WGS™/WORD ERA™ Glossary Functions

First Edition April 1988 73−00428−A

CHAPTER 1

CHAPTER 2

CHAPTER 7

CHAPTER 8

CHAPTER 9

CHAPTER 10

APPENDIX C

APPENDIX D

APPENDIX E

ABOUT THIS BOOK

This book is a learning and reference guide for WORD ERA Glossary Functions. It shows you:

How to create and use Glossary-by-Example entries

How to create glossary documents and write glossary entries

How to use glossary programming functions

How to use your glossary entries productively

If you know how to use WORD ERA you can learn to use Glossary Functions.

Glossary Functions is a WORD ERA feature that provides you with a special *glossary document* where you store frequently-typed words and phrases. You can then have the stored text automatically typed in your document by recalling it from the glossary document with two keystrokes. In addition to text, you can store frequently-preformed word processing functions and recall them to perform automatically in your document.

Using a glossary entry, you can automate almost any word processing task. You can use glossary entires to insert standard paragraphs for contract, reports, form letters, wills, leases, and other repetitive, standard applications. Glossary entries help you type complicated documents that include tables, forms, multiple format lines, numbered lists, or financial data.

CHAPTER 1

ABOUT GLOSSARY

WHO CAN LEARN GLOSSARY?

Anyone who knows WORD ERA can learn to use Glossary.

If you are not familiar with WORD ERA you can learn how to use it by studying the *WORD ERA Self-Paced Learning Guide.*

WHAT IS GLOSSARY?

Glossary Stores Text and Function Keywords

WORD ERA Glossary is a function that provides you with a special glossary document where you can store frequently-typed words and phrases in a glossary entry. You can then have the stored text automatically typed in your document by recalling the entry from the glossary document with just two keystrokes. You can also perform word processing functions automatically (like Center, Indent, or Execute) by storing keywords in your glossary document.

Glossary Entries are Easy to Create and Use

You type the text and function keywords as a glossary entry in the glossary document. Each glossary entry is identified by a one-character label that you assign. You can have as many as 94 entries in one glossary document.

To use a glossary entry while you are editing a document, you attach the glossary document, press the GL key, then type the one-character entry label. The text stored in the entry is typed at the cursor location in your document just as though you had typed it from the keyboard, only much faster. The following entry example types a company name. (Braces mark the beginning and ending of the glossary entry.)

All References to *archive diskette* should be changed to *archive media*. *Archive Media* refers to diskettes, cartridge tapes, or reel-to-reel tapes.

The appearance of the different screen symbols that are displayed on your terminal may be different than described in the documentation. The reason for this difference is that the screen symbols are terminal independent. For example, the center symbol appears diamond-shaped on some terminals. On other terminals, the center symbol may be displayed as another shape.

Refer to the WORD ERA terminal specification sheet for the terminal type you are using, to see the screen symbols used to display each WORD ERA text formatting character.

```
entry a
{
     "Tigera Corporation"
}
```

If you want the text to be centered and followed by returns, you include keywords as instructions in your glossary entry. The following entry example types a centered company name followed by two returns.

```
entry c
{
     insert "(trademark pending)" execute
}
```

Glossary Entries Save Time

You save typing time and improve typing accuracy when you use glossary entries to type text and perform repetitive functions. Glossaries are great time savers when you frequently type legal or engineering phrases like, "hereinafter referred to," or "gallium aluminum arsenide." For example, using glossary entry c, you can insert the phrase "(trademark pending)" at the cursor location in your document.

```
entry b
{
"Tigera Corporation"
}
```

You Can Use Glossary Entries to Automate Your Typing

You can use a glossary entry to automate almost any word processing task. You can create glossary entries that will insert standard paragraphs for contracts, reports, or form letters anywhere you want them in your document. You can use a glossary entry to type field labels for Records Processing list documents. Glossary entries help you type complicated documents that include tables, forms, multiple format lines, numbered lists, or financial data.

Glossary Gives You Programming Power

In addition to text and key function storage, Glossary gives you full programming capability. In Part 2 you will learn how to use all the following programming features:

Variables
Relational and assignment operators
Conditional testing
Control statements
Operating system command access
Document reading and writing functions
Display functions
Error and logical functions
Interactive functions
Mathematical functions
String functions

The following is just a small sample of the types of glossary programs you can write to serve all your production needs:

Interactive glossary entries: Write interactive glossary entries with your own prompts and error messages using the **prompt** and **error** functions. Using the **keysin** function, write entries that stop during recall and permit the operator to enter data, then continue the entry.

The combination of **prompt**, **keysin**, and **error** are invaluable tools for designing glossary entries that fill out forms, request variable information from the operator, or create lists.

Mathematical functions and calculations: Write glossary entries that work with the WORD ERA Math function to perform calculations on numbers in your document; update parts lists; calculate financial data; or do incremental counts.

Conditional testing: Write entries that can ask questions about document conditions and perform functions based on the answers. For example, if the cursor is under the character "a", you may want some text deleted. If it is not under "a", perhaps you want to insert text.

You will learn you how to use these and other glossary programming functions in Part 2.

You Can Create Glossary Entries in Two Ways

The following steps show two ways to create a glossary entry:

1. **Glossary by example:** creating a glossary entry by example in your glossary document while you edit your text document.

   The quickest way to store simple, short glossary entries is to create them by example. While you are typing text and using functions in your text document you are also storing them in your glossary document for later recall. Chapter 3 teaches you how to create a glossary entry by example.

2. **Write a glossary entry:** writing a glossary entry by typing it in your glossary document.

   You can utilize the full programming power of Glossary when you type your glossary entries directly into your glossary document. A glossary entry by example allows you to only store keystrokes (keywords and characters). When you write a glossary program, you can use the full range of Glossary functions, like **if** statements, **while** loops, subroutines, math and string functions.

   You will learn how to use these and other glossary functions to write glossary programs in Part 2.

   Chapter 4 in this Part teaches you how to write a glossary entry in your glossary document.

You must have a glossary document before you can create a glossary entry by example or write a glossary entry. Chapter 2 shows you how to create a glossary document.

CHAPTER 2

CREATING A GLOSSARY DOCUMENT

THE GLOSSARY DOCUMENT

Before you can create a glossary entry you must create a glossary document. The major differences between a text document and a glossary document are:

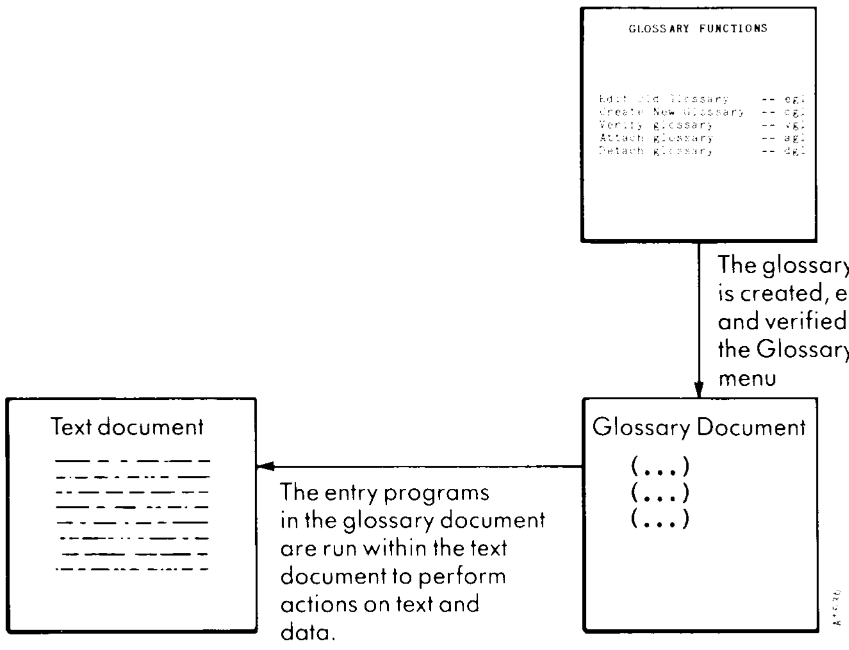
> The glossary document is usually created and edited from the Glossary Functions menu. The Glossary Functions menu is accessed by selecting Glossary Functions from the WORD ERA Main menu.
>
> The glossary document contains glossary entries instead of standard text.
>
> The glossary document must be verified each time you add or change an entry. The verification process compiles your entries into executable programs and checks for programming errors at the same time. The verification process is explained in detail in Chapter 4.
>
> The glossary document must be attached, either from a menu or from a document edit screen before you can use an entry. This chapter tells you how to attach glossary documents.

You can perform all WORD ERA editing and formatting functions in a glossary document. You can perform menu functions like delete, rename, move, copy, print, or archive on a glossary document.

Figure 1 illustrates the relationship of your glossary document and glossary entries to your text document.

As you can see from Figure 1, glossary documents are created separately from text documents. When you edit your text document, you attach the glossary document and use the entries to enter text or perform functions.

Figure 1 Relationship Between the Glossary Document, Glossary Entries, and the Text Document


GLOSSARY FUNCTIONS
Edit Old Glossary -- egl
Create New Glossary -- cgl
Verify glossary -- vgl
Attach glossary -- agl
Detach glossary -- dgl
The glossary document is created, edited and verified from the Glossary Functions menu
Glossary Document
(...)
(...)
(...)
The entry programs in the glossary document are run within the text document to perform actions on text and data.
Text document

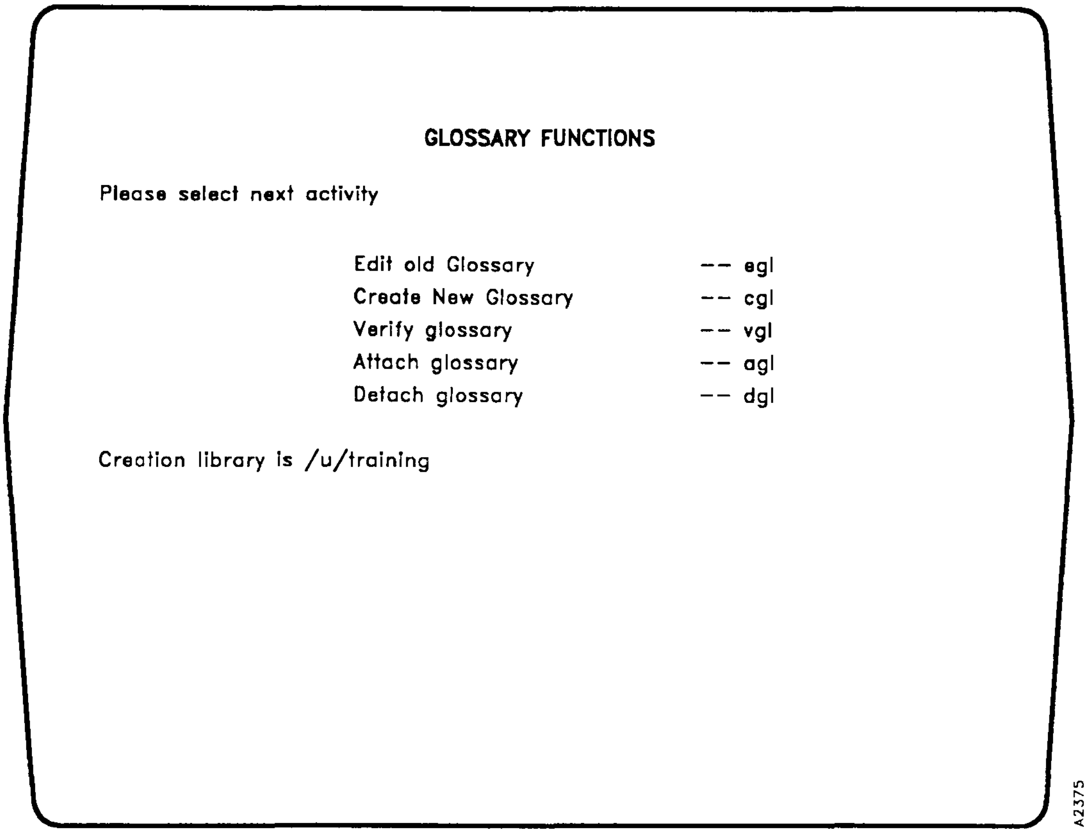
THE GLOSSARY FUNCTIONS MENU

You can perform most glossary document activities from the Glossary Functions menu. The Glossary Functions menu is shown in Figure 2. To display the menu, select Glossary Functions from the WORD ERA Main menu.

You are already familiar with the first two selections on the Glossary Functions menu, editing and creating documents. Verifying, attaching, and detaching are activities specific to glossary documents.

The step-by-step instructions following Figure 2 show you how to create, verify, and attach a glossary document.

Figure 2 The Glossary Functions Menu



```
                         GLOSSARY FUNCTIONS

Please select next activity

                 Edit old Glossary          -- egl
                 Create New Glossary        -- cgl
                 Verify glossary            -- vgl
                 Attach glossary            -- agl
                 Detach glossary            -- dgl

Creation library is /u/training
```

HOW TO CREATE A GLOSSARY DOCUMENT

The following steps show you how to quickly create, verify, and attach a glossary document:

You should begin these steps from the WORD ERA Main menu.

Create the Glossary Document: There are three ways to create a glossary document: Select Create New Glossary from the Glossary Functions menu. (The new glossary document is automatically verified when you use this selection.)

Select Create New Document from the Main menu. (A new glossary document is not automatically verified if you use this selection.) Use the shortcut code **cgl** from any menu. (The new glossary is verified when you use this selection.)

Move the MARKER to Glossary Functions

Press EXECUTE

Move the MARKER to Create New Glossary

Press EXECUTE

Name the glossary document: The same naming conventions that apply to a text document apply to naming a glossary document. Glossary documents are identified on the Document Index by two asterisks (**) displayed before the glossary document name. The asterisks are automatically added to a glossary name when it is verified. The prototype document can be a text document, a glossary document, or the default "0000."

Type **gloss1**

Press EXECUTE

Press EXECUTE to accept the prototype default "0000."

Fill in the Glossary Summary: Notice that the summary screen is a "Glossary Summary" rather than a "Document Summary." You fill in the Glossary Summary just like you do a Document Summary. The Statistics portion of the Glossary Summary is also the same as a Document Summary.

Fill in the Glossary Summary by typing the Document Title, the Author, the Operator, and the Comments lines

Press EXECUTE

The glossary document edit screen is displayed

The glossary document edit screen: The glossary document is just like a text document. You can change the format line or add alternate format lines. You can perform all editing and word processing functions. Also like a text document, the glossary document has pages H, F, N, and W.

Do not enter any text in the glossary document. You have created it so you can learn how to create a glossary by example. You should end the edit of the glossary document now.

DO NOT enter any text in the glossary document.

Press CANCEL to display the END OF EDIT options menu

Press EXECUTE

Verify the glossary document: The verification process compiles glossary entries into an executable code so you can run them as programs in your text document. (It also checks for errors, as you will learn in the next chapter.) Verification occurs automatically if you create or edit your glossary using the Glossary Functions menu or the shortcut code **cgl**.

If you use the Main menu to create a new glossary document, you must go to the Glossary Functions menu and use the Verify Glossary selection or remain at the Main menu and use the shortcut code **vgl** to verify the glossary. Thereafter, the glossary is verified no matter which edit selection you use.

The message Verifying is briefly displayed

The messages *Press EXECUTE to continue* and *An empty glossary is attached* are displayed

(The glossary document is empty because you have no entries in it at this time)

Press EXECUTE

Press CANCEL to return to the Main menu.

You have created, verified, and attached your new glossary document **gloss1**

In addition to creating and verifying a glossary document, you must attach it before you can use it. You can also detach a glossary document.

Attach the glossary: The glossary document must be attached before you can use its entries. There are five ways to attach a glossary document:

Verify the glossary; it is automatically attached after it is successfully verified.

Use the Attach Glossary selection on the Glossary Functions menu.

Attach a glossary document while editing a text document, first press COMMAND, then press GL, then type the glossary document name.

Use the shortcut code **agl** to attach a glossary document from any menu.

Select Index from the Main menu (or use the shortcut code **ixs** to display a Short Form Index), position the cursor on the glossary name, and press the GL key. (A glossary document cannot be attached from the Index if you use Command I or i to access the index while you are editing a document.)

Detach the glossary document: When you attach a glossary document you automatically detach any previously attached glossary document. You can detach an attached glossary by selecting Detach Glossary from the glossary functions menu or by using the shortcut code **dgl** from any menu.

SUMMARY

In this chapter you learned how to use the Glossary Functions menu to create, verify, and attach a glossary document.

In Chapter 3 you will learn how to create and use a glossary entry by example while you edit a text document. Chapter 3 also gives you suggestions for creating several glossary by example entries.

In Chapter 4 you will learn:

how to write and edit entries in your glossary document,

the elements of a glossary entry,

and more about glossary entries and glossary documents.

Chapter 4 also provides examples and suggestions for writing glossary entries.

CHAPTER 3

CREATING A GLOSSARY ENTRY BY EXAMPLE

In Chapter 2 you learned how to use the Glossary Functions menu and how to create a glossary document. In this chapter you will learn how to create and use a glossary entry by example.

Using the Glossary by Example feature, you create a glossary entry that duplicates your keystrokes as you perform them. (Keystrokes include text typing and word processing functions like Insert or Return.)

Once you have created the entry, you can use it immediately within the document you are editing. A glossary created by example becomes a permanent entry in your glossary document, so you can use it with other documents as well.

Remember, you must have an existing glossary document attached before you can create an entry by example.

> **NOTE**: If you do not have a glossary document, follow the steps in Chapter 2 to create one before you begin this section.

HOW TO CREATE A GLOSSARY BY EXAMPLE

The following steps show you how to create three glossary entries by example and recall them in your document:

entry c:	This entry is a company name and address
entry d:	This entry is a company name and address, centered and typed on three lines
entry e:	This entry inserts a name and title

Entry c: Creating an Entry to Type a Company Name

Create a new text document: When you are learning to create glossary entries or testing new entries, it is always a good idea to try them in a "test" document before you use them in your regular documents.

Use Create New Document on the Main menu to create a new text document

Name the document **learngloss**

To begin, the cursor should be on Line 1, Pos 1 of your text document

Attach the glossary document: Before you can create a glossary by example, you must attach your glossary document **gloss1.**

Press COMMAND

Press GL

Type gloss1

Press EXECUTE

Start Glossary by Example Mode: To start creating your glossary entry, press MODE, then press GL. The flashing message Glossary entry is displayed at the bottom center of the screen. This message will continue to flash until you have completed your entry.

Press MODE

Press GL

Type the entry: Type the entry exactly as you would like it to appear in your document.

Remember: Every keystroke you make, including mistakes, is being duplicated in your glossary entry.

If you make a mistake: If you make a mistake while you are typing the entry use the Backspace key to back up and correct the error. Or, you can press CANCEL to terminate the entry and start over.

Type **TIGERA Corporation**

Concluding the entry: When you have finished typing the entry press MODE, then press GL. You must now assign a label in response to the Which entry? prompt.

Press MODE

Press GL

Assigning a label: To assign a label, type one character and press EXECUTE. You can use any one of 94 keyboard characters as an entry label (allowing you 94 entries per glossary document). You should not use quotation marks, and there are special considerations when you use space or backslash. You cannot duplicate entry labels in the same glossary document. For example, you cannot have two entries labeled a. If you inadvertently assign a duplicate label, the error message Entry in use is displayed. If this occurs, press EXECUTE and assign a different label. (See Chapter 4 for detailed information on entry labels).

Type **c**

Press EXECUTE

Recalling the entry: Now that you have created entry c, you can use it. To recall an entry, press GL, then type the one character entry label. Try recalling the entry several times. Notice how fast it is typed in your document.

To recall entry c:

Press GL

Type **c**

You have created your first glossary entry by example. If you have to type your company name frequently, this entry is a practical example for you to use. Try creating another entry, substituting the name of your company for "TIGERA Corporation."

If you did not type a space following "TIGERA Corporation" and recalled entry c a few times, your screen probably looks like the following example:

> TIGERA CorporationTIGERA CorporationTIGERA CorporationTIGERA Corporation

If you had typed a space following the word "Systems" when you created the entry, the example above would look like this:

> TIGERA Corporation TIGERA Corporation TIGERA Corporation TIGERA Corporation

If you type the space at the end of the phrase when you create the entry, you won't have to type it in your document. If, however, you will occasionally need to end the phrase with punctuation, like a period or comma, don't include the space in the entry. You can see that you should give some consideration to possible uses for your entries.

You can use short entries in a variety of ways. A company name is one example. Other examples might be: proper names, lengthy titles, or words you have difficulty typing or spelling.

You can end the edit of your document **learngloss** now, or remain in it and continue with the next exercise.

Entry d: Creating an Entry to Type a Company Name and Address

In entry d you will see how you can create an entry that will type both text and function keys in your document. The function keys used in this example are Center and Return. In entry e you will use the function keys Insert and Execute.

Follow the steps below to create entry d:

Creating entry d: To begin this exercise, edit your document **learngloss**, position the cursor a few lines below existing text, then attach your glossary document **gloss1**.

If you ended the edit of **learngloss** when you completed the last exercise, perform the following steps:

1. Edit your text document, **learngloss**

2. Attach the glossary document, **gloss1**

Remember, the exact keystrokes you type are being duplicated in the glossary entry. If you make too many mistakes or unnecessary keystrokes, the entry will take longer to recall. If you want to terminate the entry and start over, press CANCEL. The entry is not preserved until you assign an entry label.

To start entry d:

Position the CURSOR below any existing text in your document

Press MODE

Press GL

To type entry d:

Press CENTER

Type **TIGERA Corporation**

Press RETURN

Press CENTER

Type: **350 Bridge Parkway**

Press RETURN

Press CENTER

Type **Redwood City, CA 94065**

Press RETURN

To conclude entry d:

Press MODE

Press GL

Type **d**

Press EXECUTE

Recalling entry d: When you recall entry d, notice that it is typed in your document exactly as you typed it when you were creating the entry, complete with centers and returns.

To recall entry d:

Press GL

Type: **d**

Entry d should look like the following example when you recall it in your text document.

TIGERA Corporation
350 Bridge Parkway
Redwood City, CA 94065

You can see from entry d that the ability to use function keys (like Center and Return) in your entries provides more possibilities for creative glossary applications.

When you use entry c or d, you must position your cursor beyond or below any existing text before you recall the entry or the text will be overwritten by the entry text. You can include the Insert function in your glossary entry to avoid overwriting existing text. The next exercise shows you how to create an entry that inserts text in your document.

You can end the edit of your document **learngloss** now, or remain in it and continue with the next exercise.

Entry e: Creating an Entry That Inserts Text

Follow the steps below to create entry e:

Creating entry e: To begin this exercise, edit your document **learngloss**, position the cursor a few lines below existing text, then attach your glossary document **gloss1**.

Perform these steps if you ended the edit of **learngloss** when you completed the last exercise:

1. Edit your text document, **learngloss**

2. Attach the glossary document, **gloss1**

To start entry e:

Position the CURSOR below any existing text in your document.

Press MODE

Press GL

To type entry e:

Press INSERT

Type **Mr. John Jones, President**

NOTE: Be sure to include the space following the title so you don't have to insert it later.

Press SPACE

Press EXECUTE

To conclude entry e:

Press MODE

Press GL

Type: **e**

Press EXECUTE

Recalling entry e: To understand how entry e inserts text, position the cursor within existing text. Press GL, then type e.

Position the cursor at the correct place to insert text

Press GL

Type **e**

Entry e should look like the following example when you recall it in your text document.

Mr. John Jones, President

You have now created and recalled three types of glossary entries by example. The next section, Tips on Creating and Using Glossary Entries by Example, gives you additional information about glossary by example entries.

TIPS ON CREATING AND USING GLOSSARY ENTRIES BY EXAMPLE

This following list provides additional information and gives you some points to remember about glossary-by-example entries.

Cursor position: When you recall a glossary entry that does not include the Insert function, be sure your cursor is positioned at the exact location where you want the entry to be typed or inserted. If the cursor is positioned on existing text, the text will be overwritten by the recalled entry.

Using function keys: Remember, anything you can type on the keyboard you can save in a glossary entry. If you do a large volume of production typing, you can use glossary entries to reduce the number of keystrokes you have to type. For example, you can create a glossary by example entry to copy an alternate format line. The keystroke sequence you type to create an alternate format line is:

INSERT COPY FORMAT 2 EXECUTE

Typing this sequence requires five keystrokes, whereas performing it with a glossary entry takes two keystrokes. To make this entry easy to remember, use the number of the alternate format line as the entry label (in this example the label would be entry 2).

Format lines: A recalled glossary entry always uses the current format line in your text document (unless you include a format line as part of the glossary entry).

Using text emphasis modes: If you always highlight or underline certain words or phrases, you can shorten the time it takes to type them by including the text emphasis modes in your glossary entries.

Paragraphs: You can use a glossary by example entry for short paragraphs or forms; however, there is a finite limit to the amount of keystrokes you can store by example (see "Length" below).

When you have a large volume of text or keystroke combinations you would like to use in a glossary entry, you must write the entry directly in the glossary document. Chapter 4 shows you how to do this.

Length: A glossary by example cannot exceed approximately 1024 characters in length. The character count includes text, screen symbols (like RETURN and TAB), page and/or column breaks, and format lines. Since a glossary by example records every keystroke you make, it includes the keys you press to make corrections or move the cursor around. Unless you are very sure exactly what the entry should contain, you may quickly reach the maximum entry size.

If you exceed the character limit while you are creating a glossary by example entry, the Glossary entry prompt stops flashing and the Which entry? prompt is displayed. You can enter a label and press EXECUTE to save the entry, or press CANCEL to terminate the entry.

Modifying or adding to an entry: You can edit your glossary document and modify or add to any entry you have created by example. Chapter 4 shows you how to do this.

Using the Numeric keypad: You can use the numeric keypad just as you would any other key on the keyboard while you are creating a glossary by example.

Number of entries in a glossary document: You can create as many glossary documents as you need, although you can only attach one at a time. You can have up to 94 separate entries in each document. They can be either entries you create by example or write.

Creating glossary by example entries from menus: You can create glossary by example entries to automate keystrokes you perform from menus. For example, if you frequently change between two libraries, you may want to create a glossary by example to perform the following keystroke sequence:

COMMAND **chl library pathname** RETURN

Although the shortcut code **chl** is quick to use, this menu glossary entry is even quicker. From any menu, use the shortcut code **agl** to attach a glossary document. Then follow the same procedure you learned in this chapter to create a glossary entry by example.

Suggestions for creating glossary entries by example: There are as many ways to use glossary by example entries as there are types of work. Consider using entries for repetitious typing; standard paragraphs; Records Processing field labels (the Records Processing User's Guide gives you an example); and technical words and phrases.

You can create a glossary by example entry as you use the Math function to quickly add rows or columns.

Remember, you can print, archive, and perform other document filing functions with a glossary document. When you have created a number of entries, it is easy to lose track of what your entries do and which labels you have used. You may want to print your glossary documents and keep them in a binder for reference.

In Chapter 4 you will learn how to write entries in your glossary document and how to modify your existing glossary entries by example.

CHAPTER 4

WRITING GLOSSARY ENTRIES

In Chapter 3 you learned how to create glossary entries by example. In this chapter you will learn how to write glossary entries in your glossary document. By learning to write entries you will be able to:

modify or add to your glossary-by-example entries

create longer (up to 33,000 characters) glossary entries

use all of the programming functions available in glossary

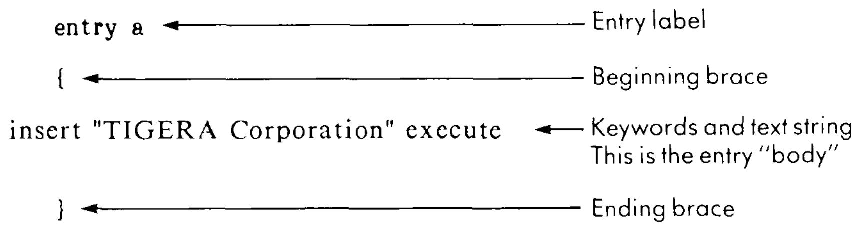
Before you write a glossary entry, you need to understand the elements that compose the entry. Every glossary entry, including the entries you created by example in Chapter 3, contain the same basic elements. These elements are described in the following section:

BASIC ELEMENTS OF A GLOSSARY ENTRY

A glossary entry is composed of the following basic elements:

Entry label
Braces
Keywords
Strings
Comments

Figure 3 shows you a diagram of the elements of a short glossary entry that inserts the text, "TIGERA Corporation," in a document.



```
entry a
{
insert "TIGERA Corporation" execute
}
```

Figure 3 A diagram of the elements of a glossary entry

Entry **e** (the glossary-by-example entry from Chapter 3) is similar to the example shown in Figure 3. If you would like to compare the entries, edit your glossary document **gloss1**, and look at entry **e**, it should look like the following example. (If you made any corrections while you were creating entry **e**, your entry may contain extra keywords such as backspace or left.)

```
entry e
{
insert "Mr. John Jones, President " execute
}
```

As you can see, although the entries insert different text, they both contain the same structural elements. Read the following descriptions of these glossary entry elements before you begin writing entries in your glossary document.

Entry Labels

Each glossary entry starts with the word entry. The single character after this word is the label. You have 94 keyboard characters available to use as entry labels. These may be any uppercase or lowercase letter, numeral, or symbol such as !, @, and ~.

To use either a space or a backslash as an entry label, you must precede the label with a backslash. To use a space as a label, type "entry \ ". To use a backslash as a label, type "entry \\".

Each entry label must be unique; you cannot have two entries labeled x in the same glossary document. When you run a glossary entry in your word processing document, you recall the entry by pressing the GL key and typing the single character entry label. The label of the sample entry in Figure 4-1 is the character **a**.

Braces

Two braces { } mark the beginning and ending of the entry. The text between the braces is called the body of the entry. The body of the sample entry in Figure 4-1 contains the keywords **insert** and **execute** and the text string "TIGERA Corporation."

Keywords

Keywords are names that represent the formatting, editing, and cursor movement keys on the keyboard, such as Return, Tab, Delete, and Left (cursor left key).

When you use an entry containing keywords, each keyword will perform its designated function. For example, in the sample entry below, the cursor moves down three lines and deletes a character.

```
entry f
{
     down(3)
     delete execute
}
```

As shown in entry f, the repeated activation of a key can be specified by a number in parentheses immediately following the keyword. See the list of keywords by usage in Appendix D for keywords that accept a parenthetical number.

Strings

A string is any contiguous set of characters that will be typed or inserted into the text document. It may be as short as one character, or may include several paragraphs of text.

A string may consist of any combination of alphabetic or numeric characters, including spaces and special characters such as a Required Space or Required Hyphen.

To differentiate strings from keywords in an entry, you must enclose the strings in quotation marks, as shown in the following example:

```
entry g
{
      insert center "Monthly Report" return execute
}
```

The string in entry g is "Monthly Report." The keywords are **insert, center, return**, and **execute**. When this entry is used in a document, the heading "Monthly Report" is inserted and centered one line above any existing text.

Embedding Keywords in Strings

Entry f uses only keywords. Entry g uses both keywords and a text string. The keywords in entry f are whole keywords, typed outside the string. You can also embed certain keywords within the text string. These keywords are called "abbreviated keywords." A list of keyword abbreviations is provided in Appendix D.

In the following example, entry h is the same as entry f, except the keyword abbreviations for center and return are embedded in the text string. Keyword abbreviations are always preceded by a backslash.

```
entry h
{
      insert "\cMonthly Report\r" execute
}
```

The first two characters in the string, **\c**, are an abbreviation for the keyword center. The last two characters in the string, **\r**, are an abbreviation for the keyword return. Abbreviated keywords must always be placed inside the quotes in a string.

Since double quotation marks are used to define a string, you must always use the keyword abbreviation **\q** instead of the symbol (") when you want to quote a word or phrase within a string. The following example shows the keyword abbreviation for double quotation marks embedded in the string:

```
      "The name of the company is \qTIGERA Corporation\q"
```

This string appears in the text document as:

The name of the company is "TIGERA Corporation"

You can also use the keyword quote to enclose words or phrases in quotation marks as shown in the following example:

"The name of the company is" quote "TIGERA Corporation" quote

As you can see, using the keyword quote is awkward, since you have to split the string into quoted and non-quoted segments.

Single vs Double-Quoted Strings

You can also enclose a string in single quotes ('). However, a single-quoted string is interpreted differently from a double-quoted string. For example, when the string

"\cTIGERA Corporation\r"

is typed in the text document, the center symbol is typed, the string "TIGERA Corporation" is typed, a return symbol is typed, and the cursor advances one line.

When the same string is enclosed in single quotes, it is typed in the text document exactly as it appears in the glossary entry. The string

'\cTIGERA Corporation\r'

is typed in the text document as

\cTIGERA Corporation\r

The Backslash in Strings

The backslash (\) is an escape character; it tells the system that the character following it is to be treated in a special way (for example, \r performs a different function than the solitary character "r"). When you include a backslash in a string you must always precede it with another backslash as shown in the following example:

"The backslash (\\) is a special character."

The string is typed in the document as

The backslash (\) is a special character.

Comments

Comments make glossary entries easier to understand and use by describing the entry. Any text enclosed by /* and */ within an entry is a comment. When a glossary document is verified, or an entry is executed, comments are ignored. As a result, instructional and explanatory comments can be used frequently. Comment lines are used to clarify the function of entry i, as shown in the following example.

```
entry i
{
        /* boldface 5 characters */
        mode "b"        /*Turn boldface on*/
        right(5)        /*Move cursor right five characters*/
        mode "b"        /*Turn boldface off*/
}
```

Comments can be composed of several lines of instructions. In the following example, expanded comments have been added to entry i. Note that the comments now provide instructions on how to use the entry as well as describing it.

The comment paragraph must begin and end with the comment symbols.

```
entry i
{
        /*This entry is used to boldface 5 characters. To use it, place the cursor on
        the first character to be boldfaced. Press GL and type the label i*/

        mode "b"        /*Turn boldface on*/
        right(5)        /*Move cursor right five characters*/
        mode "b"        /*Turn boldface off*/
}
```

Never mix keywords and comments. If you have comments following keywords on one line, be sure the commented section begins and ends with the comment symbols and does not include any keywords. The second example of entry i, above, shows the correct usage of both instructional and explanatory comments.

Two other important issues to consider when you write a glossary entry are screen symbols and format lines.

> **REMEMBER:** Regardless of how complicated your glossary entries become, they share the same structural elements. Be sure you begin each entry with a label, start the body of the entry with a left brace, enclose strings in quotes, spell whole keywords correctly, use the correct keyword abbreviation, and finish the entry with a right brace.

SCREEN SYMBOLS AND FORMAT LINES IN THE GLOSSARY DOCUMENT

Screen Symbols

Screen symbols that are displayed on the editing screen of your glossary document, such as the Return and Tab triangles and the Center diamond, are not recognized as keywords in the glossary program. You must type the full name of the keyword, or use a keyword abbreviation in a string, for that keyword to become part of the glossary program. You can use the standard Returns, Tabs, Indents, and other screen symbols to format your glossary entry so it is easier to read on the editing screen.

Format Lines in Glossary Documents

The format line in a glossary document has no effect when the entry is recalled in a text document. The quoted strings in the glossary entry will wrap to adjust to the right margin of the text document format line. To use a glossary entry to change or insert a format line in the text, document you must make the format line part of the glossary program.

Entry j is a short program that inserts an alternate format line in the text document.

```
entry j
{
     /*inserts alternate format line. Tabs at 8 and 37. Margin at 68.*/
     insert
         format space(7) tab space(28) tab space(30) return execute
     execute
}
```

WRITING GLOSSARY ENTRIES

This section shows you how to modify an existing entry created by example. It also gives you an example of a glossary entry you can write and try. If you would like to understand more about the action of a particular keyword as you are writing the examples refer to the Keywords by Usage list in Appendix D.

Modifying a Glossary-by-Example Entry

The only method you can use to add additional text or functions to an entry created by example is to edit the entry in your glossary document. You can't use the glossary-by-example feature to change or add to an entry created by example. To see how you can add a phrase to entry d (the glossary-by-example entry you created in Chapter 3),perform the following steps:

To modify entry d:

1. Select Glossary Functions from the Main menu.

2. Select Edit Old Glossary and press EXECUTE.

3. Type **gloss1** and press EXECUTE twice.

4. Entry d in your glossary document should look like the following example: (Text lines may be wrapped differently, depending on the format line in your glossary document. You may also have additional keystrokes in your entry, depending on how many corrections you made while you were creating the entry.)

```
entry d
{
center "TIGERA Corporation" return center "350 Bridge Parkway" return center
"Redwood City, CA  94065" return
}
```

5. Place the Cursor at the beginning of the entry body (the "c" in center), and press INSERT

6. Type the following line:

   "Please send correspondence to:" return(2)

7. Press RETURN, then press EXECUTE. Your entry d should now look like the following example

```
entry d
{
"Please send correspondence to:" return(2)
center "TIGERA Corporation" return center "350 Bridge Parkway" return center
"Redwood City, CA  94065" return
}
```

The keyword return(2) is used to place two returns between the line and the address. Instead of return(2), you could use the keyword abbreviation for return. However, because keyword abbreviations do not take a number argument, you must type the abbreviation twice, like the following example.

"Please send correspondence to:\r\r"

Of course, if you only want one return after the line, you can type one keyword abbreviation in the string, like the following example:

"Please send correspondence to:\r"

Either method, embedding the abbreviation or typing the keyword outside the string, is correct. Use the method that seems most natural to you and accomplishes your purpose efficiently.

Modify and Recall the Entry: Once you have modified the entry, perform the following steps to verify, attach, and recall it in a document:

1. Press CANCEL

2. Press EXECUTE

3. The status message (Verifying) is displayed at the bottom of the screen.

4. If the glossary document verifies correctly, the menu from which you edited the glossary document is displayed.

   If the glossary document does not verify correctly, the verification screen is displayed. If this occurs, press RETURN and read the Verifying and Troubleshooting section in this chapter.

5. The glossary document is automatically attached when it is successfully verified. Press CANCEL to return to the Main menu.

6. Edit your text document, **learngloss**.

7. Position the Cursor below any existing text.

8. Press GL, then type **d**.

9. The new version of entry d is typed in your document.

You can save time by creating glossary-by-example entries as you perform your regular typing, then modifying them as necessary. When you add text to an entry created by example, the 1024 character limit no longer applies. You can add as much text as you like, up to approximately 33,000 characters. In Part 2 of this guide you will learn how to create even longer entries by calling other entries as subroutines.

Writing A Glossary Entry Memorandum Form

If you want to practice before you begin writing your own entries, try writing entry k in the next example. Entry k is a memorandum form; you can change any of the headings to match the ones you normally use.

To write entry k, perform the following steps:

1. Select Glossary Functions, then edit your glossary document, **gloss1**.

2. You can type entry k on the same page as the other entries, or you can put in a page break and begin entry k on the next page. You can put as many page breaks as you like in your glossary document (up to the 999 document page limit). It is easy to find entries quickly if you start each entry on a new page; however, you may want to group several short entries on one page.

3. Type entry k exactly as shown in the following example. The entry is shown with whole keywords outside the strings. If you prefer, you can use the appropriate keyword abbreviations instead (refer to the list of keyword abbreviations in Appendix D).

   This entry inserts an alternate format line that sets the right margin at 65 and a tab stop at 10. Note that the keyword space takes a number argument, this feature simplifies typing spaces in a glossary entry.

```
entry k
{
        insert format space(9) tab space(54) return execute(2)
        center "MEMORANDUM" return(2)
        "DATE:" tab return(2)
        "TO:" tab return(2)
        "FROM:" tab return(2)
        "cc:" tab return(2)
        "SUBJECT:" tab return(2)
}
```

4. Press CANCEL, then press EXECUTE to verify and attach the glossary document.

5. When the Glossary Functions menu is displayed, use the shortcut code **edd** to edit your text document, **learngloss**.

   Using the shortcut code **edd** from the Glossary Functions menu is a quick way to edit a text document and check the action of a new glossary program.

6. When the editing screen is displayed, press GL, then type **k** to recall entry k.

7. If you need to modify entry k after you have seen it perform in the text document, leave the document, edit the glossary, verify it, then recall it in your text document. If you are writing a complicated glossary program, you may need to go back and forth between the glossary document and the text document several times until you are satisfied with the program's performance.

Writing Menu Glossary Entries

As you learned from the glossary tip in Chapter 3, you can create a glossary by example entry to store keystrokes you perform from a menu. You can also write a glossary entry to perform menu keystrokes. However, neither glossary by example entries nor written entries cross from the menu to the editing screen, or vice versa. For example, you can write a glossary entry that will create a document, fill out the document summary, and take you to the editing screen, but the entry will terminate at that point. To enter text you must use a different entry. Both entries can be in the same glossary document.

Learning More About Glossary

You have completed the basic exercises showing you how to create glossary entries by example and how to write entries directly in the glossary document.

The remainder of this chapter provides reference information about verifying, troubleshooting, attaching, and detaching glossary documents.

The Keywords by Usage list and the Keyword Abbreviations list in Appendix D are particularly useful. Study these lists carefully before you begin Part 2 Learning Glossary Programming.

If you feel you need to gain a little more understanding of glossary and how it applies to your word processing tasks, write and use several of your own programs before you attempt to learn the new functions in Part 2.

VERIFYING AND TROUBLESHOOTING

When you write a new glossary entry or modify an existing one, you MUST verify the glossary document before you can use the entry. When verification occurs, all entries in the glossary document are verified (even if you modified only one entry). Therefore, the amount of time it takes to verify a glossary document depends on the length of individual entries and how many entries you have in the glossary document.

The verification process checks your glossary entries for several possible error conditions. The basic error conditions are:

Every entry must have a label

Labels cannot be duplicated within the same glossary document

Every entry must begin and end with a brace

Keywords must be spelled and used correctly

Strings must begin and end with a single or double-quotation mark

Comment lines must begin and end with the correct comment symbols

Error conditions that are concerned with programming syntax are covered in Part 2. A list of error messages is provided in Appendix E.

Glossary Verification Options

You have a variety of options available to verify a glossary document. You can also choose not to verify it if you have edited it just to scan the contents or look at an entry. All of these options are described in the following list:

Glossary Verification END OF EDIT Options

Like a text document, the END OF EDIT options menu is displayed when you press CANCEL to end a glossary document editing session.

The optional choices on the menu are EXECUTE, RETURN, COPY, DELETE, and FORMAT. Verification occurs automatically with some options and not with others. Each option is described below:

> **NOTE**: You can use the functions Autosave or Command Return while you are editing your glossary document. Your changes are saved as you edit; however, entries are not verified until you end the edit of the glossary document and press EXECUTE.

