editor. The **modifiers** array will be the same as the **shiftstate** array except during the playback of Learn sequences. During Learn playbacks, the **modifiers** array will be artificially altered to simulate the pressing and releasing of special keys. At the end of a Learn sequence, the contents of the **shiftstate** array, which always holds the current physical states of the special keys, is copied into the **modifiers** integer.

13.0.3 Keyboard Translation Table

The Cat editor supports the keyboard layouts of 17 different countries. All of the keyboards have essentially the same number of keys arranged in similar layouts. It is only the assignment of character to key which varies from country to country.

The keyboard service routine returns position-specific information about a keypress, that is, which key was pressed. A keyboard translation table translates key position information to character information for any keyboard layout. There is a keyboard translation table for each of the 17 keyboard configurations supported. Every translation table consists of four subtables. The first subtable contains the data for characters found on the unshifted version of KB-I, the second contains the character data for the shifted version of KB-I, the third contains data for the unshifted version of KB-II, and the fourth contains data for the shifted version of KB-II.

There are 59 keys on the USA Cat's keyboard (other layouts may have 61). Each subtable contains 64 16-bit character data entries, one for each key on the keyboard with a few entries left blank. The following diagram shows how each translation table, and the subtables within, are arranged:

Offset	Translation Table Subtables
$00 ->	KB-I, non-shifted character data
$40 ->	KB-I, shifted character data
$80 ->	KB-II, non-shifted character data
$C0 ->	KB-II, shifted character data

Figure 13.3: Layout of a Keyboard Translation Table

The scan code returned in the lower six bits of the event code is actually the offset into a translation table subtable to the desired character data. The translation table used by a particular Cat is determined by a software switch. The translation table subtable used is determined by information found in the current **shiftstate** bit array.

The data entries in the keyboard translation table are two bytes each. If the upper byte contains a 00, indicating that the lower byte is the character to be returned. If the upper byte contains a $FF, it indicates that this key is a modifier type key and the type of modifier is in the lower byte. (This is the same as the bit position in the **shiftstate** array.) If the upper byte is something else, it is assumed the two bytes are two separate characters to be returned. This is most commonly used for accented characters. The accent byte ($C0-$CF) is in the upper byte, and the real character is in the lower byte.

13.1 PROCESSING KEYPRESS INFORMATION

do-event is the Forth word which removes the next available event code from the event queue and translates the key scan code information in the event code to keyboard-specific character information. If the event code does not reflect a change in state of one of the special keys, **do-event** uses the following algorithm to find the character information which corresponds to the event queue information:

1. First, **do-event** checks to see if any of the shift key bits in the **shiftstate** array are set.

The state of the special keys determines how the character information is treated. If a Shift key bit is set, $40 is added to the scan code offset. This bumps the offset into the shifted portions of the character data (see the translation table diagram).

2. Next, the state of the KB-I/II bit is checked.

If the KB-I/II bit is set, if Keyboard II is selected, $80 is added to the scan code offset. This bumps the offset into the keyboard II part of the translation table.

Now that the subtable to be used has been determined, **do-event** can index into the subtable, using the offset in the **scancode** information, and fetch the two bytes of translation table character data.

If the key event code does correspond to a special key change in state, **do-event** will alter the **shiftstate** information. A special key-down event will usually cause the special key's corresponding bit in the **shiftstate** array to be set. The exceptions are (1) a Shift Lock key-down event is not accepted when a Use Front key is down, and (2) a left or right Shift key-down event will cause the Shift Lock bit to be cleared in addition to the left or right Shift key bit being set. A special key-up event will clear the special key's corresponding bit in the **shiftstate** array. The exception is that Shift Lock key-up events are ignored.

13.1.0 Returning Character Information

When **do-event** has completed execution, 32 bits of character information will be stored in the **kval** (key-value) system integer and a true ("character available !") flag will be stored in the **kstat** (key-status) system integer. The character information returned in **kval** has the following format:

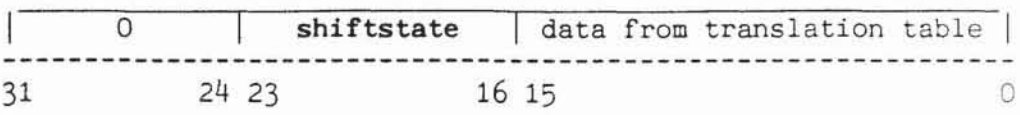


Figure 13.5: Format of the Character Information Returned in kval

If a key-up event for a nonspecial key is received, and the new translation table character data is the same as the current character data found in **kval**, (if the nonspecial key currently held down is being released) autorepeating will be disabled.

Before **do-event** finishes it will check the Shift Lock bit and turn the Shift Lock light on or off.

13.2 TYPES OF KEY INFORMATION

Key information can come from two sources, the keyboard event queue, or a prerecorded Learn string in playback. Key information from the event queue is called real key information, because it was generated directly from the keyboard.

13.2.0 Real Key Information

The Forth words **<<?k>>** and **@k** check for and obtain real keypress information. **<<?k>>** spins in a loop calling **do-event** until a key event is returned (until the **kstat** integer contains a true flag) or until the event queue is empty (**?ev** checks for an empty event queue). **<<?k>>** returns a true flag if new key information is available (in **kval**).

@k is used after it is determined that real key information is available. **@k** turns autorepeating on if the character is not a special key, sets up the next autorepeat time, stores a false flag in **kstat** (indicating that the current character is no longer available), and returns just the character value on the stack, which is the lower byte of the translation table data found in **kval**.

13.2.1 Recorded Key Information

The words **playback?** and **playback** are the Learn equivalents to **<<?k>>** and **@k**.

playback? returns a true flag if the character information for a prerecorded key event is available. The contents of the **learnbuff** and **learning?** system integers determine whether a Learn sequence is being played back. **learnbuff** will return a true flag if any type of Learn operation, recording or playing back, is occurring. **learning?** will hold a true flag if a Learn sequence is currently being recorded and a false flag if a Learn sequence is being played back. If these integers indicate that a Learn playback is occurring, and if there are still characters in the string which must be played back, **playback?** will return a true flag.

If a prerecorded character is available, **playback** obtains the prerecorded character information. A character entry in a Learn string contains essentially the same data found in the **kval** integer. The difference is that **scancode** information is placed in the upper byte:

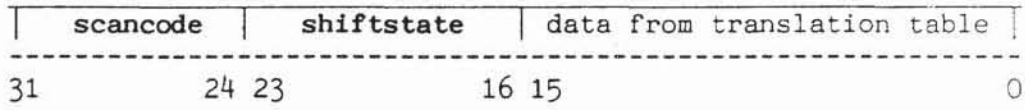


Figure 13.6: Format of a Character Entry in a Learn String

playback unpacks the key event information recorded in the Learn string. The actual character data, the data from the translation table, is returned on the stack. To simulate the environment in which the key was originally typed, the **scancode** and **shiftstate** information is placed in the corresponding key-state system integers.

13.3 OBTAINING KEY INFORMATION

<?k> is the main Forth word used to check for available keypress information of any type, real or recorded. If a real key is not available, **<?k>** will check for a recorded key. If either type of key is available, **<?k>** will set up a duplicate version of the key-state environment and will return a true flag. If no key is available, a false flag is returned. If **<?k>** returns true, **<key>** can be used to get the actual key that would be returned.

13.3.0 The Key-State Environment

Complete information about a key press is stored in two sets of system integers. Both sets hold equivalent information:

kval	< - - - - >	**char**
kstat	< - - - - >	**char?**
shiftstate	< - - - - >	**modifiers**

The integers on the left have already been introduced. **kval**, **kstat**, and **shiftstate** always contain current key information about the most recent keyboard keypress. The integers on the right contain key information which corresponds to the state of the keyboard as viewed by the editor. Usually **char**, **char?**, and **modifiers** contain the same information as their left column counterparts. During the playback of Learn sequences however, the integers on the right will be deliberately modified by the Learn routines.

When playback of a Learn sequence terminates, the contents of the integers on the left are copied into the integers on the right to restore order to the system.

13.3.1 Setting Up the Editor Key State

The word **!char** (store-char) is used by **<?k>** to set up the editor version of the key-state information. **!char** is very similar in function to **do-event**, except

1. **do-event** is an assembler routine, while **!char** is written in Forth.

2. **!char** is passed translation table data on the stack, while **do-event** obtains the translation table data itself.

3. If **!char** receives special key event information, it will alter the special key bit array kept in the **modifiers** system integer instead of the **shiftstate** bit array.

4. **!char** returns character information in the **char** and **char?** system integers instead of in the **kval** and **kstat** integers.

13.3.2 Getting the Character

If **<?k>** indicates that a character is available, **<key>** gets the character and, if a Learn sequence is being recorded, appends the key information to the Learn string. **<key>** spins in a loop until **<?k>** indicates that a key of any type is available. When a key is available, **<key>** takes the character value from **char**, leaves it on the stack, and sets the **char?** flag to false to indicate that this key is no longer available.

13.4 THE LEARN COMMAND

The Learn command records sequences of keypresses which may be played back at a later time. Since the playback and recording of Learn strings is connected to the keyboard interface at a very low level, the rest of the system never needs to be concerned with the actual source of key information received. As far as the rest of the system is concerned, there is always a typist at the keyboard.

13.4.0 Learn Strings

The Cat system can store up to ten Learn sequences at once. The Learn sequences -- actually just strings composed of 4-byte packets of keypress information -- are stored in ten string variables named **learn0**, **learn1**, etc. A 20-byte table in memory holds the ten 2-byte tokens for each of the Learn string variables. Execution of the word **learnstrings** will place the address of this table on the stack.

13.4.1 Important Learn Integers

learnbuff returns a true flag if any type of Learn operation -- recording or playing back -- is occurring. **learning?** holds a true flag if a Learn sequence is currently being recorded, and a false flag if a Learn sequence is being played back. **curlearn** holds the number of the Learn string currently being played back or recorded. **learnpos** holds the offset into the current Learn string to either the next keypress to be played back (during playback) or to the location where the next keypress information received will be stored (during recording). **maxlearn** holds the maximum allowable length of a Learn string (4096 bytes).

This value can be changed to allow for longer Learn strings if needed. The actual maximum is the minimum of the current **maxlearn** and the total free space left in the system. That is, if there isn't enough room to set up a record buffer of **maxlearn** bytes, the buffer will be set to whatever is available.

13.4.2 Recording a Learn Sequence

[Use Front]-[Learn] is pressed to initiate the recording of a Learn sequence. When [Use Front]-[Learn] is pressed, the word **Learn** is executed. **Learn** will perform one of two actions.

If **Learn** is executed when no Learn activity is currently occurring (**learnbuff** holds a zero), **Learn** will first use **indicate** to display the "**Learn ?**" indicator light. Next, **Learn** must wait in a loop until it receives the digit that indicates to which Learn string the upcoming Learn sequence should be assigned.

If the next keypress received does not correspond to a digit (0-9), **Learn** will turn the indicator light off and terminate execution. If a digit is received, **Learn** uses **showlearn** to replace the "?" in the indicator light with the number received, and **newlearn** sets the system up for Learn string recording. **newlearn** places the chosen Learn string number in **curlearn**, tries to expand the chosen Learn string to the maximum allowable Learn string length, sets **learnpos** (the offset into the Learn string) to zero, turns **learning?** on (to indicate that recording is occurring), and turns **learnbuff** on (to indicate that a Learn activity is occurring).

Now the system is ready to record any subsequent keypress information received. Whenever **<key>** obtains keypress information, the last word it executes is **record**. **record** checks the **learning?** and **learnbuff** integers to see if recording is on. If recording is on, and if there is enough room in the Learn string for one more keypress entry, **record** will store the current **scancode** contents in the upper byte of the current keypress information and will place all four bytes in the next position in the current Learn string.

13.4.3 Terminating a Learn Recording

A Learn recording terminates when the user presses [Use Front]-[Learn] again. If a Learn activity is occurring when **Learn** is executed, **Learn** will terminate recording with the use of **clearlearn**. **clearlearn** turns the indicator light off and uses **clr-kbd** to clean up. If recording is on, **clr-kbd** will reduce the current Learn string size to its actual size, will transfer the contents of **shiftstate** to **modifiers** (to synchronize the two special key-bit arrays), will turn **learning?** and **learnbuff** off, will make sure that any leaping activities properly terminate (with the use of **finish-lex**), and will use **rule** to redraw the ruler-bar/status area. If the selection is not extended at this point, **Learn** will terminate execution and recording will be stopped. Learning will also be terminated if another Learn is played back.

13.4.4 Phrase Storage

The Learn command also supports phrase storage. If the selection is extended and the Learn string is empty when a recording is terminated, the selected phrase will be stored in the current Learn string, with a 4-byte zero header, before **Learn** terminates execution. When the Learn string is played back, it will cause the stored phrase to be placed in the text starting at the current cursor location.

To store a phrase, the user must

1. Highlight it.
2. Start the recording as described above.
3. Continue holding the Use Front key and, immediately after specifying the Learn string number, press [Learn] again. This second press of [Learn] will cause **Learn** to be executed. (The Use Front key cannot be released in between uses of [Learn] or the phrase will not be recorded.)

Learn will note that recording is on and will stop recording in the manner described above. Next, if the current Learn string is empty, and if the selection is extended, **Learn** will proceed with the phrase storage process.

First, the phrase will be temporarily stored in the gap area while **Learn** checks to see if a format packet needs to be inserted into the phrase. **Learn** will check to see if there is enough room in the gap for the selection and a paragraph format packet. If there is enough room, a 4-byte zero flag followed by the selection text will be placed in the gap. Next, **Learn** checks to see if a format packet should be inserted into the phrase. If a format packet is requred, **Learn** inserts one in the correct position. Finally, the phrase string is moved into the current Learn string and execution of **Learn** terminates.

13.4.5 Playing Back a Learn Sequence

To initiate playback of any of the recorded Learn sequences, the user holds down the Use Front key, and, while holding it down, presses the digit key associated with the Learn string.

This passes the number of the desired Learn string to **lrncmd** and executes it. If a Learn sequence is currently being recorded, the current Learn string is closed down (reduced to its proper length). After **lrncmd** has terminated any recordings in progress, it uses **showlearn** to display the string number of the Learn sequence selected for playback and uses **newplayback** to initiate playback of the selected string.

newplayback sets the current Learn string and checks the contents of the first four bytes in the current Learn string. If the first four bytes hold a zero, the Learn string holds a phrase. The handling of stored phrases is discussed below. If the Learn string does not hold a phrase, **newplayback** puts the system in the playback state by clearing all bits in the **modifiers** bit array, setting the offset into the Learn string, **learnpos**, to zero, turning **learning?** off to indicate that playback is occurring, and turning **learnbuff** on to indicate that a Learn activity is underway. Now, when **<?k>** is asked for key information, it will return playback characters until all characters in the Learn string have been played back.

13.4.6 Inserting Stored Phrases

If a Learn string with a stored phrase is selected for playback, **newplayback** will insert the phrase into the text at the gap and will adjust the format packets as necessary.

13.5 FORTH KEYBOARD ROUTINES SUMMARY

13.5.0 Preparing Keypress Information

!char
(pronounced stor' kair)

do-event (-)
(pronounced doo' ee-vent')
do-event is the only Forth keyboard I/O word to interact directly with the keyboard event queue. It does four things:

1. Gets the next keyboard event from the event queue and adjusts the queue accordingly
2. Gets the translated value of the event code from the keyboard codes table
3. Uses the translated value to set up the data to be returned in **kval**
4. Puts a true flag in the system integer **kstat** to indicate that key information is available

?kval (pronounced kwes'chun kay vall)

13.5.1 Obtaining Keypress Information

?ev (- f)
(pronounced kwes'chun e-vent')
Returns a true flag if there is key event information in the keyboard event queue. If the **inptr** and **output** contain different addresses, the queue is not empty.

<<?k>> (- f)
(pronounced brak'it brak'it kwes'chun kay)
Returns a true flag if a real key event, from the event queue, is available.

<?k> (- f)
(pronounced brak'it kwes'chun kay)
Returns a true flag if a character is available, either a real character from the event queue or a simulated character from a Learn sequence.

?k (- f)
(question-key)
(pronounced kwes'chun kay)
If a nonspecial key is down returns a true flag and disposes of the key. This and **key** are used by the underlying Forth language and are never used by the editor.

@k (- char)
(pronounced fetsch' kay)
Fetches the key data for the next available character from **kval** and, if the character is not a special character, enables autorepeating. Sets up the next autorepeat time in **ktime** and turns **kstat** off. Only called when a key is available.

<key> (- c)
(bracket-key)
(pronounced brak'it ki)
Waits in a loop until a character is available, gets the character data from **char**, and puts a false flag in **char?** to indicate that the character has been taken. If the system is currently recording, records the character.

key (- c)
(pronounced ki)
Waits in a loop until a nonspecial key is available. Returns the ASCII code (0<=code<=$7F) for the key.

?t (- f)
(pronounced kwes'chun tee)
Polls the keyboard for an ASCII keypress value (0<=value<=$7F). Returns a true flag if an ASCII key was available. The key is discarded.

13.5.2 Autorepeat Routines

?auto (- f)
(pronounced kwes'chun aw'toe)
Returns true flag if it is time to autorepeat a character. If the number of ticks has exceeded the next scheduled autorepeat time, stored in **kticks**, it is time to repeat.

clear-auto (-)
(pronounced cleer' dash aw'toe)
Disables autorepeating for the last key returned by storing a false (0) flag in **auto**.

set-auto (-)
(pronounced set' dash aw'toe)
Turns on autorepeating. Stores a true flag in **auto** and calculates and stores the next scheduled time for an autorepeat in **kticks**.

13.5.3 Words That Check and Affect the shiftkey and modifiers States

?ctl (- f)
(pronounced kwes'chun see' tee ell)
Returns a true flag if the editor thinks one of the Use Front keys is down.

down? (n - f)
(pronounced down' kwes'chun)
Returns a true flag if bit n in the **modifiers** bit array is set. Since each bit in the **modifiers** array corresponds to one of the special keys, this routine checks whether a certain special key is considered to be down.

```
: down? ( n - f )
          1 swap                ( Put bit number on top.  )
          shl                   ( Shift the "1" into the )
                                ( specified bit position. )
          modifiers
          and                   ( Isolate the specified )
                                ( bit position in the )
                                ( modifiers or )
                                ( shiftstate bit array. )
          0=
          0=  ;                 ( Return true flag if bit )
                                ( was set. )
```

Kb1/2 (-)
(pronounced kee'bord wun' slash too')
Checks the state of the KB-II bit in the **modifiers** bit array. If the bit is set, the KB-II bit is cleared in both the **modifiers** and **shiftstate** bit arrays, and vice versa.

?kb2 (- f)
(pronounced kwes'chun kee'bord too')
Returns a true flag if the editor thinks the KB-I/II is down, that is, that KB-II is currently in use.

?keystep (- f)
(pronounced kwes'chun kee'step)
If a space is currently available, waits in a loop for the next real key event. Returns a true flag if a carriage return becomes available.

?lex (- f)
(pronounced kwes'chun leks)
Returns a true flag if the editor thinks the left Leap key is down.

?rex (- f)
(pronounced kwes'chun reks)
Returns a true flag if the editor thinks the right Shift key is down.

?shift (- f)
(pronounced kwes'chun shift)
Returns a true flag if the editor thinks one of the Shift keys is down.

?shifted (- f)
(pronounced kwes'chun shift'ed)
Returns a true flag if the editor thinks one of the Shift keys, or the Shift Lock key is down.

?shiftlock (- f)
(pronounced kwes'chun shift lahk)
Returns a true flag if the editor thinks the Shift Lock key is down, that is, if Shift Lock is currently in effect.

sync-shiftkeys (-)
(pronounced sink' dash shift' kees)
Puts a copy of the special key-bit vector in **shiftstate** in **modifiers**.

toshiftlock (f -)
(pronounced too shift' lahk)
If the flag is true, the Shift Lock bit in the special key bit vector found in **modifiers** will be set and the Shift Lock light turned on. If the flag is false, the Shift Lock bit in the special key bit vector in **modifiers** will be cleared and the Shift Lock light will be turned off. If a Learn is not currently being played back, the newly modified **modifiers** value will be copied into **shiftstate**.

13.6 LEARN ROUTINES SUMMARY

0-cmd
1-cmd
2-cmd
3-cmd
4-cmd
5-cmd
6-cmd
7-cmd
8-cmd
9-cmd (-)
(pronounced wun' kom-mand, too' kom-mand, ...)
One of these words will be executed if -- while the Use Front key is held down -- the user presses [Learn] and then a digit key. All of the "**n-cmd**" words pass a buffer number to **lrncmd** and cause a Learn sequence to be recorded or played back.

clr-kbd (-)
(pronounced kleer' kee-bord)
Ends playback of a Learn recording or Learn playback session. Copies the special key-bit array settings found in **shiftstate** to **modifiers** and places false flags in the **learning?** and **learnbuff** system integers.

#key? (n - f)
(pronounced sharp kee' kwes'chun)
Compares the scancode of the character most recently received to the scancodes for the numbers 0 through 9. Returns a true flag if the scancode corresponds to a number. **Learn** uses **#key?** to get the number which is to be assigned to the Learn sequence about to be recorded.

Learn (-)
(pronounced lern)
Stops a Learn recording or playback and sets the appropriate flag.

learnsize (- n)
(pronounced lern'syze)
Returns the maximum number of bytes available for storage of a Learn sequence. Checks to see if **maxlearn** bytes are available.

learnstrings (-)
(pronounced lern'strings)
Returns the address of a 10-entry array of Learn string tokens.

lrncmd (n -)
(pronounced lern kom'mand)
Starts recording a Learn using buffer n, or plays back the Learn sequence located in the n buffer.

newlearn (n -)
(pronounced noo'lern)
Performs all the initialization required prior to the start of a Learn recording.

newplayback (n -)
(pronounced noo' play bak)
Performs the initialization required before a Learn string can be played back.

numberkeys (- a)
(pronounced num'ber kees)
Pushes the address of a 10-byte array of the **scancodes** corresponding to the digits 0 through 9. Used by the word #key?

?panic (- f)
(pronounced kwes'chun pan'ik)
Checks to see if the user has panicked and pressed a key in order to stop the playback of a Learn sequence. The first time the user hits a panic key, a true flag is stored in the **panicked** system integer. This flag will not be cleared until the panic condition is handled by some other part of the system.

If a panic key is not currently available, the current panic state flag, stored in **panicked**, will be returned. If a panic key is available, a true value will be OR'ed with the current panic state value. The use of the OR operation ensures that the panic state will never be accidently cleared. A special key going up is not considered a valid panic key event.

If a special key going up is encountered, the special key's corresponding bit in the **modifiers** bit array will be cleared.

playback (- c)
(pronounced play' bak)
Returns the next Learn sequence character to be played back.

playback? (- f)
(pronounced play' bak kwes'chun)
Returns a true flag if there is a Learn sequence character to play back.

record (c - c)
(pronounced ree kord')
Records the character c in the current Learn string.

setlearn (n -)
(pronounced set' lern)
The n is the string number to be used for the current Learn recording. Checks to see that the number is within the allowable range of Learn string numbers. If it is, the string number is stored in the **curlearn** system integer.

showlearn (n -)
(pronounced sho' lern)
Causes the Learn indicator light to be displayed with the string "LEARN (#n)" listed in the indicator.

thislearn (- a n)
(pronounced this' lern)
Returns the address a and length n of the current Learn string.

13.7 KEYBOARD INTEGERS SUMMARY

auto (pronounced aw'toe)
Holds a flag indicating whether or not autorepeating is on.

cbuff (pronounced see'buff)
Integer used to hold the start address of the $20-byte circular event queue.

char (pronounced kair)
Logical version of **kval**. Holds the next key value that should be passed to the editor.

?char (pronounced kwes'chun kair)
Logical version of **kstat**. Holds a flag indicating whether or not the editor should be told that a key is available.

inptr (pronounced in' point'er)
Keyboard buffer pointer. **inptr** holds address where next key code should be inserted into buffer.

kcodes (pronounced kay' kohds)
Holds a pointer to the keyboard translation table.

kstat (pronounced kay'stat)
Holds a flag set up by **do-event** which indicates whether key information is currently available.

kticks (pronounced kay'tiks)
Autorepeat timer.

kval (pronounced kay'vall)
Holds key value returned by **do-event**.

modifiers (pronounced mahd'i-fy-ers)
Holds a bit vector of the imagined special keys (as affected by Learn).

outptr (pronounced owt' point'er)
Keyboard buffer pointer. **outptr** holds address from which next requested key code should be taken.

scancode (pronounced skann' kohd)
Holds just the scan code portion of the event byte from the last keyboard event processed.

shiftstate (pronounced shift' stayt)
Holds a byte-long bit vector which represents the current physical states of the special keys.

ticks (pronounced tiks)
Time ticks.

time0 (pronounced tyme zeer'oh)
Holds number of ticks between autorepeats (30).

time1 (pronounced tyme' wun)
Holds number of ticks between autorepeats (10).

13.8 LEARN INTEGERS SUMMARY

curlearn (pronounced kur' lern)
Holds offset into the Learn strings table to the current Learn string.

learnbuff (pronounced lern' buff)
Holds a flag which tells the keyboard whether a Learn-related operation is underway.

learning? (pronounced lern'ing kwes'chun)
Holds a flag which is true when a Learn sequence is being recorded.

learnpos (pronounced lern'paws)
Holds offset into the current Learn string.

#learns (pronounced sharp'lerns)
Holds the maximum allowable number of Learn strings.

maxlearn (pronounced maks'lern)
Holds the maximum allowable length of a Learn string.

panicked (pronounced pan'ickt)
Holds a flag which indicates whether the user has panicked and pressed a key in order to terminate playback of a Learn sequence.

13.9 LEARN STRINGS CREATION

```
: learn0        <string>    [   0 ,
: learn1        <string>    [   0 ,
: learn2        <string>    [   0 ,
: learn3        <string>    [   0 ,
: learn4        <string>    [   0 ,
: learn5        <string>    [   0 ,
: learn6        <string>    [   0 ,
: learn7        <string>    [   0 ,
: learn8        <string>    [   0 ,
: learn9        <string>    [   0 ,

code learnstrings   ( - a )   nx )    jsr,    ;c

    t'  learn0  w,          ( an array of Learn string tokens )
    t'  learn1  w,
    t'  learn2  w,
    t'  learn3  w,
    t'  learn4  w,
    t'  learn5  w,
    t'  learn6  w,
    t'  learn7  w,
    t'  learn8  w,
    t'  learn9  w,
```

14. THE SORT COMMAND

Introduction

The Sort command allows the user to sort a highlighted selection of text into ascending or descending alphabetical or numerical order. Part 14.0 provides an introduction to some important terms and concepts. Then the five steps needed to carry out a Sort operation are discussed:

Step	Part
Finding the key field to be used in sorting	14.2
Adjusting the highlighted text in size and content so that it has only complete records	14.3
Constructing a description of the highlighted text (making a linked list of sort entries)	14.4
Reordering the sort entries	14.5
Rearranging the text to match the reordered sort entries	14.6

14.0 INTRODUCTION TO RECORDS, FIELDS AND KEY FIELDS

Records

The items the Cat rearranges when it does a sorting operation are called records. Example records might be the lines in a single-column list, or the rows in a table with multiple columns. The Cat can also sort paragraphs, or names and addresses in an address list.

A record is like an index card. Shuffling index cards puts them in a new order, but it doesn't change what's written on each index card. Similarly, records change order when they are sorted, but the sequence of characters and the arrangement of text inside of each record does not change.

A record begins and ends with a record separator. Record separators are defined with the Setup command as one, two, or three break characters. Break characters include carriage returns, page breaks, and document separators. Any combination of consecutive break characters in the appropriate number constitutes a record separator.

Fields

Each record contains zero or more fields. For example, the zip code and last name parts of an address would be separate fields. Each column in a table is a separate field.

A field begins and ends with one or more field separators. Any tab or break character constitutes a field separator. The first character in a record separator is also a field separator.

Key Fields

Sort focuses on a key field when it rearranges records that contain more than one field.

For example, if addresses are being sorted by zip code, then the zip code is the key field. After sorting, the addresses will be arranged according to zip code, with the lowest zip code at the top of the list and the highest at the bottom. As a result of placing the addresses in ascending numerical order according to zip codes, the last names will not necessarily be in alphabetical order.

Another example: If a four-column table is sorted according to the list of words in the third column, the third column is the key field. Sorting rearranges the records (rows) so that the third column will be in alphabetical order. As a result of placing the third column in alphabetical order, the other three columns are not likely to follow any particular order.

14.1 INTRODUCTION TO THE CODE FOR THE SORT COMMAND

When the Sort command is given, the word **aSort** is executed. **aSort** -- shown below -- rearranges the highlighted text in ascending order (A to Z, 0 to 9):

```
: aSort  ( -- | ascending sort )  descending off  sort ;
```

The word **dSort** is executed when the user holds down the Shift key when giving the Sort command. **dSort** rearranges the text in descending order (Z to A, 9 to 0):

```
: dSort  ( -- | descending sort )  descending on  sort ;
```

Each of these words calls **sort** (shown below). **sort** contains almost the entire process of the Sort command:

```
: sort  ( -- )
 indsort rule                    ( Turn the "SORT" light on )
 presort                         ( Prepare the selection )
 if  buildlist                   ( Build the initial sort list )
    0 sorttop quicksort drop     ( Sort the list )
    sorttop shuffle              ( Shuffle the records into place )
    pop op to  display           ( Redraw the screen contents )
 else  nosort error              ( Warn user )
    gap prevchar dup bos <>
    if  bos op to  then          ( Deselect and prepare for reselect )
    bos to  redisplay            ( Show collapsed highlight )
 then  widecursor                ( Set cursor to wide )
 forceop on                      ( Further typing marked for select )
 fixcursor
 0 0 3 indicate  rule ;          ( Turn the SORT light off )
```

14.2 FINDING THE KEY FIELD TO BE USED IN SORTING

Before invoking the Sort command, the user specifies the records to be sorted and the key field to be used. All the records touched by the highlight will be sorted, while the field in which the highlight ends will be the key field. For example, if a four-column table is highlighted with the highlight ends in column 3 of the last record, the records (lines of the table) will be sorted so as to arrange column 3 in alphabetical or numerical order.

