FORTUNE:WORD™

GLOSSARY USER'S GUIDE

Office Automation

FORTUNE SYSTEMS

Fortune:Word™ Glossary User's Guide

FORTUNE SYSTEMS

300 Harbor Boulevard
Belmont, CA 94002

1006923-01

Contents

About This Guide

This guide is a learning and reference guide for Fortune:Word Glossary functions. Glossary is a Fortune:Word feature that you use to store frequently-typed words and phrases. You can recall the stored text in your document with just two keystrokes.

Using a glossary entry, you can automate almost any word processing task. You can use glossary entries to insert standard paragraphs in your document as you are editing it. If you have extensive document or paragraph assembly requirements, consider using the Fortune:Word Document Assembly feature for this purpose.

In addition to text and keyword storage and recall, Glossary gives you full programming capabilities with such functions as:

- Mathematical
- String
- Interactive
- Document reading and writing
- Display

You can define and initialize variables, use relational and assignment operators, logical operators, conditional functions, and control functions.

HOW THIS GUIDE IS ORGANIZED

This book has 13 chapters, five appendices, and an index. Following is a brief description of each part:

Chapter 1: About Glossary contains a brief introduction to Glossary.

Chapter 2: Creating a Glossary Document describes how to create, verify, attach, and detach a glossary document.

Chapter 3: Creating a Glossary By Example Entry describes how to store keystrokes in a glossary document while you are typing them in a text document.

Chapter 4: Writing Glossary Entries describes the basic syntax required for glossary entries, how to write simple entries, and how to modify a glossary by example entry. It contains additional information about correcting verification errors, and attaching and detaching a glossary document.

Chapter 5: Introduction to Glossary Syntax describes additional syntax and provides an overview of the elements of the Glossary language.

Chapter 6: Elements of the Glossary Language contains information about statements, variables, values, logical values, operators, functions, arguments, expressions, and parentheses.

Chapter 7: Conditional Statements describes the statements that you can use to test for specific conditions while a glossary entry is running.

Chapter 8: Control Statements describes the Glossary functions that allow you to control the flow and execution order of the elements of a glossary entry.

Chapter 9: Function Description List contains an alphabetical list of all the Glossary functions. The type, value, syntax, and a brief description is provided for each function. You can use this chapter as a reference guide.

Chapter 10: Function Usage List contains information on all Glossary functions grouped by the types of actions they perform. You can use this chapter together with Chapter 9 as a reference guide.

Chapter 11: Administering Glossary Entries contains suggestions for maintaining, updating, and filing glossary documents.

Chapter 12: Glossary Information for FOR:PRO Users contains operating system information that relates to glossary documents.

Chapter 13: Glossary Entry Examples contains an index of the glossary examples that are provided on diskette with this guide.

Appendix A: Reserved Words and Symbols contains a list of the words and symbols that are reserved for specific use by the Glossary language.

Appendix B: Comparison of Glossary Keywords and Functions contains an alphabetical list of keywords and functions used by the Glossary language and by Records Processing.

Appendix C: Character Codes contains information about the ASCII collating sequence, octal number conversions, attribute codes, and Fortune:Word document format control codes.

Appendix D: Keywords by Usage provides a list and description of Glossary keywords grouped by the functions they perform.

Appendix E: Error Messages contains a list of verification error messages and glossary operation error messages.

HOW TO USE THE GLOSSARY ENTRY EXAMPLES

A Glossary Examples Diskette is provided with this guide. It contains documents with all the entries used as examples. Retrieve the glossary documents on this diskette and try the entries as you use this guide. Observing a glossary entry as it performs its functions can help clarify the descriptive text.

One way to learn to write your own glossary entries is to take an existing entry and make modifications to it. The entries on this diskette represent one approach to Glossary. There are many ways you can write an entry for any application. The suggestions in this guide are not the only solutions, and they may not even be the best.

HOW TO USE THIS GUIDE TO LEARN GLOSSARY

Do not feel that you need to learn everything in this guide before you begin to use Glossary for your word processing needs. Read the first four chapters of this guide for introductory information to Glossary. You can then begin to use Glossary to store and recall frequently typed words and phrases. This convenient production tool may be all that you ever use of the Glossary language.

As you become more familiar with Glossary, you may want to further automate some of your entries. When this happens, refer to the part of this guide that discusses the function you want to perform. For example, if you want to add a test for a specific condition, refer to Chapter 7, "Conditional Statements." If you want to check to see if the cursor is at the end of the document, look up the "end_doc" function in Chapter 9, "Function Description List."

Another way to learn Glossary is to take an existing entry and modify it to suit your needs. You can experiment with the entries on the Glossary Examples Diskette, modifying them to suit your specific needs.

Once you learn to use glossary by example to store text and keystrokes, you can modify those entries, adding conditional statements and testing for specific conditions. Even if you write complex entries, you can start the entry by capturing the basic function keystrokes using glossary by example.

CONVENTIONS USED IN THIS GUIDE

The following conventions are used throughout this guide:

- The names of keyboard functions and editing keys you press are capitalized, as shown in the following examples:

  RETURN   SEARCH   INDENT   EXECUTE

- Glossary keywords have different functions depending on whether the keyword is lowercase or uppercase. Glossary keywords are shown in the case that represents the function that is performed. For example **search** in a glossary entry searches from the cursor position for the string specified in quotation marks. **SEARCH** searches from the beginning of the document for the string specified in quotation marks.

- The four directional arrow keys (up, down, left, and right) that move the cursor are identified as follows:

  UP   DOWN   LEFT   RIGHT

- To perform certain Fortune:Word functions, you have to hold down one key while you press a second key. This combination of keystrokes is shown in the following way:

  Press SHIFT/COPY
  Press SHIFT/MERGE
  Press CTRL/y

  In the examples above, you hold down SHIFT while you press COPY, MERGE, or y.

- Words or phrases you type are in boldface type, as shown in the following example:

  Type **rad**

- Screen prompts, messages, and menu selections are in italic type, as shown in the following examples:

  Press EXECUTE to continue
  Select *Edit Old Document* from the Main menu.

- Fortune:Word document names are in bold type as shown in the following example:

  Glossary document **gloss1**

- The term "text document" refers to a standard Fortune:Word document, to distinguish it from a "glossary document" containing executable entries. The term Glossary is used to refers to the Glossary language. The term "glossary" is used to refer to a glossary document.

- Glossary keywords, functions, and variables in the descriptive text are in bold type as shown in the following examples:

  it adds 1 to the variable **figure_no**
  If the search fails, the statement **if(globerr) {execute exit}** causes the entry to end.

- 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 shown in the following example:

```
function(expression1,expression2,...)
```

In some diagrams and figures, omitted program statements are represented by three periods enclosed in parentheses, as shown in the following example:

```
(...)
```

RELATED FORTUNE SYSTEMS DOCUMENTS

Following is a list of related Fortune Systems Fortune publications:

- *How to Use Fortune:Word*
- *Fortune:Word Glossary User's Guide*
- *Fortune:Word Records Processing User's Guide*
- *FOR:PRO User's Guide*
- *Using Fortune Terminals*

Chapter 1
About Glossary

Anyone who knows Fortune:Word can learn to use Glossary.

If you are not familiar with Fortune:Word you can learn how to use it by studying *How to Use Fortune:Word* and referring to the *Fortune:Word Reference Guide*.

WHAT IS GLOSSARY?

Fortune:Word Glossary is a special type of document where you can store frequently-typed words and phrases. You can then recall the text with just two keystrokes, automatically typing it in a document. You can also perform word processing functions such as Center, Indent, or Execute automatically by storing keywords in your glossary document.

You type the text and keywords as a glossary entry in a 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.

To use a glossary entry while you are editing a document, you attach the glossary, 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. Entry a is an example of a simple glossary entry that types a company name. Braces mark the beginning and ending of the glossary entry.

```
entry a
{
  "Fortune Systems"
}
```

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 b
{
   center "Fortune Systems" return(2)
}
```

You save typing time and improve typing accuracy when you use glossary entries to type text and perform repetitive functions. Glossaries are great timesavers when you frequently type legal or engineering phrases such as "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 c
{
   insert "(trademark pending)" execute
}
```

Glossary entries can automate almost any word processing task. You can create glossary entries that 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. Any repetitive keystrokes you perform can be stored in a glossary entry and recalled with only two keystrokes.

In addition to text and key function storage, Glossary contains the following tools:

- 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

You can use glossary entries to serve your production needs in the following ways:

- Interactive glossary entries: write interactive glossary entries with your own prompts and error messages. Write entries that stop during recall and permit the operator to enter data, then continue the entry. You can design 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 Fortune:Word 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.

HOW TO CREATE GLOSSARY ENTRIES

You can create glossary entries in two ways:

1. Glossary by example: create a glossary by example entry in a glossary 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 can also storing them in your glossary for later recall. Chapter 3 describes how to create a glossary by example entry.

2. Write a glossary entry: write a glossary entry by typing it in a glossary.

   You can utilize the full power of Glossary when you type your glossary entries directly into a glossary. A glossary by example entry only allows you to store keystrokes (keywords and characters). When you write a glossary entry directly, you can use the full range of Glossary functions, such as **if** statements, **while** loops, subroutines, and math and string functions. Chapter 4 describes how to write a glossary entry in a glossary.

Chapter 2
Creating a 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 is usually created and edited from the Glossary Functions menu. The Glossary Functions menu is accessed by selecting *Glossary Functions* from the Fortune:Word Main menu.
- The glossary contains glossary entries instead of text.
- The glossary must be verified each time you change or add an entry. The verification process compiles entries into executable programs, checks for syntax errors, and attaches the glossary when verification is successful.
- The glossary must be attached, either through verification, from a menu, from a Document Index screen, or from a document editing screen, before you can use an entry.

You can perform all Fortune:Word editing and formatting functions in a glossary. You can also perform menu functions such as Delete, Rename, Move, Copy, Print, or Archive on a glossary.

Glossaries are created separately from text documents. When you edit a text document, you attach the glossary and use the glossary entries to enter text or perform functions. Figure 2-1 illustrates the relationship of a glossary and glossary entries to a text document.

THE GLOSSARY FUNCTIONS MENU

You can perform most glossary activities from the Glossary Functions menu. The Glossary Functions menu is shown in Figure 2-2. To display the menu, select *Glossary Functions* from the Fortune:Word 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 glossaries.

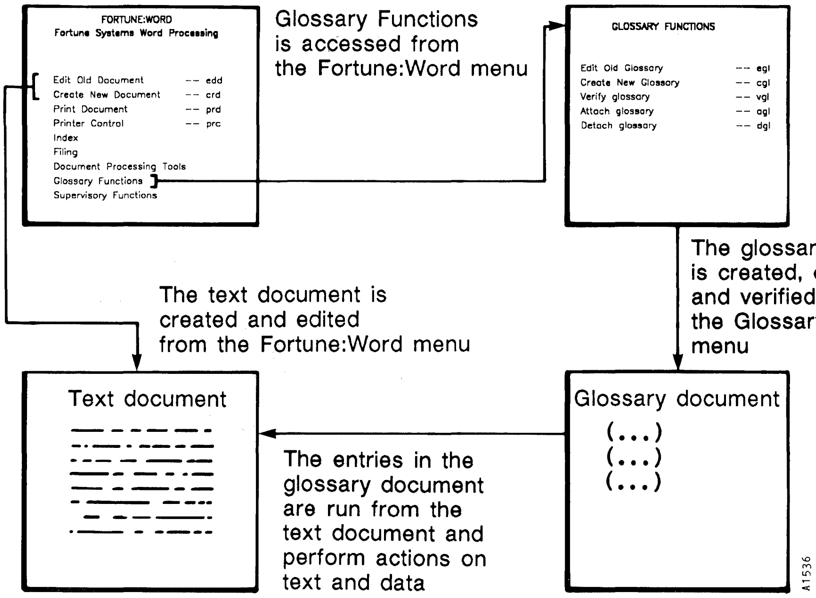

FORTUNE:WORD
Fortune Systems Word Processing
Edit Old Document -- edd
Create New Document -- crd
Print Document -- prd
Printer Control -- prc
Index
Filing
Document Processing Tools
Glossary Functions
Supervisory Functions
Glossary Functions is accessed from the Fortune:Word menu
GLOSSARY FUNCTIONS
Edit 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
The text document is created and edited from the Fortune:Word menu
Text document
Glossary document
(...)
(...)
(...)
The entries in the glossary document are run from the text document and perform actions on text and data
A1536

Figure 2-1. *Relationship Between the Glossary Document, Glossary Entries, and the Text Document*

The step-by-step instructions in the next section show you how to use each of the selections on the Glossary Functions menu.

HOW TO CREATE A GLOSSARY

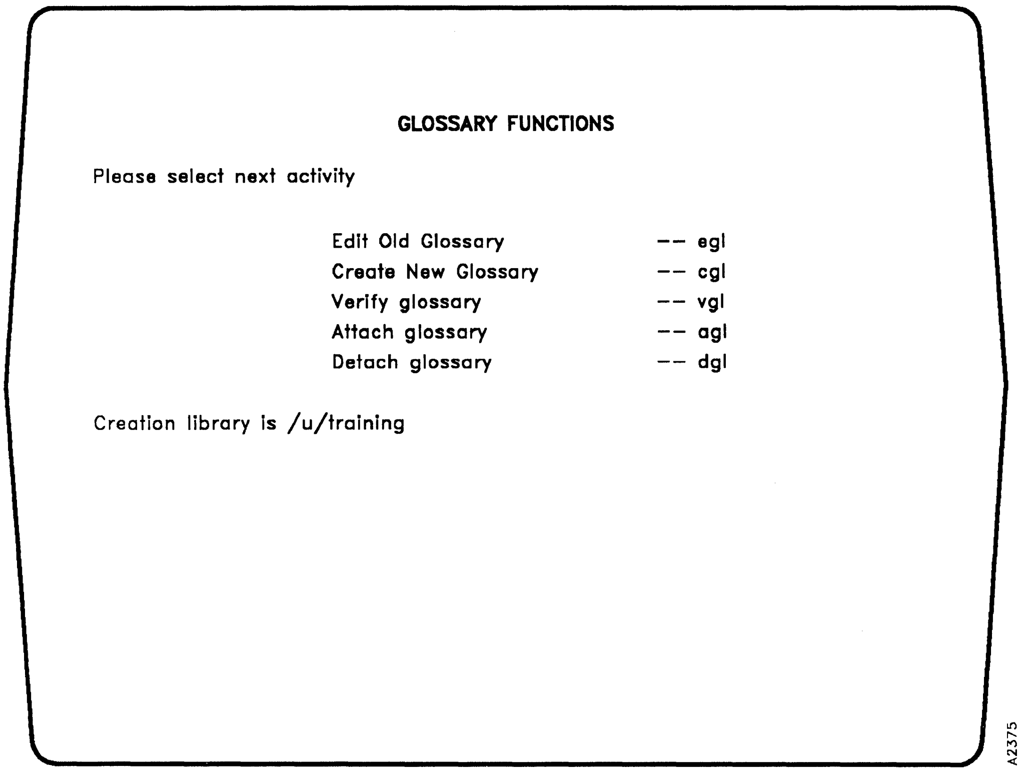
This section shows you how to create, verify, attach, and detach an empty glossary.

Creating the Glossary

You can create a glossary in three ways:

- Select *Create New Document* from the Main menu. The new glossary is not automatically verified if you use this selection.

- Use the shortcut code **cgl** from any menu. The new glossary is verified at the end of editing when you use this selection.



```
                         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
```

Figure 2-2. The Glossary Functions Menu

- Select *Create New Glossary* from the Glossary Functions menu. The new glossary is automatically verified at the end of editing when you use this selection.

1. Select *Glossary Functions* from the Main menu.
2. Press EXECUTE.
3. Move the marker to *Create New Glossary*.
4. Press EXECUTE.

When you name your glossary, follow the same naming conventions that apply to a text document. The prototype document you select can be a text document, a glossary, or the default **0000**.

5. Type **gloss.**

6. Press EXECUTE.

7. Press EXECUTE to accept the prototype default **0000**.

Notice that the summary screen is called a Glossary Summary rather than a Document Summary. Fill in the Glossary Summary just as you do a Document Summary. The *Statistics* portion of the Glossary Summary is the same as a Document Summary.

8. Fill in the Glossary Summary by typing the *Document title*, the *Operator*, the *Author*, and the *Comments* lines.

9. Press EXECUTE.

10. The editing screen is displayed.

You use the editing screen just as you would for a text document. You can change the format line or add alternate format lines. Changing the format line in a glossary does not affect the format of an entry when you recall it in a document. Like a text document, the glossary has header, footer, note, and work pages.

Do not enter any text in the glossary at this time. You have created it so you can create a glossary by example.

Leaving the Glossary

1. Press CANCEL to display the End Of Edit Options menu.

2. Press EXECUTE. The message *Verifying* is briefly displayed at the bottom of the screen. The messages *Press EXECUTE to continue* and *An empty glossary is attached* are displayed. The glossary is empty because you have no entries in it at this time.

3. Press EXECUTE.

4. Press CANCEL to return to the Main menu.

You have created, verified, and attached your new glossary **gloss**.

Verifying the Glossary

Any time you leave an existing glossary and save the changes, the verification process checks for errors and compiles glossary entries into an executable form so you can use them in your text document. When the verification is successful, the glossary is automatically attached so it is available for immediate use. Chapter 4 describes the glossary verification procedure.

If you use the Main menu to create a new glossary, you must either 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 for the first time. Thereafter, the glossary is verified at the end of editing no matter which menu selection you use.

1. If you created the glossary from the Main menu, select *Verify glossary* from the Glossary Functions menu or use the shortcut code **vgl**.

2. Press EXECUTE.

Attaching the Glossary

Before you can use the entries in a verified glossary, you must attach it. You can attach a glossary in five ways:

- 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 while editing a text document by pressing COMMAND, pressing GL, and typing the glossary name.

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

- Select *Index* from the Main menu (or use one of the index shortcut codes to display a Document Index screen), position the cursor on the glossary name, and press the GL key. Glossaries are identified on the Document Index screen by two asterisks (**) displayed before the glossary name. The asterisks are automatically added to a glossary name when it is verified.

Once a glossary is attached, you can use any entry in it. When the glossary is attached from a menu, the glossary is available for use with any document you edit during an editing session. When you attach a glossary from an editing screen, the glossary is automatically detached when you leave the document.

Detaching the Glossary

When you attach a glossary you automatically detach any previously attached glossary. You may need to detach a glossary from one terminal to allow a user to edit or archive it from another terminal on a multiuser system. You can detach an attached glossary in two ways:

- Select *Detach Glossary* from the Glossary Functions menu.
- Use the shortcut code **dgl** from any menu.

Chapter 3
Creating a Glossary By Example Entry

Using the Glossary by Example feature, you create a glossary entry that duplicates your keystrokes as you perform them. Keystrokes include typing text, Fortune:Word formatting keys like TAB and RETURN, and function keys like INSERT and EXECUTE.

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 attached before you can create an entry by example.

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

HOW TO CREATE A GLOSSARY BY EXAMPLE ENTRY

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

- Entry c types a company name.
- Entry d types a centered company name and address.
- Entry e inserts a name and title.

Entry c: Creating an Entry to Type a Company Name

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. Follow these steps to create a new text document:

1. Select *Create New Document* from the Main menu.

2. Name the document **learngloss**. Go to the editing screen.

3. The cursor is on Line 1, Pos 1 of your text document.

You must attach your glossary **gloss** before you can create a glossary by example. If you created and verified **gloss** and have not left Fortune:Word, it is attached. If you left Fortune:Word or attached another glossary, follow the steps below to attach **gloss**.

4. Press COMMAND.

5. Press GL.

6. Type **gloss**.

7. Press EXECUTE.

Start Glossary by Example mode:

8. Press MODE.

9. Press GL.

The flashing message *Glossary entry* is displayed at the bottom center of the screen. This message continues to flash until you have completed your entry.

Type the entry exactly as you want it to appear in your document. 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 end the entry and start over.

> **REMEMBER:** Every keystroke you make, including mistakes, is duplicated in your glossary entry.

10. Type **Fortune Systems**.

When you have finished typing the entry, follow these steps to conclude your entry.

11. Press MODE.

12. Press GL.

You must now assign a label in response to the *Which entry?* prompt. An entry label consists of one character. You can use any one of 94 keyboard characters as an entry label including the Space bar (allowing you 94 entries per glossary). Do not use quotation marks as an entry label.

You cannot duplicate entry labels in the same glossary. 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.

13. Type c.

14. Press EXECUTE.

Once entry c has been created it can be used in any document. To recall the entry, follow these steps:

15. Press GL.

16. Type c.

Try recalling the entry several times. Notice how quickly it is typed in your document.

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

If you did not type a space following "Fortune Systems" and recalled entry c a few times, your screen probably looks like this:

    Fortune SystemsFortune SystemsFortune SystemsFortune Systems

If you typed a space following the word "Systems" when you created the entry, your screen probably looks like this:

    Fortune Systems Fortune Systems Fortune Systems Fortune Systems

Remember that if you recall a word or phrase that requires punctuation following it, you probably do not want to type a space at the end of the phrase when you create the entry. If you know the phrase will never occur at the end of a sentence or be followed by punctuation, including the space means you do not have to type it in your document. Give some consideration to possible uses for your entries. It is a good idea to be consistent in the use of spaces following phrases to ensure a smooth work flow as you recall each entry.

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

Entry d includes both text and function keys. The function keys used in this example are CENTER and RETURN.

Begin this exercise from the editing screen of your text document. Skip the first three steps if you did not leave the document at the end of the last exercise.

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

2. Attach the glossary, **gloss**

   a. Press COMMAND and then GL.
   b. Type **gloss**, and press RETURN or EXECUTE.

3. Move the cursor to the end of the document.

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

Start glossary by example:

4. Press MODE.

5. Press GL.

Begin typing the entry:

6. Press CENTER.

7. Type **Fortune Systems**.

8. Press RETURN.

9. Press CENTER.

10. Type **300 Harbor Boulevard**.

11. Press RETURN.

12. Press CENTER.

13. Type **Belmont, CA 94002**.

14. Press RETURN.

Conclude entry d:

15. Press MODE.

16. Press GL.

Assign the entry label:

17. Type **d**.

18. Press EXECUTE.

Recall entry d:

19. Press GL.

20. Type **d**.

Notice that entry d is typed in your document exactly as you typed it when you were creating the entry, complete with Centers and Returns. When you recall it in your text document, the entry should look like this example:

```
                    Fortune Systems
                  300 Harbor Boulevard
                   Belmont, CA  94002
```

You can see from entry d that using function keys (such as CENTER and RETURN) in an entry provides more possibilities for creative glossary applications.

When you use entry c or d, you must put the cursor beyond or below any existing text before you recall the entry. Otherwise the entry text overwrites the existing text. You can use 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

Begin this exercise from the editing screen of your text document. Skip the first three steps if you did not leave the document at the end of the last exercise.

1. Edit your text document, **learngloss**.
2. Attach the glossary, **gloss**.
3. Move the cursor to the end of the document.

Start glossary by example:

4. Press MODE.
5. Press GL.

Begin typing the entry:

6. Press INSERT.
7. Type **Mr. John Jones, President**.
8. Press EXECUTE.

Conclude entry e:

9. Press MODE.
10. Press GL.
11. Type **e**.
12. Press EXECUTE.

To understand how entry e inserts text, put the cursor where you want the text inserted, and recall the entry.

13. Press GL.
14. Type **e**.

Entry e should look like the following example:

    Mr. John Jones, President

TIPS ON CREATING AND USING GLOSSARY BY EXAMPLE ENTRIES

The 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 at the place where you want the entry to be typed. If the cursor is positioned on existing text, the text is overwritten when the entry is recalled.

