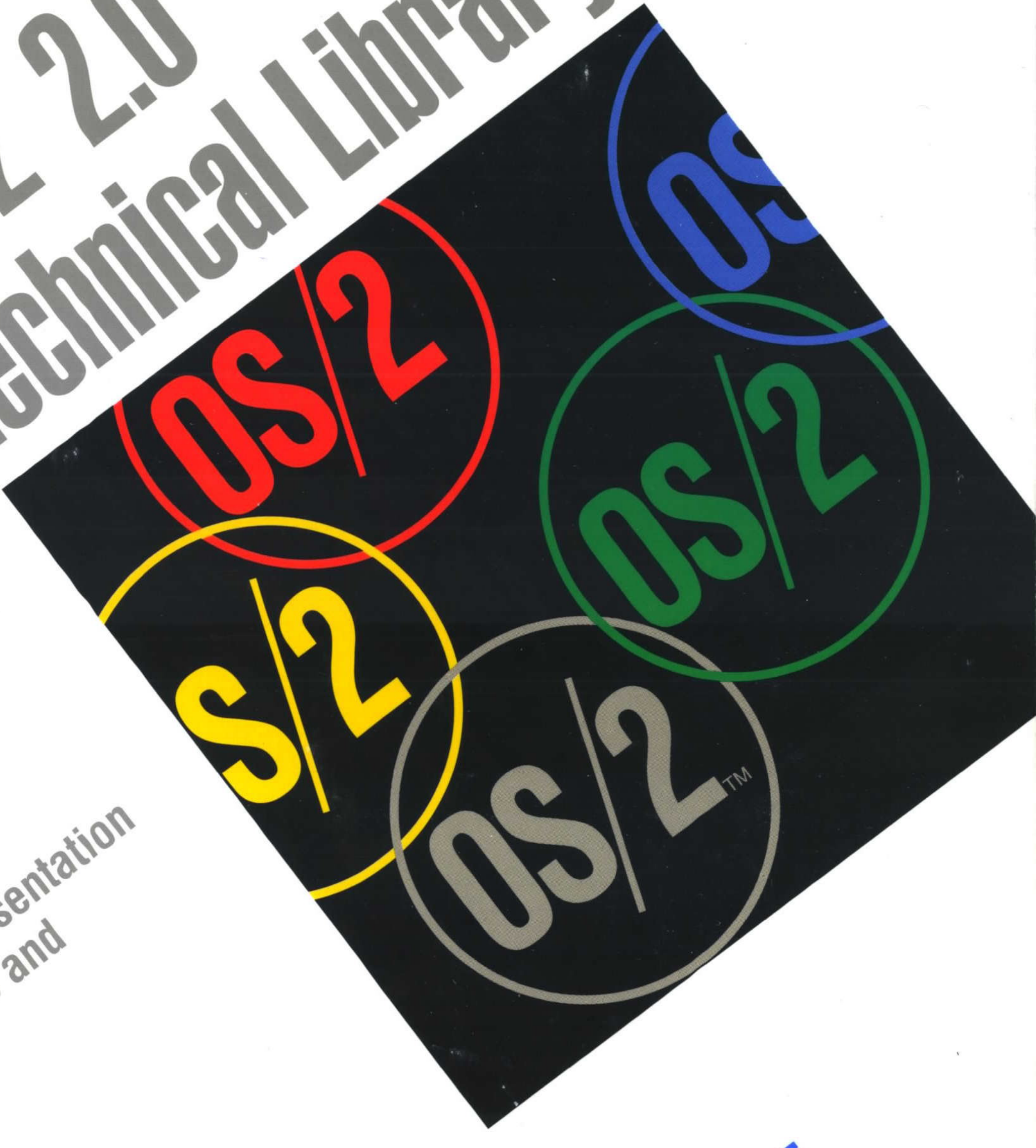
OS/2 2.0 Technical Library



Information Presentation Facility Guide and Reference

Version 2.00

IBM

Programming Family

OS/2 2.0 Technical Library

Information Presentation Facility Guide and Reference

Version 2.00

IBM

Programming Family

Note

Before using this information and the product it supports, be sure to read the general information under "Notices" on page v.