After turning the "SORT" sign on in the ruler, **sort** calls **presort** (shown below), which first checks to see whether the selection contains a locked region of text. If it does, sorting stops and the Cat beeps.

```
: presort  ( -- flag \ prepare selection for sorting )
 ( preceding breaks are left alone by sorting )
 ( number of highlighted fields in last record becomes keyfield, )
 ( except that if cursor follows at least sortbreaks breaks )
 ( then the keyfield is 0, highlight is trimmed or enlarged )
 ( so that it ends in sortbreaks breaks )

 showmove? off                    ( move&adjtext won't display )
 bos pop to  undop off
 selected  lockedsel              ( see if selection is locked )
 bos nextchar gap >  if  0 exit  then
 adjustleadingbrks
 gap findfield dup tab# to        ( establish sorting column )
 0<  if  0  exit  then            ( nothing to sort )
 adjustrailingbrks  dup
 if  drop  adjustformats  -1 then ;
```

Then, within **presort**, the phrase

```
gap findfield dup tab# to
0< if 0 exit then
```

counts the fields between the end of the last record and the last character in the highlight, including the field that contains the last highlighted character. This number is stored in the variable **tab#** for use throughout the remainder of the Sort operation.

If the highlight ends on the last character of a record separator, the first (leftmost) field in the record will be used as the key field, and **tab#** will be set to zero. **findfield** will return a **-1** if there aren't enough records to sort, in which case sorting stops and the Cat beeps. Within **presort**, the phrase

```
bos nextchar gap > if 0 exit then
```

determines whether the highlight contains more than one character. If it doesn't, sorting stops and the Cat beeps.

14.3 ADJUSTING THE HIGHLIGHTED TEXT IN SIZE AND CONTENT SO THAT IT CONTAINS ONLY COMPLETE RECORDS

Within **presort**, the word **adjustleadingbrks** trims any break characters from the beginning of the selection so that the highlight begins at the beginning of the record in which it is situated. If the highlight doesn't begin at the beginning of a record, nothing is done.

Later in **presort**, the word **adjustrailingbrks** modifies the end of the highlight so that it ends on the final break character of a record separator. If there aren't enough break characters between the end of the highlight and the end of the text (or the end of the local leap region of text), extra carriage returns will be added.

Finally, the word **adjustformats** in **presort** ensures that the format of the text following the sorted text does not change. This is necessary because a record in the highlighted text might hold the format packet used by the unselected text which follows. If this record should be sorted into a different location in the text, it would take the format packet with it, causing the unsorted text below to receive a new, and possibly unexpected format.

To eliminate this problem, a copy of the last format packet in the selected text is saved just after the gap. After the text has been sorted and rearranged, the saved format packet will be placed after the last record in the newly sorted text.

After **presort** has been executed, the selection must be checked once more to make sure that **presort**'s selection trimming hasn't eliminated the selection.

14.4 CONSTRUCTING A DESCRIPTION OF THE HIGHLIGHTED TEXT -- A LINKED LIST OF SORT ENTRIES

Rather than sorting and rearranging the records in the text area, **buildlist** constructs a description of the highlighted text, and lets **quicksort** sort the description instead. This description consists of a sort entry for each record. A sort entry (described below) is **rsize** (decimal 22) bytes long and contains information about the corresponding record in the text and pointers into it. These sort entries are constructed just below the end of the gap. The figure below shows the contents of memory after the sort entry table has been constructed:

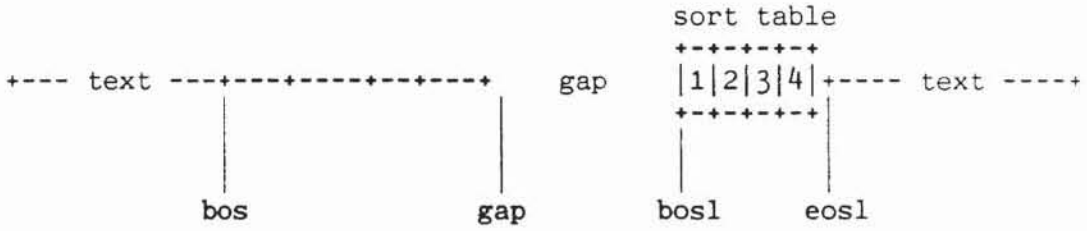


Figure 14. Sort Table With Four Entries

To illustrate the details of the sort table entries and to serve as an example throughout the rest of this chaper, the record list below will be sorted:

```
cat
dog
zebra
bird
fish
ant
lizard
fly
mosquito
```

Figure 14. A Record List Before Sorting

The structure of a sort entry that has just been built is shown in the table below. There are seven fields altogether:

1 record-address field (4 bytes)
1 key field offset (2 bytes)
2 length fields (2 bytes each)
3 link fields (4 bytes each)

The first column in this table is the address at which the sort entry begins.

list	slink	ulink	olink	record	len	off	flen	first 10 bytes
43C792	43C77C	410270	43C77C	42F5B6	4	0	3	cat.dog.zebra.bi
43C77C	43C766	43C792	43C766	42F5BA	4	0	3	dog.zebra.bird.f
43C766	43C750	43C77C	43C750	42F5BE	6	0	5	zebra.bird.fish.
43C750	43C73A	43C766	43C73A	42F5C4	5	0	4	bird.fish.ant.li
43C73A	43C724	43C750	43C724	42F5C9	5	0	4	fish.ant.lizard.
43C724	43C70E	43C73A	43C70E	42F5CE	4	0	3	ant.lizard.fly.m
43C70E	43C6F8	43C724	43C6F8	42F5D2	7	0	6	lizard.fly.mosqu
43C6F8	43C6E2	43C70E	43C6E2	42F5D9	4	0	3	fly.mosquito....
43C6E2	0	43C6F8	0	42F5DD	9	0	8	mosquito........

Table: Sort Table Entries Before Sorting

The sort entries can be displayed with the word **slist**, which is left in the Cat ROM for debugging and for use with this reference manual.

The following table was generated by highlighting the original sort list (above), exiting to Forth with
[Use Front]-[Shift]-[Space], and typing the phrase

presort buildlist re

Then the word **slist** was typed, highlighted, and executed.

The addresses will of course be different for you. In this example, the end of the gap is 43C7A8, the first byte past the last sort table entry.

The record address field (**recaddr**) contains the address of the beginning of the record corresponding to the sort entry. The record length field (**reclen**) contains the number of bytes in that record. The key field offset (**foffset**) contains the number of bytes from the beginning of the record to the beginning of the key field for that record. The field length (**flen**) contains the number of bytes in the key field.

The link fields provide three different orderings for the records. Each link field contains either an address pointing to the beginning of another sort entry or a zero.

One of the three link fields from all of the sort entries taken together comprise a chain. The system integer **sorttop** points to the first sort entry in the sorted (or unsorted) chain. Within that entry, the **slink** field points to the second entry, and so on until -- within the last entry -- the **slink** field contains a zero.

Of the three link fields (**slink**, **ulink** and **olink**), only the slink and **ulink** fields are modified by **quicksort**. The **olink** fields hold the original sort order and aren't changed. The **slink** fields will contain the sorted order when **quicksort** is finished. The **ulink** fields will always hold the reverse order used by **quicksort** (with the **slink** fields) for quickly getting around in the list.

Initially the **olink** and **slink** fields will hold the same addresses. After sorting has completed, the **olink** fields will be unchanged but the **slink** fields will contain the sorted order.

The system integer **bosl**, or bottom of sort list, points to the bottom of the sort entry table. The system integer **eosl**, or end of sort list, points to the top of the sort entry table. These values aren't changed by **quicksort**, since they may be needed by **undosort** afterward.

Note that in Table 14. **sorttop** and **eosl** point to 43C792, the first sort entry just before the end of the gap and corresponding to the first record in the highlight. **bosl** points to 43C6E2, the first sort entry after the beginning of the gap and corresponding to the last record in the highlight.

The sort list is built at the end of the text gap, immediately below the undo buffer. For records in typewritten order in text, the corresponding sort entries are created in reverse typewritten order. The sort entry which corresponds to the record at the beginning of the highlighted text just below the undo buffer lies at the end of the sort list and is pointed to by **eosl**.

To build the sort list, **buildlist** starts at the beginning of the selected text and moves forward, isolating records in the text. Each time a record is found, **newnode** allocates gap memory for another sort entry. In order to sort successfully, there must always be enough memory in the gap to hold the largest record in the text selection. **newnode** checks to make sure this memory requirement is met; if not, it stops.

After **newnode** has allocated memory for the new entry, **buildlist** fills in all fields in the entry (see Table 14.).

14.5 REORDERING THE SORT ENTRIES

The sorting routines can sort records into ascending or descending order, according to the value of the flag in the **descending** system integer. If the flag is zero, the records are arranged in ascending order. Otherwise, they are sorted into descending order. The state of **descending** is determined by whether the Shift key is used when invoking the Sort command:

[Use Front]-[,]	**descending** is zero
[Use Front]-[Shift]-[,]	**descending** is non-zero

After the record list is sorted into ascending order, it looks like this:

ant
bird
cat
dog
fish
fly
lizard
mosquito
zebra

The word **quicksort** is the main sorting word. If the selection contains more than seven records, **quicksort** uses a recursive algorithm to sort the records. Selections which contain less than seven records will be sorted with the simpler sorting algorithm performed by the word **selectionsort**.

Working with seven or fewer records, **selectionsort** scans them with **scansublist**, looking for the record that belongs at the beginning. Using **insertrec** -- a highly optimized code word -- **selectionsort** moves this record to the beginning. It then scans all of the records (including the one it just moved) for another record that belongs at the beginning. If another record isn't found, it is satisfied taht the first record has been found. It then does the same thing for the all the remaining records excepting the first until they are all in the correct order.

quicksort counts records first to make sure that it has more than seven. If it has seven or less, it calls **selectionsort** and then returns to **sort**. If not, it divides the records into three groups: two equally long sublists (upper and lower) and a solitary record between them (pointed to by the integer **focal**).

It scans the upper list (again using **scansublist**) for records that sort below **focal**, and the lower list for records that belong above **focal**. It moves each record that it finds between **focal** and the opposite list. After it is done, the record which focal points to is properly located between two smaller unsorted lists. It passes each of these lists, beginning with the upper one, to **quicksort** -- a recursive call.

The comparisons used by the Sort routines are not strict ASCII string comparison algorithms. If a corresponding series of characters (subfields) in two strings being compared contains digits, the series of characters will be compared as numerical values (as opposed to plain ASCII codes).

This means that certain kinds of lists, such as parts lists, which usually have fields containing alphabetic characters as well as digits, will sort into the expected order. Consider the following list of parts numbers for the Cat.

Original List	Strict ASCII Sorted Order	Cat Sorted Order
7404	7404	74LS14
74LS274	7406	74LS24
74LS138	74LS138	74LS138
74LS24	74LS14	74LS274
74LS14	74LS24	7404
7406	74LS274	7406

Figure 14. Parts List for the Cat

This non-ASCII ordering is accomplished through the use of a translation table called **sortmap**.

The main comparison word used by the sorting routines is $<. This word breaks each key field into numeric subfields and alphabetic subfields. When two subfields are compared, $< first determines what kind they are. If they are both numeric, it calls **comparenumbers**. If either one or both are alphabetic, it calls **comparestrings**.

Each of the latter two words is a rather large code word. Of the two, **comparenumbers** is far more complex because of the need to sort outlines, inventories, addresses, names, and financial statements.

After rearranging the entries in the sort entry table to reflect the ascending sort order, it will look like the table below:

list	slink	ulink	olink	record	len	off	flen	first 10 bytes
43C792	43C77C	43C750	43C77C	42F5B6	4	0	3	cat.dog.zebra.bi
43C77C	43C73A	43C792	43C766	42F5BA	4	0	3	dog.zebra.bird.f
43C766	0	43C6E2	43C750	42F5BE	6	0	5	zebra.bird.fish.
43C750	43C792	43C724	43C73A	42F5C4	5	0	4	bird.fish.ant.li
43C73A	43C6F8	43C77C	43C724	42F5C9	5	0	4	fish.ant.lizard.
43C724	43C750	410270	43C70E	42F5CE	4	0	3	ant.lizard.fly.m
43C70E	43C6E2	43C6F8	43C6F8	42F5D2	7	0	6	lizard.fly.mosqu
43C6F8	43C70E	43C73A	43C6E2	42F5D9	4	0	3	fly.mosquito....
43C6E2	43C766	43C70E	0	42F5DD	9	0	8	mosquito........

Table: Sort Table Entries After Sorting

These sort entries can be displayed with the word **olist**, a word left in the Cat ROM for debugging and for use with this reference manual. The following table was generated by highlighting the original sort list in Figure 14, exiting to Forth ([Use Front]-[Shift]-[Space]), and typing the phrase

presort buildlist 0 sorttop quicksort drop re

Then typing, highlighting and executing the word:

olist

The addresses will of course be different for you.

sorttop points to 43C724, the sort entry pointing to the record "ant" and the first record in the newly sorted order. Within that sort entry, the **slink** field points to 43C750, the sort entry which points to the record "bird".

Within that sort entry, the **slink** field points to 43C792, which is the sort entry pointing to the record "cat". The last sort entry in this chain -- at 43C766, pointing to the record "zebra" -- has in its **slink** field the value of **zero**, the end of the chain.

Comparing this table to 14. we note that only the **slink** and ulink fields have changed. Also, **eosl** still points to 43C792, the first sort entry, which corresponds to the first record in the highlight, and **bosl** still points to 43C6E2, the last sort entry, corresponding to the last record in the highlight.

14.6 REARRANGING THE TEXT TO MATCH THE REORDERED SORT ENTRIES

After the sorting routines have positioned the **slink** pointers to reflect the sorted order for the records, the records must be "shuffled" into place by the word **shuffle**.

sort passes **shuffle** the address (stored in **sorttop**) of the sort entry corresponding to the first record in the sorted order. Beginning with this record, **shuffle** works its way down the **slink** pointers, and appends a copy of each corresponding text record to the end of the sorted list being constructed at the beginning of the gap.

Ideally, if enough memory is available in the gap, the entire body of sorted text records will be constructed in the gap area before being moved back into the correct position in the text.

If there isn't enough memory in the gap, as many text records as will fit will be placed in the gap in the proper order. Those text records in the text that have not yet been moved into the gap will be pushed up in memory towards the gap, overwriting all the records which have already moved into the gap or into their proper place in the text. Then the sorted records in the gap will be moved into the opening created. This process will be repeated as often as necessary.

preshuffle and **postshuffle** take care of format and display details.

14.7 UNDOING THE SORT COMMAND

Since shuffling is the first Sort activity that actually alters the text, the undo operation is set only after shuffling has been completed.

The undo operation for **sort** is **undosort**. undosort goes through the sort list and swaps all **olink** pointers with the **slink** pointers and then uses **shuffle** to place the text records back in their original order. **undosort** also rehighlights the selection so that the selection will be exactly as it was before the sort operation was started.

```
: undosort ( -- )
 indsort rule                    ( Turn "SORT" light on )
 undop off  pop bos to
 eosl swaplinks shuffle          ( Swap all olinks with slinks )
 pop op to  extend               ( Extend the selection )
 ['] redosort undop to           ( Set redosort as the undo op )
 fixcursor
 0 0 3 indicate  rule ;          ( Turn "SORT" light off )
```

Undoing undosort

The undo operation for **undosort** is **redosort**. **redosort** swaps the **olinks** and **slinks** again (so that the sorted order list uses the **slink** field again) and then uses **shuffle** to put the text back into its sorted order.

```
: redosort          ( - )
 indsort rule       ( indicate sorting )
 0 sparepkt !
 undop off  pop bos to
 sorttop swaplinks shuffle
 display  widecursor  forceop on  fixcursor
 0 0 3 indicate  rule ;
```

14.8 SORT ROUTINES SUMMARY

14.8.0 Sort Preparation Routines

adjustformats (-)
(pronounced a-just'for'mats)
Copies the last format packet in the highlighted text (if it exists) to the end of the selection so that **shuffle** can heal the format of the paragraph following the selection when it is finished shuffling records.

adjustleadingbrks (-)
(pronounced a-just'leed'ing-brakes)
Moves the beginning of the highlight past any leading break characters.

adjustrailingbrks (- flag)
(pronounced a-just'trale'ing-brakes)
The end of the selection either (1) contains **sortbreak** consecutive break characters, (2) more than **sortbreak** consecutive break characters, or (3) less than **sortbreak** consecutive break characters. In case (2) the end of the highlighted text is moved backward until condition (1) is met. In case (3) the text after the highlight is scanned for the next set of **sortbreak** consecutive break characters, to which point in text the highlight is extended. If there exists no such set of break characters, the highlight is extended to the end of text and enough break characters are added there.

buildlist (- a)
(pronounced bild'list)
Builds the original sort list and sets up the **bosl** (bottom-of-sort-list) and **eosl** (end-of-sort-list) system integers. **newnode** allocates memory for each new list entry and checks for out-of-memory errors. **nextrecord** isolates text records within the text selected for sorting. **nextfield** finds the key field in the record. Each entry built will have its **slink**, **olink**, **recaddr**, **reclen**, **foffset**, and **flen** fields initialized. The address of the first sort entry created is placed in the **eosl** integer and the address of the last entry placed in the **bosl** integer. Returns the address of the top of the last sort entry.

findfield (a - n)
(pronounced fined'feeld)
Establishes the sorting column or key field within the record list that is about to be sorted. Looks backwards from the address of the end of selection, a, to find the first previous break, that is, the position just before the start of the last record in the highlighted selection. Then steps forward from the start of the last record, keeping a count of how many fields in the last record are highlighted. Returns the column number, or field number, within the record which is to be sorted on. If the entire last record is highlighted, the first field in the record

(n = 0) is the sort field.

newnode (n a - a1 flag)
(pronounced noo'node)
Tries to allocate memory for a new sort entry for the sort list. **newnode** is passed the size n of the largest text record encountered so far and the pointer a to the start of the last record in the sort list being constructed.

If there is not enough memory in the gap to accommodate both the new sort entry and the text for the largest record, **newnode** will abort with a "No room." error message. If **newnode** is being asked to allocate memory for the first sort entry, a will be 0 and **newnode** will initialize the list pointer by positioning it at the first even address which is **rsize** bytes (the size of a sort entry) below the start of the undo buffer. Otherwise, **newnode** decrements the pointer address by **rsize** bytes, stores the address of the previous sort entry in the **olink** field of the entry just created, and returns the new pointer address (which is now the last sort entry in the list).

A flag is also returned: true, if the sort entry just created was the first sort entry; false, if not.

nextfield (a1 - a2)
(pronounced next'feeld)
Given an address a1 of a character in a text record, returns the address a2 of the end of the field in which the character resides. **nextfield** searches forward from a1 looking for the first occurrence of a tab character (tabs are field delimiters). The search progresses byte-by-byte through the record. If a skip character is encountered, **nextfield** will skip across the gap and continue searching. **Note**: this makes for a long search if there are no subsequent tabs.

nextrecord (a1 - a2)
(pronounced next're-kord)
Takes the address a1 of a character in the highlighted selection to be sorted and returns the address a2 of the end of the record to which the character belongs (of the last break character). A real end-of-record must have **sortbreaks** break characters in series. **nextrecord** uses **nextbrk** to find the next occurrence of a break character and **nextchar** to make sure the required number of break characters follow the first break characters.

presort (- flag)
(pronounced pree'sort)
Prepares the highlighted selection for sorting. Uses **findfield** to establish the sorting column. Makes sure that the selection does not include the beginning and ending document separator characters, does not include a trailing page break or document separator character, and that the selection does end with **sortbreaks** break character. If the selection does not end on a break character, it is extended to include the rest of the current line, including the break character at the end it.

prevpkt? (a - a1)
(pronounced preev'pak'it-kwes'tchin-mark)
Looks at the character immediately before the character located at address a in the text. If the previous character is a break character with a format packet following, the address a1 of the start of the format packet is returned. Otherwise, the address a1 of the previous character is returned.

14.8.1 Low-Level Sort Routines

countlist (a1 a2 - n)
(pronounced kownt'list)
a1 and a2 are the addresses of two sort table entries. **countlist** will return the total number of sort entries n between them, inclusive.

getstring (a1 - a2 n)
(pronounced get'string)
a1 is the address of a sort table entry. **getstring** returns the address a2 and length n of the key field within the record to which the sort entry corresponds.

insertrec (**record pointer lowerbound - prevrecord**)
(pronounced in'sert-reck)
record, **pointer**, **lowerbound**, and **prevrecord** are all addresses of sort table entries. **insertrec** modifies the sort table entries so that the entry pointed to by **record** follows the one pointed to by **pointer** and precedes the one that used to follow **pointer**. **prevrecord** is the sort entry pointing to the record that the one pointed to by **record** used to follow. **lowerbound** is used when **record** is the last record in the sort table.

prevrec (**record - prevrecord**)
(pronounced preev'reck)
record is the address of a sort entry. **prevrec** returns the address **prevrecord** of the entry whose sort position is immediately before it (this is another entry whose link field points to **record**. A zero is returned if no entry can be found whose link field points to **record**.

quicksort (**bottom top - newbottom**)
(pronounced kwik'sort)
bottom is the address of the sort entry at the bottom of the chain to be sorted. **top** is the address of the sort entry at the top of the chain to be sorted. The contents of the **slink** and **olink** fields, in the region of the table beginning at **top** and ending at **bottom**, will be modified so that the links will be in sorted order. **newbottom** is the address of the sort entry at the bottom of the modified list. See Part 14.5, "Reordering the Sort Entries," for a more detailed description.

scansublist (**record bottom top direction** - pointer)
(pronounced skan-sub'list)
The sort entries in the part of the table beginning with **top** and ending with **bottom** are compared with **record**. **pointer** is the record that will come after **record** in the search order. **direction** is the search order direction, zero means sorting in descending order and **record** is alphabetically larger than pointer.

selectionsort (bottom top - bottom)
(pronounced see-leck'shun-sort)
As with **quicksort**, described above, **selectionsort** modifies the part of the sort table between **top** and **bottom**, inclusive, so that the entries are in sort order. See the Part 14.5, "Reordering the Sort Entries," for a more detailed description.

14.8.2 Sort Comparison Routines

$< (a1 a1' a2 a2' - f)
(pronounced string' less than)
Compares the two strings which start at addresses a1 and a2, and end at address a1' and a2'. The string is analyzed to determine whether a numerical or string comparison algorithm should be used. The flag will hold one of three values when completed:

Value	Meaning
-1	String 2 is greater than string 1.
0	The two strings are equal.
1	String 1 is greater than string 2.

checksigns (a1 a2 - a1 a2 f1 f2)
(pronounced chek' synes)
Checks to see if the numbers found in the strings located at a1 and a2 are negative. If the value for string 1 is a negative number (contains a - sign), f1 will be a true flag. Likewise, if string 2 is a negative number, f2 will be a true flag. The original addresses are left on the stack, untouched.

comparestrings (a1 a1' a2 a2' - a1 a2 f)
(pronounced kom-payr strings)
Compares the two strings which start at addresses a1 and a2, and end at address a1' and a2'. The flag will hold one of three values when completed:

Value	Meaning
-1	String 2 is greater than string 1.
0	The two strings are equal.
1	String 1 is greater than string 2.

comparenumbers (a1 a1' a2 a2' - a1 a2 f)
(pronounced kom-payr' num-bers)
Compares the two numbers held in strings which start at addresses a1 and a2 and end at address a1' and a2'. The flag will hold one of three values when completed:

Value	Meaning
-1	Number 2 is greater than number 1.
0	The two numbers are equal.
1	Number 1 is greater than number 2.

@digit (a - c f)
(fetch-digit)
(pronounced fetch' dij'it)
Extracts the next character from the string at address a and returns it on the stack. If the character is a digit or decimal point, the flag returned is true.

signed? (a1 a1' a2 a2' - f)
(pronounced synd' kwes'chun)
Takes passed the start addresses, a1 and a2, and end addresses, a1' and a2', of the two strings being compared. A true flag is returned if at least one of the strings is a signed number (digit preceded by a + or - sign) and if the other string is at least a number (a digit, comma, or decimal point).
These are examples of signed numbers: -3 , -3az , +3 , +3az , +.abz , -,e .
These are examples of numbers: 3 , 3xyz , 3333 , ,ac , .r .

14.8.3 Shuffle Routines

adjustsortlist (**delta top** -)
(pronounced a-just-sort-list)
Adjust the sort table entries beginning with the first one (pointed to by **top**) to reflect that the text to which they refer was moved by **delta** bytes, up or down. This word is called by both **preshuffle** and **postshuffle**, which in turn is called by **shuffle**.

largestrec (a1 a2 - a2')
(pronounced larj'est-rek)
Returns the address a2' of the sort entry which corresponds to the text record which has the largest text record address (**recaddr**) that is still less than the text record address found in the sort entry found at address a2 (the sort entry at a2 acts as an upper limit for the search). Used by **moveunsorted**, when there isn't enough room, to determine the next record to be moved out of the area into which the next sorted record will be put.

moverecord (flag dest record - dest' record')
(pronounced moov-re-kord)
Moves the text record corresponding to the sort entry at address record to the destination address dest (usually somewhere in the gap) and returns the address of the next entry in the sort list, record', and the properly incremented destination address, dest'. If flag is true, the text record is moved below the destination address (its last byte will be the byte just before dest), otherwise the text is moved above dest (its first byte will be at dest). The new address of the first byte of the record so moved is stored into the **recaddr** field in its corresponding sort entry.

moverecords (source dest count top -)
(pronounced moov-re-kordz)
Moves count number of bytes of text records, located in memory at address source (usually at the **gap**), into the text, starting at address dest. Before the records are moved, the **recaddr** fields in the corresponding sort entries are updated with the new text addresses. This ensures that the sort table entries will accurately reflect where the records are because undosort will need this information.

moveunsorted (top -)
(pronounced moov-un-sort-id)
Called when the gap has filled up during the shuffling process. **moveunsorted** is passed the address top of the first of the sort entries whose corresponding text records have not yet been placed in their new sorted positions in the text. **moveunsorted** collects all of these unpositioned text records together just below the gap. This allows **shuffle** to continue to place the properly arranged text records (which have accumulated in the gap) into the resulting empty region and thereby free up the gap area for further shuffling.

postshuffle (top prevpointer -)
(pronounced post-shuh-full)
After a sorting operation, the **op** will be positioned on the last break character in the sort selection. If there was a format packet associated with this last break (the **workpkt** area will contain a saved format packet) the saved packet is placed back in the text. The gap interval is marked as completely changed, all intervals in the second partition of text are marked as partially changed, the entire contents of the window table is recalculated and the display redrawn, the **gap**, **bos**, and **eos** are reset, **undosort** is established as the undo operation, and the text is marked as dirty.

preshuffle (top -)
(pronounced pree-shuh-full)
First **preshuffle** checks to make sure there is enough room for the shuffling operation. This is done by examining the length of every record in the sort list beginning with the one pointed to by top. **preshuffle** then checks to see if there is a format packet associated with the last break in the selection. If there is, the format packet is copied into **workpkt** (the scratch format area) temporarily.

shuffle (top -)
(pronounced shuh-full)
shuffle rearranges the text records to match the sorted order specified in the sort list located at the address top. If enough memory is available, the entire sorted body of text records will be constructed in the gap area before being moved back into the correct position in the text. Otherwise, as many records as will fit will be placed into the gap in the proper order. The text records in the text which have not yet been moved into the gap will be collected just below the gap. Any original records which have already been moved into the gap, or into their proper place in the text, will be overwritten during this operation since they are no longer needed. The sorted records in the gap will be moved into the opening created. This process will be repeated as often as necessary. **preshuffle** and **postshuffle** take care of format and display details.

14.8.4 High-Level Sort Routines

aSort (-)
(ascending-sort)
(pronounced aa-sort)
descending is set to false (causing an ascending sort) and **sort** is called.

dSort (-)
(descending-sort)
(pronounced dee-sort)
descending is set to true (causing a descending sort) and **sort** is called.

redosort (-)
(pronounced re-doo-sort)
"Un-does" the effect of an **undosort** operation. **swaplinks** reverses the contents of the **olink** and **slink** fields in all of the sort table entries and then **shuffle** rearranges the records in the text according to the new data in the sort list. The "Sort" indicator light is turned on before the redo operation starts and is turned off after the redo operation ends. **shuffle** sets the word **undosort** as the undo operation for **redosort**.

sort (-)
(pronounced sort)
Sorts the records in the current highlighted selection in ascending or descending order (the flag in the **descending** system integer controls the sort order). **presort** adjusts the boundaries of the selection as necessary before sorting and then **buildlist**, **quicksort**, and **shuffle** perform the main sorting tasks. The "Sort" indicator light is turned on at the start and off at the end of the sorting operation.

swaplinks (a - a)
(pronounced swap-leenks)
Given the address of the top of a sort table entry list, a, **swaplinks** swaps the contents of **olink** and **slink** fields in each sort entry.

undosort (-)
(pronounced un-doo'sort)
"Un-does" the effect of a sort operation. **swaplinks** reverses the contents of the **olink** and **slink** fields in all sort entries in the sort list and then **shuffle** rearranges the records in the text according to the new data in the sort list. The selection is re-highlighted (as it was just before **sort** was used). The "Sort" indicator light is turned on before the undo operation starts and off again after the undo operation ends. The word **redosort** is established as the undo operation.