- 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 only 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 could 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 as part of 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 in a glossary by example entry (see "Length").

  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. Chapter 4 explains how to do this.

- Length: A glossary by example entry 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 entry records every keystroke you make, it includes the keys you press to make corrections or move the cursor. Unless you are very sure exactly what the entry should contain, you may quickly reach the maximum entry size. Text formatting keys such as RETURN and TAB count as more than one character. See Appendix C in this guide for more information.

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 stop.

- Modifying or adding to an entry: You can edit your glossary 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: You can create as many glossaries 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 either be entries you write or create by example.

- 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/sublibrary/document*** 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. Then follow the same procedure you learned in this chapter to create a glossary by example entry.

- Suggestions for creating glossary by example entries: There are many ways to use glossary by example entries. Consider using entries for repetitious typing, standard paragraphs, Records Processing field labels (the *Fortune:Word 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 and archive a glossary. 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 glossaries and keep them in a binder for reference.

Chapter 4

Writing Glossary Entries

This chapter describes how to write entries in a glossary. When you know how to write glossary entries, you can:

- Modify or add to your glossary by example entries
- Create longer glossary entries (up to 33,000 characters)
- Use all the functions available in Glossary

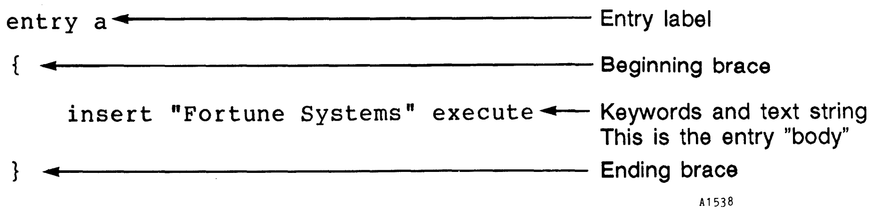
Before you write a glossary entry, you need to understand the elements that compose the entry. Every glossary entry, including the entries you created by example in Chapter 3, contains the same basic elements. These elements are described in the following section.

BASIC ELEMENTS OF A GLOSSARY ENTRY

A glossary entry is composed of the following basic elements:

- Entry label
- Braces
- Keywords
- Strings
- Comments

Figure 4-1 shows the elements of a short glossary entry that inserts the text, "Fortune Systems Corporation," in a document.



```
entry a
{
     insert "Fortune Systems" execute
}
```

Figure 4-1. Elements of a Glossary Entry

Entry e (the glossary by example entry from Chapter 3) is similar to the example shown in Figure 4-1. If you would like to compare the entries, edit your glossary **gloss**, 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 entry e and Figure 4-1 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.

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. If you use a space or backslash as an entry label when you create a glossary by example, the backslash before the character is automatically entered in the glossary. To use a space as a label, type **entry \ (space)**. To use a backslash as a label, type **entry \\**.

Each entry label must be unique; you cannot have two entries with the same label in the same glossary. When you use a glossary entry in your Fortune:Word document, 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

A brace marks the beginning and the end of an 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 "Fortune Systems."

Keywords

Keywords represent the formatting, editing, and cursor movement keys on the keyboard, such as RETURN, TAB, DELETE, and LEFT.

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

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

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

Strings

A string is any continuous set of characters that is 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 entry g:

```
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, centered, and followed by a Return.

Embedding Keywords in Strings

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

Entry h is the same as entry g, 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 \qFortune Systems Corporation\q"
```

This string appears in the text document as:

> The name of the company is "Fortune Systems Corporation"

You can also use the keyword **quote** to enclose words or phrases in quotation marks:

```
"The name of the company is" quote "Fortune Systems 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. When surrounded by double quotes, the embedded keywords in a string are interpreted and their functions are carried out; when single-quoted, any embedded keywords in a string are printed verbatim rather than interpreted. For example, when the string

> "\cFortune Systems Corporation\r"

is typed in the text document, the center symbol is typed, the string "Fortune Systems 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

> '\cFortune Systems Corporation\r'

is typed in the text document as

> \cFortune Systems 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:

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

The string is typed in the document as

> The backslash (\) is a special character.

Comments

Comments describing the entry make glossary entries easier to understand and use. Any text enclosed between /* and */ within an entry is a comment. When a glossary is verified or an entry is executed, comments are ignored. Instructional and explanatory comments can therefore be used frequently. Comment lines are used to clarify the function of entry i.

```
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 shows the correct usage of both instructional and explanatory comments.

> **REMEMBER:** Regardless of how complicated your glossary entries become, they share the same structural elements. 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 A GLOSSARY

Screen symbols that are displayed on the editing screen of your glossary, such as the Return and Tab triangles and the Center diamond, are not recognized as keywords in the glossary entry. 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 entry. You can use Return, Tab, Indent, and other screen symbols to format your glossary entry so it is easier to read on the editing screen.

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 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 entry.

Entry j is a short entry 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 suggests a glossary entry you can write and try. If you want to understand more about the action of a particular keyword as you are writing the examples, refer to Appendix D, "Keywords by Usage."

Modifying a Glossary by Example Entry

You can add additional text or functions to an existing entry created by example by editing the entry in your glossary. You cannot use the glossary by example feature to change or add to an existing entry created by example. To add a phrase to entry d (the glossary by example entry you created in Chapter 3), perform the following steps:

1. Select *Glossary Functions* from the Main menu.

2. Select *Edit Old Glossary*.

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

4. Entry d in your glossary should be similar to the following example. Text lines may be wrapped differently, depending on the format line in your glossary. 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 "Fortune Systems Corporation" return center "300 Harbor Boulevard"
return center "Belmont, CA  94002" return
}
```

5. Put the cursor at the beginning of the entry body (the "c" in the first "center"), and press INSERT.

6. Type the following line:

   "Please send correspondence to:" return(2)

7. Press RETURN, then press EXECUTE. Entry d should now look similar to the following example:

```
entry d
{
"Please send correspondence to:" return(2)
center "Fortune Systems Corporation" return center "300 Harbor Boulevard"
return center "Belmont, CA  94002" 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, as in 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:

```
"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 most efficiently.

Recalling 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. When the glossary verifies correctly, the menu from which you edited the glossary is displayed.

   If the glossary does not verify correctly, the verification screen is displayed. Refer to the "Verifying and Troubleshooting" section later in this chapter for information on how to correct verification errors.

5. The glossary 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.

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, **gloss**.

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 (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 together on one page.

3. Type entry k as shown in the following example, substituting any of the headings with ones you normally use. 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.

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 entry.

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 again in your text document. If you are writing a complicated glossary entry, you may need to go back and forth between the glossary and the text document several times until you are satisfied with the entry's performance.

Writing Menu Glossary Entries

You can write a glossary entry to perform keystrokes on a menu. 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 creates a document, fills out the Document Summary, and takes you to the editing screen. The entry stops at that point, even if there are more keystrokes in the entry. To enter text you must use a different glossary entry. Both entries can be in the same glossary.

Learning More About Glossary

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

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

The "Keywords by Usage" list and the "Keyword Abbreviations" list in Appendix D are particularly useful. Study these lists carefully before you begin the next chapter."

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 entries. You may find that you want to add more functions, or test for certain conditions. You can learn additional Glossary functions by gradually incorporating them as you find you need them. You do not need to learn all of Glossary before you begin to write and use entries.

VERIFYING AND TROUBLESHOOTING

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

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

- Entries without labels
- Duplicated labels within the same glossary
- Entries not beginning and/or ending with a brace

- Keywords misspelled and/or used incorrectly
- Strings not beginning and/or ending with a single or double quotation mark
- Comment lines not beginning and/or ending with the correct comment symbols

A list of error messages is provided in Appendix E.

Glossary Verification Options

You have a variety of options available when verifying a glossary. 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 below.

Verification Through the End of Edit Options Menu

When you leave a glossary, the End Of Edit Options menu is displayed. The functions on this menu are the same as for a text document. In addition, the glossary may be verified and attached, depending on the selection you choose.

The options on the menu are EXECUTE, RETURN, COPY, DELETE, and FORMAT. Each option is described below.

> NOTE: You can use the Autosave or COMMAND RETURN functions while you are editing your glossary. Your changes are saved as you edit; however, entries are not verified until you leave the glossary and press EXECUTE.

- EXECUTE: Verification automatically occurs when you press EXECUTE.

  Once a glossary has been created and verified, the EXECUTE option automatically verifies the document. When verification is successful, the glossary is automatically attached.

  You can create a glossary from the Main menu. If you do this, the EXECUTE option does not automatically verify the glossary. You must verify it from the Glossary Functions menu or by using the shortcut code **vgl** before you can use the entries.

- RETURN: The document is not verified. You are returned to the glossary editing screen.

- COPY: Only the glossary is verified and attached; the copy is not verified or attached. This option saves the glossary with all changes, and creates a copy of the glossary that includes only the editing changes saved by using Autosave or COMMAND RETURN.

- DELETE: The glossary is not verified, but it is attached. Any changes made during the editing session, except changes that were saved by using Autosave or COMMAND RETURN, are deleted.

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

- FORMAT: The glossary is automatically verified, attached, and the Print Document menu is displayed.

The same End Of Edit options apply when you edit a glossary from the Document Index; only the FORMAT option works differently. When you use the FORMAT option from a Document Index screen, a Print Document menu is not displayed, and the document is sent to the printer using the last settings on the Print Document menu.

Menu Verification Options

You can verify a glossary two ways without editing it:

- Select *Verify Glossary* from the Glossary Functions menu.

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

Verification Limitations

A glossary 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 glossary. The changes remain in the entry but are not executable until the glossary is verified. You can work around this by closing all other windows first. You can then leave the glossary normally and select an option that automatically verifies it.

A glossary is verified when you edit it from a Document Index screen you access by using COMMAND i or I from a text document editing screen. However, the glossary is only attached until you leave the Document Index screen. You can press GO TO PAGE and edit a different document, using entries from the glossary. Once you leave the Document Index screen, that glossary is no longer attached. You may type or edit an entry using this method, but if you want to use the glossary in the current document, you must attach it 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 verification process, the Verification Errors Options menu is displayed:

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

RETURN   to editing screen
DELETE   to original menu
```

The menu in the example shows that one error was detected in the glossary. You have two options:

- RETURN: You are returned to the glossary editing screen. Choose this option to view and correct entry errors.

- DELETE: The glossary is not verified. Any changes made during the editing session (including the errors) are saved.

  If you choose the DELETE option, you cannot access any entries in the glossary until it is successfully verified. You must eventually edit the document and correct the errors or delete the incorrect entry and verify the glossary before you can use the entries.

The verification process places messages about entry errors on the glossary work page (Page W). To see this information, edit the glossary, 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 Jan 27, 1987 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 opening 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 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 or errors.

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

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

You can only attach and use one glossary 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 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 that is in use. If you attempt to edit an attached glossary, the message *Document in use* is displayed.

You cannot delete, rename, or move a glossary that is attached from an editing screen or is attached to more than one workstation. Detach the glossary before using it with these functions.

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

- Verify the glossary. A glossary is automatically attached after it is successfully verified.
- Use the *Attach Glossary* selection on the Glossary Functions menu.
- From a document editing screen, press COMMAND, press GL, then type the glossary name.
- From any menu, use one of the index shortcut codes.
- From a Document Index screen accessed from the menu system, put the marker on a glossary name, then press GL. Glossary names are preceded by two asterisks (**).

If you use COMMAND **i** or **I** to access a Document Index screen from an editing screen, and then edit and modify a glossary, you must attach the glossary again when you return to the editing screen.

DETACHING A GLOSSARY

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. You may want to detach a glossary from another workstation so you can make editing changes or perform archiving functions.

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

Figure 4-2 summarizes the glossary entry writing, verifying, attaching, and recalling procedure.

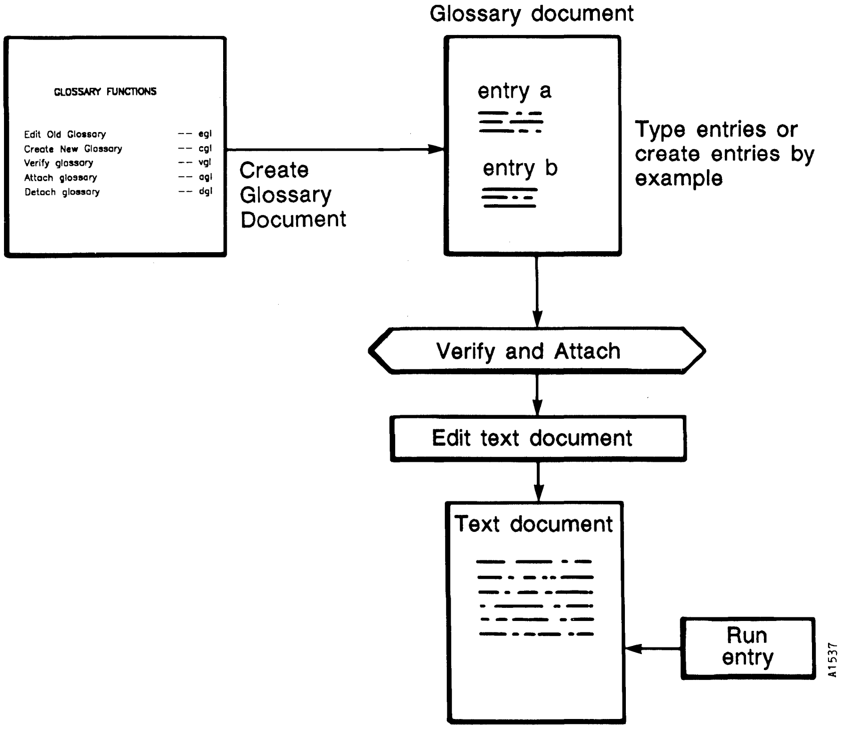


Figure 4-2. The Glossary Writing, Verifying, Attaching and Recalling Procedure

Chapter 5

Introduction to Glossary Syntax

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 Fortune:Word text document. When you write a glossary entry, you are writing a program.

The basic glossary elements you have already learned can be used to type, create headers and footers, and format your documents. The glossary functions presented in the rest of this guide greatly expand the range of possible uses for glossary entries.

WHAT IS GLOSSARY PROGRAMMING?

A glossary entry is a computer program. Your glossary entry is a set of instructions that tells the computer what to do, how to do it, and in what order to do it. When you use special glossary reading and writing functions, interactive functions, string functions, and mathematical functions, you can write more complicated instructions. Conditional and control statements permit you to control the order of entry execution.

Glossary is a Programming Language

Entries must be written in a language that the computer understands. 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."

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 language will be familiar to you.

The Verification Process Compiles Your Entries

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

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

- An interpreted language (like some forms of BASIC) is translated line-by-line as the entry executes. If you made a syntax error in your entry code, the interpreter stops the entry 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 entry and then compiles it into machine-readable form. Any errors in the entry are reported to you after it is compiled. The compiled form is called the object program. The typed form is the source program. How the object and source programs affect the glossary is described in Chapter 12, "Glossary Information for FOR:PRO Users."

The Structure of a Fortune:Word Document

A Fortune:Word document usually consists of three files:

filename	The text of a document
filename.dc	The history, statistics and page pointer information for the document
filename.fr	The formats, header, footer, work, note, and footnote pages for the document

This structure is not apparent from Fortune:Word Document Index, which displays only the base filename of the document. Fortune:Word treats all three files as one for the purposes of document control. However, for purposes of using Glossary, it is helpful for you to understand what is happening "behind the scenes."

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

filename.gl	The compiled and executable form of a glossary.

See Chapter 12 for more information about the **.gl** file and Fortune:Word document file structure.

The glossary compiler reports entry errors to you on Page W of a glossary. The glossary language has an extensive syntax error list. Chapter 11, "Administering Glossary Entries," tells you how to troubleshoot (debug) your entries. 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 LEARN GLOSSARY

You have already learned how to create and use glossary by example entries, and to write simple entries using entry labels, braces, keywords, and strings. Chapters 6 through 8 contain the fundamental knowledge you need to write more sophisticated Glossary entries. While the information is specific to Fortune:Word Glossary, the principles apply to most computer languages.

You can read through the information in these chapters to familiarize yourself with what Glossary can do for you. Do not expect to learn everything at once. Start simply, creating entries that you use on regular basis. You will find that gradually you want to do more complex things with the entries. You may want to check to see if you are at the end of the document. You may want to include a statement in an entry that does something specific when a search string is not found. Glossary can perform many complicated tasks, or it can be used simply and effectively.

Chapters 9 and 10 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 9 and 10. Functions are listed alphabetically and by usage. After you develop a working familiarity with Glossary, you can use these chapters for reference while you are writing your entries.

When new functions are introduced in entry examples, they are briefly described in the context of the program. If you would like detailed information on any function, refer to Chapter 9.

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

All of the examples in this and following chapters are functional glossary entries you can type and try. Typing and recalling some of the entries can help you understand how each entry accomplishes its task.

When you type the entries yourself, you become familar with Glossary syntax. However, you can save typing time by using the entries on the Glossary Examples Diskette provided with this book. The entries for this chapter and Chapter 6 are in **gloss2a** on the Glossary Examples Diskette.

OVERVIEW OF GLOSSARY LANGUAGE ELEMENTS

The following list provides a brief description of the elements of the glossary language presented in Chapters 6 through 8.

Statements

A statement is a declaration of purpose. A Glossary 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 entry must be declared (given a name) and initialized (given a value).

Values

Variables and functions that contain values can use those values throughout the execution of a program. For example, the value of the **word** function is the word at the cursor location in the text document. After assigning the value of the **word** function to a variable, 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 is true; if it is not at the top of the page, the value is 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 entries.

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 language has a built-in library of functions that can detect the location of the cursor, 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 puts the word "hello" (expression 3) on the document editing 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}
```

tests to see whether or not the cursor is at the end of the document. If it is, the cursor is sent to page 1 by the following statement:

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

The glossary entry ends:

```
{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 entry call another glossary entry (in the same glossary) 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" causes the entry 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 (combine) 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 *Fortune:Word Records Processing User's Guide* provides many 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 language, you can write long and complex entries. Long entries are difficult to read unless you follow a specific style or convention. Glossary is a free-form language. The style you use is not important as long as your syntax is correct. However, using style conventions can assist you in writing, understanding, and reviewing your entries.

Entry a, An Example of Programming Style

The syntax of entry a in the following example is correct, but the logical execution of the entry 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 entry 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 entry when Returns, Tabs, and comments are added.

```
entry a
{
  title = word        /*read the word at the cursor location*/
  command note        /*mark the cursor position in the document*/
  goto "w"            /*go to the workpage*/
  goto down           /*go to the bottom of the workpage*/
  call feed(title)    /*insert the word saved in the variable*/
  goto note           /*go back to the bookmark cursor position*/
}
```

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

In entry a, the **word** function assigns the word at the cursor location in the text document to the entry 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 were being typed from the keyboard; **call** precedes a function when the function is used as a statement.

All of the functions shown in entry a are described more completely in the following chapters.

Programming Style Conventions

The style recommended here is the standard C language style that is adapted to Glossary.

Comments

Descriptive, well-placed comments make complex entries much easier to understand. Even if you are not familiar with all the functions used in entry a, you can 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 an entry can wrap for several lines. The commented paragraph must start with the symbols /* (slash and asterisk), and end with the symbols */ (asterisk and slash). When an explanatory comment following an entry statement requires two lines, use an Indent symbol 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 not to accidentally include any entry statements within the comment symbols. The compiler does not tell you if a symbol has been omitted and some very unpredictable results may occur when you recall the entry.

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 an entry 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 the sequence of entry 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 entry stops.

3. The braces that surround each **if** function body. The function body for the first **if** test is the single statement **exit**, which stops the entry 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 creates steps through your glossary entry. This is helpful when you write the entry, but more helpful at a later time 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 5-1 shows how braces are used with function bodies. The figure uses the **if else** function.

This function is described in Chapter 7. Dots (...) represent omitted statements.

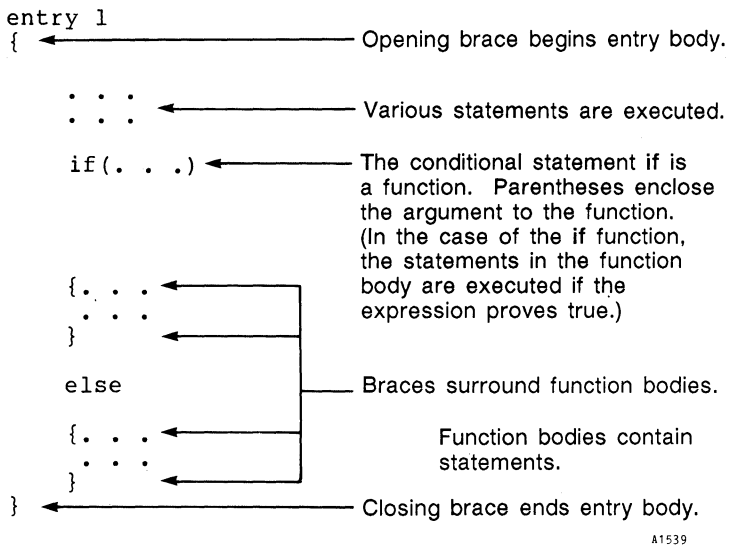

entry 1
{
. . .
. . .
if(. . .)
{. . .
. . .
}
else
{. . .
. . .
}
}
Opening brace begins entry body.
Various statements are executed.
The conditional statement if is a function. Parentheses enclose the argument to the function. (In the case of the if function, the statements in the function body are executed if the expression proves true.)
Braces surround function bodies.
Function bodies contain statements.
Closing brace ends entry body.
A1539

Figure 5-1. Using Braces to Enclose Function Bodies

SYNTAX

Syntax is the order in which the glossary language must be written. This chapter tells you about general syntax usage. Chapter 9 gives you the specific syntax required for each function.

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

```
entry c
{
  insert "Fortune Systems Corporation" execute
}
```

To insert a text string, you must first invoke insert mode by pressing the Insert key. After typing the string, press the Execute key to end the insert mode. You must follow the same sequence when you type keywords and strings in an entry. The entry would not work correctly if you wrote it in a different syntax, such as the one shown in entry C.

```
entry C
{
  "Fortune Systems Corporation" execute insert
}
```

Type both entry c and entry C in a glossary. 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 5-2 shows the standard function syntax for arguments and expressions, using the **prompt** function as an example. Chapter 6 discusses arguments and expressions.

Another type of syntax structure is shown in Figure 5-3, which illustrates the syntax for a conditional statement. Conditional statements are discussed in Chapters 7 and 8.