EXECUTE: Verification automatically occurs when you press EXECUTE.

If you have just created the glossary document, you can only use the EXECUTE verification option if the document was created from the Glossary Functions menu.

You can create a glossary document from the Main menu; however, you must verify it from the Glossary Functions menu or by using the shortcut code **vgl** before you can use the entries.

Once a glossary document has been verified, you can edit it and use end of edit verification options from any menu.

RETURN: The document is not verified. Returns you to the glossary document editing screen.

COPY: Automatically verifies only the glossary document, not the copy. Saves the glossary document with all changes. Creates a copy of the glossary document that includes only the editing changes saved by using Autosave or Command Return.

DELETE: Does not verify the glossary document. Deletes any changes made during the editing session except changes that were saved by using Autosave or Command Return.

The DELETE option can be very useful if you want to look at the glossary document without making any changes. Of course, like a text document, any changes you make are deleted, so use this option with discretion.

FORMAT: Automatically verifies the glossary document and displays the print menu.

The same END OF EDIT options apply when you edit a glossary document from the Document Index.

Glossary Document Menu Verification Options

You can verify a glossary document without editing it in two ways, they are:

Select Verify Glossary from the Glossary Functions menu

Use the shortcut code **vgl** from any menu that accepts shortcut codes.

Glossary Document Verification Limitations

A glossary document cannot be verified in an open document window. If you edit a glossary entry in a window, you must return to the menu to verify the edit. The changes remain in the entry but are not executable until the glossary document is verified. You can work around this by jumping to the glossary document window, then closing all other windows. You can then end the edit of the glossary document normally and automatically verify it.

A glossary document verifies correctly when you use Command i from a text document editing screen to edit a glossary document; however, it is not automatically attached. You may type or edit an entry using this method, but you must attach the glossary document when you return to the text document editing screen before you can use the new or modified entry.

Correcting Verification Errors

If errors in an entry or entries are detected during the vertification process, the Verification errors options menu shown in the following example is displayed:

```
No. of errors detected :1
Verification errors options

RETURN   to editing screen
DELETE   to Glossary menu
```

The menu in the example shows that one error has been detected in the glossary document. You are offered a choice of two options, RETURN or DELETE, which are described below:

> **DELETE**: Does not verify the glossary document. Saves any changes made during the editing session (including the errors).
>
> If you choose the DELETE option and do not correct the errors, you must correct them. You must repeat the verification process before you can use any entry in the glossary document.
>
> **RETURN**: Returns you to the glossary document editing screen. Choose this option to view and correct entry errors.

The verification process places messages about entry errors on the glossary document work page (page w). To view this information, edit the glossary document, press GO TO PAGE, then type w. The message displayed on the work page shows the date and time that the verification was performed and lists the error or errors detected. A sample error message display is shown in the following example:

```
************************************
Tue Apr 29, 1986 at 20:53:49
************************************

page 2 , line 4 : syntax error :  '{'
```

The error message example indicates that the entry on page 2 is probably missing an ending brace.

After you have looked at the work page, you can go to the page and line number indicated and correct the error or errors.

When you reverify the glossary document after making corrections, any new errors detected are added at the bottom of the work page below existing messages. You should delete error messages from the work page after you have corrected the error.

Most glossary entry errors are simple mistakes, like misspelled keywords, missing braces, or duplicated entry labels. They are usually easy to spot and correct.

Appendix E provides a complete list of verification error messages and gives you suggestions for correcting them.

ATTACHING A GLOSSARY DOCUMENT

To use an entry in a glossary document, you must first attach the glossary document. You can attach a glossary document from a menu, from a document editing screen, or from the Document Index.

You can only attach and use one glossary document at a time. For example, if you are using **gloss1**, and attach **gloss2**, then **gloss1** is detached and you can only use **gloss2**.

Several users can attach and use the same glossary document at the same time. For example, you and your co-worker can both attach and use **gloss1** at the same time.

You cannot edit an attached glossary. If you attempt to edit an attached glossary, the message "Document in use" is displayed.

You cannot delete, rename, or move an attached glossary document. Detach the glossary document before using it with these functions.

You can use any one of the following methods to attach a glossary document:

Verify the glossary. A glossary document is automatically attached when it is successfully verified.

Use the Attach Glossary selection on the Glossary Functions menu.

To attach a glossary from a document editing screen, press COMMAND, press GL, then type the glossary document name.

Use the shortcut code **agl** to attach a glossary document from any menu.

To attach a glossary document from the Document Index, select Index from the Main menu (or use the shortcut code "ixs"), position the cursor on the glossary name, then press GL.

If you use Command i to access the Document Index from your document editing screen, and then edit and modify a glossary document, you must re-attach the glossary document when you return to your document editing screen.

DETACHING A GLOSSARY DOCUMENT

You can detach an attached glossary by selecting Detach Glossary from the Glossary Functions menu or by using the shortcut code **dgl** from any menu.

When you attach a glossary document, you automatically detach any previously attached glossary document.

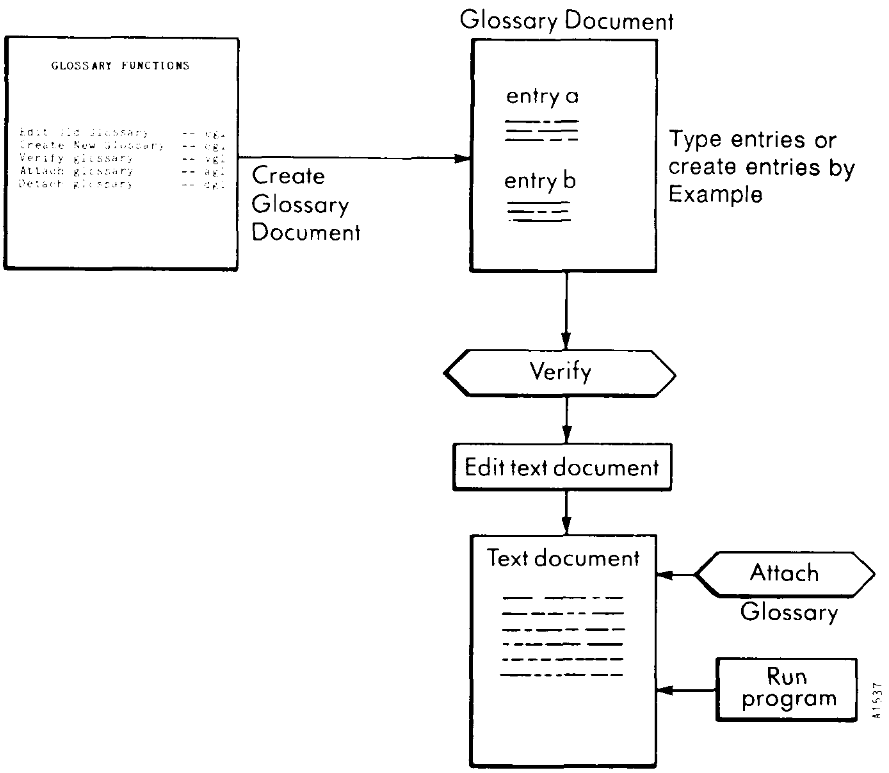
Figure 4 summarizes the glossary program writing, verifying, attaching, and recalling procedure.

SUMMARY

You have completed Part 1 of this guide. You learned how to create a glossary document and how to verify, attach, and detach it. You also learned how to create a glossary by example entry, how to modify it, and how to write a glossary entry.

In Part 2 you will learn how to use the glossary programming language and glossary functions.

Figure 4 **The Glossary Writing, Verifying, and Program Execution Procedure**



CHAPTER 5

INTRODUCTION TO GLOSSARY PROGRAMMING

While you perform the exercises and try out the examples in this book, you are learning to write computer programs using the Glossary programming language.

While Glossary has a great deal in common with other types of computer programming, it was specifically designed to manipulate text and data inside a WORD ERA text document. When you write a glossary entry, you are writing a program. The basic glossary elements you learned in Part 1 can be used to type, create headings and footings, and format your documents. Glossary programming functions greatly expand the range of possible uses for glossary entries.

WHAT IS GLOSSARY PROGRAMMING?

A glossary entry is a computer program. Your glossary program is a set of instructions that tells the computer what to do, how to do it, and in what order to do it. In Part 1 you learned how to use combinations of keywords and strings as program instructions.

In this part you will learn how to write more complicated instructions using special glossary document reading and writing functions, interactive functions, string functions, and mathematical functions. Conditional and control statements permit you to control the order of program execution.

Glossary is a Programming Language

Programs have to be written in a language that the computer understands. Glossary is a programming language that was developed especially for WORD ERA users. Very much like a spoken language, glossary language has a grammatical structure called syntax. It uses declarations called statements and action words (verbs) called functions.

There are special programming language rules you must follow to communicate with and instruct the computer. Chapters 6 through 9 give you these rules.

You may already know a programming language such as BASIC, PASCAL, or C (the language most frequently used on the UNIX operating system). If so, the syntax and logic of the Glossary programming language will be familiar to you.

The Verification Process Compiles Your Programs

Glossary is a compiled programming language. During the verification process, your written glossary entry code is compiled into a compact form that can be read by the machine.

Most programming languages must be converted to a machine-readable form by an interpreter or a compiler. The two methods are similar in result but are different in execution:

> An interpreted language (like some forms of BASIC) is translated line-by-line as the program executes. If you made a syntax error in your program code, the interpreter terminates the program and reports the error to you. Usually you have to correct the error before you can rerun the program.
>
> A compiled language (like Glossary), waits until you have typed the entire program and then compiles it into a machine-readable form. Any errors in the program are reported to you after it is compiled (the glossary verification process). The compiled form is called the object program. The typed form is the source program. (How the object and source programs affect the glossary document is described in Chapter 13, Glossary Information for Operating System Users.)

The glossary program you write is compiled by the glossary verification process into a form that is executable within your text document. You can also execute a glossary program from a WORD ERA menu. This executable form is a **.gl** file. To understand the **.gl** file, you need to understand the structure of a WORD ERA document.

The Structure of a WORD ERA Document

A WORD ERA document actually consists of the three following files:

filename	The textual portion of a document
filename.dc	The history, statistics and page pointer information for the document
filename.fr	The formats, header page, footer page and work page for this document

This structure is not apparent from the WORD ERA Document Index, which displays only the base filename of the document. You don't need to be concerned about this fact, as WORD ERA treats all three files as one, for the purposes of document control. However, for purposes of glossary, it is helpful for you to understand what is happening "behind the scene."

When you compile a glossary document, a fourth file, the **.gl** file, is created. The **.gl** file is the compiled and executable portion of the glossary document.

filename.gl	The binary form of a glossary; only present if the document is a compiled glossary.

If you are interested in learning more about the .gl and WORD ERA document file structure, read Chapter 13.

As you learned in Part 1, the glossary compiler reports program errors to you on Page W of your glossary document. The glossary programming language has an extensive syntax error list. Chapter 13, Administering Glossary Programs, tells you how to troubleshoot (debug) your programs. Appendix E provides a list of error messages received from the compiler. It also gives you a list of error messages associated with glossary procedures.

HOW TO STUDY PART 2

This part gives you the fundamental knowledge you need to write Glossary programs. While the information is specific to WORD ERA Glossary, the principles apply to most computer languages. You'll be learning more than you realize!

Chapters 10 and 11 give you a detailed description of each Glossary function. When you want to know what a function does and how to use it, refer to Chapters 10 and 11. Functions are listed alphabetically and by usage.

After you develop a working familiarity with these chapters, you will know exactly where to turn for reference while you are writing your programs.

When new functions are introduced in program examples, they are briefly described in the context of the program. If you would like detailed information on any function, refer to Chapter 10.

Comment lines are deliberately omitted in some of the program examples to give you an opportunity to read and understand an uncommented program.

All of the entry examples in this and following chapters are actual glossary entries you can type and try. Typing and recalling some of the entries that interest you the most will help you quickly learn the principles of glossary programming.

Although you will learn best if you actually type the entries, you can save typing time by using the entries on the Glossary Diskette provided with this book. The entries for this chapter and Chapter 6 are in glossary document **gloss2a** on the Glossary Diskette. Chapter 14 tells you how to retrieve and use the glossary documents on the Glossary Diskette.

OVERVIEW OF GLOSSARY PROGRAMMING LANGUAGE ELEMENTS

The following list provides a brief description of the elements of the glossary programming language you will learn in Chapters 6 through 9.

Statements

A statement is a declaration of purpose. A programming statement may be either a single keyword or a whole series of words consisting of variables, keywords, functions, and strings. Statements belonging to conditional and control statements must be enclosed in braces { }. Other types of statements do not need braces.

Variables

Variables are names you assign to store alphabetic or numeric strings for reference and manipulation. The content of a variable is called its value. The variable name can be almost anything you wish. All variables that you use in your program must be declared (given a name) and initialized (given a value).

Values

Variables and functions that contain values can return these values throughout the execution of a program. For example, you can use the **word** function to return the word at the cursor location in the text document. You assign the value of the **word** function to a variable, then you can type that word elsewhere in the document by feeding the variable to the document. Entry a in the Programming Style section in this chapter illustrates this principle.

Logical Values

Logical values of true or false allow you to check for true or false conditions in the document. For example, you can test for a true or false cursor condition by using this statement: **if(top_page)**. If the cursor is at the top of the page, the value will be true; if is not at the top of the page, the value will be false.

Relational, Equality, and Logical Operators

Relational and equality operators, such as > (greater than), == (equal to), and >= (greater than or equal to), allow you to compare two values.

Logical operators, such as & (and), | (or), and ! (not), allow you to apply the logic principles of Boolean algebra to glossary programming.

Assignment Operators

The assignment operator = allows you to assign a value to a variable. Mathematical assignment operators, such as += or -=, are used to perform mathematical operations on variables.

Functions, Arguments, and Expressions

The Glossary programming language has a built-in library of functions that can detect the cursor's location, prompt the operator for information, read text from a document, manipulate strings, call another entry as a subroutine, and even interact directly with the operating system.

Functions have "arguments" that may contain one or more "expressions." For example, the function **posmsg** has an argument in parentheses with three expressions separated by commas as in **call posmsg(2,12,"hello")**. This statement will place the word "hello" (expression 3) on the document edit screen at line 2 (expression 1) and position 12 (expression 2).

Conditional Statement Functions

Conditional statement functions such as **if** and **while** allow the glossary entry to make decisions based on document conditions. For example, the statement

```
if(end_doc) {goto "1" execute exit}
```

bases its decision on whether or not the cursor is at the end of the document. If it is, the cursor is sent to page 1

```
(goto "1" execute)
```

and the glossary program terminates

```
{exit}.
```

Control Statement Functions

Control statement functions such as **call** and **jump** change the order of statement execution. Using the example for conditional statements, you could have your program call another glossary entry (in the same glossary document) as a subroutine by writing the statement: **if(end_doc) {goto "1" execute call a}**. Program execution control is transferred to glossary entry a by the **call** function.

Labeled Statements (Identifiers)

A word enclosed in brackets and followed by a statement or statements may become the destination of a **jump** control statement. For example, the statement "**jump** counter" will cause the program to continue execution at the statement following the identifier **[counter]**.

Braces { }

In addition to beginning and ending an entry, braces are also used to begin and end bodies of conditional or control statements within the body of the entry.

Brackets [ ]

Brackets are used to enclose the identifying word for labeled statements.

Parentheses ( )

Parentheses enclose arguments and expressions. For example, in the statement **prompt("Enter Date")**, the text string "Enter Date" is the expression and is enclosed in parentheses as the argument to the **prompt** function.

String Operations

String functions allow you to perform a variety of operations on strings. For example, you can select and use specific parts of strings, you can substitute one part of a string for another, and you can concatenate two strings into one string. Strings can be assigned to variables or they can be used as expressions within function arguments. Mathematical calculations can be performed on numeric strings.

String functions are used extensively in Records Processing control glossaries. The *WORD ERA Records Processing* manual provides many useful entry examples that use string functions.

Mathematical Operations

Mathematical operations, such as addition, subtraction, multiplication and division, can be performed on numeric strings.

PROGRAMMING STYLE

Using the Glossary programming language, you can write long and complex programs. Longer programs are difficult to read unless you follow a specific style or convention. The style recommended here is the standard C language style that is adapted to glossary programming.

Glossary is a free-form language that doesn't care what style you use as long as your syntax is correct. However, using style conventions can assist you in writing, understanding, and reviewing your programs.

Entry a, An Example of Programming Style

The syntax of entry a in the following example is correct, but the logical execution of the program is very hard to follow. Also, since the compiler lists the line an error is on, it becomes difficult to pinpoint the error when the program runs together on one or two lines.

```
entry a{title=word command note goto"w" goto down call feed(title) goto
note}
```

Formatting helps to clarify the entry. Formatting means using spaces between the keywords, putting blocks of action on separate lines, indenting, and adding comments. Entry a is retyped in the following example. Notice how much easier it is understand the logical action of the program when returns, indents, and comments are added.

```
entry a
{
        title = word
        command note
        goto "w"
        goto down
        call feed(title)
        goto note
}
```

You can use entry a when you are typing or editing your document to create a word list on Page W. When you have finished editing, you can then copy Page W into a "word list document" to use with the Index Generator.

In entry a, the **word** function returns and assigns the word at the cursor location in the text document to the program variable **title**. The location of the cursor is marked by the keywords command note. The cursor is then sent to the bottom of Page W, where the value of **title** is typed by the statement, **call feed(title)**. The cursor is then sent back to the marked location in the text document.

The **feed** function feeds string values into the document as though they are being typed from the keyboard; **call** precedes a function when the function is used as a statement.

You will learn more about all the function shown in entry a in the following chapters.

Programming Style Conventions

Comments: Descriptive, well-placed comments make complex entries much easier to understand. Even though you are not familiar with all the functions used in entry a, you should be able to follow the logical sequence in the entry by reading the comment lines. Comments can also provide instructions or information for other people who are using your glossary entries.

Instructional comments at the beginning of a program can wrap for several lines. The commented paragraph must start with the symbols /* (foreslash and asterisk), and end with the symbols */ (asterisk and foreslash). When an explanatory comment following a program statement requires two lines use an Indent to wrap the lines or begin and end each line with comment symbols.

Be especially observant about enclosing your comment lines with the comment symbols. Also, be sure you do not accidentally include any program statements within the comment symbols.

The compiler will not tell you if a symbol has been omitted and some very unpredictable results may occur when you recall the program. Try using the interactive glossary entry in Chapter 13 for entering comments while you are typing a program.

Spaces: Be sure to put spaces between keywords, variables, and functions. You do not need to put spaces between a function and its argument. It makes no difference to the compiler if you write return (2) instead of return(2). However, eliminating the space clearly associates the parenthetical argument with its function, as in **feed(title)** in entry a.

Indenting: Indenting establishes a visible hierarchy of execution in a program and links statements with their functions. It helps you to see exactly where to place braces for the beginning of the entry and for function bodies.

Entry b is used to "de-center" top-of-page headings throughout a document and ignore centered headings that occur elsewhere in the document. The entry performs recursively by using the labeled statement **jump** [loop] to repeatedly execute its statements until it reaches the end of the document. See if you can understand the logical action of the entry before you read the description following the example. Note that indenting clearly establishes and helps you to understand the sequence of program execution.

```
entry b
{
     [loop]
         if(end_doc)
         {
              exit
         }
         goto center
              if(top_page)
              {
                   delete execute
                   jump loop
              }
     jump loop
}
```

In entry b, three large blocks that form the structure of the entry are:

1. The beginning and ending braces around the body of the entry.

2. The [loop] and **jump** loop block. This loop keeps executing all the **if** tests and instructions between **[loop]** and **jump** loop until the first test, **if(end_doc)**, proves true.

   When the cursor reaches the end of the document the **exit** statement following **if(end_doc)** is executed and the program terminates.

3. The braces that surround each **if** function body. The function body for the first **if** test is the single statement **exit** which terminates the program if the cursor is at the end of the document.

   The second test **if(top_page)**, is comprised of the statements delete execute **jump** loop, which delete the center symbol if the cursor is at the top of the page.

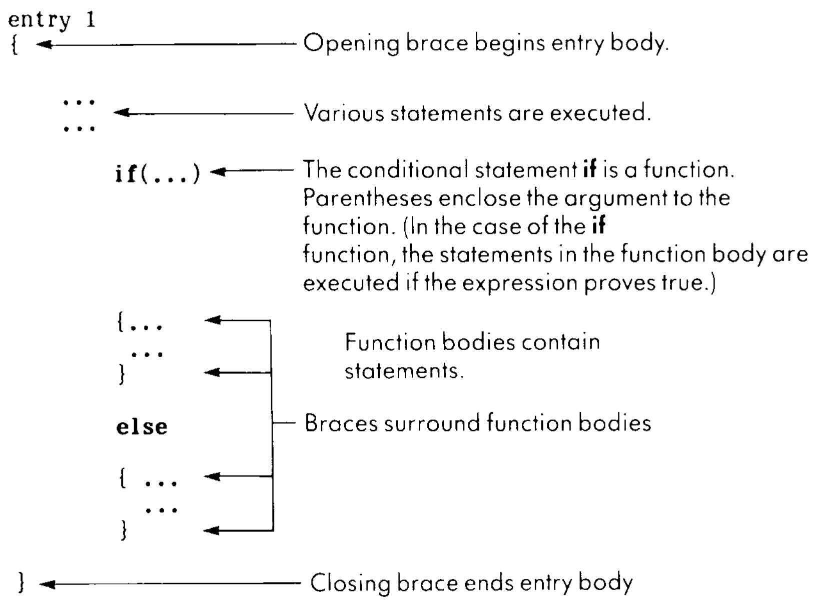
Indenting provides steps through your glossary program. This is helpful when you write the program, but more helpful a few weeks later when you read it again and try to remember why you wrote it.

Braces: Braces are used to begin and end function bodies. Although it is not necessary to use braces for only one statement, in the interest of consistency and good programming practice, it is recommended that you enclose all function bodies in braces.

Figure 5 shows how braces are used with function bodies. The figure uses the **if else** function.

This function is described in Chapters 7 and 8. Dots (...) represent omitted program statements.

Figure 5 Using Braces to Enclose Function Bodies


entry 1
{
Opening brace begins entry body.
...
...
Various statements are executed.
if(...)
The conditional statement **if** is a function. Parentheses enclose the argument to the function. (In the case of the **if** function, the statements in the function body are executed if the expression proves true.)
{...
...
}
Function bodies contain statements.
else
Braces surround function bodies
{ ...
...
}
}
Closing brace ends entry body

SYNTAX

Syntax is the order in which the glossary language must be written. This chapter tells you about general syntax usage. Chapter 10 gives you the specific syntax required for each function.

The following entry inserts "Tigera Corporation" in a document. To work properly the entry must be written in the correct execution order, or syntax.

```
entry c
{
    insert "Tigera Corporation" execute
}
```

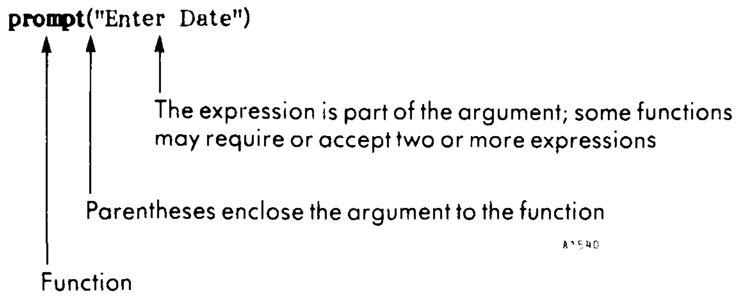
To insert a text string you must first invoke insert mode by pressing the Insert key, second, you type the string, and third, you press the Execute key to exit the insert mode. The entry would not work if you wrote it in a different syntax such as the one shown in entry C.

```
entry C
{
        "Tigera Corporation" execute insert
}
```

Type both entry c and entry C in a glossary document. Recall them in a text document and analyze the results. Entry C leaves you hanging in insert mode because there is no execute keyword following the insert.

Figure 6 shows the standard function syntax for arguments and expressions, using the **prompt** function as an example. You will learn about arguments and expressions in the Chapter 6.

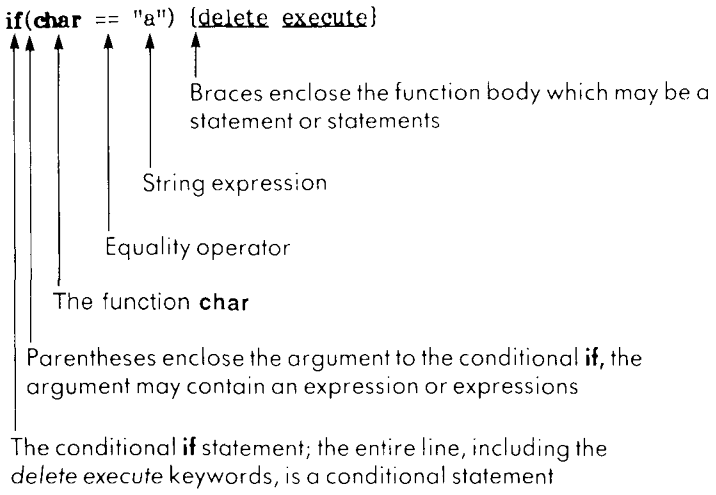
Figure 6 The Syntax for Functions, Arguments, and Expressions


prompt("Enter Date")
The expression is part of the argument; some functions may require or accept two or more expressions
Parentheses enclose the argument to the function
Function

Another type of syntax structure is shown in Figure 7, which illustrates the syntax for a conditional statement. You will learn about conditional statements in Chapters 7 through 9.

The specific syntax requirements for functions, conditional statements, control statements, and parenthetical expressions are explained in text, examples, and diagrams in the following chapters.

Figure 7 The Syntax for a Conditional Statement



```
if(char == "a") {delete execute}
```

Braces enclose the function body which may be a statement or statements

String expression

Equality operator

The function **char**

Parentheses enclose the argument to the conditional **if**, the argument may contain an expression or expressions

The conditional **if** statement; the entire line, including the *delete execute* keywords, is a conditional statement

SUMMARY

You will find your glossary programs easy to type, verify, correct, and use if you take the time to apply the programming principles described in this chapter.

Chapters 6 through 9 use many glossary entry examples to illustrate all the glossary programming elements. You can type these entries in your glossary document and recall them in a text document to see how they work.

Or, to save time, you can use the appropriate glossary document on the Glossary Example Diskette enclosed with this Guide. The glossary document **gloss2a** contains all the entry examples for Chapters 5 and 6. The glossary document **gloss2b** contains entries for Chapters 7 through 9. There are similar glossary documents for Parts 1, 3, 4, and 5. The Glossary Diskette is described in Chapter 14.

CHAPTER 6

ELEMENTS OF GLOSSARY PROGRAMMING

In this chapter you will learn how to use the following glossary programming elements:

Statements

Variables

Values

Logical Values

Operators

Functions, Arguments, and Expressions

Parentheses

The labeled entries in this chapter are example glossary programs you can type in a glossary document and recall in a text document. As you learn a programming element you should try incorporating it into some of your earlier glossary entries or write new entries using the entries in this chapter as guidelines.

If you want to save typing time, all labeled glossary entries in this chapter are in glossary document gloss2a on the Glossary Diskette provided with this book. Chapter 14 tells you how to use the Glossary Diskette.

STATEMENTS

A glossary programming statement can be a single keyword or a whole series of words consisting of keywords, variables, functions, and strings.

Types of Statements

Table 1 shows several types of glossary program statements.

TABLE 1 Examples of Statement Types

STATEMENT	TYPE OF STATEMENT
return	Keyword statement
insert "x" execute	Keyword statements
cost = 27.32	Assignment statement
call prompt("Enter Name")	Function call statement
if(char == "x") {delete execute}	Conditional statement
do {right}while(char != "x")	Conditional loop statement
jump loop	Control statement
[loop] goto "e" execute	Labeled statement

Single and Multiple Statements

Multiple statements to conditional statements must be enclosed in braces { }; a single statement does not require braces (however, enclosing all conditional function statements in braces clearly identifies the relationship of the statement to the conditional function).

Entry d contains examples of both single and multiple statements to conditional functions:

> The first conditional statement, **if(globerr) exit**, has the single statement, **exit**, which is not enclosed in braces.
>
> The second and third conditional **if** statements have multiple statements which are enclosed in braces.

```
entry d
{
     [repeat]
          goto indent
               if(globerr) exit
          right
               if(char == "o")
               {
                    goto indent
                    jump repeat
               }
               if(char != "o")
               {
                    insert
                         "o" indent
                    execute
                    jump repeat
               }
}
```

Note that entry d assumes the standard bullet format for indented items to be: Indent o Indent, since the lowercase "o" is most frequently used for bullets on impact printers. If you are using a laser printer you can substitute the laser printer bullet code for the lowercase "o." Also, indented items must begin with characters other than "o" for the entry to work properly.

The **globerr** function is used in entry d to perform a graceful exit from the program if the statement goto indent does not find an indent. The **globerr** function is described in Chapter 8, in the section Trapping Function Errors Using the **globerr** Function.

Statement Execution Order

Statements in a glossary program are executed in a top-down order, beginning with the first statement after the opening brace and ending at the last statement before the closing brace. As you can see from entry d, you can modify the execution order of an entry by using conditional and control statements like **if** and **jump**. Chapters 7 through 9 tell you how to control the execution sequence of your programs.

VARIABLES

An important feature of Glossary is the capability of storing a value and recalling it as a constant or of changing it as the program runs. A value may be a numeric string, an alphabetical string, or a mathematical expression. The storage location for the value is called a variable.

Declaring and Initializing Variables

Each variable you are going to use must be declared and initialized in your program by giving it a name and assigning it an initial value. It is important that you initialize each variable either to 0 (zero) or to an initial value the first time you use it in your program. Doing this resets the variable each time you use the program, or at each iteration of a program loop.

You can initialize all variables at the beginning of your program or immediately prior to their use. Analyze the entry examples in this book and note where the variables are initialized.

For example, entry e initializes the variables **ourcost** to a constant value of 64.25, **markup** to 0, and **theircost** to 0, then uses the **keys** function to assign a value to **markup**. (The **keys** function pauses the program during execution and allows you to enter data from the keyboard.)

The variables **overcost** and **markup** are added, and the result is assigned to **theircost**, which is typed in the document. The equal sign (=) following the variables is an assignment operator, it assigns the value to the variable. You will learn about assignment and other types of operators in this chapter.

```
entry e
{
        ourcost = 64.25
        markup = 0
        theircost = 0
        markup = keys
        theircost = ourcost + markup
        call feed(theircost)
}
```

Initializing variables to 0 at the beginning of a program is not strictly necessary, but it is a good programming habit to acquire and can be very important when you use programming languages other than glossary.

Variable Names

Variable names may be as many characters as you wish. Using short names keeps your program concise.

These are the rules for variable names:

A variable name cannot be the same as a glossary reserved word name. Glossary reserved words are the names of functions and keywords. See Appendix A for a list of reserved words and symbols.

Two word variable names must be joined by an underbar (_) or a period (.). Joining two-word variables by a period is a good way to distinguish them from two-word function names (which are joined by an underbar) like **end_doc** or **top_page**. You cannot use a space or a required space between two-word variable names. Spaces in any form are not allowed as any part of variable names. When you type the underbar, type SHIFT/Underline. Do not use Mode "_" (Mode Underline) to type the underbar.

The variable name must always begin with an uppercase or lowercase letter. It cannot begin with a number.

The variable name may consist of any combination of uppercase or lowercase letters or the numbers 1 through 9. The only symbols that may be used are the underbar (_), the period (.), or diacritical marks, such as the umlaut (··) or the grave (‘) and acute (’) accents.

VALUES

A value may be a string or a mathematical expression. Values are returned by functions or are assigned to and returned by variables. During program execution you may pass a value to a variable or function or cause a current value to be changed.

Functions that return values have a standard value type. For example, the **line** function always returns the line number of the current cursor position in the text document. The **date** function returns the system time and date. The **unixpipe** function returns the standard output of an operating system command.

Some functions return true or false values. These are called logical values. The **beg_doc** function returns a numeric value of 1 (true) if the cursor is at the beginning of the text document or 0 (false) if it is not.

Logical values are covered later in this chapter. You can refer to Chapter 5 for a description of the value returned by each function.

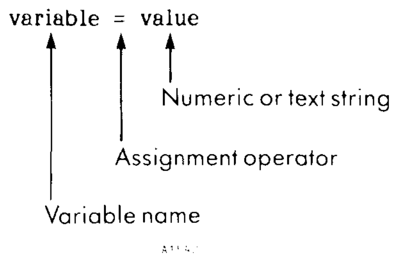
Assigning Values to Variables

When you assign a value to a variable, you must use an assignment operator. The standard assignment operator is the equal sign. This does not mean "equal to" but instead means "assign the value on the right-hand side of the = sign to the variable on the left-hand side of the = sign." (The equality operator is two equal signs ==, which means "equal to." Equality operators are described in the Operators section of this chapter.)

The syntax for assigning a value to a variable is shown in Figure 8.

It is valuable to remember the "right-hand, left-hand" definition. Mathematical assignment operators (which you will learn about later in this chapter) depend on the "right-hand side, left-hand side" assignment principle.

Figure 8 The Variable Assignment Syntax



The following are examples of different types of values assigned to variables. The variable names are arbitrary. You may use whatever names you choose.

```
figure_no = 0  month = "April"  cost = $44.37  month.end = 31

X1 = "2137A"  X2 = 22,370  count = 31  last.year = 86
```

The output of a function may also be assigned to a variable. The following example assigns the output of the **date** function to the variable **today**. (The **date** function returns the current system date and time.)

```
today = date
```

The current value of a variable can be assigned to another variable. For example:

```
month.end = cost    figure.number = count
```

Rules for Values

A numeric value is a number string, which may consist of the numbers 0 through 9 in any combination, and the dollar sign, the period, or the comma. Mathematical calculations may be performed on a numeric value. A numeric string does not need to be enclosed in quotation marks.

If numeric values containing commas are used as expressions in a function argument, they must be enclosed in double quotes. This rule applies to expressions in the form of variables or numbers.

An alphabetic value is a character string enclosed in double or single quotation marks, (") or ('). If you are embedding keyword abbreviations or octal numbers in your strings, use double quotation marks. (The use of octal numbers is described in Chapter 12 and Appendix C.) The character string may consist of any combination of letters, numbers, symbols, or keyword abbreviations.

Double quotation marks (") within the string must be embedded by using the keyword abbreviation \q.

Single quotes (') may be used to enclose strings. However, codes such as keyword abbreviations or octal representations are interpreted literally and are typed in the document.

Octal code numbers embedded in strings must be preceded by a backslash \. (See Appendix C for information about octal code numbers.)

Each keyword abbreviation symbol counts as one character in a quoted string. The string "\cTigera Corporation\r" has 19 characters even though the abbreviations are translated to a single screen symbol when the string is recalled in a text document. This is an important consideration if you are using string functions. (Examples of string functions using string counts are given in Chapter 11.) Other considerations in string character counts include the use of octal numbers and WORD ERA document control codes. (Appendix C describes the use of octal numbers and control codes.)

Mathematical calculations cannot be performed on an alphabetic string or on alphanumeric combinations. For example, the string "12th" is considered as an alpha string, not a numeric string.

LOGICAL VALUES

You can assign logical values to variables using the functions **true** or **false**. Entry f below illustrates one way to use logical values.

Entry f is a glossary program that types a memorandum form. It uses **false** as an argument to the **display** function to turn the editing screen display off while the form is being typed. The true function turns the display back on at the conclusion of the entry. Turning the display off during glossary execution causes the program to run faster.

```
entry f
{
     call display(false)                       /*turn display off*/
     "\cMEMORANDUM"                            /*center heading*/
         return(2)
     "DATE: " call feed(date)                  /*type system date and time*/
         return(2)
     "TO: John Brown"                          /*types who memo to line*/
```

```
          return(2)
      "FROM: Helen Smith"                          /*types who memo from line*/
          return(2)
      "SUBJECT: MEETING ON WEDNESDAY"              /*types memo subject line*/
          return(2)
      call display(true)                            /*turn display on*/
}
```

Functions used in entry f are **display**, **call**, **feed**, and **date**. The **display** function always refers to the current display on the document edit screen. Note that the **date** function is placed in parentheses as an argument to the **feed** function. The value returned by one function can become the value of another. The **call** function is a statement that transfers execution control to another function. A function is only preceded by **call** when it is used as a statement.

The **true** and **false** functions can be assigned to variables as shown below.

```
        today = true          yesterday = false
```

A true value returns the number 1, and a false value returns 0. Entry g below is an example that uses **true** and **false** functions with a conditional **if** statement. The entry is an interactive test that asks a question requiring a true or false answer.

```
entry g
{
      answer = 0

      "GLOSSARY TEST" return(2)
      "Enter 1 if your answer is true.  Enter 0 if your answer is false."
      return(2)
      [question1]
      "QUESTION 1:  A function can return a value." return(2)
      "ANSWER:  "
      call prompt("Enter 1 or 0: ")
      answer = keys
      call feed(answer)
      return(2)
      call clrpos(1,50,31)
```

```
        if(answer == true)
        {
                "Correct.  Most functions do return values, please refer to Chapter 5
                for a description of the value type returned by each function."
                return(2)
                exit
        }
        if(answer == false)
        {
                "Incorrect.  Most functions return values, please refer to Chapter 5
                for a description of the value type returned by each function."
                return(2)
                exit
        }
        if((answer != true) | (answer != false))
        {
                "The number entered is not 1 or 0, please re-enter your answer."
                return(2)
                jump question1
        }
}
```

The student enters 1 if the answer is true or 0 if the answer is false. The 1 or 0 is assigned to the variable **answer**. The value of **answer** is checked three times by **if** statements. If the answer is 1 (true) the "Correct" message is printed. If the answer is 0 (false) the "Incorrect" message is printed. If the student accidentally enters a number other than 1 or 0, a message is printed and the student is given another opportunity to answer the question.

Entry g uses the **if**, **prompt**, and **clrpos** functions, the equality operators == and !=, and the logical operator | (logical or). The **prompt** function displays a message in the prompt area of the screen. The message is whatever you type as a quoted string in the argument to **prompt**. The **clrpos** function clears a designated area of the screen. The screen area is defined by the expressions in the argument to **clrpos**. Expression 1 is the line number, expression 2 is the starting position, and expression 3 is the number of characters to be replaced with blanks.

The **if** function is covered in Chapter 7. Equality and logical operators are covered later in this chapter.

Functions That Return True or False Values: Many functions return a numeric value of true (1), or false (0). The **beg_doc** function, for example, returns a value of true if the cursor is on the first character in the document and a value of false if it is not. You can use these values by assigning **beg_doc** to a variable, as in the following example.

```
entry h
{
        whereindoc = beg_doc
        if(whereindoc == 1)
        {
              goto "e"
        }
}
```

When a function requires a logical interpretation of an argument, any nonzero value (equal to or greater than 1) returns a true value. A zero value in a logical interpretation is always false.

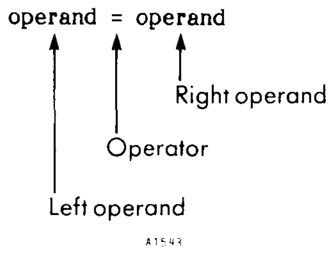
OPERATORS

Operators are programming symbols that assign values, perform math, determine the relationship of one value to another, assess equality, and designate logical operations.

Binary and Unary Operators

There are two basic types of operators: binary and unary. Binary operators require two operands, one to the left of the operator and one to the right of the operator, as show in Figure 9. Operand means "that which is operated on," and can be a variable, a function, or an expression.

Figure 9 **The Syntax for Binary Operators**



Unary operators require only one operand. There are two unary operators, the logical not (!) and the unary minus (-). Unary operators are placed to the left of the operand as shown in the following examples.

The logical not (!) performs logical negative operations as shown in the entry fragment below.

```
        if(!bot_page) {...}
```

If the cursor is not (!) at the bottom of the page, the statements represented by {...} are performed. Normally, the function **bot-page** returns a value of true if the cursor is on the page break line, and false if it is not. These values are reversed by the inclusion of the not operator **(!bot_page)**, a true value is returned only if the cursor is not at the bottom of the page.

The unary minus is an operator that takes the negative value of a number, as in **-z** in entry i.

```
entry i
{
     y = 200
     z = 50
     x = y * -z              /*x is assigned a value of -10000*/
     call feed(x)
}
```

The expression **y * -z** results in the negative value **-10000**, since the unary minus before the **z** converts the initialized value of **z** (50) to negative 50 (-50).

Assignment Operator

The section on variables in this chapter discussed the assignment operator =. This operator is used to assign a value to a variable, a value to a function, the output of a function to a variable, or the result of a mathematical operation to a variable or function.

In addition to the standard assignment operator =, there are mathematical assignment operators you can use. These math operators are discussed in this chapter under the section Mathematical Assignment Operators.

Mathematical Operators

You have a set of mathematical operators to perform addition, subtraction, multiplication, and division in your programs. These operators are shown below.

OPERATOR	FUNCTION
+ plus	Addition
- minus	Subtraction
* multiply	Multiplication
/ divide	Division
% modulo	Yields remainder of division

Mathematical operations can be performed on numbers and numeric variables. Numeric variables can store the results of a calculation, as shown in the examples below.

```
entry j
{
        balance = $25.20 + $100.00
        call feed(balance)
}
```

In entry j, the numbers $25.20 and $100.00 are totaled and the result, $125.2, is placed in the variable **balance**. You can do subtraction, multiplication, and division in the same way, as shown in entry k.

```
entry k
{
        top     = 190 - 3.2
        bottom  = 54 * 2
        percent = (top - bottom) / 100
        call feed(percent)
}
```

A numeric variable can be used anywhere a number can. Consequently, **top** and **bottom** can be used in a mathematical expression such as the one shown in entry k.

Parentheses are used to ensure that the value of **bottom** is subtracted from **top** before division occurs. The order of calculation precedence observed by the mathematical operators is covered in this chapter under the section Precedence.

When you write glossary programs to perform arithmetic you can generally follow standard mathematical principles for parentheses. The sections Functions, Arguments and Expressions and Parentheses in this chapter provide more information on parenthetical syntax.

Notice that numeric values can appear on either side of a mathematical operator.

The Modulo Operator

The remainder of a division operation can be determined with the modulo % remainder operator, as shown below.

```
leftover = 10 % 3
```

The variable **leftover** contains the value 1, since 10 divided by 3 leaves 1 (the remainder).