14.9 SORT INTEGERS SUMMARY

bosl (pronounced bee-oh-ess-ell)
The address of the bottom sort entry position (the sort entry which is located lowest in memory).

descending (pronounced dee-sen-deeng)
A flag which controls the sort order. A true flag causes a descending sort and a false flag causes an ascending sort.

eosl (pronounced ee-oh-ess-ell)
The address of ending sort entry position (the sort entry which is located highest in memory). This is also the beginning of the sort list before it is sorted.

flen (pronounced eff-len)
The offset to a 2 byte field in each sort entry, containing the length of the key field for the corresponding record.

foffset (pronounced eff-off-set)
The offset to a 2 byte field in each sort entry, containing the offset within a text record of the start of the key field within that record.

lip (pronounced ell-eye-pee)
Lower insertion pointer. During **quicksort** and **selectionsort**, this integer points to the sort entry before which the next record will be inserted.

lpp (pronounced ell-pee-pee)
Lower partition pointer. During **quicksort** and **selectionsort**, this integer points to the last in the sublist of sort entries being sorted.

olink (pronounced o-link)
The offset to a 4 byte link field in a sort entry, containing the address of the next sort entry in the original order.

recaddr (pronounced rek-ad-der)
The offset to a 4 byte field in a sort entry, containing text record address.

reclen (pronounced rek-len)
The offset to a 2 byte field in a sort entry, containing the length of the corresponding text record.

rsize (pronounced arr-size)
The size of a sort entry ($16).

sortbreaks (pronounced sort-brakes)
The number of breaks used to separate records (1, 2 or 3). It is set by the Setup command.

sortmap (pronounced sort-map)
The address of the sort table used by **comparestrings** and **comparenumbers**.

tab# (pronounced tab-num-ber)
The number of tabs preceeding the key field within the current record.

ulink (pronounced yoo-link)
The offset to a 4-byte link field in a sort entry, containing reverse order. This is used by **prevrec** and **insertrec**.

upp (pronounced yoo-pee-pee)
Upper partition pointer. During **quicksort** and **selectionsort**, this integer points to the first in the sublist of sort entries being sorted.

sorttop (pronounced sort-top)
The address of the first record in the sorted order list.

15. CALC

Introduction

The Calc command evaluates expressions and insert the evaluated results in the text. The expressions used by the Calc command are ordinary text typed by the user. After evaluation they remain stored, but hidden from view in a data structure called a pocket. Pockets contain the source code for the expression and a Forth token. The Forth token is executed by the Forth interpreter to evaluate the expression. The expression stored in the pocket can easily be retrieved by the user.

15.0 CALC COMMAND GLOSSARY

The special terms used in the description of the calculation package are defined below.

answer field In a compiled expression, 12 bytes are reserved to contain the answer. The first time a compiled expression is executed during the second pass of **recalc** (q.v.), the answer is copied to the answer field. Later executions of that expression don't actually execute the expression; they use the already computed answer stored in the answer field.

arithmetic Stack Arithmetic is performed on the arithmetic stack rather than the Forth data stack. The arithmetic stack uses 12-byte stack elements containing 11 bytes of BCD digits and a tag byte, rather than the 32-bit binary numbers used by the Forth data stack.

attribute byte A byte that follows a character and indicates that the character will be displayed with some special characteristic, such as underlining. The attribute byte allows the description of three attributes: boldface, underline, and dotted underline. All answers and popped expressions used by the Calc command have dotted-underline attribute bytes. Answers and popped expressions may also be underlined or bolded.

autopush Pushing two or more highlighted expressions with a single use of the Calc command. In order to make the calculation package easier to use, several expressions can be popped, edited, then highlighted and pushed all at once in succession, or autopushed. Autopushing takes place in two steps: The edited expressions are compiled during pass 1 and the edited source code is hidden during pass 3 (see recalc).

bit flags The flags that store state information about the compiled expression and the result associated with it. Each compiled expression includes five bit flags in its data structure.

calctoken Pockets in the text always begin with a calctoken (stored as an E4 in text), which makes them easily identifiable. Thus the three scan passes of Calc can simply look for calctokens to find all the pockets in the text.

column From the user's point of view, a column is a vertical grouping of numbers at the same tab stop (decimal tab in the case of numbers). "Column" means something more specific to the Calc package.

The column of a value in a horizontal row of numbers is determined by counting tab characters, starting from the previous return character. Columns 1 and 2, for example, have one tab character between them; thus column 3 is after the second tab character, and so on.

A user's interpretation of a table of numbers will agree with the Calc command's interpretation as long as the tabbing for each column is the same.

An error in interpretation might arise, for example, if a number is long enough to cause a tab field to overflow. In that case the next number in the row will have fewer tabs to its left than the numbers above and below it in the same column. This might mean that a number that appeared to the user to be in column 3 might be in column 2 as far as the Calc command was concerned.

Another example occurs when a return has been left out. In that case, a number in one column may have more tabs between it and the previous return than the numbers immediately above. A number that appears to be in column 3 might be in column 6 as far as Calc was concerned.

Despite some slight possibilities for misinterpretation, columnar calculations work very well when tabbing is done with care.

compiling Compiling is the process of converting the expression that the user types into a compiled expression that can be executed by the Forth interpreter.

dotted underline A dotted underline is an attribute associated with the characters in a result. The dotted underline distinguishes numerals that result from calculations from numerals that result merely from typing.

element An entry on the arithmetic stack.

encoding The process of converting a normal byte into two hidden bytes in the text. Hidden bytes always have their four high-order bits set; the low nibble contains one nibble from the byte being encoded. When an expression is pushed, the token of the compiled expression is encoded, as is the source text of the expression.

execution During **recalc** (q.v.), the token associated with each result is executed. The execution calculates the value for that result.

expression An arithmetic statement that can be compiled and executed to produce a result.

Forth dictionary The data structure used by Forth to store Forth code that is ready to be executed.

hidden text Text that has been encoded and is stored in the text. Hidden text is never displayed.

locked calctoken When a region of text is locked, all the calctokens in the region are converted to locked calctokens. Locked calctokens have the value E5, so the three **recalc** passes (which look for calctokens valued E4) do not find locked calctokens. As a result, these results are not recalculated.

named references An expression that has been given a name. For example, 2+3 can be named cat (highlight and Calc cat:2+3). The name cat can appear in other expressions. For example, cat*5 produces the result 25.00.

NaN Not a number. The value NaN (displayed as >???.??) is returned by any operation that has an overflow -- division by 0, for example. If one of the inputs to an operator is NaN, the result returned by that operator will be NaN. Thus once a result is NaN, then any other results dependent on that result (either by a named reference or a relative reference) will also return NaN.

number formatting The way numbers are punctuated. The Calc package supports several types of number formatting. Every third digit to the left of the decimal point may be set off by a comma, period, or apostrophe, the decimal point may be displayed, and the precision (number of digits to the right of the decimal point) may also be adjusted.

Number punctuation and decimal point can be adjusted to meet local customs as well. In the USA, for example, a comma punctuates the numbers to the left of the decimal, and a period marks the decimal point. In Europe, .

The number of digits to the right of the decimal point is set initially using the Setup command. Once a result is generated, the user controls the precision by deleting characters or adding 0's at the end of a result.

operand Elements of an expression used as input values for operators. Operands may be literals (numbers), named references, or other expressions.

operators Elements of an expression that specify the calculation to be performed. Operators may be unary, taking take only one operand (- and % for example), or they may be binary, taking two operands (+, -, *, /, and the logical operators).

orphans Answer digits that have become separated from their result because the user has edited the text. This can happen by dragging around a result or by deleting the pocket associated with a result (by deleting the first answer digit). Orphans are dotted underlined, but the next execution of **recalc** that ends normally (that is without error or interruption) will remove the dotted underlines, turning the orphans into ordinary text.

overflow A result which is too large (more than 12 digits to the left of the decimal point) to be represented in the number system used by the Calc package. An overflow is represented by NaN (q.v.). NaN is represented in a number as a bit in the tag byte (q.v.). Since overflows are the only cause of NaN's, the NaN bit in the tag byte is also referred to as the overflow bit.

passes The process of recalculation (q.v.) requires three scans through the entire text. These scans are called passes. An integer called pass stores the value of the current pass (1, 2, or 3).

placemarker A special result which is inserted into the text by **precalc**. It remembers the location of the gap so that the cursor can be returned to the gap when Calc finishes. Since Calc can increase or decrease the size of the text (by pushing, popping, or computing answers with a different number of digits), the text pointers are not valid at the end of Calc. Since the placemarker locates the gap, it can be used to adjust all the pointers to their appropriate values when Calc finishes.

pocket The part of a result that is hidden (that is the part that is stored in the text as encoded bytes). The pocket contains the encoded token and the encoded expression.

pointer field Part of the data structure in a compiled expression. Pass 1 fills in the pointer field with the address of the calctoken associated with the compiled expression.

popping Bringing a copy of the hidden expression out of hiding and into the visible text. A popped expression follows the last answer digit and is preceded and followed by an underline character. The entire result from the first answer digit to the last underline character has a dotted underlined. Once an expression is popped, it can be edited. The next **recalc** (q.v.) will automatically push any popped expressions.

precision The number of answer digits to the right of the decimal point (adjustable from 0 to 10 digits). The answer stored in the answer field of the compiled expression always has 10 digits to the right of the decimal point, so calculations that depend on this result will always have the full precision available to them.

punctuation Numbers may be punctuated at every third digit to the left of the decimal point. Commas, periods, or apostrophes may be used.

pushing Converting an expression into a result and hiding the expression under the result (see pocketing). The expression is highlighted and the Calc command given, which compiles the expression, encodes the token and the expression, pushes them,

and produces the result.

recalc The routine that calculates all expressions and updates all results. Each time the Calc command is used, **recalc** performs three passes through the text, examining and processing all expressions.

redefinerror A token used by copy-up, getforward, and receive to indicate to **recalc** that the hidden expression associated with a particular pocket must be compiled during pass 1. It is used in cases where the compiled expression is not available (whenever the Forth part of Calc is separated from the text part, specifically the three cases mentioned) to recreate the compiled expression.

reference counts A count of how many times a name is referred to in the text contained in the compiled part of all named. Used in order to permit forward references. If the reference count ever reaches 0, the name, token, and compiled expression can be recycled.

relative reference An expression that refers to a number or expression located elsewhere in a table. This allows the user to combine numbers from two (or more) locations and leave the result at a third location. The relative reference shows the row and column of the number referred to. For example, [2 3] is a relative reference indicating a number 2 columns to the right and 3 rows up.

result The in-text data structure generated by Calc. A result consists of answer digits and a pocket. Results have dotted underlining (q.v.).

row A row is a horizontal line of text in a table. Relative address and the sum operators make use of this concept.

While the user tends to see each line of text in a table as a new row, the Calc package determines the address of a row by scanning for return characters. Consequently, if a line in a table does not end with a return, the user's interpretation of a table's structure will not correspond with the Calc package's interpretation (see discussion in "column").

Page characters and document characters terminate row counting. For example, when **sum** encounters a page character, it quits adding new elements to the sum, even if there are more numbers just before the page character.

skip markers Markers located at the beginning and end of the gap. The markers start with the value "E0" and are followed by a 3-byte offset that holds the number of bytes to the next character in the text.

token Forth is token-threaded and executes tokens. The token refers to code via one level of indirection through the token table, a table of addresses that relate tokens to code addresses. The calculation package uses the Forth data

structures to calculate answers. Each result has an associated token which binds the result to its corresponding compiled (and hence executable) expression via the token table.

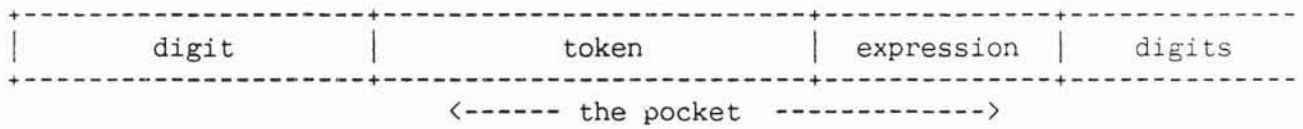
uNaN Means "undefined -- not a number." uNaN is simlar to NaN in that it propagates to subsequent calculations. But instead of being generated by an overflow, it is generated by an unresolved forward reference. uNaN is represented in a number by a bit in the tag byte called the uNaN bit. It is displayed as ????.??

15.1 STRUCTURE OF CALCULATIONS IN THE TEXT

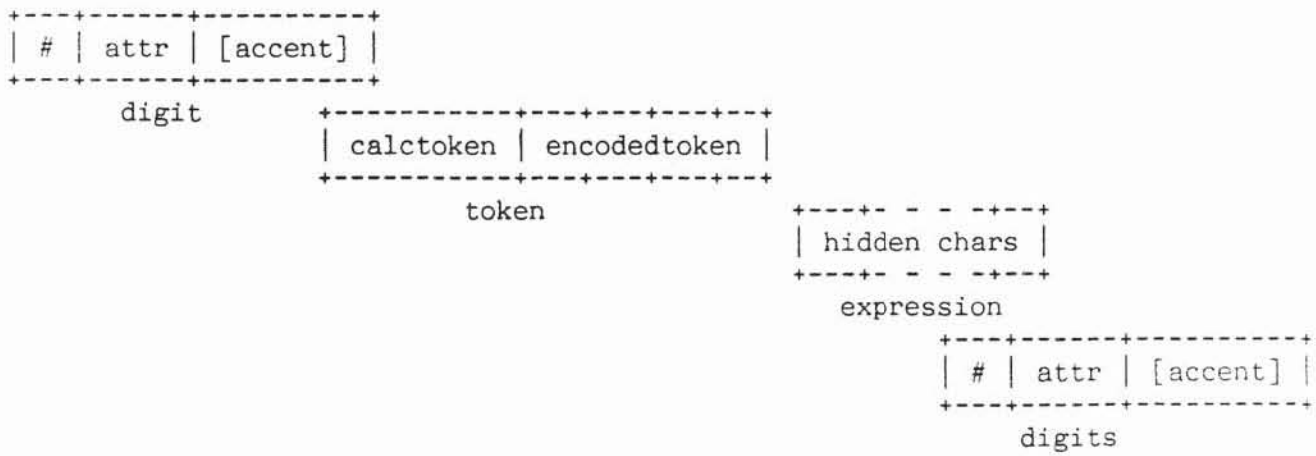
Although calculation results look like ordinary text, they have a very different structure. The first difference is that the calculation data structure is stored in two places: the compiled code that produces the result is stored in the Forth dictionary while the answer and source expression are stored in the text. The data structure in the text is called the result. The data structure in the Forth dictionary is called the compiled expression and is uniquely identified by the token.

A result has a rather complicated structure. It has four main components, (1) the first answer digit, (2) the token, (3) the hidden expression, and (4) the trailing answer digits. The first digit is the only component that is always visible (an answer always has at least 1 digit). The token is executed to calculate the answer. The expression is kept in text so that it can be reviewed or modified. The trailing digits are the remaining digits in the answer. Thus, the hidden information (the token and the hidden expression) is stored in the text between the first digit of the answer and the remaining answer digits. The hidden information is called a pocket.

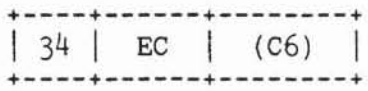
Thus a result is stored in text as follows:



Each of these main components have smaller components:

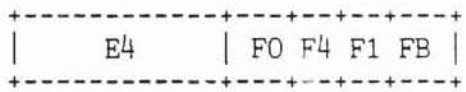


A digit is composed of two (and sometimes three bytes if the user has added accented characters). The first byte is the ASCII code for the visible number. The second byte is an attribute code, with a value between hex EC and EF, indicating which type of emphasis, along with dotted underlining, is applied to the character in the display. Dotted underlining is displayed on the screen but not printed. A third byte is associated with the character if it is accented.



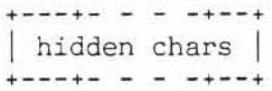
digit

The token component is five bytes long, beginning with a unique byte value E4, called a calctoken. The first byte in a pocket is always a calctoken, so pockets can be found in the text by searching for calctokens. The next four bytes contain a two byte token encoded so that all the high nybbles have the hex value F. For example the token 041B is stored as the encoded value F0F4F1FB. This encoding hides the token and prevents LEAP from landing on it. In locked documents, the calctoken is replaced with the value locked calctoken (which has the unique value E5). The locked calctoken won't be found when searching for calctokens, so pockets preceded by a locked calctoken won't be executed.



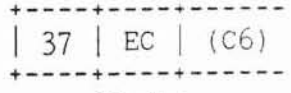
token

When an expression is pushed, a copy of the source text is hidden by encoding it in the same way as is the token. Each byte is divided into nybbles, and each nybble stored in the low nybble of a byte with the high nybble of hex value F.



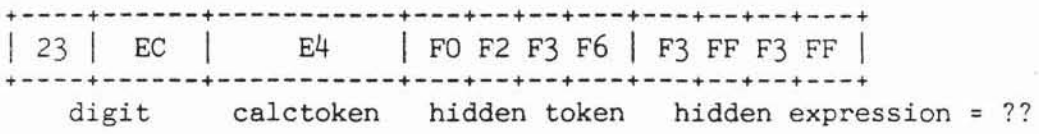
expression

The trailing digits are constructed in exactly the same manner as the first digit. The pocket is attached to the first digit, so when the first digit is erased or moved the pocket goes with it. The trailing digits can be separated from the first one and in that condition are called orphans.



digits

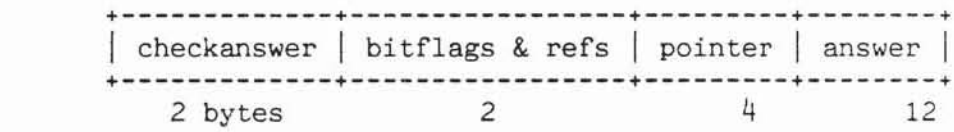
The placemarker structure is a result consisting of a single pound sign (#) with no trailing digits and hidden expression consisting of two question marks (??). The placemarker marks the location of the gap during the execution of **recalc**.



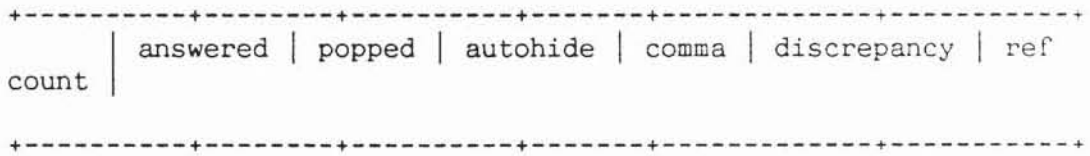
15.2 STRUCTURE OF COMPILED EXPRESSIONS

When an expression is pushed, Forth code is generated, the execution of which computes the answer, which is placed into the text. All Forth words begin with a jump to the **nest** subroutine, which is compiled as "$4ED3", a machine code instruction that performs an indirect jump through register A3. The function of **nest** is to begin interpretation of the Forth word, starting at the next token in the word.

checkanswer is the first word executed by all compiled expressions and is followed by several data structures. These data structures hold information which improve calculation speed, and minimize usage of memory.



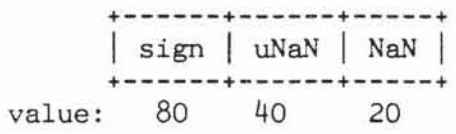
There are 5 bitflags: answered, popped, autohide, comma, and discrepancy. The answered flag is set when the answer field (see below) is valid. The popped flag is set when a copy of the hidden expression is displayed. The autohide flag is set by autopush once the surface expression has been successfully compiled. The comma flag is set when the result is to have commas to mark the thousands places. The discrepancy flag is set when autopush successfully compiles and cleared during pass 3. This bit is used by the next execution of **recalc** to force recompilation of the pocket in case pass 3 was not completed due to an error or user interrupt.



The remaining 11 bits (low 3 bits in this byte and 8 bits in the next byte) are only used if the definition is named. They are for the number of named (as opposed to relative addressed) references to this definition (2,047 maximum - 07ff hex). The reference count determines how many other expressions refer to this expression. The use of the reference count is discussed below in connection with "forwarderror".

During pass1 the pointer field is filled with the location in the text of the calctoken within the result. It is used by sum and relative addressing to extract appropriate values from the surrounding text.

The answer field (which is identical in size and structure to elements on the arithmetic stack) begins with a one byte tag field, which includes the sign bit, undefined bit, and overflow bit. The remaining 11 bytes hold the answer.



The rest of the compiled expression comes after the answer field. checkanswer advances the ip around the data structure so execution will commence with the rest of the compiled expresson. The token for "placeanswer" terminates all pushed definitions; its function is similar to ";" in normal Forth definitions.

"Executing the Calc command -- Pushing," describes the contents of compiled expressions in more detail.

15.3 EXECUTING THE CALC COMMAND

15.3.0 Recalculation

Recalculation occurs when the Calc command is given and there is nothing to push (either nothing is selected or what is selected contains only popped pockets) or when all pushing is completed. Recalculation is performed by the word **recalc**. The basic function of **recalc** is to scan the text three times locating all the calctokens in the text, executing the associated compiled expressions and updating all answers in the text.

When **recalc** begins, the location of the gap is indicated by the blinker, with all text pointers set correctly. The execution of **recalc** performs its function in the following order:

1. Compress the gap -- **precalc**

In normal text operations, the gap begins and ends with a special byte sequence called the skip marker. Since **recalc** scans the text three times, any efficiency matters a great deal. To make scanning faster, before the first scan, the placemarker is placed at the end of the gap, overwriting the skip marker. Then all the text before the gap is moved up against the placemarker. Thus the gap is compressed to 0 and both skip markers are removed. Since there is now no gap in the text, scanning the text for calctokens consists of simply looking at the next byte to see if it is a calctoken; there is no need to take the gap or its associated skipmarkers into account.

The placemarker will be executed during all passes of **recalc**, however, it doesn't do anything except during the third pass, when it removes itself and stores its position in the integer "marker".

Two other optimizations are made to make scanning the text faster. First, since a calctoken is always associated with an encoded token (which takes 4 bytes to store in the text) and at least one expression byte (encoded as two hidden bytes), scanning the text can proceed by checking every 6th character to see if it is >EF and then, if it is, determine whether it is part of a result by scanning backwards for the calctoken. This makes it faster to scan the text since not as many bytes must be examined. This speedup is only used during pass 1 and pass 2. During pass 3, each byte is scanned so that orphans can be turned into plain text.

The other optimization is that a special pocket is located after the end of the text. Every time a calctoken is encountered, a test is made to determine whether it is beyond the end of text (eot). If it is, the scan is terminated.

2. **Recalc** Pass 1 -- **ascan1**

During the first scan the pointer fields in each compiled expression are filled in with the location of the calctoken in the text. This pointer value will be used during pass 2 to evaluate sums and relative expressions. During pass 1 the answer bit flag in each expression is cleared. If the expression was popped (the poppedflag was set), the surface expression is compared to the hidden expression and if they are different, autopush is called. If autopush isn't called, the rest of the compiled expression is skipped.

Autopush, compiles the surface expression, replacing the old compiled expression with the new one, and executes the new compiled expression. Autopush also sets the autohide and discrepancy bits. The newly compiled expression affects the Forth area not the text.

3. **Recalc Pass2 -- ascan2**

During the second scan, each calctoken is executed and placeanswer copies the result of the execution from the arithmetic stack into the answer field and sets the answer bit. Since the compiled expressions are simply Forth code which is executed, executing a token will in turn execute all tokens used in that word. Thus a commonly used name could be executed many times during pass2. The first time a token is executed, the answer bit is set. The second time it is called, it isn't executed; rather, checkanswer copies the answer from the answer field into the arithmetic stack and skips the rest of the compiled expression.

During this pass, one function of checkanswer is to push the address contained in the pointer field onto the return stack. Placeanswer Sums and relative references use the pointer field which locates the pocket in the text. For those named expressions that contain a sum or a relative reference and are themselves refered to only by name, there is no other means of knowing their location in the text. Since pass 2 doesn't insert or delete anything in the text, the pointer field is valid during all of pass 2.

4. **Recalc** Pass 3 -- **tscan** and **textify**

During the third scan all the answers are updated. Several kinds of information are incorporated into the answers extracted from the answer field. The result may have been edited, popped, orphaned, or moved up against another result (by dragging, copying, or deleting). All of this information is taken into account when updating an answer.

The number of digits to the right of the decimal point can be decreased all the way to zero (including and even limited to the

removal of the decimal point). A decimal point can be added. Zeroes can be added to the right of a decimal point. The new precision is determined by scanning for a decimal point (dotted underlined or not) and then counting how many digits (including only dotted underlined digits or zeroes) there are to the right of it.

Commas can be added anywhere in an answer (although to count, it must have been inserted after the first digit and before the last one). This comma is removed and one is inserted to indicate the thousands, millions, and trillions places, if they exist. The commabit is set, so that even if there aren't enough digits to the left of the decimal point during this **recalc**, if there are enough later on, commas will automatically be inserted.

If the autohide bit is set, the hidden expression is discarded and replaced by surface expression. All surface expressions are discarded.

An orphaned result is merely one in which the first digit (and its associated pocket) has been removed (either by dragging, deleting, or copying). In these cases the attribute characters are simply removed from the digits.

If two results are contiguous, pass 3 will insert a tab character. Unfortunately, if the expression associated with the result on the right uses a sum or a relative reference, pass 2 will have used an invalid location in the text and generated an erroneous answer. Because of this, (only for versions 2.00 and later) a flag is set and after pass 3 is done, **recalc** is begun again.

When the placemarker is executed during pass 1 or 2, it does nothing. During pass 3, it removes itself from the text and stores its location into the integer "marker"; this will be the new location for the cursor.

5. Uncompressing the gap -- **aftercalc** and **showcalc**

Once pass three is done, the location of the new cursor is in the integer marker. The text before this location is shifted back down to the beginning of the text area. The structures which support display of text (the interval and window tables) are updated and the text displayed on the screen.

6. Interrupting **recalc** (manually or through error handling)

If **recalc** is interrupted or an error occurs during any of the passes, special routines restore the text to a displayable condition.

7. Inserting tabs causes another **recalc** (not available until version 2.00)

15.3.1 Calc Command Logic

When the user gives the Calc command ([Use Front]-[Calc]), the result depends on the location and state of the cursor in the text. The Calc command logic decides which operation (pushing, popping, or recalc) to perform. The structure of the decisions performed by the Calc command logic is as follows:

```
Highlight Extended?
      Yes:  Does the highlight contain a result?
            Yes:  is everything popped?
                  Yes:  recalc
                   No:   multipop
            No:   does the highlight contain dotted underlines?
                  Yes:  push redefinition, recalc
                   No:   insert tab, push (new expression), recalc
      No:   Is the cursor on a result?
            Yes:  is that result already popped?
                  Yes:  recalc
                   No:   pop
            No:   recalc
```

The highest level of the Calc command is the word Calc. The word getselect figures out whether the highlight is extended and if it is, sets pointers into the selection. If the highlight is extended, the word push|multipop is executed which handles the highest Yes clause above. If the highlight is not extended, the word **pop|recalc** is executed which handles the highest No clause above.

15.3.2 Pushing (Compiling Expressions)

The operation of compiling an expression in the text into a result (an answer plus a pocket) is called pushing. When the Calc logic will determine that pushing is needed when the highlight is extended and there are no calctokens in the highlighted text.

The highlight may contain one or more expressions. If the highlight contains more than one expression, the individual expressions must be separated by separator characters: tab, return, page or document characters. As each expression is identified, that expression is pushed and its answer displayed, resulting in a display that is "animated" showing the progress of pushing.

Pushing consists of several parts:

1 - Scanning the highlighted text -- parser

The highlighted text is scanned looking for items to be compiled. Separator characters are skipped, and individual items that can be compiled (literals, names, :, the operators, and the functions) are identified and passed to the recursive descent

compiler.

2 - Compiling the code

The recursive descent compiler takes the items that were identified by the parser and generates the actual Forth code for the compiled expression. A recursive descent compiler is used because the system supports operator precedence. The operator precedence is described below, and the next section of this manual describes the operation of a recursive descent compiler. When the recursive descent compiler completes compiling an expression, it checks next item produced by the parser. If the parser has reached the end of the highlight or is on a separator character, then the expression has compiled correctly. The compilation completes (by putting the placeanswer code at the end of the compiled expression) and inserts the dummy result.

3 - Inserting a dummy result

Once the expression is compiled, a dummy result in inserted in the text. This result contains the encoded token and expression corresponding to the just-compiled expression, but the answer is set to 0 (in the default case 0.00 since the default precision is 2 digits). The dummy result is not displayed.

4 - Computing the immediate result -- immediacy

Once the dummy result has been inserted in the text, the immediate value of the result is calculated and displayed. This is done by inserting a placemarker (serves the same purpose as the stop marker that terminates the **recalc** passes through the entire text) in the gap right after the dummy result and then setting pass to 2 and executing ascan2, followed by setting pass to 3 and executing textify (which inserts the new result into the text). Finally, appropriate interval table and window table operations are performed to permit display of the intermediate result. This is what causes multipush to "animate" the display: each intermediate result is displayed as multipush pushes a pocket.

5 - Executing **recalc** when pushing is complete

When all pushing is complete, **recalc** is begun, which updates all values in the entire text as well as making sure that the immediate values calculated for the just-pushed expressions are correct.