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

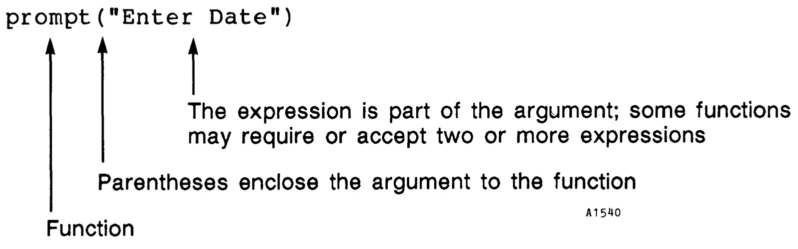

prompt("Enter Date")
The expression is part of the argument; some functions may require or accept two or more expressions
Parentheses enclose the argument to the function
Function
A1540

Figure 5-2. The Syntax for Functions, Arguments, and Expressions

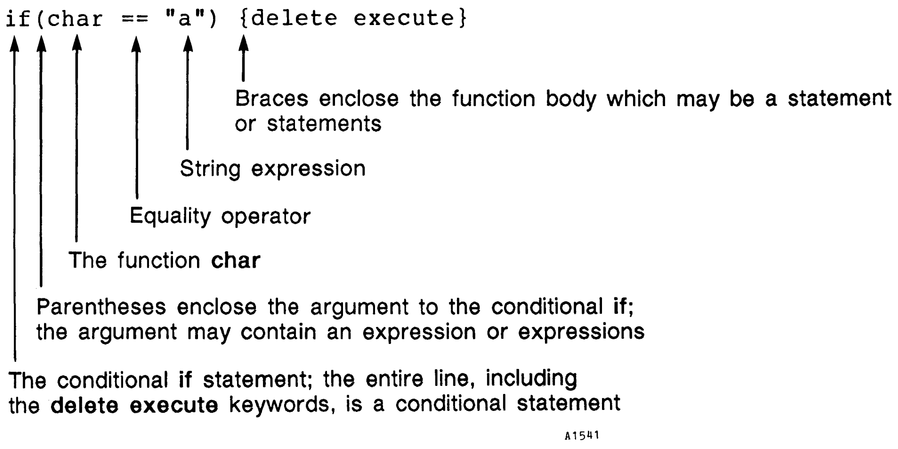

if(char == "a") {delete execute}
Braces enclose the function body which may be a statement or statements
String expression
Equality operator
The function **char**
Parentheses enclose the argument to the conditional **if**; the argument may contain an expression or expressions
The conditional **if** statement; the entire line, including the **delete execute** keywords, is a conditional statement
A1541

Figure 5-3. The Syntax for a Conditional Statement

Chapter 6
Elements of the Glossary Language

This chapter describes the following glossary elements:

- Statements
- Variables
- Values
- Logical Values
- Operators
- Functions, Arguments, and Expressions
- Parentheses

The labeled entries in this chapter are example glossary entries you can type in a glossary and recall in a text document. As you learn how to use each language element, try incorporating it into some of your earlier glossary entries or writing new entries using the examples 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 Examples Diskette provided with this book. Chapter 13 provides an index of the entries on the Glossary Examples Diskette.

STATEMENTS

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

Types of Statements

Table 6-1 shows several types of glossary statements.

Table 6-1. 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

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

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

- The first conditional **if** 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")
```

(**entry d** continued on next page)

(entry d continued)

```
          {
                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 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 in the text document must begin with a character other than "o" for the entry to work properly.

The **globerr** function is used in entry d to perform a graceful exit from the entry 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 Statement."

Statement Execution Order

Statements in a glossary entry 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 with **if** and **jump**. Chapters 7 and 8 describe how to control the execution sequence of your entries.

VARIABLES

An important feature of Glossary is the ability to store a value and recall it as a constant or change it as the entry 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 use must be declared and initialized in your entry 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 some other value the first time you use it in your entry. Each time you use the entry, or at each iteration of an entry loop, the variable is reset to its initial value.

You can initialize all variables at the beginning of your entry or immediately prior to their use. Look at the entries in this book for examples of where 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 entry during execution and allows you to enter data from the keyboard.)

The variables **ourcost** 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. Assignment and other types of operators are described later 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 an entry is not strictly necessary, but it is a good habit to acquire and can be very important when you use programming languages other than Glossary.

Variable Names

Variable names may have as many characters as you want. Using short names, however, keeps your entry concise.

The rules for variable names are as follows:

- A variable name cannot be the same as a glossary reserved word. 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 like **end_doc** or **top_page**, which are joined by an underbar. Spaces in any form are not allowed as 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 or other symbol.

- The variable name may consist of any combination of uppercase and/or lowercase letters and the numbers 1 through 9. The only symbols that may be used are the underbar (_), the period (.), and 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. The word "returned" means that a function reads information specified by the function and uses that information in the way specified by the glossary entry. The value can be stored in a variable to be used at a later time in the glossary, or used to test for a specific condition. Returned values are not entered directly into the document as text unless you provide specific instructions in the entry to do so. During entry 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 described later in this chapter.

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 6-1.

It is useful to remember the "right-hand, left-hand" definition. Mathematical assignment operators (described later in this chapter) depend on the "right-hand side, left-hand side" assignment principle.

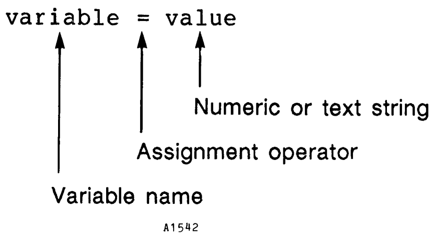


Figure 6-1. The Variable Assignment Syntax

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, 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 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 when single quotes are used.

- Octal code numbers embedded in strings must be preceded by a backslash “\.”

- Each keyword abbreviation symbol counts as one character in a quoted string. The string “\cFortune Systems Corporation\r” has 29 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. Other considerations in string character counts include the use of octal numbers and Fortune:Word 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 an alphabetic string, not a numeric string.

LOGICAL VALUES

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

Entry f is a glossary entry 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 entry to run faster.

```
entry f
{
    call display(false)                  /*turn display off*/
    "\cMEMORANDUM"                       /*center heading*/
       return(2)
    "DATE: " call feed(date)             /*types system date and time*/
       return(2)
    "TO: John Brown"                     /*types memo to line*/
       return(2)
    "FROM: Helen Smith"                  /*types memo from line*/
       return(2)
    "SUBJECT: MEETING ON
         WEDNESDAY"                      /*types subject*/
       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 editing 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 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:  "
```

(**entry g** continued on next page)

(entry g continued)

```
   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 9 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 9 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 reenter 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 entered is 1 (true), the "Correct" message is printed. If the answer entered 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 or operator (**|**). 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 described in Chapter 7. Equality and logical operators are described 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, is true if the cursor is on the first character in the document and false if it is not. You can use these values by assigning **beg_doc** to a variable, as shown 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) is true. A zero value in a logical interpretation is always false.

OPERATORS

Operators are 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 shown in Figure 6-2. Operand means "that which is operated on," and can be a variable, a function, or an expression.

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 example:

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

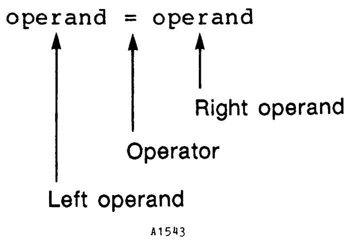


Figure 6-2. The Syntax for Binary Operators

The logical not (!) performs logical negative operations. If the cursor is not at the bottom of the page, the statements represented by {...} are performed. Normally, the function **bot_page** is 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 (!); 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 shown 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 assignment 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 in the section “Mathematical Assignment Operators.”

Mathematical Operators

Mathematical operators can be used to perform addition, subtraction, multiplication, and division in your entries. 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 entries j and k.

```
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, the variables **top** and **bottom** can be used in a mathematical expression such as the one shown in entry k. Note that numeric values can appear on either side of a mathematical operator.

Parentheses are used to ensure that the value of **bottom** is subtracted from **top** before division occurs. You can generally follow standard mathematical principles for parentheses when you write glossary entries to perform arithmetic. The section "Using Parentheses" in this chapter provides more information on parenthetical syntax.

The Modulo Operator

The remainder of a division operation can be determined with the modulo **%** remainder operator, as shown in this example:

```
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 1 is used to number figure illustrations in a document. First, it searches the text document for the string "=Figure ". The equal sign is included as part of the search string to specify a case-sensitive search. 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 entry to end. (The **globerr** function is described in Chapter 8.)

```
entry 1
{
  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** increased by 1 at each repeat of the search loop. The syntax for that addition is:

```
    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 then assigned to the variable on the left side of the assignment operator, so **figure_no** 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 cannot 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."

In entry m, the two uses of the assignment statements to the variable **count** produce two different results. In the first assignment, **count** is declared and initialized to 1. 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 is increased by 1 for each loop
                               repeat*/
}
```

Entry m is an endless loop. If you use it, the entry continues to write numbers in your document until you press CANCEL to stop the glossary entry. Chapter 7 shows you how to use conditional statements to break endless loops.

Other mathematical operations are shown in entries n, o and p. Remember, the numeric variable must be declared and initialized in the entry before beginning the math calculation.

```
entry n
{
  linenumber = 8
  linenumber = linenumber - 3   /*linenumber now has a value of 5*/
  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 following syntax:

```
figure_no = figure_no + 1
```

Using the mathematical assignment operator +=, the same addition to **figure_no** is achieved with fewer keystrokes, as shown in this 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)                  /*if the current value of cost is 24*/
  {
          cost -= 14              /*14 is subtracted from cost*/
     call feed(cost)              /*the current value of cost becomes*/
  }                               /*10, and is typed in the document*/
  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 the variable **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. To place commas, periods, and dollar signs in function and variable number values, which are then typed in a document, use the **pic** function. 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 quotation marks. 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 9 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 entry by using relational operators in the syntax shown in Table 6-2.

Table 6-2. 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,...)) {...}

Table 6-2. (continued)

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 6-2, 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 use a statement like entry z in an entry to verify that the entry is doing what you want it to do at a particular place in its execution.

Identifiers and **jump** statements are described in more detail 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 following the **if** statement are executed unless the condition specified by the relational operator is true. Entries B and D use the document reading functions **page_no** and **line**. The **page_no** function reads page number of the cursor location in the text document during entry execution. The **line** function reads the current cursor line number. The page number or line number is not entered in the document unless you assign it to a variable and feed it into the document.

Using Relational Operators with Alphabetic Strings

You can use all of the relational operators with alphabetic strings. Just as Glossary 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, that corresponds to the position of the character in the ASCII (American Standard Code for Information Interchange) collating sequence. Appendix C contains a table of characters in ASCII collating sequence order.

Since characters can be represented numerically, they can be ranked and collated numerically. This means you can write a glossary entry that selects 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. Try the entry by typing it in your glossary and verifying it (or use entry E in **gloss2a** on the Glossary Examples Diskette.) Then follow the instructions after the entry.

```
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
        }
}
```

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

Put the cursor on the first character of the first word on the list. Be sure your glossary is attached, then press the GL key and type your entry label.

Entry E inserts 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, and note that uppercase letters appear before lowercase letters, meaning that they have a lower numeric value. This is why the statement **if(char >= "a")** places a tab before lowercase letters.

> **NOTE**: Glossary uses the case-sensitive collating sequence from the operating system. Fortune:Word uses a case-insensitive ASCII collating sequence (where upper and lowercase letters are assigned the same value) for functions that sort such as Index Generator, Records Processing, and sort in a document. You can use the Fortune:Word case-sensitive sort in glossary entries by using the appropriate keywords to select and sort text. See the *Fortune:Word Reference Guide* for information about sorting in a document.

The functions in entry E are the **char**, **end_doc**, and **exit** functions. The **char** function reads the character at the cursor location in the document. The **end_doc** function is 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. When the value is false, the **exit** statement stops the entry immediately. Any statements following the **exit** statement are not executed.

Entry E uses page W as a place to store items during entry 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 an entry.

When you write an entry like entry E, carefully consider the cursor position in the document at each step of the entry execution. If you encounter any bugs during execution, print a copy of the entry. Walk through the entry by performing the steps from the keyboard. You can quickly spot places where the cursor is in the wrong place. Walking through an entry in this fashion is useful when troubleshooting a problem 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 the **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. Since the "1" in "10" has a lower ASCII equivalent than the "J" in "June," the result is that **this.month** compares less than **month**, even though month 10 (October) comes after June.

When performing comparisons, be sure you really want to compare an alphabetic value to a numeric value. Also remember that the ASCII collating sequence does not compare numbers in true numeric order. It ranks the numbers in order according to the first digit. Even if, in entry G, you change the value of **month** to 6 (June), to provide an accurate value comparison, October still considers June (6) to come after October (10) because the "1" in "10" has a lower ASCII equivalent than 6.

```
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)
```

(**entry G** continued on next page)

(entry G continued)

```
        }
        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 to or not equal to the value on the left; == means "equal to" and != means "not equal to." Entry H and entry I show examples of equality operators.

```
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** statement is 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 entry execution, logical operators depend on whether a value is true or false. True is evaluated numerically as 1 (one), and false is evaluated as 0 (zero).

The Logical and (&) Operator

There is only one possible true condition for the & operator: expressions on either side of the & operator must be true.

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 recipe. 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.

In entry J, 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. The statement **{ingredients = apples + bananas}** is only executed if both subexpressions are true. You can state this logically by saying, "If apples == 6 and bananas == 2, then the expression is true, so execute the following statements (add apples to bananas and store the result in ingredients)."

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. For more information on the use of parentheses, refer to the section "Using Parentheses" later in this chapter.

The Logical or (|) Operator

There are three possible true conditions and one false condition for the | operator.

The three true conditions are as follows:

- Both statements are true
- The statement to the left of the operator is true, the statement to the right is false
- The statement to the left of the operator is false, the statement to the right is true

The one false condition is when all statements are false.

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: " cal: 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 recipe.  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 is true and executes the statement **{ingredients = apples + bananas}**.

- 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}** is not executed is when apples does not == 6 and bananas does not == 2. In this case the full expression is false, so the statement in braces is ignored and the entry skips 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 either 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 Logical not (!) Operator

You have already had an introduction to the logical not operator (!) 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. The following examples show its use:

```
    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 **{...}**.

Entries 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 or 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 "
```

(**entry** L continued on next page)

(entry L continued)

```
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

Tables 6-3, 6-4, 6-5, and 6-6 summarize all the operators you can use in your entries.

Table 6-3. 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 6-4. 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 6-5. 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 6-6. Mathematical Operators

Operator	Definition	Syntax Example	Explanation
+	Plus (performs addition)	inventory = 27 + 114	Add 27 and 114 and assign a value of 141 to **inventory** (141 is the sum of 27 and 114).
-	Minus (performs subtraction)	stock = inventory - sales	Subtract the value in **sales** from the value in **inventory** and assign the result to **stock**.
*	Multiply (performs multiplication)	forcast = avgsales * 12	Multiply the value in **avgsales** by 12 and assign the result to **forcast**.

Table 6-6. (continued)

Operator	Definition	Syntax Example	Explanation
/	Divide (performs division)	avgsales = sales83 / 12	Divide the value in **sales83** by 12 and assign the result to **avgsales**.
%	Modulo	sales.remainder = sales83 % 12	Divide the value in **sales83** by 12 and assign the remainder to **sales.remainder**.
+=	Addition assignment operator	personnel += 4	Add 4 to the value in **personnel** and assign the result to **personnel**.
-=	Subtraction assignment operator	personnel -= 2	Subtract 2 from the value in **personnel** and assign the result to **personnel**.
*=	Multiplication assignment operator	expenses *= 12	Multiply the value in **expenses** by 12 and assign the result to **expenses**.
/=	Division assignment operator	expenses /= 12	Divide the value in **expenses** by 12 and assign the result to **expenses**.
%=	Modulo (division remainder) assignment operator	expenses %= 6	Divide the value in **expenses** by 6, and assign the remainder 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 use to perform standard operations in your glossary entries. Chapter 9 provides you with an alphabetical list of each function, including its use and syntax. Chapter 10 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 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.**

Functions Operate on Data

Functions can gather and return data to the glossary entry during its execution. This data is the value in the function. For example, when the **date** function is called in your entry, it reads the system date as its value.

```
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 entry.

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 functions that perform operations are **error** and **posmsg**. The **error** function displays a message in the error section of your screen. (The error section 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 a location you specify.

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.

Expressions

Values inside the parenthetical argument are expressions. Some functions can take several expressions, as in the examples for logical operators shown earlier. 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 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 9 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.

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

When the **cursor** function is called in a glossary entry, 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 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. The variable 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 of the glossary language is the ability to use expressions with keywords that take an argument. You can move the cursor up 12 lines using the following syntax:

```
up(12)
```

And you can type the tab symbol in your document four times using the following syntax:

```
tab(4)
```

You can also give an expression to the keyword, and it can be a variable. For example you can move the cursor up 12 lines using the following syntax:

```
moveup = 12
up(moveup)
```

When you recall the glossary entry, the result is exactly the same as if you wrote the expression as **up(12)**.

You can use a mathematical expression as an argument for a variable, or an expression to the keyword, as shown in the examples below:

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

up(3 * 3)
```

The examples in this book illustrate some ways you can use this feature. If you like to experiment, try using expressions with keywords in other combinations. 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 6-3.

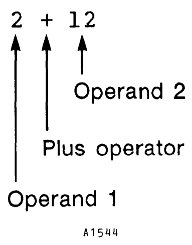


Figure 6-3. The Syntax of a Mathematical Expression

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

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

If you add another expression you 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*/
```

Multiplication has a higher precedence order than addition and is performed first. If the addition expression were not enclosed in parentheses in the example above, the result would be quite different. In the example below, 12 and 14 are multiplied first, and then 2 is added to the result.

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

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

The fully parenthesized form looks 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 in Table 6-7 in order from highest to 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 6-7. 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 those shown below, you do not need to enclose the subexpressions in parentheses.

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

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

Of course, if you add more complicated math to the first expression, you 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 need to use parentheses with most logical expressions using the & and | operators because they usually contain several subexpressions, as shown in the following examples.

Sometimes the syntax of a function requires parentheses around logical expressions:

```
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.

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

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

TROUBLESHOOTING

When you begin to use the Glossary elements presented in this chapter, you may receive more verification errors than usual, or your entries may not execute the way you think they should. Some points to help you troubleshoot your entries 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, but does not check for logic errors.
- Be sure you have used parentheses properly.
- Logical operators can be tricky to use. If you are not familiar with the principles of Boolean algebra, refer to a book on the fundamentals.
- 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.

Chapter 7

Conditional Statements

Statements in a glossary entry are always executed from the top to the bottom 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 used in examples in this chapter and described in detail in Chapter 8.

- Control statements such as **jump** and **call** transfer execution control to a different part of the entry or to another entry. The **globerr** function allows you to exercise control by setting a trap for keyword function error conditions. You can gracefully stop an entry at any point in its execution by using the **exit** statement. These statements are described in Chapter 8.

Correct placement of conditional and control statements within an entry is essential. Write a few entries of your own using these functions. The glossary compiler notes syntax errors; it does not check your logic. If an entry does not work the first time, your conditional tests or loop instructions may be in the wrong place. Shift them around and try again.

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 useful to help distinguish between conditional statements and other statements in the entry.

The four conditional functions and their syntax structures are shown below. 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 decision makers. Decisions are based on the evaluation of three types of conditions:

- Conditions in the text document during execution
- Conditions arising from interactive operator input during execution
- Conditions in the entry 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.

Evaluating Conditions in the Text Document

When you use a conditional statement to evaluate document conditions, you are evaluating the cursor position in the document. You use a "document reading function" (discussed in Chapter 10) to ask 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 document reading functions read number values from the document status line. The **line** function, for example, reads the line number of the current cursor location. Your entry 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. Some document reading functions use this information when an entry is executed in the document.

Other document reading functions, such as **beg_doc** and **left_margin**, test for true or false. If someone asks you if your cursor is at the beginning of the document while you are editing, you respond "yes" (true) if it is and "no" (false) if it is not. The **beg_doc** function makes this same evaluation while the entry 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 **counter** by 1 as long as the cursor is not at the end of the document. The logical not operator (!) changes **end_doc** so that the function is true if the cursor is not at the end of the document. Normally, **end_doc** is 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 entry execution.

Evaluating Interactive Operator Input

The interactive functions **key**, **keys**, **keyin**, and **keysin** allow you to enter data as the entry 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 also 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)
```

When you need to evaluate interactive data with a conditional statement, **key** and **keys** are the most direct functions to use.

When you execute an entry that uses the **keyin** and **keysin** functions, the data you type is entered directly into the document and is not stored in a variable.

The interactive method you choose for your entry depends on the result you want to achieve.

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

Evaluating Conditions in an Entry

Values in variables are usually the internal entry 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 entry can be written to continually evaluate a variable by using a conditional statement.

For example, entry A 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. When the variable **count** reaches 80, the entry stops.

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

Entry c in this chapter is an example of evaluating the same variable for two different entry conditions.

Conditional Statements Can Change Execution Order

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

Evaluating Fortune:Word Screen Symbols

One of the document conditions you may want to evaluate is whether or not the cursor is on a screen symbol, such as a Return, Tab, or Center symbol. Asking these questions with a conditional statement involves using both Fortune:Word 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 as follows:

```
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 entry 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 entry stops. 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 must add a loop. Chapter 8 describes how to rewrite this entry using a conditional **while** loop.

```
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(1,50,31)
   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 entry execution. This entry interactively types an invoice in the text document.

In entry b, the conditional **if** statement is used 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 described in Chapter 8.

If the operator enters an **n** or **N**, the entry stops.

```
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(1,50,31)
      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(1,50,31)
      return(2)
      page
      goto north

      call prompt("Invoice? Type y or n: ")
      answer = keys
      call clrpos(1,50,31)
```

(**entry b** continued on next page)

(entry b continued)

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

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

When you use this entry, note the difference between the **keysin** function and the **keys** function. 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.

> NOTE: When you use the **keysin** function following an **insert**, pressing EXECUTE to conclude **keysin** also ends the insert. To continue inserting text from the glossary entry after a **keysin**, precede the text in the glossary entry with the keyword **insert**.

The **keys** function also allows entry of unlimited keystrokes; however, the data entered is stored in a variable, and is 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 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 described in Chapter 8)

Evaluating Conditions in the Entry

Entry c is an example of an **if** test that evaluates conditions in the entry during execution. This entry types the numbers 1 through 50, each on a separate line. The numbers 1 through 9 are preceded by a zero (0), as in 01, 02, 03, and so on. 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
}
```

Entry c can be modified for typing line numbers before existing text lines by adding the keywords **insert** and **execute**. This can be very helpful when typing legal documents that require line numbers. Try writing another entry 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 fifty 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 Entries

Figure 7-1 shows a typical flow chart used to diagram an entry using the conditional **if** statement. Flow charts are a device you can use for planning your entries. You do not need to be formal with them; just sketch your ideas on a piece of scratch paper. Use squares for the action parts of the entry and diamonds for the conditional decision-making sections. Draw lines to indicate the flow of execution through the entry. Using flow charts to analyze your logic can make the actual entry-writing easier.