Using Mathematical Operators with Variables

Entry I is used to number figure illustrations in a document. First, it searches the text document for the string "Figure ". When it finds the first "Figure " it inserts the number 1 after the space following "Figure ". It continues to search the document, and each time it finds "Figure " it adds 1 to the variable **figure_no** and inserts the incremented value. If the search fails the statement **if(globerr) {execute exit}** causes the program to terminate. (The **globerr** function is described in Chapter 8.)

```
entry I
{
        figure_no = 0

        [loop]
            search "Figure " execute
                if(globerr)
                    {execute exit}
            cancel
            right(7)
                figure_no = figure_no + 1
            insert
                call feed(figure_no)
            execute
        jump loop
}
```

In entry l, the value of the variable **figure_no** was increased by 1 at each repeat of the search loop. The syntax for that addition was

```
        figure_no = figure_no + 1
```

where the current value of the variable is increased by 1, and the resulting value is assigned to itself.

The math operation is performed on the right side of the assignment operator using the current value of **figure_no**. If **figure_no** has a value of 10, the addition of **figure_no + 1** produces the sum 11. This sum is now assigned to the variable on the left side of the assignment operator, so **figure_no** now has a current value of 11.

The assignment operator allows a variable to perform a mathematical calculation on itself and reinitialize itself with the result. This is why you can't simply say **figure_no + 1**. You must use the assignment operator =, as in **figure_no = figure_no + 1**. (You can use mathematical assignment operators to shortcut the syntax. These operators are described in the next section, Mathematical Assignment Operators.)

The two uses of the assignment statements to the variable **count** in entry m produce two different results. In the first assignment **count** is declared and initialized to 0. In the second assignment, 1 is added to the value of **count**, and the result is assigned to **count**.

```
entry m
{
     count = 1                                   /*count has a value of 1*/
     [loop]
          count = count + 1                      /*count has a value of 2*/
     call feed(count) return
     jump loop                                   /*count will be increased by 1
                                                 for each loop repeat*/
}
```

Entry m is an endless loop. If you try it, the entry will continue to write numbers in your document until you press the Cancel key to terminate the glossary entry. Chapter 7 shows you how to break endless loops with conditional statements.

Other mathematical operations can be performed the same way as shown in the following entry examples. Remember, the numeric variable must be declared and initialized to 0 or an initial value in the program prior to the math calculation.

```
entry n
{
        linenumber = 8
        linenumber = linenumber - 4          /*linenumber now has a value
                                             of 4*/
        call feed(linenumber)
}


entry o
{
        cost = 35
        markup = 10
        cost = cost * markup                 /*cost now has a value of 350*/
        call feed(cost)
}


entry p
{
        average = 472
        average = average / 16               /*average now has a value of
                                             29.5*/
        call feed(average)
}
```

Mathematical Assignment Operators

Mathematical assignment operators provide a shortcut for calculation assignments by performing the calculation and the assignment in one statement. The mathematical assignment operators are shown in the following list.

OPERATOR	ASSIGNMENT	FUNCTION
+=	plus	Addition
-=	minus	Subtraction
*=	multiply	Multiplication
/=	divide	Division
%=	modulo	Yields remainder of division

In the previous examples, a calculation was performed on a variable and the result assigned to the variable by using the syntax below

```
        figure_no = figure_no + 1
```

Using the mathematical assignment operator +=, the same addition to **figure_no** is achieved with fewer keystrokes, as show in the following example.

```
        figure_no += 1
```

The += operator adds the value on the right to the value of the variable on the left, then stores the result in the variable. Examples using all the mathematical assignment operators are shown below.

```
entry r
{
     silo.storage = 1375000
     current.crop = 478245
     silo.storage += current.crop
     potato.surplus = silo.storage
     call feed(potato.surplus)
}

entry s
{
     cost = 24
     if(cost == 24)
     {    cost -= 14
          call feed(cost)
     {
     else
     {
          "The value of cost is not 24"
     }
}

entry t
     {
          headcount = 12740
          ticket.cost = $15
          headcount *= ticket.cost
          gate.receipts = headcount
          call feed(gate.receipts)
}
```

```
entry u
     {
          performers = 5
          gate.receipts = 191100
          gate.receipts /= performers
          divideup = gate.receipts
          call feed(divideup)
}


entry v
     {
          volunteers = 17
          gate.receipts = 191100
          gate.receipts %= volunteers
          charity = gate.receipts
          call feed(charity)
     }
```

Remember, when using mathematical operators, the number source for the calculation is on the right, and the storage destination for the result is on the left. The current value on the left is changed by the operation; the value on the right remains the same.

Use the standard assignment operator when the result of a calculation on a variable is assigned to another variable. Entry w is an example of this type of calculation. The variable **avgsales** is multiplied by 12, and the standard assignment operator is used to assign the result to **forcast**.

```
entry w
{
     avgsales = 121,334
     forcast = avgsales * 12
     call feed(forcast)
}
```

When you try this example, note that the comma in the value assigned to **avgsales** has no effect on the calculation of **avgsales * 12**. The value in **forcast** does not contain commas. The **pic** function is used to place commas, periods, and dollar signs in function and variable number values, which are then typed in a document. Entry x uses the **pic** function to format the value in **forcast** with a dollar sign and a comma.

```
entry x
{
      avgsales = 121,334
      forcast = avgsales * 12
      call feed(pic(forcast,"$,"))
}
```

The syntax for the **pic** function is

> **pic**(expression1,"expression2").

The number in expression 1 is formatted with the symbols in expression 2. The symbols must be in quotes. Expression 1 and expression 2 are separated by a comma. Expression 2 can contain one or more of the following symbols:

> **dollar sign "$", plus sign "+", minus sign "-", comma ",", period "."**

Refer to Chapter 10 for a complete description of the **pic** function.

Relational Operators

The relational operators are shown in the following list.

OPERATOR	FUNCTION
>	**greater than**
<	**less than**
>=	**greater than or equal to**
<=	**less than or equal to**

Relational operators relate one value to another, asking such questions as

> **Is cost greater than saleprice?**
> **Is temp less than 35?**
> **Is total greater than or equal to 12444?**
> **Is char less than or equal to "a"**

You write these questions in your program by using relational operators in the syntax shown in Table 2.

TABLE 2 Syntax and Examples for Relational Operators

SYNTAX	EXAMPLE
VARIABLES AND FUNCTIONS:	
variable operator variable	if(cost > saleprice) {...}
variable operator function	while(docpage < page_no) {...}
function operator variable	do {...} while(line >==line.no)
function operator function	if(number=< (min(e1,e2,e3,...)) {...}

SYNTAX	EXAMPLE
STRING EXPRESSIONS:	
string operator variable	if("A" > letter) {...}
string operator function	while(22 < number) {...}
variable operator string	if(zipcode >== 94401) {...}
function operator string	do (...) while(char <== "m")

As illustrated in Table 2, numbers and letters may be substituted for the functions and variables on the right side of the operator. Entries y, z, B, and D give examples using numbers on either side of the relational operator. The section following the entries tells you about relational operators and alphabetic strings.

```
entry y
{
        cost = 3366
        if(cost < 3368) {cost += 50}
        call feed(cost) return
}
```

```
entry z
{
      x = 1
      if(x < 2) {jump z}
      [z] "This is a z jump"
}
```

Note that entry z jumps to an identifier in brackets [z] and executes the statements following the identifier. (You can insert a statement like entry z in your glossary entry to verify that the entry is performing as you think it should be performing at a particular place in its execution.)

You will learn more about using identifiers and the **jump** statement in Chapter 8.

```
entry B
{
      if(page_no >= 2)
           {
           insert
                 "\cFigure " return(10) page
           execute
           }
}


entry D
{
      if(line <= 12)
      {
      insert
           return(5)
      execute
      }
}
```

None of the function statements enclosed in braces will be executed by the glossary program unless the condition specified by the relational operator is true. Entry B and entry D use the document reading functions **page_no** and **line**. The **page_no** function reads the text document during program execution and returns the current page number where the cursor is located. The **line** function returns the current cursor line number.

Using Relational Operators with Alpha Strings

You can use all of the relational operators with alphabetical strings. Just as the computer interprets true and false values as 1 and 0, it interprets alphabetic, numeric, and symbol characters as numbers. Each character has a number equivalent that can be represented as either an octal or a hexadecimal number. Appendix C contains a table of characters with octal number equivalents.

Since characters can be represented numerically, they can be ranked and collated numerically. The majority of computers in the U.S. use the ASCII (American Standard Code for Information Interchange) collating sequence for characters. (The table in Appendix C is arranged in the ASCII collating sequence.) This means you can write a glossary program that can select only those strings whose beginning characters collate higher or lower than a specified character. An example is shown in entry E.

Entry E is deliberately uncommented. To try the entry perform the following steps.

1. Type entry E in your glossary document and verify it. (Or use entry E in the glossary document *gloss2a* on the Glossary Diskette.)

```
entry E
{
        [loop]
             if(end_doc)
                  {exit}
        command note
             if(char >= "a" )
             {
                  insert tab(2) execute
                  down
                  goto left
             jump loop
             }
             else
             {
                  insert tab execute
                  goto left
```

```
                    copy
                        return execute
                        goto "w"
                        goto down
                    execute
                    goto note
                    down
                    goto left
                    jump loop
                }
}
```

2. Create a text document and type the following list on page 1 of the document. Begin each word at the left margin and type a RETURN at the end of every word. Be sure you capitalize the words exactly as shown on the list.

```
Bicycle
gears
tires
handlebars
Pencil
eraser
lead
Computer
cpu
console
keyboard
```

Edit the document and place the cursor on the first character in the first word in the list. Be sure your glossary document is attached, then press the GL key and type your entry label.

Entry E formats a list into hierarchical order by placing a tab before each word that begins with an uppercase letter and copies the word to page W (the workpage). It places two tabs before each word that begins with a lowercase letter.

Look at the ASCII table in Appendix C, notice that uppercase letters have a lower number equivalent than lowercase letters. This is why the statement **if(char >= "a")** places a tab before lowercase letters.

The functions in entry E are the **char**, **exit**, and **end_doc** functions. The **char** function returns the character at the cursor location in the document. The **exit** statement causes the entry to immediately terminate. Any statements following the **exit** statement will not be executed. The **end_doc** function returns a value of true if the cursor is at the end of the document, and false if it is not. When the value is true, the statements (in braces) to the conditional **if** are executed.

Entry E illustrates an interesting combination of keywords and functions in a glossary program. It takes advantage of page W as a place to store items during program execution. It also uses command note and goto note to mark its place in the document and return to that place. These are valuable features to remember when you are planning a program.

When you are writing a program like entry E, it is important that you carefully consider the cursor position in the document at each step of the program execution. If you encounter any bugs during execution, print a hard copy of the entry.

Walk through the program by reading the entry and manually performing the action from the keyboard. You can quickly spot places where the cursor isn't where you thought it was supposed to be. Walking through a program in this fashion is always a helpful procedure when you are troubleshooting your entry.

Relational Operators and Alpha/Numeric Comparisons

When you use relational operators to compare two numbers they are compared according to their numeric value. When you compare a numeric value and an alphabetic value, they are compared according to their ASCII collating order.

For example, entry F compares two numeric values (provided **keys** entry is numbers only). The variable **buy.price** is assigned a value as a result of a numeric comparison.

Entry G, however, compares a numeric and an alphabetic value. The result is that **this.month** compares less than **month**, even though month 10 (October) comes after June.

The point here is to be sure you really want to compare an alphabetic value to a numeric value. In entry F, the value of **month** should have been 6 (June) to provide an accurate value comparison.

```
entry F
{
        stock = 10.25
        today.market = keys
              if(today.market < stock)
              {
              buy.price = today.market
              call feed(buy.price)
              }
        sell.price = today.market
        call feed(sell.price)
}


entry G
{
        month = "June"
        this.month = 10
              if(this.month > month)
              {
                      call feed(this.month)
              }
              else
              {
                      call feed(month)
              }
}
```

The ASCII collating sequence in Appendix C shows you the hierarchical ranking order of numbers, letters, and symbols.

Equality Operators

Equality operators determine if the value on the right is equal or not equal to the value on the left; == means "equal to" and != means "not equal to." Some examples are given below.

```
entry H
{
        grandtotal = 7836
              if(grandtotal == 7836)
              {
              insert
                     "Grand Total"
              execute
              }
}

entry I
{
        day = 29
              if(day != 1)
              {
              jump x
              }
        [x] "This is an x jump"
}
```

In entry H, **grandtotal** must have a value of 7836 before "Grand Total" can be typed in the document. In entry I, the **jump [x]** statement will be executed if **day** has a value other than 1.

Logical Operators

Logical operators perform logical operations on values. The logical operators are & (logical and), | (logical or), and ! (logical not).

At their most basic level in program execution, logical operators depend on whether a value is true or false. True is evaluated numerically as the number 1, and false is evaluated numerically as the number 0 (zero).

The Logical and (&) Operator

There is only one possible true condition for the & operator. Expressions on either side of the & operator must satisfy the true condition.

Entry J uses the logical & operator in a conditional **if** statement.

```
entry J
{
     ingredients = 0

     call prompt("Enter amount of apples: ")
     apples = keys
     call clrpos(1,50,31)
     "Number of apples: " call feed(apples) return

     call prompt("Enter amount of bananas: ")
     bananas = keys
     call clrpos(1,50,31)
     "Number of bananas: " call feed(bananas) return

         if((apples == 6) & (bananas == 2))
         {
              ingredients = apples + bananas
         }
         else
         {
              "Not the right amount of fruit for this receipt, you need 6 apples
              and 2 bananas"
              return(2)
         }

     fruitsalad = ingredients
     "Total apples and bananas in the fruitsalad: " call feed(fruitsalad)
     return(2)
}
```

If you typed and recalled entry J, you noticed that you had to enter 6 apples and 2 bananas. Because the logical & linked the two variables together, you could not enter 8 apples and 2 bananas.

The conditional **if** has one full expression that includes the logical & operator:

```
         if((apples == 6) &(bananas == 2))
```

and two subexpressions:

```
         (apples == 6)
         (bananas == 2)
```

The subexpression **(apples == 6)** is only true if its value is 6. The subexpression **(bananas == 2)** is only true if its value is 2.

The full expression **((apples == 6) & (bananas == 2))** is only true if both subexpressions are true. You could state this logically as: if apples == 6 and bananas == 2, therefore the expression is true, so execute the following statements (add apples to bananas and store the result in ingredients).

The statement **{ingredients = apples + bananas}** is only executed if both subexpressions are true.

Note that the full expression in entry J is contained in one set of parentheses. The two subexpressions are separated by the & operator and each have their own set of parentheses. The section Functions, Arguments, and Expressions later in this chapter tells you more about using parentheses.

The Logical or (|) Operator

There are three possible true conditions and one false condition for the | operator. Using the same example that was used for the & operator, you could construct entry K, substituting the | operator for the & operator.

```
entry K
{
      ingredients = 0

      call prompt("Enter amount of apples: ")
      apples = keys
      call clrpos(1,50,31)
      "Number of apples: " call feed(apples) return

      call prompt("Enter amount of bananas: ")
      bananas = keys
      call clrpos(1,50,31)
      "Number of bananas: " call feed(bananas) return
```

```
        if((apples == 6) | (bananas == 2))
        {
                ingredients = apples + bananas
        }
        else
        {
                "Not the right amount of fruit for this receipt, you need 6 apples
                and 2 bananas"
                return(2)
        }

     fruitsalad = ingredients
     "Total apples and bananas in the fruitsalad: " call feed(fruitsalad)
     return(2)
}
```

Since **(apples == 6)** is true if **apples** equals 6, and **(bananas == 2)** is true if **bananas** equals 2, any one of the following three conditions will prove true and execute the statement {**ingredients = apples + bananas**}.

This can be stated logically as:

> when apples == 6 and bananas == 2, the full expression is true.
>
> when apples == 6 and bananas does not == 2, the full expression is true.
>
> when apples does not == 6 and bananas == 2, the full expression is true.

The only possible false condition where {**ingredients = apples + bananas**} would not be executed is

> when apples does not == 6 and bananas does not == 2, the full expression is false, ignore the statement in { L and skip to the next statement.

The logical or is an either/or condition; one or the other may be true, both may be true, but neither may be false.

If you tried entry K, you noticed that you could enter any number for apples and a 2 for bananas; any number for bananas and a 6 for apples; or a 6 for apples and a 2 for bananas; and you received a total. However, if you entered 5 for apples and 7 for bananas, for example, you received a zero. The | operator provides more options than the & operator.

The Logical not (!) Operator

You have already had an introduction to logical not (!) in the section on unary operators and relational operators. Summarizing that introduction, logical not (!) requires only one operand on the right side. It reverses the normal true condition of the function, so that it returns a value of true only if the function is false. Some examples are

```
          if(!end_doc) {...}
```

If the cursor is not at the end of the document, perform the statements represented by {...}

```
          if(!top_page) {...}
```

If the cursor is not at the top of the page, perform the statements represented by {...}

Entry J and K assume you always want bananas in your fruit salad. If you are indifferent to bananas you can use entry L, where the combination of the **keys** function, the logical | operator, and the equality operator != give you the option of defaulting to bananas or selecting your choice of fruit.

```
entry L
{
      ingredients = 0
      apples = "6 apples "
      bananas = "2 bananas "

      call posmsg(20,15,"Entering \qapples\q or \qbananas\q defaults amount")
      call prompt("Enter amount and fruit: ")
          fruit = keys
      call clrpos(1,50,31)
      call clrpos(20,15,46)

                if((fruit == "apples") | (fruit == "bananas"))
                {
                      ingredients = cat(apples,bananas)
                      fruitsalad = ingredients
                      "FRUITSALAD INGREDIENTS:  "
                      call feed(fruitsalad) return(2)
                }

                else if((fruit != "apples") | (fruit != "bananas"))
                {
                        ingredients = cat(apples,fruit)
                        fruitsalad = ingredients
                        "FRUITSALAD INGREDIENTS: "
                        call feed(fruitsalad) return(2)
                  }

         }
```

Functions that are new to you in this entry are **posmsg** and **cat**. The syntax for **posmsg** is

posmsg(expression1,expression2,expression3).

The syntax for **cat** is

cat(expression1,expression2).

The **posmsg** function displays expression 3 at the line and position specified by expression 1 and expression 2. Expression 3 may be a numeric or alphabetical string, a variable, or a function that returns a value. The **cat** function concatenates (brings together) expression1 and expression2, providing one continuous string expression.

Tables of Operators

The tables on the next few pages provide a list of all the operators you can use in your programs.

TABLE 3 Relational Operators

Operator	Definition	Syntax Example	Explanation
<	Less than	if(total < 2254) {"debit"}	If total is less than 2254, type "debit" in document at cursor location.
>	Greater than	if(total > 2254) {"credit"}	If total is greater than 2254, type "credit" in document at cursor location.
<=	Less than or equal to	if(cost <(10) {cost += 2}	If the value of cost is less than or equal to 10, add 2 to cost.
>=	Greater than	if(percent >=2) {jump loop}	If the value of percent is greater than or equal to 2, jump to [loop].

TABLE 4 Equality Operators

Operator	Definition	Syntax Example	Explanation
==	Equal to	if(char == "X") {insert "XX" execute}	If character at cursor is equal to X, insert XX in the document at cursor location.
!=	Not equal to	if(char != "X") {delete execute}	If character at cursor is not equal to X, delete it.

TABLE 5 Logical Operators

Operator	Definition	Syntax Example	Explanation
&	Logical and	if((month == "Feb")& (day == 29)) {"leap year"}	If the value of month is Feb, and the value of day is 29, type "leap year" in the document at cursor location.
|	Logical or	if((name == "Joe") | (name == "Jane")) {call feed (name)}	If the value of name is Joe or Jane, type Joe or Jane in the document at the cursor location.
!	Logical not	if(!end_doc) {goto "e"}	If the cursor is not at the end of the document, go to the end of the document.

TABLE 6 Mathematical Operators

Operator	Definition	Syntax Example	Explanation
+	Plus (performs addition)	inventory = 27 + 114	Add 27 and 114 and assign a value of 141to the variable inventory (141 is the sum of 27 and 114).
-	Minus (performs subtrac-tion)	stock = inventory - sales	Subtract the value in sales from the value in inventory and assign the result to the variable stock.

TABLE 6 Mathematical Operators (continued)

Operator	Definition	Syntax Example	Explanation
*	Multiply (performs multipli-cation)	forcast = avgsales * 12	Multiply the value in avgsales by 12 and assign the result to the variable forcast.
/	Divide (performs division)	avgsales = sales83 / 12	Divide the value in sales83 by 12 and assign the result to the variable avgsales.
%	Modulo	sales.remainder = sales83 % 12	Take the remainder of sales83 divided by 12 and assign it to sales.remainder.
+=	Addition assignment operator	personnel += 4	Add 4 to the value in personnel and assign the result to personnel.
-=	Subtrac-tion assignment operator	personnel -= 2	Subtract 2 from personnel and assign the result to personnel.
*=	Multipli-cation assignment operator	expenses *= 12	Multiply expenses by 12 and assign the result to expenses.
/=	Division assignment operator	expenses /= 12	Divide expenses by 12 and assign the result in expenses.

TABLE 6 Mathematical Operators (continued)

Operator	Definition	Syntax Example	Explanation
%=	Modulo (divison remainder assignment operator	expenses %= 6	Divide expenses by 6, take the remainder and assign it to expenses.
-	Unary minus takes negative number of operand)	loss = 12 *- sales	Multiply the negative value of sales by 12 and assign the result to loss.

FUNCTIONS, ARGUMENTS, AND EXPRESSIONS

Functions

Glossary gives you a library of built-in functions you can call upon to perform standard operations in your glossary programs. Chapter 10 provides you with an alphabetical list of functions that describes the use of each function and its syntax. Chapter 11 groups functions by usage and gives function application examples for each use.

Functions can be used as statements or expressions. When you use a function as a statement, you must call it from the function library by using the **call** function, as shown below.

```
call prompt("Enter Date")
call feed(avgsales)
call feed(date)
```

The **call** function is not required when a function is used as an expression. In the last example shown above, the **date** function is used as an expression in the argument to **feed**. Since the **feed** function is used as a statement, it must be preceded by **call**.

Use of the **call** function is specified for each function that requires it in Chapter 10.

Functions operate on data: Functions can gather and return data to you during the execution of the glossary program. This data is the value in the function. For example, when the **date** function is called in your program, it sets its value to the system date.

```
today = date                    /*the value of date is set to
                                system date and time and
                                assigned to today*/
```

You can store the value of **date** in a variable (**today = date**) and use it elsewhere in your program.

You can have the value returned to you by calling the **feed** function to type the value of **date** in your document as shown in the example below.

```
call feed(date)
```

Functions perform operations: Two of these operations are performed by the **error** and **posmsg** functions. The **error** function displays a message in the error area of your screen. (The error area is at the bottom right of the screen. System messages such as No glossary appear there.) The **posmsg** function places a message on the screen at the location you specify.

For detailed information on functions refer to Chapters 10 and 11.

Arguments

Most functions require arguments. The expressions in parentheses following the **prompt** and **feed** functions are the arguments to those functions.

```
call prompt("Enter Date")
call feed(avgsales)
call feed(date)
```

The argument to **prompt** is the alphabetic string expression "Enter Date." The argument to the first **feed** function is the variable **avgsales**. The argument to the second **feed** function is the **date** function.

The function **date** does not require a parenthetical argument. Its argument is built-in because the only function it performs is to return the system date. Chapter 10 tells you which functions require arguments and how to use them.

Expressions

Values inside the parenthetical argument are expressions. Some functions can take several expressions, as in the previous examples for logical operators. In those examples, the argument to the **if** function contains a full expression in parentheses. The full expression has two subexpressions, each with its own set of parentheses. The example for logical & is repeated below.

```
if((apples == 6) & (bananas == 2))
```

Since Glossary allows you to use an expression anywhere a value is allowed, arguments can contain either mathematical or string expressions. A mathematical expression using the function **max** is shown below.

Note that multiple expressions in an argument must be separated by commas.

```
highest = max(110,a + b,227)
```

The function **max** evaluates its list of expressions and returns the highest number for its value. If the value of **a** is 85 and the value of **b** is 72, what value would **max** assign to **highest**? Try writing this example as an entry. Remember to declare and initialize **a** and **b** as variables.

Five different types of expressions are shown in the following example.

call prompt("Enter Date")	Alphabetical string expression
call feed(avgsales)	Variable used as an expression
call feed(date)	Function used as an expression
call clrpos(22,48,12)	Numeric string expressions
highest = max(110,a + b,227)	Math calculation as an expression

Some functions require more than one expression in their arguments. The **posmsg** function shown below requires three expressions. Multiple expressions in an argument are separated by commas. Variables or functions can also be used in most multiple expression arguments. The function descriptions in Chapter 10 show how many expressions are required for each function.

```
call posmsg(6,5,"Glossary in Progress")
```

Expressions with more than one part: In some functions, the argument takes only one expression, but the expression is split into parts. The **cursor** function is one example. Other functions requiring multiple part expressions are described in Chapter 10.

```
call cursor("2,10,27")
```

When the **cursor** function is called in a glossary program, the cursor moves to the page, line, and position numbers that are specified in the string expression in its argument.

The three numbers separated by commas in the example for the **cursor** function are not three separate expressions. They are parts of one expression that is enclosed in quotes.

Variables cannot be substituted for parts of an expression because each expression is considered as a separate argument to the function. One variable may serve as the expression. It must contain all parts of the expression as a quoted string. It is not quoted in the argument to **cursor**. An example is shown below.

```
a = "2,10,27"
call cursor(a)
```

Using Expressions with Cursor Movement Keywords: An added benefit the glossary programming language gives you is the capability of using expressions with keywords that take an argument. In Part 1 you learned how you can specify

```
up(12)
```

when you wanted to move the cursor up 12 lines. Or you could specify

```
tab(4)
```

when you wanted to type the tab symbol four times in your document.

You can also give an expression to the keyname, and it could be a variable. For example:

```
moveup = 12
up(moveup)

x = 6 + 4
tab(x)
```

It can even be a mathematical expression as in

```
up(3 * 3)
```

This book does not explore everything you can do with this feature. If you like to experiment, using expressions with keywords could provide you with some unique glossary programs. Keywords that take arguments are identified in Appendix D.

USING PARENTHESES

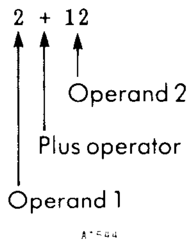
When more than one expression is part of an argument to a function, parentheses may be required.

Parentheses and Mathematical Expressions

Using mathematical expressions requires care. Mathematical operators follow rules of precedence when calculations are performed. Using parentheses correctly with math expressions helps you avoid calculation errors.

A mathematical expression is the combination formed by an operator and its two operands, as shown in Figure 10.

Figure 10 The Syntax of a Mathematical Expression



The result of a mathematical expression could be assigned to a variable without parenthesizing the expression:

```
X = 2 + 12                    /*X has a value of 14*/
```

If you add another expression you also need to add parentheses to be sure the calculations are performed in the correct order.

```
X = (2 + 12) * 14             /*X has a value of 196*/
```

In the example above, multiplication has a higher precedence order than addition and is performed first. If the addition expression were not enclosed in parentheses the result would be quite different, as shown in the example below.

```
X = 2 + 12 * 14                    /*X has a value of 170*/
```

Since the operation in parentheses has a higher precedence order than multiplication, the parentheses ensure that addition is performed first. Multiplication is performed second, and the result is assigned to the variable X.

The fully parenthesized form would look like this:

```
X = ( (2 + 12) * (14) )
```

Fortunately, this level of parentheses is not necessary because the glossary compiler knows what you mean by

```
X = (2 + 12) * 14
```

The precedence for mathematical operators is listed below in order from the highest to the lowest. Operators listed on the same line are equal in precedence. If the full expression has subexpressions with operators that are equal in precedence, calculations are performed from left to right.

TABLE 7 Precedence Order for Mathematical Operators

Operator	Definition	Order
()	Parentheses	First
-	Unary minus	Second
* / % *= /= %=	Multiplicative	Third
+ - += -=	Additive	Fourth
=	Assignment	Fifth

Parentheses and Relational and Equality Expressions

The precedence order for relational and equality operators ranks lower than that of mathematical operators. Relational and equality operators are of equal precedence. In arguments such as the following you don't need to parenthesize the subexpressions.

```
        if(a + b < a + c) {...}

        if(medflies > fruittrees) {...}
```

Of course, if you add more complicated math to the first expression you will need to add the appropriate parentheses around the subexpressions, as in the example below.

```
        if((a + b) * 4 < (a + c)) {...}
```

For lists and examples of relational and equality operators, see Tables 6-3 and 6-4 in this chapter.

Parentheses and Logical Expressions

Logical expressions have a lower precedence order than relational and equality operators. However, you will need to use parentheses with most logical expressions using the & and | operators because they usually contain several subexpressions as shown in the examples below.

Sometimes the syntax of a function requires parentheses around logical expressions. These cases are explained in Chapter 5.

```
        if( (a + b > a + c) & (d - g <( x - y) ) {...}

        if( (char == "z") | (char == "y") ) {...}
```

The logical not ! is a unary operator and is always included within the parenthetical expression. Some examples are given below.

```
        if(!end_doc) {...}

        if(!top_page) {...}
```

You will learn more about parentheses by reading and trying the program examples in the following chapters.

SUMMARY

When you begin to use the programming elements you learned in this chapter you may initially receive more verification errors than you normally do, or your programs may not execute the way you think they should. Some points to help you troubleshoot your programs are:

Be sure you have not used a reserved word to name a variable. Appendix A provides a list of reserved words and symbols.

If any mathematical operations do not seem to be calculating correctly, check your logic on a calculator. The compiler checks for syntax errors, it does not check for logic errors.

Also be sure you have used parentheses properly.

Logical operators can be tricky to use. If you're not familiar with the principles of Boolean algebra you can get a book on the fundamentals from the library or a textbook store.

Be sure you have provided the correct number and type of arguments to a function.

Remember, the problem is usually something simple like a missing quotation mark, a missing closing brace, or a misspelled keyword.

In Chapters 6 through 10 you will learn how to use conditional and control statements to perform loops, call subroutines, and to change the execution order of your programs.

CHAPTER 7

CONDITIONAL STATEMENTS

Program statements are always executed in a straight line from the top to the bottom of the glossary entry unless you change the execution order with a conditional or control statement.

Conditional statements such as **if**, **if else**, **while**, and **do while** change the execution order by evaluating conditions and making decisions. The **if** and **if else** conditional statements are described in this chapter.

The **while** and **do while** conditional statements are shown in examples in this chapter and described in detail in Chapter 9.

Control statements such as **jump** and **call** transfer execution control to a different part of the entry or to another entry. The **jump** and **call** statements are described in Chapter 8.

The **globerr** function allows you to exercise control by setting a trap for keyword function error conditions. You can gracefully terminate an entry at any point in its execution by using the **exit** statement. The **globerr** and **exit** statements are described in Chapter 8.

Correct placement of conditional and control statements within a program is essential. Write a few programs of your own using these functions as you study this chapter and Chapters 8 through 9. The glossary compiler gives you syntax errors; it doesn't check your logic. If an entry doesn't work the first time, your conditional tests or loop instructions are probably in the wrong place. Shift them around and try again. It may take an entry or two before the logic of conditional and control statements is completely clear. The majority of entry examples in this book use these functions.

CONDITIONAL STATEMENTS

The term conditional statement means the function, its arguments and expressions, and the statement or statements that are executed as a result of the conditional test.

Multiple statements to conditional functions are always enclosed in braces. Although you are not required to enclose single statements in braces, it is helpful to use braces because they distinguish conditional statements from other statements in the program.

The four conditional functions and their syntax structures are shown in the following list. The functions are: **if**, **if else**, **while**, and **do while**.

if

```
if(expression)
{
        statement or statements
}
```

if else

```
if(expression)
{
        statement or statements
}
else
{
        statement or statements
}
```

while

```
while(expression)
{
        statement or statements
}
```

do while

```
do
{
        statement or statements
}
while(expression)
```

General Principles for Using Conditional Functions

Conditional functions are your program decision makers. Decisions are based on the evaluation of three types of program runtime conditions:

Conditions in the text document during program execution

Conditions arising from interactive operator input during program execution

Conditions in the program during execution

These conditions are described in the following text and shown in entries a, b, and c.

The expressions in the argument to conditional functions specify the conditions for evaluation. If the expressions evaluate as true, the statements following the argument are executed. If false, they are skipped.

The following general usage principles apply to all conditional functions.

EVALUATING CONDITIONS IN THE TEXT DOCUMENT

When you are using a conditional statement to evaluate document conditions, you are evaluating the cursor position in the document. Look at the Document Reading Functions section in Chapter 11. Each of these functions is asking a question about the cursor location. Is the cursor at the beginning of the document (**beg_doc**)? What character is it on (**char**)? What line number is it on (**line**)? What is the vertical spacing of the current format line (**spacing**)? What is the exact page, line, and position location of the cursor in the document (**loc**)?

All document reading functions return a value that can be assigned to a variable, evaluated as a conditional expression, or used by a function.

Some functions return number values from the document status line. The **line** function, for example, returns the line number of the current cursor location. Your program works somewhat like you do; while you are editing a text document, you can glance at the status line at the top of the screen and tell which line number the cursor is on. The program also uses this information during its execution in the document.

Other functions, such as **beg_doc** and **left_margin**, return a logical true or false value. A true value is always returned as 1, and a false value is always returned as 0. If someone asks you if your cursor is at the beginning of the document while you are editing, you will respond "yes" (true) if it is and "no" (false) if it is not. The **beg_doc** function makes this same evaluation while the program is executing and responds with a true or false answer.

You can take another logical step with true/false functions by using the logical not (!) operator. The statement

```
while(!end_doc)
{
    counter +=1
}
```

increases the variable **vounter** by 1 as long as the cursor is not at the end of the document. The logical not (!) operator changes **end_doc** so that it only returns a value of true if it is not at the end of the document. Normally, **end_doc** returns a value of true if it is at the end of the document.

Entry a in this chapter uses two conditional **if** statements to evaluate document conditions during program execution.

EVALUATING INTERACTIVE OPERATOR INPUT

The interactive functions **key**, **keys**, **keyin**, and **keysin** allow you to enter data as the program is being executed. The entered data can be stored in a variable then evaluated and acted upon by a conditional statement.

When you use **key** and **keys**, the data is assigned to a variable. If you want the data to be typed in the document you can use the **feed** function with the **key** or **keys** variable as shown in the following entry fragment.

```
    name = keys
    call feed(name)
```

The **keyin** and **keysin** functions type the data directly into the document. If you want to store the data it must be read back from the document and assigned to a variable.

Which interactive method you choose for your program depends on the result you want to achieve. If you need to evaluate the data with a conditional statement, **key** and **keys** are the most direct functions to use.

Entry b in this chapter is an example that uses both methods of interactive data entry.

EVALUATING CONDITIONS IN THE PROGRAM

Values in variables are usually the internal program conditions that are evaluated during execution. As you have seen from several previous examples in this book, variable values can change as a result of mathematical calculations, reassignment of string expressions, or other factors. Your program can be written to continually evaluate a variable by using a conditional statement.

For example, entry A below types the value of **count** in the document until **count** reaches 80. The variable **count** is typed and incremented by the **do** statement and is continuously evaluated by the conditional **while** until it reaches 80.

```
entry A
{
    count = 2
    do
    {
        call feed(count) return
        count += 2
    }
    while(count <= 80)
}
```

Entry c in this chapter is an example that evaluates the same variable for two different program conditions.

CONDITIONAL STATEMENTS CAN CHANGE EXECUTION ORDER

Execution order of the program can be changed by the statements to a conditional function. Entries a, b, and c in this chapter illustrate this action.

EVALUATING WORD ERA SCREEN SYMBOLS

One of the document conditions you will want to evaluate is whether or not the cursor is under a screen symbol, such as a Return, Tab, or Center symbol. Asking these questions with a conditional statement involves using both WORD ERA document format codes and octal numbers. Appendix C gives you syntax requirements and examples for this type of evaluation.

The Conditional if Statement

The syntax for the conditional **if** statement is

```
if(expression)
{
    statement or statements
}
```

The argument to **if** may contain various combinations of expressions and operators, as shown in examples in this book.

Examples of the **if** statement are shown in entries a, b, and c in this chapter.

USING AN IF STATEMENT TO EVALUATE DOCUMENT CONDITIONS

Entry a is an example of **if** statements that evaluate conditions in the text document during program execution. This entry goes to the top of the next page and inserts format line 2. If there is no next screen (the end of the document), the program terminates. If the next screen is page 10, it inserts the string "This page intentionally left blank."

This entry only executes once. If you want to use it to reformat several pages, you will need to add a loop. The entry is rewritten to use a conditional **while** loop in Chapter 9. Entries b and c, which follow this entry, both use the **jump** statement to perform loops. Loops are covered in Chapter 8, under the section Looping.

```
entry a
{
     goto nextscrn
         if(globerr)
         {
              cancel execute
         }

     insert copy format "2" execute execute

         if(page_no == 10)
         {
              goto south
              insert page
              return(6)
              "\cThis page intentionally left blank\r"
              execute
         }

     call error("Entry Concluded")
     call prompt("Press Execute to Continue")
     call keyin
     call clrpos(1,50,29)
     call clrpos(25,51,28)

     }
```

Entry a is concluded gracefully by a series of prompts to the operator. The error message notifies the operator that the entry has concluded, the prompt message asks the operator to enter a keystroke to continue, the **keyin** function allows the operator to enter one keystroke, and the **clrpos** statements clear the error and prompt messages.

EVALUATING INTERACTIVE INPUT

Entry b is an example of an **if** test that evaluates conditions arising from interactive operator input during program execution. This entry interactively types an invoice in the text document.

Using the conditional **if** statement to perform a yes/no branch, the invoice is typed again on a new page if the operator enters a y or Y in response to the prompt "Invoice? Type y or n." A **jump** statement is used to repeat the loop and type the invoice again. Jump statements are covered in Chapter 8.

If the operator enters an **n** or N, the program terminates.

```
entry b
{
     [typeagain]

          "\cAMALGAMATED WIDGETS, INC.\r"
          "\cINVOICE\r"
          "NAME:  "
          call prompt("Enter Name")
          call keysin
          return
          "ADDRESS:  "
          call prompt("Enter Address")
          call keysin
          call clrpos(1,50,29)
          return(2)

          "Thank you for your patronage.  Your balance for widgets and goodies
          purchased through June 30 is: " return(2)

          tab(2)
          "AMOUNT DUE:  "
          call prompt("Enter Balance")
          call keysin
          call clrpos(1,50,29)
          return(2)
          insert
               page
          execute
          goto north
```

```
        call prompt("Invoice? Type y or n: ")
        answer = keys
        call clrpos(1,50,29)

                if((answer == "y") | (answer == "Y"))
                {
                        jump typeagain
                }

                if((answer == "n") | (answer == "N"))
                {
                        exit
                }
    }
```

Note the difference between the **keysin** function and the **keys** function in entry b. The **keysin** function allows the operator to enter an unlimited number of keystrokes. When all data is entered the operator must press EXECUTE to restart the entry. The data entered in response to **keysin** is typed directly in the document. The **keys** function also allows entry of unlimited keystrokes, however, the data entered is stored in a variable, not typed in the document. The value of the variable (data entered in response to **keys**) can then be compared by a conditional function as illustrated in entry d, or it can be typed in the document at any point by using the **call feed(variable)** statement.

Also note that entry b provides for lowercase or capital letter input by using the logical or | operator in the argument to the conditional **if**. The yes/no branch is a convenient programming device for many applications. Some of the more common uses are

Educational tests requiring yes/no answers
Queries to repeat a loop
Decision to call a subroutine (subroutines are covered in Chapter 8)

EVALUATING CONDITIONS IN THE PROGRAM

Entry c is an example of an **if** test that evaluates conditions in the program during execution. This entry types the numbers 1 through 50, each on a separate line. When the 50th line is typed, the cursor goes to the top of the page and the entry concludes.

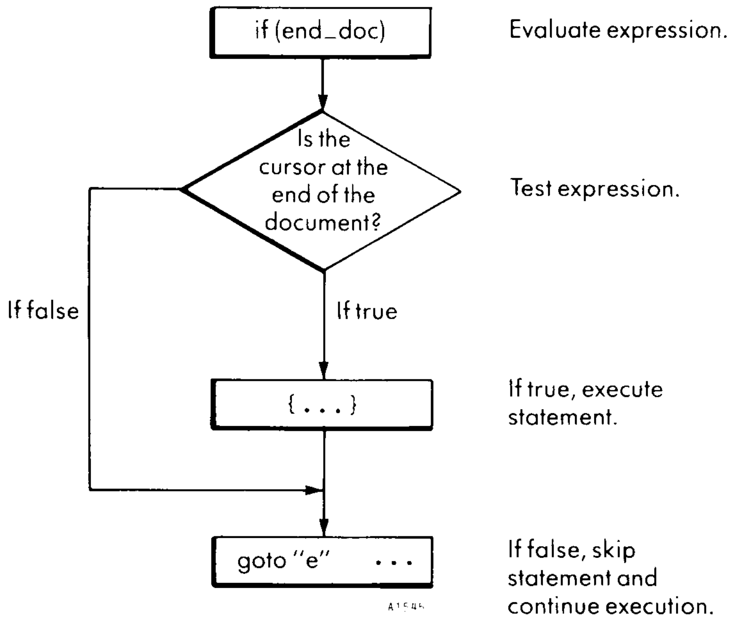
```
entry c
{
     linenumber = 0

     [typenumbers]
     linenumber += 1
         if(linenumber <= 9)
              {
                   "0"
              }

              if(linenumber > 50)
              {
                   return goto north exit
              }
          call feed(linenumber)
          return
          jump typenumbers
}
```

Many legal documents require line numbers, entry c can be modified for typing line numbers before existing text lines by adding the insert execute keywords. Try writing another program to perform this function. You can base it on entry c and add the appropriate keywords. First, create a text document and type the required 50 separate lines of text. Second, write the entry to insert a number before each line. Third, recall the entry and make sure it works.

USING FLOW CHARTS TO PLAN PROGRAMS

Figure 11 shows a typical flow chart used to diagram a program using the conditional **if** statement. Flow charts are a device you can use for pre-thinking your programs. You don't have to be formal with them, just sketch your programming ideas on a piece of scratch paper. Use squares for the action parts of the program and diamonds for the conditional decision-making sections. Draw lines to indicate the flow of execution through the program. Using flow charts to analyze your logic will make the actual program writing much easier than if you approached it cold.