First Edition (March 1992)

Contents

Part 1: User's Guide

Notices

The following terms, denoted by an asterisk (*) in this publication, are trademarks of the IBM Corporation in the United States and/or other countries:

Common User Access
CUA
IBM
Operating System/2
OS/2
Personal System/2
Presentation Manager
Systems Application Architecture

The following terms, denoted by a double asterisk (**) in this publication, are trademarks of other companies as follows:

Adobe	Adobe Systems Incorporated
Helvetica	Linotype AG
Intel486	Intel Corporation
PostScript	Adobe Systems Incorporated
Times New Roman	Monotype Corporation

About This Book

This book describes the elements that make up the Information Presentation Facility (IPF). IPF is a tool that supports the design and development of online documents, and online help facilities.

Who Should Read This Book

This book is for the information author and the application programmer using the OS/2 2.0 operating system, and the Developer's Toolkit for OS/2 2.0 (Tookit).

To the Information Author: Use "Part I: User's Guide" and "Part III: Tag Reference" if you are designing and writing online information.

To the Application Programmer: Use "Part II: Programmer's Guide" if you are developing code that creates the Help interface for an OS/2 application. You are expected to be knowledgeable in the C-language, and know how to use the programming library associated with the Toolkit.

How this Book is Organized

Part I: User's Guide

This section introduces you to the features of IPF and shows you how to use the tagging language. You will learn to present text, connect information, and create and customize windows.

Part II: Programmer's Guide

This section describes the programming techniques used to develop C-language source code for a Help interface. Extensive use of sample source code is provided.

The reference chapters include the IPF Help Manager messages, and the application programming interface (API) that supports dynamic data formatting and the creation of help windows.

Part III: Tag Reference

This section provides an alphabetic reference of the IPF tagging language, including the symbols that are used to display special characters.

An appendix provides IPF error messages.

Part 1: User's Guide

Information Presentation Facility

Chapter 1. Introducing IPF

The Information Presentation Facility (IPF) is a tool that enables you to create online information, to specify how it will appear on the screen, to connect various parts of the information, and to provide help information that can be requested by the user.

IPF was introduced with the Operating System/2* Version 1.2, and was improved for OS/2 2.0 (OS/2*). It is a tool for both the information author and the application programmer. IPF implements guidelines recommended by the Common User Access*(CUA*) element of the Systems Application Architecture*(SAA*) platform.

What It Offers

As a writer of online information, you need to know what type of information users need — tutorial, reference, or help. For example, they might need a tutorial to learn a software program, reference information for additional topics, or help information for assistance with the program. As a designer of online information, you need to know what IPF features support your design. IPF features include:

- A tagging language that formats text, provides ways to connect information units, and customizes windows
- A compiler that creates online documents and help windows
- A viewing program that displays formatted online documents.

The Tag Language

The IPF tagging language provides the instructions for how online information is to be displayed. With these instructions, or tags, you can:

- Highlight text
- Set margins
- Add lists, notes, and notices
- Create tables
- Change the size and style (font), and the color of displayed information
- Control the formatting of lines of text
- Illustrate with examples, figures, and art
- Customize windows
- Define ways to connect information units
- Establish communication links to other applications.

The IPF Compiler

When you have finished writing and tagging, information is ready to be compiled. The IPF compiler interprets the tags in your source file and converts the information into the appropriate format. The compiler is able to distinguish between tags and text because each tag starts with a colon (:), is immediately followed by the tag name; and then ends with a period (.). For example, the tag that indicates a new paragraph is the **:p.** tag. When the compiler encounters this tag, it interprets it as, "Insert a blank line before the paragraph tag and start the text that follows the paragraph tag."