5.3.3 Operator Precedence

When compiling, the operators are not executed in left-to-right order, rather they are executed according to the operator precedence described below:

highest

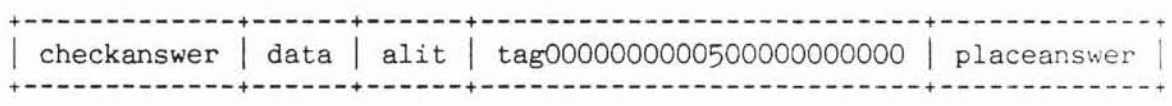
real number, named reference
parenthetical expression
negative (and positive)
logical negate
percent
exponentiation
multiplication, division
addition, subtraction
logical operations (excluding negate)

lowest

The next section describes the actual code that is compiled for various expressions and parts of expressions that the user may enter.

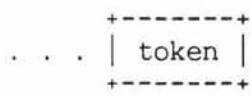
15.3.4 Literals

A single value operand is compiled as an inline literal following the token for alit (arithmetic literal). For a comprehensive example, an expression containing only the constant 50 will look like:



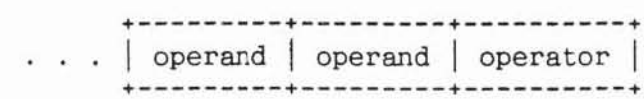
15.3.5 Names

Named references are compiled as the token of the name found in the arithmetic vocabulary. Such named references can be used as an operand in further computations or returned as the answer.



15.3.6 Operators

Most operators remove two operands from the stack. They are compiled in reverse polish order (like an HP calculator) following two operands (or equivalent). The result can be used as an operand in further computations.



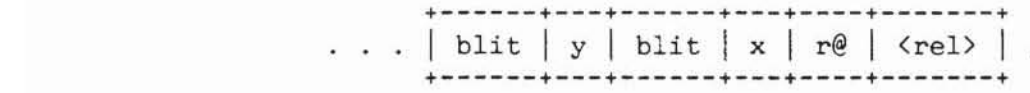
15.3.7 Sums

sum (or avg), which use the pointer field, compile r@ and <sum> (or <avg>). r@ places the address of the flag bits byte onto the

stack. <sum> uses this address to fetch the pointer field, which serves as the starting point for adding vertically. The result returned by <sum> can be used as an operand in further computations.

15.3.8 Relative Addressing

Relative addressing uses two single byte literals to specify the relative position of the value to be extracted from the text. The vertical coordinate is placed on the stack first. The result returned by <rel> can be used as an operand in further computations.



Forward references

Forward references are references to names whose expressions do not exist. They can be created either by using a name in an expression before the name is defined (by pushing an expression with that name), or by deleting a named pocket to which other expressions refer.

When pushing an expression, if a named operand doesn't yet exist (a forward reference), a dummy definition is created whose token is compiled into the new expression's code. The dummy definition is assigned the non-existent operand's name and consists of the token for "forwarderror" followed by a 2 byte reference field (initialized with 1 reference). Since no other information is required, this terminates the definition. A dummy definition returns the value uNaN (displayed in the text as "????.??") to all expressions which refer to it. An encoded token in the text never points directly to a dummy definition.

Other expressions may also use the same nonexistent name. When this happens, the same token (for the dummy definition) is used and the reference count in the dummy definition is incremented by 1.

Later, the user may create a definition for the non-existent name. When this happens, a normal compiled expression (beginning with checkanswer, containing the data structure, code, and ending with placeanswer) is compiled and the reference field from the dummy definition is copied into the new compiled expression and

incremented by 1 (a named pocket uses its compiled expression at least once). The same token points to this new code and the dummy definition is removed. All previously compiled references to the dummy definition now automatically refer to the newly compiled expression, thereby resolving the forward reference.

When a named pocket is deleted, its reference count is decremented by one. If there are no references to it, the referenced count will be zero, in which case the compiled expression, name, and token are recovered. If the reference count isn't zero, there are still references to that name. In this case the code is recovered but the name and token are preserved and a dummy definition is created and the reference count is copied to it.

15.3.9 Recursive Descent Example

The following is an example of a recursive descent compiler that works with a syntax that is similar to the syntax of the Cat Calc command. It is simplified to reduce the size of the example. For this example, the precedence is:

() (parentheses), numbers, variables, and unary + and -
^ (exponentiation)
* or /
+ or -
& or @ (and or or)

```
::=          means "is defined as"
|                  means "or" as in separating choices
...          means "repeat this definition zero or more times but with a loop"
<nnn>        means "nnn is defined here"
                   other characters are literals
num          is a sequence of digits
var          is a variable

<log> ::= <fac> & <fac> ... | <fac> @ <fac> ... | <fac>
<fac> ::= <pro> + <pro> ... | <pro> - <pro> ... | <pro>
<pro> ::= <exp> * <exp> ... | <exp> / <exp> ... | <exp>
<exp> ::= <val> ^ <val> ... | <val>
                    ( if you allow 2^3^5, otherwise <exp> ::= <val> ^ <val> )
<val> ::= + <trm> | - <trm> | <trm>
<trm> ::= num | var | ( <log> )
```

So, to compile: 3+4*5

Begin by examining the 3, and begin at the top of the first routine.
log calls fac
fac calls pro
pro calls exp
exp calls val
val says first item isn't + or - so calls trm
trm says first item is a number. It removes the number from the input stream and compiles the 3. Then it returns to val.
val returns to exp
exp says next item isn't ^ so returns to pro
pro says next item isn't * or / so returns to fac
fac says item is + so it remembers the operator and calls pro again.
pro calls exp
exp calls val
val says current item isn't + or - so it calls trm
trm recognizes the number so it compiles the 4 and returns
val returns to exp
exp says next item isn't ^ so returns to pro
pro recognizes the * so it remembers it and calls exp again
exp calls val
val doesn't recognize + or - so it calls trm
trm recognizes the number so compiles the 5 and returns to val
val returns to exp
exp doesn't see ^ so returns to pro
pro has finished one part of definition so compiles * and doesn't recognize another * or / so returns to fac
fac has finished one part of definition so compiles + and doesn't recognize another + or - so returns to log.
log doesn't see & or @ so returns to who called it.
main says that the input stream is empty so exp was successfully compiled. (if anything left, the expression was ill formed).

We compiled 3 4 5 * + (which is the correct RPN form of the expression).

Now, to write one of these words is very simple. Take pro for example:

```
pro ::= exp * exp ... | exp / exp ... | exp
```

All choices start with exp so it knows it must call exp. Then it checks the input stream for either a * or /. If found it is working on either choice 1 or 2. If not, it was choice 3 and it is complete, just return. If it was choice 1 or 2, remember the operator and call exp again. After it has returned compile the operator and check for a * or / again. If not the choice is complete so return. Otherwise do again. So, the word looks like:

```
: pro ( -- )
 local operator                        ( a place to hold the operator )
 exp                                   ( all choices start with this )
 begin                                 ( a way to do ... )
      item dup ascii * =
            swap ascii / = or          ( is current item * or /? )
 while  item operator to               ( if so, save it )
      parsenext                        ( and remove it from input stream )
      exp                              ( if it was then call exp again )
      operator ascii * =               ( now compile appropriate operation )
      if [compile] f* else [compile] f/ then
 again ;
```

15.3.10 Popping

When the Calc logic determines that one or more pockets must be popped, it pops the pockets one at a time, thereby "animating" popping in a manner similar to the animation produced by multipush.

Popping consists of three parts:

1 - Move the gap to to the end of a result

First the gap is moved to the end (that is one character after) the last answer digit of the result to be popped. This makes it possible to use the gap as a work area to insert the popped expression.

2 - Pop the expression into the gap

Next the hidden expression is unhidden and moved to the gap as a sequence of dotted underlined characters. This operation takes place in several steps. First the gap is checked to make sure that there is enough room for the popped expression. Next the leading underline character is inserted into the gap. Then the dotted underlined expression is inserted into the gap. Finally a trailing underline character is inserted into the gap.

3 - Display the popped result

The interval table and window table are updated as required and the newly popped expression is displayed.

When the display of the popped result is completed, the next result to be popped is searched for. Thus, each result is popped individually, resulting in an animated display.

15.4 ARITHMETIC AND FUNCTIONS -- ARITHMETIC OPERATORS

The arithmetic performed by the calculation package is fixed point BCD with 12 digits to the left of the decimal point and 10 digits to the right. Arithmetic is performed on the arithmetic stack using special arithmetic operators and functions, as opposed to using the Forth data stack and the normal Forth arithmetic operators and functions. Thus when an expression is compiled, the Forth words used in the compiled expression are arithemtic words rather than the ususal Forth words. This section describes the arithmetic words.

15.4.0 The Arithmetic Stack

The first set of words to be described are the words that manipulate the arithmetic stack. These words are similar to the words that manipulate the Forth data stack, but since they operate on the arithmetic stack, all words are preceded by the letter a.

The words are:

alast, **adrop**, **anew**, **NaN**, **uNaN**, **adup**, **aswap**, aover, arot

The words are mostly familiar. **alast** prepares the stack for a new entry by adjusting the stack pointer. **anew** clears the arithmetic stack. **NaN** and **uNaN** put those values on the top of the arithmetic stack.

15.4.1 The Arithmetic Operators (+, -, *, /, %, and Logical Operators)

The next set of words are the acutal operators used to perform operations on the arithmetic stack.

The words are:

a-, a+ These words perform subtraction and addition. They consume two stack entries and produce one entry (the result).

aneg Negates the top stack entry.

a*, a/ Multiplies or divides the top two stack entries. The consume two stack entries and produce one entry (the result).

a% Multiplies the top stack entry by .01.

a<, a>, a=, a~, a|, and a& Used to perform the logical operations provided by the calculation package.

In addition to these words, there are a variety of support words used to implement these words. All words associated with arithmetic and the arithmetic stack are located on pages 1 to 12 of the Disk C side 1 listing.

15.4.2 Functions -- **abs, int, sqrt**

When the user specifies a function in an expression, the following words are compiled into compiled expressions:

aabs Takes the absolute value of the top stack entry.

aint Zeroes the fractional part of the top stack entry.

asqrt Takes the square root of the top stack entry.

These functions are compiled by words that appear in the function vocabulary. The compiling words in the function vocabulary are the same as what the user uses in the expression: abs for absolute value, int for integer value, and sqrt for square root.

The recursive descent compiler detects these function names and compiles the appropriate token from the above list into the compiled expression.

15.4.3 Relative References and Sums

When the user specifies a relative reference or sum (or average) in an expression, the following words are compiled into the compiled expression:

<sum>, <sumdisplay>, <avg> Expects an address on the stack. This is the location of the calctoken (in the text) of the result containing the sum or average.

<rel>, <reldisplay> Also expects an address on the stact which points to the calctoken (in the text) of the result containing the relative reference. In addition, the stack contains numbers that are the x and y offsets referred to by the relative reference.

All five routines return a value on the arithmetic stack. If the referred cell does not contain a number, NaN is returned, otherwise the number is the result of the selected calculation.

These functions are compiled by words that appear in the function vocabulary.

The recursive descent compiler detects these function names and compiles the appropriate token from the above list into the compiled expression.

15.5 SUPPORT FOR ERASE, COPY, DOCUMENT LOCK, COPY-UP, GETFORWARD, AND RECEIVE

Other commands in the system may sometimes have to handle results. This section describes the associated support routines provided by the Calc package.

Erase When a result is erased, the reference counts in the dependent expressions must be reduced accordingly.

Since erasing can be undone, this adjustment is divided into two pieces. During Erase, a linked list of the tokens associated with the results being erased is created (performed by **linkcalc**). The next use of the Calc command descends the linked list and adjusts the reference counts of the affected words in the remainder of the system, and recovers the Forth dictionary space formerly occupied by the erased results (performed by **removecalcs**).

If the erasing is undone, the results in the undo buffer are removed from the linked list (performed by **unlinkcalc**). The results from several consecutive erasures can be accumulated in the linked list.

Copy When the Copy command detects a calctoken (value E4) it calls copypocket, which copies the result in one of three ways. If the result hasn't been popped the answer is copied as plain text (the pocket and the dotted underlines are stripped from the new copy).

If the result is popped, the result is copied "active" in one of two ways. If the expression is not named, the entire result is copied unchanged except that the token is changed to **redefinerror**. If the expression is named, the encoded token is changed to "**redefinerror**" and only the name of the expression is copied (the rest of the expression is discarded, including the colon). The task of redefinerrror is either to compile the expression from the surface text or, if one doesn't exist there, from the hidden expression.

Document Lock When a document is locked, any calctokens in the document (bytes whose value is E4) are changed to locked calctokens (value E5). When the document is unlocked, the reverse is done. Since this operation is so simple no special support is provided by the Calc package.

copy-up, getforward and receive In all of these cases, the text containing results is separated from the associated compiled expressions (in the Forth dictionary). The encoded token is changed to "**redefinerror**" (explained above under Copy).

15.6 ERROR HANDLING

Since **recalc** compresses the text while it is executing, the text is in an abnormal state and cannot be used by the editor. If an error is detected while **recalc** is executing, the text must be returned to its normal state before control is returned to the editor.

On the other hand, if an error is detected while compiling, the Forth dictionary is in an abnormal state and must be returned to its normal state before control is returned to the editor.

Both types of errors set a variable aerror# to a value to indicate which type of error was detected and prepare an Explain message.

To aid in the debugging process, the error numbers have the following interpretations:

Non-Aborting Errors

#	Meaning	Word
30	element is too large or too precise	<sum>, <rel>
31	sumcount won't convert to arithmetic stack	<avg>

Aborting Errors

#	Meaning	Word
29	running out of text space	tscan, recalc, pushpocket
33	missing operator (add op char to op+tokens)	compileop
34	number too large or too precise to compile	allot#
35	too many parentheses	numerical
36	out of dictionary room	numerical
37	number syntax error	numerical
38	more than 2 or less than 1 coordinate	relative
39	no delimiter	getphrase
40	naming collision with a copied up expression	redefinerror
44	stack underflow or overflow	ainterpret
45	can't find	"
46	out of tokens	"
47	name is too long	acreate
48	accented character in expression	ainterpret
49	name already exists	acreate
50	attempt to use a reserved (function) name	acreate
51	syntax error?	clause
52	attempt to push result	compileop
54	no opening paren in use	get(
55	closing without opening parenthesis	buildbody

15.7 LAYOUT OF THE CALC CODE

Disk A Side 0	Page 3	Functions
Disk C side 1	Pages 1-12	Arithmetic code
"	Pages 13-24	**recalc**, checkanswer, placeanswer
"	Pages 24-25	Recycling tokens, remove-word
"	Page 26	Error handling
"	Pages 26-36	Compiling
"	Pages 37-48	Calc command logic
"	Pages 48-52	Relative references
"	Pages 53-55	Support of Erase, Copy, Copy-up, etc.

16. SPELL CHECK LEAP, ADD SPELLING, DELETE SPELLING

Introduction

Canon has supplied a block of code that supports the spelling verification commands. This document describes the interface routines used with Canon's spelling code.

16.0 SPELL CHECK LEAP

The Spell Check Leap function is performed by two words called by the Leap code. The word **spellcheckleap** is used when the user has just pressed the Spell Check Leap key. The word **spellcheckleapagain** is called when the user uses the Leap Again command.

These two words check which Leap key is pressed to determine which direction to check for misspelled words. If the Leap Forward key is pressed, the scan starts at the first word after the gap, loops around the end of text to the beginning of text, and ends by checking the word at the gap. If the Leap Backward key is pressed, the scan starts by checking the word at the gap and then scans backward, looping around to the end of text and ending at the first word after the gap. If no misspelled words are found, the gap is not moved.

The scanning is accomplished by using the code words **nextsep**, **nextsep?**, **prevsep**, **nextnosep**, and **prevnosep**. For example, when scanning forward, the pattern of calls is this: Find the end of the word containing the cursor (if any) by using **nextsep?** Then loop, finding the beginning of the next word with **nextnosep** and then the end of the word with **nextsep**. Once the beginning and end of the word have been found, use the word **translate** to convert the word from the Cat character set to the spelling checker character set. Finally, call spellcheck, which uses the Canon-supplied code and dictionary to determine whether the word exists.

If the word doesn't exist, it is scanned for hyphens. If any are found, each hyphenated part is passed to the spelling checker. If all the parts are found, the word scanning continues, otherwise, the word is considered misspelled.

Misspelled words are displayed by placing the cursor on the first character (using movegap) and updating the screen. **op** is adjusted so that pressing both Leap keys will highlight the misspelled word.

There are two tables used by the spelling code. The table **spellchars** is used by the scanning words (**nextsep**, etc.) to identify separator characters (which appear as an ff in the table) and valid characters (any other value).

spellchars is also used by translate to convert the Cat character codes to character codes that can be used by the spelling checker. This is done by a two-step process. First the character being translated is used as an index into **spellchars**. If the corresponding byte in **spellchars** contains a 0, the character is discarded and translation proceeds. If the byte contains a 1, further translation is required. Finally, if the byte contains any other value, that value is the translated value of the character.

If the byte in the spellchars table contains a 1, then a second table, **spellaccents** is used. This table allows the code to take the 2-byte sequence used by the Cat for accented characters into the 1 byte code used by the spelling checker.

The gap may break the word located at the gap into two pieces. The character at the gap is spellchecked at either the beginning or end of the scan, depending on which way the leap is proceeding. When the word at the gap is translated, a special word, **translategap** is used.

16.1 ADD/DELETE SPELLING

Add/Delete spelling commands scan the text using the same scanning words used by Spell Check Leap. The high-level words for these two commands are **addspelling** and **deletespelling**. When the command begins, if the highlight is extended, it is increased in size so that entire words are always added to the dictionary. Then the word **translate** translates each word in the highlight. When the word is translated, it is **spellchecked**. The result of the **spellcheck** avoids adding words that are in the ROM dictionary to the RAM dictionary. Also, words that do not exist do not need to be deleted. Thus the following tests are used:

spellcheck result:	exists	doesn't exist
addspelling:	noop	addspell
deletespelling:	deletespell	noop

Add Spelling and Delete Spelling can be undone. The **undop** for one is the other (that is the undo for add spelling is delete spelling).

16.2 SPELLCODE INTERFACE

Canon has supplied a block of code to access the spelling dictionary. The code is simply copied into the ROM and there are 5 interface routines that make it easy to execute the various spelling routines from Forth. The interface routines are:

spellcheck	spellcode+2
addspell	spellcode+6
deletespell	spellcode+a
initdictionary	spellcode+e
emptycheck	spellcode+12

These routines use several data structures:

svram	The address of the SV RAM.
svrom0	The address of the SV ROM.
svbuf	The address of a 64 byte RAM buffer used to pass the word to be spellchecked.
svwork	The address of a 256 byte RAM buffer used by the spelling code and **xplen**.

17. EXPLAIN COMMAND

Introduction

The Explain command provides an on-line manual and explains the meaning of error beeps. The code operates by displaying an explanatory message when invoked. When the Use Front key is released, the screen returns to normal and normal editor operation resumes.

The routine error stores the value **%explain** in the variable **curop**. It also sets the variables **xplint** and **xplen**. Normal execution of **equit** moves **curop** to **lastop** in preparation for the next operation.

The Explain command checks the variable **lastop** to see if it contains the value **%explain**. If it does, the user has requested explanation of a previous error. In this case, the message pointed to by **xplint** (for length **xplen**) is displayed as long as the user holds the Use Front key.

If the **lastop** is not **%explain**, then **xplint** and **xplen** are pointed to the default message and displayed just like an error message.

While holding the Use Front key, the user can press any other command key. When this happens, the routine **extexpl** looks in the array **xplntbl** to see if the key corresponds to one of the extended Explain messages. If it does, that message is displayed.

18. TITLES COMMAND

Introduction

The Titles command displays the contents of the first page of all documents with an initial page number of less than 1. The Leap keys can be used to scroll the display up or down one title at a time if all titles won't fit on the screen at once.

TITLES COMMAND

The Titles command is implemented by a word Titles on disk C side 0 page 29. The command operates by blanking the screen, an then scanning the text for document characters. When it finds a document character, it checks the variable #ipage (a 16 bit quantity) to see if the first page is less than 1. If it is, then all the text between the document character and the first page break is defined to be the document title.

Once a title has been found, it is displayed on the screen (including the page break at the end of the title). Then the process is repeated until either there are no more titles or there is no more room on the screen.

If there is more than one screenful of titles, pressing either Leap key will cause the display to scroll by one title at a time. This is done by picking which title will be at the top of the screen and then redisplaying the entire screen using the same procedure described above.

The two Leap keys are key $36 and key $3e, as can be seen in the code.

When Use Front is released, the original text screen returns.

19. DISK

Introduction

The Disk command controls all operations relating to the disk in the drive. This includes moving information to and from the disk, and moving information from one text to another. Having one Disk command greatly simplifies the user's task. Disk's built-in safety features protect him or her from destroying text by trivial error.

By comparing the text in memory to the text on the disk, the Disk command can determine whether to record the text in memory onto disk, play back the text recorded on the disk, or beep and do nothing.

Record transfers text in memory onto the disk for safe storage. Recording removes the last version of the text recorded on that disk, fully replacing it with the new version.

Playback means copying the information from the disk into the memory and putting a portion of it on the screen where it can be seen and worked on. Playback will also copy up the highlighted portion of the old text and insert it into the new text being copied from disk to memory.

A beep is a warning sound made by the Cat when recording or playing back might lose information, such as when the text on-screen has not been recorded, and the disk in the drive is not the disk it came from. When there is a beep, the Explain screen describes the problem and suggests a remedy. A beep is issued and a ruler sign displayed when something unusual has happened. For example, when the Cat plays back a disk recorded on a different version of the Cat a DISK TRANSFER sign appears along with the beep.

19.0 OVERVIEW OF THE DISK COMMAND: RECORD AND PLAYBACK

The Logic of the Disk Command

The Disk command can examines various bits of information about the text both in memory and on the disk to determine the action most likely to satisfy the user. All required information pertaining to the disk is located in one area of the disk called the **idblock**. This is the first structure read from the disk.

Before presenting a detailed discussion of the rules of this logic, a description of the fundamentals lays the ground work.

What Is Recorded and What Is Not

The text, Forth dictionary (including system integers and token table), machine state, and display memory are all recorded on disk. Not recorded on the disk is the memory above **ramend** (the end of text memory), including **system.status**, the disk and print buffers.

Compression

To reduce the time it takes to record these structures, redundant space is removed from them during recording. Space is removed from the text with the word **packtext** and from the Forth dictionary with the word **packforth**. Space is restored during playback with the words **unpacktext** and **unpackforth**, respectively. All of these words are called from the first half of **<save>**.

Playing Back the Text From the Disk

The word **<restore>** reads useful information from the **idblock** (already read from the disk by the Disk command; see "The Logic of the Disk Command, below). After this, it reads the display memory, immediately displaying it on the screen. Then it reads the remaining off-screen contents of the disk into memory.

It then executes the second half of **<save>** by use of a non-standard programming method. The entire machine state, including return stack, program counter and instruction pointer, is recorded. When it is restored (copied back into memory), execution resumes where it left off when **<save>** began writing the image into memory. This creates an unusual situation.

The same code (in the second half of **<save>**) is used by both **save** and **<restore>**. After a "record" is performed, the contents of memory are almost the same as when the "record" began. However, after **<restore>** is done, the contents of memory have been completely changed, including any indication that **<restore>** has

taken place. Both **<restore>** and the latter half of <save> use two special memory areas - **.scratch** and the copy in memory of the idblock, both located at the end of the display memory - to distinguish between them. Once the display memory has been read into memory, it is available for such purposes.

Maintaining Compatibility Across Different Versions

As the Cat program is modified to accomodate various changes, the text and many of its support structures will not function in a Cat program version differing from the one that wrote it onto the disk. However, enough information is recorded on disk in a table called the **idblock** so that a special word, **getforward**, can sort out the differences and place the text into memory areas appropriate to the new conditions. This use of **getforward** to read one Cat version disk on a different Cat version produces the DISK TRANSFER ruler sign.

Copying Up

When a part of the text is highlighted before the Disk command is given, the highlighted text is "copied up" to the new text. This is done by copying the highlighted text to just below **ramend**, reading the disk text in the normal manner and then moving the highlighted copy into the new gap.

Backing Up

Because the information on the disk is not invulnerable, a simple method was provided for making several disks with identical copies of the text in memory. A table called the **idtable** is stored on the disk in the **idblock**. It contains identifying numbers uniquely associated with the text. **idtable** is used by the logic of the Disk command to allow two or more disks to record the same text. A system integer **idadvance** causes this table to be updated in a special way when a backup is recorded.

19.1 THE LOGIC OF THE DISK COMMAND

The Disk command uses text information from both memory and disk to determine the action most likely to satisfy the user. This logic is incorporated into the word **<Disk>**, presented below.

Get this information from Canon Cat source disk C, side 1.

Information About the Text in Memory

The word **cleantext?** examines the system integer **dirtytext?** and, with the word **allselected** determines the extent of the selection. If all of the text has been selected, the first and last document characters being optional, **allselected** returns a true flag.

dirtytext? contains a false flag immediately after text has been played back from disk and until the text in memory has been changed, when it is set to true. All commands except the Print, Send, Send Control, Local Leap, Document Lock, Titles, Leap, Spell Check Leap and Explain commands store a true flag in **dirtytext?**. The only command that sets this integer to "false" is the Disk command.

The word **emptytext?** returns a true flag if there is no text in memory.

Information About the Text Recorded on the Disk

The word **idblock** attempts to read a specific structure from the disk. There are three identical copies of this structure located on tracks 0, 1 and 59 hex and an attempt is made to read them in that order. If all three attempts are unsuccessful, the Disk command can't proceed and a beep is issued with the "NO DISK IN DRIVE" explain screen.

The word **nontextdisk?** examines the drive identifier. If this number is neither 3325 nor 3326, it returns a true flag. If a true flag is returned, the Disk command responds with a beep and the "UNRECOGNIZED DISK" explain screen.

The word **samedisk?** compares the memory **idtable** describing the memory text with the one in the **idblock** it read from the disk. If they are identical, a "true" flag is returned.

The word **backupdisk?** also compares these two tables. If the last n consecutive numbers in the two tables are identical, a true flag is returned. The maximum number of identification number pairs is 32.

THE CONTENTS OF THE ID BLOCK

This information can be found in the words **notepointers** and **noteramsize**.

0		address registers	
4			
8			
C			
10			
14			
18			
1C			
20		data registers	
24			
28			
2C			
30			
34			
38			
3C			
40		xx	
44		xx	
48		xx	
4C		xx	
50		xx	
54		xx	
58		xx	
5C		xx	
60		xx	
64		sr	(2 bytes) 68008 status register
	66	->	(2 bytes) drive identifier
68		->	(2 bytes) number of tracks to write
	6A	xx	
6C		xx	
70		xx	
74		xx	
78		xx	
7C		xx	

80	**romchecksum**	
84	**top**	of Forth
88	**current**	vocabulary into which defs. are compiled
8C	->	end of recorded image
90	xx	
94	xx	
98	xx	
9C	xx	
A0	**idadvance**	-1 indicates a restore
A4	->	address (not 0) is a "copyup pointer"
A8	**system.status**	copy of **system.status** for .<save>
AC	xx	
B0	xx	
B4	xx	
B8	xx	
BC	xx	
C0	xx	
C4	**bot**	
C8	**eot**	
CC	**gap**	
D0	**bou**	
D4	**beot**	
D8	**bos**	
DC	**blackscreen**	color of text
E0	0	?
E4	**diskbou**	beginning of undo on disk
E8	**text**	
EC	**endtext**	
F0	**disktext**	beginning of undo on disk
F4	->	"lockedness" at blinker location
F8	**svid**	svram identifiers
FC	xx	

```
100       ->            idtable begins here (80 bytes long)
104
108
10C
110
114
118
11C
120
124
128
12C
130
134
138
13C
140
144
148
14C
150
154
158
15C
160
164
168
16C
170
174
178
17C       ->            idtable ends with this long word
```

```
180
184     learn0          address
188     ->              length
18C     learn1          address
190     ->              length
194     learn2          address
198     ->              length
19C     learn3          address
1A0     ->              length
1A4     learn4          address
1A8     ->              length
1AC     learn5          address
1B0     ->              length
1B4     learn6          address
1B8     ->              length
1BC     learn7          address
1C0     ->              length
1C4     learn8          address
1C8     ->              length
1CC     learn9          address
1D0     ->              length
1D4     xx
1D8     xx
1DC     xx
1E0     xx
1E4     xx
1E8     xx
1EC     xx
1F0     xx
1F4     xx
1F8     xx
1FC     xx
200     xx
```

Here is an example of the **idblock** taken from the disk the text you are reading was once recorded onto:

```
407600  00 00 11 3E 00 00 6C B4 00 00 04 F2 00 00 04 D8  ...>..l.........
407610  00 00 04 E6 00 01 B3 8D 00 40 7A E8 00 40 7C FC  .........@z..@|.
407620  00 00 00 3E 00 00 FF 11 00 00 00 12 00 00 FF FF  ...>............
407630  00 41 26 E8 00 01 B3 74 00 41 0F D4 00 41 05 B3  .A&....t.A...A..
407640  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
407650  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
407660  FF FF FF FF 20 08 33 26 00 12 FF FF FF FF FF FF  .... .3&........
407670  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
407680  01 64 5D 35 00 42 00 00 00 00 00 FB 00 41 5C 20  .d]5.B.......A\
407690  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
4076A0  00 00 00 00 00 00 00 00 00 45 4C 28 FF FF FF FF  .........EL(....
4076B0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
4076C0  FF FF FF FF 00 42 00 08 00 45 38 00 00 42 00 6A  .....B...E8..B.j
4076D0  00 45 33 CD 00 45 33 D1 00 42 00 69 00 00 00 00  .E3..E3..B.i....
4076E0  00 00 00 00 00 41 57 CD 00 42 00 00 00 45 38 20  .....AW..B...E8
4076F0  00 41 54 00 00 00 00 00 00 04 00 15 FF FF FF FF  .AT.............
407700  51 73 41 E6 00 00 00 00 00 00 00 00 00 00 00 00  QsA.............
407710  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
407720  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
407730  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
407740  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
407750  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
407760  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
407770  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
407780  00 41 51 7E 00 00 00 00 00 41 51 86 00 00 00 2C  .AQ~.....AQ....,
407790  00 41 51 BA 00 00 00 00 00 41 51 C2 00 00 00 00  .AQ......AQ.....
4077A0  00 41 51 CA 00 00 00 00 00 41 51 D2 00 00 00 00  .AQ......AQ.....
4077B0  00 41 51 DA 00 00 00 00 00 41 51 E2 00 00 00 00  .AQ......AQ.....
4077C0  00 41 51 EA 00 00 00 00 00 41 51 F2 00 00 00 00  .AQ......AQ.....
4077D0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
4077E0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
4077F0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
```

DISK ROUTINES

allselected (-> f)
(pronounced all-se-lec'ted)
Returns a "true" for the parameter "f" if the entire text is selected.

backup (->)
(pronounced bak'up)
Saves on disk but dosn't adavance the id number before saving.

backupdisk? (-> f)
(pronounced bak'up-disk-ques'chun)
Returns a "true" for parameter "f" if the disk is a backup disk of the disk whose text appears on the screen.