Figure 11 Flow Chart Using the Conditional if Statement


if (end_doc)
Evaluate expression.
Is the cursor at the end of the document?
Test expression.
If false
If true
{ . . . }
If true, execute statement.
goto "e" . . .
If false, skip statement and continue execution.

The Conditional if else Statement

The **else** statement gives you an alternative statement to execute when the expressions in the argument to **if** prove false.

As you have seen in the previous examples, **if** can be used by itself as a conditional statement. The **else**, however, is dependent on **if** and can't be used alone.

Although there are two separate statement blocks in the **if else** structure (one for **if** and one for **else**), it is considered as a single conditional statement.

The syntax for a conditional **if else** statement is

```
if(expression)
{
     statement or statements
}
else
{
     statement or statements
}
```

The argument to **if** may contain various combinations of expressions and operators.

The execution of an **if else** statement follows this order:

> True Condition: When the **if**(expression) proves true, the **if {statement}** is executed, and the **else {statement}** is skipped.
>
> False Condition: When the **if(expression)** proves false, the **if {statement}** is skipped, and the **else {statement}** is executed.

The statements to **else** are always enclosed in braces.

Examples using **if else** are shown in entries d and e.

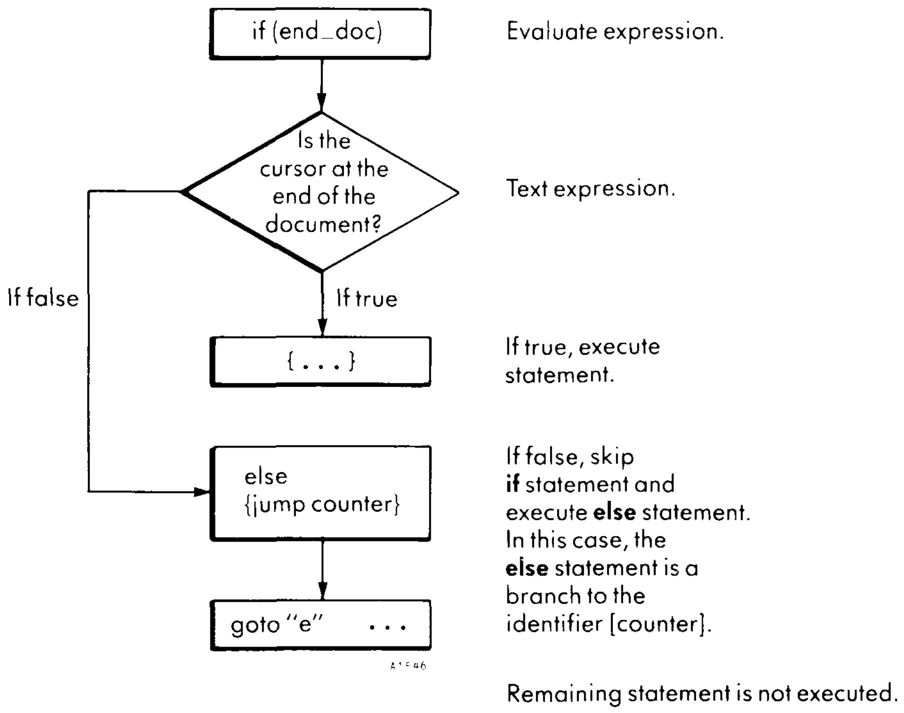
Figure 12 shows a sample flow chart for the **if else** statement.

USING FLAGS IN A PROGRAM

Entry d is a program to consecutively number figures and tables throughout a document. When the document was first typed, the operator used a glossary entry to place a flag before each figure and table heading. A flag is a symbol you put in a document for several programming purposes. The two most common purposes are:

> To provide a unique search string for the program. For example, the Table of Contents Generator (which is a program) searches for the symbol combination MERGE NOTE MERGE to extract headings. When you choose a flag for this purpose be sure and use a symbol or symbol combination that is not used elsewhere in the document. The merge symbol is a convenient character to use for flags because it isn't generally used in a text document. Also, you want to choose a flag that doesn't print.

Figure 12 **Flow Chart Using the Conditional if else Statement**


if (end_doc)
Evaluate expression.
Is the cursor at the end of the document?
Text expression.
If false
If true
{. . .}
If true, execute statement.
else {jump counter}
If false, skip **if** statement and execute **else** statement. In this case, the **else** statement is a branch to the identifier [counter].
goto "e" . . .
Remaining statement is not executed.

To provide a signal to terminate the program if the flag is encountered. This type of flag is typically used in conjunction with a conditional statement.

Though it is unlikely that a merge symbol will be encountered in the document, entry d uses **else** to ignore a non-flag merge symbol. Entry d is a universal type of program that could be used in many different documents. When you write a program like this it is always a good idea to provide a trap for other uses of the flag character in the document.

> **NOTE**: Merge symbols are used in Records Processing List and Format documents, so choose another flag symbol if you're writing glossaries to perform operations in these documents.

In entry d, the flag used for figures is <fx, and the flag used for tables is <tx, which are unlikely combinations to encounter in a document.

```
entry d
{
        figureno = 0
        tableno = 0
        [searchloop]
        search "<" execute
              if(globerr)
              {
                      execute exit
              }
        cancel
        right
              if((char == "f") | (char == "t"))
                      {
                              jump typenumb
                      }
              else
                      {
                              jump searchloop
                      }
[typenumb]
        if(char == "f")
        {
              figureno += 1
              goto left delete "x" execute
              insert
                      "FIGURE "
                      call feed(figureno)
              execute
        }
        else
        {
              tableno += 1
              goto left delete "x" execute
              insert
                      "TABLE "
                      call feed(tableno)
              execute
        }
        jump searchloop
}
```

Entry d searches for the left-facing merge symbol <. The **globerr** function is used to trap a search failure and terminate the entry. When it finds a <, it moves one character to the right. If the character is an "f" or a "t" the program jumps to the identifier [typenumb], otherwise it repeats the search by jumping to [searchloop].

At [typenumb], the character is checked again. If it is an "f," the variable **figureno** is incremented by 1, the flag is deleted, and the string "FIGURE " and the value of **figureno** are typed in the document.

If the character is not an "f" it has to be a "t" because the first **if** used the | operator to make sure the character was an "f" or a "t."

The **else** provides the statement to increment the **tableno** variable, delete the flag, and type the string and value of **tableno** in the document.

The final jump statement goes to [searchloop] and starts the search for the next < symbol.

CONSIDERING PROGRAM RUNTIME

The application performed by entry d could be a less complex program to write by using two separate entries. One would search for <fx, increment the **figureno** variable, delete the flag, and type the string and value in the document. The other entry would perform the same operations for <tx. However, this method would require two passes through the entire document. The runtime would be double the runtime for entry d.

NESTING IF AND IF ELSE STATEMENTS

When you nest statements, you put one statement inside another, rather like putting a series of smaller boxes inside bigger boxes. There are two ways to type nested **if else** statements in a program. Both methods perform the same way when the program is executed. Using method 1 or 2 is a style convention. Whichever method you choose, be consistent throughout your program.

Note that the **else** statement to the first **if** is another **if else** statement.

Method 1: One statement follows another. The second **if else** statement is indented to indicate that it is subordinate to the first statement.

Method 2: The following nested statement is called an **else if** structure. If it seems to be a clearer way of nesting than method 1, use it.

Method 1:

```
if(expression)
{
    statement or statements
}
else
    if(expression)
    {
         statement or statements
    }
    else
    {
         statement or statements
    }
```

Method 2:

```
if(expression)
{
    statement or statements
}
else if(expression)
{
    statement or statements
}
else
{
    statement or statements
}
```

Nesting your **if else** statements helps you to write tighter programs with fewer jump statements. You can nest as many **if else** statements as you need for making multiple decisions.

Entry e shows a more concise way to write entry d using nested **if else** statements.

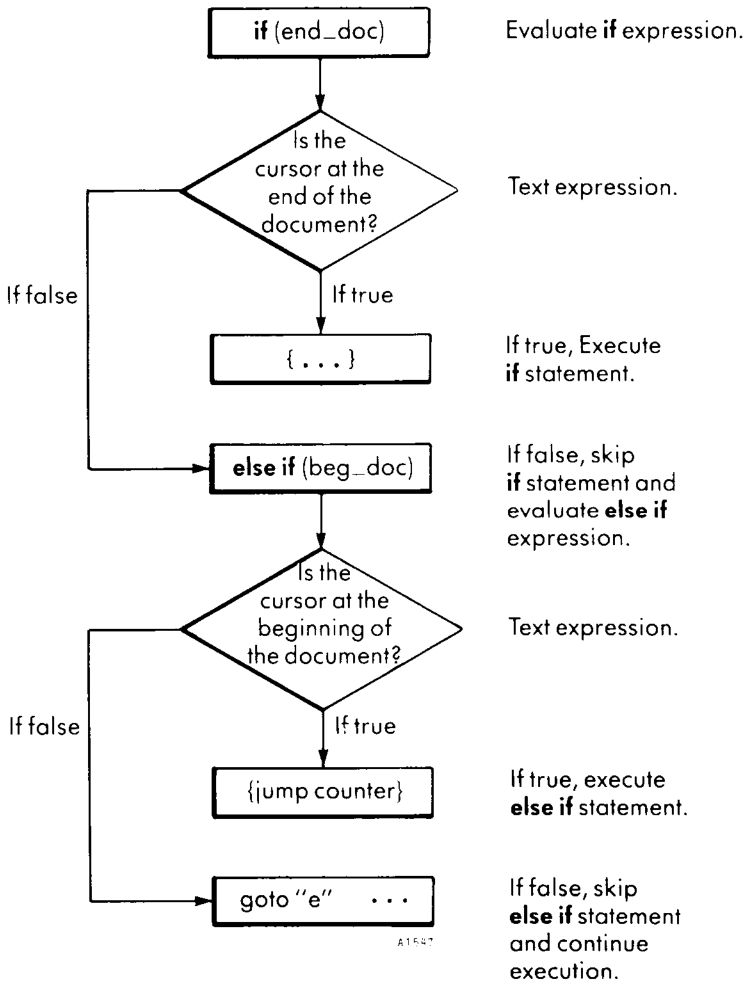
Entry d used two **if else** statements. One checked the character flag and jumped to [typenumb] or [searchloop]. The second **if else** at [typenumb] checked the character again and incremented **figureno** or **tableno**, depending on the character.

Entry e nests its statements using the **if else** structure. This method eliminates the jump to the identifier [typenumb].

```
entry e
{
     figureno = 0
     tableno = 0
[searchloop]
     search "<" execute
         if(globerr)
         {
              execute exit
         }
     cancel
     right
         if(char == "f")
         {
              figureno += 1
              goto left delete "x" execute
              insert
                   "FIGURE "
                   call feed(figureno)
              execute
         }
         else if(char == "t")
         {
              tableno += 1
              goto left delete "x" execute
              insert
                   "TABLE "
                   call feed(tableno)
              execute
         }
         else
              {
                   jump searchloop
              }
      jump searchloop
}
```

Figure 13 shows a sample flow chart for nested **if else** statements.

Figure 13 Flow Chart Using Nestled if and if else Statements


if (end_doc)
Evaluate **if** expression.
Is the cursor at the end of the document?
Text expression.
If false
If true
{ . . . }
If true, Execute **if** statement.
else if (beg_doc)
If false, skip **if** statement and evaluate **else if** expression.
Is the cursor at the beginning of the document?
Text expression.
If false
If true
{jump counter}
If true, execute **else if** statement.
goto "e" . . .
If false, skip **else if** statement and continue execution.
A1547

SUMMARY

In this chapter you learned how to use the conditional **if** and **if else** statements. In Chapter 9 you will learn how to use the conditional loop statements **while** and **do while**.

Remember, when you are using a conditional statement to evaluate conditions in the text document during program execution you are evaluating the cursor location. If your program is not working properly be sure you know where the cursor is (or is supposed to be).

A good way to check your program's action is to edit the glossary document, then create a window for the text document. Attach and run the glossary entry in the text document window. Using this method you can look at the entry and watch it run at the same time.

CHAPTER 8

CONTROL STATEMENTS

As you learned in Chapter 7, conditional statements make decisions in your programs. Based on those decisions, control statements can transfer execution control to another part of your glossary entry or transfer control to another entry in the same glossary document.

While frequently used with conditional statements, control statements are not dependent on them. Execution control can be transferred at any point in a program.

Glossary control statements are

call

glossary

jump

exit

globerr

The **call** and glossary functions transfer execution control to subroutines, (glossary is a keyword, not a function; however, it is included here because it performs as a control statement.)

The **jump** function branches to an identifier within the same entry.

When the **exit** function is invoked, it terminates the currently executing entry.

While not strictly a control function, the **globerr** function allows you to exercise control by setting a trap for keyword function error conditions.

SUBROUTINES

Subroutines are glossary entries that can be called and used by other glossary entries. They can be used two ways:

> As dependent programs that can only be used for a specific calling entry.
>
> As independent programs that can be called from several different entries in the same glossary document. (Entries w, x, and y, in this section are examples of independent subroutines. Entries w, x, and y are used as multiple-choice subroutines for entry f, but they could also be called by other entries.)

Using the call Statement

The **call** function transfers execution control to a function or a subroutine. The syntax for **call** is

> **call function(expression)**
>
> **call** entry label

You were introduced to **call** in many previous examples in this book. This section tells you in more detail how to use the **call** statement for subroutines.

HOW TO CALL A SUBROUTINE

The calling entry specifies the called entry by the entry label following the **call** function:

> **call label**

For example:

> **call a** **call x** **call B** **call F**

The **call** statement can only call entries that have alphabetical character labels with the letters a-z or A-Z. To call entries with numeric or symbol labels, use the glossary statement. For example:

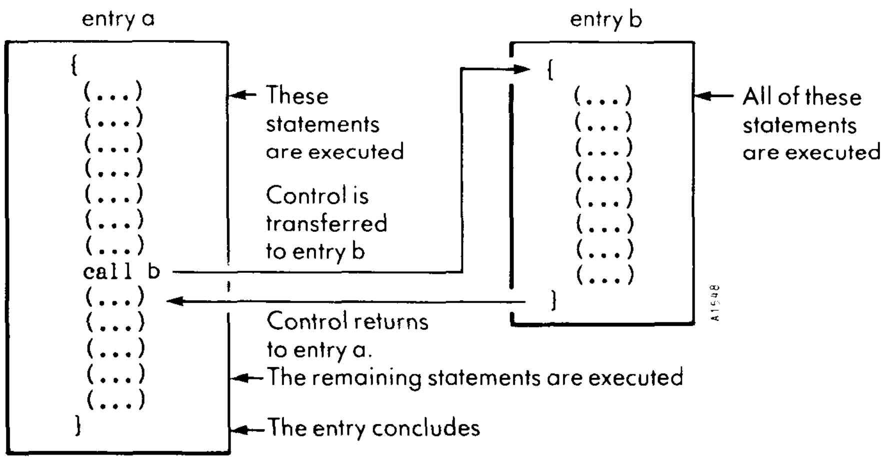
```
glossary "1"          glossary "@"
glossary "9"          glossary "$"
```

ORDER OF SUBROUTINE EXECUTION

When **call** is used to call another entry as a subroutine, the statements in the subroutine are executed, then program execution continues at the statement immediately after the subroutine call (unless directed elsewhere by the subroutine).

Figure 14 illustrates the flow of statement execution between the calling entry and the subroutine.

Figure 14 Calling a Glossary Subroutine Using the call Function



entry a
entry b
call b
These statements are executed
Control is transferred to entry b
All of these statements are executed
Control returns to entry a.
The remaining statements are executed
The entry concludes
A1548

Examples of the call Statement

Entry f uses nested **if else** statements to choose between four alternate subroutines.

Entry f is a form letter responding to a customer's request for information. It pauses processing after the second paragraph and prompts the operator for a choice of dealer addresses.

The typed character is assigned by the **keys** statement to the variable **dealer**. Nested **if else** statements determine which character was assigned to **dealer** and also call the correct subroutine.

> NOTE: Chapter 11 shows you how to use the **substr** function to alter the value returned by **date** so it reads June 3, 1984.

```
entry f
{
        insert
            format space(7) tab space(23) tab space(35) return execute
        execute
        tab(2) call feed(date)
        return(4)

        "Dear Customer:" return(2)

        tab "We are pleased you are considering us for your major supplier of
        widgets.  Our widgets are the finest in the world.  They are available in 24
        vibrant colors and make a variety of sounds at random moments."
        return(2)

        tab "The Amalgamated Widget dealer in your area is:"
        return(2)
        indent

        call prompt("Choose: w,x,y,z")
        dealer = keys
        call clrpos(1,50,30)
```

```
            if(dealer == "w")
                  { call w }
            else if(dealer == "x")
                  { call x }
            else if(dealer == "y")
                  { call y }
            else
                  { call z }

      return(2)

      tab "Again, thank you for your interest.  There is no better tool than a
      colorful, pleasant sounding Amalgamated widget."
      return(2)

      tab(2) "Sincerely yours" return
      tab(2) "AMALGAMATED WIDGETS, INC." return(4)
      tab(2) "J. Redd Widget, Jr." return
      tab(2) "Vice-President, Sales" return
}
```

Entries w, x, y, and z are used as subroutines for entry f. Because they do not contain dependencies on entry f, they may also be called by other entries in the same glossary document.

```
entry w
{
      "GENERAL WIDGETS, Box 123, Chicago, Illinois"
}


entry x
{
      "HAND-HELD WIDGETS, Pluto Street, Anaheim, California"
}


entry y
{
      "ALL PURPOSE WIDGETS, Steep Hill Blvd., San Francisco, California"
}
```

```
entry z
{
    "There is no ALMAGAMATED WIDGET dealer in your area, please contact
    our headquarters sales department at the address on this letterhead."
}
```

Using the glossary Statement

Using the keyword **glossary** in a program produces the same effect as pressing the GL key from the keyboard. If a glossary document is attached, the prompt "Which entry?" appears. You can respond to the prompt by providing the entry label in the program or by using the **keyin** or **keysin** function to type the label interactively.

You can use the keyword **glossary** to temporarily transfer control from one glossary entry to another glossary entry with any one of the following statements.

```
        glossary "label"
or
        glossary call keyin

or
        glossary call keysin
```

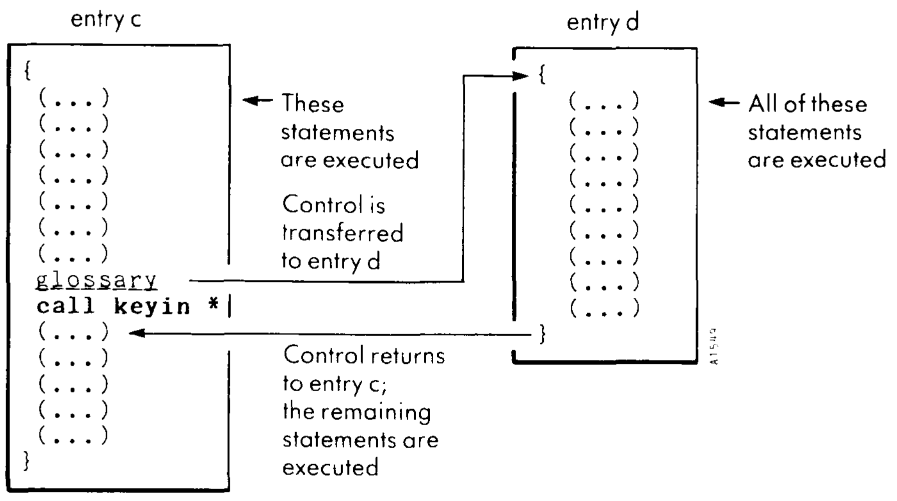
Both glossary entries must be in the same glossary document.

When you use the glossary statement you must always enclose the entry label in quotes.

In most instances, using **keysin** is preferable to using **keyin**. The **keyin** function permits the input of only one keystroke from the operator. If a mistake is made, no correction is possible. The **keysin** function permits backspacing to correct an incorrectly typed character.

Entry g provides an example using an interactive glossary statement to call a subroutine. Figure 15 illustrates the glossary **call keyin** statement.

Figure 15 **Interactively Calling a Glossary Entry as a Subroutine Using the glossary Statement**


entry c
entry d
{
(...)
glossary
call keyin *
}
These statements are executed
Control is transferred to entry d
Control returns to entry c; the remaining statements are executed
All of these statements are executed
A1549

*Control is transfered to entry d when operator types d.

Entry g uses the statement glossary **call keysin** to interactively select a glossary subroutine. Entries o and p are examples you can use as subroutines if you want to try this entry. The sales figures could be an entry containing a lengthy sales report that is also used as a subroutine by other report-type entries. Note that entry g calls entry y (shown as a subroutine for entry f) to type the "TO:" line of the memorandum.

```
entry g
{
        "\cMEMORANDUM" return (2)
        "TO: " call y return(2)
        "FROM: J. Redd Widget, Jr., Vice-President, Sales" return(2)
        "SUBJECT:  SALES QUOTA" return(2)
        "Congratulations on achieving 100 percent over sales quota last month.  You
        will be crowned WIDGET KING OF THE MONTH at this month's
        Almalgamated Widget Bash."
        return(2)
        "Last month's sales figures for all regions were:"
        return(2)
        glossary call keysin

        return(2)

        "Your sales quota projection for this month is being sent by separate memo."
        return(2)

}


entry O
{
        "$1,500,000.00"
}


entry P
{
        "$250.00"
}
```

Nesting Subroutine Calls

Subroutine calls can be nested so that entry a calls entry b, which calls entry c, which calls entry d.

When a subroutine has executed all its statements, it always returns control to the entry that called it. When a subroutine returns to its calling entry, execution resumes at the statement immediately after the **call** statement.

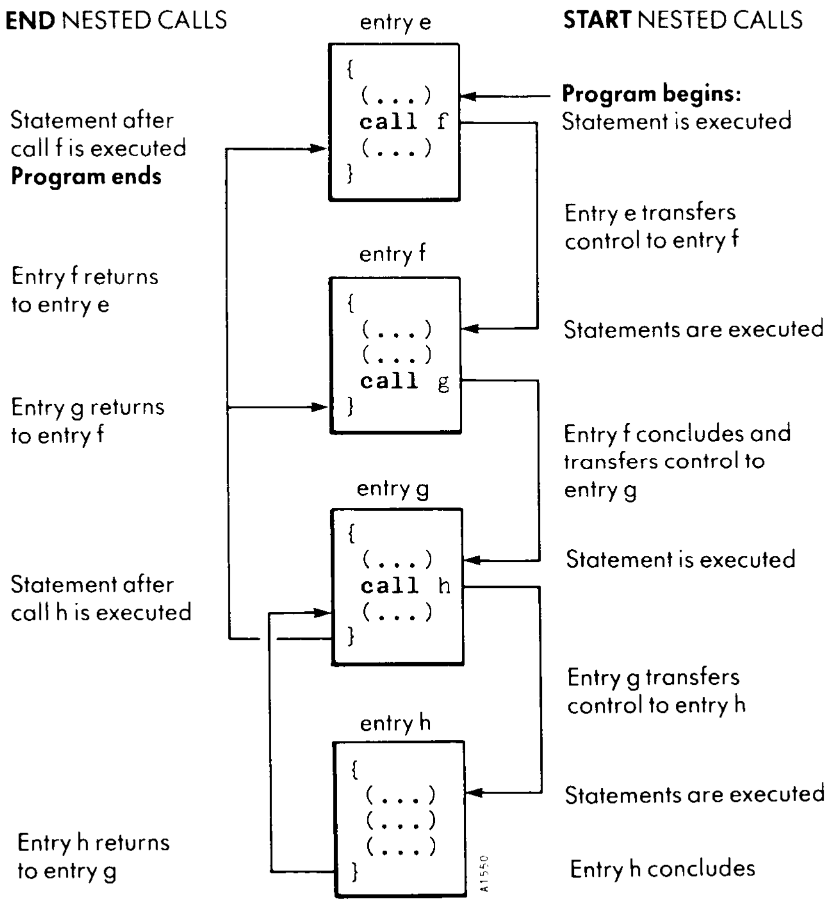
Entry h gives you a simple example of nested subroutines. Try it and you'll get an idea of what the flow is like through the subroutine structure.

Figure 16 illustrates four nested subroutines.

```
entry h
{
        "Help! I'm in a maze! Where do I go next?" return

        call i

        "Oh! Safe at last!" return(2)
}
```

```
entry i
{
        "Not this way.  Maybe there's a door here..." return
        call j
        "Great, its a light at the end of the tunnel!" return
}

entry j
{
        "No door, I'm in a tunnel.  Which way now?" return
        "I think the tunnel is winding up.  What's that I see?" return
}
```

Figure 16 Nested Subroutine Calis Using the call Function


END NESTED CALLS
entry e
START NESTED CALLS
{ (...) call f (...) }
Program begins: Statement is executed
Statement after call f is executed **Program ends**
Entry e transfers control to entry f
entry f
{ (...) (...) call g }
Entry f returns to entry e
Statements are executed
Entry g returns to entry f
Entry f concludes and transfers control to entry g
entry g
{ (...) call h (...) }
Statement is executed
Statement after call h is executed
Entry g transfers control to entry h
entry h
{ (...) (...) (...) }
Statements are executed
Entry h returns to entry g
Entry h concludes
A1550

BRANCHING

Program execution control can be unconditionally transferred to a different part of an entry or to another entry in the same glossary document. This procedure is called branching because you branch off the main (or trunk) statement execution line.

Unlike subroutines, branches do not return to their point of departure; execution continues from the branch.

The jump Statement

The **jump** function performs a branch to a labeled statement within the same entry.

Jumping always unconditionally transfers execution control to a labeled statement. The **jump** statement is usually invoked by a conditional statement as in the following examples:

```
if(!end_doc)
{
        jump typemore
}


if(bot_page)
}
        exit
}

else
{
        jump goagain
}
```

The **jump** statement can also be used to perform loops. More information about looping can be found in the next section, Looping.

The syntax for the **jump** statement is

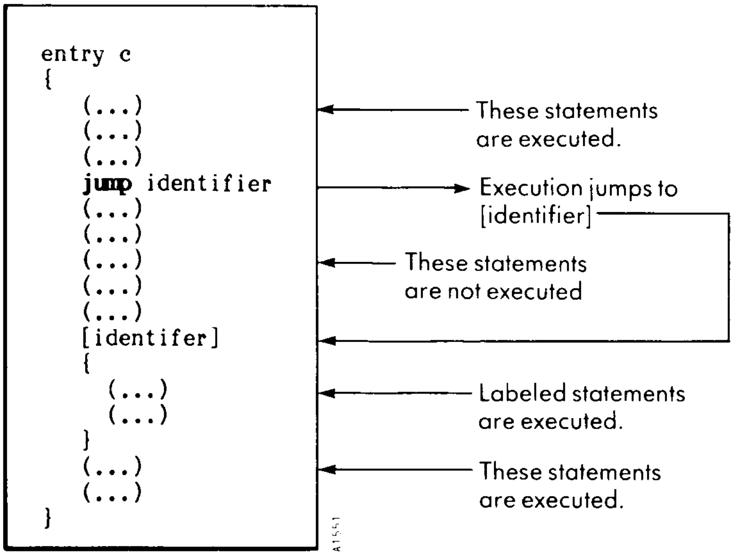
> **jump** identifier
> [identifier]
> statement or statements

The **jump** statement must always have an identifier to jump to. It may be any word you choose except a reserved word (see Appendix A for a list of reserved words). The identifier permits the same character composition rules as variable names. The identifier must always be enclosed in brackets [ ].

The statements following the identifier are called labeled statements.

The **jump** statement and identifiers are used in many of the previous examples in this and other chapters. Figure 17 illustrates the execution flow for jumps.

Figure 17 Branching within the Same Entry Using the jump Statement


entry c
{
(...)
(...)
(...)
jump identifier
(...)
(...)
(...)
(...)
(...)
[identifer]
{
(...)
(...)
}
(...)
(...)
}
These statements are executed.
Execution jumps to [identifier]
These statements are not executed
Labeled statements are executed.
These statements are executed.
A1551

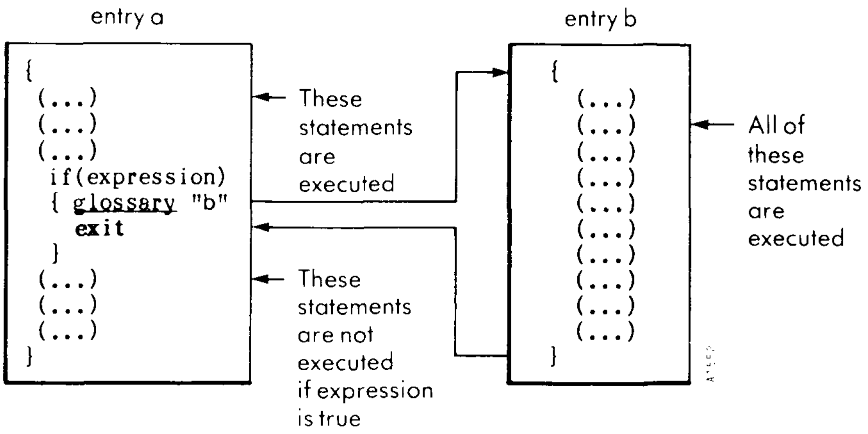
USING A GLOSSARY OR CALL STATEMENT TO BRANCH

You can use glossary or **call** to branch to another entry, but you must be sure it is the last statement in the main entry.

If statements exist after the **glossary** or **call** statement, place an **exit** statement after the **glossary** or **call** statement. When the called entry returns to the calling entry, it will read the **exit** statement and terminate.

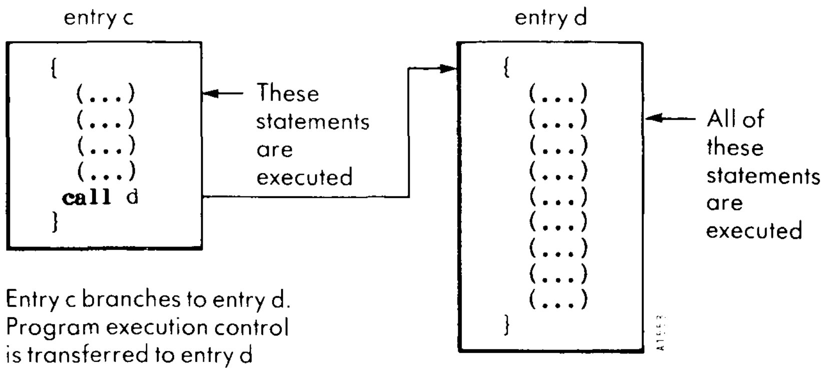
Figures 18 and 19 illustrate the use of glossary and **call** to perform branches to another entry in the same glossary document.

Figure 18 **Branching to Another Entry Using the glossary Statement**


entry a
{
(...)
(...)
(...)
if(expression)
{ glossary "b"
exit
}
(...)
(...)
(...)
}
These statements are executed
These statements are not executed if expression is true
entry b
{
(...)
}
All of these statements are executed

Entry a branched to entry b, execution resumes at the top of entry b. After statements in entry b are executed, control returns to entry a. The exit function ensures that any statements which follow "b" statement are executed.

Figure 19 **Branching to another entry using the call Function**


entry c
{
(...)
(...)
(...)
(...)
call d
}
These statements are executed
entry d
{
(...)
}
All of these statements are executed

Entry c branches to entry d. Program execution control is transferred to entry d

Execution resumes at the top of entry d and concludes at the bottom of entry c.

(When there are statements in entry c following the **call** statement, entry d returns control to entry c and the remaining statements in entry c are executed).

Control is temporarily transferred from one glossary entry to another glossary entry by the **call** function. The **call** function is normally used as a subroutine or function call. It can be used to branch when no other statements follow the **call** statement. If other statements follow **call**, you must use an **exit** statement as shown in Figure 18. Both glossary entries must be in the same glossary document.

In Figure 19 entry c executes its statements then branches to entry d using the statement **call d**. Execution resumes at the top of entry d. The statements in entry d are executed, and control is returned to entry c. Since there are no more statements to execute in entry c, the entry is terminated.

LOOPING

When you write a program without a loop instruction, the program executes straight through its statements one time only. When you add a loop to the same program, the program will repeat itself over and over again. You have to stop this repetition by the strategic placement of a conditional expression.

Two very powerful looping functions are the **while** and the **do while** statements. They accomplish the loop and at the same time provide the conditional expression for stopping the loop. These looping functions are covered in Chapter 9 Conditional Loop Statements. Both **while** and **do while** observe the general principles of conditional statements that were covered in Chapter 7.

The **jump** statement can be used to perform loops by placing a **jump** statement at the bottom of the loop statements and an identifier at the top.

Whether you use **while**, **do while**, or **jump** to perform your loop depends on the result you wish to achieve with your program and, to a large extent, with the performance of WORD ERA.

Since many of the previous examples in this book used loops, you probably have a pretty good idea of what loops can do by now. Entries k and l show you more examples of programs using loops.

There are as many ways to use loops as there are programs. They can count pages or lines in the document for you; they can increment variables; they can repeatedly search for a string in the document. Your specific application will dictate when and which types of loops you need in your programs.

Points to Remember About Loops

There are some important points you should always remember when you construct program loops:

A loop will keep looping unless it is terminated at some point by a conditional expression. The **globerr** function is frequently used to break a loop (**globerr** is covered later in this chapter).

Variables must always be declared and initialized outside the loop, or their value will be reinitialized each time the loop repeats.

Subroutines can be nested to perform counting or calculation loops on variables in the calling entry. Be sure the variables a subroutine uses are declared and initialized in the calling entry outside of any loops.

A **do while** loop will always execute its statement at least once. (See Chapter 9.)

A **while** loop will never execute its statement if the condition starts out false. (See Chapter 9.)

The **call** and **glossary** statements can be used to make an entry recall itself. When **call** and **glossary** are used to perform loops, execution always begins at the statement immediately following the entry label. Any variables will be reinitialized at the next repeat of the loop. It is generally better programming practice to use the **while**, **do while**, and **jump** statements to loop.

When you write programs using loops, first write the program without the loop. Recall it in the text document and be sure the first iteration works. Then add the loop to the entry.

A runaway loop that isn't working properly can trample right through your document, perhaps causing some damage on the way. It is always a good idea to make a copy of your document to use for testing new glossary programs. If the program works properly in the copy you'll know you can safely make it available for general use.

Using the jump Statement for Loops

The **jump** statement is frequently used to perform loops because the beginning and the ending of the loop can be specifically set by the jump statement and the identifier.

When you use **jump** to perform a loop, remember there must always be an identifier for the **jump** statement to jump to. The identifier marks the place where the loop repeats its statements. If you forget to put the identifier in your program the glossary compiler will remind you with a verification error.

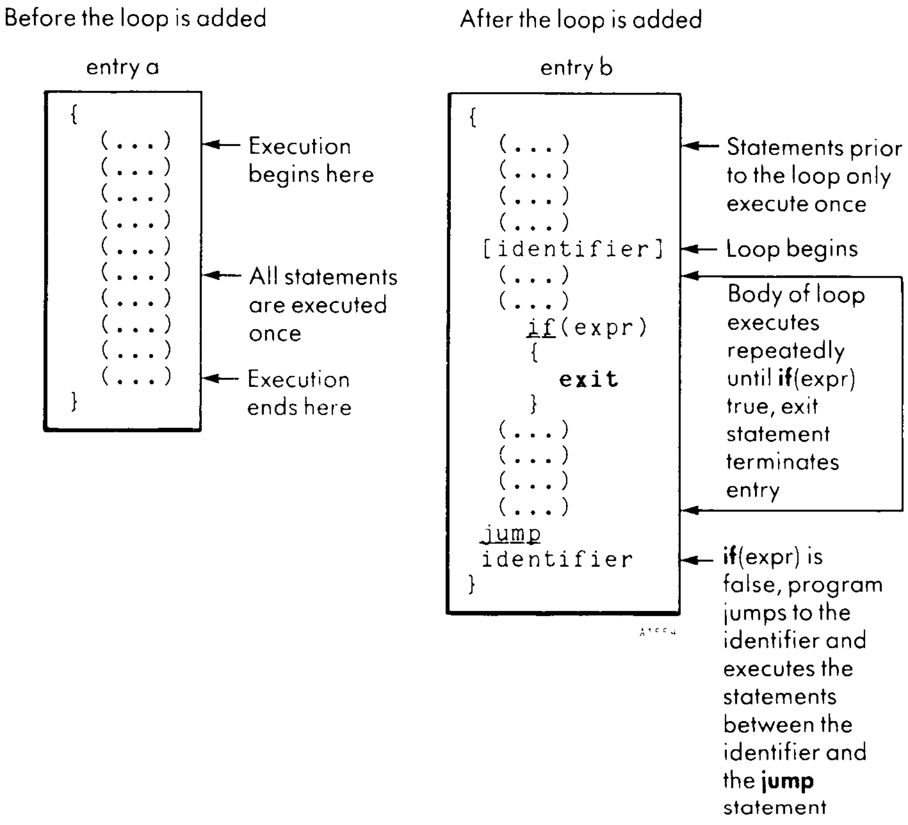
You must have a conditional statement that breaks the loop somewhere between the identifier and the jump statement, or the loop will repeat itself indefinitely. The conditional statement may direct program execution elsewhere by branching, calling a subroutine, or causing the entry to terminate by using an **exit** statement. The conditional statement is the predictable, graceful way to break a loop.

A loop can break itself unpredictably when it encounters an error condition, such as search not finding its string, no next screen, or the cursor is at the end of the document. This is not a graceful way to allow the loop to break because you are not fully controlling the course of program execution, and the result may be totally unpredictable.

You can use the **if** or **if else** conditional statements to break loops. The **while** and **do while** functions are in themselves conditional statements. Both **while** and **do while** are covered in Chapter 9.

Figure 20 shows the program execution flow when the **jump** statement is used to loop.

Figure 20 Using the jump Statement to Perform a Loop


Before the loop is added
entry a
Execution begins here
All statements are executed once
Execution ends here
After the loop is added
entry b
Statements prior to the loop only execute once
Loop begins
Body of loop executes repeatedly until if(expr) true, exit statement terminates entry
if(expr) is false, program jumps to the identifier and executes the statements between the identifier and the jump statement

```
{
   (...)
   (...)
   (...)
   (...)
   (...)
   (...)
   (...)
   (...)
   (...)
   (...)
}
```

```
{
   (...)
   (...)
   (...)
   (...)
  [identifier]
   (...)
   (...)
     if(expr)
     {
        exit
     }
   (...)
   (...)
   (...)
   (...)
  jump
  identifier
}
```

EXAMPLES USING THE JUMP STATEMENT

Entry k is an example of the **jump** statement. This entry gives you a good example of how to use mathematical operators to perform calculations on a table in a document. (You can also use the Math function to manually perform these calculations from your document editing screen, or use a combination of Math and Glossary by Example.)

To try entry k, first, type the following table example in a document. Be sure to use decimal tabs to enter the numbers as the entry searches for a decimal tab to begin its calculations. This table is on Page N of the glossary document **gloss2b** on the Glossary Diskette. If you don't want to take the time the type the table, copy it from Page N of **gloss2b** into your text document.

Second, type entry k which follows the example. (Entry k is also included on the Glossary Diskette in the glossary document **gloss2b**). An analysis of entry k follows the entry example.

EXAMPLE FOR USE WITH ENTRY K:

AMALGAMATED WIDGETS, INC.

MONTH END SALES STATEMENT

Part	Qty	Price Per Item	Gross Sales	Mfg Cost per Item	Net Sales
red widget	400	$25		$2	
green widget	327	$27		$3	
blue widget	728	$48		$7	
orange widget	120	$17		$1	
yellow widget	247	$86		$8	
black widget	124	$14		$2	
white widget	867	$14		$2	
violet widget	974	$87		$9	
TOTAL					

```
entry k
{
        salesqty = 0
        priceper = 0
        grossale = 0
        mfgcosts = 0
        netsales = 0
        [loop]
        search decimaltab execute
              if(globerr)
              {
                      execute
                      return(2) call finsert(time)
                      exit
              }
        cancel
        salesqty = number
        priceper = number
        grossale = priceper * salesqty
        right
        insert
            call feed(grossale)
        execute
        mfgcosts = number
        netsales = grossale - mfgcosts * salesqty
        right
        insert
            call feed(netsales)
        execute
        jump loop
}
```

Entry k performs several calculations on the "Amalgamated Widgets Month End Sales Statement" table that precedes the entry example. The same calculations are performed for each line. First, "Price Per Item" is multiplied by "Quantity," which becomes the "Gross Sales" amount. Second, "Manufacturing Cost per Item" is multiplied by "Quantity" and is subtracted from "Gross Sales." The result becomes the "Net Sales" amount.

When the entry is recalled, the first line is calculated and the amounts entered. The **jump** loop statement at the end of the program causes the entry to resume execution at the search decimaltab statement following the [loop] identifier. The next line is calculated, and the entry continues looping until search fails to find another decimaltab. When this occurs, the **if(globerr)** statement causes the **exit** statement to execute and the entry terminates.

Variables are set for each number value required by the calculations. Note that the variables are declared and initialized outside the [loop] identifier.

The **number** function is new to you in this entry. It is a document reading function that returns the number at the cursor location. When **number** is used, the cursor moves to the space or character following the number in the document. The number is read if the cursor is on the number itself or on the screen symbol immediately preceding it.

Entry l adds the number formatting function **pic**, a "Total" branch, and some finishing touches to entry k. You were introduced to the **pic** function in Chapter 6. The syntax for **pic** is **pic(expression1,expression2)**. It formats the number in expression 1 with the symbols in expression 2.

The last statement in entry l is **call Z**. Entry Z is a subroutine that is shown at the end of this chapter. It is a nice way to finish an entry and notify the operator of its conclusion.