* Trademark of the IBM Corporation

At compile time, you specify what format you want. For online documents, you direct IPF to generate a file with an INF file extension. For help information, you specify a file with an HLP file extension. For information about compiler commands and options, see "Starting the IPF Compiler" on page 7-3.

The View Program

The View program (VIEW) enables you to display your compiled document. VIEW retrieves files with an INF extension and displays the formatted information in a standard OS/2 window.

Note: You cannot use VIEW to display files with an HLP extension. For information about how to use VIEW, see "Viewing an Online Document" on page 7-4.

What You Get

Online designs need to communicate information through a simple structure that lets the user find information quickly and easily. With IPF you can develop a design that provides unique usability features, including:

- Hypertext links
- Push buttons
- Customized windows
- Master index.

Hypertext and Hypergraphic Links

IPF gives the user the ability to connect to different units of text and graphics. The connections that join these units are known as hypertext or hypergraphic links. For example, a user can select a particular link to obtain related information or perhaps to see a graphical description of the topic.

An advantage of using hypertext or hypergraphic links is that the author can present information in a nonlinear way. Users can then access information both sequentially and randomly. This lets them explore or branch into subject matter that may be unclear or that needs to be reviewed. For information about hypertext and hypergraphic links, see Chapter 5, "Linking."

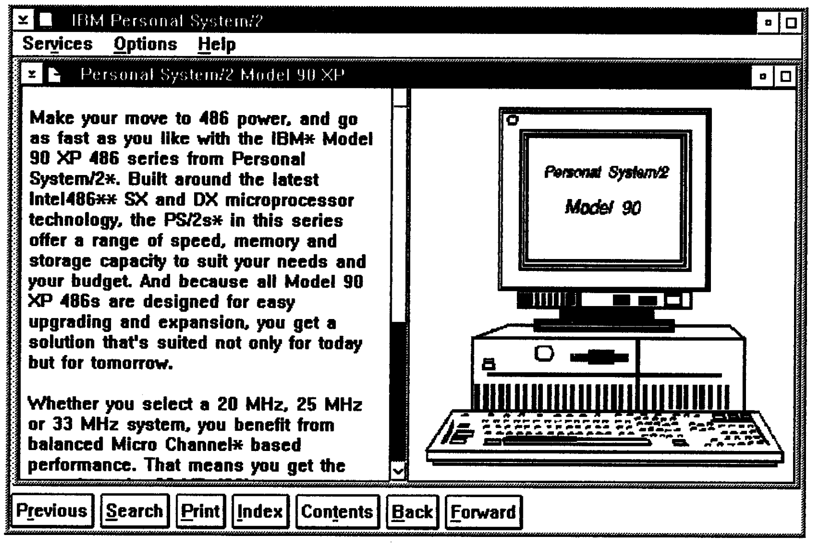
Push Buttons

Push buttons provide users with a fast and easy way to access commonly used IPF tasks. When a user selects a push button, the action represented by the text on the push button is carried out immediately. IPF provides one set of push buttons for online documents and another set for Help windows. IPF also provides help on how to use the push buttons.

As a designer, you can change the text of a push button, select which push buttons you want to use, add your own push buttons, and specify the area of a window to place them. For more information, see "Push Buttons" on page 3-7.

Customized Windows

A window is the area of the screen in which information is displayed. As an author of online information, you can customize windows. Different windowing effects are achieved with the IPF tagging language. For example, a window can be split so that scrollable text can be displayed beside a stationary illustration that the text describes. Figure 1-1 shows an IPF split-window design that describes the IBM* Personal System/2* Model 90 XP 486 series.**



Make your move to 486 power, and go as fast as you like with the IBM* Model 90 XP 486 series from Personal System/2*. Built around the latest Intel486** SX and DX microprocessor technology, the PS/2s* in this series offer a range of speed, memory and storage capacity to suit your needs and your budget. And because all Model 90 XP 486s are designed for easy upgrading and expansion, you get a solution that's suited not only for today but for tomorrow.