Bdisk (->)
(pronounced b disk)
Makes a backup disk on drive A, side 0
Makes

Bdisk1 (->)
(pronounced b disk 1)
Makes a backup disk on drive A, side 1

BdiskB (->)
(pronounced b disk b)
Makes a backup disk on drive B, side 0

BdiskB1 (->)
(pronounced d disk b one)
Makes a backup disk on drive B, side 1

<Bdisk> (->)
(pronounced bracket b disk)
Root word of **Bdisk**, **Bdisk1**, **BDiskB** and **BdiskB1**. Makes a backup disk on the selected disk side.

cleantext? (-> f)
(pronounced cleen'text-ques'chun)
Returns a "true" if the text is clean, or if the entire text is selected.

copyup (->)
(pronounced kaap'ee-up)
Copy selected text up to new disk if there is enough room and destination text is unlocked.

Disk (->)
(pronounced disk')
Save, playback or copyup text on disk A, side 0.

Disk1 (->)
(pronounced disk' one)
Save, playback or copyup text on disk A, side 1.

Bdisk (->)
(pronounced bee' disk)
Save, playback or copyup text on disk B, side 0.

Bdisk1 (->)
(pronounced bee' disk one)
Save, playback or copyup text on disk B side 1.

<Disk> (->)
(pronounced bracket disk)
The root word for **Disk**, **Disk1**, **Bdisk** and **Bdisk1** that performs the disk function for the selected disk side.

disk>mem (**offset track# addr len -- error-flag**)
(pronounced disk'-too-mem)
Reads data on disk into memory. Transfers "len" number of bytes from the disk location specified by the parameters "offset" and "track#" into the memory location specified by "addr". The parameter "errorflag" is the result code of the disk reading, and signals "no error" or an error of a certain type.

diskaddr (**addr -> o t**)
(pronounced disk-ad'er)
Given the address on the stack, calculates the offset and track number on a disk.

diskcmd? (-> f)
(pronounced disk' com-mand' qwes'chun)
Returns a "true" for the parameter "f" if the last command executed was the Disk command.

displaydisk (->)
(pronounced dis-play' disk')
Displays the last screen image recorded on the disk.

driveA (->)
(pronounced drive' ay')
This word is unique to development Cats that have two disk drives. Selects the "A" or first drive for all disk operations.

driveB
(pronounced drive bee')
This word is unique to development Cats that have two disk drives. Selects the "B" or second drive for all disk operations.

emptytext? (-> f)
(pronounced emp'tee text ques'chun)
Returns "true" for the parameter "f" if the text is empty (no characters in the text other than the initial document characters).

!id (id ->)
(pronounced stor' eye dee)
Stores the given i.d. into i.d. table moving the other i.d.'s down first if **idadvance** is true.

idblock (-> f)
(pronounced eye dee' block)
Reads the identification block from the disk, returning the parameter "f", which is the result code of the disk reading, and signals "no error" or an error of a certain type.

killdisk (->)
(pronounced kill-disk)
Completely erases any information on a disk.

nontextdisk? (-> f)
(pronounced nahn-text' disk kwes'chun)
Returns a "true" for parameter "f" if the disk is not recognizable as a Cat disk.

samedisk? (-> f)
(pronounced same disk ques'chun)
Returns a "true" for parameter "f" if the disk has the same i.d. number as the disk whose text appears on the screen.

save (->)
(pronounced sayv')
Do the Disk command save part with a new unique identifier.

save&backup (->)
(pronounced sayv' and bak'up)
Makes booted disk a backup of this disk.

savenew (->)
(pronounced sayv' noo)
Clear id table and then **save**.

showdisk (->)
(pronounced sho'disk)
Shows screen from disk until use front key is released.

verify&erase (->)
(pronounced vair-i-fy and ee-rase')
Show screen from disk until use front key is released or until shift erase is pressed then erase disk.

INITIALIZATION WORDS

cold (->)
(pronounced cold)
Executes a complete system reset by executing the 68000's reset instruction which pulls the processor reset line low and resets all external devices. Because the reset line is connected to the processor halt line, it also restarts the processor via <<cold>> ;

<cold> (->)
(pronounced brak'it-cold)
Starts system by executing <<cold>> then does initialization of forth and the editor. Exits to the diagnostic code if the diagnostic switch was pressed. If the diagnostic switch was not pressed, **<cold>** looks for a disk and tries to read it if at all possible. If the disk is readable it is read and the editor is entered but if no disk is present or the disk is unreadable, it simply enters the editor.

<<cold>> (->)
(pronounced brak'it-brak'it-cold')
First code that executes when the power is turned on as its address is in the 68000's startup vector at address 00000004. Initializes both the hardware and the software environment. Initializes the hardware using **reset.hardware**, then clears the screen. Next, reads the country code of the keyboard and records it in the system status area. (If the self-test switch is held down before and during the switching on of power, the system will checksum the ROMs, beep if an error is detected, and execute a RAM test.) Then, copies the tokentable into the RAM, initializes the stacks, sets the processor's internal registers and begins executing Forth. An entrance near the end of <<cold>> is common code for **.level7** to jump into to finish any warm starts and begin Forth execution.

initialize (->)
(pronounced in-i'sha-lyz)
Initializes Forth, hardware related RAM space, and the editor variables.

initinterruptvecs (->)
(pronounced in-it' in'ter-rupt' veks)
Target compiler word that stores the interrupt vectors into the interrupt vector address space.

initkeyboard (->)
(pronounced in-it' kee'bord)
Initializes the autorepeat timers for the keyboard.

.initlist (table - no stack diagram)
(pronounced in-it' list)
A table of 32-bit numbers used to initialize the hardware when the system powers up. The upper eight bits of the 32-bit value is the data that is to be written to the address specified by the lower 24 bits.

initnumbers (->)
(pronounced in-it' num'bers)
Initializes all integers from rom and setting zeroed ones to zero. It also initializes a few arrays to zero.

initstate (->)
(pronounced in-it' stayt)
Initializes the general machine state.

initstrings (->)
(pronounced in-it' strings)
Initialize the RAM strings from ROM and then adjust the affected token table addresses to point to RAM.

inittables (->)
(pronounced in-it' tay-bls)
Initialize the editor command table from ROM and clear the disk id table to zero.

initvocab (->)
(pronounced in-it' vo-cab')
Initialize all of the RAM vocabularies and open the user vocabulary.

reset.hardware (->)
(pronounced re'set-hard'ware)
Initializes the hardware according to the data in **.initlist**.

.s/r (pronounced dot-ess' slash arr)
A 200 byte table is located at the end of the last track containing the screen image. **idblock** copies this table into memory at **.s/r** for use by **save** and **restore** and their respective support words. It contains enough information to transfer everything stored on disk back into memory. It is filled in mostly by **notepointers** and a bit more by **noteramsize**, both of which are called by **<save>** as its first task, shortly before it calls **.<save>**. **.<save>** then fills it in with information about the machine state - the address and data registers and the 68008 status register.

.scratch (pronounced dot' scratch)
This is an area of memory that, while saved to disk, is restored by the word **displaydisk**, rather than by **restore**. It communicates between the first part of **save** and the last part since neither of the stacks nor any integers can serve this purpose. This area is located above the screen and below the area in memory that becomes the **idblock** on disk.

system.status (pronounced sis-tem dot' sta'tis)
Immediately above the end of the text, and below the track buffer, this structure holds 80 bytes (?) of information. The restore value of **ramend** and **trkbuf** are located in it.

idadvance (pronounced eye-dee' add-vanss')
This system integer tells the next invocation of **save** to shift all the text checksum numbers in the **idtable** down by one position in that table before it stores the current checksum. After the **idblock** has been copied from the disk, **restore** puts a -1 (flag) into the long word A0 bytes from the beginning of the copy in memory. .<**save**> (after restore begins executing it) transfers the contents of this long word to **idadvance**.

20. SEND, RECEIVE AND PHONE COMMANDS

Introduction

The telecommunication features of the Cat are supported by the routines described in this section. Also described in this section are the commands Send, Send Control, Send Password, and Phone. The receive part of the system (which is automatic and does not involve a command) is also described.

20.0 THE PHONE COMMAND

The Phone command dials the phone, disconnect the phone at the end of a call, and establish a data connection. The code that implements the Phone command is on Disk C0, pages 1-6.

The Phone command itself is on page 4 of Disk C0. The code first checks to see if the Send command is attached to the serial port. If it is, the Phone command is disabled and no further action is taken. The code then checks for an extended selection, and if the phone is not off-hook, it begins dialing the number contained in the extended selection. Non-numbers are ignored.

First a message appears in the ruler to indicate that dialing is taking place. Then the phone is taken off-hook and there is a 2-second wait for the dial tone. Then a begin-again loop that analyzes the selection is executed. Whenever a number is found, it is dialed. If the character is underlined, it is pulse dialed, otherwise it is tone dialed. This loop also puts in 0.5 second waits when it encounters a comma in the selection.

If the Cat is off-hook, and the selection is not extended, then the Phone command should hang up. It does this by first sending the ETX string, in the case that the other end is a Cat, and then drops the line.

If the Cat is not off hook, the Phone command should attempt to establish a connection. The code first waits for ringing (if there is any) to stop, since it could damage the circuitry if the phone were off-hooked while the ring voltage were on the line. When ringing finishes, the Cat is taken off-hook and an attempt to establish a carrier via **phonetest** for up to **carriertimeout** seconds is made.

If the Phone command was used, we did not autoanswer, so the autoanswer flag is reset.

The support routines used by the Phone command are located either near the Phone command (pages 1-6 of Disk C0), or in the lower-level support words on Disk A0.

20.1 RECEIVE ROUTINES

Receiving characters is automatic in the Cat, so there is no "Receive Command." Rather, every cycle through the main loop checks to see if there is a character ready to receive. The main loop is called **<equit>** and it appears on Disk CO, page 51. Near the end of the routine, a check is made to see if there are any characters received. If there are, they are inserted into the text.

The support words **?rxch**, **rxget**, and **txchr** are the interface between the high-level send and receive words and the lower-level routines. Buffering is performed by the low level words, and **?rxch** is a flag that indicates there is a character in the rx buffer. The interface to the Setup command is also performed by the low-level words, so **rxget** always gets characters from the correct port as defined by the Setup command.

If there is a character ready to receive, the word **receive** is called. **receive** first checks if any of the Use Front or Leap keys are down. If so, receiving will not occur and nothing happens until all such keys are released. Otherwise it checks the state of the **cattocat?** flag and calls the appropriate receive word: **ctocreceive** if it is Cat-to-Cat, and **<receive>** if it is non-Cat-to-Cat.

<receive> simply gets data from the rx buffer (using **rxget**) and inserts it into the text at gap or at op if **forceop** is on (in the case where the user has typed). Then the interval table is adjusted appropriately and the display is refreshed. Finally, the **von** timer is reset as if someone had just typed a key.

ctocreceive receives if the system is in Cat-to-Cat mode. This routine is much more complex than **<receive>** because it checks the received characters to make sure that they represent valid character strings. This checking is performed by the routine **verifychar**, which breaks the received data down into individual elements which can be checked sequentially. In order to make sure that there is a full character plus associated data (for example, a full format packet or a full calculation expression), **ctocreceive** waits until two valid characters are in the receive buffer before taking on character out. This is done because the text would likely be broken if partial hidden text was also in the text. Also, if the user leapt while in the middle of receiving hidden text, the hidden text could be in two separate chunks. This is done because most of the effort required to verify a character consists of making sure that the hidden information in the encoded text is correct.

verifychar breaks the received data into characters that will appear in the text followed by hidden characters. For a break character, the contents of the format packet (if one exists) are checked for validity (including reasonable values for margins, etc.). Document characters are checked for document packets as

well as format packets (if any). The first answer digit of a Calc pocket is checked for a correct calctoken. The token part of the pocket is always set to **redefineerror**. The encoded expression is checked for bad characters, and if any bad characters are found, they are replaced with a question mark (represented as two encoded bytes). Characters are also checked for correct accents and to make sure that reasonable attribute bytes are appended.

If an error is found, the bad characters are replaced by question marks. This can result in a large number of question marks being inserted in the text. If there is a bad byte in a format packet all of the hidden characters following will just be discarded. If two characters have not been received after a one second timeout, the routine assumes that the end of the transmission has occurred and attempts to remove the last character from the buffer and insert it into the text.

20.2 SEND COMMAND

The Send command sends selected (or autoselected) text. Depending on the setup, it will send surface text only or both the text and its underlying structure. The Send command code is on Disk CO, page 49.

The command first tests to make sure whether or not the modem is selected. If it is not, the code attempts to establish a connection. Once a connection is established, the sending mode is checked. If the mode is Cat-to-Cat, then the routine **deepsend**, which sends the text and it's structure, is called. If the mode is not Cat-to-Cat, then a test of the length of the string **sendend$** is made. If this string is length 0 (that is the Setup command has selected **None** for the line-end string), then the routine **unformattedsend** is called. This routine sends all surface characters plus tabs and returns. It does not fill out the margins or tabs with spaces and it does not send the underline, bold, or dotted underline status of the selected text. The contents of pockets or format packets are not sent either.

If the **sendend$** string is not zero length, then the **formattedsend** routine is called. This routine fills out the left margin and tabs with spaces, so a normal terminal at the other end will receive text that looks formatted like it is on the Cat screen. The sendend$ string is sent at the end of each line, so each line will end with a CR or CRLF, depending on the user's choice.

20.3 SEND CONTROL AND SEND PASSWORD COMMAND

The Send Control command sends a control character for each press of a key as long as the Use Front key is held down. Send Password simply sends the character typed as long as the Use Front key is held down. Neither command echoes to the screen. The code for these two commands is on Disk C0 page 50. In addition, there is an array that insures that Send Control only sends control characters according to the keyboard map in the spec. If a key that is not assigned to a control key is pressed, nothing is sent. Send password simply sends the key that is pressed, after converting it to an ASCII character using the same **sendtable** used by the formatted and unformatted Send words.

20.4 COMMUNICATIONS ROUTINES

connectone (--)
(pronounced con-nekt' tone)
Produces the sound effects and Explain message which indicates that a connection has been made.

ctocreceive (--)
(pronounced see' to see' ree-seev')
Handles Cat-to-Cat receiving (read the source code for all of the details, as **ctoreceive** is fairly complicated and handles many cases).

deepsend (--)
(pronounced deep' send)
Sends the text and all the underlying structure to another Cat.

externaldial (addr len --)
(pronounced eks-ter'nul dy'il)
Dials the string to an external Hayes-compatible modem.

formattedsend (--)
(pronounced for'mat-id send)
Sends the current selection with leading spaces and line endings at the end of each displayed line.

hangup (--)
(pronounced hang'up)
Hangs up the phone and throws away any extra characters in the receive buffer.

matchanswerback (-- flag)
(pronounced match an'ser bak)
Checks the incoming stream of characters with the answerback message. Used when deciding if the Cat is talking to another Cat. Returns true flag if the answerback message is fully matched. Otherwise returns a false flag as soon as an error is detected.

modemconnect? (-- flag)
(pronounced mo'dim kon-nekt' kwes'chun)
True if the Cat is using its internal modem or an external modem for its communications port.

noreceiving? (-- flag)
(pronounced no ree-seev'ing kwes'chun)
Returns true if no new activity has occurred in the receive buffer since the last time the variable **oldrxptrs** was updated.

Phone (--)
(pronounced phone')
The Phone command. Handles all dialing, unhooking, and so forth of the phone, stemming from the user's commands. (Phonetest handles the automatic answer and hangup operations.)

phonetest (--)
(pronounced phone' test)
Called every time through the main loop. Checks whether a carrier is present after dialing, verifies that the carrier is still present once a connection has been established, and keeps the call-progress monitoring up to date until a connection is established. Auto-answers the phone if ringing occurs and hangs up when the connection is lost.

receivable? (char -- flag)
(pronounced ree-seev'a-bl kwes'chun)
Performs elementary checking to see if the character is in the range of legal ASCII that the Cat understands. Returns true if so.

receive (--)
(pronounced ree-seev')
Called from the main loop every time there are characters that may potentially be received. Does nothing if a Leap or Use Front key is down, since it would be confusing to receive while trying to perform some command or while leaping through the text. Otherwise it selects the appropriate receive routine.

<receive> (--)
(pronounced brak'it ree-seev')
Handles non-Cat-to-Cat receiving.

receivetable (-- address)
(pronounced ree-seev' tay'bl)
An array used to translate characters coming from an IBM PC-type character set into the Cat's internal character codes.

Redial (--)
(pronounced ree'dy-il)
Redials the last phone number dialed. Similar to the Phone command.

resetphonelight (--)
(pronounced ree'set phone' lyte)
Checks the state of the modem and refreshes the indicators in the ruler.

Send (--)
(pronounced send')
Makes sure there is a connection (or tries to establish one if it can), then calls the appropriate sending routine.

SendCtrl (--)
(pronounced send kon-troll')
Handles sending control characters. It takes keystrokes until the Use Front key is released and translates them into control characters and sends them. Uses the table **specialctrl** for translating non-alphabetic characters into control characters.

sendline (addr-in-lbuff --)
(pronounced send' lyne)
Sends a line in the **lbuff**, starting at the address, with all the formatting.

SendPswrd (--)
(pronounced send' pass'wurd)
Works like **SendCtrl**, but does not translate the characters into control characters.

sendstring (addr len --)
(pronounced send' string)
Sends the given string to the appropriate device.

sendtable (-- address)
(pronounced send' tay'bl)
An array that translates sent characters into the IBM-PC character set.

setmodem (flag --)
(pronounced set' mo-dem)
Handles the handshake between two Cats that decides if the other end is a Cat (to set Cat-to-Cat mode). Returns a "0" if the other send is not a Cat, otherwise returns a "-1".

transend (addr --)
(pronounced tran'send)
Looks at the character and its accents at the address and translates and sends the character. Uses **sendtable**.

unformattedsend (--)
(pronounced un'for-mat-id send)
Sends each character in the selection. Doesn't add any extra characters. Used if **lineend$** is empty.

unhidebyte (pronounced un'hyde byte)
verifyaccentable (pronounced vair'i-fy ak-sent'i-bl)
verifybreak (pronounced vair'i-fy brayk')
verifycalc (pronounced vair'i-fy kalk')
verifydoc (pronounced vair'i-fy dahk')
verifyfmtpkt (pronounced vair'i-fy eff-em-tee' pak'it)
verifynonaccentable (pronounced vair'i-fy non-ak-sent'i-bl)
Used by **verifychar** to ensure that a range of text is legal and won't corrupt the editor.

verifychar (addr end -- addr' end')
(pronounced var'i-fy kair)

Takes two addresses and verifies the next character out of the text. It returns the addresses pointing at a new range that consists of the old text without the last verified character. The end may change if some hidden bytes are removed .

21. PRINTING

Introduction

Given the diversity of printers supported by the Cat, the majority of the print code must be independent of the printer. When the user selects a printer, the Cat generates a set of printer command strings. These command strings tell the printer how to

- perform a carriage return
- feed a sheet of paper out of the printer
- boldface a character
- initialize the printer

and all other required print activities.

Individual printer setup words prepare these strings for the selected printer. The print code can then communicate with the printer through the command strings without having to deal with the idiosyncrasies of each printer.

The other printer-dependent aspect of printing is character availability. Every printer supports a different character set. To circumvent this problem, each printer has a printer table which contains the ASCII codes it requires to print a given character in the defined Cat printer character set. If a particular character is not easily supported by a printer, a special execution vector prints the character in the best available manner.

21.0 MAINTAINING PRINTER INDEPENDENCE

The Cat supports eleven different printers and typewriters, which use three different printing technologies. The table below lists the printers and typewriters supported and the print technology each printer uses. The **printercode** is a value used by the print code to identify the printer currently in use.

printercode	Printer Name	Printer Type
0	Cat180	Daisy wheel
1	VP3103II	Laser beam
2	New AP	Daisy wheel
3	AP400,500,550	Daisy wheel
4	AP300,350	Daisy wheel
5	AP100	Daisy wheel
6	BJ80	Dot matrix
7	FX80	Dot matrix
8	No printer	Not applicable

In order for any new printers/typewriters to be added to the above list (if necessary), the print code had to be written in a very printer independent manner. Ideally, the print code should never have to know which printer it is using. (In reality, a very small number of these special cases do exist.) The two areas in which printer independence is a problem are printer commands and printer character selection.

21.1 PRINTER COMMAND STRINGS

The Cat communicates with a printer through a serial or parallel link. The data to be printed and the command codes which tell the printer how to print the data are sent over this link. Although there is a basic set of commands required to print a page of text, the codes required to implement these commands are usually quite different for each printer.

To allow the print code to remain independent of all printer-specific command codes, a set of printer command strings are defined -- one for each basic print action required. The basic Print command string names, and the actions they perform, are listed below:

backspace" (pronounced bak'spays kwote)
Instructs the printer to perform a backspace

-bold" (pronounced dash' bold qwote)
Instructs the printer to stop boldfacing characters

+bold" (pronounced plus' bold qwote)
Instructs the printer to boldface all subsequent characters

endline" (pronounced end' line qwote)
Tells the printer what to do when it reaches the end of a line

endprint" (pronounced end' print qwote)
Instructs the printer to eject the current page without feeding in another page

evenhalfspace" (pronounced ee'vin haff' spays qwote)
Tells the printer to move forward one half-space from an even half-space position

halfline" (pronounced haff' lyne qwote)
Instructs the printer to move the paper up one half-line

hmi" (pronounced aytch' em eye qwote)
Tells the printer how far to move after printing a character

initprint" (pronounced in-it' print qwote)
Initializes the printer

oddhalfspace" (pronounced odd' haff spays qwote)
Tells the printer to move forward one half-space from an odd half-space position

overstrike" (pronounced o'ver-stryke qwote)
If the printer knows about overstriking (has the variable **knowsos?**) then it will overstrike characters by sending **overstrike"** followed by the two characters it wants overstruck

printforward" (pronounced print' for'word qwote)
Tells the printer to print from left to right (forward)

printreverse" (pronounced print' ree-vers' qwote)
Tells the printer to print from right to left (in reverse)

startline" (pronounced start' lyne qwote)
Instructs the printer to prepare to start printing a new line

topofform" (pronounced top-uv-form' qwote)
Instructs the printer to eject the current page and feed in another

-underline" (pronounced dash' un'der-line qwote)
Instructs the printer to stop underlining characters

+underline" (pronounced plus un'der-line qwote)
Instructs the printer to underline all subsequent characters

unoverstrike" (pronounced un-o'ver-stryke qwote)
Sent after the two characters that need to be overstruck; instructs the printer to start moving the carriage again

Whenever a printer is selected by the user, the contents of these strings are changed. Each printer has a setup word which is responsible for filling each command string with codes understood by the printer. These are the names of the printer initialization words:

Printer	Setup Word
Cat180	**cat180setup**
LBP8	**lbp8setup**
NewAP	**cat180setup**
AP400	**ap400setup**
AP300	**ap300setup**
AP100	**ap100setup**
BJ80	**bj80setup**
FX80	**fx80setup**
No printer	**noprintersetup**

21.2 PRINTER "KNOWLEDGE"

Aside from preparing the command strings for a printer, the printer setup words also set the contents of the printer integers which describe what type of activities the printer can perform, or has "knowledge" of. The main printer knowledge integers are as follows:

boustrophedon (pronounced boo-stref'a-don)
The printer knows how to print bidirectionally.

knowstof? (pronounced nose' top-uv-form' kwes'chun)
Is the printer aware of form feeds?

knowsbold? (pronounced nose' bold kwes'chun)
Can the printer boldface?

knowsul? (pronounced nose' un'der-lyne kwes'chun)
Can the printer underline?

knowsos? (pronounced nose' o'ver-stryke kwes'chun)
Does the printer prefer overstrike to backspace?

knowshmi? (pronounced nose aytch'em-eye kwes'chun)
Does the printer use Diablo-like HMI setting?

ulinehack? (pronounced un'der-lyne hack' kwes'chun)
Translate underlined whitespace to underline characters.

Note: "HMI" stands for "Horizontal Motion Index." If a printer knows about HMI, it is able to adjust the width of a character, that is, the distance the carriage travels after printing a character.

21.3 CHARACTER SELECTION

The character sets supported by all of the printers differ both in content and in the codes used to access a given character. Printer tables hide these character set differences from the print code, translating from the Cat printer character set to the character set supported by an individual printer (see diagram). Each printer has its own printer table.

21.3.0 Printer Tables

A printer table has $10A word-length entries. The data in each entry maps the ASCII code used to represent a character in the Cat printer character set to the actual printer codes used to generate the character.

Entries $00 through $1F, which correspond to non-printable characters, are treated as spaces in the Cat printer character set. Since all printers use a $20 code to represent a space character, all of the printer table entries $00 through $1F consist of a null value in the upper byte and an ASCII space value in the lower byte: $0020.

Entries $20 through $7E correspond to the characters in the standard ASCII character set. Most of the printers respect the standard ASCII character set and use the standard ASCII character codes to print the characters in this range. For this reason, most of these entries will consist of a null value in the upper byte and the corresponding ASCII value in the lower-order byte: $0041 for the character A.

Entries $7F through $CF are for the characters in the extended Cat character set. The extended Cat character set contains special, country-specific characters and accent marks which may or may not be supported by a particular printer. Even if the printer supports a character in this range, it is very likely that the code used to generate the character is different than the code another printer will use. Therefore the codes found in the entries in this range tend to vary widely among the different printer tables. If a character in this range is not supported by a printer, its corresponding entry in the printer' printer table will hold a space value, $0020.

Entries $D0 through $10A are for printing overstrike combinations which are directly supported by some printers but not actual characters in the Cat character set. These entries mainly correspond to the accented vowels and consonants commonly used in foreign countries, for example, "u" with an umlaut on a German daisy wheel.

21.3.1 Handling Character Set Exceptions

When print codes are looked up in a printer table during printing (see **print**), the contents of the upper byte of the printer table entry determines how the character will be printed.

21.3.2 Simple Characters

If the upper byte of the entry holds a zero, the character is a simple character (usually one in the $00 through $7F range). The lower byte of the entry contains the printer code which will be sent directly to the printer.

21.3.3 Overstruck Characters

If the upper byte of the entry holds a value which is greater than $1F, the character is an overstruck character. An overstruck character cannot be printed directly by the printer, but it can be constructed using available characters from the printer.

For example, although the Laser Beam printer cannot directly print an E with a circumflex accent (Cat character code = $E5), it can print both the accent and an E separately. To fake this character, entry $E5 in the Laser Beam print table contains a $B2 (the character code for the accent) in the upper byte, and a $45 (character code for an E) in the lower byte.

When an entry with an overstruck character combination is encountered, the overstrike character is printed first, and then the main character is printed on top of the overstrike character.

21.3.4 Weird Characters and the 'weirdprint Execution Vector

If the upper byte of the entry holds a value between $01 and $1F, the character is a weird character. Weird characters cannot be printed using the normal mechanisms of the print code, so special steps have to be taken. For example, the Laser Beam printer supports character sets for several different countries. To reach a character in one of its international character sets, printer codes which tell the printer to use a new character set must be sent before the weird character can be printed. After the weird character is printed, more printer codes must be sent to set the printer back to its main character set.

When the print routines encounter a weird character, they will execute the routine whose address is found in the **'weirdprint** integer. Each printer has its own weird print routine. The **'weirdprint** integer will always hold the address of the weird print routine for the printer in use.

Figure 21.1 Cat Printer Character Set

21.3.5 FX80 Character Selection

The weird print routine for an Epson FX80 printer is called **fx80magic**. Although the FX80 printer supports a few different country character sets, the main character set it prints from is the USA character set. The **fx80magic** routine allows the FX80 to access characters in its other character sets.

21.3.6 Daisy Wheel Character Selection

A daisy wheel has 96 petals, each holding one character. Ninety-four of those characters are the ASCII codes $21-7E. To access the petals associated with ASCII codes $20 and $7F, a special escape sequence must be sent to the daisy wheel printer. The weird print routine **daisymagic** allow the daisy wheel printers to access the two hard-to-reach character petals.

21.3.7 Laser Beam Character Selection

The Laser Beam printer also supports several country character sets. The **countries** table lists the codes used to select different character sets on the Laser Beam printer:

```
code countries    nx )  jsr,  ;c
            8701  w,    ( IBM1 )
            8702  w,    ( IBM2 )
            642   w,    ( USA )
            641   w,    ( UK )
            645   w,    ( Norway/Denmark )
            64A   w,    ( Japan )
            632   w,    ( Netherlands )
            742   w,    ( IBM1, low half )
            652   w,    ( France )
            633   w,    ( Switzerland )
            64B   w,    ( West Germany )
            630   w,    ( Canada )
```

The Laser Beam weird print routine **LBPmagic** allows characters these other character sets to be reached.