When you recall entry l, note that it runs considerably faster than entry k. The increase in execution time is caused by the following changes:

> The addition of the statements **call display(false)** and **call display(true)**, which turn the display off at the beginning and on at the end of the program. The program runs faster if the screen display is not constantly refreshed.
>
> The replacement of the statement search decimaltab execute... cancel with the statement goto decimaltab. This is a more direct method that eliminates several keystrokes and a screen refresh.
>
> The replacement of the statements insert call feed... execute with the statements **call finsert(pic(...))**. The **finsert** function inserts the contents of its expression at the cursor location. Again, this is a faster method that eliminates keystrokes and a screen refresh. See Chapter 10 for a detailed description of the **finsert** function.

```
entry l
{
        call display(false)
        salesqty = 0
        priceper = 0
        grossale = 0
        mfgcosts = 0
        netsales = 0
        salestotal = 0
        grosstotal = 0
        nettotal = 0
```

```
      [loop]
      goto decimaltab
          if(globerr)
          {
               jump total
          }
      salesqty = number
          salestotal += salesqty
      priceper = number
      grossale = priceper * salesqty
          grosstotal += grossale
      right
          call finsert(pic(grossale,"$,"))
      mfgcosts = number
      netsales = grossale - mfgcosts * salesqty
          nettotal += netsales
      right
          call finsert(pic(netsales,"$,"))
      jump loop

      [total]
      {
          command search "TOTAL" execute cancel
          goto right
          insert
               decimaltab call feed(pic(salestotal,","))
               decimaltab
               decimaltab call feed(pic(grosstotal,"$,"))
               decimaltab
               decimaltab call feed(pic(nettotal,"$,"))
          execute
          return(2)
          "Gross sales: "
          call feed(pic(grosstotal,"$,"))
          " are calculated by multiplying the quantity, "
          call feed(pic(salestotal,","))
          ", times price per each." return(2)
          "Net sales: "call feed(pic(nettotal,"$,"))
          " are calculated by multiplying manufacturing cost per each times
          quantity, and subtracting the result from gross sales."
          return(2)
      call display(true)             return(2) call finsert(time)
      }        goto up
      call Z
}
```

TERMINATING PROGRAM RECALL

A program is terminated when all its statements are executed or it can be deliberately terminated at any point in its execution by the **exit** statement.

The **exit** statement makes an entry cease execution. When the **exit** statement is encountered during the program, execution will immediately stop. Statements following the **exit** statement will not be performed.

When you use an **exit** statement in a subroutine, it pertains only to the subroutine. It will cause the subroutine ONLY to stop execution. The entry that called the subroutine continues at the statement immediately following the subroutine call.

Gracefully Terminating an Entry

A graceful way to conclude an entry is to notify the operator that it is finished. Entry Z is a short subroutine that can be called by any entry in your glossary document. Modify it to your taste or use it as is.

Notice that the prompts follow standard WORD ERA message conventions (Press Execute to Continue). This is a principle to keep in mind when you are designing programs. People become used to pressing keys automatically at certain times (most of the time without looking at the prompts). If you deviate from standard practice, do so with good reason, and make sure your glossary users understand which keys to press and when.

```
entry Z
{       call error("Entry Concluded")
        call prompt("Press Execute to Continue")
        call keyin
        call clrpos(1,50,29)
        call clrpos(25,51,28)
        exit
}
```

In entry Z, the **keyin** function allows the operator to enter one keystroke. Although the prompt calls for EXECUTE to be entered, any key will work.

TRAPPING FUNCTION ERRORS USING THE GLOBERR STATEMENT

There are several word processing functions that search for characters or symbols as part of their function. This type of function sounds a beep tone when it fails to find the specified object of its search. For example, a beep tone sounds when a GO TO PAGE function (like GO TO PAGE INDENT, or GO TO PAGE CENTER) does not find the specified symbol. The beep tone sounds when the search function fails to find its specified string. Nextscrn and Prevscrn cause a beep if there is no next or previous screen.

As you have seen from previous glossary program examples, you can use the **globerr** function as part of a conditional statement to trap the failure of a word processing "search" function and exit the program, jump to a branch of the program, or call a subroutine. The **globerr** function is particularly valuable for breaking any type of "search" loop.

The **globerr** function can return its value to a variable, a conditional statement, or a function. The value returned by **globerr** is 1 if true and 0 if false. The initial value is false, **globerr** returns a value of true if a preceding glossary operation resulted in an error condition that caused a beep tone. The value of **globerr** is reset to false after it is used.

See Chapter 10 for a functional description of **globerr.**

TIMING YOUR PROGRAMS

Entry 1 in this chapter showed you how to increase the response time of a glossary program by turning the display off and substituting functions. If you want to evaluate glossary execution time for comparison or scheduling purposes you can use the **time** function to include a time stamp statement in your program. Entry K and L give you an example of how to use the **time** function to determine the difference in execution time between the **feed** and **finsert** functions. Both entries insert the same paragraph of text.

```
entry K
{
     call feed(time)

     return(2)

     call feed("While creating or editing a document, you can automatically save
     the document every time a preset number of keystrokes is reached. Pressing
     STOP prompts you to enter the desired number of keystrokes allowed before
     the document is written to the hard disk. The default number of keystrokes
     is 1024. You can also press COPY before entering the number of keystrokes
     to save a copy of the document before making any further editing
     changes.")

     return(2)

     call feed(time) return(2)

}


entry L
{
     call finsert(time)

     return(2)

     call finsert("While creating or editing a document, you can automatically
     save the document every time a preset number of keystrokes is reached.
     Pressing STOP prompts you to enter the desired number of keystrokes
     allowed before the document is written to the hard disk. The default
     number of keystrokes is 1024. You can also press COPY before entering the
     number of keystrokes to save a copy of the document before making any
     further editing changes.")

     return(2)

     call finsert(time) return(2)

}
```

Depending on your system load, you'll find approximately five seconds execution time difference between entry K and entry L. The run time difference between entry K and entry L is approximately one minute.

SUMMARY

In this chapter you learned how to use control statements, to call subroutines, perform loops, and terminate an entry. In the next chapter you will learn how to use the conditional loop statements **while** and **do while**.

CHAPTER 9

CONDITIONAL LOOP STATEMENTS

The **while** and **do while** statements provide an expedient method of combining the conditional evaluations performed by **if** and **if else** and the looping functions performed by **jump** and its identifier.

All of the general principles about conditional functions described in Chapter 7 apply to the **while** and **do while** statements.

The main distinction to remember between **if** and **while** conditional statements is that **if** asks a question before it executes its statements. The conditional **if** can branch or use the **if else** combination to execute an alternative statement. The **while** function executes its statements as long as its condition proves true and has no other alternative. The two types of conditional statements can be nested together to form programming combinations. Entry o in this chapter shows an example of this combination.

THE CONDITIONAL while STATEMENT

The syntax for the conditional **while** statement is

```
while(expression)
{
     statement or statements
}
```

The **while** function repeatedly executes its statement or statements as long as its expression remains true. Multiple statements to **while** are always enclosed in braces. The argument to **while** may consist of various combinations of expressions and operators.

When the expression becomes false, the statement or statements are not executed, and the program continues after the closing brace in the **while** statement.

The condition stated by **while(expression)** is evaluated before the execution of the {statement or statements}. The statements to **while** will never be executed if the condition starts out false. In the following syntax examples, the statements are executed if the cursor is on the character "x" (in the first example) or the line at the cursor location is less than line 20 (in the second example).

```
        while(char == "x")
        {
                delete execute
        }
        while(line < "20")
        {
                insert
                        tab
                execute
                return
        }
```

Entries m and n use the **while** statement to perform conditional loops. Entry o combines entries m and n into one program.

Entry m is a rewrite of entry a in Chapter 7, which used a conditional **if** statement. A loop was also added to the entry by the **while** statement. To add a loop to entry a, you would have to use the **jump** statement. The **while** statement takes care of both requirements, the conditional test and the loop.

```
entry m
{
        while(page_no != 10)
        {
                insert
                        copy format "2" execute
                execute
                goto nextscrn
                        if(globerr) {exit}
        }
        insert
                return(6)
                "\cThis page intentionally left blank\r"
        execute
}
```

```
entry n
{
          while(page_no <= 4)
          {
                goto down
                insert
                     center call feed(page_no) "--Introduction"
                execute
                goto nextscrn
                     if(globerr) {exit}
          }
}
```

Entry o combines entries m and n to reformat a document. Note the nested **if** statement inside the **while** statement.

```
entry o
{
      while(page_no < 10)
      {
          insert
               copy format "2" execute
          execute
               if(page_no <= 4)
               {
                    goto down
                    insert
                         center call feed(page_no) "--Introduction"
                    execute
               }
          goto nextscrn
               if(globerr) {exit}
      }
      insert
          page return(6)
          "\cThis page intentionally left blank\r"
      execute

}
```

THE CONDITIONAL do while STATEMENT

Like the **while** function, the **do while** function allows repeated execution of a statement or statements based on the true condition of its expression.

The **do** statement enclosed in braces is executed repeatedly as long as the value of the expression or expressions in the argument to **while** remain true. When the value becomes false, the statement is not executed and the program continues at the statement following the **while(expression)**.

Since the test of the expressions takes place after each execution of the **do** statement, the statement will be executed at least once whether or not the expression to **while** is true.

The syntax for the **do while** statement is

```
do
{
     statement or statements
}
while(expression)
```

The argument to **while** may consist of various combinations of expressions and operators. Multiple statements to **do** must be enclosed in braces.

Entry p is an example of the **do while** statement. It uses a **do while** conditional loop to add the "Inventory" column on the "Amalgamated Widgets Parts List." example which follows entry p.

```
entry p
{
     inventory = 0
     numbertest = 0

     do
     {
          search decimaltab execute cancel
          right
          numbertest = num(char)
```

```
                if(numbertest == 0)
                {
                        insert
                                call feed(inventory)
                        execute
                }
        inventory += number
    }
        while (numbertest == 1)
}
```

The **num** function is new to you in this entry. It returns a value of 1 (true) if the value of its expression is a number. If the expression is not a number **num** returns a value of 0 (false).

When you type the "Parts List" in your text document, be sure to use decimal tabs with the numbers, and place a decimal tab and a return following "TOTAL" in the text document. (This example is on Page N of the glossary document gloss2b on the Glossary Diskette).

Type this example in a text document and recall entry p. The column is added and the total entered.

AMALGAMATED WIDGETS, INC.
PARTS LIST

Part Description	June 30 Inventory
red widget	20
green widget	40
blue widget	69
orange widget	17
yellow widget	34
black widget	34
white widget	56
violet widget	72
TOTAL	

Entry q is provided as a contrasting example to entry p. It performs exactly the same column addition as entry p, using the **jump**, **if**, and **globerr** functions. Note that this entry searches for TOTAL after the column is added and inserts the decimal tab and the value of the variable inventory. (Entry p requires an existing decimal tab following TOTAL.)

```
entry q
{
     inventory = 0

     [loop]
     search decimaltab
          if(globerr)
          {
               execute
               search "TOTAL" execute cancel
               goto right
               insert
                    decimaltab call feed(inventory)
               execute exit
          }
     execute cancel

     inventory += number

     jump loop
}
```

These two examples illustrate that there is no absolutely correct way to write a program. Many methods will work and work well; use the method that is easiest and most comfortable for you.

SUMMARY

You have completed Part 2 of this book Learning Glossary Programming. Part 3 Glossary Functions Reference and Usage Guide provides you with detailed descriptions of the glossary functions and guides to using them. Part 4 tells you how to administer your glossary programs, and offers suggestions and information for operating system users. Part 5 describes the Glossary Diskette provided with this book, and gives you some additional glossary programs to use.

Remember, all the glossary entry examples shown in this part are in glossary document **gloss2a** and **gloss2b** on the Glossary Diskette. You can use these programs without modification, or modify them to suit your requirements.

CHAPTER 10

FUNCTION DESCRIPTION LIST

This is an alphabetical reference chapter for all functions used in the Advanced Glossary programming language. Use it as you would a dictionary to look up a function. Each function entry includes:

A description of the function

The type of value returned by the function

The permissible syntax statements for the function

Program examples are provided in this chapter where appropriate to clarify the nature of the function. For additional programming examples, refer to the other chapters and appendices in this book.

Chapter 11 provides a compendium of functions by usage. Examples are provided for each usage group.

HOW TO USE THE ALPHABETICAL LIST OF FUNCTIONS

Functions are listed alphabetically by name. Information about the function is arranged in the format shown in Figure 21. The **beg_doc** function is used as an example in Figure 21.

Suppose you looked up the **beg_doc** function on the alphabetical list. How would you use this information to help you write a program? The list following Figure 21 describes each part of the format and gives you a few suggestions on its usage.

beg_doc

Type:	document reading
Value:	1 if true, 0 if false
Syntax:	conditional function(**beg_doc**) conditional function(!**beg_doc**) variable = **beg_doc** call function(**beg_doc**)

The **beg_doc** function returns a value of true if the cursor is on the first character of the document. Otherwise, it returns a value of false.

Figure 21 Example of Function Information Format

The following list provides a detailed description of the format shown in Figure 21.

Function:	Functions are listed in alphabetical order by name.
Type:	This is a cross-reference to the usage list of functions in the second section of this chapter. If you are not familiar with this type of function, look it up on the usage list and read the description and example for that usage group. The **beg_doc** function is a document reading function, so you know that it returns information from the text document as its value.
Value:	This is the type of value returned by the function. In the case of **beg_doc**, it is a numeric value that returns a number 1 if true (the cursor is at the beginning of the document) or a 0 (zero) if false (the cursor is not at the beginning of the document). Other types of functions return alphabetic or numeric string values.

Syntax: These are possible and permissible ways of using the function. Most functions can be used in a variety of statements; some are restricted to only one or two. The **beg_doc** function can be used as an expression to a conditional **if** or **while**. It can have its value assigned to a variable, or it can be used by another function, such as **status** or **error**.

The syntax combinations shown may not represent all possible combinations for the function. For example, if the syntax is shown as "conditional function(**position** operator expression)," you can just as easily reverse the expressions to have "conditional function(expression operator **position**)." Don't limit yourself to experimentation with just the combinations shown. You will probably find others that suit your programming requirements exactly.

Description: The paragraph following the syntax list tells you what the function does and how it performs. The values required for each expression in the argument are listed and explained. Brief program examples are used where appropriate to clarify the action of the function. This descriptive paragraph is similar to a dictionary definition and should be used in the same way. Program examples of how functions are used are in the Usage List of Functions in this chapter and in other chapters and appendices in this book.

TEXT CONVENTIONS USED IN THIS CHAPTER

In syntax examples, expressions may be shown as

function(expression1,expression2,expression3)

or as

function(e1,e2,e3,e4,e5)

Three dots following the last expression in an argument mean more expressions are allowed, as

function(expression1,expression2,...)

In some diagrams and figures, omitted program statements are represented by **(...)**

GENERAL RULES FOR USING FUNCTIONS

Functions that return values can be used anywhere an expression can be used, as shown in the following syntax examples:

```
variable = function

conditional function(function operator expression)

call function(function)
```

Functions that return values of true or false can be used anywhere an expression can be used. Returned values will always be 1 if true or 0 if false.

When a function is used as a statement it must be preceded by the **call** function.

The argument to a function is always enclosed in parentheses.

Unless otherwise stated, an expression to a function can be a string expression, a variable, a mathematical expression, or a function.

Multiple expressions within a function argument are separated by commas unless otherwise stated in the function description. (Expressions to conditional functions are treated differently. See the syntax descriptions for **if**, **if else**, **while**, and **do while**.)

LIST OF FUNCTIONS THAT REQUIRE ARGUMENTS

abs(expression)
cat(expression1,expression2)
clrpos(expression1,expression2,expression3)
cursor(expression)**
display(expression)
do while(expression)
error(expression)
feed(expression1,expression2*)
finsert(expression)
if(expression)
if(expression)**else**
index(expression1,expression2,expression3*)
len(expression)
max(expression1,expression2,...)
min(expression1,expression2,...)
num(expression)
occur(expression1,expression2)
pic(expression1,expression2)
posmsg(expression1,expression2,expression3)
prompt(expression)
round(expression1,expression2)
seg(expression1,expression2,expression3,expression4*)
status(expression)
sub(expression1,expression2,expression3,expression4,expression5)
substr(expression1,expression2,expression3*)
text(expression1,expression2)**
truncate(expression1,expression2)
unixfun(expression)**
unixpipe(expression1,expression2)**
while(expression)

* The numbered expression marked by an * is optional in the argument.

** The entire expression must be enclosed within quotation marks

ALPHABETICAL LIST OF FUNCTIONS

abs

Type:	mathematical
Value:	absolute value of a number
Syntax:	variable = **abs**(expression) conditional function(**abs**(expression) operator expression) call function(**abs**(expression))

The **abs** function provides the absolute or positive value of the xpression. The value in the expression must be a number. It may contain a leading dollar sign, commas, a decimal point, and/or leading or trailing minus or plus signs. It may not contain any alphabetic characters or other symbols.

beg_doc

Type:	document reading
Value:	1 if true, 0 if false
Syntax:	conditional function(**beg_doc**) conditional function(!**beg_doc**) variable = **beg_doc** call function(**beg_doc**)

The **beg_doc** function returns a value of true if the cursor is on the first character of the document. Otherwise, it returns a value of false. When it is preceded by the logical not operator ! the combination !**beg_doc** returns a value of true only if **beg_doc** is not on the first character of the document. The **beg_doc** function treats all of the following as characters: screen symbols, characters from alternate character sets, spaces, alphabetic characters, numeric characters.

bot_page

Type:	document reading
Value:	1 if true, 0 if false
Syntax:	conditional function(**bot_page**) conditional function(!**bot_page**) variable = **bot_page** call function(**bot_page**)

The **bot_page** function returns a value of true if the cursor is beyond the last character of the page. Otherwise, it returns a value of false. When it is preceded by the logical not operator (!) the combination !**bot_page** returns a value of true only if **bot_page** is not beyond the last character of the page. The **bot_page** function treats all of the following as characters: screen symbols, characters from alternate character sets, spaces, alphabetic characters, numeric characters.

call

Type:	control
Value:	**call** does not return a value
Syntax:	**call** function(expression) **call** label

The **call** function is a statement that transfers execution control to a built-in function. A function is only preceded by **call** when it is used as a statement. The **call** statement is not required when a function is used as an expression. When a function is called, the function is executed. Control then returns to the statement immediately following the function call.

The **call** function is also used to transfer execution control to a glossary entry in the same glossary document. When **call** is used to call another entry as a subroutine, the statements in the subroutine are executed, then program execution continues at the statement immediately after the subroutine call (unless directed elsewhere by the subroutine).

cat

Type:	string
Value:	a continuous string expression that results from the concatenation of expression1 and expression2
Syntax:	variable = **cat**(expression1,expression2) conditional function(**cat**(e1,e2) operator expression) call function(**cat**(e1,e2))

The **cat** function concatenates (brings together) expression1 and expression2 and provides one continuous string expression.

char

Type:	document reading
Value:	character at cursor location
Syntax:	variable = **char** conditional function(**char** operator expression) call function(**char**)

The **char** function passes the character found at the cursor position to the variable. The cursor remains on the character on the document screen.

clrpos

Type:	display
Value:	**clrpos** does not return a value
Syntax:	call **clrpos**(expression1,expression2,expression3)

The **clrpos** function displays the number of blanks specified by expression3 at the line specified by expression1 and the character position specified by expression2. Permissible screen parameters for **clrpos** are lines 1 through 25 and positions 1 through 80. Messages extending beyond position 80 or line 25 will result in screen display anomalies. The blanks posted by **clrpos** can be cleared at the end of glossary execution by pressing CANCEL and RETURN, CANCEL and EXECUTE, or CTRL w (simultaneously press CTRL and w). The **clrpos** function does not replace characters in the document. It is a temporary display.

cursor

Type:	display
Value:	**cursor** does not return a value
Syntax:	call **cursor**(expression)

The **cursor** function moves the cursor to the location specified by the expression. If the expression contains the string value "3,9,12", the cursor moves to page 3, line 9, position 12. The page designation may be a numbered page, or page h, f, or w. The expression in the argument to **cursor** must be a quoted string in the form "page, line, position," or it may be a single unquoted variable whose value is the quoted string expression.

date

Type:	operating system access
Value:	the current system date and time
Syntax:	variable = **date** conditional function(**date** operator expression) call function(**date**)

The **date** function returns the current system date and time in the form Thu May 1 09:40:00 1986.

display

Type: display

Value: **display** does not return a value

Syntax: call **display**(expression)

The **display** function turns the display on if the value of expression is true (non-zero) and off if the value is false (zero). The syntax to turn the display OFF is **call display(false)**. To turn the display ON the syntax is **call display(true)**.

do while

Type: conditional

Value: **do while** does not return a value

Syntax:
```
do
{
      statement or statements
}
while(expression)
```

(expression) may consist of various combinations of expressions and operators:

    while(expression operator expression)

As long as proper parenthetical syntax is followed, the argument to **while** may contain a theoretically unlimited number of expressions and operators.

{Multiple statements} to **do** must be enclosed in braces.

The **do while** function allows repeated execution of a statement or statements based on true or false conditions. The true or false conditions are specified by the expressions in the argument to **while**. The **do** statements enclosed in braces are executed repeatedly as long as the value of the expression in the argument to **while** remains true. When the value becomes false, the **do** statements are not executed, and the program continues after the **while** argument. Since the test of the expressions takes place after each execution of the **do** statements, the statements will be executed at least once whether or not the argument to **while** is true.

end_doc

Type:	document reading
Value:	1 if true, 0 if false
Syntax:	conditional function(**end_doc**) conditional function(!**end_doc**) variable = **end_doc** call function(**end_doc**)

The **end_doc** function returns a value of true if the cursor is beyond the last character of the document. Otherwise, it returns a value of false. When it is preceded by the logical not operator (!) the combination !**end_doc** returns a value of true only if **end_doc** is beyond the last character of the document. The **end_doc** function treats all of the following as characters: screen symbols, characters from alternate character sets, spaces, alphabetic characters, numeric characters.

error

Type:	display
Value:	string
Syntax:	call **error**(expression)

The **error** function displays the value of expression highlighted in the error section of the screen (line 25, character locations 51 to 79). The **error** display is accompanied by a beep tone. The length of the error string cannot exceed 29 characters. Strings longer than 29 characters will result in screen display anomalies. The **error** message can be cleared with the **clrpos** function, by pressing CTRL/w, by including the CTRL/w statement "\027" in the program, or by invoking an editing function, like INSERT or DELETE.

exit

Type:	control
Value:	**exit** does not return a value
Syntax:	**exit**

The **exit** statement makes an entry cease execution. When the **exit** statement is encountered in the program, execution will immediately stop. Statements following the **exit** statement will not be performed. The **exit** statement in a subroutine pertains only to the subroutine. It will cause the subroutine to stop execution. The entry that called the subroutine will continue at the statement immediately following the subroutine call.

false

Type:	logical
Value:	provides a numeric value of 0
Syntax:	variable = **false** conditional function(expression operator **false**) call function(**false**)

The **false** function is used to provide a **false** value for a variable or a function. It can also serve as an expression in a conditional statement. The **false** function always returns a value of zero.

feed

Type:	document writing
Value:	**feed** does not return a value
Syntax:	call **feed**(expression1,expression2)

The **feed** function types the value in expression1 as if it came from the keyboard. Expression2 is optional. If it is included, the value in expression1 will be typed the number of times specified by expression2. The typed characters from expression1 remain as part of the document text. The **feed** function does not insert, and it overwrites existing text if the cursor is not in a blank area of the screen. To insert, nest a **call feed {...}** statement in an insert modeas follows:

insert call feed(expression) execute.

finsert

Type:	document writing
Value:	**finsert** does not return a value
Syntax:	call **finsert**(expression)

The **finsert** function inserts the contents of the expression into a document at the cursor location. The **finsert** function must be used when a returned value contains screen symbols such as a RETURN, TAB, INDENT, or CENTER. These symbols are displayed in the document as symbols; however, they are read by functions such as **char** or **text** as WORD ERA document control codes. (Appendix C describes WORD ERA document control codes.) Use **finsert** to insert values returned by the **text** function. (See the description of the **text** function in this chapter.)

globerr

Type:	error
Value:	1 if true, 0 if false
Syntax:	variable = **globerr** conditional function(**globerr** operator expression) call function(**globerr**)

The initial value of **globerr** is false. It only returns a value of true if the preceding search or goto [symbol] glossary operation resulted in an error condition that caused a beep tone. For example, the search function fails to find its specified string, and the beep tone sounds. The **globerr** function is particularly useful for breaking a "search" loop. The value of **globerr** is reset to false after it is used. The **globerr** function can be used with the glossary keywords:

```
search   nextscrn   prevscrn   goto command indent
goto indent   goto center   goto dectab   goto tab
```

in the form: **keyword(s) if(globerr) {...}**

if

Type:	conditional
Value:	**if** does not return a value
Syntax:	**if**(expression)

```
if(expression)
{
     statement or statements
}
```

(expression) may consist of various combinations of expressions and operators:

if(expression operator expression)

As long as proper parenthetical syntax is followed, the argument to **if** may contain a theoretically unlimited number of expressions and operators.

Multiple {statements} to **if** must be enclosed in braces.

The **if** conditional statement allows the glossary program to make decisions based on specified conditions in the document. The expression in the argument is evaluated, and if true, the statement or statements enclosed in braces are executed. If the expression in the argument is false, the statements enclosed in braces are skipped, and program execution continues immediately beyond the last brace in the **if** statement.

if else

Type: conditional

Value: the **if else** statement does not return a value

Syntax:

```
if(expression)
{
     statement or statements
}
else
{
     statement or statements
}
```

(expression) may consist of various combinations of expressions and operators:

```
     if(expression operator expression)
```

As long as proper parenthetical syntax is followed, the argument to **if** may contain a theoretically unlimited number of expressions and operators.

Multiple {statements} to **if** must be enclosed in braces.

Multiple {statements} to **else** must be enclosed in braces.

The **if else** conditional statement allows the program to execute either the **if** statements or the **else** statements, depending on a true or false condition of the expression in the argument to **if**, (**else** does not require an argument; it relies on the argument to **if**).

The expression in the argument to **if** is evaluated. If true, the statement or statements enclosed in braces are executed. The statements following **else** are skipped, and program execution continues at the statement immediately after the closing brace in the **else** statement (unless directed elsewhere by the **if** statements).

If false, the statements to **if** are skipped, and the statements to **else** are executed. Program execution then continues at the statement immediately after the closing brace in the **else** statement (unless directed elsewhere by the **else** statements).

index

Type:	string
Value:	character number where expression2 begins inside expression1

Syntax:

```
variable = index(expression1,expression2)
variable = index(expression1,expression2,expression3)
conditional function(index(e1,e2,e3) operator expression)
call function(index(e1,e2,e3))
```

The **index** function searches for an occurrence of expression2 inside of expression1, beginning at the character number provided by expression3. Expression3 is optional. If it is not present, the search begins at character 1 of expression1. If expression2 is not found inside expression1, a false (zero) value is returned. If it is found, the value returned is the first character position inside expression1 where expression2 begins.

jump

Type:	control
Value:	**jump** does not return a value

Syntax:

```
jump identifier
     [identifier]
     statement or statements
```

The **jump** statement unconditionally transfers program execution control to the statement immediately following a labeled identifier. The identifier may be any word other than reserved keywords and must be enclosed in brackets. The rules for naming variables also apply to identifiers.

The labeled statement can be anywhere in the entry. Unlike a subroutine call, the program does not return to the statement following the **jump** statement after executing the labeled statements.

key

Type:	interactive
Value:	**key** accepts one keystroke from the keyboard. Typically, this value is passed to a variable
Syntax:	variable = **key** call function(**key**) conditional function(**key** operator expression)

The **key** function pauses program execution until the operator types one key. This key can be assigned to a variable or used by a function. The typed key is not written in the document. Any key on the keyboard is accepted by key and can be assigned to a variable. This includes character keys, cursor movement keys, and function and editing keys such as RETURN, TAB, EXECUTE, DELETE, or INSERT. The **key** function does not accept special characters accessed by CTRL y keystroke combinations.

keyin

Type:	interactive
Value:	**keyin** does not return a value
Syntax:	call **keyin**

The **keyin** function pauses program execution so that the operator can type one key. When a character key or screen symbol key such as RETURN or TAB is pressed, it is typed in the document and remains as part of the text. Any key pressed counts as a keystroke, including cursor control keys and function and editing keys such as EXECUTE or DELETE. Execution of the entry resumes after the key is typed.

keys

Type:	interactive
Value:	**keys** accepts unlimited keystrokes from the keyboard. Typically, this value is passed to a variable
Syntax:	variable = **keys** call function(**keys**) conditional function(**keys** operator expression)

The **keys** function pauses program execution so that any number of characters may be typed. Only standard character keys are accepted by the **keys** function. A beep will sound if a function or editing key is pressed. When the EXECUTE or RETURN key is pressed by the operator, program execution continues, and the entered string of characters is passed to the variable or function. Characters entered to **keys** will appear to overwrite existing text in the document. This is a temporary condition and can be cleared at the end of glossary execution by pressing CTRL/w or by including the statement "\027", which is the octal representation for CTRL/w, in the program.

keysin

Type:	interactive
Value:	**keysin** does not return a value
Syntax:	call **keysin**

The **keysin** function pauses program execution so that the operator can type an unlimited sequence of keys. These may be character keys for data entry or formatting keys such as TAB, RETURN, or PAGE.

Characters are typed in the document and remain as part of the text. Execution of the entry resumes when the EXECUTE key is pressed by the operator.

left_margin

Type:	document reading
Value:	1 if true, 0 if false
Syntax:	conditional function(**left_margin**) conditional function(!**left_margin**) variable = **left_margin** call function(**left_margin**)

The **left_margin** function returns a value of true if the cursor is on the first character of a line. Otherwise, it returns a value of false. When it is preceded by the logical not operator (!) the combination !**left_margin** returns a value of true only if **left_margin** is not on the first character of a line.

len

Type:	string
Value:	number of characters in expression
Syntax:	variable = **len**(expression) conditional function(**len**(expression) operator expression) call function(**len**(expression))

The **len** function returns a number value equivalent to the number of characters in its expression. Keyword abbreviations, WORD ERA document control codes, and octal numbers that are embedded in the string are included in the character count. Keyword abbreviations, such as \r, count as one character. Octal numbers, such as \007, count as one character. WORD ERA document control codes, such as \B\, count as 2 characters. (The backslash (\) is used as an escape character for embedments and does not count as a character unless it is escaped by another backslash, as "\\". The combination "\\" counts as one character.)

line

Type:	document reading
Value:	line number for the cursory
Syntax:	variable = **line** conditional function(**line** operator expression) call function(**line**)

The **line** function returns the line number of the cursor location in the document.

loc

Type:	document reading
Value:	page, line, and position of the cursor in the form "1,4,6"
Syntax:	variable = **loc** conditional function(**loc** operator expression) call function(**loc**)

The **loc** function returns a value that specifies the page, line, and position of the cursor location in the document. The value is returned in three segments separated by commas; for example, the string h,2,44 translates as header page, line 2, position 44. The string 10,18,66 translates as page 10, line 18, position 66. The page designation may be a numbered page or page h, f, or w.

max

max as used with numeric expressions

Type:	mathematical
Value:	the expression containing the highest number of all stated expressions

Syntax:	variable = **max**(expression1,expression2,...) conditional function(**max**(e1,e2,...) operator expression) call function(**max**(e1,e2,...))

The **max** function evaluates all of its stated expressions and returns the highest expression (number) as its value.

max

max as used with alphabetical string expressions

Type:	string
Value:	the highest alpha string expression based on ascending order of the ASCII collating sequence
Syntax:	variable = **max**(expression1,expression2,...) conditional function(**max**(e1,e2,...) operator expression) call function(**max**(expression1,expression2,...))

The **max** function returns as its value the highest of its alphabetic string expressions in ascending order according to the ASCII collating sequence provided in Appendix C. Any number of string expressions can be compared.

min

min as used with numeric expressions

Type:	mathematical
Value:	the expression containing the lowest number of all stated expressions
Syntax:	variable = **min**(expression1,expression2,...) conditional function(**min**(e1,e2,...) operator expression) call function(**min**(expression1,expression2,...))

The **min** function evaluates all of its stated expressions and returns the lowest expression (number) as its value.

min

min as used with alphabetical string expressions

Type:	string
Value:	the lowest alpha string expression based on descending order of the ASCII collating sequence
Syntax:	variable = **min**(expression1,expression2,...) conditional function(**min**(e1,e2,...) operator expression) call function(**min**(expression1,expression2,...))

The **min** function returns as its value the lowest of its alphabetic string expressions in descending order according to the ASCII collating sequence provided in Appendix C. Any number of string expressions can be compared.

num

Type:	mathematical
Value:	1 if true, 0 if false
Syntax:	variable = **num**(expression) conditional function(**num**(expression)) call function(**num**(expression))

The **num** function returns a value of true if the expression is numeric, and a value of false if it is not. Only numeric strings are recognized. If the string contains any alphabetic characters, a value of 0 is returned. The number may contain a leading dollar sign, commas, a decimal point, and/or leading or trailing minus or plus signs.

number

Type:	document reading
Value:	**number**-at the cursor location
Syntax:	variable = **number** conditional function(**number** operator expression) call function(**number**)

The **number** function passes the **number** found at or to the immediate right of the cursor position to the variable. Only numeric strings are recognized. If the string contains any alphabetic characters, "12th" for example, a value of 0 is returned to the variable. The number may contain a leading dollar sign, commas, a decimal point, and/or leading or trailing minus or plus signs. If the cursor is on or after a decimal point, only the decimals, including the period, are returned.

The number may not contain any change of text emphasis attributes such as boldface, underlines, or double underlines. For example, the number 4428 will return a zero value because the underline attribute changes halfway through the number. The number 4428 will return the correct value only if it is preceded by a space. A number that is part of a sequence of emphasized characters will return the correct value. The cursor moves past the end of the number on the document screen.

occur

Type:	string
Value:	the number of segments in a delimited string
Syntax:	variable = **occur**(expression1,expression2) conditional function(**occur**(e1,e2) operator expression) call function(**occur**(expression1,expression2))

The **occur** function provides the number of segments in expression1 delimited by the character in expression2. The character in expression2 must be enclosed in quotes. If a variable is used in expression2, it does not need to be quoted. (See the **seg** function for a description of delimiting characters.)

page_no

Type:	document reading
Value:	page number for the cursor
Syntax:	variable = **page_no** conditional function(**page_no** operator expression) call function(**page_no**)

The **page_no** function returns the page number of the cursor location in the document.

pic

Type:	mathematical
Value:	**pic** does not return a value
Syntax:	variable = **pic**(expression1,expression2) call function(**pic**(expression1,expression2)) conditional function(**pic**(e1,e2) operator expression)

The **pic** function formats the number in expression1 with common numeric symbols such as $ or -. Expression1 may be a numeric string, a variable with a numeric value, or a function that returns a numeric value. Expression2 specifies the symbols to be used by expression1. Expression2 must be a quoted string or a variable that contains the quoted string. Expression2 may have one or more of the following symbols:

Symbol	**Meaning**
$	Precede number with a dollar sign
+	Precede number with a plus sign
-	Follow number with a minus sign
,	Insert a comma every three digits if number is greater than 999
.	Insert a decimal point two decimal places from right of number

position

Type:	document reading
Value:	character position for the cursor
Syntax:	variable = **position** conditional function(**position** operator expression) call function(**position**)

The **position** function returns the character position of the cursor location in the document.

posmsg

Type:	display
Value:	**posmsg** does not return a value
Syntax:	call **posmsg**(expression1,expression2,expression3)

The **posmsg** function displays expression3 at the line specified by expression1 and the character position specified by expression2. Expression3 may be an alphabetic or numeric string, a variable, or a function. Permissible screen parameters for posmsg are lines 1 through 25 and positions 1 through 80. Messages extending beyond position 80 or line 25 will result in screen display anomalies. (See the "Display Functions" section that appears later in this chapter for additional information about screen display functions.)

The message posted by **posmsg** can be cleared by the **clrpos** function or, at the end of glossary execution, by pressing CTRL w (simultaneously press the CTRL key and w) or by including the CTRL w statement "\027" in the program. The **posmsg** function does not replace characters on the screen. It is a temporary display.

prompt

Type:	display
Value:	string
Syntax:	call **prompt**(expression)

The **prompt** function displays the value of the expression highlighted in the prompt section of the screen (line 1, characters 50 to 79). The length of the **prompt** string cannot exceed 30 characters. Strings longer than 30 characters result in screen display anomalies. The **prompt** message can be cleared by the **clrpos** function; by including the null **prompt** statement, **call prompt("")** in the program; by pressing CTRL/w; by including the CTRL/w statement "\027" in the program; or by invoking an editing function, like INSERT or DELETE.

right_margin

Type:	document reading
Value:	1 if true, 0 if false
Syntax:	conditional function(**right_margin**) conditional function(!**right_margin**) variable = **right_margin** call function(**right_margin**)

The **right_margin** function returns a value of true if the cursor is on the last character of a line. Otherwise, it returns a value of false. When it is preceded by the logical not operator ! the combination !**right_margin** returns a value of true only if **right_margin** is not on the last character of a line.

round

Type:	mathematical
Value:	rounded value of a numeric expression to the specified decimal place
Syntax:	variable = **round**(expression1,expression2) conditional function(**round**(e1,e2) operator expression) call function(**round**(e1,e2)

The **round** function rounds expression1 at the number of decimal places specified by expression2. If the fractional part beyond the specified decimal place is 5 or greater, 1 is added to the last decimal. If it is less than 5, nothing is added to the last decimal.

seg

Type:	string
Value:	the string segment from expression3 to expression4 or to the end of the entire string if expression4 is omitted

Syntax:	variable = **seg**(expression1,expression2,expression3)
	variable = **seg**(e1,e2,e3,e4)
	conditional function(**seg**(e1,e2,e3,e4)) operator expression)
	call function(**seg**(e1,e2,e3,e4))

The **seg** function evaluates strings whose discrete segments are separated by a specific delimiting character. Examples are a social security number segmented with hyphens, "526-43-9090," or a string segmented by spaces, "table 5x10 15 $95." Any character may be used to segment the string. This character is called the delimiter.

Expression1 is the entire segmented string. Expression2 is the character used for the segment delimiter. The character in expression2 must be enclosed in quotes. If a variable is used in expression2, it doesn't need to be quoted.

The value returned by **seg** is any portion of the string beginning with the segment specified by expression3 and ending with expression 4. If expression4 is omitted, the value returned begins at the segment specified by expression3 and concludes at the end of the entire string.

spacing

Type:	document reading
Value:	present format setting for vertical line spacing
Syntax:	variable = **spacing** conditional function(**spacing** operator expression) call function(**spacing**)

The **spacing** function returns the vertical line spacing of the closest format line above the cursor location in the document. The vertical line spacing is displayed on the document editing screen in two locations: 1. In the second status line following the word "Spacing." 2. In the first position of the format line. To change the vertical line spacing during program execution use the keyword combination: command "s n" where "n" stands for the vertical line space number or letter. The line number returned by the **line** function reflects the Spacing setting, not the relative line position displayed on the editing screen.

status

Type:	display
Value:	string
Syntax:	call **status**(expression)

The **status** function displays the value of the expression in the status area of the screen (line 25, characters 27 to 50). The length of the **status** string cannot exceed 24 characters. Strings longer than 24 characters result in screen display anomalies.

The **status** message can be cleared by the **clrpos** function; by including the null **status** statement, **call status("")** in the program; by pressing CTRL/w; by including the CTRL/w statement "\027" in the program; or by invoking an editing function, like INSERT or DELETE.

sub

Type:	string
Value:	**sub** performs a substitution function; if it can be said to return a value, the value would be the substitution segment in expression5
Syntax:	variable = **sub**(e1,e2,e3,e4,e5) conditional function(**sub**(e1,e2,e3,e4,e5) operator expression) call function(**sub**(e1,e2,e3,e4,e5))

The **sub** function substitutes the string in expression5 for the string segments specified by expression3 and expression4. Expression1 gives the entire segmented string. Expression2 gives the delimiter character used to segment the string in expression1. Expression3 gives the segment number where the substitution should begin. Expression4 gives the segment number where the substitution should end. Expression5 gives the string to be substituted for expression3 through expression4. (See the **seg** function for a description of delimiting characters.)

substr

Type:	string
Value:	the string segment extracted from a string

Syntax:	variable = **substr**(expression1,expression2)
	variable = **substr**(expression1,expression2,expression3)
	conditional function(**substr**(e1,e2,e3) operator expression)
	call function(**substr**(e1,e2,e3))

The **substr** function returns as its value a substring that is extracted from a string. The string is specified by expression1, which may be a numeric or alphabetic string, a variable, a function, or a math calculation. It is taken from the character position specified in expression2 to the end of the string. Expression3 is optional. If it is used, the substring is taken from expression2 to the character position specified by expression3.

text

Type:	document reading
Value:	text extracted from a document from expression1 through expression2
Syntax:	variable = **text**(expression1,expression2) call function(**text**(expression1,expression2)) conditional function(expression operator **text**(expr1,expr2))

The **text** function extracts text from a document between the document locations specified by expression1 and expression2. Document locations are specified in the form page, line, position. Each expression must be enclosed in quotation marks. For example, the statement:

```
variable = text("1,14,22","2,17,33")
```

assigns the block of text from page 1, line 14, position 22, through page 2, line 17, position 33, to the variable.

The **loc** function may be used to specify beginning or ending text extraction locations. (The **loc** function returns the current cursor location in the document.)

```
variable = text(loc,"4,1,6")
```

You can extract one character at the cursor position by using the statement

```
variable = text(loc,loc)
```

Use **text** when you want to know if the cursor is on a screen symbol such as RETURN, TAB, INDENT, DEC TAB, or CENTER.

The following syntax example uses a conditional **if** and the **text** function to determine if the cursor is on a RETURN symbol.

```
ret1 = "\\B\\\012"
ret2 = text(loc,loc)
if(ret1 =(ret2) {...}
```

The value in **ret1** is the WORD ERA document control code for the RETURN symbol you see on the screen. (The RETURN symbol looks like a left-facing triangle.) Appendix C provides more information about WORD ERA document control codes.

The value (text from the document) returned by **text** can be assigned to a variable or placed directly in the document by using the **finsert** function, as in

```
call finsert(text("1,4,1","1,10,31")
```

You must use **finsert** to insert the value returned by **text** into your document. The **text** function retains WORD ERA document control codes for screen symbols such as RETURN, TAB, or CENTER in the text that it reads from the document. The **finsert** function recognizes these document control codes and inserts their equivalent screen symbols in the document.