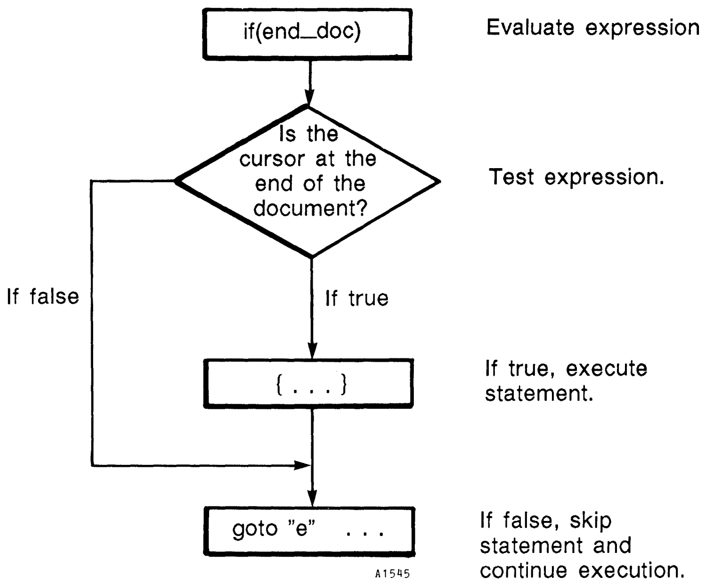

if(end_doc)
Evaluate expression.
Is the cursor at the end of the document?
Test expression.
If false
If true
{ . . . }
If true, execute statement.
goto "e" . . .
If false, skip statement and continue execution.
A1545

*Figure 7-1. Flow Chart Using the Conditional **if** Statement*

THE CONDITIONAL IF ELSE STATEMENT

The **else** statement gives you an alternative statement to use 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 cannot 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 as follows:

```
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 7-2 shows a sample flow chart for the **if else** statement.

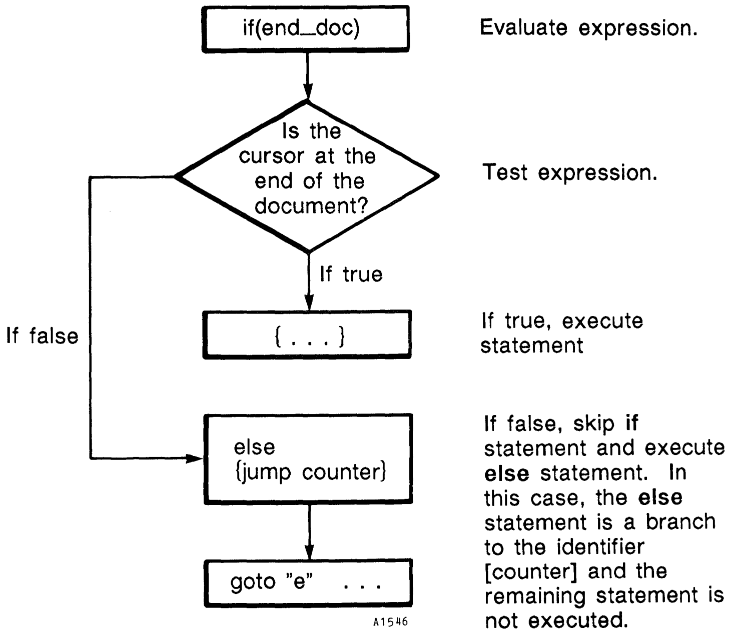

if(end_doc)
Evaluate expression.
Is the cursor at the end of the document?
Test expression.
If true
If false
{. . .}
If true, execute statement
else {jump counter}
If false, skip **if** statement and execute **else** statement. In this case, the **else** statement is a branch to the identifier [counter] and the remaining statement is not executed.
goto "e" . . .
A1546

Figure 7-2. Flow Chart Using the Conditional if else Statement

USING FLAGS IN AN ENTRY

A flag is a symbol you put in a document for a specific purpose. The two most common purposes are:

- To provide a unique search string for an entry. For example, the Table of Contents Generator searches for the symbol combination MERGE NOTE MERGE to extract headings. When you use a flag in a glossary entry, choose 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 is not generally used in a text document.
- To provide a signal to stop the entry. When a flag is encountered, the entry stops. This type of flag is typically used with a conditional statement.

  NOTE: Merge symbols are used in Records Processing List and Format documents, so choose another flag symbol if you are 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. To use this entry, the document must be prepared in a specific way. When the document is first typed, the operator puts a <fx flag before each figure heading and a <tx before each table heading. A glossary entry can be used to do this. In case the document contains other flags that use the Merge symbol, entry d uses an **else** statement to ignore a non-flag merge symbol. Entry d could be used in different documents. When you write an entry like this, it is always a good idea to provide a trap for other uses of the flag character in the document.

```
entry d
{
  figureno = 0
  tableno = 0

[searchloop]

  search "<" execute
     if(globerr)
     {
           execute exit
     }
  cancel
```

(**entry d** continued on next page)

(**entry d** continued)

```
  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 stop the entry. When the entry finds a <, it moves one character to the right. If the character is an "f" or a "t," the entry 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 or 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 Entry Runtime

The application performed by entry d could be written more simply as 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 requires 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 an entry. Both styles perform the same way when the entry is executed. Whichever style you choose, be consistent throughout your entry.

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

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

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

- The following nested statement is called an **else if** structure. If it seems to be a clearer way of nesting than the first style, use it.

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

Nesting your **if else** statements helps you write tighter entries 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
```

(**entry e** continued on next page)

(entry e continued)

```
  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 7-3 shows a sample flow chart for nested **if else** statements.

TROUBLESHOOTING

Remember, when you use a conditional statement to evaluate conditions in the text document during entry execution, you are evaluating the cursor location. If your entry is not working properly, check the cursor location. A good way to check your entry is to edit the glossary, 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.

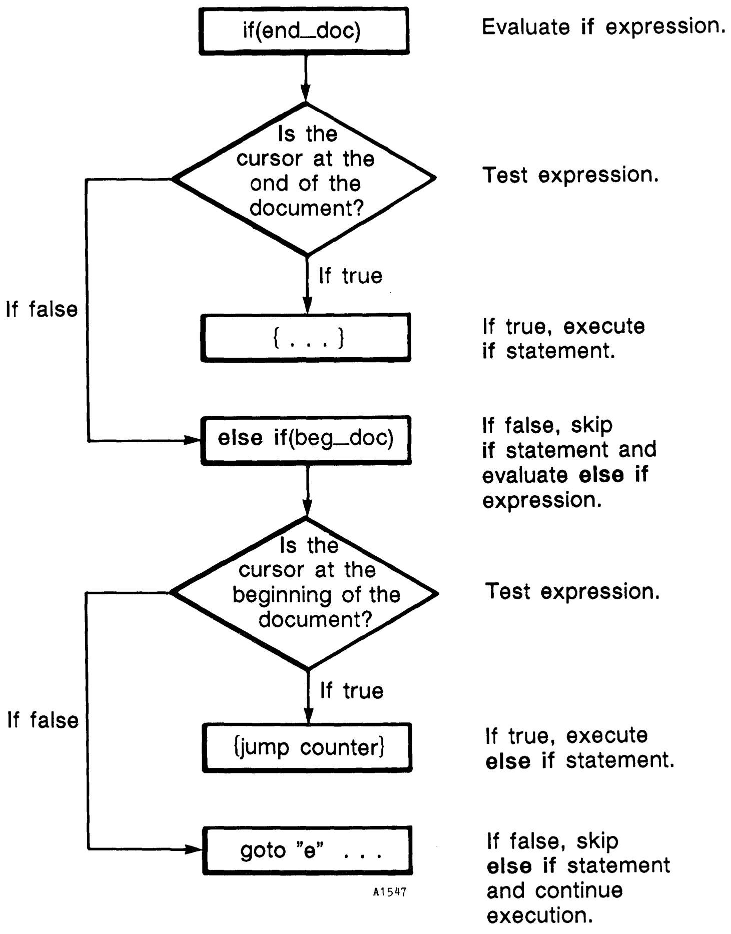

if(end_doc)
Evaluate **if** expression.
Is the cursor at the ond of the document?
Test expression.
If false
If true
{ . . . }
If true, execute **if** statement.
else if(beg_doc)
If false, skip **if** statement and evaluate **else if** expression.
Is the cursor at the beginning of the document?
Test expression.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

*Figure 7-3. Flow Chart Using Nested **if** and **if else** Statements*

Chapter 8
Control Statements

Conditional statements make decisions in your entries. Based on those decisions, control statements can transfer execution control to another part of your glossary entry or 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 an entry.

Glossary control statements include the following:

call

glossary

jump

exit

globerr

The **call** function and **glossary** 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. The **exit** function stops 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 entries. They can be used two ways:

- As dependent entries that can only be used for a specific calling entry.

- As independent entries 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 as follows:

call function(expression)

call entry label

The **call** function was used in many previous examples. 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

The **call** statement might look like these examples:

call a   call x   call B   call F

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

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 entry execution continues at the statement immediately after the subroutine call (unless directed elsewhere by the subroutine).

Figure 8-1 illustrates the flow of statement execution between the calling entry and the subroutine.

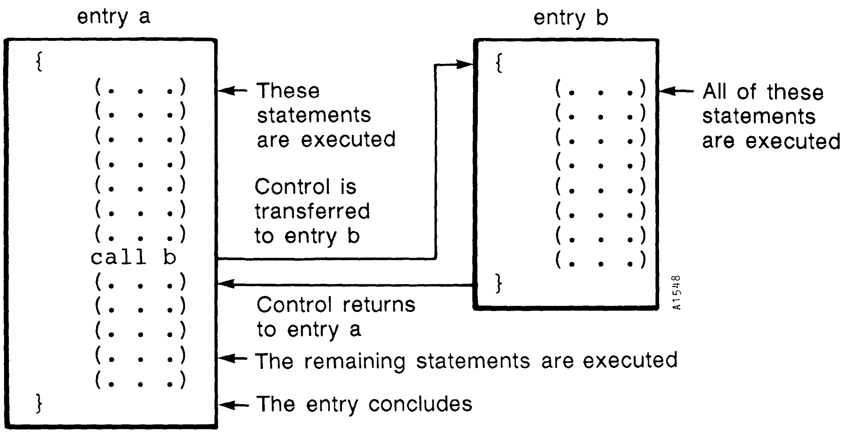

entry a
{
(. . .)
call b
}
These statements are executed
Control is transferred to entry b
Control returns to entry a
The remaining statements are executed
The entry concludes
entry b
All of these statements are executed
A1548

Figure 8-1. Calling a Glossary Subroutine Using the ***call*** *Function*

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 10 shows you how to use the **substr** function to change the value returned by **date** to the format **July 14, 1987.**

```
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)
```

(**entry** f continued on next page)

(entry f continued)

```
  tab "We are pleased you are considering us as 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(1,50,31)

      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.

```
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 AMALGAMATED 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 an entry is the same as pressing the GL key from the keyboard. When you run the entry, the prompt *Which entry?* is displayed. You can include the entry label in two ways:

- By including the entry label as part of the entry when you write it
- By using the **keyin** or **keysin** function as part of the entry. When you recall the entry, the entry label is entered interactively from the keyboard while the entry is running.

In both instances, both the calling entry and the called entry must be in the same glossary document.

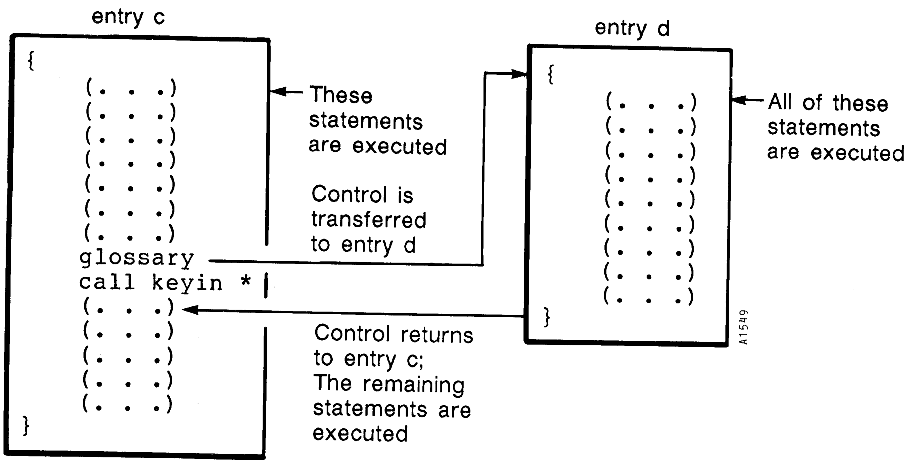
You can use the keyword **glossary** to temporarily transfer control from one entry to another entry with any one of the following statements. When you use the **glossary** statement, you must always enclose the entry label in quotes.

```
glossary "label"
glossary call keyin
glossary call keysin
```

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. The operator must press EXECUTE to continue with the glossary entry.

Figure 8-2 illustrates the glossary **call keyin** statement.

Entry g uses the interactive statement **glossary call keysin** to select a glossary subroutine. Entries O and P are examples you can use as subroutines with this entry. The sales figures could be an entry containing a lengthy sales report which 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.



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

*Figure 8-2. Interactively Calling a Glossary Entry as a Subroutine Using the **glossary** Statement*

```
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 your sales quota last
  month. You will be crowned WIDGET KING OF THE MONTH at this
  month's Amalgamated 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 separately."
  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. Execution resumes at the statement immediately after the **call** statement.

Entry h gives you an example of nested subroutines. Try it to get an idea of what the flow is like through the subroutine structure.

```
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, it's 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 8-3 illustrates four nested subroutines.

BRANCHING

Execution control can be unconditionally transferred to a different part of an entry or to another entry in the same glossary. 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.

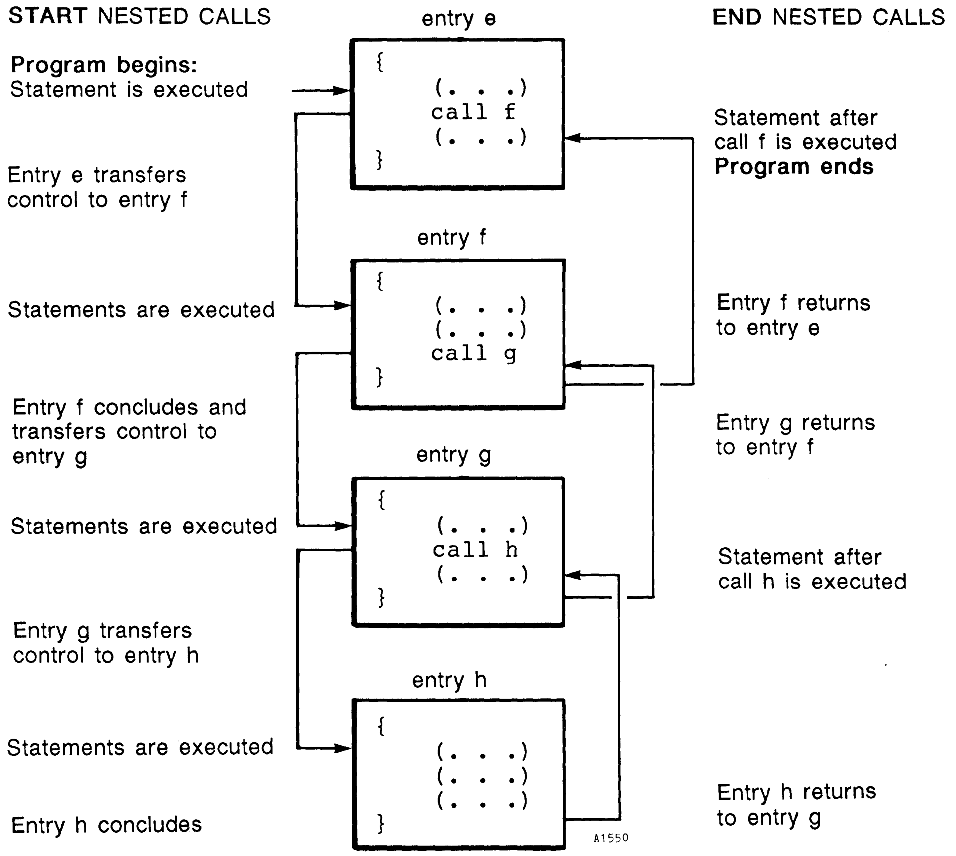

START NESTED CALLS
entry e
END NESTED CALLS
Program begins:
Statement is executed
{ (. . .) call f (. . .) }
Statement after call f is executed
Program ends
Entry e transfers control to entry f
entry f
Statements are executed
{ (. . .) (. . .) call g }
Entry f returns to entry e
Entry f concludes and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

*Figure 8-3. Nested Subroutine Calls Using the **call** Function*

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
}
(. . .)
[typemore]
(. . .)
```

```
[goagain]
(. . .)
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 as follows:

```
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 must follow 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.

Figure 8-4 illustrates the execution flow for jumps.

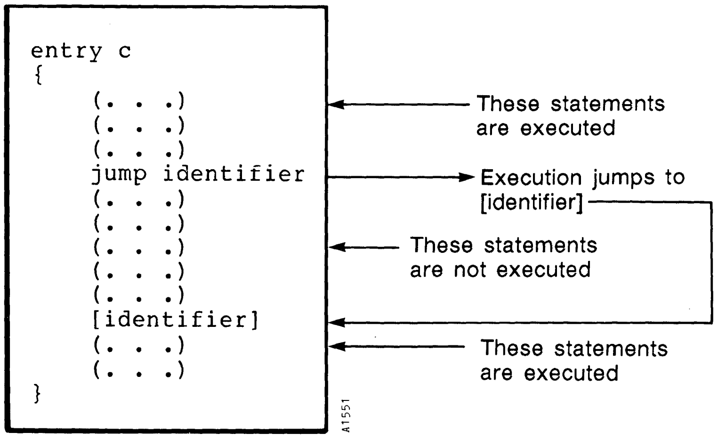

entry c
{
(. . .)
(. . .)
(. . .)
jump identifier
(. . .)
(. . .)
(. . .)
(. . .)
(. . .)
[identifier]
(. . .)
(. . .)
}
These statements are executed
Execution jumps to [identifier]
These statements are not executed
These statements are executed
A1551

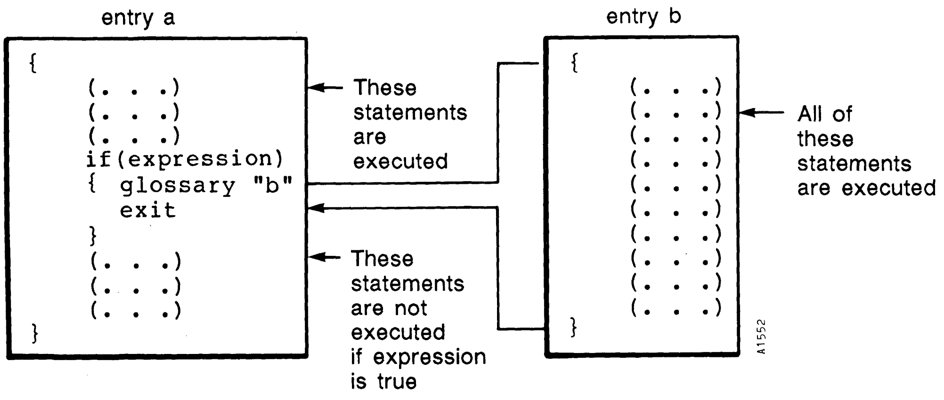
*Figure 8-4. Branching Within the Same Entry Using the **jump** Statement*

Using a Glossary or Call Statement to Branch

Control is temporarily transferred from one entry to another with the **call** function or **glossary** statement. These statements can be used as a subroutine or function call. Control is returned to the main entry after a **call** or **glossary** statement. If you want the entry to branch, place an **exit** statement after the **glossary** or **call** statement. When the called entry returns control to the calling entry, it reads the **exit** statement and stops (see Figure 8-5). Another way to branch is to put the **call** statement at the end of the main entry (see Figure 8-6).

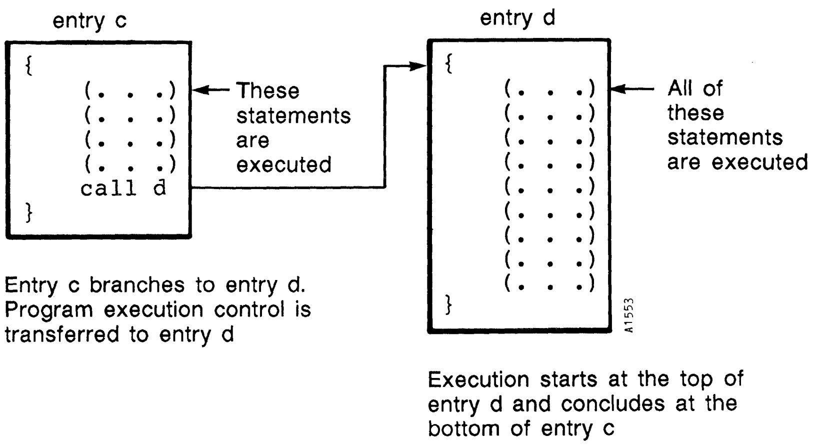
In Figure 8-5, entry a executes its statements. If the conditional statement is true, execution resumes at the top of entry b. When control returns to entry a, the **exit** statement stops the entry, ending the branch. If the conditional statement is false, entry b is not executed, and the statements following the conditional statement are executed.

In Figure 8-6 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 ends.


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
A1552

Entry a branches 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 8-5. Branching to Another Entry Using the **glossary** Statement*


entry c
{
(. . .)
(. . .)
(. . .)
(. . .)
call d
}
These statements are executed
Entry c branches to entry d. Program execution control is transferred to entry d
entry d
{
(. . .)
(. . .)
(. . .)
(. . .)
(. . .)
(. . .)
(. . .)
(. . .)
(. . .)
}
All of these statements are executed
A1553
Execution starts 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.)

*Figure 8-6. Branching to Another Entry Using the **call** Function*

LOOPING

When you write an entry without a loop instruction, it executes once straight through its statements. When you add a loop to an entry, it repeats itself over and over again. You stop this repetition by the strategic placement of a conditional expression.

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. There are many ways to use loops. They can count pages or lines in the document for you, they can increment variables, or they can repeatedly search for a string in the document. Your specific application dictate when and which types of loops you need.

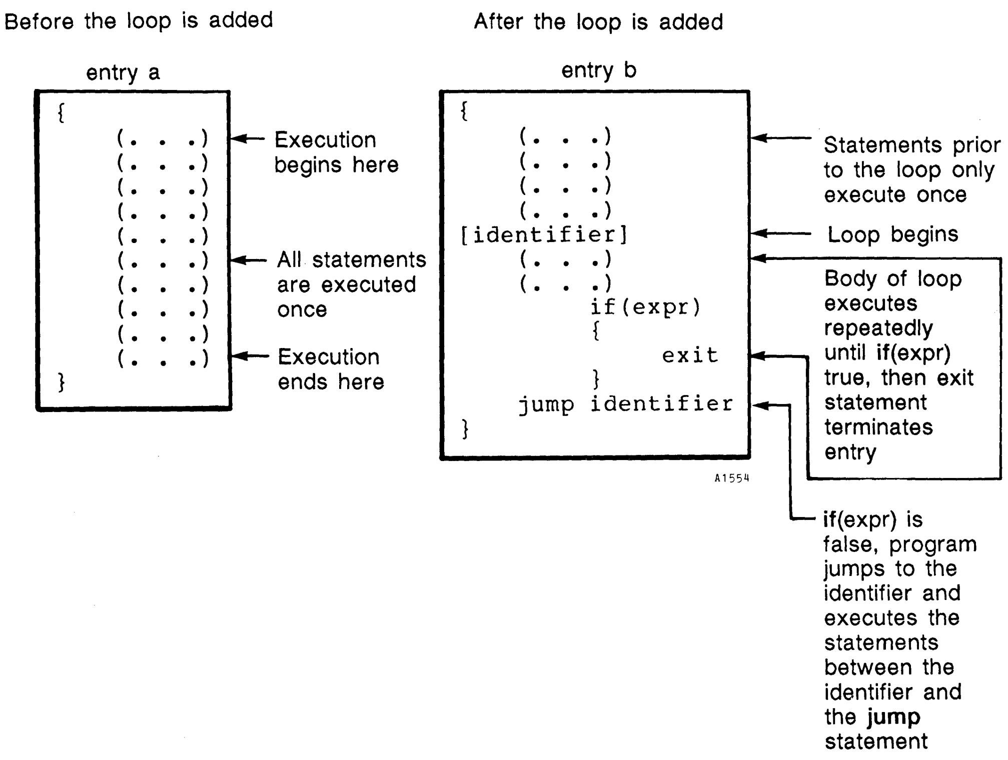
Using the Jump Statement for Loops

The **jump** statement can be used to perform loops by placing an identifier at the top of the loop and a **jump** statement at the bottom, specifying the beginning and end of the loop. 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. When you forget to put the identifier in your entry, the glossary compiler reminds 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 repeats itself indefinitely. The conditional statement may direct entry execution elsewhere by branching, calling a subroutine, or causing the entry to stop 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 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 entry execution. The results may be totally unpredictable. You can use the **if** or **if else** conditional statements to break jump loops.

Figure 8-7 shows the entry execution flow when the **jump** statement is used to loop.



Before the loop is added
After the loop is added
entry a
{
(. . .)
(. . .)
(. . .)
(. . .)
(. . .)
(. . .)
(. . .)
(. . .)
(. . .)
(. . .)
}
Execution begins here
All statements are executed once
Execution ends here
entry b
{
(. . .)
(. . .)
(. . .)
(. . .)
[identifier]
(. . .)
(. . .)
if(expr)
{
exit
}
jump identifier
}
Statements prior to the loop only execute once
Loop begins
Body of loop executes repeatedly until if(expr) true, then exit statement terminates entry
if(expr) is false, program jumps to the identifier and executes the statements between the identifier and the jump statement
A1554

*Figure 8-7. Using the **jump** Statement to Perform a Loop*

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 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 symbol to begin its calculations. If you do not want to take the time to type the table, copy it from Page N of **gloss2b** on the Glossary Examples Diskette.

Second, type entry k in a glossary document. (Or attach **gloss2b** and recall entry k.)

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
{
call finsert(time) return(2)

  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. 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 entry 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 stops.

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** and **finsert** functions are new to you in this entry. The **number** function is a document reading function that reads 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. The **finsert** function is used to insert the information provided by a function such as **time**, or to insert text stored in a variable. See Chapter 9 for a more detailed description of the **finsert** function.

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 in the next section 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 improvement 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 entry. The entry 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(...))**.