Whether you select a 20 MHz, 25 MHz or 33 MHz system, you benefit from balanced Micro Channel* based performance. That means you get the

Figure 1-1. IPF Split Window

For more information, see Chapter 6, "Customizing Windows."

Master Index

The OS/2 operating system provides online help that can be accessed through the Master Help Index. The Master Help Index contains an alphabetic list of topics related to using the operating system. It also points the user to further information. Using IPF tags, you can participate in the Master Help Index by adding global index entries to your online help windows. For more information, see "Master Help Index" on page 3-13.

* Trademark of the IBM Corporation

** Trademark of Intel Corporation

Chapter 2. Viewing the User Interface for Online Information

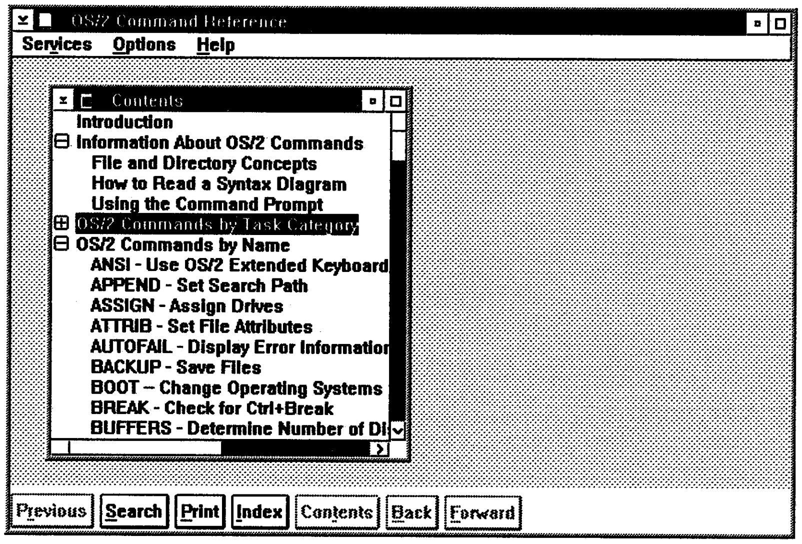
This chapter describes the components of the IPF user interface. As an author, you will use these components when developing an online document or online help. Online documents include reference or procedural information, such as converted printed material, tutorials, and organization charts. Online help includes information that users of online documents or application programs might want to access.

Because they have varying backgrounds, interests, motivations, and experiences, no two users of online information are exactly alike. To accommodate differences, IPF provides a flexible interface that can be customized according to personal preference. However, when an online document requires no special design considerations, the IPF compiler provides an automatic default design that includes:

- A Contents window
- Standard OS/2 windows
- Help.

The Contents Window

When users first select a document for viewing, IPF displays an OS/2 window that includes a table of contents (Contents window) similar to the window shown in Figure 2-1.



Introduction
⊟ Information About OS/2 Commands
File and Directory Concepts
How to Read a Syntax Diagram
Using the Command Prompt
⊞ OS/2 Commands by Task Category
⊟ OS/2 Commands by Name
ANSI - Use OS/2 Extended Keyboard
APPEND - Set Search Path
ASSIGN - Assign Drives
ATTRIB - Set File Attributes
AUTOFAIL - Display Error Informatio
BACKUP - Save Files
BOOT – Change Operating Systems
BREAK - Check for Ctrl+Break
BUFFERS - Determine Number of Di

Figure 2-1. A Contents window. Users select the highlighted item and go directly to a window of text.