Format lines in the extraction location in the document are not retained by **text**. When **text** values are inserted they observe the closest format line above the insertion location.

text_len

Type:	document reading
Value:	present default setting for document text length
Syntax:	variable = **text_len** conditional function(**text_len** operator expression) call function(**text_len**)

The **text_len** function returns the current text length setting of the document. (The text length setting is shown on the second status line on the edit screen.)

time

Type:	Operating system access
Value:	the current system time
Syntax:	variable = **time** conditional function(**time** operator expression) call function(**time**)

The **time** function returns the current system time in the form 09:40:00. The time is represented in military or 24-hour, format. For example, 2:00 in the afternoon is shown as 14:00:00.

top_page

Type:	document reading
Value:	1 if true, 0 if false
Syntax:	conditional function(**top_page**) conditional function(!**top_page**) variable = **top_page** call function(**top_page**)

The **top_page** function returns a value of true if the cursor is on the first character of the first line of a page. Otherwise, it returns a value of false. When it is preceded by the logical not operator ! the combination !**top_page** returns a value of true only if **top_page** is not on the first character of the first line of a page.

true

Type:	logical
Value:	provides a numeric value of 1
Syntax:	variable = **true** conditional function(expression operator **true**) call function(**true**)

The **true** function is used to provide a true value for a variable or a function. It can also serve as an expression in a conditional statement. The **true** function always returns a value of 1.

truncate

Type:	mathematical
Value:	truncated value of a numeric expression to the specified decimal place
Syntax:	variable = **truncate**(expression1,expression2) conditional function(**truncate**(e1,e2) operator expression) call function(**truncate**(e1,e2))

The **truncate** function truncates expression1 at the number of decimal places specified by expression2. The fractional part beyond the specified point is deleted regardless of its value.

unixfun

Type:	UNIX access
Value:	**unixfun** does not return a value
Syntax:	call **unixfun**(expression)

The **unixfun** function executes the operating system command in the expression. The output of the command is not written to the document. The **unixfun** function operates similarly to the WORD ERA function "Command !." The Operating System Access Functions section in Chapter 11 gives examples of how to use both **unixfun** and command "!" in glossary programs.

unixpipe

Type:	Operating system access
Value:	returns the output of a operating system command
Syntax:	variable = **unixpipe**(expression1,expression2)

The **unixpipe** function assigns the standard output of a operating system command to a variable. The data in expression2 is piped to the command in expression1.

Expression1 is the entire operating system command line. Expression1 must be enclosed in quotation marks.

Expression2 is data required for the command line. In entry a below, the operating system command expr is accessed for a simple calculation. Variable **a** is assigned the calculation. Since expr only requires command line input, **b** is used as a null expression for expression2.

```
entry a
{
        a = "expr 44 + 77"
        b = ""
        x= unixpipe(a,b)
        call finsert(x)
}
```

Entry b is another example of the **unixpipe** function. This entry uses the **keys** function to assign the variables for **unixpipe**. If the command in variable **a** does not require data from variable **b**, enter a null by pressing EXECUTE when the program pauses for keys entry to variable **b**.

```
entry b
{
        "Enter a: "
        a= keys
        call feed(a)
        insert return execute
        "Enter b :"
```

```
    b = keys
    call feed(b)
    insert return(2) execute
    x = unixpipe(a,b)
    call finsert(x)
}
```

You must use the **finsert** function instead of the **feed** function to type the value returned by **unixpipe** in the document.

The **unixpipe** function operates similarly to the WORD ERA function "Command |." The Operating System Access Functions section in Chapter 11 gives examples that use both **unixpipe** command "|" in glossary programs.

while

Type: conditional

Value: **while** does not return a value

Syntax:

```
while(expression)
{
    statement or statements
}
```

(expression) may consist of various combinations of expressions and operators:

```
    while(expression operator expression)
```

As long as proper parenthetical syntax is followed, the argument to **while** may contain a theoretically unlimited number of expressions and operators.

Multiple {statements} to **while** must be enclosed in braces.

The **while** function allows repeated execution of a statement or statements based on true or false conditions in the document. The true or false conditions are specified by the expressions in the argument to **while**. The statement enclosed in braces is executed repeatedly as long as the value of the expression in the argument to **while** remains true.

When the value becomes false, the statement is not executed and the program continues after the closing brace in the **while** statement. The test of the expressions takes place before each execution of the statement.

word

Type:	document reading
Value:	word at cursor location
Syntax:	variable = **word** conditional function(**word** operator expression) call function(**word**)

The **word** function passes the word found at the cursor position or the nearest word to the right of the cursor to the variable. When the **word** function is used during program execution, the cursor is moved past the end of that word on the document screen. A word is defined as a sequence of characters, including punctuation, that begins and ends with a space or spaces. Spaces surrounding the word are not stored in the variable.

CHAPTER 11

FUNCTION USAGE LIST

FUNCTION USAGE LIST

Functions can be grouped by the type of actions they perform when a glossary program is executing. For example, when you use display functions such as **prompt**, **clrpos**, and **status**, you can place messages on the text document editing screen. Document reading functions such as **char** and **page_no** return information about the text document. Operating system access functions allow you to use a wide range of operating system commands in your glossary programs.

In this list, functions are organized by the usage groups shown in Table 8. The introduction to each usage group describes the functions in that group and provides programming suggestions. Glossary entry examples are provided for each group.

See the Alphabetical List of Functions in Chapter 10, and other chapters in this book for additional examples and more detailed information about specific functions.

TABLE 8. Usage Groups and Their Functions

Usage Group	Function
Conditional Functions	**do while**
	if
	if else
	while

Control Functions	call exit glossary jump
Display Functions	clrpos cursor display error posmsg prompt status
Document Reading Functions	beg_doc bot_page char end_doc left_margin line loc number page_no position right_margin spacing text text_len top_page word
Document Writing Functions	feed finsert
Error and Logical Functions	globerr false true
Interactive Functions	key keyin keys keysin

Mathematical Functions	abs max min num number pic round truncate
Operating System Access Functions	date time unixfun unixpipe
String Functions	cat index len max min occur seg sub substr

CONDITIONAL FUNCTIONS

do while

if

if else

while

Using Conditional Functions

Use a conditional function when you want your program to ask a question and execute different statements based on the response. Typical questions are: What is the cursor position in the document? What did the operator just type? What is the current value of a particular variable, string, or function?

A conditional function is part of a conditional statement. The conditional statement includes the function, its arguments and expressions, and the statement or statements that are executed as a result of the conditional test. The conditional test is based on the evaluation of expressions in the argument to the function. Conditional functions are typically used to perform a branch or a loop, call a subroutine, or terminate the program.

The **while** and **do while** functions perform conditional loops. The statement repetition action of a loop is combined with the conditional test. Remember that **do while** always executes its statements at least once because the conditional test is made after the statement is executed. The **while** function performs its test before its statements are executed. If the condition starts out false, the **while** statement or statements are never executed.

The **if** and **if else** functions perform conditional tests. Their statements are executed based on the true or false result of the test. They do not perform loops as the **while** and **do while** functions do. You have to use a **jump** or **while** statement in combination with **if** to perform a loop.

Chapters 7 and 8 give you in-depth information and program examples for each conditional function.

Entry c is a handy program that boldfaces only alphabetical characters. You can use this entry to boldface a word in parentheses, a word that ends with some form of punctuation, or a word followed by a space.

```
entry c
{
      mode"b"
      while(((char >= "A") & (char <= "Z")) | ((char >= "a") & (char <= "z")))
      {
            right
      }
      mode "b"
}
```

This entry uses the conditional loop **while**, the logical operators & and |, and the **char** function. It makes use of the ASCII collating sequence to restrict the boldface to alphabetic characters. (The ASCII sequence is described in Appendix C.) You can use this entry for any text emphasis mode by changing mode "b" to another mode.

Entry c could also be written using a **do while** statement, as shown in entry d. There is a subtle distinction in the way each entry performs its operation. Entry c, which uses **while**, will terminate execution if the word begins with an excluded character such as a number or symbol. Entry d, which uses **do while**, always executes its statement once regardless of the character. You can prove this by trying both entries on the character combination: 2word

Entry c will not bold "2word" because the test is made before the statements are executed. Entry d does bold "2word" because the statements are executed before the test is performed.

```
entry d
{
       mode "b"

       do
       {
            right
       }
       while(((char >= "A") & (char <= "Z")) | ((char >= "a") & (char <= "z")))


       mode "b"
}
```

See entry D under the section Mathematical Functions and entry 1 under Document Reading Functions section in this chapter for examples that use the conditional **if** and **if else** statements.

CONTROL FUNCTIONS

call

exit

glossary

jump

Using Control Functions

Use control functions to control the statement execution order of your program.

The **jump** statement transfers control to a block of labeled statements in the program. The statements are labeled by an identifying name in brackets called the identifier. You can use **jump** to perform a branch or a loop.

The **call** statement transfers execution control to a built-in function or a subroutine. When a function is used as a statement (rather than an expression) it is always preceded by the **call** statement. A subroutine is another glossary entry in the same glossary document. The **glossary** statement transfers execution control to another entry in the same glossary document. Either **call** or **glossary** can be used to make an entry recall itself.

The **exit** statement causes the program to terminate.

Chapter 8 gives you in-depth information and program examples for each control function.

See entry 1 under the section "Document Reading Functions" in this chapter for an entry that uses control functions. Entry 1 also provides an example of how to construct an entry that switches execution sequence between various parts of the program, depending on conditional tests.

DISPLAY FUNCTIONS

clrpos

cursor (See also Document Writing Functions)

display

error

posmsg

prompt

status

Using Display Functions

You can use display functions in your programs to place messages on the text document editing screen, put the cursor in a specified location, or turn the screen display refresh function off and on.

Following is a brief summary of each display function. Refer to the alphabetical list of functions in Chapter 10 for detailed function descriptions.

clrpos	Clears the editing screen by displaying the number of blanks specified by expression3 at the line and position specified by expressions 1 and 2. Does not overwrite existing text and is cleared by pressing CTRL/w or by the next function that cause the editing screen to refresh. The statement **call clrpos(6,14,10)** clears the screen on line 6 from position 14 through 24
cursor	Sends the cursor to the page, line, and position specified in the expression. The statement **call cursor("2,4,33")** sends the cursor to page 2, line 4, position 33 of the text document.
display	The statement **call display(false)** turns the screen display off, and the statement **call display(true)** turns the screen display on.
error	Displays a message in the error area of the editing screen, line 25, positions 51 through 79.
posmsg	Displays the message in expression3 at the line and position locations specified by expressions 1 and 2. The statement **call posmsg(4,6,"Hello")** displays the word "Hello" at line 4, position 6 on the document editing screen.
prompt	Displays a message in the prompt area of the editing screen, line 1, positions 50 through 79.
status	Displays a message in the status area of the editing screen, line 25, positions 26 through 79.

To best utilize display functions you need to understand how your WORD ERA editing screen works.

THE EDITING SCREEN

When you are working in your document, you are looking at a visual display called the editing screen. The editing screen is a grid that measures 25 vertical lines (rows) by 80 horizontal character positions (columns).

This grid is all you can see of your document at one time even if you have a document that is 60 lines long and 250 characters wide. To see more of your document, you must press Prevscrn or Nextscrn to jump from screen to screen or use the cursor control keys to scroll horizontally or vertically between screens.

Figure 11-1 shows the editing screen grid layout of 25 vertical lines by 80 horizontal positions.

The grid line and position numbers are called the physical editing screen locations. They never change in relation to the changing line and position numbers of your text. The text line and position numbers are called the logical editing screen locations.

PHYSICAL AND LOGICAL LOCATIONS

Physical screen location line and position numbers are not the same as logical screen location line and position numbers. Physical location refers to the non-changing grid in Figure 22 and the reserved areas of the screen shown in Figure 23. Logical locations refer to changing text line and position numbers.

Figure 22 Editing Screen Grid

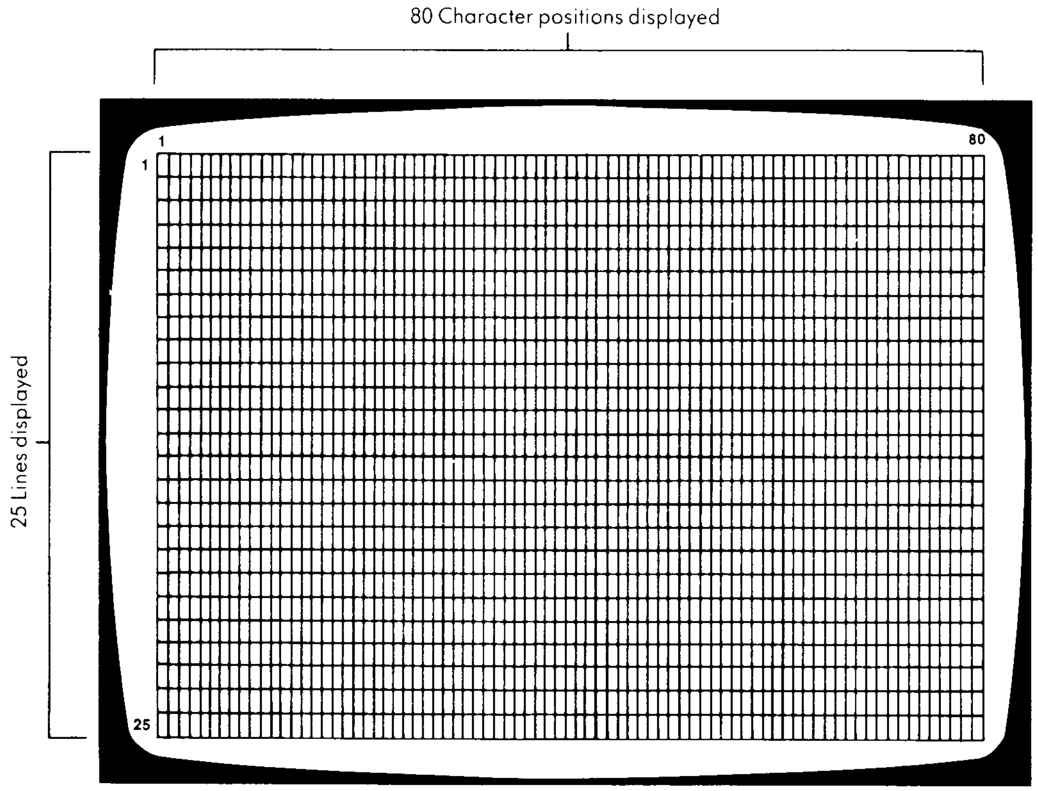
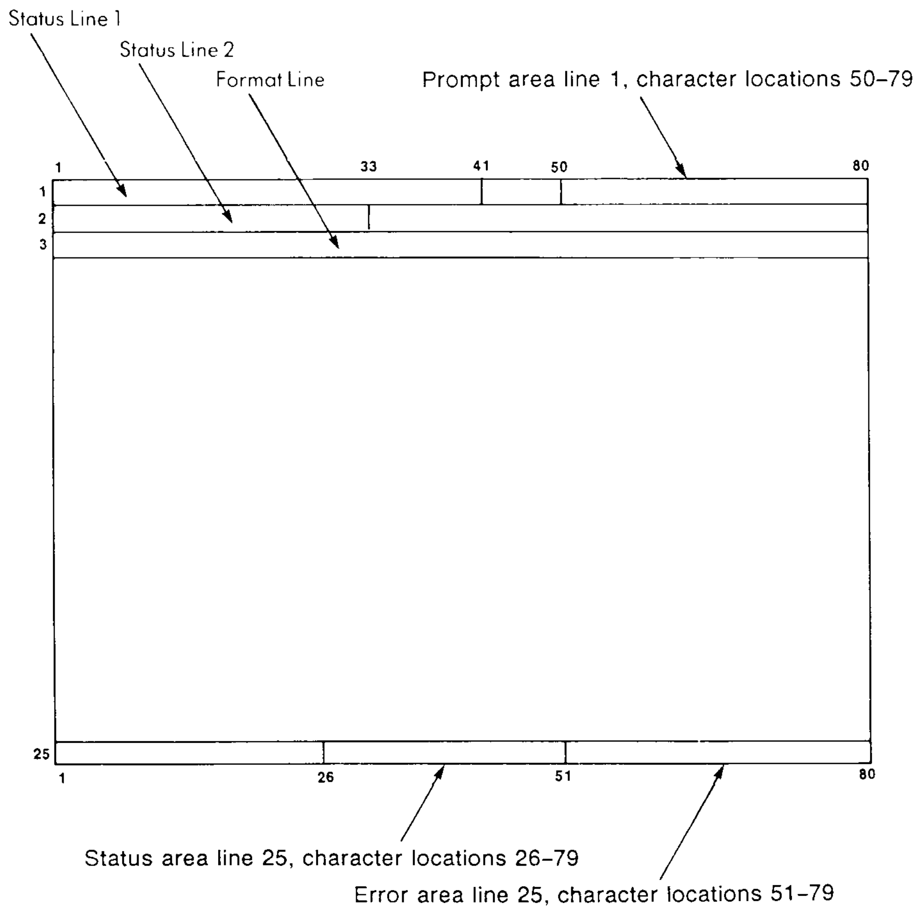


Figure 23 **Editing Screen Reserved Areas**


Status Line 1
Status Line 2
Format Line
Prompt area line 1, character locations 50–79
1
33
41
50
80
1
2
3
25
1
26
51
80
Status area line 25, character locations 26–79
Error area line 25, character locations 51–79

Try the following two short entries to see the difference between a physical location and a logical location. Entry e uses the **posmsg** function to place a message on line 1, position 43. Entry f uses the **cursor** function to send the cursor to line 1, position 1, then inserts a message. (Since the **cursor** function has to go to a page location and **posmsg** does not, be sure you are on page 1 of your document so that you can see both messages displayed at once.)

```
entry e
{
        call psmsg(1,43,"THIS IS A POSMSG ON LINE 1,POS 43."  )
}


entry f
{
        call cursor("1,1,1")
        insert
            "THIS IS TEXT LINE 1, POS 1."
        return(2)
        execute
}
```

Although both **posmsg** and **cursor** specified line 1, the messages appeared on different lines. What is line 1 to the **cursor** function, which uses a logical location, is actually line 4 to the **posmsg** function, which uses a physical location.

Notice that the **posmsg** message was displayed at position 43 so it would not conflict with status line 1. Status line 1 is in a reserved screen area as shown in Figure 23. Reserved screen areas are described later in this section.

If you are trying this exercise and are still in your document, press CTRL and w (CTRL/w). The physical location message disappears because **posmsg** is a display function and is not inserted as text in the document. The message inserted by entry f remains in the document. When you use **posmsg** to display a message over existing text the text is never actually overwritten. This principle is illustrated by entry i in this section.

Physical Screen Locations: From the examples, you can see why the physical screen locations on the grid in Figure 22 are the numbers you must use when you specify line and position locations for the **posmsg** and **clrpos** functions.

The **posmsg** and **clrpos** functions can be positioned at any location on the editing screen grid you choose. Acceptable line locations are 1 through 25. Acceptable character positions are 1 through 80. If you specify a line or position location outside these numbers, the error message bad location will appear during program execution, and the entry will terminate. Messages that extend beyond 80 characters on line 25 will cause screen display anomalies.

The **prompt**, **status**, and **error** functions have their message placement predefined in specific editing screen reserved areas as shown in Figure 23.

Logical Editing Screen Locations: As you can tell from entry e, reserved areas of the editing screen are important considerations when you are using display functions. These areas are reserved for system-generated prompt, status, and error messages.

When you use the **prompt**, **status**, **error**, **posmsg** or **clrpos** functions in reserved areas, you can overwrite system prompts and document status lines. In addition, your display function messages can be totally or partially cleared by system-generated or your own program-generated messages.

Depending on your program, you may want to replace a system message or status line with your own message. You should always try your program in a text document that you keep for testing glossary entries. Check your placement of display functions and see how they interact with system-generated displays. If you are using keywords such as insert, search, delete, copy, or replace, the messages that are integral to these functions will clear your program-generated messages. You may need to put additional messages in your program, or just rely on the system messages if they serve your purpose.

Figure 23 shows the reserved areas of the editing screen. Lines 1 through 3 are reserved for the two document status lines, the format line, and system-generated prompt messages. Line 25 is reserved for system-generated status and error messages.

The **prompt**, **status**, and **error** functions all have predefined message display locations in the reserved areas as follows:

> **prompt** messages display on line 1, positions 50 through 79
>
> **status** messages display on line 25, positions 26 through 79
>
> **error** messages display on line 25, positions 51 through 79.

These function messages display only in their reserved areas. Use the **posmsg** function to display messages anywhere on the editing screen.

As you have seen from entries e and f, the line and position numbers for physical locations are not the same as logical locations for text line and position numbers. Logical locations reflect the number of text lines per page or the number of characters per line.

Logical Editing Screen Locations: Although 25 lines are displayed on the editing screen, only 21 lines are available for text typing. These are lines 4 through 24 on Figure 24, which shows physical screen locations versus logical screen locations. The line indicator in the first status line reflects the number of the text line on the screen. If you place your cursor on Page 1, Line 1, Pos 1 in your document, the cursor is actually on line 4 of the screen display, but the line indicator will read "Line 1."

Think of the logical text location on the screen as a moving picture under a piece of transparent glass. The text is framed by the reserved areas of the screen, which are lines 1-3, and line 25. In Figure 25 text lines 1 through 21 are framed by the reserved areas. The text within the frame can be scrolled up and down (vertical scroll), or from side to side (horizontal scroll).

Figure 24 Physical Screen Locations vs. Logical Screen Locations

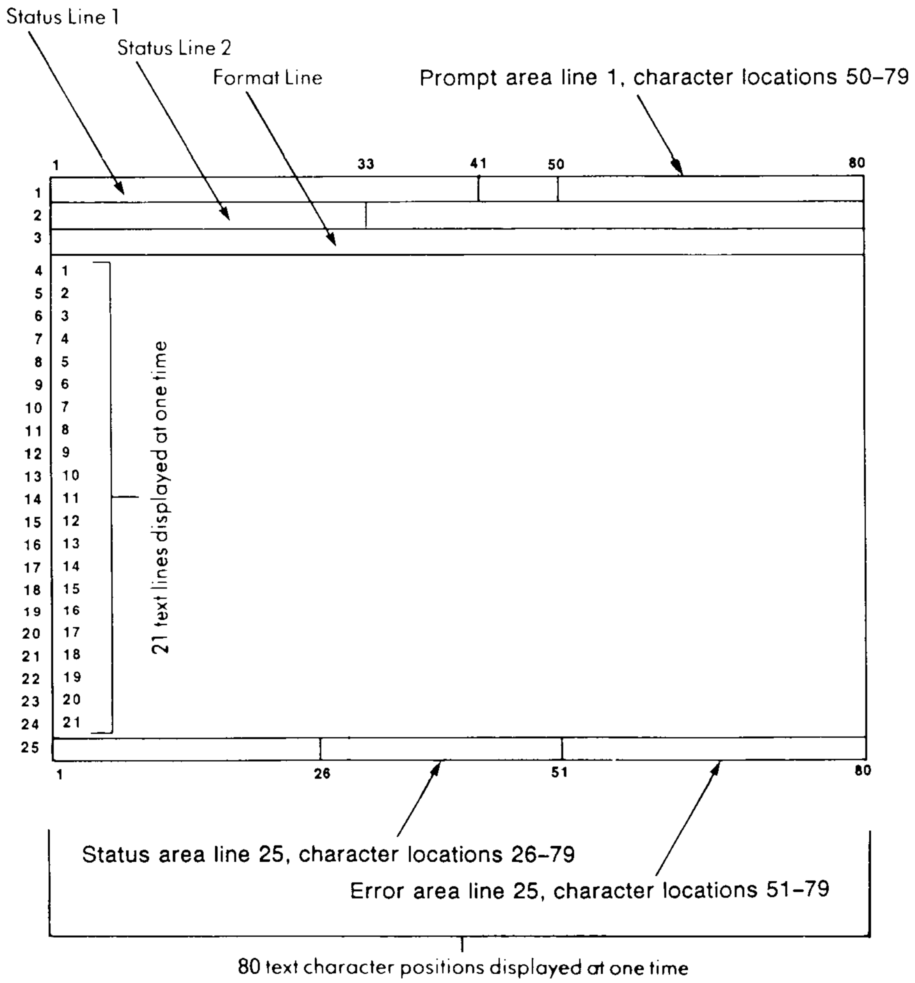
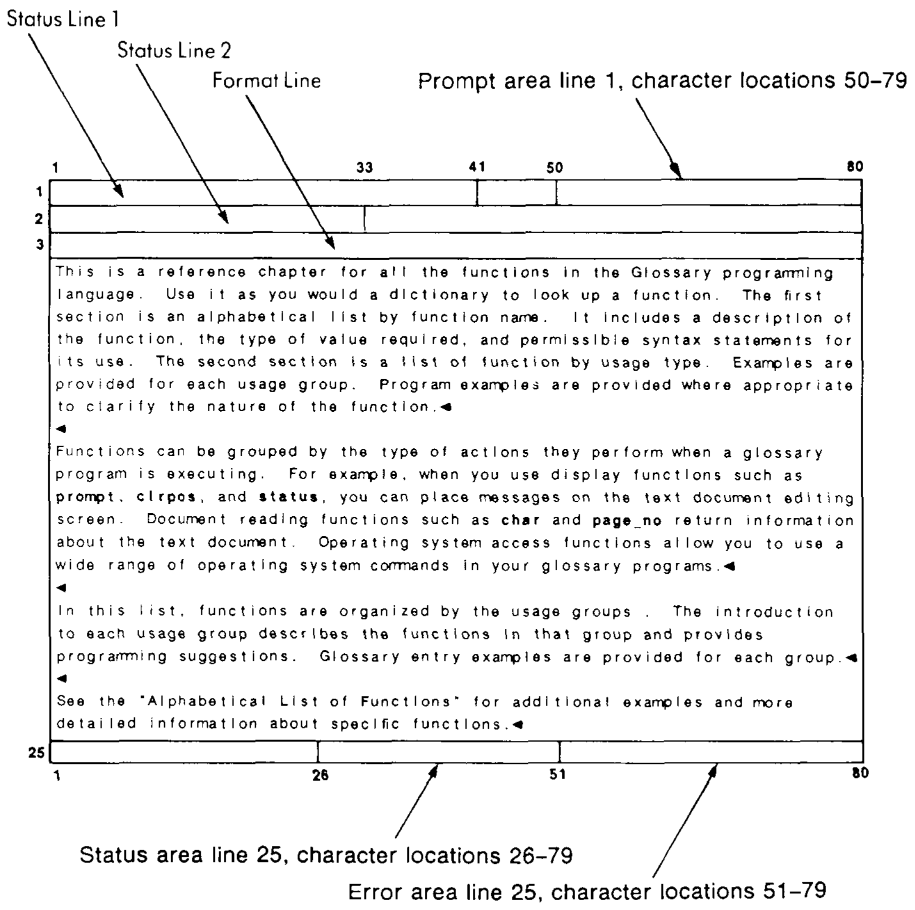

Status Line 1
Status Line 2
Format Line
Prompt area line 1, character locations 50–79
1
33
41
50
80
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
21 text lines displayed at one time
1
26
51
80
Status area line 25, character locations 26–79
Error area line 25, character locations 51–79
80 text character positions displayed at one time

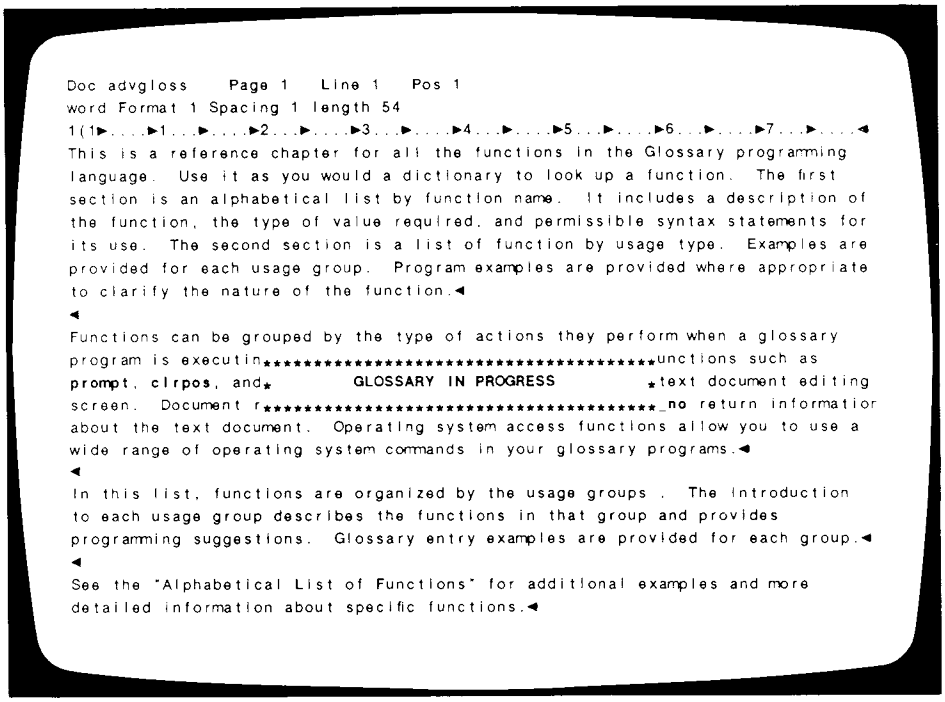
Figure 25 Text is Framed by Reserved Areas of the Screens


Status Line 1
Status Line 2
Format Line
Prompt area line 1, character locations 50–79
1
33
41
50
80
1
2
3
25
1
26
51
80
Status area line 25, character locations 26–79
Error area line 25, character locations 51–79

This is a reference chapter for all the functions in the Glossary programming language. Use it as you would a dictionary to look up a function. The first section is an alphabetical list by function name. It includes a description of the function, the type of value required, and permissible syntax statements for its use. The second section is a list of function by usage type. Examples are provided for each usage group. Program examples are provided where appropriate to clarify the nature of the function.◄
◄
Functions can be grouped by the type of actions they perform when a glossary program is executing. For example, when you use display functions such as **prompt**, **clrpos**, and **status**, you can place messages on the text document editing screen. Document reading functions such as **char** and **page_no** return information about the text document. Operating system access functions allow you to use a wide range of operating system commands in your glossary programs.◄
◄
In this list, functions are organized by the usage groups . The introduction to each usage group describes the functions in that group and provides programming suggestions. Glossary entry examples are provided for each group.◄
◄
See the "Alphabetical List of Functions" for additional examples and more detailed information about specific functions.◄

Messages posted by the **posmsg** and **clrpos** functions can be "pasted" anywhere on the "glass" overlaying the text. They do not become part of the text; they temporarily overlay it. Figure 26 shows text with an overlaid message posted by **posmsg**.

Figure 26 A posmsg Message Temporarily Overlays Screen Text



```
Doc advgloss    Page 1   Line 1   Pos 1
word Format 1 Spacing 1 length 54
1(1►....►1...►....►2...►....►3...►....►4...►....►5...►....►6...►....►7...►....◄
This is a reference chapter for all the functions in the Glossary programming
language.  Use it as you would a dictionary to look up a function.  The first
section is an alphabetical list by function name.  It includes a description of
the function, the type of value required, and permissible syntax statements for
its use.  The second section is a list of function by usage type.  Examples are
provided for each usage group.  Program examples are provided where appropriate
to clarify the nature of the function.◄
◄
Functions can be grouped by the type of actions they perform when a glossary
program is executin****************************************unctions such as
prompt, clrpos, and*          GLOSSARY IN PROGRESS          *text document editing
screen.  Document r****************************************_no return informatior
about the text document.  Operating system access functions allow you to use a
wide range of operating system commands in your glossary programs.◄
◄
In this list, functions are organized by the usage groups .  The introduction
to each usage group describes the functions in that group and provides
programming suggestions.  Glossary entry examples are provided for each group.◄
◄
See the "Alphabetical List of Functions" for additional examples and more
detailed information about specific functions.◄
```

You can demonstrate this concept shown in Figure 26 by trying entry g. Use a text document that has a format line of 250 characters. Place your cursor on position 250 and recall the entry. The message is displayed at the physical screen location of line 4, position 7, which is the logical screen location of line 1, position 187.

```
entry g
{
     call posmsg(4,7,"This is a posmsg on line 4, pos 7,(text line 1).")
}
```

When you have used display functions in a few programs, the physical locations and logical locations will become easy to remember. You can use display functions in a variety of programming situations. They are particularly valuable when you are writing programs for others to use, as you can post non-disruptive messages to the user on the screen during the execution of your program.

Clearing Display Messages from the Screen: Removing your messages is as important as posting them. You can use three methods to clear display functions:

Clear **prompt**, **status**, and **error** messages by system-generated prompt, status, or error messages, or by another message in the currently executing glossary entry. If you use this method you must be sure a replacement message is generated immediately after your message. This method doesn't work for **posmsg** unless its message is posted in a reserved area.

Use the **clrpos** function to replace the message with blanks. Remember, **clrpos** is an overlay of text; it does not replace text in your document. Entry i uses **clrpos** in a **while** loop to clear the entire screen. This method works well for **prompt**, **status**, and **error** messages because they are in reserved (non-text) areas. It doesn't work as well for **posmsg** because you are still obscuring underlying text with blanks.

Use CTRL/w to clear messages. Using CTRL/w in your program is generally the best method for clearing the **posmsg** function. Try recalling entry h, which uses all four message display functions.

```
entry h
{
     call prompt("HI")
     call status("HI")
     call error("HI")
     call posmsg(4,7,"HI")
}
```

After the entry has displayed in your document, press CTRL/w. All messages are cleared. To use CTRL/w in your program, you have to represent it by its octal number, 027. (Octal numbers are described in Appendix C.)

Entry i in this section uses octal 027 (CTRL/w) as a quoted string. Like keyword abbreviations in a quoted string, the octal number is preceded by a backslash.

When the **status** and **error** functions are displayed simultaneously in a program, be sure the **status** message does not extend into the **error** message area (positions 51 through 79). Any characters in the **status** message at position 51 and beyond will be overwritten by the **error** message which begins at position 51.

The display Function

The editing screen display is restored each time you perform a standard function such as Insert, Delete, Copy, or Replace. When these standard functions are part of your program, the screen is restored during program execution, just as it is while you are editing. Although the glossary restores the screen faster than normal editing does, it still slows the program down.

You can use the **display** function to turn the screen display restore off during program execution. This is particularly valuable for lengthy programs, as the runtime is reduced because the screen does not have to be continually restored.

A good example is entry l in Chapter 8, which uses the statement **call display(false)** at the beginning of the program to turn the display off. The screen display is turned back on at the end of the program by the statement **call display(true)**.

If you typed and recalled entry l in Chapter 8, using the "Amalgamated Widgets" example, you observed a semi-static display. Try removing both **display** statements from the entry and running it. Notice that the entry runs longer because the screen is being restored.

Entry i shows you how to use a combination of **display**, **posmsg**, and **clrpos** to speed up entries and provide notification that an entry is running. Entry i also uses the keyword statement command format to turn off the status line update. This is another technique to speed up glossary program execution time.

Entry i incorporates a variation of entry l shown in Chapter 8.

The "Amalgamated Widgets Month End Sales Statement" from Chapter 8 is repeated in this chapter for your reference in trying entries i (the example is on Page N of glossary document **gloss2b** on the Glossary Diskette.

The **posmsg** messages in entry i use octal numbers and attribute codes to display the messages in reverse video and flash modes. (Keyword abbreviations for modes cannot be embedded in **posmsg** messages).

The octal numbers and attribute codes shown in **posmsg** strings apply to terminals manufactured by Fortune Systems' Corporation. Consult your terminal manufacturer for information about character sets, attributes, and octal equivalents that apply to your terminal.

If the **posmsg** strings shown in programs in this book do not work correctly on your terminal, remove the octal and attribute codes from the message string. For example, if the **posmsg** statement

```
call posmsg(7,26,"\034HD \034Id \034HBGLOSSARY IN PROGRESS\034Ib
\034HD \034Id")
```

does not display properly, change it to:

```
call posmsg(7,26,"GLOSSARY IN PROGRESS")
```

Appendix C describes octal numbers and attribute codes.

The "Glossary in Progress" flashing module in entry i could be placed in a separate entry and used as a subroutine with many different entries.

```
entry i
{
     call display(false)
     command format

     linenumber = 1
     while(linenumber < 25)
     {
         call clrpos(linenumber,1,80)
         linenumber += 1
     }

call posmsg(5,26,"\034HD                    \034Id")
call posmsg(6,26,"\034HD  \034Id                    \034HD  \034Id")
call posmsg(7,26,"\034HD  \034Id  \034HBGLOSSARY IN PROGRESS\034Ib
\034HD  \034Id")
call posmsg(8,26,"\034HD  \034Id                    \034HD  \034Id")
call posmsg(9,26,"\034HD                    \034Id")

     salesqty = 0
     priceper = 0
     grossale = 0
     mfgcosts = 0
     netsales = 0

     salestotal = 0
     grosstotal = 0
     nettotal = 0

     [loop]

     goto decimaltab
         if(globerr)
         {
              jump total
         }

     salesqty = number
         salestotal += salesqty
     priceper = number
     grossale = priceper * salesqty
         grosstotal += grossale
     right
         call finsert(pic(grossale,"$,"))
```

```
    mfgcosts = number
    netsales = grossale - mfgcosts * salesqty
        nettotal += netsales
    right
        call finsert(pic(netsales,"$,"))

    jump loop

    [total]

    {
        command search "TOTAL" execute cancel
        goto right
        insert
             decimaltab call feed(pic(salestotal,","))
             decimaltab
             decimaltab call feed(pic(grosstotal,"$,"))
             decimaltab
             decimaltab call feed(pic(nettotal,"$,"))
        execute

        return(2)

        "Gross sales: "
        call feed(pic(grosstotal,"$,"))
        " are calculated by multiplying the quantity, "
        call feed(pic(salestotal,","))
        ", times price per each." return(2)
        "Net sales: "call feed(pic(nettotal,"$,"))
        " are calculated by multiplying manufacturing cost per each times
        quantity, and subtracting the result from gross sales."
        return(2)
    }
    "\027"

command format
call display(true)

}
```

Entry i uses the same "Amalgamated Widgets Month End Sales Statement" example as entry l in Chapter 8. You can find the example on Page N of glossary document gloss2b on the Glossary Diskette.

The cursor Function

The **cursor** function can be considered both a display and a document writing function. You can move the cursor to any logical location you choose. The cursor cannot be placed in a reserved screen area unless you call it there with a combination of **posmsg** and **key** or **keys** functions. Entry D (under the section Mathematical Functions in this chapter) gives you an example of moving the cursor with the **posmsg** and **key** functions.

Try writing some sample entries to test the **cursor** function. Examples are shown in entries j and k. For additional information about the **cursor** function, see the section "Document Writing Functions" in this chapter.

```
entry j
{
        call cursor("4,6,22")
}


entry k
{
        call cursor("1,47,2")
        insert
            "COMPANY CONFIDENTIAL"
        execute
}
```

Entry j sends the cursor to page 4, line 6, position 22. Note that the expression to **cursor** is a quoted string with its parts separated by commas. Entry k sends the cursor to page 1, line 47, position 2, then inserts the string "COMPANY CONFIDENTIAL." For additional information about the cursor function see the section Document Writing Functions in this chapter.

DOCUMENT READING FUNCTIONS

beg_doc

bot_page

char

end_doc

left_margin

line

loc

number

page_no

position

right_margin

spacing

text

text_len

top_page

word

Using Document Reading Functions

During program execution, document reading functions return values from the text document that reflect the cursor position, status line information, or format line information. Document reading functions can be categorized by different types of cursor location functions, as follows.

Cursor Location Functions

Cursor location functions are grouped into two types:

those that return numeric or alphabetic string values, and

those that return true or false values.

Numeric or Alphabetic Values: The **line**, **loc**, **page_no**, and **position** functions return numeric values equal to the cursor location in the text document. The **number** function returns a numeric value equal to the number at the cursor location. The **char**, **text**, and **word** functions return alphabetic string values equal to the character, text block, or word at the cursor location.

Some points to remember about using the **text** function are that the **finsert** function must be used to insert the value returned by **text** in the document; format lines in the document are not retained by the **text** function; when the value of **text** is inserted, it observes the format line immediately above the insertion location. See the alphabetical listing for **text** in Chapter 10 for additional information. Entry b in Chapter 12 shows you an interesting way to use the **text** function.

True or False Values: The **beg_doc**, **bot_page**, **end_doc**, **left_margin**, **right_margin**, and **top_page** functions return true or false (1 or 0) values based on the cursor location in the text document. The two most common conditional tests for functions that return true or false values, are the **if** test and the **if** not test (using the logical not operator !) shown in entry 1.

```
entry 1
{
        if(top_page)
        {
            jump legend
        }
        else if(!top_page)
        {
            goto up jump legend
        }
        [legend]
        insert
            center "For Immediate Release" return(2)
        execute
}
```

Entry l performs two conditional tests on the cursor position. If the cursor is at the top of the page **if(top_page)** the string is inserted. If the cursor is not at the top of the page it is sent there by the statement goto up, then the string is inserted. Note that the second test, **if(!top_page)** uses the logical not operator.

You could also write the conditional tests by literally checking the numeric true or false values as shown in entry m. This method is a bit more cumbersome that entry l, but works equally well. Again, illustrating that there is more than one way to write a glossary entry.

```
entry m
{
        if(top_page == 1)
        {
            jump legend
        }
        else if(top_page == 0)
        {
            goto up jump legend
        }
        [legend]
        insert
            center "For Immediate Release" return(2)
        execute
}
```

Format and Status Line Functions

The **spacing** function returns the vertical spacing value for the current format line. The **text_len** function returns the current text length default for the document.

DOCUMENT WRITING FUNCTIONS

cursor (See also Display Functions)

feed

finsert

Using the feed and finsert Functions

The **feed** and **finsert** functions write the values returned from a function or a variable in your document. Both functions must be preceded by the **call** function when they are used as statements (rather than expressions). There are three major differences between **feed** and **finsert**: document writing performance, treatment of WORD ERA document control codes, and treatment of keyword abbreviations.

Document Writing Performance: The **feed** function writes characters in the document as if they were being typed from the keyboard (except much faster). If the cursor is on existing text in the document when feed is called by the program, the text will be overwritten. You can avoid overwriting by using the keywords insert and execute as part of the **feed** statement. The following example inserts the date at the cursor location in the document.

```
insert call feed(date) execute
```

The **finsert** function inserts characters in the document. Existing text is not overwritten, and the action is very fast since characters or blocks of text are inserted all at once. You do not have to use the keywords insert and execute since insertion is automatically performed by **finsert**. Using finsert, the statement example above is written as

```
call finsert(date)
```

Treatment of Document Control Codes: The **finsert** function recognizes and writes WORD ERA document control codes as screen graphics, such as a left-facing triangle for Return, a diamond for Center, or an arrow for Indent. Document control codes can be part of values returned by the **text** function, values used by **unixpipe**, values assigned to variables, or values returned by document reading functions.

The **feed** function treats document control codes as string values and writes them as strings, such as \B\ (return), \c\ (center).