```
entry 1
{
  call finsert(time) return(2)
  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,"$,"))
```

(entry 1 continued on next page)

(entry 1 continued)

```
            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
}
```

Using While and Do While for Loops

Two very powerful looping functions are the **while** and the **do while** 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. They accomplish the loop and at the same time provide the conditional expression for stopping the loop. Both **while** and **do while** observe the general principles of conditional statements that were described in Chapter 7.

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 is an example of this combination.

The Conditional While Statement

The syntax for the conditional **while** statement is as follows:

```
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** are never 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 if 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 entry.

Entry m is a rewrite of entry a in Chapter 7, which used a conditional **if** statement. A loop was also added to entry m 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 is executed at least once whether or not the expression to **while** is true.

The syntax for the **do while** statement is as follows:

```
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 to the "Amalgamated Widgets Parts List" example which follows entry p.

```
entry p     /*the example for use with this entry is on Page N of gloss2b*/
{
  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 is true if the value of its expression is a number. If the expression is not a number, **num** is 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 Examples Diskette.) When you recall entry p, the column is added and the total entered.

Example for use with entry p:

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. If a Decimal Tab follows TOTAL, the cursor moves past TOTAL, and the total is not entered in the document. (Entry p requires an existing Decimal Tab following TOTAL.)

```
entry q
{
  inventory = 0

  [loop]
  search decimaltab execute
      if(globerr)
      {
            execute
            search "TOTAL" execute
                  if(globerr)
                  {
                        execute
                        exit
                  }
```

(**entry q** continued on next page)

(entry q continued)

```
          cancel
          goto right
          insert
               decimaltab call feed(inventory)
          execute exit
     }
  cancel

  inventory += number

  jump loop
}
```

These two examples illustrate that there is no absolutely correct way to write a program; many methods work and work well. They also illustrate the importance of setting up the correct format in your document so that the entry runs correctly. In entry p, the presence of the Decimal Tab following TOTAL is essential for correct program execution. In entry q the Decimal Tab following TOTAL prevents correct program execution. Use the method that is easiest and most comfortable for you.

Points to Remember About Loops

Important points to consider when you construct entry loops are listed below:

- A loop keeps looping unless it is stoped 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 is 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 that a subroutine uses are declared and initialized in the calling entry outside of any loops.
- A **do while** loop always executes its statement at least once.
- A **while** loop never executes its statement if the condition starts out false.

- 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 are reinitialized at the next repeat of the loop. It is generally better to use **while**, **do while**, and **jump** statements to loop.
- When you write entries using loops, first write the entry without the loop. Recall the entry in a text document and be sure the first iteration works. Then add the loop to the entry. A runaway loop can trample 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 entries. If the entry works properly in the copy, you can make it available for general use.

STOPPING ENTRY RECALL USING THE EXIT STATEMENT

An entry ends when all its statements have been executed. When you want an entry to stop before all statements have been executed, use an **exit** statement.

The **exit** statement makes an entry cease execution. When the **exit** statement is encountered during the entry, execution stops immediately. Statements following the **exit** statement are not performed.

When you use an **exit** statement in a subroutine, it pertains only to the subroutine. It ONLY stops execution of the subroutine. The entry that called the subroutine continues at the statement immediately following the subroutine call.

Gracefully Stopping an Entry

When an entry is running, a graceful way to conclude it is to include screen messages that notify the operator that the entry is finished. Entry Z is a short subroutine that can be called by any entry in your glossary. Modify it to your taste or use it as is.

Notice that the prompts follow standard Fortune:Word message conventions (such as *Press EXECUTE to continue*). This is a principle to keep in mind when you are designing entries. 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(1,50,31)
   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 works.

TRAPPING FUNCTION ERRORS USING THE GLOBERR STATEMENT

There are several Fortune:Word functions that search for characters or symbols as part of their function. This type of function sounds a beep when it fails to find the specified object of its search. For example, a beep 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 sounds when the search function fails to find its specified string. NEXTSCRN and PREVSCRN beep if there is no next or previous screen.

As you have seen from previous examples, you can use the **globerr** function as part of a conditional statement to trap the failure of a Fortune:Word Search function and exit the entry, jump to a branch of the entry, 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. The value of **globerr** is reset to false after it is used.

TIMING YOUR ENTRIES

Entry 1 in this chapter showed you several ways to increase the response time of a glossary entry. 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 entry. Entries K and L show use the **time** function to determine the difference in execution time between the **feed** and **finsert** functions. The system time is inserted at the beginning and end of each entry as part of the text displayed on the screen. You can compare the two figures to determine how long it took to run each entry. 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 system 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
   system 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 configuration and load, entry K takes approximately five seconds longer to execute than entry L.

Chapter 9
Function Description List

This chapter contains an alphabetical reference list of all the functions in the Glossary 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
- Permissible syntax statements for the function

Some examples are provided in this chapter to help clarify the nature of the function. For additional examples, refer to the other chapters and appendices of this guide.

Chapter 10 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 9-1, using the **beg_doc** function as an example.

The following list provides a detailed description of the format shown in Figure 9-1.

Function:	Functions are listed in alphabetical order by name.
Type:	The type of function is shown. Refer to the usage list of functions in Chapter 10 for more information. The **beg_doc** function is a document reading function, so you know that it returns information from the text document as its value.

beg_doc

```
Type:     document reading

Value:    1 if true, Ø 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 9-1. Example of Function Information Format

Value: This is the type of value returned by the function. In the case of **beg_doc**, the function returns a numeric value; it 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**)." Experiment with other combinations than those shown.

Description:	This paragraph describes what the function does and how it performs. The values required for each expression in the argument are listed and explained. Brief examples are used when appropriate to clarify the action of the function. This descriptive paragraph is similar to a dictionary definition. Refer to the "Function Usage List" in Chapter 10 and in other chapters and appendices in this book for more detailed information about a function.

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 in this example:

```
function(expression1,expression2,...)
```

In some diagrams and figures, omitted entry 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 are always 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.

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 expression. 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 is true if the cursor is on the first character of the document. Otherwise, it is false. When it is preceded by the logical not operator (!), the combination !**beg_doc** is 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 is true if the cursor is on an optional page break, a required page break, or the end of document line. Otherwise, it is false. When it is preceded by the logical not operator (!), the combination **!bot_page** is true only if **bot_page** is NOT on a page break. An optional or required column break is not considered to be a page break.

---

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 another entry in the same glossary. When **call** is used to call another entry as a subroutine, the statements in the subroutine are executed, then entry 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 reads the character at the cursor position. You can use this function to test for a specific character, or to pass the character found at the cursor position to a variable. When the character is read, the cursor position does not change.

clrpos

Type: display

Value: **clrpos** does not return a value

Syntax: call **clrpos**(expression1,expression2,expression3)

The **clrpos** function displays the number of blank characters specified by expression3 at the line specified by expression1 and the character position specified by expression2. You can only clear lines 1 through 25 and positions 1 through 80. Messages that extend beyond position 80 or line 25 result in screen display anomalies. The blanks displayed by **clrpos** can be cleared at the end of glossary execution by pressing CANCEL and RETURN, CANCEL and EXECUTE, or CTRL/w. The **clrpos** function is a temporary display. It does not replace characters in the document.

cursor

Type: display

Value: **cursor** does not return a value

Syntax: call **cursor**(expression)

The **cursor** function moves the cursor to the location in the document specified by the expression. If the expression contains the string value "3,9,12," the cursor moves to page 3, line 9, position 12. If the screen position is not an open area of the screen, the cursor moves as close as possible to the location specified. The page designation may be a numbered page, or the header, footer, work, note, or footnote pages. The expression in the argument to **cursor** may 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. The **cursor** function only works correctly in the leftmost column of a document with a multiple-column format line.

---

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 "Fri May 1 09:40:00 1987."

---

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 the 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)**. A glossary entry runs faster when the screen display is turned off.

---

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 entry continues after the **while** argument. Since the test of the expressions takes place after each execution of the **do** statements, the statements are 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 is true if the cursor is on the end of document line. Otherwise, it is false. When it is preceded by the logical not operator (!), the combination !**end_doc** is true only if **end_doc** is not on the end of document line.

---

error

Type:	display
Value:	string
Syntax:	call **error**(expression)

The **error** function displays the value of expression in the error section of the screen (line 25, character locations 51 to 79). The error section of the screen automatically displays any text as bold. The **error** display is accompanied by a beep. The length of the error string cannot exceed 29 characters. Strings longer than 29 characters 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 entry (the octal representation for CTRL/w), or by invoking an editing function, such as **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 entry, execution immediately stops and statements following the **exit** statement are not performed. The **exit** statement in a subroutine pertains only to the subroutine. It causes the subroutine to stop execution. The entry that called the subroutine continues 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 is 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 is 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 mode as follows:

```
insert call feed(expression) execute
```

If you use feed in an entry for Forms Processing, be sure the characters you feed correspond to the type of the field in the form template document and that the field length is not exceeded.

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 as Fortune:Word document control codes by functions such as **char** or **text**. (Appendix C describes Fortune:Word document control codes.) Use **finsert** to insert values returned by the **text** function. (See the description of the **text** function in this chapter.) The **finsert** function should not be used in a Forms Processing entry, since it modifies the form.

---

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 is true if the preceding search or goto [symbol] glossary operation resulted in an error condition that caused a beep. For example, the search function fails to find its specified string, and the beep 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 these glossary keywords:

```
search   nextscrn   prevscrn   goto nextscrn
goto prevscrn   goto command indent   goto indent
goto center   goto dectab   goto tab
```

Use the following syntax:

```
keyword(s)  if(globerr) {...}
```

---

if

Type: conditional

Value: if does not return a value

Syntax:

```
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 entry 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 entry 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 entry 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 entry 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. 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 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 entry 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 entry 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 entry 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.

---

keyin

Type:	interactive
Value:	**keyin** does not return a value
Syntax:	call **keyin**

The **keyin** function pauses entry 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 entry execution so that any number of characters may be typed. Only standard character keys are accepted by the **keys** function. A beep sounds if a function or editing key is pressed. When the Execute or Return key is pressed by the operator, entry execution continues, and the entered string of characters is passed to the variable or function. Characters entered to **keys** 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 entry.

keysin

Type:	interactive
Value:	**keysin** does not return a value
Syntax:	call **keysin**

The **keysin** function pauses entry 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 EXECUTE is pressed by the operator. When you use **keysin** following an **insert**, pressing EXECUTE to end the **keysin** also ends the insert. When using an entry with the **keysin** function, pressing CANCEL stops the glossary entry.

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 is true if the cursor is on the first character of a line. Otherwise, it is false. When it is preceded by the logical not operator (!) the combination !**left_margin** is 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, Fortune:Word 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. Refer to Appendix C for information on counting characters in Fortune:Word document control codes. (The backslash (\) is used as an escape character for embedments and does not count as a character unless it is escaped by another backslash, (\\). The combination "\\" counts as one character.)

---

line

Type:	document reading
Value:	line number for the cursor
Syntax:	variable = **line** conditional function(**line** operator expression) call function(**line**)

The **line** function reads the line number of the cursor location in the document. The line number returned by the **line** function is the line number shown in the status line, which reflects the Spacing setting.

---

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 reads the page, line, and position of the cursor location in the document. These values are 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 the header, footer, work, note, or footnote page.

---

max with numeric expressions

Type: mathematical

Value: the expression containing the highest number of all the 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 with alphabetical strings

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 with numeric expressions

Type: mathematical

Value: the expression containing the lowest number of all the 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 with alphabetical strings

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 is true if the expression is numeric, and false if it is not. Only numeric strings are recognized. If the string contains any alphabetic characters, the statement is false. 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. When the string contains any alphabetic characters, "12th" for example, a value of 0 is returned to the variable, indicating that the string is not numeric. The number may contain a leading dollar sign, commas, a decimal point, and/or leading or trailing minus or plus signs. When the cursor is on or after a decimal point, only the decimal point and the numbers following it are returned.

The number may not contain any change of text emphases such as boldface, underlines, or double underlines. For example, the number 4428 returns a zero value because the underline attribute changes halfway through the number. The number 4428 returns the correct value only if it is preceded by a space. A number that is part of a sequence of emphasized characters, such as the number 4428, is, returns 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 reads 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. If more than one symbol is used, they do not need to be separated by commas or formatted in any way, and can appear in any sequence. Expression2 must be a quoted string or a variable that contains the quoted string. When expression2 is a variable, the variable name should not be enclosed in quotation marks. Expression2 may include 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 reads 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 positions for posmsg are lines 1 through 25 and positions 1 through 80. Messages extending beyond position 80 or line 25 result in screen display anomalies. (See the "Display Functions" section in Chapter 10 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, or by including the statement "\027" (the octal representation for CTRL/w) in the entry. 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 entry; by pressing CTRL/w; by including the CTRL/w statement "\027" in the entry; 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 is true if the cursor is on the last character of a line. Otherwise, it is false. When it is preceded by the logical not operator (!), the combination !**right_margin** is 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 to 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 does not 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:	current 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, both in the second status line following the word "Spacing," and in the first position of the format line. To change the vertical line spacing during entry 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 is the line number shown in the status line, which reflects the Spacing setting.

---

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 26 to 50). The length of the **status** string cannot exceed 26 characters. Strings longer than 26 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 entry, by pressing CTRL/w, by including the CTRL/w statement "\027" in the entry, or by invoking an editing function such as **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 begins. Expression4 gives the segment number where the substitution ends. 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**(e1,e2))

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 Fortune:Word document control code for the Return symbol. Appendix C provides more information about Fortune:Word 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 Fortune:Word 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: current setting for document text length

Syntax: variable = **text_len**
conditional function(**text_len** operator expression)
call function(**text_len**)

The **text_len** function reads the current text length setting of the document. (The text length setting is shown on the second status line on the editing 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 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 is true if the cursor is on the first character of the first line of a page. Otherwise, it is false. When it is preceded by the logical not operator (!), the combination !**top_page** is 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 is 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 to the number of decimal places specified by expression2. The fractional part beyond the specified point is deleted regardless of its value.

unixfun

Type: operating system 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 Fortune:Word **command "!"** function. The "Operating System Access Functions" section in Chapter 10 gives examples of how to use both **unixfun** and **command "!"**.

unixpipe

Type: operating system access

Value: returns the output of an 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 entry pauses for key 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 Fortune:Word function **command "|"**. The "Operating System Access Functions" section in Chapter 10 gives examples that use both **unixpipe** and **command "|"**.

---

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 entry 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 entry 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, the left margin, or a Tab, Decimal Tab, Right-flush Tab, Indent, or Return symbol. Spaces surrounding the word are not stored in the variable.

Chapter 10

Function Usage List

Functions can be grouped by the type of actions they perform when a glossary entry is executing. For example, when you use display functions such as **prompt**, **clrpos**, and **status**, you can put messages on the text document editing screen while the entry is running. Document reading functions such as **char** and **page_no** provide information about the cursor position in the text document. Operating system access functions allow you to use a wide range of operating system commands in glossary entries.

In this list, functions are organized by the usage groups shown in Table 10-1. The introduction to each usage group describes the functions in that group, suggests ways to use them, and provides examples.

See the "Alphabetical List of Functions" in Chapter 9, and refer to other chapters in this book for additional examples and more detailed information about specific functions.

Table 10-1. Usage Groups and Their Functions

Usage Group	Function
Conditional Functions	do while
	if
	if else
	while
Control Functions	call
	exit
	glossary
	jump

Table 10-1. (continued)

Usage Group	Function
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	false
	globerr
	true
Interactive Functions	key
	keyin
	keys
	keysin
Mathematical Functions	abs
	max
	min

Table 10-1. (continued)

Usage Group	Function
	num
	number
	pic
	round
	truncate
Operating System Access Functions	date
	time
	unixfun
	unixpipe
	command "!"
	command "|"
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 entry 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 stop the entry.

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 entry examples for each conditional function.

Entry c 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**, stops 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 does not bold "2word" because the test is made before the statements are executed. Entry d does bold "2word" because the test is performed after the statements are executed.

```
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 entry.

The **jump** statement transfers control to a block of labeled statements in the entry. There are two parts to a jump statement:

- A statement which is enclosed in brackets, and labeled by an identifying name, called the identifier.
- A "jump identifier" statement at the place in the glossary entry where you want to transfer control. In this statement, the identifier is not enclosed in brackets.

You can use **jump** statements 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 must be preceded by a **call** statement. A subroutine is another glossary entry in the same glossary. The **glossary** statement transfers execution control to another entry in the same glossary. Either **call** or **glossary** can be used to make an entry recall itself.

The **exit** statement causes the entry to stop.

Chapter 8 gives you in-depth information and entry 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 entry, 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 to put messages on the text document editing screen while an entry is running, to put the cursor in a specified location, or to turn the screen display refresh function off and on.

Physical Editing Screen Locations

When you work in your document, you use the editing screen. The screen is a grid that measures 25 vertical lines (rows) by 80 horizontal character positions (columns). Figure 10-1 shows the screen grid layout of 25 vertical lines by 80 horizontal positions.

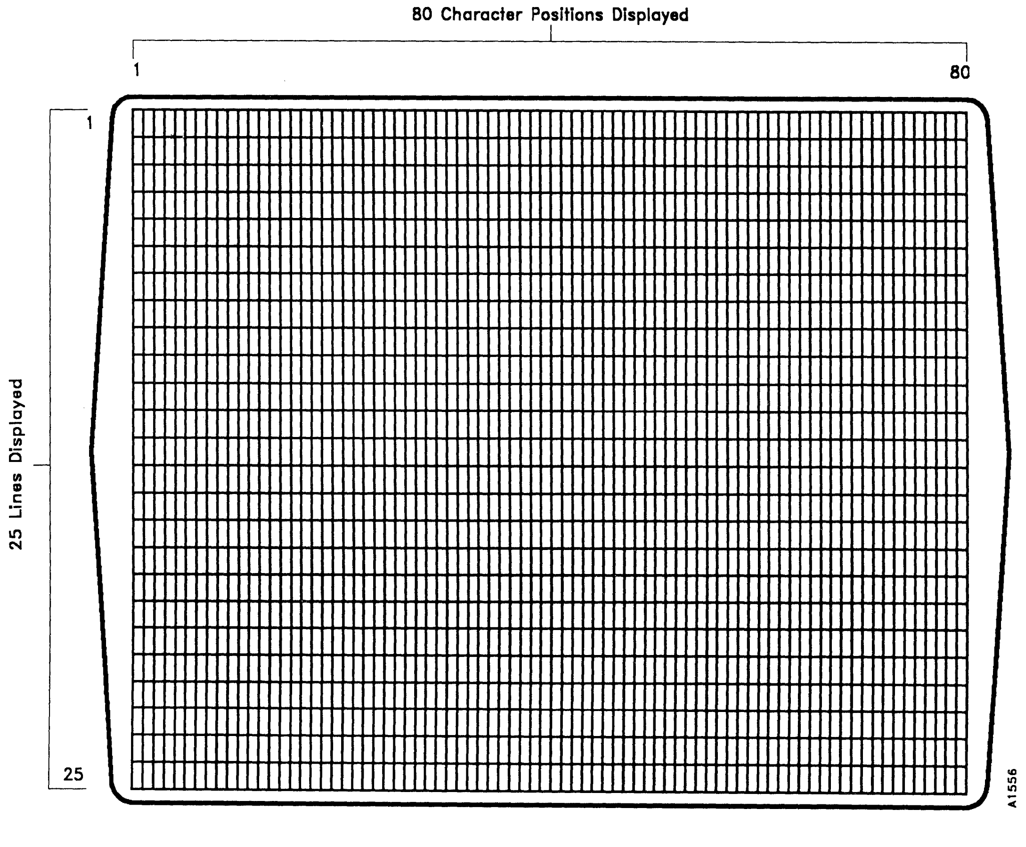


Figure 10-1. Editing Screen Grid

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.

When you use Fortune:Word, four of the 25 vertical lines are reserved to display status information and messages. The editing screen reserved areas are shown in Figure 10-2.

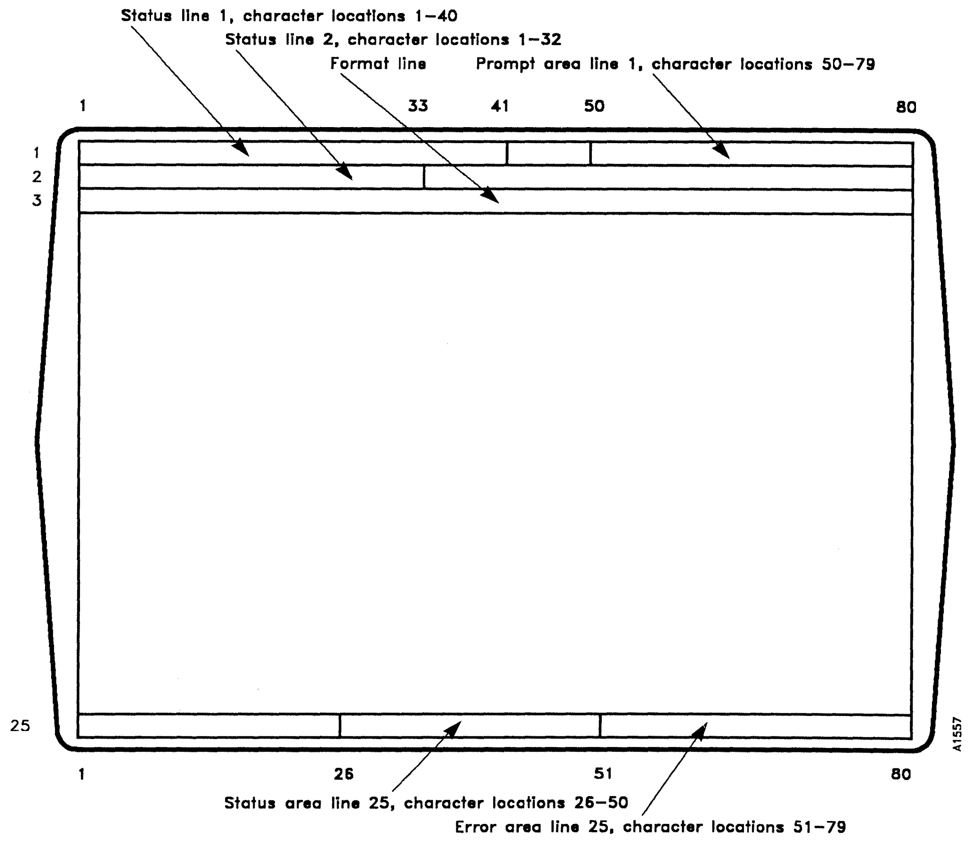
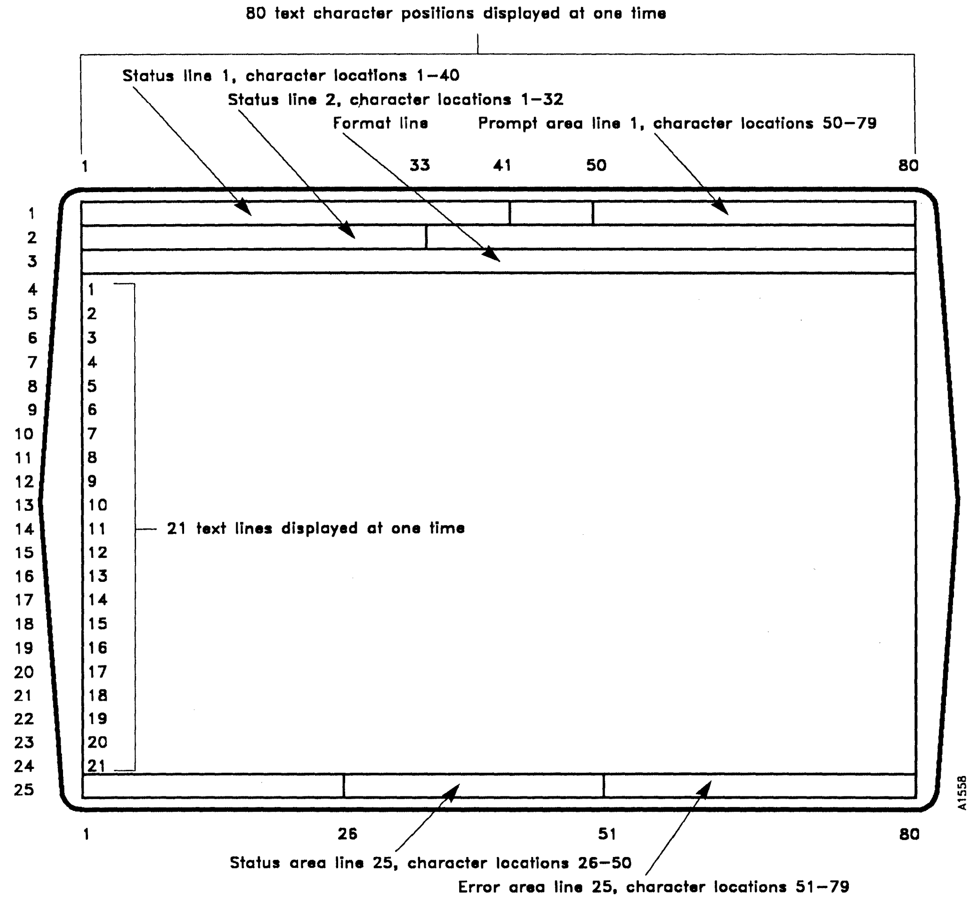

Status line 1, character locations 1–40
Status line 2, character locations 1–32
Format line
Prompt area line 1, character locations 50–79
1
33
41
50
80
1
2
3
25
A1557
1
26
51
80
Status area line 25, character locations 26–50
Error area line 25, character locations 51–79

Figure 10-2. Editing Screen Reserved Areas

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. 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 reads "Line 1." Figure 10-3 shows both physical and logical screen locations.



80 text character positions displayed at one time
Status line 1, character locations 1–40
Status line 2, character locations 1–32
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 26–50
Error area line 25, character locations 51–79
A1558

Figure 10-3. Physical Screen Locations vs. Logical Screen Locations

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, and then inserts a message.