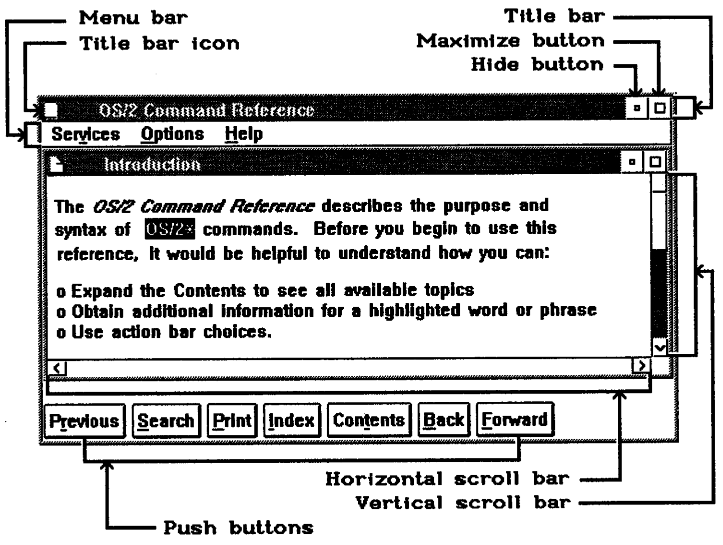
To find out how to use the Contents window, see page 2-9.

The Standard Window

Unless special window characteristics are defined with IPF tags, the IPF compiler formats a window that includes the following elements.

- Menu bar
- Title bar icon
- Title bar
- Maximize button
- Hide button
- Horizontal scroll bar
- Vertical scroll bar
- Push buttons.

Figure 2-2 shows a standard window and its elements.



The *OS/2 Command Reference* describes the purpose and syntax of OS/2 commands. Before you begin to use this reference, it would be helpful to understand how you can:

o Expand the Contents to see all available topics
o Obtain additional information for a highlighted word or phrase
o Use action bar choices.

Figure 2-2. Standard window and elements.

The title bar icon, and the maximize and hide buttons allow a user to change the size and position of a window. The menu bar, push buttons, and scroll bars allow a user to work with the window's contents. The window title indicates the subject of the information, or name of the object, seen in the window.

Help

While using an online document or application program, a user occasionally requires additional information about choices, fields, or procedures for a task. CUA guidelines recommend that a product provide information to a user about how to use the product. Information about how to use a product is known as help information. The OS/2 user interface for help information is developed with IPF and is accessible from the menu bar. Help can also be accessed from push buttons located at the bottom of the window or by pressing the F1 key.

Main Help Window

When a user requests help from a window, IPF displays the main help window, the characteristics of which are:

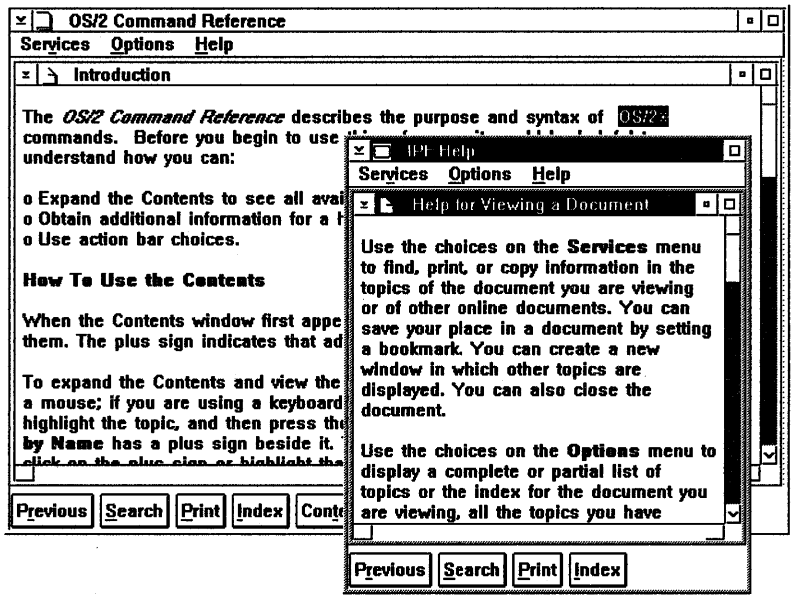
- Menu bar
- Title bar icon
- Title bar
- Maximize button
- Horizontal scroll bar
- Vertical scroll bar
- Push buttons.

The main help window cannot be minimized.