You can demonstrate the different ways **feed** and **finsert** treat document control codes by trying entries n and o. When you recall these entries, place your cursor on a screen graphic such as Return, Indent, or Decimaltab in your text document. The **feed** function types the control code, and the **finsert** function types the actual screen graphic.

```
entry n
{
    character = char
    insert
        call feed(character)
    execute
}


entry o
{
    character = char
    call finsert(character)
}
```

Appendix C gives you a list of document control codes and tells you how to use them in your glossary programs.

Treatment of Keyword Abbreviations: You must use **feed** to write string values that contain keyword abbreviations such as \r, \c, or \t. The **finsert** function does not recognize keyword abbreviations.

CTRL/Y Characters: Control Y characters are special characters that are accessed by typing CTRL/y then typing a character. Control Y characters are most frequently used for coding laser printer fonts and typing foreign or accented characters. Both **finsert** and **feed** recognize and print Control Y characters.

Using the cursor Function

While not strictly a document writing function, you can control **cursor** location in the document with the cursor function. You can send the cursor to a specified location, then call **feed** or **finsert** to write a value.

The **cursor** function uses the logical screen location line and position numbers discussed in the Display Functions section in this chapter. The cursor cannot be placed in the reserved screen areas, lines 1 through 3 and line 25.

Open and Unopened Editing Screen Areas: When you are using the **cursor** function, you must consider unopened areas of the screen. For example, assume that your text begins on page 1, line 1, with an indent set at position 6. The text ends on line 3, position 36. The screen is "open" from line 1, position 1, to line 3, position 36. The remainder of the screen is "unopen" because it does not contain characters.

The statement: **call cursor("1,3,48")** would send the cursor to line 3, position 36, because position 48 is not an open area of the screen. The **cursor** function gets as close as it can to the specified location.

When a program is executing in a document, you can't always know which areas of the screen are open or closed. If this is a concern in your program, assign the desired cursor position to a variable. Then use the **loc** function to return the cursor position and compare it to the variable.

Entry p gives you an example that uses the **cursor** and **loc** functions to control the cursor position. If you want to try this entry, set up a document with a return (no text or spaces) on line 22, position 1, of page 4. The cursor is not able to go to position 12 **(curpos = "4,22,12")** because the screen area is not open at that position.

The cursor is called and sent to the location specified by **curpos**. The arrival location of the cursor is checked by comparing **loc** against **curpos**. If they don't match, the incorrect location is displayed in the **status** area, and the **error** message tells you the cursor is in the wrong location. If they do match, **dollars** is inserted by **finsert** at the **cursor** location.

The program sends the cursor as close as it can get, "4,22,12", then allows you the option of moving the cursor to position 12. To move the cursor, you have to space to position 12, which opens that screen area, then press EXECUTE. If you don't want to move the cursor, type **quit** and press EXECUTE. The **exit** statement terminates the entry.

```
entry p
{
     dollars = $4,782.25
     curpos = "4,22,12"
     call cursor(curpos)
          if(loc != curpos)
          {
               call status(loc)
               call error("Cursor is in wrong location")
               call posmsg(1,43,"\034H'Move cursor to 4,22,12?\034I'")
               call posmsg(2,43,"\034H'Type y & EXECUTE\034I'")
               call posmsg(3,43,"\034H'Quit? type quit & EXECUTE\034I'")
               response = keys
               "\027"
                    if((response == "y") | (response == "Y"))
                    {
                         call posmsg(1,43,"\034H'Move cursor to 4,22,12\034I'")
                         call posmsg(2,43,"\034H'& press EXECUTE\034I'")
                         call keysin
                         call finsert(dollars)
                         "\027"
                         exit
                    }
                    if((response == "quit") | (response == "QUIT"))
                    {
                         exit "\027"
                    }
          }
     call finsert(dollars)
}
```

ERROR AND LOGICAL FUNCTIONS

false

globerr

true

Using the globerr Function

Use the **globerr** function to trap standard word processing function errors. A standard function such as Search, Nextscrn, Prevscrn, or Go To Page [symbol], is considered to be in an error condition when it sounds a beep because it cannot actualize its function. Search beeps when it cannot find another instance of the word it is searching for. Nextscrn beeps when there is no next screen to go to.

The **globerr** function helps you to branch, loop, or terminate an entry gracefully if a standard function fails. Entry r and entry s are examples that use the **globerr** function.

```
entry r
{
      search "manufacturer" execute
          if(globerr)
          {
                execute exit
          }
      cancel

      insert
          "computer "
      execute
}


entry s
{
      while(!globerr)
      {
          goto nextscrn
          call finsert(text("1,2,1","1,6,27"))
      }
}
```

Using true and false Logical Functions

Use the **true** and **false** functions to assign logical values or to perform logical comparisons with other values. The **true** function always returns a value of 1. The **false** function always returns a value of 1. Entries f and g in Chapter 6 are examples that use **true** and **false** functions.

Entry C under the Mathematical Functions section in this chapter shows another way to use logical functions.

INTERACTIVE FUNCTIONS

key

keys

keyin

keysin

Using Interactive Functions

Interactive functions let you stop the program so that you can type data from the keyboard. There are two types of interactive functions, the **key** and **keys** functions, which return their input to a variable or to a function, and the **keyin** and **keysin** functions, which type their input directly in the document.

The key and keys Functions

When you use the **key** and **keys** functions, the data is stored in a variable or used by a function. It is not typed in the document unless you use **feed** or **finsert** statements. There are two ways to write **key** or **keys** input to the document with **feed** (or **finsert**, which is interchangeable with **feed** in most instances):

> Assign **key** or **keys** input to a variable, then write the value to the document by using the statements
>
> ```
> variable = key
> call feed(variable)
>
> variable = keys
> call feed(variable)
> ```

Use the following **feed** statement (in this case, **key** or **keys** is not stored in a variable):

call feed(key) or **call feed(keys)**

The key function: The **key** function accepts one typed key from you, then immediately continues program execution.

Any key on the keyboard (including Control Y character combinations) is accepted by **key** and may be assigned to a variable. Because the character you type as input to **key** does not appear on the screen you may want to use a conditional statement to check the validity of the entered character.

Entries t and u show two methods you can use to validate **key** entry. Entry t assigns a value to a variable and uses **key** as the first expression to the conditional **if**. When the key is entered, it is compared to the second expression, **answer**. The entered key is not assigned to a variable. In entry u the entered key is stored in a variable and the two variables are compared. Since entry u captures the key in a variable, an incorrect answer (as well as a correct answer) can be typed in the document.

```
entry t
{
        answer = 7
        call prompt("Enter Answer")
        if(key == answer)
        {
            "Correct, the answer is " call feed(answer)
        }
        else
        {
            "Incorrect, the answer is " call feed(answer)
        }
        "\027"
}
```

```
entry u
{
     realanswer = 7
     call prompt("Enter Answer: ")
     answer = key
     if(answer == realanswer)
     {
         "Correct, the answer is " call feed(answer)
     }
     else
     {
         "Incorrect, the answer is " call feed(realanswer) ", not "
         call feed(answer)
     }
     "\027"
}
```

The keys function: The **keys** function accepts an unlimited number of character keys that are assigned to a variable or a function. Program execution continues when you press EXECUTE or RETURN. Only character keys (including Control Y characters) are accepted by **keys**. A beep sounds if a formatting or editing key, such as Copy, Insert, Delete, or Search is entered.

The string from **keys** can be checked by a conditional function, but it is more difficult than checking **key** because a greater amount of data can be input. The characters you type in response to **keys** are typed on the editing screen. Like **posmsg** messages, they overlay existing text and are not cleared until you press CTRL/w or CANCEL and RETURN. This overlay feature of **keys** can be confusing because it obscures existing text. If you include the CTRL w statement ("\027") immediately after the **keys** statement in your program you can clear **keys** input without terminating the program or disrupting your text.

Remember, input to **key** or **keys** does not become part of the document. You must use **feed** or **finsert** to write **key** or **keys** directly or to write their assigned variables to the document. When you want to write directly to the document, it is simpler to use **keyin** or **keysin**, which perform this function naturally.

The keyin and keysin Functions

These functions are very similar to **key** and **keys** except that their input cannot be stored in a variable. Instead, it is written directly to the document. Unless they are used as expressions, both functions are preceded by the **call** statement.

The **keyin** function accepts any keyboard key and writes it to the document. Program execution continues immediately after you type the key. You cannot correct a mistake (even though you can see it on the screen) until the program concludes.

The **keysin** function accepts an unlimited numbers of keys. It will accept any key on the keyboard except EXECUTE. Pressing EXECUTE terminates **keysin** entry, and program execution continues.

Since **keysin** accepts multiple keys, you can use the Backspace or Cursor Control keys to correct typing mistakes. You cannot use the standard editing functions, such as Insert or Delete, because they require EXECUTE to conclude their function, which also concludes **keysin** entry. Pressing CANCEL during **keysin** entry terminates the glossary program.

Because both functions place their input directly in the document, existing text will be overwritten unless your cursor is in a blank area of the screen or you use the keywords insert and execute. Entry x uses insert and execute to insert **keysin** input in the document.

Because of the text overwriting characteristics inherent in **keyin** and **keysin**, placement of the cursor is an important consideration when you use interactive functions with display functions.

```
entry v
{
        insert
              call keysin
        execute
}
```

Using Interactive Functions with Display Functions

When you place a **key** or **keys** statement in your program after a display function statement like **prompt**, **status**, **error**, or **posmsg**, the cursor jumps to the position immediately following the function message. After you enter the requested data, the cursor jumps back to its original location.

This is not a particularly important consideration when you are using the **key** function, since its single key is not displayed on the screen. However, when you use **keys**, the input to **keys** appears to overwrite whatever text exists at the message location. As you have discovered, these characters can be cleared by using a CTRL w statement (octal 027) in your program.

When **keyin** and **keysin** functions are used with display functions, the cursor remains in its position and entered data becomes part of the document at that location.

Try writing some short test programs similar to the following examples if you are uncertain how interactive and display functions affect one another. When you find a combination that works best for your application, put it in your program.

```
entry x
{
        call posmsg(11,40,"enter key: ")
        x = key
        call feed(x)
}


entry y
{
        call prompt("enter keyin: ")
}


entry z
{
        call error("enter keys: ")
        y = keys
        call feed(y)
}
```

```
entry A
{
        call prompt("enter keysin: ")
        call keysin

}
```

Refer to other chapters in this book for additional examples that use interactive functions.

Entry D in the Mathematical Functions section in this chapter makes extensive use of interactive functions.

MATHEMATICAL FUNCTIONS

abs

max

min

num

number

pic

round

truncate

Mathematical Operators

+ - * / %

Mathematical Assignment Operators

+= -= *= /= %= =

Using Mathematical Functions

Mathematical functions can be used in a wide variety of programs. You need not restrict their use only to mathematical applications. For example, if you want to be sure the cursor is not on a number, use a combination of **num** and **char** to check the character.

The **num** function returns a value of 1 (true) if its expression is numeric and a value of 0 (false) if its expression is not numeric. The **char** function is a document reading function that returns the character at the cursor location. Entry B shows you how to be sure a character is not a number. The entry moves the cursor right to boldface characters until a number is encountered, then turns off the boldface mode.

```
entry B
{
        mode "b"
        while(num(char) == false)
        {
            right
        }
        mode "b"
}
```

As you have seen from several programs examples in previous chapters, mathematical operators can be used for counting loops. Entry C can be used as a subroutine with other entries to clear the entire screen. The entry uses the mathematical assignment operator += to increment the variable **linenumber**.

```
entry C
{
        linenumber = 1
        while(linenumber < 25)
        {
            call clrpos(linenumber,1,80)
            linenumber += 1
        }
}
```

Creating a Calculator

Entry D is a program for creating a calculator that can perform simple mathematical calculations from your text document editing screen.

Although WORD ERA has a built-in Math function that is much more complete and faster than Entry D, the program is included in this book because it provides an excellent example of mathematical functions usage. (Entry D is in glossary document gloss3 on the Glossary Diskette.)

Entry D uses all of the mathematical operators, the interactive functions **key** and **keys**, conditional **if else** functions, and the display functions **posmsg** and **clrpos**.

Note that **posmsg** uses octal numbers and attribute codes to display its message in reverse video and sound a beep. Octal numbers and attribute codes are described in Appendix C.

An analysis of entry D follows the entry example. To use the entry recall it from your document editing screen and follow the instructions in the **posmsg** prompts.

```
entry D
{
        operand1 = 0
        operator = 0
        operand2 = 0
        result = 0

        call posmsg(25,1,"\034HD CALCULATOR IS ON \034Id\007")
        call posmsg(1,42,"\034HD Use Document Number? Type y or n : \034Id")
        call posmsg(2,50,"")
        answer = key
              if((answer == "y") | (answer == "Y"))
              {
                     call clrpos(1,42,38)
                     call posmsg
                     (1,42,"\034HD Place Cursor on Number; Press Execute \034Id")
                     call keysin
                     operand1 = number
                     call clrpos(1,42,39)
```

```
            call posmsg(2,42,"\034H‘ Absolute Value of Number? y or n:
            \034I‘")
            absolute = key
            if((absolute == "y") | (absolute == "Y"))
            {
                 call clrpos(1,42,38)
                 call clrpos(2,42,38)
                 operand1 = abs(operand1)
            }
            else if((absolute == "n") | (absolute == "N"))
            {
                 call clrpos(1,42,38)
                 call clrpos(2,42,38)
            }
       }
       else if((answer == "n") | (answer == "N"))
       {
            call clrpos(1,42,38)
            call clrpos(2,42,38)
            call posmsg(1,42,"\034HD Enter Number & Press Execute:
            \034Id\007")
            call posmsg(2,50,"")
            operand1 = keys
            call clrpos(1,42,38)
            call clrpos(2,50,30)
       }
   call posmsg
   (1,42,"\034HD Enter Operator(+,-,*,/,%) & Execute: \034Id\007")
   call posmsg(2,50,"")
   operator = keys
   call clrpos(1,42,38)
   call clrpos(2,50,30)

   call posmsg(1,42,"\034HD Use Document Number? Type y or n : \034Id")
   call posmsg(2,50,"")
   answer = key
       if((answer == "y") | (answer == "Y"))
       {
            call clrpos(1,42,38)
            call posmsg
            (1,42,"\034HD Place Cursor on Number; Press Execute \034Id")
            call keysin
            operand2 = number
            call clrpos(1,42,39)
```

```
          call posmsg(2,42,"\034H‘ Absolute Value of Number? y or n:
          \034I‘")
          absolute = key
          if((absolute == "y") | (absolute == "Y"))
          {
               call clrpos(1,42,38)
               call clrpos(2,42,38)
               operand2 = abs(operand2)
          }
          else if((absolute == "n") | (absolute == "N"))
          {
               call clrpos(1,42,38)
               call clrpos(2,42,38)
          }
     }
     else if((answer == "n") | (answer == "N"))
     {
          call clrpos(1,42,38)
          call clrpos(2,42,38)
          call posmsg(1,42,"\034HD Enter Number & Press Execute:
          \034Id\007")
          call posmsg(2,50,"")
          operand2 = keys
          call clrpos(1,42,38)
          call clrpos(2,50,30)
     }

     if(operator == "+")
     {
          result = operand1 + operand2
     }
     else if(operator == "-")
     {
          result = operand1 - operand2
     }
     else if(operator == "*")
     {
          result = operand1 * operand2
     }
     else if(operator == "/")
     {
          result = operand1 / operand2
     }
```

```
          else if(operator == "%")
          {
               result = operand1 % operand2
          }
     call posmsg(1,42,"\034HD  Calculation result is: \034Id\007")
     call posmsg(2,50,round(result,2))
     call posmsg(25,42,"\034HD Press EXECUTE to continue \034Id\007")
     call keyin
     "\027"
}
```

Analysis of Entry D Functions

The round and truncate functions: The **round** or **truncate** function reduces result numbers to a manageable size. In the syntax example below, the **round** function rounds **result** to two decimal places. The example uses the **result posmsg** statement from entry D. If you did not use **round** or **truncate** on a calculation like 222 / 13, the result would be 17.076923076923076923O7, which is a very long number.

```
          call posmsg(2,50,round(result,2))
```

The **round** function adds 1 if the fractional part beyond the specified decimal place is 5 or greater. If it is less than 5, nothing is added. The result of the calculation 222 / 13 using **round** is 17.08.

Alternatively, you can use **truncate**. In the syntax example below, the **truncate** function truncates **result** at two decimal places.

```
          call posmsg(2,50,truncate(result,2))
```

The **truncate** function does not mathematically round **result**, it just chops off the end. Using **truncate**, the result of 222 / 13 is 17.07 (rather than 17.08, the result achieved from rounding).

Be sure to use **round** if you want a truly rounded **result** number.

Calculating with a Number in the Document: The **number** function reads a number from the document as part of your calculation. Entry D uses a yes/no branch and the **number** function to allow you to use a number typed in the text as an operand in the calculation.

The **number** function returns the number at the cursor location. It recognizes numbers only; a value of 0 is returned if the number is preceded by a required space or contains any alphabetical characters. The number may contain a leading dollar sign, commas, a decimal point, and/or leading or trailing plus or minus signs. (Refer to the functional definition of **number** in Chapter 10 for a detailed description.)

Using the abs function: The **abs** function takes the absolute value of a number. Essentially, it strips away any signs (such as + or -) and treats the number as an unsigned number. For example, suppose the office administrator for the "Leche Dairy," has prepared the following letter.

> Dear Customer:
>
> Your bill for home dairy delivery in February was 47.32,
> minus -4.27 crediting overpayment for January, which amounts to $_______.
>
> Thank you for your patronage of Leche Dairy.
>
> Vaca Bovine, Office Administrator

Ms. Bovine wants to use the late charge number in the letter as an operand to the glossary calculator (entry s). However, the calculator doesn't deal well with the minus sign in front of the number (-4.27). By using the **abs** function she can use the signed number in the document.

Adding Additional Functionality to the Calculator: The calculator is still a basic program. As such, it is a good program to experiment with. See what additional features you can add to the calculator. Here are some suggestions: Add a feature that allows you to insert the calculation result in your document. Add a loop to allow another calculation without exiting the calculator.

Add features that save the numbers entered on the first calculation and allow you to use them in the second calculation.

Use the **pic** function to add commas to **result** numbers above 999. See the description of **pic** in the alphabetical function list in Chapter 10.

Using the max and min functions

When you write glossary programs for others to use, you have no way of knowing the exact document conditions during program execution. This means you have to build more document reading or program analysis functions into your programs.

The **max** and **min** functions provide an example of this concept. Assume that you want to use the highest number value of three variables in your program. Two of the variables are declared and initialized in the program, but the third variable must be read from the document. You don't know what the document number variable is because it varies each time the glossary is used.

You can use the **max** function to provide the variable evaluation. Entry E gives you an example that uses **max** to display the highest variable in the **status** area. You could then write another programming statement that allows the operator to make a decision based on the highest variable number displayed by **status**. Entry E could also be written using the **min** function to provide the value of the lowest variable.

```
entry E
    {
        thisis = 25
        thatis = 44
        call posmsg
            (1,42,"\034HD Place cursor on number & press Execute \034Id")
        call keysin
        call clrpos(1,42,38)
        docnumis = number
        highest = max(thisis, thatis, docnumis)
        call status(highest)
        call posmsg(1,42,"\034HD Press Execute to Continue \034Id\007")
        call keyin
        "\027"
    }
```

OPERATING SYSTEM ACCESS FUNCTIONS

command "!"

command "|"

date

time

unixfun

unixpipe

Using UNIX Operating System Access Functions

These functions allow you to include UNIX operating system commands in your glossary programs. If you have never used UNIX at the shell command level, you should read a commercial book on the UNIX operating system before using **unixfun** and **unixpipe**. You don't need to know UNIX to use the **date** and **time** functions.

The date and time Functions

Use the **date** function to return the system date and time. Use the **time** function to return only the time. Entry J in the section "String Functions" in this chapter shows you how to use the **substr** and **if else** functions to modify the value returned by **date** so that it can be used in a business letter.

Entry F is a glossary entry you can use to periodically display the date and time from your text document editing screen. Remember, both the date and time function take their values from the system date and time, so if your system date and time have been set incorrectly your entry will also be incorrect.

```
entry F
{
      call posmsg(2,38,"\034HD IT IS NOW: \034Id\007")
      call posmsg(2,51,date)
      call posmsg(25,37,"\034HD Press EXECUTE to clear time \034Id")  call
      keyin
      "\027"
}
```

Note that the **date** function is used as an expression to the **posmsg** function. Since the **date** function returns a string value (the date) it is used in place of the third expression in the **posmsg** argument, which is normally a quoted string. You could use this method for any function which returns a displayable value.

Entry F uses octal number and attribute code combinations to place the **posmsg** message in reverse video. Octal numbers and attribute codes are described in Appendix C. The octal number for CTRL/w is used to clear the **posmsg** messages from the editing screen when you press EXECUTE.

Entry G below, uses the **time** function to display just the time. Note that entry G uses **unixfun** to execute the UNIX command "**sleep** 7" to display the time for seven seconds.

```
entry G
{
        call posmsg(2,42,"\034HD THE TIME IS: \034Id\077")
        call clrpos(2,56,1)
        call posmsg(2,58,time)
        call unixfun("sleep 7")
        call clrpos(2,40,35)
}
```

The unixfun and unixpipe Functions

Both of these functions give you access to a wide range of UNIX commands you can invoke from the document editing screen.

When you call **unixfun** in your program, it performs the following actions.

1. Escapes from the document to the standard UNIX Bourne shell **sh** (this action is transparent, you do not literally see this occurring).

2. Executes the UNIX command in its argument, for example, the statement **call unixfun("pwd")** displays the current working directory pathname at the cursor location in your text document.

3. Displays the standard output from the UNIX command at the cursor location in your text document. Characters displayed by **unixfun** are not written to the document and can be cleared from the screen by pressing CTRL/w or including the octal "\027" in your glossary program.

Because the UNIX command must be an expression in the argument to **unixfun**, you cannot use the interactive functions **keys** or **keysin** to enter the argument to **unixfun**. You can, however, use the interactive functions **key** or **keys** by assigning the input to a variable and using the statement

```
xinput = "keys"
call unixfun(xinput)
```

A programming alternative to **unixfun** is the keyword statement command "!". Command "!" is described in this section.

When you call **unixpipe** in your program, it performs the following actions.

1. The **unixpipe** function escapes to the standard UNIX Bourne shell **sh**

2. Executes its commands and writes the standard output of the UNIX command in your text document.

3. The standard output from a **unixpipe** command is written directly to the document and becomes a part of it.

The keyword statement command "|" is an alternative to **unixpipe**. Command "|" is described below.

You must assign the value returned by **unixpipe** to a variable as shown in entry H. There are two expressions in the argument to **unixpipe**, expression1 is the UNIX command in quotes. Expression2 is the data expected by the command. Since very few UNIX commands expect data, the second expression may be a null as shown in entry H.

```
entry H
{
        x = "who"
        b = ""
        y = unixpipe(x,b)
        call finsert(y)
}
```

Refer to the functional description of **unixpipe** in Chapter 10 for two glossary entry examples that use **unixpipe**.

Using command "!" and command "|"

Both command "! and command "|" are not functions, they are keywords. They are included here because their action is equivalent to **unixfun** and **unixpipe**. Entry I is an example that uses both command "!" and command "|".

```
entry I
{
        call display(false)
        command "!"
        "sort documentb -o documentb"
        return
        execute
        command "|"
        execute
        "cat documentb"
        return
        call display(true)
}
```

Entry t uses command "!", command "|", and the UNIX command **sort** to sort a WORD ERA document and write the sorted result to a text document you are currently editing. (To sort text on the document editing screen use the keyword statement command merge or command MERGE.)

To try this entry, prepare the document to be sorted (documentb in the example) by typing a simple list of words, one word per line. Change the glossary entry so that the name of this document replaces "documentb."

Edit the document where you want the sorted list and recall the entry. As the comments in entry I indicate, you could replace the document name with the **keysin** function and interactively enter the name while the entry is executing.

If you are totally unfamiliar with command "!" and command "|", try using both commands a few times from your text document before you use them in a glossary entry.

To use command "!" from your document editing screen, press COMMAND then type ! You are now in the UNIX command shell. Type a UNIX command and press EXECUTE or RETURN. The output of the command is displayed on the screen, and the prompt "Press execute to continue" appears. Press EXECUTE. You are returned to the document editing screen.

To use command "|" from your document editing screen, press COMMAND, then type |. The prompt "Replace what?" appears. Highlight the document text you want to replace and press EXECUTE. You are now in the UNIX command shell. Type a UNIX command and press EXECUTE or RETURN. The output of the command replaces the text you highlighted in your document.

Good UNIX commands to practice with are **who**, which gives you a listing of all users currently logged onto the system, or **ls**, which gives you a listing of your current directory.

When the output of **unixpipe** is returned to your document, it is not formatted like it is in the UNIX shell. You may want to write a glossary program to reformat it in your document.

STRING FUNCTIONS

cat

index

len

max

min

occur

seg

sub

substr

Using String Functions

The previous entries in this book have shown you how to assign alphabetic or numeric strings to variables, how to type the string value of a variable in your document, and how to compare one string value against another.

String functions provide even more flexibility in your programming use of string values. You can use string functions in many ways. Some suggestions for their use might be to

Extract a portion of the string and assign it to another variable.

Substitute a segment of a string with a different segment.

Find out if a specific sequence of characters is included in the string.

Combine strings from two different variables to form one string.

Find out how many characters a string contains.

Compare multiple strings to determine their highest or lowest ASCII collating value.

The following entries all use string functions.

Using substr to Reformat the date Function

Entry J uses the **substr** and **cat** functions to format the value returned by the **date** function so that it can be used in a business letter. The **date** function returns the system date and time in this format:

```
Fri Jul 14 19:25:14 1987
```

For most business letters, you probably want the date to read

July 14, 1987

In entry J, the output of **date** is assigned to **today**. The **substr** function extracts the month from **today** by specifying positions 5 through 7. This value is stored in **month**. The full spelling of the current month plus a space is then reassigned to the variable **month**.

The day and year are extracted from today by **substr** and assigned respectively to **day** and **thisyear**. The **cat** function is used to concatenate a comma and the year (with the leading space) into the variable **year**. The variables **month** and **day** are assigned to **thisday** by **cat**, **thisday** and **year** are concatenated and assigned to **currentdate**, which is typed in the document.

Entry J can be called as a subroutine by other entries that require the date in a standard business format.

```
entry J
{
     today = date

     month = substr(today,5,7)
         if(month == "Jan") {month = "January "}
         if(month == "Feb") {month = "February "}
         if(month == "Mar") {month = "March "}
         if(month == "Apr") {month = "April "}
         if(month == "May") {month = "May "}
         if(month == "Jun") {month = "June "}
         if(month == "Jul") {month = "July "}
         if(month == "Aug") {month = "August "}
         if(month == "Sep") {month = "September "}
         if(month == "Oct") {month = "October "}
         if(month == "Nov") {month = "November "}
         if(month == "Dec") {month = "December "}

     day = substr(today,9,10)
     thisyear = substr(today,20)
     year = cat(",",thisyear)
     thisday = cat(month,day)
     currentdate = cat(thisday,year)
     call feed(currentdate) return
}
```

When you use a string function like **substr**, you must know the position of the string segments to extract them. To determine the positions before you write your program, write a short program to check the string. Be careful, however, which date function you use in your program. Entries K and L both return the date, however the value returned by entry K (which uses the glossary **date** function) and entry L (which uses command "|" to return the system date is quite different, as shown below.

```
entry K
{
        today = date
        call feed(today)
}
```

```
entry L
{
        command "|"
        execute
        "date"
        execute
}
```

This is the date returned by entry L, which uses command "|" to bring the system date directly from UNIX.

```
              Sat Jul 14 08:17:57 PDT 1984
```

This is the system date returned by the **date** function in entry K.

```
              Sat Jul 14 08:19:59 1984
```

The direct system date includes the timezone "PDT." If you used this date to count positions for substr, your position count after the time segment would be off by four characters.

Using the len function

Use the **len** function to determine how many characters are in a string. Entry M is used as a subroutine for an interactive mailing list program. If the operator typed the full name for a state rather than the two-character abbreviation, an error message appears, and the operator is asked to re-enter the state.

```
entry M
{
     call prompt("Enter state: ")
     state = keys
     call clrpos(1,50,30)
         if(len(state) > 2)
         {
              call error("State too long, re-enter: ")
              state = keys
         }
     call feed(state)
     "\027"
}
```

The most frequent uses for the **index**, **occur**, **seg**, and **sub** functions are in Records Processing control glossary entries. See the Records Processing User's Guide for program examples.

SUMMARY

This chapter completes Part 3 Glossary Functions Reference and Usage Guides. Chapters 12 and 13 provide administrative and operating system information for the word processing supervisor and UNIX user. Chapter 14 provides usage instructions for the Glossary Diskette accompanying this manual.

CHAPTER 12

ADMINISTERING GLOSSARY PROGRAMS

Administering programs is just as important as writing them. Whether you are writing programs for yourself or for others, you will be responsible for testing, review, maintaining, and updating your programs. You will derive the greatest benefit from your glossary programs by reviewing and updating them frequently. This chapter offers you practical advice on glossary program administration. Such topics as program planning, troubleshooting, and program obsolescence are covered. You are shown how to set up and maintain a glossary program log book. Administering programs in a multiuser environment and program security are also discussed.

When you become proficient in writing glossary programs, two things usually occur: You rapidly acquire a large collection of programs, and you write programs for other people to use.

When you reach this point, the following considerations become vital:

> Program planning: Why do we need it? How long will it take to write it? Is the time spent writing it worth it? Is the planned application suitable for a glossary program? Would it be better to use a spreadsheet or Records Processing?
>
> Program applications: What does the program do? Who should use it? How do you use it?
>
> Program access: Where are the glossary documents on the system? Which program is in which glossary document?
>
> Program runtime: How long does it take to run? How do I schedule runtime? How much system space does it consume?

Program backup, storage, and retrieval: Where are the programs stored? How often do programs need to be backed up? Where is the hard copy kept?

Program obsolescence: Is it still good for anything? Can it be updated or should a new program be written?

Program duplication: Why are there different versions of the same program on the system? Which one is correct?

Programs in a multiuser environment: How did my glossary document get renamed? Who's editing my glossary when I want to edit it? Who made all those weird changes to my program?

Program debugging (troubleshooting): What syntax error? This isn't a bug, it's a monster! Hooray, I fixed the bug, let's go to lunch.

This chapter discusses each of these considerations and gives you some technical tips.

ADMINISTERING PROGRAMS

The following considerations are detailed from the bulleted items on the previous page. Each item illustrates a different facet of program administration.

Program Planning

You should consider writing a glossary program for the following reasons:

You or someone you are supervising is continually re-keying or copying the same text. It could be an address, legal paragraphs, form letters, technical terms, or standard forms.

You are typing tables in your document, then hand-calculating them and inserting the results. You can decide to use the Math feature, use a glossary program, or a combination of both.

You are required to fill out complicated standard forms that are pre-printed on tractor-fed computer paper or snap-apart carbon copies. You can use the Forms Processing feature, a glossary program, or a combination of both.

You are working with mailing lists, parts lists, or inventory lists where some items remain standard and other items change periodically. Glossary programs are an integral part of the Records Processing feature.

Of course, there are many other reasons for writing a program. Some of them will be particular to your own working environment. The reasons listed are the most universal programming applications. As a criterion, if you are performing the same task on a periodic basis, you should consider it a program candidate.

When you are planning a glossary program to perform production tasks, you should analyze the amount of time it will take to write the program against the amount of time saved by the program. Obviously, you don't want to spend three hours writing a program that will save five minutes one time only. But, if you spend three hours writing a program that will save five minutes a day, your time is well spent.

Sometimes you can get too ambitious with a glossary program. If you are planning to use glossary programs for extremely large financial spreadsheets or massive mailing lists, you should consider using another application, such as a spreadsheet, database program, or Records Processing. These applications are specifically designed to serve your spreadsheet or data base needs and run much faster than a glossary program for these uses.

Program Applications

You need to provide three kinds of information for every program you write:

What the program does (its application)

How the program flow is executed (why it works the way it does)

How the program is used (which keys you press or what you type if it is interactive)

Set up a "Glossary Program Information" notebook. Provide a separate tab for each glossary document and include the following information:

> Index of the glossary document that shows each entry label with a one-line comment about the entry, (see entry b for a program that does this for you)
>
> Printed hard copy of every entry
>
> Instruction sheet for entries that are long enough or complicated enough to require instruction

The amount of information required about a program increases with the complexity of the program. Comment lines in the entry are probably sufficient for simple programs.

Longer programs or programs designed for temporary employees to use require hard copy documentation. See entry a in the section "Program Examples" in this chapter for an example of a good, concise instruction sheet for a glossary program.

The effort you make in setting up and maintaining adequate glossary records is rewarded by the amount of time you save in keeping track of your programs and their uses.

Program Access

In addition to knowing how your programs work and how to use them, you need to know where they are on the system. If you are working on a single-user system, you probably don't have any difficulty remembering which glossary documents are in which library. On a multiuser system with four or more users, this can sometimes be a problem.

One solution is to create a library specifically for glossaries. Call this library "glos" (or something similar and short). Keep all multiuser glossaries in this library so that they are in one place and are accessible by all users.

Do not create this library through WORD ERA. Instead, use the newuser login to make a new account. This way, you place the library under the user login directory and shorten the pathname you must use to attach the glossary from your document. To attach a glossary from another user, you must give the full pathname, which includes the /u/ directory, the user's login directory, the glossary library, and the glossary name. For example:

```
/u/barbara/Glossaries/usegl
```

To attach the glossary from the /u/ directory the pathname is /u/glos/glossaryname, which is shorter. To edit a glossary you can either log in as "glos" or use the full pathname, /u/glos/glossaryname.

You can expedite this further by writing a program such as entry 1 below to do everything for you except enter the glossary document name.

```
entry 1
{
     command
     glossary
     "/u/glos/"
}
```

If you are a UNIX user, you can use the **ln** command to link glossaries to all user home directories. Chapter 13 shows you how to do this.

When you are working in your text document and need to use an entry but can't remember the label, use entry i in Chapter 14 to give you an index of your glossary document. You can also include an index display as an entry in your glossary document.

Program Runtime

Runtime is a real consideration in glossary programming. All glossary programs execute in the "foreground," and your terminal is unavailable for use with other applications while the program is running. With a multiuser system, you can use another terminal to run the program. However, in a busy production environment, this is not always a feasible alternative. As you have learned, turning the display refresh off by using the **call display(false)** statement helps speed up glossary runtime.

You could create a schedule to run lengthy programs during lunch hours, in the evenings, or overnight. The "glossary in progress" program shown in Chapter 11 is helpful for this purpose. It notifies people that a program is running and that the terminal should not be used until the program is finished.

Again, if your programs take excessively long to run, glossary may not be a suitable solution for your application.

Program Backup, Storage, and Retrieval

Like any WORD ERA document of value, programs should be backed up to an archive Diskette every time a change is made. The Diskettes should be stored in a safe place, be clearly labeled, and be available for quick retrieval.

You can keep a record of program archive Diskettes by printing the program archive Diskette index and placing it in the front of your "Glossary Information Notebook."

Always keep a hard copy of your program filed in your notebook, your filing cabinet, your desk drawer, or your pocket, but do keep a hard copy. If someone accidentally deletes it from the system and you lose the archive Diskette at the same time, its going to be difficult to rewrite all that programming code.

Program Obsolescence

Programs become obsolete when your office procedures change, when you think of a better program, or when you update an old program. Periodically review the programs you have on the system or on archive Diskettes and delete obsolete programs. They take up space and can cause confusion if someone tries to use them.

When you update a program, get rid of the previous version. You can keep a hard copy file of all your old programs if you like or have a special archive Diskette just for obsolete programs.

Try to write your programs with an eye toward future modification. Comment lines and documentation help a lot when you are updating an old program. It's too easy to forget what your logic was if the program flow is not clearly described. Working on someone else's program is even more difficult.

If possible, have periodic meetings of all the glossary writers in your department to review programs. You can also discuss and establish standard commenting and documenting procedures for programs. If your group writes a large number of programs, these meetings can help spread information about glossary usage and can provide a vehicle for sharing new programs.

Program Duplication

Several versions of the same program can cause a lot of confusion. Be sure to note revision numbers on your programs. File or delete old versions. Notify all your program users when you replace an old program with a new one.

Programs in a Multiuser Environment

As you learned in Chapter 4, several users can attach and use the same glossary document at the same time. You cannot edit, archive, copy or move a glossary document while another user has it attached or is editing it. You can, however, attach, use, and edit a glossary document (although no other user can attach, or otherwise access the glossary document while you are editing it).

You must be especially careful with glossaries on a multiuser system. A user who is logged on the system under a different account than yours can delete, rename, or move the glossary to an archive Diskette without your knowledge. Be sure to check with all users on your system before you perform any of these functions on a glossary document.

Use the comments line on the glossary document summary to establish ownership of your glossary programs. A comment like "Please see System Administrator before making any program changes" informs someone accessing the document that you do not want changes made without your permission.

Notations in your glossary hard copy notebook will help clarify questions regarding who has responsibility for specific programs.

If you are concerned about security or do not want to permit other users to access your glossary documents you can:

Password your glossary documents (see Chapter 4 for information about password protecting glossary documents).

Change file permissions on your glossary documents (refer to the WORD ERA Reference Guide and FOR:PRO User's Guide for information about file permissions).

Program Debugging (Troubleshooting)

A bug is programming parlance for an error in a program. Debugging is the process of finding and correcting bugs. Most bugs can be classified as errors in the following categories:

Syntax
Execution
Logic

SYNTAX BUGS: Syntax bugs occur when you violate a rule in the Glossary language. They can be errors in statement construction, incorrect use of a function, a misspelled keyword or function, too many or too few expressions in a function argument, missing identifiers, or incorrectly named variables.

As you have already experienced, your glossary compiler helps you find and correct syntax bugs. Sometimes the messages are a bit cryptic, but they generally point you in the right direction. Refer to Appendix E for a descriptive list of all error messages associated with Glossary Functions.

EXECUTION BUGS: Execution bugs occur during program execution and usually cause the program to terminate abruptly (crash). They can result from trying to divide by zero, from using incorrect keyword sequence for a standard WORD ERA function, or from leaving out an input statement (like **key** or **keysin**). Even if the program verified as correct it may not execute correctly. The glossary compiler cannot detect execution bugs unless they are related to syntax bugs.

LOGIC BUGS: Logic bugs are sometimes the most difficult errors to track down because they are particularly prevalent when you are using loops and branches. Some of the following ideas may help you to detect these errors: Try temporarily removing a loop to test the statement execution for one pass; check all your variable names and be sure they are initialized to 0 or an initial value; if you are using the same variable more than once in the program, make sure it's spelled correctly each time; be sure you don't inadvertently duplicate variables; check your subroutine calls are you calling the correct entry?

A very subtle logic bug can occur when you are programming mathematical calculations. The program can appear to be running properly, but the calculations are wrong. You should always check your program results against a set of known results. If you write a program to add a column of figures, also add the column on a hand calculator to be sure the program is adding correctly.

POINTS TO REMEMBER: Here are some cardinal rules to remember about writing and debugging programs:

Don't get frustrated if it doesn't work right the first time.

The bug is probably something simple.

Debug your program systematically by developing a troubleshooting routine.

Rely on your sixth sense and intuitive judgement.

CHAPTER 13

GLOSSARY INFORMATION FOR UNIX USERS

WORD ERA File Structure

As a UNIX user, you can perform filing operations on WORD ERA documents from the UNIX shell. Before you do this, however, you should be acquainted with the files that comprise a WORD ERA document. WORD ERA utilizes these files to store and manipulate document information. The files and their extensions are listed in Table 13-1.

Table 8 WORD ERA Document Files

WORD ERA Document Files	Description
filename	The textual portion of a document
filename.dc	The history, statistics and page pointer information for the document
filename.fr	The formats, header page, footer page and work page for this document
filename.gl	The compiled and executable binary form of a glossary document
filename.ex	The compiled and executable form of an exception dictionary

When you perform a UNIX command such as **rm**, **cp**, or **mv**, you must follow the WORD ERA document name by the metacharacter (*) to ensure that all files are included. For example, the following command removes the WORD ERA file report from the directory (library).

```
rm report*
```

The .gl File

As indicated above, the .gl file is only present if the document is a compiled glossary. The .gl file is a binary file. When a glossary is created, either by example or hand-coding, the glossary compiler creates the .gl file.

The text file (base file without an extension) is the glossary source file. The object file is the .gl binary code for the glossary entries.

Because the .gl file is fully executable without the presence of the other associated files, there are some interesting things you can do with it. Some of them are:

> Use the **ln** command to link the .gl file across directories. Users can then conveniently use the glossary entries without giving the full pathname when they attach the glossary document.
>
> If you want to maintain write security on your glossary documents, change read and write permissions on the files. The .gl file is still executable, but users without the proper permissions cannot edit the glossary and change the entries.
>
> To save space on the system you can delete all but the .gl file. Since it is binary object code, it is executable without the other files. Be sure to copy the entire file to an archive disk before you delete the files. When you want to make additions to the glossary document or edit an entry, you can load the three files back on the system, edit the glossary document and recompile it.

SUMMARY

In Chapter 11 you learned how to use the UNIX access functions **unixfun** and **unixpipe**, as a UNIX user you will find many interesting ways to use these functions in your glossary programs.

Refer to Appendices C and E for additional information of interest to the UNIX user.

CHAPTER 14

GLOSSARY ENTRY EXAMPLES

Your WORD ERA Installation Diskette number four (labeled 4 of 4), also contains the following four glossary documents:

gloss1, **gloss2a**, **gloss2b**, and **gloss3**

All the glossary entry examples shown in this book are in these glossary documents.

To retrieve the glossary documents from the installation diskette:

1. Be sure you are in the correct WORD ERA library. (You may want to create a special library for the glossary documents.)

2. Select Filing from the Main menu and press EXECUTE

3. Select Retrieve from archive from the Filing menu and press EXECUTE

4. Insert the WORD ERA Installation Diskette and press EXECUTE

5 When the diskette information is displayed, press EXECUTE to continue

6. Type **gloss1**, the name of the first glossary document

7. Press EXECUTE twice

8. Retrieve the remaining three glossary documents: **gloss2a, gloss2b, gloss3**

9. When you have retrieved all four glossary documents, select Remove archive diskette from the Filing menu and Press EXECUTE

10. Press EXECUTE, then remove the diskette.

You can attach any one of the glossary documents and use the entries in it as you learn glossary. Using the glossary documents saves you typing time if you want to try some of the longer entries in the book.