```
entry e
{
  call posmsg(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
}
```

Notice that when you recall entry f, part of the message from entry e is cleared to allow room for other prompts. Although both **posmsg** and **cursor** specified line 1, the messages are displayed 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 is displayed at position 43 so it does not conflict with status line 1. Although status line 1 is in a reserved screen area (as shown in Figure 10-2) messages displayed with the **posmsg** function can temporarily overwrite the information on the status lines. You can see this if you change entry e to display the **posmsg** at position 1.

Pressing CTRL/w clears the remainder of the message from the prompt area. The text from **posmsg** is temporarily displayed on the screen but is not inserted 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.

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.

From the examples, you can see why the physical screen locations on the grid in Figure 10-1 are the numbers you must use when you specify line and position locations for the **posmsg** and **clrpos** functions.

The **posmsg** and **clrpos** function messages can be put at any location you choose on the editing screen grid. Acceptable line locations are 1 through 25. Acceptable character positions are 1 through 80. Glossary verification does not check for incorrect error locations. If you specified a line or position location outside these numbers, when you recall the entry the error message ***bad location*** is displayed and the entry stops. Messages that extend beyond 79 characters or line 25 cause screen display anomalies.

Using Editing Screen Message Locations

The **prompt**, **status**, and **error** functions have their message placement predefined in specific editing screen reserved areas as shown in Figure 10-2.

As you can tell from entry e, reserved areas of the editing screen are important considerations when you use 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 entry-generated messages.

Depending on your entry, you may want to replace a system message or status line with your own message. You should always try your entry 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 use keywords such as **insert, search, delete, copy,** or **replace,** the messages that are integral to these functions clear your entry-generated messages.

As shown in Figure 10-2, 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.

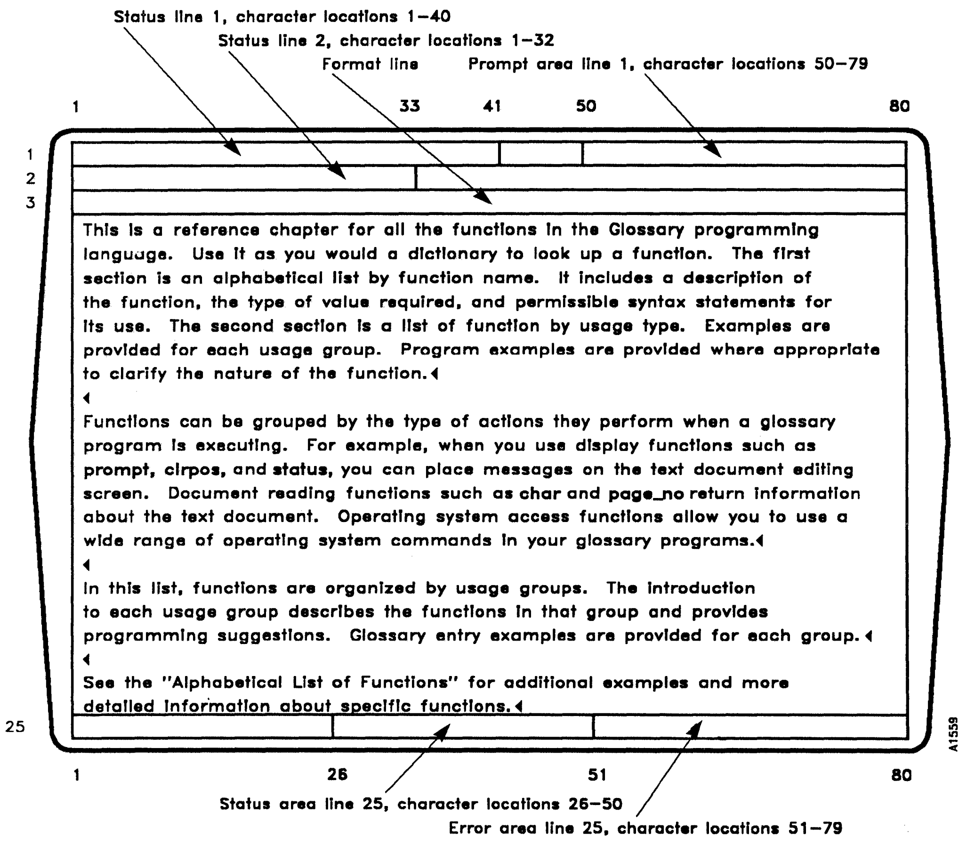
The **prompt**, **status**, and **error** functions all have predefined message display locations in the reserved areas as follows:

- **prompt** messages display on line 1, positions 50 through 79
- **status** messages display on line 25, positions 26 through 50
- **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.

Document Scrolling

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 10-4, 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).



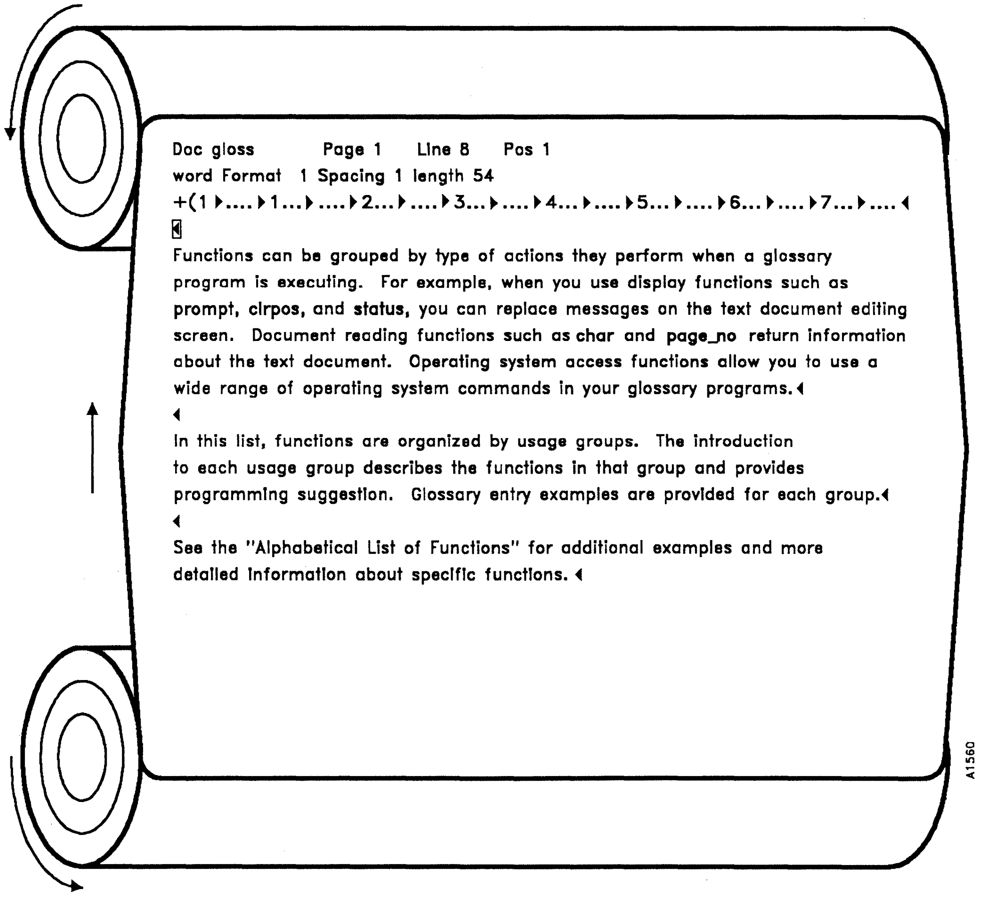
Status line 1, character locations 1–40
Status line 2, character locations 1–32
Format line
Prompt area line 1, character locations 50–79
1
33
41
50
80
1
2
3
25
1
26
51
80
Status area line 25, character locations 26–50
Error area line 25, character locations 51–79
A1559

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 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.◂

Figure 10-4. Text is Framed by Reserved Areas of the Screens

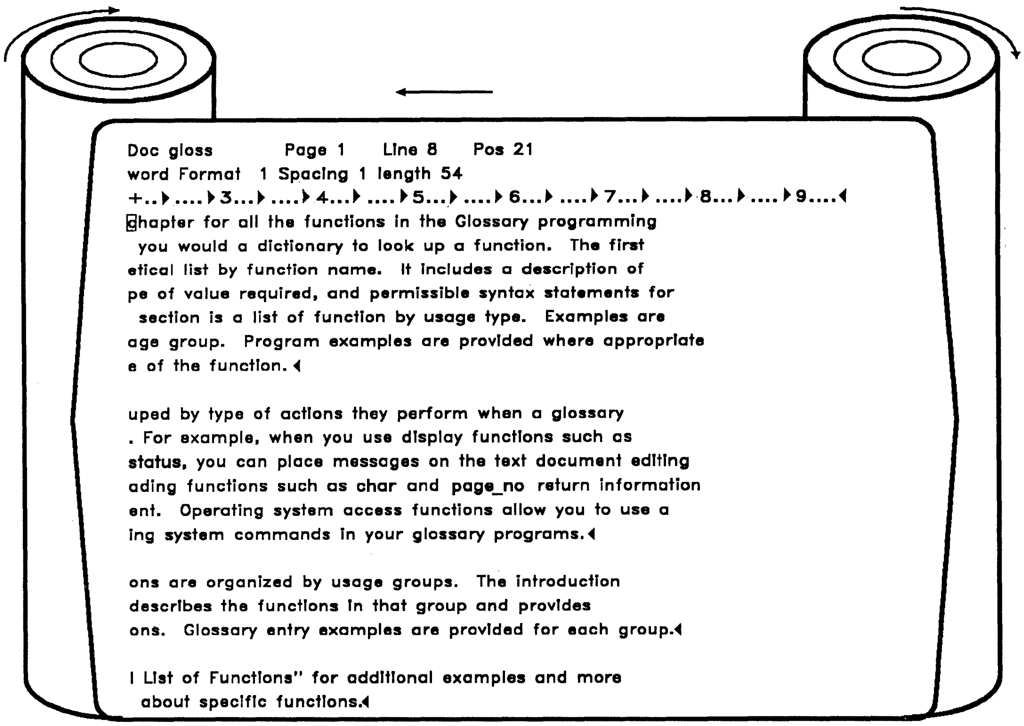
Figures 10-5 and 10-6 show the same text being scrolled vertically and horizontally. Notice how the reserved areas framing the text remain stationary as the text moves. The only changing elements in the reserved areas are the line and position numbers on status line 1, which reflect the cursor movement.

You can illustrate the principles shown in Figures 10-5 and 10-6 by scrolling text in your document. First, be sure your document page exceeds 21 lines and has a format that exceeds 80 characters. Scroll vertically through the text by pressing DOWN. Notice how the text lines change as the top three status lines remain stationary. Now scroll horizontally by pressing RIGHT. Notice how the text slides off the left side of the screen, but the status lines remain stationary in their physical screen location.



```
Doc gloss         Page 1    Line 8    Pos 1
word Format  1 Spacing 1 length 54
+(1 ▸....▸1...▸....▸2...▸....▸3...▸....▸4...▸....▸5...▸....▸6...▸....▸7...▸....◂
◂
Functions can be grouped by 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 replace 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 usage groups. The introduction
to each usage group describes the functions in that group and provides
programming suggestion. Glossary entry examples are provided for each group.◂
◂
See the "Alphabetical List of Functions" for additional examples and more
detailed information about specific functions. ◂
```

Figure 10-5. Text Can Scroll Vertically within Reserved Areas



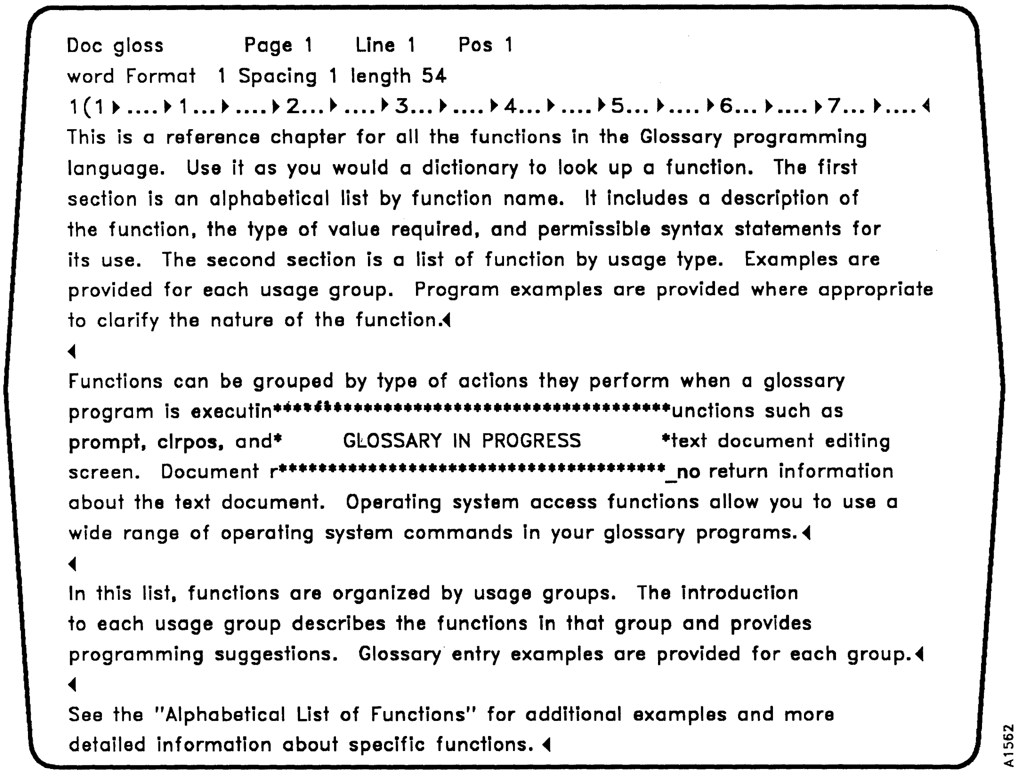
```
Doc gloss        Page 1    Line 8    Pos 21
word Format  1 Spacing 1 length 54
+..▸....▸3...▸....▸4...▸....▸5...▸....▸6...▸....▸7...▸....▸8...▸....▸9....◂
chapter for all the functions in the Glossary programming
 you would a dictionary to look up a function.  The first
etical list by function name.  It includes a description of
pe of value required, and permissible syntax statements for
 section is a list of function by usage type.  Examples are
age group.  Program examples are provided where appropriate
e of the function.◂

uped by type of actions they perform when a glossary
. For example, when you use display functions such as
status, you can place messages on the text document editing
ading functions such as char and page_no return information
ent.  Operating system access functions allow you to use a
ing system commands in your glossary programs.◂

ons are organized by usage groups.  The introduction
describes the functions in that group and provides
ons.  Glossary entry examples are provided for each group.◂

l List of Functions" for additional examples and more
 about specific functions.◂
```

Figure 10-6. Text Can Scroll Horizontally within Reserved Areas

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 10-7 shows text with an overlaid message posted by **posmsg**.

You can demonstrate this concept shown in Figure 10-7 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).")
}
```



```
Doc gloss         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 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 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 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. ◂
```

A1562

Figure 10-7. A posmsg Message Temporarily Overlays Screen Text

When you have used display functions in a few entries, the physical locations and logical locations become easy to remember. You can use display functions in a variety of situations. They are particularly valuable when you are writing entries for others to use, as you can post messages to the user on the screen while the glossary is running.

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 does not 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. This method works well for **prompt**, **status**, and error messages because they are in reserved (non-text) areas. It does not work as well for **posmsg** because you are still obscuring underlying text with blanks. Entry i uses **clrpos** in a **while** loop to clear the entire screen.
- Use CTRL/w to clear messages. Using CTRL/w in your entry 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 entry, 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 **status** and **error** functions are displayed simultaneously in an entry, 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 are 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 entry, the screen is restored during entry execution, just as it is while you are editing. Although a glossary entry restores the screen faster than normal editing does, it still slows the entry down.

You can use the **display** function to turn off the screen display restore during entry execution. This is particularly valuable for reducing runtime for lengthy entries.

A good example is entry 1 in Chapter 8, which uses the statement **call display(false)** at the beginning of the entry to turn the display off. The screen display is turned back on at the end of the entry by the statement **call display(true)**.

If you typed and recalled entry 1 in Chapter 8, using the "Amalgamated Widgets" example, you saw a semi-static display. Try removing both **display** statements from the entry and running it. Notice that the entry takes longer to run 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 display a message that an entry is running. Entry i incorporates a variation of entry 1 shown in Chapter 8.

The **posmsg** messages in entry i use octal numbers and attribute codes to display the messages in reverse video and flashing modes. (Keyword abbreviations for modes cannot be embedded in **posmsg** messages.) 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)

   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")
```

(**entry** i continued on next page)

(entry i continued)

```
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,"$,"))
```

(entry i continued on next page)

(entry i continued)

```
            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"
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 **gloss2b** on the Glossary Examples 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.

Write some entries to try 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 entry execution, document reading functions read values from the text document that reflect the cursor position, status line information, or format line information.

Cursor Location Functions

Cursor location functions are grouped into two types:

- Those that return numeric or alphabetic string values
- 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 listed below:

- 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.

True or False Values

The **beg_doc**, **bot_page**, **end_doc**, **left_margin**, **right_margin**, and **top_page** functions are true or false (1 or 0) 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
   }
```

(**entry** 1 continued on next page)

(**entry l** continued)

```
   [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,** and 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 to write than entry l, but works equally well. Again, this illustrates 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 Fortune:Word document control codes
- 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 entry, the text is 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 Fortune:Word document control codes as screen symbols, 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) or \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, put your cursor on a screen symbol such as Return, Indent, or Decimal Tab 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 glossary entries.

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 Characters

CTRL/y characters are special characters that are accessed by typing CTRL/y and then typing a character. CTRL/y characters are most frequently used for changing laser printer fonts or typing foreign or accented characters. Both **finsert** and **feed** recognize and print CTRL/y characters.

Characters from alternate character sets can also be accessed in Fortune:Word using CTRL/] and CTRL/n key sequences. You cannot use CTRL/] and CTRL/n characters with glossary.

Using the Cursor Function

While **cursor** is 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.

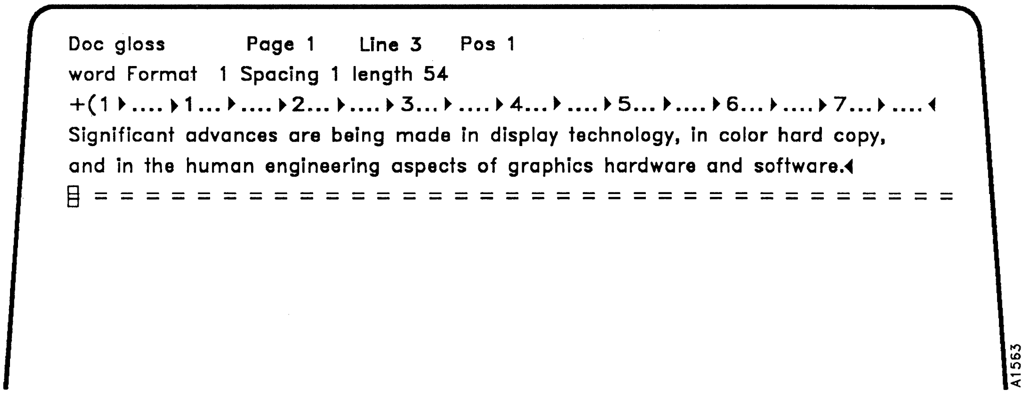
The **cursor** function uses the logical screen location line and position numbers discussed in the "Display Functions" section in this chapter. The cursor cannot be put in the reserved screen areas of lines 1 through 3 and line 25.