Within the main help window is the help-text window. The help-text window contains the response to the user's request for help. The characteristics of this window are:

- Title bar icon
- Title bar (shows title of the selected help window)
- Maximize button
- Hide button
- Horizontal scroll bar
- Vertical scroll bar.

The windows shown in the lower right corner of Figure 2-3 are main help and help-text windows.


OS/2 Command Reference
Services Options Help
Introduction
IPF Help
Services Options Help
Help for Viewing a Document
Previous Search Print Index Conte
Previous Search Print Index

The *OS/2 Command Reference* describes the purpose and syntax of OS/2 commands. Before you begin to use
understand how you can:

o Expand the Contents to see all avai
o Obtain additional information for a h
o Use action bar choices.

How To Use the Contents

When the Contents window first appe
them. The plus sign indicates that ad

To expand the Contents and view the
a mouse; if you are using a keyboard
highlight the topic, and then press the
by Name has a plus sign beside it.

Use the choices on the **Services** menu to find, print, or copy information in the topics of the document you are viewing or of other online documents. You can save your place in a document by setting a bookmark. You can create a new window in which other topics are displayed. You can also close the document.

Use the choices on the **Options** menu to display a complete or partial list of topics or the index for the document you are viewing, all the topics you have

Figure 2-3. An IPF main help window with its help-text window.. The title "Help for Viewing a Document" that appears in the title bar was created by the author of the help-text window.

When the main help window is first opened, its position is such that it covers the smallest part of the online document or application window as possible. The help-text window is opened at its maximum size within the main help window.

However, when the main help window is opened, it can be moved and resized, as can the help-text window. If the user makes the help-text window larger or smaller, the text within the window is reformatted to fit the new window size.

Selection Lists

Selection lists appear when any of the menu bar choices are selected:

- **Viewed pages**, under **Options**
- **Contents**, under **Options**
- **Help index**, under **Help**.

Figure 2-4 shows the Viewed Pages selection list.

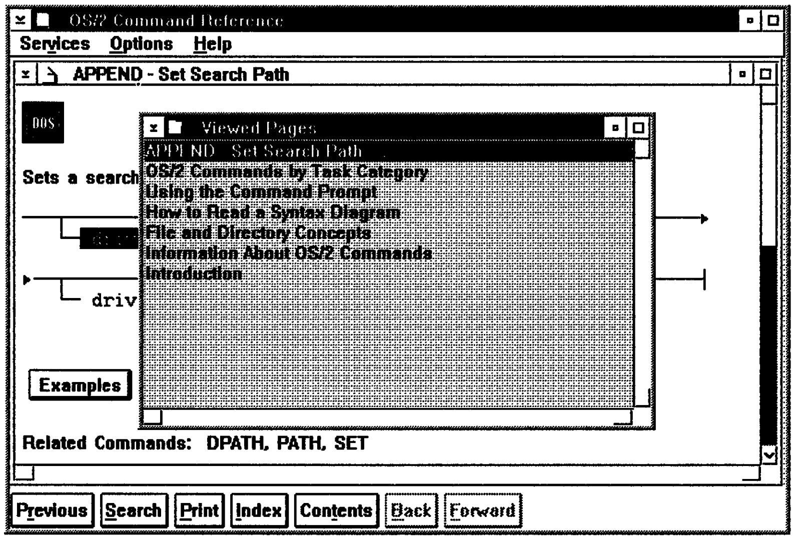


Figure 2-4. Viewed Pages selection List.

Search results also are displayed in a selection list. Selection lists differ from the help-text window in that they can be closed, either by selecting the **Close** symbol or by pressing the Esc key.

Selecting the hide button from the title bar while a help-text window, Contents window, Viewed Pages window, Index window, or Search results window is displayed, results in the window being replaced by an icon.