In addition to glossary entries, the example tables used with entries k and l in Chapter 8 are provided on Page N of glossary document **gloss2b**.

You can edit the glossary documents and modify any entry. As you read this book you will probably find several entries you can use for your own word processing activities.

Contents of Glossary Documents

The contents of each glossary document are shown in the following list. The chapter in this book where the entry appears is shown in a comment line after the entry label. Page numbers correspond to the entry page in the glossary document.

Entries in Glossary Document: gloss1

entry a	/*in Chapter 1*/	1
entry b	/*in Chapter 1*/	2
entry c	/*in Chapters 1 and 3*/	3
entry d	/*in Chapter 3*/	4
entry e	/*in Chapter 3*/	5
entry f	/*in Chapter 4*/	6
entry g	/*in Chapter 4*/	7
entry h	/*in Chapter 4*/	8
entry i	/*in Chapter 4*/	9
entry j	/*in Chapter 4*/	10
entry D	/*in Chapter 4 (modified entry d)*/	11
entry k	/*in Chapter 4*/	12

Entries in Glossary Document: gloss2a

entry A	/*in Chapter 5*/	1
entry a	/*in Chapter 5*/	2
entry b	/*in Chapter 5*/	3
entry c	/*in Chapter 5*/	4
entry C	/*in Chapter 5*/	5
entry d	/*in Chapter 6*/	6

entry e	/*in Chapter 6*/	7
entry f	/*in Chapter 6*/	8
entry g	/*in Chapter 6*/	9
entry h	/*in Chapter 6*/	10
entry i	/*in Chapter 6*/	11
entry j	/*in Chapter 6*/	12
entry k	/*in Chapter 6*/	13
entry l	/*in Chapter 6*/	14
entry m	/*in Chapter 6*/	15
entry n	/*in Chapter 6*/	16
entry o	/*in Chapter 6*/	17
entry p	/*in Chapter 6*/	18
entry r	/*in Chapter 6*/	19
entry s	/*in Chapter 6*/	20
entry t	/*in Chapter 6*/	21
entry u	/*in Chapter 6*/	22
entry v	/*in Chapter 6*/	23
entry w	/*in Chapter 6*/	24
entry x	/*in Chapter 6*/	25
entry y	/*in Chapter 6*/	26
entry z	/*in Chapter 6*/	27
entry B	/*in Chapter 6*/	28
entry D	/*in Chapter 6*/	29
entry E	/*in Chapter 6*/	30
entry F	/*in Chapter 6*/	31
entry G	/*in Chapter 6*/	32
entry H	/*in Chapter 6*/	33
entry I	/*in Chapter 6*/	34
entry J	/*in Chapter 6*/	35
entry K	/*in Chapter 6*/	36
entry L	/*in Chapter 6*/	37

Entries in Glossary Document: gloss2b

entry A	/*in Chapter 7*/	1
entry a	/*in Chapter 7*/	2
entry b	/*in Chapter 7*/	3
entry c	/*in Chapter 7*/	4
entry d	/*in Chapter 7*/	5
entry e	/*in Chapter 7*/	6
entry f	/*in Chapter 8*/	7

entry w	/*in Chapter 8*/	8
entry x	/*in Chapter 8*/	9
entry y	/*in Chapter 8*/	10
entry z	/*in Chapter 8*/	11
entry g	/*in Chapter 8*/	12
entry O	/*in Chapter 8*/	13
entry P	/*in Chapter 8*/	14
entry h	/*in Chapter 8*/	15
entry i	/*in Chapter 8*/	16
entry j	/*in Chapter 8*/	17
entry k	/*in Chapter 8*/	18
entry l	/*in Chapter 8*/	19
entry Z	/*in Chapter 8*/	20
entry K	/*in Chapter 8*/	21
entry L	/*in Chapter 8*/	22
entry m	/*in Chapter 9*/	23
entry n	/*in Chapter 9*/	24
entry o	/*in Chapter 9*/	25
entry p	/*in Chapter 9*/	26
entry q	/*in Chapter 9*/	27

Entries in Glossary Document: gloss3

entry a	/*in Chapter 10*/	1
entry b	/*in Chapter 10*/	2
entry c	/*in Chapter 11*/	3
entry d	/*in Chapter 11*/	4
entry e	/*in Chapter 11*/	5
entry f	/*in Chapter 11*/	6
entry g	/*in Chapter 11*/	7
entry h	/*in Chapter 11*/	8
entry i	/*in Chapter 11*/	9
entry j	/*in Chapter 11*/	10
entry k	/*in Chapter 11*/	11
entry l	/*in Chapter 11*/	12
entry m	/*in Chapter 11*/	13
entry n	/*in Chapter 11*/	14
entry o	/*in Chapter 11*/	15
entry p	/*in Chapter 11*/	16
entry r	/*in Chapter 11*/	17
entry s	/*in Chapter 11*/	18
entry t	/*in Chapter 11*/	19

entry u	/*in Chapter 11*/	20
entry v	/*in Chapter 11*/	21
entry x	/*in Chapter 11*/	22
entry y	/*in Chapter 11*/	23
entry z	/*in Chapter 11*/	24
entry A	/*in Chapter 11*/	25
entry B	/*in Chapter 11*/	26
entry C	/*in Chapter 11*/	27
entry D	/*in Chapter 11*/	28
entry E	/*in Chapter 11*/	29
entry F	/*in Chapter 11*/	30
entry G	/*in Chapter 11*/	31
entry H	/*in Chapter 11*/	32
entry I	/*in Chapter 11*/	33
entry J	/*in Chapter 11*/	34
entry K	/*in Chapter 11*/	35
entry L	/*in Chapter 11*/	36
entry M	/*in Chapter 11*/	37

APPENDIX A

RESERVED WORDS AND SYMBOLS

The words and symbols in this appendix are reserved for Glossary and Records Processing keywords, functions, and operators. The list of reserved words below cannot be used as variable names or identifier names in any glossary program.

Functions marked with an asterisk (*) can only be used in Records Processing Control-Glossary Documents.

Reserved Words

abs	DOWN	insert	occur	spacing
ascending*	down	jump	PAGE	status
backspace	EAST	key	page	STOP
beg_doc	east	keyin	page_no	stop
bot_page	else	keys	pic	sub
call	end_doc	keysin	position	subscript
CANCEL	entry	LEFT	posmsg	substr
cancel	error	left	PREVSCRN	SUPERSCRI
cat	EXECUTE	left_margin	prevscrn	T
CENTER	execute	len	prompt	superscript
center	exit	line	quote	TAB
char	false	loc	REPLACE	tab
clrpos	feed	max	replace	text
COMMAND	finsert	MERGE	RETURN	text_len
command	FORMAT	merge	return	thru*
COPY	format	min	RIGHT	time
copy	gl	MODE	right	top_page
cursor	globerr	mode	right_margin	true
date	glossary	MOVE	round	truncate
DECIMALTAB	GOTO	move	save_record*	unixfun
decimaltab	goto	NEXTSCRN	SEARCH	unixpipe
DECTAB	HELP	nextscrn	search	UP
dectab	help	NORTH	seg	up
DELETE	if	north	select_record*	WEST
delete	INDENT	NOTE	sort*	west
descending*	indent	note	SOUTH	while
display	index	num	south	word
do	INSERT	number	space	

Reserved Symbols

The characters in the following list are reserved for use by Glossary and can only be used for their designated purpose.

Function	Symbol
Mathematical	+ - * / %
Relational and Equality	< > <= >= == !=
Logical	! & |
Assignment	=
Mathematical Assignment	+= -= *= /= %=
Statement	( ) { } [ ]

APPENDIX B

COMPARISON OF GLOSSARY KEYWORDS AND FUNCTIONS

The glossary language is provided for two WORD ERA applications: Glossary and Records Processing Control Glossary Documents. Although the glossary writing procedure is the same for both, the compliment of glossary keywords and functions is different for each application, as shown by the following list:

> Glossary uses all keywords and functions except special Records Processing selection and sorting functions.
>
> Records Processing uses most Glossary functions, plus special record selecting and sorting functions.

The following list shows which keywords and functions can be used for each application.

Keywords and Functions Used in WORD ERA Applications

Name	Type	Glossary	Records Processing
abs	function	x	x
ascending	function		x
backspace	keyword	x	x
beg_doc	function	x	
bot_page	function	x	
call	function	x	x
cancel	keyword	x	x
cat	function	x	x

Name	Type	Glossary	Records Processing
center	keyword	x	x
char	function	x	
clrpos	function	x	
command	keyword	x	x
COPY	keyword	x	x
copy	keyword	x	x
cursor	function	x	
date	function	x	
decimaltab	keyword	x	x
dectab	keyword	x	
delete	keyword	x	
descending	function		x
display	function	x	
do	function	x	x
DOWN	keyword	x	
down	keyword	x	
EAST	keyword	x	
east	keyword	x	
else	function	x	x
end_doc	function	x	
entry	label	x	x
error	function	x	x
execute	keyword	x	
exit	function	x	x
false	function	x	x
feed	function	x	
finsert	function	x	
FORMAT	keyword	x	
format	keyword	x	
gl	keyword	x	
globerr	function	x	
glossary	keyword	x	
goto	keyword	x	
help	keyword	x	
if	function	x	x
indent	keyword	x	
index	function	x	x
insert	keyword	x	
jump	function	x	x

Name	Type	Glossary	Records Processing
key	function	x	x
keyin	function	x	
keys	function	x	x
keysin	function	x	
LEFT	keyword	x	
left	keyword	x	
left_margin	function	x	
len	function	x	x
line	function	x	
loc	function	x	
max	function	x	x
MERGE	keyword	x	
merge	keyword	x	
min	function	x	x
mode	keyword	x	
MOVE	keyword	x	
move	keyword	x	
nextscrn	keyword	x	
NORTH	keyword	x	
north	keyword	x	
note	keyword	x	
num	function	x	x
number	function	x	
occur	function	x	x
PAGE	keyword	x	
page	keyword	x	
page_no	function	x	
pic	function	x	
position	function	x	
posmsg	function	x	
prevscrn	keyword	x	
prompt	function	x	x
quote	keyword	x	
REPLACE	keyword	x	
replace	keyword	x	
return	keyword	x	
RIGHT	keyword	x	
right	keyword	x	

Name	Type	Glossary	Records Processing
right_margin	function	x	
round	function	x	x
save_record	function		x
SEARCH	keyword	x	
search	keyword	x	
seg	function	x	x
select_record	function		x
sort	function		x
SOUTH	keyword	x	
south	keyword	x	
space	keyword	x	
spacing	function	x	
status	function	x	x
stop	keyword	x	
sub	function	x	x
subscript	keyword	x	
substr	function	x	x
SUPERSCRIPT	keyword	x	
superscript	keyword	x	
tab	keyword	x	
text	function	x	
text_len	function	x	
thru	function		x
time	function	x	
top_page	function	x	
true	function	x	x
truncate	function	x	x
unixfun	function	x	
unixpipe	function	x	
UP	keyword	x	
up	keyword	x	
WEST	keyword	x	
west	keyword	x	
while	function	x	x
word	function	x	

APPENDIX C

CHARACTER CODES

This appendix describes character and attribute codes that can be used in your glossary programs. Table C-1, which is arranged in the ASCII (American Standard Code for Information Interchange) collating sequence, shows each WORD ERA character, its octal number, attribute codes, the attribute set by the code, and related document format control codes.

Each WORD ERA character is represented by a corresponding decimal, octal, and hexadecimal number. Only octal numbers are discussed in this appendix.

The octal numbers and attribute codes shown in **posmsg** strings apply to terminals manufactured by Fortune Systems' Corporation. Consult your terminal manufacturer for information about character sets, attributes, and octal equivalents that apply to your terminal.

If the **posmsg** strings shown in programs in this book do not work correctly on your terminal, remove the octal and attribute codes from the message string. For example, if the **posmsg** statement

```
call posmsg(7,26,"\034HD \034Id \034HBGLOSSARY IN PROGRESS\034Ib
\034HD \034Id")
```

does not display properly, change it to:

```
call posmsg(7,26,"GLOSSARY IN PROGRESS")
```

Table C-2 gives a descriptive listing of WORD ERA document format control codes.

ASCII COLLATING SEQUENCE

The ASCII collating sequence is a standard set of numeric codes used to represent characters. Entry a underscores a word, excluding punctuation and numbers, by comparing characters according to their ASCII number value.

```
entry c
{
        mode"_"
        while(((char >= "A") & (char <= "Z")) | ((char >= "a") & (char <= "z")))
        {
            right
        }
        mode "_"
}
```

OCTAL NUMBER CONVERSIONS

Use octal numbers in your glossary entries to include a control character in the program. For example, entry b includes two octal codes: \007 (CTRL G) sounds the keyboard bell, and \027 (CTRL W) refreshes the screen display. Table C-1 defines the action of control characters in a WORD ERA document. Not all control characters on the ASCII list are applicable to WORD ERA. Their actions are listed as undefined on Table C-1.

```
entry b
{
        call posmsg(1,43,"Enter Amount: \007")
        amount = keys
        "\027"
        call feed(amount)
}
```

ATTRIBUTE CODES

Glossary provides you with a set of keyword abbreviations for text emphasis modes, such as boldface, or underline. These abbreviations can be embedded in strings used by variables or functions. The only glossary functions that don't accept keyword mode abbreviations are display functions such as **posmsg** or **prompt**.

To emphasize display function messages you must use attribute codes. Attribute codes are letters or symbols that are assigned to octal numbers 001 through 077 on the ASCII set. Each attribute code sets a specific emphasis mode or combination of modes. Attribute codes and the modes they set are shown in Table C-1.

Entry c and entry d illustrate the difference between keyword mode abbreviations and attribute codes to highlight a message. In entry c, the keyword abbreviation \b for boldface is embedded in the **call feed** string. In entry d, which uses the **posmsg** function, the attribute code for boldface must be proceded by an octal code and an operator selector (the operator selectors H and I are described below).

```
entry c
{
     call feed("\bCustomer Name:\b  ")
     return(2)
     call keysin
}


entry d
{
     call posmsg(1,43,"\034H' Customer Name:\034I'")
     name = keys
     "\027"
}
```

The syntax to use an attribute code in a display function message is

```
     display function("\034 operator_selector_on attribute_code  message  \034
     operator_selector_off attribute_code")
```

The following examples for prompt and posmsg use the syntax shown above to display their messages in blink mode:

```
     prompt("\034HB PRESS EXECUTE TO CONTINUE \034IB")

     posmsg(25,43,"\034HB PRESS EXECUTE TO CONTINUE \034IB")
```

The rules for the attribute code syntax are:

Turn on the attribute code with the sequence

\034H attribute code

Backslash escapes the octal 034 and prevents it from being treated as text in the message.

Attribute codes are Fortune Systems' extended terminal commands, and must be preceded by a CTRL \ (octal 034).

Capital H is an operator selector to use when adding an attribute. It "turns on" the attribute code at the beginning of the message.

Turn off the attribute code with the sequence

\034I attribute code

Capital I is an "operator selector" to subtract an attribute. It "turns off" the attribute code at the end of the message.

The entire string, including octal numbers, operator selectors, attribute codes, and the message, must be enclosed in double quotation marks.

DESCRIPTION OF TABLE C-1

Table C-1 is ranked in ascending ASCII collating sequence. Column 1 shows the ASCII character; column 2 shows the corresponding octal number; columns 3 and 4 show the attribute code and the attribute set or attribute combination by the attribute code.

Column 5 shows the WORD ERA Document Format Control Code. These codes are covered in detail in this appendix under the section WORD ERA Document Format Control Codes, and also in Table C-2.

Column 6 shows the WORD ERA action performed by a control character. If the character is not applicable in WORD ERA, it is listed as undefined.

In Table C-1, a caret (^) before a character means the character is a control character. To type a control character, you simultaneously press CTRL and the character key.

TABLE C-1. Character Codes

ASCII Char	Octal Number	Attri-bute Code	Attri-bute Set by Code	Attribute Control Code ON	Attribute Control Code OFF	Document WORD ERA Action Performed
^@	000					Undefined
^A	001	A		\O\	Co\	Overstrike
^B	002	B	b	\Z\	Cz\	Flash(Blink)
^C	003	C	bo			Undefined
^D	004	D	r	\R\	Cr\	Reverse video
^E	005	E	ro			Undefined
^F	006	F	rb			Undefined
^G	007	G	rbo			Keyboard bell
^H	010	H	h			Backspace
^I	011	I	ho	011		Tab
^J	012	J	hb	012		Return (soft)
				\B\012		Return (hard)
^K	013	K	hbo	013		Ignore (Margin break)
^L	014	L	hr	014		Required page (hard)

TABLE C-1. Character Codes (continued)

ASCII Char	Octal Number	Attribute Code	Attribute Set by Code	Attribute Control Code ON	Attribute Control Code OFF	Document WORD ERA Action Performed
				\A\014		Optional page (soft)
^M	015	M	hro			Undefined
^N	016	N	hrd			Lock G1 char set*
^O	017	O	hrbo			Lock G0 char set*
^P	020	P	l	\U\	Cu\	Low underline
^Q	021	Q	lo			Undefined
^R	022	R	lb			Undefined
^S	023	S	lbo			Undefined
^T	024	T	lr			Undefined
^U	025	U	lro			Undefined
^V	026	V	lrb			Undefined
^W	027	W	lrbo			Restore
^X	030	X	d	\D\	Cd\	Double underline
^Y	031	Y	do			Invoke G2 character*
^Z	032	Z	db			Undefined

TABLE C-1. Character Codes (continued)

ASCII Char	Octal Number	Attri-bute Code	Attri-bute Set by Code	Attribute Control Code		Document WORD ERA Action Performed
				ON	OFF	
^[	033	[	dbo			Escape
^\	034	\	dr			Undefined
^]	035	]	dro			Invoke G3 character*
^^	036	^	drb			Undefined
^_	037	_	drbo			Reset character set*
(space)	040	‘	H	\X\	Cx\	Highlight (bold)
!	041	a	Ho			
"	042	b	Hb			
#	043	c	Hbo			
$	044	d	Hr			
%	045	e	Hro			
&	046	f	Hrb			
'	047	g	Hrbo			
(	050	h	Hh			
)	051	i	Hho			
*	052	j	Hhb			
+	053	k	Hhbo			

TABLE C-1. Character Codes (continued)

ASCII Char	Octal Number	Attri-bute Code	Attri-bute Set by Code	Attribute Control Code ON	OFF	Document WORD ERA Action Performed
,	054	l	Hhr			
-	055	m	Hhro			
.	056	n	Hhrb			
/	057	o	Hhrbo			
0	060	p	Hl			
1	061	q	Hlo			
2	062	r	Hlb			
3	063	s	Hlbo			
4	064	t	Hlr			
5	065	u	Hlro			
6	066	v	Hlrb			
7	067	w	Hlrbo			
8	070	x	Hd			
9	071	y	Hdo			
:	072	z	Hdb			
;	073	{	Hdbo			
<	074	;	Hdr			
=	075	}	Hdro			
>	076	~	Hdrb			
?	077	177	Hdrbo			
@	100					

TABLE C-1. Character Codes (continued)

ASCII Char	Octal Number	Attri-bute Code	Attri-bute Set by Code	Attribute Control Code ON	OFF	Document WORD ERA Action Performed
A	101					
B	102					
C	103					
D	104					
E	105					
F	106					
G	107					
H	110					
I	111					
J	112					
K	113					
L	114					
M	115					
N	116					
O	117					
P	120					
Q	121					
R	122					
S	123					
T	124					
U	125					

TABLE C-1. Character Codes (continued)

ASCII Char	Octal Number	Attri-bute Code	Attri-bute Set by Code	Attribute Control Code ON	OFF	Document WORD ERA Action Performed
V	126					
W	127					
X	130					
Y	131					
Z	132					
[	133					
\	134					
]	135					
^	136					
_	137					
`	140					
a	141					
b	142					
c	143					
d	144					
e	145					
f	146					
g	147					
h	150					
i	151					
j	152					

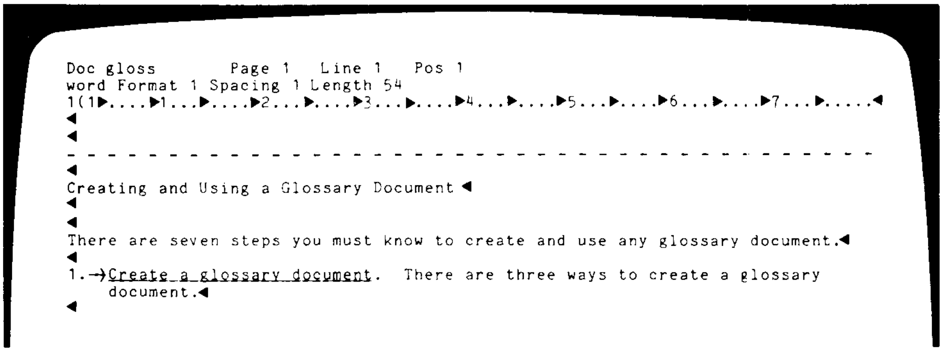
TABLE C-1. Character Codes (continued)

ASCII Char	Octal Number	Attribute Code	Attribute Set by Code	Attribute Control Code ON	OFF	Document WORD ERA Action Performed
k	153					
l	154					
m	155					
n	156					
o	157					
p	160					
q	161					
r	162					
s	163					
t	164					
u	165					
v	166					
w	167					
x	170					
y	171					
z	172					
{	173					
|	174					
}	175					
~	176					
cancel						
del	177					

WORD ERA DOCUMENT FORMAT CONTROL CODES

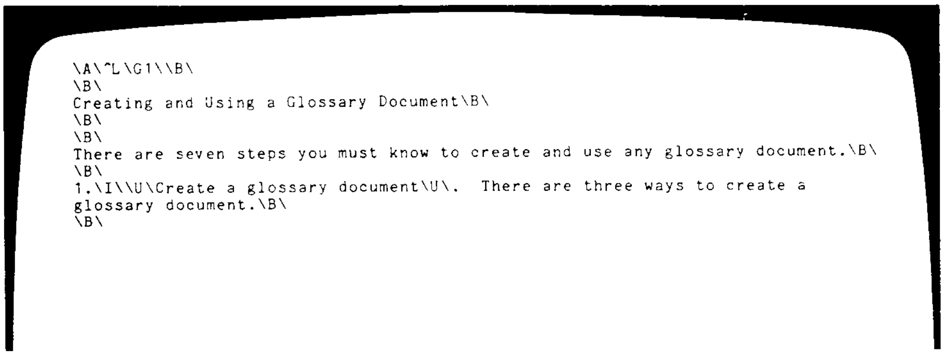
WORD ERA formatting characters are displayed on your document editing screen as graphic symbols, such as a right facing triangle for Tab, a diamond for Center, or an arrow for Indent. Each of these characters has a control code sequence embedded in the document. If you are familiar with UNIX commands, you can see these codes by using the **more** command to view the document from the UNIX shell. Figure C-1 shows you how a fragment of text looks on the document edit screen, Figure C-2 shows the same text viewed through UNIX.

Figure C-1 **How Text and Formatting Characters Look on the Document Editing Screen**



```
Doc gloss         Page 1   Line 1   Pos 1
word Format 1 Spacing 1 Length 54
1(1▶....▶1...▶....▶2...▶....▶3...▶....▶4...▶....▶5...▶....▶6...▶....▶7...▶.....◀
◀
◀
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
◀
Creating and Using a Glossary Document ◀
◀
◀
There are seven steps you must know to create and use any glossary document.◀
◀
1.→Create a glossary document.  There are three ways to create a glossary
   document.◀
◀
```

Figure C-2. **How WORD ERA Text and Control Codes Look from the UNIX Shell**



```
\A\^L\G1\\B\
\B\
Creating and Using a Glossary Document\B\
\B\
\B\
There are seven steps you must know to create and use any glossary document.\B\
\B\
1.\I\\U\Create a glossary document\U\.  There are three ways to create a
glossary document.\B\
\B\
```

As you can see by comparing the two figures, the combination of an an optional page break, the format line, and a return, requires the control sequence \A\^L\G1\\B\.

How to use Document Format Control Codes in Programs

Most of the document control codes shown in Table C-2 have keyword abbreviations that can be used in strings or can be assigned to variables. For example, the keyword abbreviations for a tab and a return are \t and \r. These keyword abbreviations, however, cannot be used in a variable when you want to evaluate a value returned by a document reading function like **char** or **text**. Entry e determines if the character at the cursor position is a return. The document control code sequence for the return symbol is assigned to **ret1**. The character at the cursor position is assigned to **ret2** by the **text** function. The two variables are compared and the string "this is a return" is inserted in the document if the cursor is under a return. If the cursor is not under a return, the string "it is NOT a return" is inserted in the document. (You could also use the **char** function to return the value of the character at the cursor position.)

```
entry e
{
ret1 = "\\B\\\012"
ret2 = text(loc,loc)
      if(ret1 =( ret2)
          {insert "this is a return" execute}
      else
          {insert "it is NOT a return" execute}
}
```

Use the **finsert** function to type the value of variables containing document control codes in your document. You can use **feed** and **finsert** interchangeably for octal or keyword abbreviations, but you must use **finsert** for document control code and octal combinations like this sequence for an optional page break \\A\\\014.

Table C-2 shows each WORD ERA document format control code, the action performed in the document, and a brief description of the code and how the code can be used in your programs.

TABLE C-2. WORD ERA Document Control Codes

Control Code	Action Performed	Description
\Gnnn\	Format number identification	A WORD ERA document may contain 100 different format lines. The control sequence \Gnnn\ sets a specific format line number. The "nnn" stands for a format number between 1 and 100 and is displayed in the document as a format line for setting tabs, columns, and margins.
^L	Required page break	A required page break that is not deleted by the pagination process; displayed in the document as a double dashed line. The syntax for assignment to a glossary program variable is: **variable** = "\014"
\A\^L	Optional page break	An optional page break that can be deleted by the pagination process; displayed in the document as a single-dashed line. The syntax for assignment to a glossary program variable is: **variable** = "\\A\\\014"

TABLE C-2. WORD ERA Document Control Codes (continued)

Control Code	Action Performed	Description
^I	Tab	Begin text line at the next tab position in the format line (if no tab exists in the format line, advance one space); displayed in the document as a right facing triangle. The syntax for assignment to a glossary program variable is: **variable** = "011"
^J	Return (soft)	Word wrap return used to change the line ending whenever editing causes the text to rewrap; not displayed in the document. The syntax for assignment to a glossary program variable is: **variable** = "012"
\B\^J	Return (hard)	Return that is not changed by word wrap; displayed in the document as a left facing triangle. The syntax for assignment to a glossary program variable is: **variable** = "\\B\\\012"
^K	Ignore	Used to break contiguous character strings at the right margin; not displayed in the document, and deleted when the line is reformatted.

TABLE C-2. WORD ERA Document Control Codes (continued)

Control Code	Action Performed	Description
\\\	Backslash	Since backslash is used as a control code delimiter, a backslash typed in the document is escaped by backslashes.
\U\	Underline on	Occurs before the first character in an underlined sequence and turns underline mode on.
\u\	Underline off	Occurs after the last character in an underlined sequence and turns underline mode off.
\D\	Double Underline on	Occurs before the first character in a double underlined sequence and turns double underline mode on.
\d\	Double Underline off	Occurs after the last character in a double underlined sequence and turns double underline mode off.
\X\	Bold on	Occurs before the first character in a bold sequence and turns bold mode on.
\x\	Bold off	Occurs after the last character in a bold sequence and turns bold mode off.

TABLE C-2. WORD ERA Document Control Codes (continued)

Control Code	Action Performed	Description
\O\	Overstrike on	Occurs before the first character in an overstrike sequence and turns overstrike mode on.
\o\	Overstrike off	Occurs after the last character in an overstrike sequence and turns overstrike mode off.
\R\	Reverse video on	Occurs before the first character in a reverse video sequence and turns reverse video mode on.
\r\	Reverse video off	Occurs after the last character in a reverse video sequence and turns reverse video mode off.
\Z\	Blink (Flash) on	Occurs before the first character in a blink sequence and turns blink mode on.
\z\	Blink (Flash) off	Occurs after the last character in a blink sequence and turns blink mode off.
\S\	Superscript	When the document is printed, a superscript symbol causes the printer to index up 1/4 line; displayed in the document as an up arrow.

TABLE C-2. WORD ERA Document Control Codes (continued)

Control Code	Action Performed	Description
\s\	Subscript	When the document is printed, a subscript symbol causes the printer to index down 1/4 line; displayed in the document as a down arrow.
\M\	Right-Flush Tab	Right justify text under a format line right-flush tab (r) until a Return, Tab, or another Right-flush Tab is encountered; displayed in the document as a left arrow.
\t\	Decimal Tab	Align numbers by decimal point (period) under tab stop in format line; displayed in the document as a short vertical line joined to an underbar.
\I\	Indent	Left justify wrapped text under a format line tab until a hard return is encountered; displayed in the document as a right arrow.
\i\	Indent, generated	Generated by the pagination process when a page break causes an indented paragraph to split between pages; deleted by the pagination process when the indented paragraph is rejoined by removing the page break; displayed in the document as a regular indent.

TABLE C-2. WORD ERA Document Control Codes (continued)

Control Code	Action Performed	Description
\c\	Center	Center a single line between the right and left margins; displayed in the document as a diamond.
\<\	Merge on	Left field name delimiter for Records Processing and as a marker for other applications; displayed in the document as a bright <.
\>\	Merge off	Right field name delimiter for Records Processing and as a marker for other applications; displayed in the document as a bright >.
\n\	Note	Document character strings enclosed in notes, or begun with a note and ended in a return, are suppressed during printing. Optionally, the characters may be printed by selecting "With notes" on the document print menu; displayed in the document as a double exclamation mark.
\Nnnn\	Footnote Reference	Footnote reference number, where nnn stands for the footnote number; displayed in text as a number in reverse video.

TABLE C-2. WORD ERA Document Control Codes (continued)

Control Code	Action Performed	Description
\-\	Hyphen, required	Placed in front of a word to prevent hyphenation during the hyphenation process; placed inside a word to mark the required break point for the hyphenation process; displayed in the document as an inverted T. (Also called a discretionary hyphen.)
\H\	Hyphen, generated	Generated by the hyphenation process; removed if subsequent document editing causes the line to rewrap; displayed in the document as a bright hyphen.
\ \	Space, required	Prevents separation of words by marginal word wrap; displayed in the document as a square U, printed as a space.
\F\	Column break, required (hard)	Required column break not deleted by the pagination process; displayed in the document as a double dotted line.

TABLE C-2. WORD ERA Document Control Codes (continued)

Control Code	Action Performed	Description
\C\	Column break, optional (soft)	Optional column break deleted by the pagination process; displayed in the document as a single dotted line.
\^Ycd\	Character from* the G2 set	Selects a character from the G2 character set. If the character is not accented the sequence \^Yc\ is present. If an accented character is chosen, then the sequence is \^Ycd\.

*Refer to the Fortune Systems' publication *Using Fortune Terminals*

APPENDIX D

KEYWORDS BY USAGE

Keywords can be grouped by the functions they perform. The following lists are a guide for using formatting, editing, and cursor movement keywords.

Cursor position and movement is an extremely important factor in glossary programming. If you are not thoroughly familiar with cursor movement during WORD ERA functions, you should study these lists carefully. They tell you about cursor action when a function is invoked.

When you use an entry containing keywords, the functions they perform are activated. Some keywords may be repeatedly activated by typing a number in parentheses after the keyword. It is easier to type return(3) than return return return.

Keywords marked with an asterisk (*) in the following lists accept numbers in parentheses.

Keywords in capital letters perform the same function that is accomplished by pressing that key and the shift key simultaneously.

Formatting Keywords

Keywords such as tab, indent, decimal tab, and return change the format of document text. On the editing screen they appear as symbols, such as a left-facing triangle for return or a diamond for center. When these keywords are used as part of a glossary program, the symbol for the keyword is typed in the document at the cursor location.

Editing Keywords

Keywords such as format, search, copy, insert, and delete cause a function to occur when the text document is being edited.

Cursor Movement Keywords

Keywords such as left, north, backspace, and prevscrn move the cursor to a specific location in the document without changing the text or the format. When you are using cursor movement keywords in your glossary program, remember that the cursor moves character-by-character, not position-by-position. The cursor cannot occupy blank areas of the screen. Sometimes an area may appear to be blank but is actually occupied by spaces. The cursor can move across spaces in the same way it moves across characters.

Combination Keywords

To use some keywords you must use a combination of keywords. For example, to invoke a Right-flush Tab you must use the keyword combination command indent.

Formatting Keywords

Keyword	Performance in glossary program
center*	The center symbol appears on the screen; any following text is centered.
dectab* or decimal tab*	The decimal tab symbol appears at the next available tab stop. Either keyword can be used.
indent*	The indent symbol appears on the screen at the next available tab stop. The text after it is indented.
page*	Inserts an optional page or column break.
PAGE*	Inserts a required page or column break.
return*	The return symbol appears on the screen and the cursor moves down one line.
subscript*	The subscript symbol appears on the screen.
superscript*	The superscript symbol appears on the screen.
tab*	The tab symbol appears on the screen at the next available tab stop.

Editing Keywords

Keyword	Performance in glossary program
cancel*	An executing function is canceled, or the document edit is canceled and the "END OF EDIT options" screen appears.
command	The command function is invoked and the message "Which command?" appears on the screen.
copy	The copy function is invoked and the message "Copy what?" appears on the screen.
COPY	The copy text between documents function is invoked. The "COPYING TEXT BETWEEN DOCUMENTS" screen appears.
delete*	The delete function is invoked and the message "Delete what?" appears.
execute*	Completes other keyword functions, such as insert, delete, copy, and move.
format*	The cursor moves up into the first available format line and the screen message "Change format" appears. To create an alternate format line use the keywords insert format.
glossary or gl	The glossary function is invoked and the message "Which entry?" appears. This is the same as pressing the GL key on the keyboard. You can use either keyword.
help	The word processing HELP screen is invoked.
insert	The insert function is invoked and the message "Insert what?" appears.

Editing Keywords (continued)

Keyword	Performance in glossary program
merge*	The left-hand symbol for merge appears on the screen.
MERGE*	The right-hand symbol for merge appears on the screen
mode	The mode function starts and the message "What mode?" appears. This must be followed by a character in quotes, indicating which mode to use, such as "b" for boldface mode, or "F" for flash mode.
move	The move function is invoked and the message "Move what?" appears on the screen.
MOVE	The move text between documents function is invoked. The "MOVING TEXT BETWEEN DOCUMENTS" screen appears.
note*	The note symbol appears on the screen.
quote*	The keyword quote must be used when quotation marks are required within a string. (The double quote symbol is not permitted in a quoted string.)
replace	The replace function is invoked and the message "Replace what?" appears.
REPLACE	The global search and replace function is invoked. The "GLOBAL SEARCH AND REPLACE" screen appears.
search	The search function is invoked and the message "Search for what?" appears.

Editing Keywords (continued)

Keyword	Performance in glossary program
SEARCH	The cursor is moved to the beginning of the document and the search function is invoked. The message "Search for what?" appears. Use the keywords command search to invoke a backward search.
stop	The Autosave function is invoked and the message "Keystrokes before saving?" is displayed.

Cursor Movement Keywords

Keyword	Performance when used in glossary program
backspace*	The cursor moves back one character.
down*	The cursor moves down one line. If there is no text immediately below it on the next line, the cursor will not occupy the same position it did on the previous line. Alternatively, you can use the keyword south.
DOWN*	Moves the cursor according to the current cursor mode. This is equivalent to pressing the Shift key and the Down cursor key simultaneously. Alternatively, you can use the keyword SOUTH.
east*	The cursor moves one character to the right. You can also use the keyword right.

Cursor Movement Keywords (continued)

Keyword	Performance when used in glossary program
goto	The cursor moves to a specified location in the document. For example: goto "12", goto "e", goto nextscrn, goto left.
left*	The cursor moves one character to the left. You can also use the keyword west.
nextscrn*	The cursor moves forward to the first character on the next full screen. The keywords goto nextscrn move the cursor to the top of the next page.
north*	The cursor moves up one line. If there is no text immediately above it, the cursor will not occupy the same position it did on the previous line. You can also use the keyword up.
NORTH*	Moves the cursor according to the current cursor mode. This is equivalent to pressing the Shift key and the Up cursor key simultaneously. Alternatively, you can use the keyword UP.
prevscrn*	The cursor moves to the first character on the previous full screen. The keywords goto prevscrn move the cursor to the top of the previous page.
right*	The cursor moves one character to the right. You can also use the keyword east.
south*	The cursor moves down one line. If there is no text immediately below the cursor on the next line, it will not occupy the same position it did on the previous line. You can also use the keyword down.

Cursor Movement Keywords (continued)

Keyword	Performance when used in glossary program
SOUTH*	Moves the cursor according to the current cursor mode. This is equivalent to pressing the Shift key and the Down cursor key simultaneously. Alternatively, you can use the keyword DOWN.
up*	The cursor moves up one line. If there is no text immediately above the cursor, it will not occupy the same position it did on the previous line. You can also use the keyword north.
UP*	Moves the cursor according to the current cursor mode. This is equivalent to pressing the Shift key and the Up cursor key simultaneously. Alternatively, you can use the keyword NORTH.
west*	The cursor moves one character to the left. You can also use the keyword left.

KEYWORD ABBREVIATIONS

Keyword abbreviations allow you to embed keywords in quoted strings. Not every keyword has a corresponding abbreviation; those that do are listed in Table D-4.

Keyword Abbreviations

Function	Code	Keyword Syntax
backslash	\\	
bold face ON	\B	mode "b"
bold face OFF	\b	mode "b"
center	\c	center
decimal tab	\.	dectab or decimaltab
		mode"f"
flash (blink) ON	\F	mode"f"
flash (blink) OFF	\f	command "n"
footnote reference	\N	help or command help
help	\h	
		command "-"
hyphen (generated)	\H	
hyphen (optional)	\-	indent
indent (generated)	\I	merge
indent	\i	MERGE
merge ON	\<	note
merge OFF	\>	mode "/"
note	\n	mode "/"
overstrike ON	\O	page
overstrike OFF	\o	PAGE
page break (optional)	\g	quote
page break (required)	\G	return
quote (double)	\q	
return (required)	\r	mode "r"
return (word wrap)	\w	mode "r"
reverse video ON	\V	command indent
reverse video OFF	\v	command " "
right-flush tab	\R	
space (required)	\(space)	

Keyword Abbreviations (continued)

Function	Code	Keyword Syntax
stop	\p	stop (Autosave)
superscript	\S	superscript
subscript	\s	subscript
tab	\t	tab
underline ON	\U	mode "_ "
underline OFF	\u	mode "‾ "
underline (double) ON	\D	mode "‾="
underline (double) OFF	\d	mode "="
octal representation**	\nnn	

* The word space in parentheses represents a typed space that is not visible.

** Octal number abbreviations in strings are covered in Appendix C; the "nnn" stands for a three-digit octal code.

APPENDIX E

ERROR MESSAGES

This appendix lists Glossary error messages. Glossary error messages are grouped in two types: verification error messages that appear on page w (workpage) of the glossary document, and glossary operation messages that occur when you attempt to attach a glossary document or use an entry.

VERIFICATION ERROR MESSAGES

All verification errors messages are preceded by the legend: page n, line n: where n stands for the page and line number of the error in the glossary document. For example, suppose that entry a below is on page 6 of glossary document gltest and the entry contains two errors. When gltest was verified, the error messages following entry a were posted on page w.

```
entry a
{
    call posmsg("Enter Amount:  \007")
    amount = keys
    "\027"
    call feed(amount)
}
```

```
***********************************
Tue Aug 7, 1984 at 14:43:39
***********************************

page 6 , line 3 : 3 arguments expected for posmsg()
page 6 , line 10 : syntax error
```

The first error message reports "3 arguments expected for posmsg()." The term "arguments," as used by the compiler, means "expression(s)." In entry a, expressions one and two, the line and position numbers were omitted. The second error message reports a "syntax error" on line 10. The syntax error is due to the reversed ending brace. The ending brace should be }.

The error message "syntax error" is reported for a wide variety of situations. The best procedure is to check for the most obvious errors first, such as missing commas between expressions, reversed braces or parentheses and so on. As you become familiar with glossary, you will be able to spot most syntax errors before verification.

Occasionally the compiler reports syntax errors on the line below the line that contains the error. Be sure to check the line above if you can't find the error on the reported line.

Verification Error Messages

Message	Possible Errors
syntax error	An error exists in the statement syntax. When "syntax error" is displayed alone, the error could be missing or incorrect symbols are present. When the error is followed by a colon and another message, the error is specific to the message.

Any of the messages below may follow "syntax error."

: Improper use of function	Function not preceded by the call statement; function misspelled; function cannot be used in the statement.
: Unexpected variable	Parentheses around a function argument are missing; extraneous text exists in the glossary document; a keyword is misspelled.
: Cannot start another entry here	The ending brace on the previous entry is missing.
: call	The call statement is not complete.
: if	The if statement syntax is not correct.
: else	The else statement syntax is not correct.
: jump	The jump statement syntax is not correct.

Verification Error Messages (continued)

Message	Possible Errors
: while	The while statement syntax is not correct.
: do	The do statement syntax is not correct.
: Assignment operator	The assignment operator = is used incorrectly.
: Comparison operator	One of the comparison operators ==, !=, <=, >= is used incorrectly.
: Keystrokes not allowed	A keyword or function name is misspelled; a syntax statement is incorrect.
unmatch character detected( )	The character in parentheses may be one of the following: (,), [,], or ". A missing quote is the most common error causing this message.
n argument(s) expected for x	n stands for number of arguments and x stands for function requiring arguments. The expected number of arguments (expressions) for the function are not present.
multiply defined entry name	Two entries in the glossary document have the same label.
Unknown symbol	An incorrect symbol appears in the entry; this error is most common in mathematical applications.

Verification Error Messages (continued)

Message	Possible Errors
Illegal glossary entry name	Too many characters are in the entry label; an illegal symbol appears in the entry label.

GLOSSARY OPERATION ERROR MESSAGES

Glossary operation error messages occur when you are attaching a glossary document or using a glossary entry.

Message	Possible Errors
Cannot attach	The glossary document entered does not exist or the name was entered incorrectly.
No glossary entry	The glossary label entered does not exist or it was entered incorrectly.
Bad location	The line and position numbers specified for the posmsg or clrpos functions exceed the allowable range (lines 1 through 25, positions 1 through 80).
Unknown function	A records processing function such as sort or select-record, is part of a regular glossary entry running in a text document.

INDEX