Open and Unopened Editing Screen Areas

When you use the **cursor** function, you must consider unopened areas of the screen. Look at the text example in Figure 10-8.

Presume that the screen placement for the example begins the text on page 1, line 1, with an indent set at position 6. The text ends on line 2, position 72. The screen is "open" from line 1, position 1, to line 3, position 1 (the end of the document). The remainder of the screen is "unopened" because it does not contain characters. The statement **call cursor("1,3,48")** sends the cursor to line 3, position 1, 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 an entry is executing in a document, you do not always know which areas of the screen are open or closed. If this is a concern in your entry, assign the desired cursor position to a variable. Then use the **loc** function to return the cursor position and compare it to the variable.



```
Doc gloss        Page 1    Line 3    Pos 1
word Format  1 Spacing 1 length 54
+(1 ▸.... ▸1... ▸.... ▸2... ▸.... ▸3... ▸.... ▸4... ▸.... ▸5... ▸.... ▸6... ▸.... ▸7... ▸.... ◂
Significant advances are being made in display technology, in color hard copy,
and in the human engineering aspects of graphics hardware and software.◂
▯ = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
```

Figure 10-8. Text Example of Unopen Screen Areas

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 called and sent to the location specified by the variable **curpos = "4,22,12"**. The arrival location of the cursor is checked by comparing **loc** against the variable **curpos.** If they do not 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 cursor is not able to go to position 12 since the screen area is not open at that position. The entry sends the cursor as close as it can get to "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 do not want to move the cursor, type **quit** and press EXECUTE. The **exit** statement stops 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"))
                     {
```

(**entry p** continued on next page)

(entry p continued)

```
                    "\027" exit
               }
     }
  call finsert(dollars)
}
```

ERROR AND LOGICAL FUNCTIONS

- false
- globerr
- true

Using the Globerr Function

Use the **globerr** function to trap standard Fortune:Word function errors. A standard function such as **search, nextscrn, prevscrn**, or **goto** sounds a beep when it cannot complete its function. For example, the **search** function beeps when it cannot find another instance of the word it is searching for. The **nextscrn** function beeps when there is no next screen to go to.

The **globerr** function helps you to branch, loop, or stop an entry gracefully if a standard function fails. Entries r and s 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 0. Entries f and g in Chapter 6 are examples that use **true** and **false** functions.

INTERACTIVE FUNCTIONS

- key
- keys
- keyin
- keysin

Using Interactive Functions

Interactive functions let you stop the entry 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 one of the following statements:

```
variable = key
call feed(variable)
```

```
        variable = keys
        call feed(variable)
```

- Use one of the following **feed** statements (in this case, **key** or **keys** is not stored in a variable):

```
        call feed(key)

        call feed(keys)
```

The Key Function

The **key** function accepts one typed key from you, then immediately continues entry execution.

Any key on the keyboard is accepted by **key** and may be assigned to a variable. CTRL/y by itself is considered as one keystroke and is accepted. Any characters that follow CTRL/y exceed the one-character limit and therefore are not accepted by **key**. 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 a **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. Execution continues when you press EXECUTE or RETURN. Only character keys (including CTRL/y characters) are accepted by **keys**. A beep sounds if a formatting or editing key, such as Copy, Insert, Delete, or Search, is pressed.

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 entry, you can clear **keys** input without terminating the entry 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 automatically.

The Keyin and Keysin Functions

The **keyin** and **keysin** functions are 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. Execution continues immediately after you type the key. You cannot correct a mistake (even though you can see it on the screen) until the entry concludes.

The **keysin** function accepts an unlimited numbers of keys. It accepts any key on the keyboard except EXECUTE and CANCEL. Pressing EXECUTE stops **keysin** entry, and entry execution continues. Pressing CANCEL during **keysin** entry stops the glossary entry.

When recalling a glossary entry with **keysin**, you can use the Backspace or cursor 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 the **keysin** entry.

When writing a glossary entry with **keyin** or **keysin**, remember that both functions place their input directly in the document. Existing text is overwritten unless the cursor is on a Return or you are at the end of a page or the end of a document. To avoid overwriting existing text in a document, you can use the keyword **insert**. When you use **keysin** with an insert in an entry, do not end the insert with an **execute**, as shown in entry v. The EXECUTE that concludes **keysin** also ends the insert

When you use **keyin** or **keysin** to enter text in an interactive entry, you may want to post messages in the prompt area to indicate when operator intervention is required. It may not be obvious from the entry when input is required.

```
entry v
{
  insert
      call keysin
}
```

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.

Using Interactive Functions with Display Functions

When you place a **key** or **keys** statement in your entry 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 unless you provide an instruction in the entry to do so. Entry x shows one way to enter the key as text in a document.

When you use **keys**, the input to **keys** appears to overwrite whatever text exists at the message location. These characters can be cleared by using a CTRL/w statement (octal 027) in your entry.

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 entries 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 entry.

```
entry x
{
  call posmsg(11,40,"enter key: ")
  x = key
  call feed(x)
}


entry y
{
  call keyin
  call prompt("enter keyin: ")
}


entry z
{
  call error("enter keys: ")
  y = keys
  call feed(y)
}


entry A
{
  call status("enter keysin: ")
  call keysin
}
```

MATHEMATICAL FUNCTIONS

- abs
- max
- min
- num
- number
- pic
- round
- truncate

Using Mathematical Functions

Mathematical functions can be used in a wide variety of entries. 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 reads 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 previous examples, mathematical operators can be used for counting loops. Entry C clears the entire screen line by line. When the cursor returns to the top of the screen, the page, line, and position numbers and the crosshair are displayed. Press CTRL/W to redraw the 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 an entry for creating a calculator that can perform simple mathematical calculations in a document. Although Fortune:Word has a built-in Math function that is more complete and faster than Entry D, this example shows how to use mathematical functions in a glossary entry. (Entry D is in **gloss3** on the Glossary Examples 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.

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
```

(**entry D** continued on next page)

(entry D continued)

```
                (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"))
            {
```

(entry D continued on next page)

(entry D continued)

```
        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
   }
```

(entry D continued on next page)

(entry D continued)

```
      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

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 17.07692307692307692307.

```
      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.

Be sure to use **round** if you want a 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 9 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,
> –4.27 credit due to overpayment for January, which amounts to $_____.
>
> Thank you for your patronage of Leche Dairy.
>
> Elsie Bovine, Office Administrator

Ms. Bovine wants to use the overpayment amount in the letter as an operand to the glossary calculator (entry D). However, the calculator does not 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 entry. As such, it is a good entry 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.

Using the Max and Min Functions

When you write glossary entries for others to use, you have no way of knowing the exact document conditions during entry execution. This means you have to build more document reading or entry analysis functions into each entry.

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 entry. Two of the variables are declared and initialized in the entry, but the third variable must be read from the document. You do not know what the document number variable is because it is different 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 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
            (1,41,"\034HD Place cursor on number & press EXECUTE
            \034ID")
      call keysin
      call clrpos(1,41,40)
      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 Operating System Access Functions

These functions allow you to include operating system commands in an entry. If you have never used the operating system at the shell command level, be very careful about experimenting with these functions. You do not need to understand operating system commands 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** 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 on your text document editing screen. Remember, both the date and time function take their values from the system date and time.

```
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 display 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 uses the **time** function to display just the time. Note that entry G uses **unixfun** to execute the 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 operating system commands you can invoke from the document editing screen.

When you call **unixfun** in your entry, it performs the following actions:

1. Escapes from the document to the Bourne shell **sh** (this action is transparent; you do not literally see this occurring).
2. Executes the 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 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 the entry.

Because the operating system 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, as shown in the following example:

```
xinput = "keys"
call unixfun(xinput)
```

An alternative to **unixfun** is the **command "!"** keyword statement. See the next section in this chapter for a description of **command "!"**.

When you call **unixpipe** in your entry, it performs the following actions:

1. Escapes to the Bourne shell **sh**.
2. Executes its commands.
3. Writes the standard output of the command in your text document. The standard output becomes a part of the document.

The keyword statement **command "|"** is an alternative to **unixpipe**. See the next section for a description of **command "|"**.

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 operating system command in quotes; expression2 is the data expected by the command. Since very few 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 9 for two glossary entry examples that use **unixpipe**.

Using command "!" and command "|"

Both **command "!"** and **command "|"** are not functions, but 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
{
  command "!"
  "sort documentb -o documentb"
  return
  execute

  command "|"
  execute
  "cat documentb"
  return
}
```

Entry I uses **command "!"**, **command "|"**, and the operating system **sort** command to sort a Fortune:Word 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, follow these steps:

1. Create a document named **documentb**.

2. Type a simple list of words to be sorted, one word per line.

3. You MUST be in a document of a different name to try this entry. If you use this entry in the document to be sorted, the operating system cannot access the document because it is in use by Fortune:Word. (You can press CANCEL twice to end the entry and return to the document.)

4. Press GL and type **I**.

If you are 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 and type ! You are now in the shell. Type a 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 shell. Type a command and press EXECUTE or RETURN. The output of the command replaces the text you highlighted in your document.

Good operating system 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 shell. You may want to write a glossary entry 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 using string values. You can use string functions in many ways. Some suggestions for their use follow:

- 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

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 in the format

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**, and **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)
  if((substr(day,1,1)) == " ")
     {
          day = substr(day,2,2)
     }

  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 entry, write a short entry to check the string. Be careful, however, which date function you use in your entry. Entries K and L both return the date; however, the values returned by entry K (which uses the glossary **date** function) and entry L (which uses **command** "|" to return the system date) are 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 the operating system:

```
Tue Jun 16 08:17:57 PDT 1987
```

This is the system date returned by the **date** function in entry K:

```
Tue Jun 16 08:19:59 1987
```

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 entry. If the operator types the full name for a state rather than the two-character abbreviation, an error message is displayed and the operator is asked to retype the state.

```
entry M
{
  call prompt("Enter state: ")
  state = keys
  call clrpos(1,50,31)
     if(len(state) > 2)
     {
          call error("State too long, re-enter: ")
          state = keys
     }
  call feed(state)
  "\027"
}
```

The **index**, **occur**, **seg**, and **sub** functions can be used to select and test for specific words or phrases. These functions can be used in Records Processing control glossary entries. See the *Fortune:Word Records Processing User's Guide* for examples.

Chapter 11
Administering Glossary Entries

Administering entries is just as important as writing them. It is a good idea to frequently review and update glossary entries. This chapter presents information on the following topics:

- Entry planning
- Troubleshooting
- Obsolescence
- Setting up and maintaining a glossary entry log book
- Administering entries in a multiuser environment
- Security

When you become proficient in Glossary, two things usually occur:

- You rapidly acquire a large collection of entries.
- You write entries for other people to use.

When this occurs, the following considerations become vital:

- Entry planning: Why do you need the glossary entry? How long will it take to write? Is the time spent writing it worth it? Is the planned application suitable for a glossary entry? Would it be better to use a spreadsheet or Records Processing?

- Applications: What does the entry do? Who should use it? How do you use it?

- Access: Where are the glossaries on the system? Which entry is in which glossary?

- Runtime: How long does an entry take to run? How do you schedule runtime? How much system space does it consume?

- Backup, storage, and retrieval: Where are the entries stored? How often do glossary documents need to be backed up? Where is the printed copy kept?

- Obsolescence: Is an entry still useful? Can it be updated or should a new entry be written?
- Duplication: Why are there different versions of the same entry on the system? Which one is correct?
- Glossaries in a multiuser environment: How did the glossary get renamed? Who's using the glossary when you want to edit it? Who made all those weird changes to an entry?
- Debugging (troubleshooting): What syntax error? This isn't a bug, it's a monster!

This chapter discusses each of these topics and gives you some practical advice and technical tips.

ENTRY PLANNING

Consider writing a glossary entry for the following reasons:

- When you continually rekey or copy the same text. For example, an address, legal paragraphs, form letters, technical terms, or standard forms.
- When you type tables in a document, then hand-calculate and inserting the results. You can decide to use the Math feature, a glossary entry, or a combination of both.
- When you must 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 entry, or a combination of both.
- When you are working with mailing lists, parts lists, or inventory lists where some items remain standard and other items change periodically. Glossary entries are an integral part of the Records Processing feature.

Of course, there are many other reasons for writing an entry. Some of them will be particular to your own working environment. The reasons listed are the most universal applications. If you are performing the same task on a periodic basis, consider writing a glossary entry.

When you are planning an entry to perform production tasks, compare the amount of time required to write the entry against the amount of time saved by the entry. Obviously, you do not want to spend three hours writing an entry that saves five minutes one time only. But, if you spend three hours writing an entry that saves five minutes a day, your time is well spent.

Sometimes you can get too ambitious with a glossary entry. If you are planning to use glossary entries for extremely large financial spreadsheets or massive mailing lists, 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 are more efficient than a glossary entry for these uses.

APPLICATIONS

It is a good idea to provide three kinds of information for every entry you write:

- What the entry does (its application)
- How the entry flow is executed (why it works the way it does)
- How the entry is used (which keys you press or what you type if it is interactive)

Set up a "Glossary Entries Information Notebook." Provide a separate tab for each glossary and include the following information:

- Index of the glossary that shows each entry label with a one-line comment about the entry
- Printed copy of each entry
- Instruction sheet for entries that are long enough or complicated enough to require instructions

The amount of information required for an entry increases with the complexity of the entry. Comment lines in the entry are probably sufficient for simple entries. Longer entries or entries designed for others to use may require printed directions.

The effort you make in setting up and maintaining adequate glossary records helps save you time in keeping track of entries and their uses.

ACCESS

In addition to knowing how your entries work and how to use them, you need to know where they are on the system. If you have a small system, you can probably remember which glossarys 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 Fortune:Word. 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 library, you must give the full pathname, which includes the **/u** directory, the user's login directory, the glossary library, and the glossary name. The full pathname might look like this 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**.

Linking A Glossary to Another Library

If you are familiar with using operating system commands, you can use the **ln** command to link glossaries to libraries and sublibraries. The **.gl** file contains the executable form of the glossary. You can link just the **.gl** file. When you do this, any changes made to the glossary from its library are always reflected in each library the **.gl** file is linked to. You can link the same file to many libraries. You can only link a file from an operating system shell. To link a **.gl** file, follow the steps below:

1. Go to an operating system shell.

2. Change to the library that contains the glossary you want to link by typing **cd *libraryname*** and pressing RETURN.

3. Type **ln *glossaryname*.gl *pathname*** and press RETURN, where ***pathname*** is the name of the library where you want the linked glossary to reside.

4. If you want to verify that the document has been linked, use the ll command. The number of links to a file are shown in the second column.

RUNTIME

Runtime is a real consideration in using entries. All glossary entries execute in the "foreground." Your terminal is unavailable for use with other applications while the entry is running. With a multiuser system, you can use another terminal to run the entry. However, in a busy production environment, th.s is not always possible. As you have learned, you can write entries in ways that help speed up runtime. For example, by turning the display refresh off with the **call display(false)** statement.

You could create a schedule to run lengthy entries during lunch hours, in the evenings, or overnight. Including a "glossary in progress" message in your entry helps notify people that an entry is running and that the terminal should not be used until the entry is finished. Entry i in the "Display Functions" section of Chapter 10 includes this message.

If your entries take excessively long to run, Glossary may not be a suitable solution for your application.

BACKUP, STORAGE, AND RETRIEVAL

Like any Fortune:Word document of value, glossary documents should be backed up to an archive diskette every time a change is made. Clearly label the diskettes and store them in a safe place.

You can keep a record of archived glossary documents by printing the archive diskette index and placing it in the front of your "Glossary Entries Information Notebook."

Always keep a printed copy of each entry. If someone deletes the entry from the system and you do not have a backup copy on an archive diskette, you can retype the entry rather than rewriting it.

OBSOLESCENCE

Entries can become obsolete when office procedures change, when you think of a better entry, or when you improve an old entry. Periodically review entries and delete those that are obsolete. They take up space and can cause confusion if someone tries to use them.

When you update an entry, get rid of the previous version. You can keep a printed copy of all your old entries, or have a special archive diskette just for obsolete entries.

Try to write entries with an eye toward future modification. Comment lines and documentation help a lot when you are updating an old entry. It is easy to forget what your logic was or exactly what the entry is doing if the entry flow is not commented. Working on someone else's entry is even more difficult.

If possible, have periodic meetings of all the glossary writers in your department to review entries. You can also discuss and establish standard commenting and documenting procedures. If your group writes a large number of entries, these meetings can help spread information about glossary usage and can provide a vehicle for sharing new entries.

ENTRY DUPLICATION

Several versions of the same entry can cause a lot of confusion. Be sure to note revision numbers on your entries. File or delete old versions. Notify all your entry users when you replace an old entry with a new one.

GLOSSARIES IN A MULTIUSER ENVIRONMENT

Several users can attach and use the same glossary at the same time. You cannot edit, archive, copy or move a glossary while another user has it attached or is 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.

Use the comments line on the Glossary Summary screen for notes about the glossary. A comment like "See System Administrator before making any changes" informs someone accessing the document that you do not want changes made without your permission.

Notations in your glossary notebook can help clarify who is responsible for creating and maintaining specific entries.

If you are concerned about security or do not want to permit other users to edit your glossaries, you can:

- Password your glossary (see the *Fortune:Word Reference Guide* for information about password protecting documents).

- Change file permissions on your glossary (refer to the *Fortune:Word Reference Guide* and *FOR:PRO User's Guide* for information about file permissions).

DEBUGGING ENTRIES (TROUBLESHOOTING)

A "bug" is programming parlance for an error in an entry. 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 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 while the entry is running and usually cause the entry to stop abruptly (crash). They can result from trying to divide by zero, from using an incorrect keyword sequence for a standard Fortune:Word function, or from leaving out an input statement (like **key** or **keysin**). Even if the entry verified, 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 suggestions 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 use the same variable more than once in the entry, make sure it's spelled correctly each time.
- Be sure you do not inadvertently duplicate variables.
- Check your subroutine calls—are you calling the correct entry?

A very subtle logic bug can occur when you are performing mathematical calculations. The entry can appear to be running properly, but the calculations are wrong. Always check your entry results against a set of known results. If you write an entry to add a column of figures, also add the column on a hand calculator to be sure the entry is adding correctly.

Points to Remember

Here are some suggestions about writing and debugging entries:

- Don't get frustrated if the entry doesn't work right the first time.
- The bug is probably something simple.
- Build the entry gradually, testing each part before you proceed to the next step.
- Debug your entry systematically by developing a troubleshooting routine.

Chapter 12
Glossary Information for FOR:PRO Users

FORTUNE:WORD FILE STRUCTURE

Each Fortune:Word document is made up of at least three files. Two other types of files are generated for special types of Fortune:Word documents (glossary and exception dictionary), which must be compiled before they are used in Fortune:Word. You can perform filing operations on Fortune:Word documents from the FOR:PRO shell. If you do this, be sure you copy or move all the files associated with each Fortune:Word document. The files and their extensions are listed in Table 12-1.

Table 12-1. Fortune:Word Document Files

Fortune:Word Document Files	Description
filename	The text of a document
filename.dc	The history, statistics and page pointer information for the document
filename.fr	The formats, header, footer, work, note, and footnote pages for the document
filename.gl	The compiled and executable form of a glossary document
filename.ex	The compiled and executable form of an exception dictionary

When you perform a FOR:PRO command such as **rm**, **cp**, or **mv**, follow the Fortune:Word document name by the metacharacter (*) to ensure that all files are included. For example, the following command removes the Fortune:Word file **report** from the directory (library).

```
rm report*
```

THE .GL FILE

The .gl file is only present if the document is a compiled glossary. When a glossary is created and verified, 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 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 listed below:

- 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. See Chapter 11 for information on how to link a file.

- 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. See Appendix C in the *Fortune:Word Reference Guide* for information on ownership and permissions.

- To save space on the system you can delete all but the .gl file. Since it contains the object code, it is executable without the other files. Be sure to copy the entire file to an archive diskette before you delete any files. When you want to make additions to the glossary or edit an entry, you can load the three files back on the system, edit the glossary and recompile it.

Chapter 13
Glossary Entry Examples

Your Fortune:Word Glossary Examples Diskette contains glossary documents **gloss1, gloss2a, gloss2b,** and **gloss3**. All the examples shown in this book are in these glossary documents except the glossary by example entries you create.

You can retrieve the glossaries from the diskette as you would any other Fortune:Word documents. See the *Fortune:Word Reference Guide* for information on retrieving documents from an archive diskette.

You can attach each of the glossary documents and use the entries in it as you learn Glossary. Using these glossaries saves you typing time. You can edit them and modify any entry for your own use.

In addition to glossary entries, the example tables used with some entries are provided on Page N of glossary document **gloss2b**.