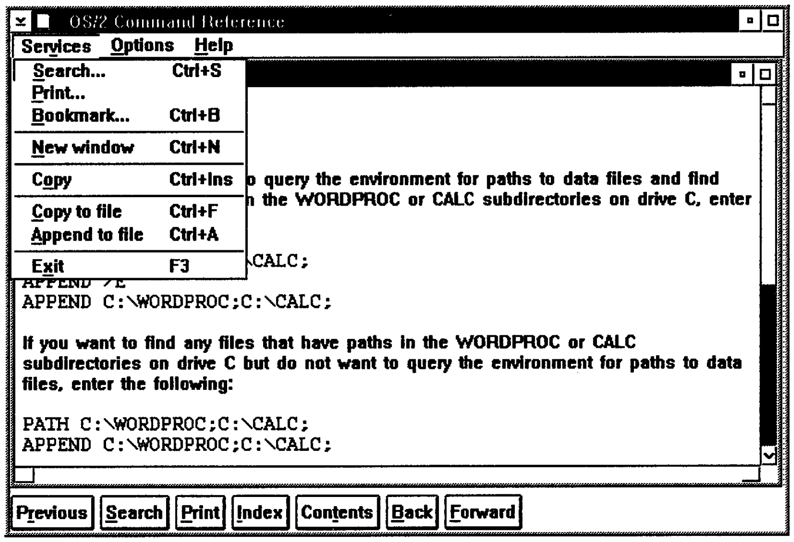
Menu Bar

Figure 2-5 on page 2-5 shows the menu bar that conforms to CUA guidelines. It has the choices **Services**, **Options**, and **Help**.

When **Services**, **Options**, or **Help** is selected, a menu appears with a list of entries that also can be selected. Some entries have an associated shortcut key or key combination. These are shown to the right of the menu item.

Services Menu

The Services menu shows a list of available services. (See Figure 2-5 on page 2-5.)



o query the environment for paths to data files and find
n the WORDPROC or CALC subdirectories on drive C, enter

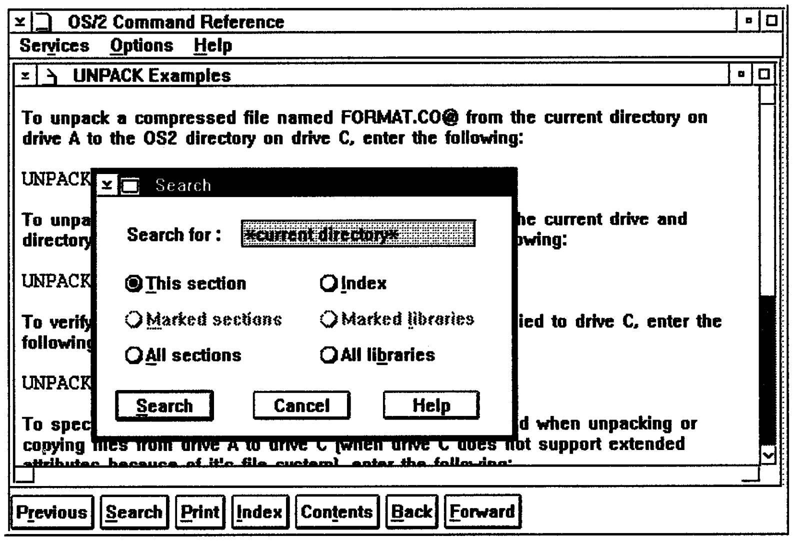
```
\CALC;
APPEND C:\WORDPROC;C:\CALC;
```

If you want to find any files that have paths in the WORDPROC or CALC subdirectories on drive C but do not want to query the environment for paths to data files, enter the following:

```
PATH C:\WORDPROC;C:\CALC;
APPEND C:\WORDPROC;C:\CALC;
```

Figure 2-5. Services Menu

Search: This choice, and the Search push button, display the window shown in Figure 2-6. The user can type a text string consisting of letters, numbers, blank spaces, and special characters, then select any of the choices to search for the text string.