21.3.8 BJ80 Character Selection

The BJ80 printer has no weird characters or print routine.

21.4 PRINT TABLE PATCHING

21.4.0 Daisy Wheel Print Table Patching

The basic character set for each daisy wheel is defined in the **daisy.printer** printer table. Each of the 14 different daisy wheels has approximately 20-23 characters which are different from and are used in place of the characters in the basic set. These daisy wheel specific characters are called "exceptions." Each daisy wheel has an "exception table" which lists the ASCII value of each exception character and the data which defines the replacement character. As an example, let's look at the exception table for the United States daisy wheel:

```
code  usa.dw   nx ) jsr,   ;c
         0023  w,    23    c,      ( sharp sign )
         003c  w,    a4    c,      ( super 2 )
         003e  w,    a5    c,      ( super 3 )
         0040  w,    40    c,      ( @ )
         005b  w,    5b    c,      ( left bracket )
         005c  w,    81    c,      ( plus/minus )
         005d  w,    5d    c,      ( right bracket )
         005e  w,    90    c,      ( degrees )
         9061  w,    86    c,      ( circle-a )
         0060  w,    9b    c,      ( cents )
         007b  w,    ac    c,      ( 1/4 )
         007c  w,    7c    c,      ( vertical bar )
         007d  w,    ab    c,      ( 1/2 )
         007e  w,    b4    c,      ( double underli  )
         007e  w,    c4    c,      ( double underlin  also )
         0100  w,    94    c,      ( paragraph )
         0101  w,    95    c,      ( section )
         -1    w,                  ( End of USA daisy exceptions. )
```

When **usa.dw** is executed, the address of the USA daisy wheel exception table is returned. Each entry in the table is 3 bytes long. The first two bytes contain the new replacement value (print code) for a character. The third byte in each entry contains the printer table offset which is being replaced. The exception data is terminated with a word-length -1 value.

For example, by referring to the table above and the **daisy.printer** printer table (in th ource code) we can see that the default daisy character set would normally cause a space to be sent to the printer whenever the Cat character code $81 is printed. When the printer is using the USA daisy wheel, the USA daisy exception table shows that a $5C code will be installed in the printer table in position $81, causing a plus/minus sign to be printed.

The tokens for each of the 14 daisy wheel exception tables are kept in yet another table, the **DW.countries** table:

```
code DW.countries (  -  a  )     nx ) jsr, ;c
      t' usa.dw       w,          ( offset = 00 )
      t' canada.dw    w,          ( offset = 02 )
      t' latin.dw     w,          ( offset = 04 )
      t' norway.dw    w,          ( offset = 06 )
      t' sweden.dw    w,          ( offset = 08 )
      t' holland.dw   w,          ( offset = 10 )
      t' german.dw    w,          ( offset = 12 )
      t' swiss.dw     w,          ( offset = 14 )
      t' france.dw    w,          ( offset = 16 )
      t' uk.dw        w,          ( offset = 18 )
      t' spain.dw     w,          ( offset = 20 )
      t' italy.dw     w,          ( offset = 22 )
      t' special.dw   w,          ( offset = 24 )
      t' japan.dw     w,          ( offset = 26 )
```

The **DW.countries** table selects which daisy wheel exception table to use based on the print wheel selection given by the user in the Setup command.

21.4.1 BJ80 Print Table Patching

The **bjsecond.dw** table contains the character exceptions to the standard BJ80 printer table. These exceptions are patched over the standard BJ80 printer table contents if the user chooses these exceptions through the Setup command.

21.5 PAPER LENGTH

Paper lengths vary from document to document, sometimes within a single selection; the calculations commands used to set the paper lengths vary from printer to printer. The paper length therefore has to be set after each document break is sent to the printer.

An execution vector for setting the page length is kept in the **'docbreak** integer. Whenever a document break is being output by the **pagebreak** routine, the routine whose address is in 'docbreak is executed to reset the printer's page length/size information. The four page-length routines shared among all the printers are as follows:

CATdocbreak	Set paper length for some daisy wheel printers
LBPdocbreak	Set paper size for LBP printer
ETWdocbreak	Set paper length for 12-steps-per-inch ETW typewriters
bj80docbreak	Set paper length for BJ80 and Epson FX80 printers

21.6 PRINTING TEXT

So far we have described the printing data structures and other printing terminology. All of the routines and integers used to implement printing are described in detail in the routines and integers summary. This section will describe the overall flow of the Print command so that you will be able to use the printing routines summary section knowledgeably.

The word called when the Print command is used is **Print**. Print uses

pickprinter	to set up the print spooler
makeprinttable	to prepare the printer table and patch it if necessary
<Print>	to perform the main printing functions

<Print> uses **printify** to light the "Print" sign on the ruler, **initprinter** to initialize various printing integers and **<<Print>>** to perform the actual printing.

<<Print>> spins in a loop, using **printline** to print one line at a time until the entire selection has been printed, or the user panics, or a page has been printed while "Pause between sheets" is on. **printline** uses **wrap** to step through the text until a line with displayable text is encountered. When a displayable line is found, **build** creates a "**disp**-format" image of the line of text in the line output buffer (**lbuff**).

<printline> steps through the **lbuff** image one character at a time, using **unbuild** to decompose the character for printing, and then using **render** to actually print the character on the page.

render uses **print** to print a single character printer independently. **<printline>** will continue until all characters on the line have been printed.

printline also checks for page break and document characters. If a page break or document character is encountered, **pagebreak** ejects the current page from the printer and to feed in another if necessary.

Refer to the individual routines listed above for further information on the printing process.

21.7 PRINTING ROUTINES

21.7.0 Print Data Tables

BJ80.printer (pronounced bee'jay-ay-tee dot print'er)
Printer code table for Bubble Jet printer.

bjsecond.dw (pronounced bee'jay sek'und dot dee du'bl-yu)
Character exceptions to the Bubble Jet printer table.

countries (pronounced kun'trees)
Table of country codes used to print "weird" characters on the Laser Beam printer.

daisy.printer (pronounced day'zee dot print'er)
Basic version of a daisy wheel print code table. Since each daisy wheel contains different characters in different locations, this table will be copied to RAM and patched whenever a daisy wheel printer is used.

fx80.printer (pronounced eff'eks ay'tee dot print'er)
Printer code table for the Epson FX80 printer.

LBP.printer (pronounced ell'bee-pee' dot print'er)
Printer code table for the Laser Beam printer.

LBPpaper (pronounced ell'bee-pee' pay'per)
Table of Laser Beam printer paper size information.

lbpsmarts (pronounced ell'bee-pee' smarts)
Table of font and character size information for the Laser Beam printer.

printers (pronounced print'ers)
Table used to map a particular **printercode** (numbered from 0-8), to its associated setup word. The **printers** table contains the tokens for the routines used to set up the printers. **setprinter** uses the **printercode** to index into the **printers** table, and executes the token at that offset. The **printers** table is shown below:

```
code printers  nx )  jsr, ;c
      t' cat180setup      w,
      t' lbp8setup        w,
      t' newapsetup       w,
      t' ap400setup       w,
      t' ap300setup       w,
      t' ap100setup       w,
      t' bj80setup        w,
      t' fx80setup        w,
      t' noprintersetup   w,
```

vanilla.unbuild (pronounced vah-nil'la dot un'bild)
Table used to map an overstruck character to its corresponding offset into the printer table. Used by **unbuild**.

wheel>country (pronounced weel' too kun'tree)
Print wheel selection codes for the AP100, AP300, and AP400 daisy wheel printers.

wheel>iso (pronounced weel' too eye'ess-oh)
Print wheel selection codes for the Cat180 and NewAP daisy wheel printers.

21.7.1 Daisy Wheel Exception Data Tables

afrikaans.dw	South African exceptions to the daisy wheel print table.
canada.dw	Canadian exceptions to the daisy wheel print table.
france.dw	French exceptions to the daisy wheel print table.
german.dw	West German exceptions to the daisy wheel print table.
holland.dw	Netherland exceptions to the daisy wheel print table.
italy.dw	Italian exceptions to the daisy wheel print table.
japan.dw	Japanese exceptions to the daisy wheel print table.
latin.dw	Latin American exceptions to the daisy wheel print table.
norway.dw	Norwegian/Danish exceptions to the daisy wheel print table.
spain.dw	Spain exceptions to the daisy wheel print table.
sweden.dw	Swedish exceptions to the daisy wheel print table.
swiss.dw	Swiss exceptions to the daisy wheel print table.
uk.dw	United Kingdom exceptions to the daisy wheel print table.
usa.dw	USA exceptions to the daisy wheel print table.
DW.countries	Table containing the addresses of the country-specific daisy wheel exception data:

```
code DW.countries (  -  a  ) nx ) jsr, ;c
        t' usa.dw      w,   ( offset = 00 )
        t' canada.dw   w,   ( offset = 02 )
        t' latin.dw    w,   ( offset = 04 )
        t' norway.dw   w,   ( offset = 06 )
        t' sweden.dw   w,   ( offset = 08 )
        t' holland.dw  w,   ( offset = 10 )
        t' german.dw   w,   ( offset = 12 )
        t' swiss.dw    w,   ( offset = 14 )
        t' france.dw   w,   ( offset = 16 )
        t' uk.dw    w,      ( offset = 18 )
        t' spain.dw    w,   ( offset = 20 )
        t' italy.dw    w,   ( offset = 22 )
        t' special.dw  w,   ( offset = 24 )
        t' japan.dw    w,   ( offset = 26 )
```

21.7.2 Print Table Construction Words (Used At Compile Time)

,accents (a n c -)
(pronounced kom'ma ak'sents)

Takes each byte-length character value from the string located at address a of length n and lays the value into the printer table under construction as a two byte, accented value where the upper byte contains the ASCII code for the accent c. Used to add many accented characters to a printer table at once.

ch (-)
(pronounced see-aytch')

Places a 2-byte value into the dictionary. Takes the next character from the input stream and lays it into the table being constructed.

,chars (a n -)
(pronounced kom'ma kairs)

Takes each byte length character value from the string located at address a of length n and lays the value into the printer table under construction as a 2-byte value (upper byte = 0). Used to add many characters to a printer table at once.

os (-)
(pronounced oh' ess)

Places a 2-byte value into the dictionary. Takes the next two characters from the input stream and lays them into the table under construction.

,unbuild (n1 a n2 c - n3)
(pronounced kom'ma un'bild)

Used for creating the **unbuild** table. Takes each byte-length character value from the string located at address a of length n2, and adds the value c as the high-order byte of the word, and places that value into the **unbuild** table under construction. A 2-byte printer table offset value is placed into the **unbuild** table immediately after the overstruck character. The original offset value for the string is n1. The offset value is

incremented for each overstruck character placed into the table and the ending offset value n3 is returned on the stack.

w,'s (n1 n2 -)
(pronounced du'bl-yu-commas)
Uses **w,** to lay count n2 occurrences of the value n1 into the dictionary. Used only at compile time.

XXX (-)
(pronounced tri'pl eks)
Adds the 2-byte value $0020 (lower byte = ASCII code for a space) to a printer table. Used as a filler for unprintable or unused characters in a printer table.

21.7.3 Basic Printer Driver Words

backspace (-)
(pronounced bak' spays)
Moves the printer carriage backward one space (two half-spaces). If the printer carriage was moving to the right, **backspace** will cause the carriage to be moved two half spaces to the left and vice versa.

If the printer doesn't have a backspace command, **backspace** will check the carriage direction and output a $08 (backspace) ASCII code to the printer if the carriage is moving to the right and a $20 (space) ASCII code if the carriage is moving to the left.

If the printer "knows" about backspacing, **backspace** will simply send the **backspace"** printer command string to the printer.

backspace also updates **motion** two half-spaces backward.

halfspace (-)
(pronounced haff' spays)
Tries to move the printer carriage a half-space in the current carriage direction.

halfspace checks **oddhalfspace** to see if the next half-space is an odd half-space. If it is, the **oddhalfspace"** printer command string is sent to the printer.

Otherwise, the **evenhalfspace"** printer command string is sent to the printer.

Finally, the contents of the **oddhalfspace** integer are toggled and **motion** is updated one half-space forward.

motion (n -)
(pronounced mo'shun)
Used to help keep track of print-head location. Adds the specified motion n, expressed in half-characters, to the current contents of the **prcol** integer. Before **prcol** is updated, it is clipped to make sure it lies within the current left and right margin boundaries.

newhalfline (-)
(pronounced noo' haff'lyne)
Moves the paper up a half-line. Uses **put"** to send the **halfline"** string to the printer.

newline (-)
(pronounced noo' lyne)
Puts the printer carriage physically and logically at the start of the next line and alters or resets several printer state integers. The **endline"** printer command string physically positions the printer carriage at its new line position. If the printer cannot print bidirectionally, the carriage must always be placed at the left edge of the paper; we don't move the carriage if printing is bidirectional. **prcol** is the integer used to hold the current logical horizontal position of the carriage.

newline uses the phrase **gutter negate prcol to** to set the logical carriage position to the left margin for nonbidirectional printers. **newline** also zeros the contents of the **oddhalfspace**, **bolded**, and **underlined** printer state flags and the **proldflags** integer and increments the **prline** logical vertical page position integer by two half-lines (since we are moving down to a new line).

If underlining or bolding was turned on at the end of the previous line, **newline** will send either the **-bold"** or **-underline"** printer command string to the printer to turn the bolding or underlining off.

paperlength (- n)
(pronounced pay'per length)
Returns the length (in half-lines) of the paper being used for printing. **paperlength** calculates the total length of the paper using the contents of the **#above**, **#long** and **#below** state integers and then subtracts the contents of the **papershort** integer from the total length to calculate the actual paper length.

printc (c -)
(pronounced print' see)
Higher-level version of the word **<printc>**. Simulates a bold character on a printer that doesn't know how to print in boldface. If the printer does not know how to print in boldface and a bold character must be printed, **printc** will doublestrike (print once, backspace, print again) the character c to simulate a bold appearance. Otherwise, the character will be printed just once. Updates **motion** forward by two half-spaces.

printerror (-)
(pronounced print' air'ror)
Presents a system error and aborts.

put" (a n -)
(pronounced put' kwote)
Sends the string at address a of length n to the printer. Uses **<printc>** to send each character individually.

21.7.4 Vertical Paper Motion

formfeed (-)
(pronounced form'feed)
Feeds the current page out of the printer, feeds a new page in if necessary and possible, and resets the logical printer carriage positioning integers.

If the printer understands a "top-of-form" command (**knowstof?** is true), **formfeed** checks to see if either the end of the printable selection has been reached (**#nextwr @ gap >**) or if the user has prematurely terminated printing (**stopprint**). If either of these cases are true, not another page will be printed, so the **endprint"** printer command string is sent to the printer to indicate that the current page should be ejected without feeding in a new page.

If neither of the cases are true, another page will be printed. The **topofform"** printer command string is sent to the printer to indicate that the current page should be ejected and a new page should be fed into the printer.

If the printer does not have a "top-of-form" command **formfeed** will specifically place the carriage at the first line past the end of the page.

formfeed's final actions are to reset **prcol** and **prline** so that the carriage is logically positioned at the left edge on line 0 on the paper. The **backwards** integer is set to false so that printing will commence in a left to right direction.

newpage? (-)
(pronounced noo' payj kwes'chun)
If necessary, nitializes the physical and logical vertical line position of the printer carriage for a new page. If a form feed has just occurred (**prline 0= if**), **newpage?** will try to move the carriage **#above** half-lines down from the top of the paper. **#above** holds the height of the top margin on a printed page, expressed in half-lines. **paperpos** holds the "top-of-form" position for a printer, expressed in half-lines. After a form feed, the carriage will be located **paperpos** half-lines from the top of the new sheet of paper.

If the **#above** position is less than the **paperpos** position, the carriage will be left at its current top-of-form position. If the **#above** position is greater than the **paperpos** position, **toline** will be used to move the carriage the remaining number of half lines required to reach the **#above** position (**#above paperpos -**). The **prline** system integer is initialized to the value held in **#above**.

pagebreak (-)
(pronounced payj' brayk)
Outputs a page break on the printer. This involves printing the page footer if required and possible, performing a form feed, and resetting document characteristics if necessary. If the single page printing mode is in use, **pageprint** sets **stopprint** to true so that printing will stop after the current page is printed.

pageprint calculates the page number for this page and compares it to the page number in the **#iprint** state integer. **#iprint** holds the page number of the first printable page in the current document. If the calculated page number for the current page is less than the **#iprint** page number, **pagebreak** will not print the footer. Also, if for some reason the carriage has already moved below the line on which the footer should be printed, the footer will not be printed.

Whether or not the footer was printed using **printfooter**, **pagebreak** uses **formfeed** to eject the paper and then checks the current character. If the current character is a document character and there is more text to print, **#wr @ nextchar findchar** gets the control variables for the next document; the document break routine for the printer is executed to set the new paper length for the document.

pagebreak? (- f)
(pronounced payj' brayk kwes'chun)
Returns a true flag if the **lbuff** contains either an implicit or explicit page break representation.

page#string (- a n)
(pronounced payj' sharp string)
Formats the page number to be printed at the bottom of a page. Gets the local page number within this document from the **#pgl** state integer and adds it to the start page number for this document, found in the **#ipage** state integer, to form the page number for the current page. Uses the basic Forth pictured numeric output words to create a page number string which includes the right frill string, the page number (positive or negative), and the left frill string. Returns the address and length of the page number string.

printfooter (-)
(pronounced print' foot'er)
Prints the page footer line. Moves the carriage to column 0 of the footer line and prints the **leftfoot"** string, if any. Then, uses **page#string** to calculate and form the page number string and **print"** to print it. The page number string is centered over column 40 of the text. Finally, if a right-hand footer string (**rightfoot"**) exists, it is also printed. The page number is always printed in decimal. The current number base is saved and restored by **printfooter**.

showpage (-)
(pronounced sho' payj)
Uses **displaybos** to display the end of the page just printed and **checkline# rule** to update the ruler display to match the display.

skippage (-)
(pronounced skip' payj)
Advances the control variables over the current page break. If the page break is explicit, **#wr @ nextchar findchar** advances over the page break character. If the break is implicit, and there is only one character on the next page, printing is terminated.

This is because when an implicit page break is selected the first character on the page is also selected.

toline (n -)
(pronounced too' lyne)
Feeds paper until the carriage is positioned at half-line n on the page. If the carriage is already at or beyond the specified half-line, **toline** will do nothing. If an odd half-line is specified **newhalfline** moves the carriage 1 half-line and then **newline** moves the carriage the remaining number of half-lines in 2 half-line increments.

21.7.5 Character Rendering

lbuffend (- a)
(pronounced ell' buff end)
Returns the address of the end of the **lbuff**.

overstrike (c -)
(pronounced oh'ver stryke)
Prints a character without moving the carriage. If the character is white, **overstrike** does nothing. If the character is visible, and the printer knows how to overstrike, the **overstrike**" and **unoverstrike**" printer command strings are used for overstriking. Otherwise, **print** prints the character and **backspace** moves the carriage back.

print (n - f)
(pronounced print')
Prints the character represented by the print code, n, printer independently. If the character is a white character that should be underlined, and the printer chosen does not underline white characters, an underline character, $5F, is output in place of the white character.

Otherwise, the print code indexes into the current printer table to find the 2-byte entry for the character to be printed. If the first byte of the 2-byte entry is zero, the character is a simple, standard ASCII character. The ASCII code for this simple character is taken from the second byte in the entry and printed using **printc**.

If the first byte of the 2-byte entry is a non-zero value greater than $1F, then the character is comprised of two characters, one overstruck over the other. **overstrike** is passed the ASCII code found in the first byte of the 2-byte entry and **printc** prints the character corresponding to the ASCII code found in the 2nd byte of the entry.

If the first byte of the entry is a non-zero value less than $1F, then the character to be printed is a special character in the printer's character set which requires printer specific commands to print. In this case the **'weirdprint** vector is executed to handle printing of the special character.

print" (a n -)
(pronounced print' kwote)
Prints the string located starting at address a of length n to the printer, printer independently. If the printer was printing backwards, the **backwards** integer is set to zero and the **printforward"** command string is sent to the printer. Then **print** prints the characters in the string one-by-one.

render (f -)
(pronounced ren'der)
After **unbuild** has decomposed the next printable character in the **lbuff**, **render** prints the character. First, **render** prepares the printer by checking the underline transition flag **pr\uline/** and the bold transition flag **pr\bold/**. If either of these flags indicates that a font style transition is occurring, **render** will send the appropriate printer command string to the printer: **+underline"**, **-underline"**, **+bold"**, or **-bold"**. The **underlined** and **bolded** integers will be set to true if the character to be printed is to be bolded or underlined.

If the character is to be underlined and the printer does not know how to underline, **render** will use **5f overstrike** to specifically print an underline character in the location where the real character will be placed. If the **prsmall?** integer indicates that the character is a half-wide character, a half-space will be emitted and the character will not be printed. If the character is not a small character, the flag on the stack returned by **unbuild**, is checked.

If the flag indicates that the character to be printed is a character found in the current printer's printer table, the character is passed to **print** for printing (through **prchar print**). If the character was not found in the printer's character set, the codes in the upper and lower bytes of the 2-byte printer code are printed separately, one overstruck on top of the other.

If the code in the upper byte is one of the accent codes in the range $c0 to $cf, **render** makes an additional check to see if a short or a tall accent should be used as the overstrike character. A short accent will be used if the main character to be printed is a lowercase character.

short? (c - f)
(pronounced short' kwes'chun)
Returns a true flag if the character c is a lowercase character.

white? (c - f)
(pronounced whyte' kwes'chun)
Returns a true flag if the character is white. A white character is a character with an ASCII code less that $21, a permanent space character ($93), or an overstrike space character.

21.7.6 Horizontal Motion Control

printblanks (n -)
(pronounced print' blanks)
"Prints" n half spaces on the current line. Used for carriage positioning on printers which do not know how to move directly to a specified horizontal position. If the number of half spaces is odd, **halfspace** will be used to print one half space and then spc **print** will be used to print the remaining even number of half spaces, two at a time. If n is negative, the carriage is not moved.

seektime (n1 - n2)
(pronounced seek' tyme)
Given a destination carriage position n1, expressed in half-characters, **seektime** returns a simpleminded estimate of the "time" required to get to that position starting from the current position by returning the absolute value of the change in distance between the two locations: **(n1) prcol - abs**.

tocol (n -)
(pronounced too' kall)
Moves the carriage to half-character position n on the current line. If the printer does not know how to print bidirectionally, the **startline"** printer command string is sent to the printer to cause the carriage to be moved to the left margin and n half spaces are "printed" on the current line using **printblanks**.

If the printer can print bidirectionally, **tocol** can use one of two methods to move the carriage to the desired position. If the **knowshmi?** integer flag indicates that the printer knows how to modify the character width (hmi), **tocol** will set the character width to 1 inch, calculate the number of whole inches between the current carriage position and the destination, and then will print x spaces, where x is the number of whole inches to move. Each space printed at this point will cause the carriage to move 1 inch in the desired direction.

To move the carriage any remaining distance, **tocol** sets the character width to the width of the remainder distance, prints a space (to move the carriage), sets the character width back to normal, and then terminates execution. If there is no remainder distance, if the original distance was a whole number of inches, **tocol** will simply set the character width back to normal and terminate execution.

If the printer does not know how to modify the character width, **tocol** will try to determine the fastest way to get to the desired position using only spaces and carriage returns. The two possibilities are: (1) move the carriage to the left margin with the use of the **startline"** printer command string and then space over to the desired position, or (2) space directly to the desired position from the current position.

21.7.7 Printing a Line of Text

initprinter (-)
(pronounced in-it' print'er)
Initializes the print-time integers: **proldflags**, **backwards**, **stopprint**, **oldcountry**, **bolded**, **underlined**, **prcol**, **prline**.

printify (-)
(pronounced print'i-fy)
Displays "PRINT" in indicator light 3.

printline (f1 - f2)
(pronounced print' lyne)
Processes one line of text. **printline** is passed a flag f1 which indicates whether the line about to be processes is the first line of text and returns a flag f2 which indicates whether there are more lines of text left to process.

printline will not start processing the line of text until the print buffer has more that $200 bytes of available space. **print.buf.free** checks the available printer buffer space.

Next, **printline** checks to see if the user has prematurely terminated the print command. If the **stopprint** integer is set to true, **quit.print** can stop printing. Also, **UnPanicPrint** is set as the undo operation in case the user changes their mind and does not want to stop the printing.

If printing was not terminated, **printline** checks the contents of **#spr** to determine what type of line is up for processing. If **#spr** holds a 2, **printline** is being asked to print the blank double half-line which is inserted between lines of double spaced text. **printline** will use **newline** to move the carriage down two half-lines and **wrap** twice to decrement the #spr count to zero.

If **#spr** holds a 1, **printline** is being asked to print the blank single half line which is inserted between lines of 1½ spaces text. **printline** will use **newhalfline** to move the carriage down 1 half line and **wrap** to decrement the **#spr** count to zero.

If **#spr** holds a 0, **printline** is being asked to print an actual line of text. **wrap** loads the address of the end of the next line of text into the control variables. This address is then stored into the **#nextwr** integer (**build** needs it) and **prevwrap** restores the control variables for the line of text about to be processed.

build gets an image of the line of text into the **lbuff**. If **pagebreak?** indicates that the **lbuff** contains a page break character, and if the flag passed to **printline** indicates that this is the first page break in the printing session, the page break will be skipped over and ignored (to avoid printing a blank page at the start of each printing session). Otherwise, if a page break which is not the first page break is encountered, **pagebreak** will be used to eject the current page, and **showpage** and **skippage** will be used to get to the next printable page, if any.

If the **lbuff** does not contain a page break, **printline** prepares the line for printing. If the line is the first line on a page, **newpage?** will set up the page parameters. **trimline** trims

non-printable characters from the start and end of the lbuff string. **startline** prepares the printer and the printer integers for printing. If **startline** indicates that the line contains printable characters, **<printline>** prints the line.

newline moves the carriage down by 2 half lines and wrap updates the control variables. **printline**'s final action is to check for more text to print and to return a flag which indicates the outcome of the check.

<printline> (-)
(pronounced brak'it print'lyne)
Print a line of text. Steps through the **lbuff** using **unbuild** until **unbuild** returns a flag which indicates that the end of the **lbuff** has been reached. Each valid character obtained by **unbuild** is printed using **render**. The definition of **<printline>** is
: <printline> begin unbuild while render again drop ;

printposition (a1 a2 - n1 n2)
(pronounced print' po-zi'shun)
Converts the starting **lbuff** print address a1 and the end **lbuff** print address a2 to their corresponding start position n1 and end position n2 on the current line. The start and end position are expressed in half-characters.

startline (a1 a2 - f)
(pronounced start' lyne)
Checks for a blank line of text, initializes the **printlimit**, **printnext** and **backwards** printing integers, and sets the printing direction. If the starting **lbuff** print address a1 and the ending print address a2 are equal, this line is a blank line (no text to print), and a false flag f is returned. If the **lbuff** addresses are not equal, there is text to print.

The address of the first printable **lbuff** character is placed in **printnext** and the address of the last printable **lbuff** character is placed in **printlimit** and a true flag will be returned when **startline** completes execution. If the printer can print bidirectionally, **startline** checks to see if the carriage is currently closer to the start column or end column position for the line. If it is closer to the start position, **tocol** moves the carriage to the start column, the **printforward**" printer command string is sent to the printer, and the **backwards** integer is set to false. If the carriage is closer to the end position, tocol moves the carriage to the end column, the **printbackward**" printer command string is sent to the printer, and the **backwards** integer is set to true.

The **printnext** and **printlimit** integer contents are switched if backwards printing is used. If the printer cannot print bidirectionally, the carriage is moved to the start column position and the **backwards** integer is set to false.

trim1 (a1 a2 - a1 a2')
(pronounced trim' wun)
Given the addresses of the start (a2) and end (a1) of the **lbuff**, **trim1** adjusts the start address so that no unhighlighted characters at the start of the current line are printed.

trim2 (a1 a2 - a1 a2')
(pronounced trim' too)
Given the addresses of the start a2 and end a1 of the **lbuff**, **trim2** adjusts the start address so that no leading white characters at the start of the printable section of the current line are printed. Used after **trim1**.

trim3 (a2 a1 - a2 a1')
(pronounced trim' three)
Given the addresses of the start a2 and end a1 of the **lbuff**, **trim3** adjusts the end address so that no unhighlighted characters at the end of the current line are printed.

trim4 (a2 a1 - a2 a1')
(pronounced trim' for)
Given the addresses of the start a2 and end a1 of the **lbuff**, **trim4** adjusts the end address so that no trailing white characters at the end of the printable section of the current line are printed. Used after **trim3**.

trimline (- a1 a2)
(pronounced trim' lyne)
Given a line of text in the **lbuff**, **trimline** determines which parts of the line can and should be printed and then returns the addresses of the first **lbuff** character a1 and the last **lbuff** character a2 to be printed. Any highlighted character which is not a leading or trailing white character is a valid printable character.

unbuild (- f1 f2)
(pronounced un'bild)
Takes a single character from the **lbuff**, translates it into the printer character set, and sets up the printer flags. f1 indicates whether the character exists in the printer character set. f2 indicates whether or not unbuild has reached the end of the printable characters. If f2 is true, then this character should be printed; if it is false, the character should not.

unbuild uses the address in **printnext** to find the next printable character in the **lbuff**. **unbuild** first checks to see if the character has any associated overstrike character. If there is an overstrike character, **unbuild** will create a word which has the overstrike character code in the upper byte and the main character code in the lower byte (same format as a printer table entry), and will compare the word to the list of overstrike combinations found in the **vanilla.unbuild** table. The **vanilla.unbuild** table contains all of the overstrike combinations which are found in the printer character set ($d0 - $109).