CONTENTS OF GLOSSARY DOCUMENTS

The entries in each glossary document are shown in the following list. The chapter in this book where the entry is discussed is shown in a comment line after the entry label. Page numbers correspond to the 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 Chapter 1*/	3
entry f	/*in Chapter 4*/	4
entry g	/*in Chapter 4*/	5
entry h	/*in Chapter 4*/	6
entry i	/*in Chapter 4*/	7
entry j	/*in Chapter 4*/	8
entry k	/*in Chapter 4*/	9

Entries in Glossary Document: gloss2a

entry a	/*in Chapter 5*/	1
entry b	/*in Chapter 5*/	2
entry c	/*in Chapter 5*/	3
entry C	/*in Chapter 5*/	4
entry d	/*in Chapter 6*/	5
entry e	/*in Chapter 6*/	6
entry f	/*in Chapter 6*/	7
entry g	/*in Chapter 6*/	8
entry h	/*in Chapter 6*/	9
entry i	/*in Chapter 6*/	10
entry j	/*in Chapter 6*/	11
entry k	/*in Chapter 6*/	12
entry l	/*in Chapter 6*/	13
entry m	/*in Chapter 6*/	14
entry n	/*in Chapter 6*/	15
entry o	/*in Chapter 6*/	16
entry p	/*in Chapter 6*/	17
entry r	/*in Chapter 6*/	18
entry s	/*in Chapter 6*/	19
entry t	/*in Chapter 6*/	20
entry u	/*in Chapter 6*/	21
entry v	/*in Chapter 6*/	22
entry w	/*in Chapter 6*/	23
entry x	/*in Chapter 6*/	24
entry y	/*in Chapter 6*/	25
entry z	/*in Chapter 6*/	26
entry B	/*in Chapter 6*/	27
entry D	/*in Chapter 6*/	28
entry E	/*in Chapter 6*/	29
entry F	/*in Chapter 6*/	30
entry G	/*in Chapter 6*/	31
entry H	/*in Chapter 6*/	32
entry I	/*in Chapter 6*/	33
entry J	/*in Chapter 6*/	34
entry K	/*in Chapter 6*/	35
entry L	/*in Chapter 6*/	36

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 8*/	23
entry n	/*in Chapter 8*/	24
entry o	/*in Chapter 8*/	25
entry p	/*in Chapter 8*/	26
entry q	/*in Chapter 8*/	27

Entries in Glossary Document: gloss3

entry a	/*in Chapter 9*/	1
entry b	/*in Chapter 9*/	2
entry c	/*in Chapter 10*/	3
entry d	/*in Chapter 10*/	4
entry e	/*in Chapter 10*/	5
entry f	/*in Chapter 10*/	6
entry g	/*in Chapter 10*/	7
entry h	/*in Chapter 10*/	8
entry i	/*in Chapter 10*/	9
entry j	/*in Chapter 10*/	10
entry k	/*in Chapter 10*/	11
entry l	/*in Chapter 10*/	12
entry m	/*in Chapter 10*/	13
entry n	/*in Chapter 10*/	14
entry o	/*in Chapter 10*/	15
entry p	/*in Chapter 10*/	16
entry r	/*in Chapter 10*/	17
entry s	/*in Chapter 10*/	18
entry t	/*in Chapter 10*/	19
entry u	/*in Chapter 10*/	20
entry v	/*in Chapter 10*/	21
entry x	/*in Chapter 10*/	22
entry y	/*in Chapter 10*/	23
entry z	/*in Chapter 10*/	24

entry A	/*in Chapter 10*/	25
entry B	/*in Chapter 10*/	26
entry C	/*in Chapter 10*/	27
entry D	/*in Chapter 10*/	28
entry E	/*in Chapter 10*/	29
entry F	/*in Chapter 10*/	30
entry G	/*in Chapter 10*/	31
entry H	/*in Chapter 10*/	32
entry I	/*in Chapter 10*/	33
entry J	/*in Chapter 10*/	34
entry K	/*in Chapter 10*/	35
entry L	/*in Chapter 10*/	36
entry M	/*in Chapter 10*/	37
entry 1	/*in Appendix C*/	38
entry 2	/*in Appendix C*/	39
entry 3	/*in Appendix C*/	40
entry 4	/*in Appendix C*/	41
entry 5	/*in Appendix C*/	42

Appendix A
Reserved Words and Symbols

The words and symbols in this appendix are reserved for Glossary and Records Processing keywords, functions, and operators. Reserved words 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	display	INDENT	NORTH	SEARCH
ascending*	do	indent	north	search
backspace	DOWN	index	NOTE	seg
beg_doc	down	INSERT	note	select_record*
bot_page	EAST	insert	num	sort*
call	east	jump	number	SOUTH
CANCEL	else	key	occur	south
cancel	end_doc	keyin	PAGE	space
cat	entry	keys	page	spacing
CENTER	error	keysin	page_no	status
center	EXECUTE	LEFT	pic	STOP
char	execute	left	position	stop
clrpos	exit	left_margin	posmsg	sub
COMMAND	false	len	PREVSCRN	subscript
command	feed	line	prevscrn	substr
COPY	finsert	loc	prompt	SUPERSCRIPT
copy	FORMAT	max	quote	superscript
cursor	format	MERGE	REPLACE	TAB
date	gl	merge	replace	tab
DECIMALTAB	globerr	min	RETURN	text
decimaltab	glossary	MODE	return	text_len
DECTAB	GOTO	mode	RIGHT	thru*
dectab	goto	MOVE	right	time
DELETE	HELP	move	right_margin	top_page
delete	help	NEXTSCRN	round	true
descending*	if	nextscrn	save_record*	truncate

RESERVED WORDS (Continued)

unixfun	UP	WEST	while
unixpipe	up	west	word

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 used by two Fortune:Word applications: Glossary and Records Processing. Although the glossary-writing procedure is similar for both, the glossary keywords and functions you can use are different for each application.

- Glossary uses all keywords and functions except special Records Processing selecting and sorting functions.
- Records Processing uses some 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 FORTUNE:WORD 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
center	keyword	x	x
char	function	x	
clrpos	function	x	
command	keyword	x	
COPY	keyword	x	
copy	keyword	x	

Name	Type	Glossary	Records Processing
cursor	function	x	
date	function	x	
decimaltab	keyword	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	
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	x
globerr	function	x	
glossary	keyword	x	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
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	

Name	Type	Glossary	Records Processing
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	
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

Name	Type	Glossary	Records Processing
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 how to use attribute codes in your glossary entries. Information is provided on the following topics:

- The ASCII collating sequence
- Octal number conversions
- Attribute codes
- Fortune:Word Document Format Control Codes

ASCII COLLATING SEQUENCE

The ASCII (American Standard Code for Information Interchange) collating sequence is a standard set of numeric codes used to represent characters. The attribute codes in Table C-1 are listed in order of the ASCII collating sequence.

> NOTE: Glossary uses the operating system ASCII collating sequence. Fortune:Word functions that sort use a case-insensitive ASCII collating sequence, and that sequence can be modified. See Appendix E in the *Fortune:Word Reference Guide* for more information.

An example of how you can use the ASCII collating sequence with Glossary is shown in entry 1. This entry underscores a word, excluding punctuation and numbers, by comparing characters according to their ASCII number value.

```
entry 1
{
   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 2 includes two octal codes: \007 (CTRL/G) sounds the keyboard bell, and \027 (CTRL/W) refreshes the screen display.

```
entry 2
{
   call posmsg(1,43,"Enter Amount: \007")
   amount = keys
   "\027"
   call feed(amount)
}
```

Octal codes you may want to use in glossary entries are listed below:

Control \	\034
Keyboard bell	\007
Page (required)	\014
Refresh screen	\027
Return (hard)	\013
Return (soft)	\012
Tab	\011

Table C-2 shows control codes and the correct glossary syntax for these octal codes.

ATTRIBUTE CODES

Glossary provides you with a set of keyword abbreviations you use to add emphasis such as boldface or underline to text you insert in a document. These abbreviations can be embedded in strings used by variables or functions. Table D-4 in Appendix D lists these keyword abbreviations.

The only glossary functions that do not accept keyword mode abbreviations are the display functions. When you want to add emphasis to screen messages that do not become part of the document, you must use attribute codes to add emphasis. The attribute codes are letters or symbols that set a specific emphasis mode or combination of modes. Attribute codes and the modes they set are shown in Table C-1.

Entries 3 and 4 illustrate the difference between using keyword mode abbreviations and attribute codes to highlight a message. In entry 3, the keyword abbreviation **\b** for boldface is embedded in the **call feed** string. In entry 4, which uses the **posmsg** function, the attribute code for boldface must be preceded by an octal code and an operator selector (the operator selectors H and I are described below).

```
entry 3
{
  call feed("\bCustomer Name:\b  ")
  return(2)
  call keysin
}
```

```
entry 4
{
  call posmsg(1,43,"\034H' Customer Name:\034I'")
  name = keys
  "\027"
}
```

Use the following syntax to include an attribute code in a display function message:

```
    display function("\034H attribute_code  message  \034I
    attribute_code")
```

The following examples for **prompt** and **posmsg** use the syntax shown above to display their messages in flashing 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 following 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 the operator selector to use to set the attribute code to "on."

- Turn off the attribute code with the following sequence:

  ```
  \034I attribute_code
  ```

  Capital I is an operator selector to turn off an attribute 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.

  NOTE: The octal numbers and attribute codes described here apply to terminals manufactured by Fortune Systems Corporation.

If your terminal does not correctly display entries that contain these attribute codes, remove them from the message string in your glossary entries.

If the **posmsg** strings shown 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 \034IB")
```

does not display properly, change it to:

```
call posmsg(7,26,"GLOSSARY IN PROGRESS")
```

DESCRIPTION OF TABLE C-1

Table C-1 is ranked in ascending ASCII collating sequence. The first column shows the attribute character, the second column describes the attribute or attribute combination set by the attribute code.

Table C-1. Character Codes

Attribute Code	Terminal Attribute Displayed
Space	None
!	None
“	None
#	None
$	None
%	None
&	None
’	None
(	None
)	None
*	None
+	None
’	None
–	None
.	None
/	None
0	None
1	None
2	None
3	None
4	None
5	None
6	None
7	None
8	None
9	None

Table C-1. (continued)

Attribute Code	Terminal Attribute Displayed
:	None
;	Bold, double underline, and reverse video
<	None
=	None
>	None
?	None
@	Resets all attributes
A	Overstrike
B	Flashing
C	Flashing and overstrike
D	Reverse video
E	Reverse video and overstrike
F	Reverse video and flashing
G	Reverse video, flashing, and overstrike
H	High underline
I	High underline and overstrike
J	High underline and flashing
K	High underline, flashing, and overstrike
L	High underline and reverse video
M	High underline, reverse video, and overstrike
N	High underline, reverse video, and double underline
O	High underline, reverse video, flashing, and overstrike
P	Low underline
Q	Low underline and overstrike

Table C-1. (continued)

Attribute Code	Terminal Attribute Displayed
R	Low underline and flashing
S	Low underline, flashing, and overstrike
T	Low underline and reverse video
U	Low underline, reverse video, and overstrike
V	Low underline, reverse video, and overstrike
W	Low underline, reverse video, flashing, and overstrike
X	Double underline
Y	Double underline and overstrike
Z	Double underline and flashing
[	Double underline, flashing, and overstrike
\	Double underline and reverse video
]	Double underline, reverse video, and overstrike
^	Double underline, reverse video, and flashing
_	Double underline, reverse video, flashing, and overstrike
`	Bold
a	Bold and overstrike
b	Bold and flashing
c	Bold, flashing, and overstrike
d	Bold and reverse video
e	Bold, reverse video, and overstrike
f	Bold, reverse video, and flashing
g	Bold, reverse video, flashing, and overstrike
h	Bold and high underline

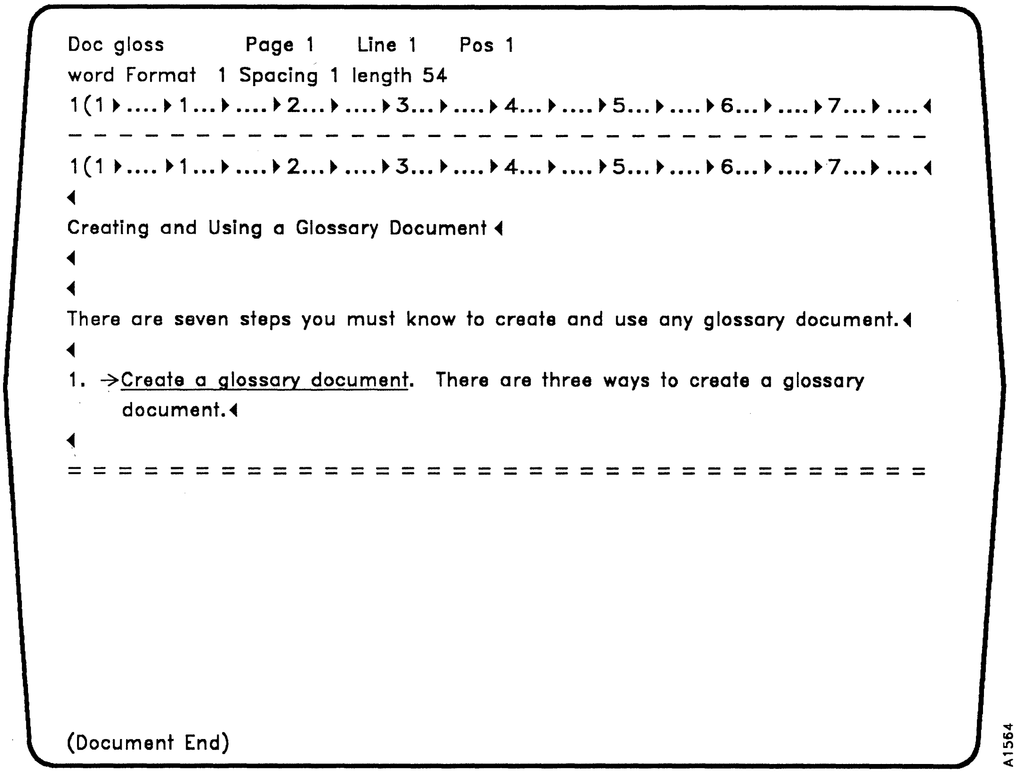
Table C-1. (continued)

Attribute Code	Terminal Attribute Displayed
i	Bold, high underline, and overstrike
j	Bold, high underline, and flashing
k	Bold, high underline, flashing, and overstrike
l	Bold, high underline, and reverse video
m	Bold, high underline, reverse video, and overstrike
n	Bold, high underline, reverse video, and flashing
o	Bold, high underline, reverse video, flashing, and overstrike
p	Bold and low underline
q	Bold, low underline, and overstrike
r	Bold, low underline, and flashing
s	Bold, low underline, bold, and overstrike
t	Bold, low underline, and reverse video
u	Bold, low underline, reverse video, and overstrike
v	Bold, low underline, reverse video, and flashing
w	Bold, low underline, reverse video, flashing, and overstrike
x	Bold and double underline
y	Bold, double underline, and overstrike
z	Bold, double underline, and flashing
{	Bold, double underline, flashing, and overstrike
|	None
}	Bold, double underline, reverse video, and overstrike
~	Bold, double underline, reverse video, and flashing

FORTUNE:WORD DOCUMENT FORMAT CONTROL CODES

Fortune:Word formatting characters are displayed on the 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 operating system commands, you can see these codes by using the **more** command to view the document from the shell. Figure C-1 shows you how a fragment of text looks on the document editing screen, and Figure C-2 shows the same text viewed from the operating system.

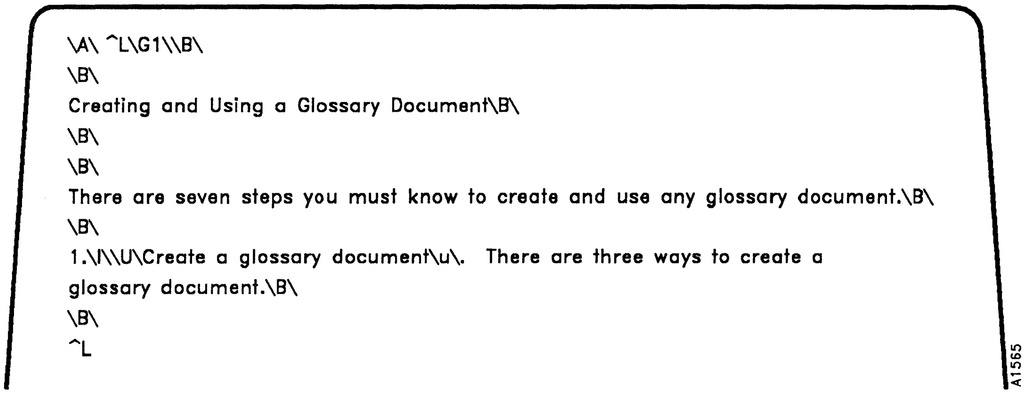
As you can see by comparing the two figures, the combination of an optional page break, the format line, and a return, requires the control sequence \A\^L\G1\\B\.



```
Doc gloss        Page 1    Line 1    Pos 1
word Format  1 Spacing 1 length 54
1(1▸....▸1...▸....▸2...▸....▸3...▸....▸4...▸....▸5...▸....▸6...▸....▸7...▸....◂
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
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.◂
◂
= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =





(Document End)
```

A1564

Figure C-1. How Text and Formatting Characters Look on the Document Editing Screen



```
\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\
^L
```

Figure C-2. How Fortune:Word Text and Control Codes Look from the Shell

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 5 shows one way to test whether the character at the cursor position is a Return. The document control code sequence for the Return symbol is assigned to **retl**. 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 on a Return. If the cursor is not on 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 5
{
retl = "\\B\\\012"
ret2 = text(loc,loc)
  if(retl =( 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
```

When you use a glossary function such as **len** that counts the number of characters, each character in the Fortune:Word document format control code is counted. For example, if a string of text is underscored, six characters are added to the length of the string to account for the \U\ at the beginning of the underscored text and the \u\ at the end. Return symbols count as four characters: three for the \B\ control characters, and one for the newline that follows each Return but is not displayed.

Table C-2 shows the action performed in the document, each Fortune:Word document format control code, and a brief description of the code and how the code can be used in your programs.

Table C-2. Fortune:Word Document Control Codes

Action Performed	Control Code	Description
Backslash	\\\	Since backslash is used as a control code delimiter, a backslash typed in the document is escaped by backslashes.
Bold off	\x\	Occurs after the last character in a bold sequence and turns bold mode off.
Bold on	\X\	Occurs before the first character in a bold sequence and turns bold mode on.
Center	\c\	Centers a single line between the right and left margins; displayed in the document as a diamond.

Table C-2. (continued)

Action Performed	Control Code	Description
Character from G2 set*	\^Ycd\	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\.
Column break, (optional)	\C\	Optional column break optional deleted by the pagination process; displayed in the document as a row of periods.
Column break, (required)	\F\	Required column break not deleted by the pagination process; displayed in the document as a row of colons.
Decimal Tab	\t\	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.
Double Underline off	\d\	Occurs after the last character in a double underlined sequence and turns double underline mode off.
Double Underline on	\D\	Occurs before the first character in a double underlined sequence and turns double underline mode on.
Flashing off	\z\	Occurs after the last character in a flashing sequence and turns flashing mode off.
Flashing on	\Z\	Occurs before the first character in a flashing sequence and turns flashing mode on.

Table C-2. (continued)

Action Performed	Control Code	Description
Footnote Reference	\Nnnn\	Footnote reference number, where "nnn" stands for the footnote number; displayed in text as a number in reverse video.
Format number identification	\Gnnn\	A Fortune:Word 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.
Hyphen, generated	\H\	Generated by the hyphenation process; removed if subsequent document editing causes the line to rewrap; displayed in the document as a bright hyphen.
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.)
Indent	\I\	Left justifies wrapped text under a format line Tab until a hard Return is encountered; displayed in the document as a right arrow.
Merge off	\>\	Right field name delimiter for Records Processing and a marker for other applications; displayed in the document as a bright >.

Table C-2. (continued)

Action Performed	Control Code	Description
Indent, generated	\i\	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.
Line feed	^K	Used to break contiguous character strings at the right margin; not displayed in the document, and deleted when the line is reformatted. The syntax to test for a CTRL/k is: **char =** "\014"
Merge on	\<\	Left field name delimiter for Records Processing and a marker for other applications; displayed in the document as a bright <.
Note	\n\	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 and printed as a double exclamation mark.
Optional page break	\A\^L	An optional page break that can be deleted by the pagination process; displayed in the document as a row of hyphens. The syntax for assignment to a glossary program variable is: **variable =** "\\A\\\014"

Table C-2. (continued)

Action Performed	Control Code	Description
Overstrike off	\o\	Occurs after the last character in an overstrike sequence and turns overstrike mode off.
Overstrike on	\O\	Occurs before the first character in an overstrike sequence and turns overstrike mode on.
Required page break	^L	A required page break that is not deleted by the pagination process; displayed in the document as a row of equal signs. The syntax for assignment to a glossary program variable is: **variable** = "\014"
Return (hard)	\B\^J	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". To test for a hard return with the **key** function, the syntax is: if(key == "015"
Return (soft)	^J	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"
Reverse video off	\r\	Occurs after the last character in a reverse video sequence and turns reverse video mode off.
Reverse video on	\R\	Occurs before the first character in a reverse video sequence and turns reverse video mode on.

Table C-2. (continued)

Action Performed	Control Code	Description
Right-flush Tab	\M\	Right justifies text under a format line Right-flush Tab until a Return, Tab, or another Right-flush Tab is encountered; displayed in the document as a left arrow.
Space, required	\ \	Prevents separation of words by marginal word wrap; displayed in the document as a square U, printed as a space.
Subscript	\s\	When the document is printed, a subscript symbol causes the printer to move down 1/4 line; displayed in the document as a down arrow.
Superscript	\S\	When the document is printed, a superscript symbol causes the printer to move up 1/4 line; displayed in the document as an up arrow.
Tab	^I	Advance cursor to the next Tab stop in the format line (if no Tab stop is 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"
Underline off	\u\	Occurs after the last character in an underlined sequence and turns underline mode off.
Underline on	\U\	Occurs before the first character in an underlined sequence and turns underline mode on.

*Refer to the Fortune Systems publication *Using Fortune Terminals.*

Appendix D
Keywords by Usage

Keywords can be grouped by the functions they perform. Tables D-1, D-2, and D-3 are guides for using formatting, editing, and cursor movement keywords.

Cursor position and movement are extremely important factors in glossary programming. If you are not thoroughly familiar with cursor movement during Fortune:Word 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

Some keywords must be used in combinations. For example, to invoke a Right-flush Tab you must use the keyword combination **command indent.**

Table D-1. Formatting Keywords

Keyword	Performance in Glossary Entry
center*	The center symbol appears on the screen; any following text up to a Return 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.

Table D-2. Editing Keywords

Keyword	Performance in Glossary Entry
cancel*	An executing function is canceled, or the document edit is canceled and the End Of Edit Options menu 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. Either keyword can be used.
help	The word processing HELP screen is invoked.
insert	The insert function is invoked and the message *Insert what?* appears.
merge*	The left-hand symbol for Merge appears on the screen.
MERGE*	The right-hand symbol for Merge appears on the screen.

Table D-2. (continued)

Keyword	Performance in Glossary Entry
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 and 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 and the Global Search and Replace screen appears.
search	The search function is invoked and the message *Search for what?* appears.
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.

Table D-3. Cursor Movement Keywords

Keyword	Performance in Glossary Entry
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.**
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 and the Up keys 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.

Table D-3. (continued)

Keyword	Performance in Glossary Entry
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.**
SOUTH*	Moves the cursor according to the current cursor mode. This is equivalent to pressing the Shift and the Down keys simultaneously. Alternatively, you can use the keyword **DOWN.**
up*	The cursor moves up one line. If there is no text immediately above the cursor, the cursor moves to the end of 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 and the Up keys 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.

Table D-4. Keyword Abbreviations

Function	Code	Keyword Syntax
backslash	\\	
boldface ON	\B	mode “b”
boldface OFF	\b	mode “b”
center	\c	center
decimal tab	\\.	dectab or decimaltab
flash ON	\F	mode “f”
flash OFF	\f	mode “f”
footnote reference	\N	command “n”
help	\h	help or command help
hyphen (generated)	\H	
hyphen (optional)	\\-	command “-”
indent (generated)	\I	
indent	\i	indent
merge ON	\\<	merge
merge OFF	\\>	MERGE
note	\n	note
overstrike ON	\O	mode “/”
overstrike OFF	\o	mode “/”
page break (optional)	\g	page
page break (required)	\G	PAGE
quote (double)	\q	quote
return (required)	\r	return
return (word wrap)	\w	
reverse video ON	\V	mode “r”
reverse video OFF	\v	mode “r”
right-flush tab	\R	command indent
space (required)	\(space)	command “ ”
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	

* 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. See Chapter 3 of the *Fortune:Word Reference Guide* for a complete list of glossary operation error messages.

VERIFICATION ERROR MESSAGES

All verification error 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** is verified, the error messages following entry a are posted on page W.

```
entry a
{
    call posmsg("Enter Amount:  \007")
    amount = keys
    "\027"
    call feed(amount)
{
```

```
***********************************
Tue Feb 3, 1987 at 14:43:39
***********************************

page 6 , line 3 : 3 arguments expected for posmsg( )
page 6 , line 7 : 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 7. The syntax error is because of 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 cannot find the error on the reported line. Sometimes a missing brace or misplaced comment symbol can cause an entire series of errors for entries that follow. If this happens, check the first entry and correct the error and reverify the glossary before you try to correct errors in subsequent entries.

Table E-1. 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 that incorrect symbols are present or that symbols are missing. 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.

Table E-1. (continued)

Message	Possible Errors
: 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.
: 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.
unmatched 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.

Table E-1. (continued)

Message	Possible Errors
Unknown symbol	An incorrect symbol appears in the entry; this error is most common in mathematical applications.
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.

Table E-2. Glossary Operation Error Messages

Message	Possible Errors
Cannot attach	The glossary document entered does not exist, is not in the library, 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

FORTUNE SYSTEMS

300 Harbor Boulevard
Belmont, CA 94002