If the overstrike combination is found in the table, it is one of the special characters known to some printers, but not available in the text. The printer character code $d0 - $109 which represents the overstrike combination in the printer character set is placed in the **prchar** integer and a true f1 flag is placed on the stack.

If the overstrike combination is not matched, the 2-byte set of character codes used to represent the character are placed in **prchar** and a false f1 flag is placed on the stack.

Next, **unbuild** checks the character attribute flags which are associated with each character in the **lbuff** and sets the related printing flags accordingly. The **smallbit** sets the **prsmall?** integer. The **invbit** sets the **printed?** integer (only inverted characters are printed). The **ulinebit**, **boldbit**, and **dlinebit** check for underline, bold, and dotted underline style transitions.

The current states of these bits are compared with the character flag bits from the previous character (saved in the **proldflags** integer). A change in any of these bits will cause either the **pr\uline/**, **pr\bold/**, or **pr\dline/** style transition integers to be set to true.

Finally, **unbuild** checks to see if the address in **printlimit** has been reached and then increments/decrements the **printnext** address by 4, depending upon the current printing direction. A flag which indicates whether the **printlimit** has been reached is placed on the stack.

UnPanicPrint (--)
(pronounced un'pan-ik)
Restarts a printing session which was terminated with a panic stop. Uses **restore.print** to reset all of the low level print buffer pointers, **extend** to re-highlight the unprinted text, **printify** to turn the "PRINT" sign on, **<<Print>>** to restart and perform the printing, **indicate** to turn the "PRINT" sign off, and **0 setprinter** to select the default printer when finished. **stopprint** is set to false.

21.7.8 Main Print Words

AltPrint (-)
(pronounced ahlt'print)
Word executed when [Use Front]-[Shift]-[Print] is pressed. Causes the current text selection, if any, to be printed out on the alternate printer. Since the main printer is always the default printer, **setprinter** initializes the alternate printer. **pickprinter** and **makeprinttable** set up the print spooler and printer table and then **<Print>** prints the text. After the completion of printing on the alternate printer **setprinter** is used again to initialize the main printer and to make the main printer the default printer.

KillPrint (-)
(pronounced kil'print)
Stops the printer spooler. If the print buffer is not empty, **quit.print** stops printing and **UnPrint** is set as the undo operation.

makeprinttable (-)
(pronounced mayk' print tay'bl)
If a printer has a non-standard print table (the BJ80 with a second character set, for example), or a country-specific print table (any of the daisy wheel printers), **makeprinttable** creates a RAM image of the print table in the **trkbuf** and patches it with an exception table. **patchprint** patches the RAM image. The **trkbuf** address is stored in the **printertable** integer.

makeprinttable does nothing if a printer with a standard print table is being used.

patchprint (a -)
(pronounced patch' print)
Used to patch non-standard print tables. The patch data at the address a patches the print table located at the address contained in the **printertable** system integer.

pickprinter (-)
(pronounced pik' print'er)
Used to switch the print spooler to the correct printing port. If there is a selection, and if the **printercode** and **printerport** system integers indicate that a valid printer port has been selected, **pickprinter** will direct the print spooler to either the serial port (with **print.serial**) or the parallel port (with **print.parallel**).

Print (-)
(pronounced print')
Word executed when [Use Front]-[Print] is pressed. Causes the current text selection, if any, to be printed out on the main printer. Uses **pickprinter** to set up the print spooler, **makeprinttable** to make a patched RAM print table image if a printer with a non-standard print table is being used, and uses **<Print>** to print the selection.

<Print> (-)
(pronounced brak'it print')
This is the highest level print word, aside from the **Print** command itself. Uses **printify** to turn on the "PRINT" sign and checks for a printable selection. If there is nothing to print, **KillPrint** terminates the command. Otherwise, **bos nextchar findchar** sets the control variables for the printable selection, **initprinter** performs printer initialization, and **<<Print>>** performs the bulk of the print operations. **<Print>** is also responsible for turning the indicator light off after printing has finished.

<<Print>> (-)
(pronounced brak'it brak'it print)
Performs the bulk of the printing activities. This word was factored out of **<Print>** so that **UnPanicPrint** could be used to restart printing. **<<Print>>** sets **undop** to zero (no undo operation) and causes the top of the selection to be displayed. The control variables are prepared and **printline** is called specially to print the first line in the selection (to handle any

initial page breaks). Then **printline** is called in a loop until there are no more lines of text to print.

When the **printline** loop is completed, **<<Print>>** checks to see if printing was prematurely terminated. If it was terminated prematurely, the first unprinted character is saved in the **op** and then the selection is reduced to a cursor at the end of the selection. If printing was not terminated prematurely, the end of the original selection is just displayed on the screen. If necessary, **formfeed** ejects a partial page.

setprinter (f -)
(pronounced set'print'er)
Sets the desired printer, f=0 for the main printer and f=1 for the alternate printer, as the current printer and performs printer preparation activities for that printer. **setprinter** initializes all printer integers and strings which contain values shared by the majority of the printers and then, using the printer code for the chosen printer as an index into the **printers** table, obtains and executes the token corresponding to the word which performs printer-specific initialization for the chosen printer.

UnPrint (-)
(pronounced un'print)
Undoes the stopping of the print spooler. Uses **restore.print** to reset the print spooler and sets **KillPrint** as the undo operation.

21.7.9 Printing Initialization Words

ap100setup (-)
(pronounced ay'pee wun hun'dred set'up)
Performs AP100 printer setup procedures.

ap300setup (-)
(pronounced ay'pee three hun'dred set'up)
Performs AP300 printer setup procedures.

ap400setup (-)
(pronounced ay'pee for hun'dred set'up)
Performs AP400 printer setup procedures.

apsetup (n -)
(pronounced ay'pee set'up)
Performs the printer setup procedures which are common to the AP300 and AP400 printers. The printer-specific steps/line parameter, n, is passed on the stack.

bj80docbreak (-)
(pronounced bee'jay ay'tee dahk brayk)
Routine which sets the page length for the BJ80 printer.

bj80setup (-)
(pronounced bee'jay ay'tee set'up)
Performs BJ80 printer setup procedures.

cat180setup (-)
(pronounced kat wun-ay'tee set'up)
Performs Cat180 printer setup procedures. Sets the **paperpos** integer to 2 (top-of-form position).

CATdocbreak (-)
(pronounced kat'doc'brayk)
Routine which sets the paper length for the Cat180 printer.

daisymagic (a c -)
(pronounced day'zee ma'jik)
Handles weird print for daisy wheel printers.

ETWdocbreak (-)
(pronounced ee-tee du'bl-yu dahk' brayk)
Routine that sets the paper length for the 12-inch ETW typewriters (AP100, AP300 and AP400).

fx80magic (a c -)
(pronounced eff'eks ay'tee ma'jik)
Handles fancy font switches for the FX80 printer.

fx80setup (-)
(pronounced eff'eks ay'tee set'up)
Performs FX80 printer setup procedures.

lbp8setup (-)
(pronounced ell'bee-pee ayt set'up)
Performs LBP printer setup procedures.

LBPdocbreak (-)
(pronounced ell'bee-pee dahk' brayk)
Routine that sets the paper size for the LBP printer.

LBPmagic (a c -)
(pronounced ell'bee-pee ma'jik)
Handles the printing of unusual characters on the Laser Beam printer. Switches to the country whose character set contains the unusual character, prints the character, and then switches back to the country character set previously being used.

newapsetup (-)
(pronounced noo ay'pee set'up)
Performs Cat180 printer setup procedures.

noprintersetup (-)
(pronounced no print'er set'up)
Does nothing.

setcountry (n -)
(pronounced set'kun'tree)
Sets the Laser Beam Printer to a country character set. The country character code n is obtained from the **countries** data table.

21.7.10 Setup Export Words

These words are part of the Setup command code. They communicate printer settings from the Setup command to the Print command.

printercode (- n)
(pronounced print'er kohd)
Returns a code indicating which printer is currently in-use on the current printer port.

printercode	Printer Name	Printer Type
0	Cat180	Daisy wheel
1	LBP8	Laser Beam
2	NewAP	Daisy wheel
3	AP400	Daisy wheel
4	AP300	Daisy wheel
5	AP100	Daisy wheel
6	BJ80	Dot matrix
7	FX80	Dot matrix
8	No printer	N/A

printerinfo (n1 - n2)
(pronounced print'er in'fo)
Returns information about the printer currently in use. The code passed in, n1, indicates what information is desired. Eight possible input codes are recogized. The input code, and the information associated with the code, are listed in the table below:

Input Code	Data Returned
0	If a daisy wheel printer is in use, returns a code indicating which wheel is being used. If a daisy wheel printer is not being used, a flag indicating whether the underlined code means underline or italicize is returned.
1	If a Laser Beam printer is in use, returns a code indicating which Laser Beam font is being used (Courier, Gothic, Pica, Elite).
2	Returns a code which indicates the pitch of the font currently in use. 0: 10-pitch 1: 12-pitch 2: 15-pitch
3	Returns the current left margin offset in characters.
4	Returns a flag which indicates whether bidirectional printing is being used.
5	Returns a true flag if sheet feeding is being used.
6	Returns a code which indicates which paper tray is being used. 0: A 1: B 2: A for first page, B afterward
7	Returns a flag which indicates whether single-page printing is being used.

printerport (- n)
(pronounced print'er port)
Returns a code which indicates which printer port is currently in use: -1 = parallel; 0 = serial; 1 = no printer port in use.

21.7.11 Print Spooling Export Words

These words are provided by the low-level spooler code. They send characters to the printer and control the spooling process.

print.buf.free (- f)
(pronounced print' dot buff' dot free)
Returns the number of free bytes in the print buffer.

print.empty (- f)
(pronounced print' dot em'tee)
Returns a true flag if the print buffer is empty.

print.parallel (-)
(pronounced print' dot pair'al-lel)
Switches spooler output to the parallel port.

print.serial (-)
(pronounced print' dot seer'ee-il)
Switches spooler output to the serial port.

<printc> (c -)
(pronounced brak'it print see)
Places a character in the print buffer.

quit.print (-)
(pronounced kwit' dot print)
Stops printing in a manner that allows it to be resumed if necessary.

restore.print (-)
(pronounced ree-stor' dot print)
Restores the low level printing state so that printing may be resumed after a **quit.print**.

backspace" (pronounced bak' spays kwote)
Instructs the printer to perform a backspace.

-bold" (pronounced dash' bohld kwote)
Contains the commands which instruct the printer to stop boldfacing all subsequent characters.

+bold" (pronounced plus' bohld kwote)
Contains the commands which instruct the printer to boldface all subsequent characters.

endline" (- a n)
(pronounced end' lyne kwote)
String sent when the printer has reached the end of a line. This string will always contain a linefeed character and, with some printers, will additionly contain a carriage return.

endprint" (- a n)
(pronounced end' print kwote)
String sent when a print job is completed. Tells a printer eject the current page without feeding in another page.

evenhalfspace" (pronounced ee'vin haff spays kwote)
Tells the printer to move forward one half space from its even half space position.

halfline" (- a n)
(pronounced haff' lyne kwote)
String which commands the printer to move the paper up one half-line.

hmi" (pronounced aytch' em eye kwote)
Used as lead-in for setting the pitch.

initprint" (- a n)
(pronounced in-it' print kwote)
String used to initialize the printer. This string is constructed for the main printer when the Setup command exits or when the system powers on. The individual printer printer setup words help construct the string. **initprint**, which is called by **<Print>**, is responsible for sending the initialization string to the printer.

leftfoot" (- a n)
(pronounced left' fut kwote)
String which contains the text for a footer to be placed to the left of the page number. If this string has a length of zero it will not be printed.

leftfrill" (- a n)
(pronounced left' frill kwote)
String which contains the "frill" mark to be placed to the left of the page number. The string is initially 2 characters long and contains a minus sign followed by a space, "- ".

oddhalfspace" (pronounced ahdd half' spays kwote)
Tells the printer to move forward one half space from its odd half space position.

overstrike" (pronounced oh'ver-stryke kwote)
Tells the printer to print the next character in the string without moving the carriage forward.

printforward" (pronounced print for'wurd kwote)
Tells the printer to print from left to right (forward).

printreverse" (pronounced print ree-vers' kwote)
Tells the printer to print from right to left (in reverse). Only used when a printer which can print bidirectionally is in use.

rightfoot" (- a n)
(pronounced ryte' foot kwote)
String which contains the text for a footer to be placed to the right of the page number. If this string has a length of zero, it will not be printed.

rightfrill" (- a n)
(pronounced ryte' frill kwote)
String which contains the "frill" mark to be placed to the right of the page number. The string is initially two characters long and contains a space followed by a minus sign, " -".

startline" (- a n)
(pronounced start' lyne kwote)
String sent when the printer should start printing a new line of text. For those printers which do not automatically perform a carriage return when they receive a linefeed (in the **endline"** string), the **startline"** string will contain a carriage return so that the printer starts printing on the correct line.

topofform" (- a n)
(pronounced tahp-uv-form' kwote)
String which contains the commands which tell a printer to eject the current page and feed in another.

+underline" (pronounced plus un'dur-lyne kwote)
Contains the commands which instruct the printer to underline all subsequent characters.

-underline" (pronounced my'nis un'dur-lyne kwote)
Contains the commands which instruct the printer to stop underlining all subsequent characters.

unoverstrike" (pronounced un-oh'vur-stryke kwote)
Tells the printer to move forward after it prints the next character.

userinit" (- a n)
(pronounced yu'sir in-it' kwote)
User-specific printer initialization string. Sent to the printer right after the **initprint"** is sent. This string is never touched by the setup commands.

21.9 PRINTER INTEGERS

21.9.0 Printer "Knowledge" Integers

boustrophedon (pronounced boo-stref'a-don)
Manual bidirectional printing.

braindamaged (pronounced brayn' dam'ijd)
Flags printers that can't reverse directions or print bidirectionally.

knowstof? (pronounced nose' tee uv kwes'chun)
Is the printer aware of form feeds?

knowsbold (pronounced nose' bohld)
Can the printer boldface?

knowsul? (pronounced nose' un'dur-lyne kwes'chun)
Can the printer underline?

knowsos? (pronounced nose' o'vur-stryke kwes'chun)
Does the printer prefer overstrike to backspace?

knowshmi? (pronounced nose' aytch' em eye kwes'chun)
Does the printer use a Diablo-like HMI setting?

ulinehack? (pronounced un'dur-lyne hak kwes-chun)
Translate underlined white space to underline characters.

21.9.1 Page Logistics Integers

char/inch (pronounced kair' slash inch')
Print pitch

footpos (pronounced fut'paws)
Offset from the bottom-of-page position to the line that holds the page number

gutter (pronounced gut'ter)
Left margin offset in half-characters

oldcountry (pronounced ohld' kun-tree)
Holds the primary country code for the Laser Beam printer

paperpos (pronounced pay'per paws)
Location where the paper is in the top-of-form position

papershort (pronounced pay'per short)
Number of lines missing from the page

rightstop (pronounced ryte' stahp)
Holds the right-carriage stop information for the Cat180 printer

steps/inch (pronounced steps' slash inch')
Granularity of the HMI setting

steps/line (pronounced steps' slash lyne')
Printer steps per line feed. Set only by typewriters

21.9.2 Print State Integers

backwards (pronounced bak'wurds)
Currently printing backwards.

bolded (pronounced bohld'ed)
Currently bolding.

oddhalfspace (pronounced ahd-haff'spays)
True if the next half-space is odd. Set to false when finished with a line.

pageprint (pronounced payj'print)
Single page printing flag.

prcol (pronounced print' kahl)
Current print column. **prcol** = 0 corresponds to ruler column zero.

prline (pronounced print' lyne)
Half-line on the current page.

stopprint (pronounced stop' print)
Switch used for an early escape from **Print**. Used either during the single-page mode or when the user issues a panic print-stop.

underlined (pronounced un'dur lynde)
Currently underlining.

21.9.3 Unbuild Integers

prchar (pronounced print' kair)
Holds either the current print code (number from $000-$109), or an unknown 2-byte value from the printer table.

printed? (pronounced print'ed kwes'chun)
Is this character part of the selection? That is, is the invert bit in the **lbuff** information for this character on? Only inverted characters are printed.

proldflags (pronounced print' ohld flags)
lbuff flags byte for the previously printed character.

prsmall? (pronounced print' smahl kwes'chun)
Is this a half-character?

prwhite? (pronounced print whyte kwes'chun)
Is this character white?

pr\bold/ (pronounced print' bak'slash bohld slash)
Is this a bold/normal transition?

pr\uline/ (pronounced print' bak'slash un'dur-lyne slash)
Is this an underlining transition?

stopprint (pronounced stahp' print)
Switch used for an early escape from **Print**. Used either during single-page mode or for a panic print-stop.

21.9.4 Printing Integers

printlimit (pronounced print' lim'it)
Address of the last **lbuff** character to print. Set up by **startline**

printnext (pronounced print' nekst)
Address of the next character to print in **lbuff**. This address is set up by **startline** and bumped either forward or backward by **unbuild**. **printnext** will be bumped forward if forward printing is used; it is bumped backward otherwise.

21.9.5 Character Selection Integers

printertable (pronounced print'er tay'bl)
Holds the address of the printer table for the printer currently in use.

printnext (pronounced print' nekst)
Address of the next character to print in **lbuff**. This address is set up by **startline** and bumped either forward or backward by **unbuild**. **printnext** will be bumped forward if forward printing is used or backward otherwise.

unbuildtable (pronounced un'bild tay'bl)
Contains the address of the **vanilla.unbuild** table. This table translates from **lbuff** character code to a printer table character code.

21.9.6 Printer Execution Vectors

'docbreak (pronounced a-pos'tra-fee dahk' brayk)
Holds execution vector for printing document breaks. This vector is initialized with the token of **noop**.

'weirdprint (pronounced a-pos'tra-fee weerd' print)
Holds execution vector which handles printer table values from $00xx - $1Fxx. This vector is initialized with the token for **printerror**.

21.9.7 Setup Integer

whichprinter (pronounced witch' print'er)
Printer usage flag: 0=main printer, 1=alternate printer. The printer setup export words, especially **printerinfo**, check this integer to determine how they should function.

21.9.8 Print Integers (Constant)

printsize (pronounced print'syze)
Size of a print table.

22. SETUP

Introduction

The Setup command adjusts settings for document parameters and connections to the Cat. The command operates on a data vector which contains the current settings. This vector is saved in the battery-backed-up RAM so that the user's settings are preserved across power off.

GENERAL OPERATION

When the user presses the SETUP command the word **Setup** is called by the edde interpreter. The first part of **Setup** checks and initializes various things and then the main setup begin loop is entered. The main loop is a state machine that picks parameters to use and the next state to execute from the **groups** array. This loop never finishes as the SETUP command exits whenever the user releases the USE FRONT key and this may occur at any time during parameter entry. The release of either USE FRONT key is detected right at the begining of the word **scode** which then calls **exitsetup**, which sets up the Cat via **setupcat** and returns to the user.

Setup Data Structures

Setup has three data structures, two vectors and a two dimensional array. The vectors are a matched pair with one holding executable tokens and the other the setup data. The array governs the logic flow, screen display lines and indicies of which series of tokens to execute in the token vector.

The Token and Data Vectors

Setup has two matched vectors of n two byte elements each. The number of elements in each vector is set by the target compiler integer **setv&tlim**. Its value depends on the Cat version software but is about 144. One vector, called **<settokens>**, is in ROM and the other, called **setdata**, is in RAM. Each screen line of information in the setup command has a forth word associated with it and each word's token is in the token vector in ROM. The RAM vector is matched to **<settokens>** and holds the corresponding data for each setup line. Where a setup screen line is a display only line, **setdata** has only filler data in that element. When the user interacts with an executing setup display word, the data generated goes into the matching location in **setdata**.

The Groups Array and Logic Flow in Setup

The logic flow in setup is controlled by data in a 5 column by n row byte array called the **groups** array. The exact number of rows depends on the Cat version software but is about 18. The rows in the **groups** array contain two kinds of information: what setup group information to display where and which setup group to go to next. The display information consists of the first and last indexes into the matched token/data vectors and the first screen display line to use for each group of setup information. The information on which setup to go to next governs the general logic flow of setup. Most of the information in **groups** is fixed at compile time but part of it is set during setup execution depending on what the user selects.

Setup Data Initialization

During power-up initialization, the information in the data structures is loaded from battery RAM if the information is intact and if not, the structures are initialized from ROM. After each user use of setup, the battery RAM is updated and verified. The setup data that are saved on disk with every DISK command may be loaded from a disk if the user requests to do so via the items on the second setup screen.

Displayed Screen Data

The forth words whose tokens are in **<settokens>** are executed in two modes governed by a flag called **cflag**. If **cflag** is off (zero) the words only display their information and if **cflag** is on (non-zero) the words display their information in bold font and can interact with the user to obtain data. Not surprisingly, most of the words in setup examine or manipulate **cflag**.

Printer Selection

Each line of the setup screens has a forth word associated with it, including each one of the eight supported printers. Each of the eight printers puts data in its part of the **setdata** vector and the chosen printer's data is moved from **setdata** to a working vector by **setupcat** when setup is left. Because there is only one working vector, all of the printer choices must accept and store data in a uniform way. This uniformity is shown (DaveA) below as a table of printer and data value types.

Condensed Printer Setup Groups

The following columns show the way the setup groups for the eight supported printers order the data that they obtain from the user and use to set the printers via the printing word **printerinfo**. Each printer uses a subset of the eight data elements assigned to it in **setdata** and **<settokens>** by the target compiler word **setv&ti**. The unused elements are filled by **setv&ti** with the 4 character string **none** and the token for the Forth no-operation word **noop**.

Each of the names of the following words is constructed by a leading 'm' and then a nemonic for the printer such as 'bjp' for the BubbleJet Printer and a nemonic for the element name such as 'dw' for daisy wheel. The ninth printer choice, 'No printer', is not shown below as it has all of its elements set to **none** and **noop**.

The first element is the Daisy Wheel, language or character set. The second element is the LBP's character font. The third element is the character pitch. The fourth element is the left margin offset (gutter in printer talk). The fifth element is direction for most printers and the portrate/landscape mode for the LBP (Cat versions 2.00 and later). The sixth element is the cut sheet feeder. The seventh element is the tray selection for

the larger typewriters. The eight element is the 'pause between sheets' selection to allow hand feeding of the non-laser printers.

The word that is set apart in each column is the word for each printer that stacks the data and token index and turns on the printer existance flag **aprinter**. The 'No Printer' word, NON, stacks the no printer indicies and turns off **aprinter** via a seperate word, **aptroff**.

```
( Cat180 printer)   ( LBP printer)      ( New AP series)    ( AP400 series)
: m18dw             : mlbpl             : mnewapdw          : map4dw
none setv&t noop    : mlbpcf            none setv&t noop    none setv&t noop
: m18pitch          : mlbppitch         : mnewappitch       : map4pitch
: m18g              : mlbpg             : mnewapg           : map4g
: m18d              : mlbpp/1           : mnewapd           : map4d
: m18csf            none setv&t noop    : mnewapcsf         : map4csf
none setv&t noop    none setv&t noop    : mnewaptray        : map4tray
: m18pbs            none setv&t noop    : mnewappbs         : map4pbs

: CAT180            : LBP               : newAP             : AP4


( AP300 series)     ( AP100 series)     ( BJ printer)       ( Common printer)
: map3dw            : map100dw          : mbjpcs            : mfxpl
none setv&t noop    none setv&t noop    none setv&t noop    none setv&t noop
: map3pitch         : map100pitch       : mbjppitch         : mfxpitch
: map3g             : map100g           : mbjpg             : mfxg
: map3d             none setv&t noop    : mbjpd             : mfxd
: map3csf           none setv&t noop    none setv&t noop    none setv&t noop
: map3tray          none setv&t noop    none setv&t noop    none setv&t noop
: map3pbs           : map100pbs         : mbjppbs           : mfxpbs

: AP3               : AP100             : BJP               : FX80
```

Setup Target Compiler Integers and Support

Note: Some target compiler integers, arrays and words have the same name as source code words but they are distinct. Target compiler names are executed when the target compiler is not compiling (including between square brackets like [...]) and source code names are compiled when actually target compiling the source code.

Setup Integers, ROM Arrays and Pointers

<leapsc (pronounced less' leep ess see)
The left Leap key scan code.

AP100sop (pronounced ay'pee wun hun'dred ess-oh-pee)
The code number for the AP100 serial printer. The AP100 is a serial interface device only.

Cat180pop (pronounced cat' wun-ay'tee pee-oh-pee)
The code number for the Cat180 parallel printer. The Cat180 is a parallel interface device only.

cclim (pronounced see-see' lim)
The numerical limit on the number of country codes.

ccwidth (pronounced see-see' width)
The width of the country setup information array.

gpwidth (pronounced jee-pee' width)
The width of the groups array.

groupi (pronounced groop' eye)
The index of the group currently on the display.

groupl (pronounced groop' ell)
Holds the number of lines in the group currently on the display.

grouplim (pronounced groop' lim)
The limit on the number of display groups.

himsetuplim (pronounced aytch'eye-em set'up lim)
Hidden internal modem setup size.

leap>sc (pronounced leep grayt'er ess-see)
Right leap key scan code.

main/altlim (pronounced mayne slash alt'lim)
The main or alternate printer vector limit.

setupsc (pronounced set'up ess'see)
Setup command key scan code.

setv&ti (pronounced set' vee' and tee' eye)
The value and token index number.

setv&tlim (pronounced set vee' and tee' lim)
The value and token index limit.

shi (pronounced ess' aytch' eye)
Uppermost serial token and data index for the current group.

slo (pronounced ess' ell' oh)
Lowermost serial token and data index for the current group.

spacesc (pronounced spays' ess see)
Space bar scan code.

starti (pronounced start' eye)
Holds the starting index number for this group.

VP3103II (pronounced vee'pee thir'tee wun' oh three too)
The code number for the Laser Beam printer.

<groups> (pronounced brak'it groops)
The ROM groups array for initialization of the corresponding RAM array.

<himsetup> (pronounced brak'it aytch'eye-em setup)
The hidden internal modem setup ROM address.

<setdata> (pronounced brak'it set' day-ta)
A ROM array holding the default setup state.

<settokens> (pronounced brak'it set toh'kins)
A ROM array holding the setup token list.

The following target compiler integers are pointers and work with the word in the source cord of the same name without the leading 'p'. These pointers are set by the index stacked by **setv&t** when compiling most of the setup display words. The pointers enable words compiled later to automatically find the appropriate data in the **setdata** vector. For this reason, the setup code must be compiled before any words are compiled that use these pointers to access setup data.

p#punct p#sortb pap papp pbotmgn pdbotmgn pdecimals pdfirstpage# pdisplay pdpagelen pdprintpage# pdpx pdtopmgn pemcfc pemct pempro pemra pemring pems pemspkr perror pfirstpage# pimbpw pimcfc pimct pimpro pimpty pimra pims pimsb pkeyboard pmp pmpcon ppagelen pprintpage# ppro pring pspcon pssetup psspell ptab ptimeout ptlt ptopmgn ptyper

The following Target Compiler integers label data that gets saved in the svram. The data must be an even number of bytes long.

#defaults (pronounced sharp' dee-falts')
Default format information.

checkspell (pronounced chek' spell)
A flag which is true if there is a spelling checker.

externalmodem (pronounced eks-ter'nul mo'dem)
A flag which is true if there is an external modem.

groups (pronounced groops')
The setup groups array.

himsetup (pronounced aytch'eye-em set'up)
Hidden internal modem setup.

kbdI/II (pronounced kay'bee-dee wun' slash two')
Keyboard I/II indicator flag.

setdata (pronounced set' day'ta)
Start of the setup svram area.

spareflg (pronounced spair eff'ell-jee)
Filler to make the number of bytes even.

svid (pronounced ess'vee-eye-dee)
Holds the svram version number.

svsetupaltptr (pronounced ess'vee set'up alt pee'tee-arr)
The alternate printer direction flag.

svsetupflg (pronounced ess'vee set'up eff'ell-jee)
A flag that shows that Setup has been loaded from ROM.

svsetupgutters (pronounced ess'vee set'up gut'ters)
Setup printer gutters (left margin offset) for three pitches.

svsetupmainptr (pronounced ess'vee set'up main pee'tee-arr)
The main printer direction flag.

svsetupscmnd (pronounced ess'vee set'up ess com-mand')
A flag that holds the direction for the Send command.

svspare (pronounced ess'vee spair)
A filler to make sure that the number of bytes is even.

svspellcs (pronounced ess'vee spell see-ess')
RAM spelling dictionary checksum. This and the following checksum must come last.

svsetupcs (pronounced ess'vee set'up see-ess')
Setup svram data checksum. This and the preceding checksum must come last.

Setup Command Ordinary RAM Vectors

altp (pronounced alt' pee)
The alternate printer data vector.

idocpkt (pronounced eye' dok pak'it)
The initial document packet.

mainp (pronounced mayne' pee)
The main printer data vector.

oldset (pronounced ohld' set)
Holds the old setup state.

Setup Command Target Compiler Support

The following words are defined in the target piler and used during the compilation of the actual Cat software.

~ (->)
(pronounced till'da)
The setup string compiling word. It allows strings with quotes in them, such as ~ **Oh, "blah"}**. Not used in 2.00 and higher versions of the target compiler. Instead, **makemsg** is used because it supports multiple languages.

startgroup (->)
(pronounced start' groop)
Starts a display group. Holds the first index number.

setgroup (bg sg dl l u ->)
(pronounced set' groop)
Loads the group array row with the space bar group jump number (bg), the setup group jump number (sg), the nominal display line number (dl) and the lower and upper index range (l and u). Increments the group index.

makegroup (bg sg dl ->)
(pronounced mayke' groop)
Loads a display group's data. Automatically stores the group index range in the group array row and then stores the space bar group jump number (bg), the setup group jump number (sg), and the nominal display line number (dl), from the stack.

setv&t (n -> i)
(pronounced set vee' and tee')
Puts the token of the following word in the next available place in **settokens**. Also sets the corresponding **setdata** value to the parameter n. Stacks the vector data index so that it may be loaded into a pointer or discarded.

SETUP ARRAYS AND INTEGERS

Setup Command ROM Arrays

<groups> (pronounced brak'it groops)
Holds the default groups information.

<himsetup> (pronounced brak'it aytch'eye-em set'up)
Holds the hidden internal modem setup default information.

<setdata> (pronounced brak'it set'day-ta)
Holds the default setup state.

<settokens> (pronounced brak'it set'toh-kins)
Holds the default setup token list.

Setup Command "Zero" Integers

#autos (pronounced sharp' aw'tohs)
The number of autorepeats that must occur before changing the rate at which the page numbers are changed on the display.

aprinter (pronounced ay print'er)
A flag which indicates whether a printer is attached.

atrib (pronounced ah-trib')
A place to assemble display attributes for the displayed characters.

cflag (pronounced see' flag)
A flag which turns on when it is the user's turn to make a choice in the Setup menu.

choicelimit (pronounced choys' lim-it)
The limit on the number of choices for a particular item on the Setup menu.

delta# (pronounced del'ta sharp)
The rate change to be applied to changing page numbers when the number of repeats equal **#autos**.

exitsc (pronounced eks'it ess-see)
Holds the scancode of the exit key so the group display can examine it.

group# (pronounced groop' sharp)
Holds the current group number.

groupstartl# (pronounced groop' start ell-sharp)
The line number to start displaying this group.

hlabove (pronounced aytch'ell ah-buv')
The number of half-lines for the top margin.

hlbelow (pronounced aytch'ell below)
The number of half-lines for the bottom margin.

hllong (pronounced aytch'ell long)
The number of half-lines for a page.

ipage# (pronounced eye' payj sharp)
The initial page number.

iprint# (pronounced eye' print sharp)
The initial printing page number.

maxgpline (pronounced macks' jee-pee lyne)
The maximum number of group lines for display line erasing.

mingpline (pronounced min' jee-pee lyne)
The minimum number of group lines for display line erasing.

numbr (pronounced num'ber)
Holds the number currently being worked on.

oldend (pronounced ohld' end)
Holds the end of the old display string for erasing the leftover portion in case the new display string is shorter.

printer (pronounced print'er)
The anothergroup number of the main printer.

Setup Command Integers

#halflines (pronounced sharp haff' lynes)
The number of half-lines for each paper size.

choicex (pronounced choyss' eks)
The vertical position at which the setup choices will be displayed.

imdm (pronounced eye' em dee' em)
The internal modem flag.

modm (pronounced em oh dee em)
The modem flag.

none (pronounced nun)
The "not-connected" flag. Means that nothing is connected to the serial channel being considered.

pprt (pronounced pee' prt)
Parallel port flag.

sprt (pronounced ess prt)
Serial port flag.

xmdm (pronounced eks' em dee' em)
External modem flag.

THE DEFAULT COUNTRY SETUP DATA

defcountry (-> addr)
(pronounced deff' kun-try)

The country code default array, an array of 17 rows by 16 bytes containing country code default information. Each row consists of 10 bytes of bit information for tabs and then 1 byte each for the codes for: external modem, paper size, top margin, bottom margin, number punctuation and one byte for the spelling checker and keyboard I/II flags.

Note: The tab position information is in the bit positions of a data area **tabcount** bytes long in data vectors such as **#defaults** and **##ctrl** which are used by tab words like **Tabs**, **Deftabs** and **tabloop**. The offset of the tab data area of these arrays is given by **%tabs**. The tab positions, starting at text column 1, start at bit 0 of the first byte in the tab data area and go up to bit 7 of that byte and then to bit 0 of the byte at the next higher address and so on. For example, if you execute:

```
#defaults %tabs + c@ . 21
```

and, since bits 0 and 5 of the first tab data byte are set, the first two tab positions would be at text columns 1 and 6. The first 10 bytes of the tab data area define the normal tab settings for all 80 columns and the second 10 bytes define the positions of the decimal tabs for the corresponding 80 columns.

#defaults (-> addr)
(pronounced sharp' dee-falts)
Gets the address of the format default data in the setup system area.

<bonw> (->)
(pronounced brak'it bee ahn du'bl-yu)
Sets the text to black-on-white, and the ruler to black.

<choose#> (a l i g lo hi ->)
(pronounced brak'it chooz' sharp)
Common code for choosing numbers.

<mbmargin> (i g ->)
(pronounced brak'it em-bee mar'jin)
Common code for the bottom margin choice.

<mpagelen> (i g ->)
(pronounced brak'it em payj len)
Common code for page length choice, and for data index i on line g

<mtmargin> (i g ->)
(pronounced brak'it em tee mar'jin)
Common code for the top margin choice.

<setline> (n ->)
(pronounced brak'it set' lyne)
Sets the absolute vertical position of the display line to screen line n, and the horizontal position of the beginning of the line to the leftmost column, puts blank characters in the entire display buffer (**lbuff**), and remembers the maximum display line number for later screen clearing.

<wonb> (->)
(pronounced brak'it du'bl-yu ahn bee)
Sets the text to white-on-black, and the ruler to black.

2nybs (# a -> a+2)
(pronounced too' nibs)
Makes the second stack item (#) into two nibbles and places it at address a, then updates a.

3nybs (# a -> a+3)
(pronounced three nibs)
Makes the second stack item (#) into three nibbles and places it at address a, then updates a.

16bitsignex (n -> n')
(pronounced siks-teen' bit syne' ee-eks)
Extends the sign of a 16-bit 2's complement number to 32 bits.

?0 (n -> n-1 f)
(pronounced kwes'chun zee'ro)
Subtracts 1 from n, returns the result and a false flag if the result is greater than zero, or returns only a true flag if the result is zero or less.

altptr (-> addr)
(pronounced alt' pee-tee-arr)
Gets the alternate printer direction address in the setup system area.

AP3 (-> l u)
(pronounced ay pee three')
Stacks the AP300's lower and upper group index range.

AP4 (-> l u)
(pronounced ay pee for')
Stacks the AP400's lower and upper group index range.

AP100 (-> l u)
(pronounced ay pee wun' hun'dred)
Stacks the AP100's lower and upper group index range.

aptroff (->)
(pronounced ay-pee-tee arr' off)
Replaceable word to turn the aptr flag off.

BJP (-> l u)
(pronounced bee jay pee)
Stacks the BubbleJet's lower and upper group index range.

boldtolbuf (a l y x ->)
(pronounced bold too ell buff)
Outputs the l character string at a, starting at the given x and y screen position, and makes it bold if the user has selected it with the Space Bar.

bonw (->)
(pronounced bee ahn du'bl-yu)
Sets the text area to black-on-white, the ruler to black and turns the black screen flag off.

buildnumber (->)
(pronounced bild' num-ber)
Builds up a number such as page number from manipulation of the Leap key.

CAT180 (-> l u)
(pronounced cat' wun ay'tee)
Stacks the Cat180's lower and upper group index range.

checknumber (l u ->)
(pronounced chek' num-ber)
If the number being built up isn't within the range l to u, roll it to the upper or lower limit, whichever is appropriate.

checkspell! (n ->)
(pronounced chek' spell store)
Stores n in the spelling checker flag in Setup's RAM.

checkspell@ (n ->)
(pronounced chek spell fetch)
Fetches n from the spelling checker flag in Setup's RAM.

choicecode (n ->)
(pronounced choyss' kode)
Gets a scan code, and, if it is a leap code, adjusts the nth choice number on the line being selected, rolling the number to the upper or lower limit when necessary. It saves the scan code if it is a space or setup code, otherwise it throws it away.

choicedisp (a l ->)
(pronounced choyss' disp)
Displays the l character string on current line and makes it bold if it is to be a choice.

clearlines (f l ->)
(pronounced cleer' lynes)
Clears lines f to l, making them ASCII blanks.

clippage#to (-> l u)
(pronounced clip' page sharp' to)
Stacks l and u, the two page-number clipping constants.

Defsetup (->)
(pronounced deff' set-up)
Installs the default setup into system RAM and svram.

dispbjgutter (ln i ->)
(pronounced disp' bee jay gut'ter)
For a given selected pitch (on BubbleJet printer), displays the left margin offset on line ln.

dispcomgutter (ln i ->)
(pronounced disp com gut'ter)
For a given selected pitch (on the FX80 printer), displays the left margin offset on line ln.

dispgutter (ln i ->)
(pronounced disp gut'ter)
For a given selected pitch (for various printers), displays the left margin offset on line ln.

displaygroup (l u ->)
(pronounced dis-play' groop)
Executes token indexes l to u, displaying the information related to the token.

exitsetup (->)
(pronounced eks'it set-up)
Exits out of setup, checks I/O assignments, and sets up the Cat.

FX80 (-> l u)
(pronounced eff' eks ay'tee)
Stacks the FX80's lower and upper group index range.

getdata (k -> addr)
(pronounced get' day-ta)
Get the k th item's address in **setdata** in the setup system area.

getsetupspell (sef spf ->)
(pronounced get set-up spell)
Recovers the setup and spell information from the disk if the appropriate flag is true.

gpdaddr (o g -> addr)
(pronounced jee pee dee ad'der)
Calculates the group g, offset o address in the groups array in the setup system area.

gutters (-> addr)
(pronounced gut'ters)
Calculates the gutter data address in the setup system area.

himsetup (k -> addr)
(pronounced aytch' eye em set'up)
Calculates the k th item's **himsetup** data address in the setup system area.

initsetup (->)
(pronounced in-it' set-up)
Initializes setup from ROM, sets directions and flags, and calculates data checksum.

initsvram (->)
(pronounced in-it' ess vee ram)
Tests svram data, and initializes from the ROM if necessary.

kbdcountry (-> cc)
(pronounced kay bee dee kun'try)
Stacks the decoded country code.

kbdI/II! (n ->)
(pronounced kay-bee-dee' wun-slash-too' stor)
Stores n in the Keyboard I/II flag in Setup's RAM area.

kbdI/II@ (-> n)
(pronounced kay-bee-dee' wun-slash-too' fetch)
Fetches n from the Keyboard I/II flag in Setup's RAM area.

LBP (-> l u)
(pronounced ell bee pee')
Stacks the LBP's lower and upper group index range.

m#punct (->)
(pronounced em' sharp punct')
Selects the way numbers are punctuated.

m#sortb (->)
(pronounced em' sharp sort bee')
Selects the number of sort breaks.

m18csf (->)
(pronounced em ay-teen' see ess eff)
Selects the new AP printer's cut-sheet feeder option.

m18d (->)
(pronounced em ay-teen' dee)
Selects the Cat180's printing direction.

m18dw (->)
(pronounced em ay-teen' dee' du'bl-yu)
Selects the Cat180's daisy wheel.

m18g (->)
(pronounced em ay-teen' jee')
Selects the LBP's left margin offset (gutter).

m18pbs (->)
(pronounced em ay-teen' pee bee ess)
Selects Cat180's pause-between-sheets.

m18pitch (->)
(pronounced em ay-teen' pitch)
Selects the Cat180's pitch.

mAB (i g ->)
(pronounced em ay bee)
Sets the tray for printer i, and displays the tray selection on line g.

mainptr (-> addr)
(pronounced mayne pee tee arr)
Gets the main printer direction address in the setup system area.

manswer (i ->)
(pronounced em' an'ser)
Displays the number of rings before autoanswer option for index i.

map (->)
(pronounced em-ay-pee')
Interacts with the user to select an alternate printer. Doesn't present the Cat 180 parallel printer. Alters the groups to go to depending on printer selection.

map3csf (->)
(pronounced em-ay-pee three see ess eff)
Selects the AP300's cut-sheet feeder.

map3d (->)
(pronounced em-ay-pee three dee)
Selects the AP300's print direction.

map3dw (->)
(pronounced em-ay-pee three dee du'bl-yu)
Selects the AP300's daisy wheel.

map3g (->)
(pronounced em-ay-pee three jee)
Selects the AP300's left margin offset (gutter).

map3pbs (->)
(pronounced em-ay-pee three pee bee ess)
Selects the AP300's pause-between-sheets option.

map3pitch (->)
(pronounced em-ay-pee three' pitch)
Selects pitch for the AP300.

map3tray (->)
(pronounced em-ay-pee three tray)
Selects the AP300's tray.

map4csf (->)
(pronounced em-ay-pee for see ess eff)
Selects the cut sheet feeder option for the AP400.

map4d (->)
(pronounced em-ay-pee for dee)
Selects print direction for the AP400.

map4dw (->)
(pronounced em-ay-pee for dee du'bl-yu)
Selects the AP400's daisy wheel.

map4g (->)
(pronounced em-ay-pee for jee)
Selects the AP400's left margin offset (gutter).

map4pbs (->)
(pronounced em-ay-pee for pee bee ess)
Selects AP400's pause-between-sheets option.

map4pitch (->)
(pronounced em-ay-pee for pitch)
Selects the AP400's pitch.

map4tray (->)
(pronounced em-ay-pee for tray)
Selects the AP400's tray.

map100dw (->)
(pronounced em-ay-pee wun-hun'dred dee du'bl-yu)
Selects the AP100's daisy wheel.

map100g (->)
(pronounced em-ay-pee wun-hun'dred jee)
Selects the AP100's left margin offset (gutter).

map100pbs (->)
(pronounced em-ay-pee wun-hun'dred pee bee ess)
Selects AP100's pause-between-sheets option.

map100pitch (->)
(pronounced em-ay-pee wun-hun'dred pitch)
Selects the AP100's pitch.

mapc (->)
(pronounced em-ay-pee see)
Selects the alternate printer and the associated setup group. Executes that setup group which displays that printer's parameter choices.

mapct (->)
(pronounced em-ay-pee see tee)
Previews the choices for the selected printer followed by the serial port choices.

mapp (->)
(pronounced em-ay-pee pee)
Interacts with the user to select an alternate printer. Doesn't present the AP100 serial printer. Alters the groups to go to depending on printer selection.

mappc (->)
(pronounced em-ay-pee pee see)
Displays the parallel port alternate printer choices.

mappct (->)
(pronounced em-ay-pee pee see tee)
Previews the choices for the selected printer.

mbjpcs (->)
(pronounced em bee jay pee see ess)
Selects the BubbleJet's character-set.

mbjpd (->)
(pronounced em bee jay pee dee)
Selects the BubbleJet's print direction.

mbjpg (->)
(pronounced em bee jay pee jee)
Selects the BubbleJet's left margin offset (gutter).

mbjppbs (->)
(pronounced em bee jay pee pee bee ess)
Selects BubbleJet's pause-between-sheets option.

mbjppitch (->)
(pronounced em-bee-jay pee' pitch)
Selects the BubbleJet's pitch.

mbotmgn (->)
(pronounced em-bee-oh-tee em-jee-en)
Selects the bottom margin.

mdbotmgn (->)
(pronounced em dee bee oh tee em jee en)
Selects the default document's bottom margin.

mdecimals (->)
(pronounced em des-i-muls)
Selects the number of decimal places in calculation results.

mdfirstpage# (->)
(pronounced em dee first payj sharp)
Selects the default document's first page number.

mdirection (i g ->)
(pronounced em dy-rek-shun)
Sets the printing direction for printer i, and displays on line g.

mdisplay (->)
(pronounced em dis-play)
Selects normal or inverted video.

mdpagelen (->)
(pronounced em dee payj' len)
Selects the default document's page length.

mdprintpage# (->)
(pronounced em dee print payj sharp)
Selects the default document's first printed page number.

mdtopmgn (->)
(pronounced em dee top em jee en)
Selects the default document's top margin.

mDWchoice (a l x i g ->)
(pronounced em dee du'bl-yu choyss')
Displays a Daisy Wheel choice for index i. The parameters a, l, x and g are passed on to **boldtolbuf**.

merror (->)
(pronounced em air-ror)
Selects the way errors are noted.

mfirstpage# (->)
(pronounced em first payj sharp)
Gets the first document page number.

mfxd (->)
(pronounced em eff eks dee)
Selects the FX80's print direction.

mfxg (->)
(pronounced em eff eks jee)
Selects the FX80's left margin offset (gutter).

mfxpbs (->)
(pronounced em eff eks pee bee ess)
Selects FX80's pause-between-sheets option.

mfxpitch (->)
(pronounced em eff eks pitch)
Selects the FX80's pitch.

mfxpl (->)
(pronounced em eff eks pee ell)
Selects the FX80's typestyle.

mimab (->)
(pronounced em eye em ay bee)
Enables or disables the internal modem's answerback option.

mimbpw (->)
(pronounced em eye em bee pee du'bl-yu)
Selects the internal modem's number of data bits per character.

mimcfc (->)
(pronounced em eye em see eff see)
Selects the internal modem's communication character set.

mimct (->)
(pronounced em eye em see tee)
Selects the internal modem's disconnect time.

mimdpx (->)
(pronounced em eye em dee pee eks)
Selects the communications mode.

mimlt (->)
(pronounced em eye em ell tee)
Selects the internal modem's line terminator.

mimpty (->)
(pronounced em eye em pee tee wy)
Selects the internal modem's parity.

mimra (->)
(pronounced em eye em arr ay)
Selects the number of rings for the internal modem to answer.

mims (->)
(pronounced em eye em ess)
Displays the title line and selects the internal modem bit rate.

mimsb (->)
(pronounced em eye em ess bee)
Selects the internal modem's number of stop bits per character.

mkeyboard (->)
(pronounced em kee-bord)
Selects which keyboard layout to use.

mlbpcf (->)
(pronounced em ell bee pee see eff)
Selects the LBP's character font.

mlbpg (->)
(pronounced em ell bee pee jee)
Selects the LBP's left margin offset (gutter).

mlbpl (->)
(pronounced em ell bee pee ell)
Selects the LBP's typestyle.

mlbpp/l (->)
(pronounced em ell bee pee pee slash ell)
Selects the LBP's portrait/landscape printing mode.

mlbppitch (->)
(pronounced em ell bee pee pitch)
Selects the LBP's pitch.

mlineterm (i ->)
(pronounced em lyne term)
Displays the line termination option for index i.

mmp (->)
(pronounced em em pee)
The main printer selection control word. Adjusts the display groups to show on the screen, depending on printer choice and connection.

mmpc (->)
(pronounced em em pee see)
Selects the main printer. Adjusts the token range executed by **mmpct** for the selected printer.

mmpcon (->)
(pronounced em em pee con)
Selects the main printer connection. Alters the group display sequence depending on the printer selected. Checks for printers that are serial or parallel only (skips them if appropriate, and substitutes the LBP).

mmpct (->)
(pronounced em em pee see tee)
Previews the choices for the selected printer.

mnewapcsf (->)
(pronounced em new ay pee see ess eff)
Selects the new AP's cut-sheet feeder option.

mnewapd (->)
(pronounced em noo ay pee dee)
Selects the new AP's print direction.

mnewapdw (->)
(pronounced em noo ay pee dee du'bl-yu)
Selects the new AP's daisy wheel.

mnewapg (->)
(pronounced em noo ay pee jee)
Selects the new AP's left margin offset (gutter).

mnewappbs (->)
(pronounced em noo ay pee pee bee ess)
Selects new AP's pause-between sheets option.

mnewappitch (->)
(pronounced em noo ay pee pitch)
Selects the new AP printer's pitch.

mnewaptray (->)
(pronounced em noo ay pee tray)
Selects the new AP printer's tray option.

mpagelen (->)
(pronounced em payj len)
Gets the page length for the selected pages.

mpc0 (-> t a l)
(pronounced em pee see zeer-oh)
Stacks a and l, the Cat180 string address and length, and also t, the token of the word that, when executed, stacks the Cat180 index range.

mpc1 (-> t a l)
(pronounced em pee see wun)
Stacks a and l, the VP3103II's string address and length, and also t, the token of the word that, when executed, stacks the LBP index range.

mpc2 (-> t a l)
(pronounced em pee see too)
Stacks a and l, the new AP printer's string address and length, and also t, the token of the word that, when executed, stacks the newAP index range.

mpc3 (-> t a l)
(pronounced em pee see three)
Stacks a and l, the AP400's string address and length, and also t, the token of the word that, when executed, stacks the AP4 index range.

mpc4 (-> t a l)
(pronounced em pee see for)
Stacks a and l, the AP300's string address and length, and also t, the token of the word that, when executed, stacks the AP3 index range.

mpc5 (-> t a l)
(pronounced em pee see five)
Stacks a and l, the AP100's string address and length, and also t, the token of the word that, when executed, stacks the AP100 index range.

mpc6 (-> t a l)
(pronounced em pee see siks)
Stacks a and l, the BubbleJet's string address and length, and also t, the token of the word that, when executed, stacks the BJP index range.

mpcc (-> t a l)
(pronounced em pee see see)
Stacks a and l, the FX80's string address and length, and also t, the token of the word that, when executed, stacks the FX80 index range.

mpcn (-> t a l)
(pronounced em pee see en)
Stacks a and l, the "No Printer" string address and length, and also t, the token of the word that, when executed, stacks the NON index range.

mpitch (g i ->)
(pronounced em pitch)
Displays the three common printer pitches for index i and i 1+. Displays on group display lines g and g 1+.

mprintpage# (->)
(pronounced em' print payj' sharp)
Gets the first page number that prints for the document.

mring (->)
(pronounced em ring)
Selects the internal modem's ring indicator.

mscdpx (->)
(pronounced em ess see dee pee eks)
Displays the title line and selects the communications mode.

mspbps (->)
(pronounced em ess pee bee pee ess)
Sets the serial bit rate.

mspbpw (->)
(pronounced em ess pee b p w)
Sets the number of serial data bits.

mspcon (->)
(pronounced em ess pee con)
Selects the serial port connection and alters the group execution sequence, depending on the choice.

msppty (->)
(pronounced em ess pee pee tee ay)
Sets the serial parity.

msps (->)
(pronounced em ess pee ess)
Previews the serial port setup.

mspsb (->)
(pronounced em ess pee ess bee)
Sets the number of serial stop bits.

mssetup (->)
(pronounced em ess setup)
Selects the option to read the setup data from the disk.

msspell (->)
(pronounced em ess spell)
Selects the option to read the spelling dictionary from the disk.

mtab (->)
(pronounced em tab)
Enables or disables the Send command answerback.

mtimeout (->)
(pronounced em tyme-out)
Selects the time before the screen goes dark.

mtlt (->)
(pronounced em tee ell tee)
Selects the Send command line terminator and executes the serial port preview.

mtopmgn (->)
(pronounced em top em jee en)
Gets the top margin for the selected pages.

mtyper (->)
(pronounced em typ'er)
Selects the typewriter mode.

myesno (a l x i g ->)
(pronounced em yes noh)

Displays the yes/no option for index i. The parameters a, l, x and g are passed on to **boldtolbuf**.

newAP (-> l u)
(pronounced noo ay pee)
Stacks the new AP printer's lower and upper group index range.

NON (-> l u)
(pronounced nahn)
Stacks the non-printer's lower and upper group index range.

numberdisp (->)
(pronounced num'ber disp)
Displays the number in **numbr** in decimal and in bold on the current line.

oldsetdata (k -> addr)
(pronounced ohld set' day-ta)
Gets the address of the kth item in the **oldset** array.

pchoicecode (n ->)
(pronounced pee choyss' kode)
Gets a scan code for the parallel printers, and, if it is a leap code, adjusts the nth choice number on the line being selected, rolling the number to the upper or lower limit when necessary. It saves the scan code if it is a space or setup code, otherwise it throws it away.

perusecode (px n ->)
(pronounced pa-rooz' kode)
Do **choicecode** or display the "can't change now" message and don't allow choice changing if spooling or off-hook. Centers the message between **px** and **choicex**.

presetgplines (->)
(pronounced preset jee pee lynes)
Presets the parameters for the maximum and minimum line number selectors.

printercode (-> code)
(pronounced print'er kode)
Stacks the printer code selected by **whichprinter**.

printerinfo (n -> v)
(pronounced print'er in'foh)
Stacks the value of printer parameter n, 0<=n<=7. If the 16-bit value is the last two characters from "none", then it stacks the ASCII string "none" as a 32-bit integer.

printerport (-> fl)
(pronounced print'er port)
Examines **whichprinter**, returning -1 if it is attached to a parallel port, 0 if it is attached to a serial port, and 1 if it is not connected.

rom>svsetup! (flg ->)
(pronounced rahm too ess vee set'up stor)
Sets the flag that indicates that the svram has been setup from ROM.

rom>svsetup? (-> flg)
(pronounced rahm too ess vee set'up kwes'chun)
Returns "true" if the svram data has been setup from ROM.

savesetup (->)
(pronounced sayve' set'up)
Calculates the setup data's checksum and saves the checksum in the svram.

savespell (->)
(pronounced sayve spell)
Moves the custom spelling dictionary from svram to a temporary location, calculates the data checksum there, and then saves the checksum back in svram.

schoicecode (n ->)
(pronounced ess choyss kode)
Gets a scan code for the serial printers, and, if it is a leap code, adjusts the nth choice number on the line being selected, rolling the number to the upper or lower limit when necessary. It saves the scan code if it is a space or setup code, otherwise it throws it away.

scode (-> sc)
(pronounced ess kode)
If both Use Front keys are up, sets up the Cat and resumes operation. Otherwise, it returns with a legal Setup keyboard scan code.

sendcommand (-> addr)
(pronounced send' kom-mand')
Gets the address of the Send command direction in the setup system area.

serialport (-> l u)
(pronounced seer'i-ul port)
Stacks the lower and upper "Serial Port Setup" group indexes.

set-modem (->)
(pronounced set mo'dem)
Sets the modem parameters from the setup data.

set-serial (->)
(pronounced set seer'i-ul)
Sets the serial port parameters from the setup data.

setblanks (c n ->)
(pronounced set blanks)
Sets n blanks into line buffer starting at column c.

setdata@ (k -> n)
(pronounced set day'ta fetch)
Get the data from item K in the **setdata** vector.

setline (n ->)
(pronounced set lyne)
Sets the relative display line number to n, sets the horizontal position to the leftmost and puts blanks in the line buffer, **lbuff**.

Setup (->)
(pronounced set'up)
The highest level setup word. This word is executed by holding down the Use Front key, and, while holding it, pressing the Setup key.

setup>temp (->)
(pronounced set'up too temp)
Moves the setup data to a temporary RAM buffer, where it is ready to write into svram.

setupcat (->)
(pronounced set'up cat)
Sets up the Cat with the data from the setup data vector.

setupcs (-> s)
(pronounced set'up see ess)
Fetches the setup data checksum from the system status RAM area.

setx (n ->)
(pronounced set eks)
Sets the horizontal place in the line buffer to text columm n.

si<>#hl (n -> m)
(pronounced ess eye to from sharp aytch ell)
Converts the number of half-lines to the vector index; also converts the vector index to the number of half-lines. The decision is made by examining the parameter n. If 0<=n<=7, m is the appropriate number of half-lines and if n>7, m is the corresponding index.

spellcs (-> s)
(pronounced spell see ess)
Stacks the custom spelling dictionary checksum that was calculated in the tempoary buffer area.

stype (a l ->)
(pronounced ess type)
Outputs a string which is l characters long, and starts at address a, to the display at the current row and column.

svid (-> id)
(pronounced ess vee eye-dee)
Stacks the svram data format identification numbers. The spelling identification number is in the upper 16 bits and the setup identification number is in the lower 16 bits.

svid! (s ->)
(pronounced ess vee eye-dee store)
Stores the saved svram data format identification numbers.

svid@ (-> s)
(pronounced ess vee eye-dee fetch)
Fetches the saved svram data format identification numbers.

svramsetup>temp (->)
(pronounced ess vee ram set'up too temp)
Moves svram setup data to temporary RAM.

svramspell>temp (->)
(pronounced ess vee ram spell too temp)
Moves svram spelling dictionary to temporary RAM.

svsetupcs! (s ->)
(pronounced ess vee set'up see ess store)
Stores the svram setup data checksum.

svsetupcs@ (-> s)
(pronounced ess vee set'up see ess fetch)
Fetches the svram setup data checksum.

svspellcs! (s ->)
(pronounced ess vee spell see ess store)
Stores the svram spelling checksum.

svspellcs@ (-> s)
(pronounced ess vee spell see ess fetch)
Fetches the svram spelling checksum.

temp>setup (->)
(pronounced temp' too set'up)
Moves setup data from temporary RAM to the system status RAM.

temp>svramsetup (->)
(pronounced temp' to ess vee ram set'up)
Moves the data in the temporary RAM to the svram setup data area.

temp>svramspell (->)
(pronounced temp' to ess vee ram spell)
Move the temporary RAM to the svram spelling dictionary.

thisdocdata (->)
(pronounced this dok day'ta)
Initializes certain parameters in Setup from the current document's data.

tolbuf (a l ->)
(pronounced too ell-buff)
Moves the l-byte string at a to the line buffer at the current x,y location. Advances x as necessary.

topmsg (->)
(pronounced top em ess jee)
Displays the instruction message at the top of the Setup screen.

tromaddr' (-> addr)
(pronounced tee rom ad'der tik)
A vector containing the number of half-lines for each paper size (computed according to Canon's method).

ultype (a l x y ->)
(pronounced yu ell type)
Outputs the l character string at a, starting at the current screen row and column, and underlines it.

wheel! (s ->)
(pronounced weel store)
Sets the spare byte in svram.

wheel@ (-> s)
(pronounced weel fetch)
Fetches the spare byte from svram.

wonb (->)
(pronounced du'bl-yu ahn bee)
Sets the screen to white-on-black; sets the ruler to black, and turns the black screen flag on.