To unpack a compressed file named FORMAT.CO@ from the current directory on drive A to the OS2 directory on drive C, enter the following:

UNPACK

To unpa … he current drive and directory … owing:

UNPACK

To verify … ied to drive C, enter the following

UNPACK

To spec … d when unpacking or copying … ot support extended

Figure 2-6. Search Window

Global file-name characters can be used with the text string; for example:

```
*current directory*
```

The global file-name character, in this case an asterisk, automatically finds all possibilities of the *current directory* string.

Following are descriptions of Search window choices:

This section
: Searches the currently displayed help-text window and highlights all occurrences of the search string that are found.

Marked sections
Searches the online-document windows or help windows whose titles were marked in the Contents window. IPF does not search unmarked secondary windows, or windows attached to the marked window by hypertext links. Before selecting **Marked sections**, the user must select **Contents** then mark the help titles to be searched. If no help titles are marked, the **Marked sections** choice is dimmed.

Sections are marked with the mouse by pressing and holding the Ctrl key then clicking mouse button 1. Sections are marked with the keyboard by using the cursor keys to highlight the item and then pressing the spacebar. The same key sequences are used to unmark the selection.

If the search is successful, IPF displays a list of the window titles where the text string was found. The search string is shown in the title bar of the search results window.

All sections
Searches the entire help library or online document and displays a list of the window titles where occurrences of the search string were found. The search string is shown in the title bar of the search results window.

Index
Searches the index and displays a list of index entries in which the text string was found.

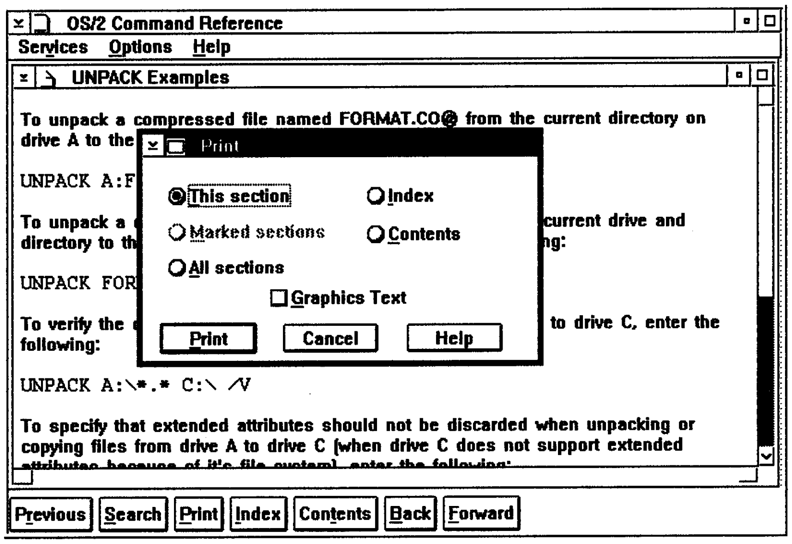
If no search string is entered, this choice displays an alphabetic list of all index topics in the help library or online document.

Marked libraries
Searches selected help libraries or online documents. The user must be in an active window of an online document and follow this procedure:

1. Select **Options** then **Libraries**.
2. Mark one or more libraries.
3. Select **Services**, then **Search**, then **Marked libraries**.
4. Select the **Search** push button.

All libraries
Searches all help libraries or online documents and displays a list of the window titles where the text string was found. The search string is shown in the title bar of the window.

Print: This choice, and the Print push button, display the window shown in Figure 2-7 on page 2-7. The output is the text within the window. The user can select any of the choices to print online information.



OS/2 Command Reference

Services Options Help

UNPACK Examples

To unpack a compressed file named FORMAT.CO@ from the current directory on drive A to the

UNPACK A:F

To unpack a … current drive and directory to th… ng:

UNPACK FOR

To verify the … to drive C, enter the following:

UNPACK A:*.* C:\ /V

To specify that extended attributes should not be discarded when unpacking or copying files from drive A to drive C (when drive C does not support extended

Print

[✓ This section] Index

Marked sections Contents

All sections

☐ Graphics Text

Print Cancel Help

Previous Search Print Index Contents Back Forward

Figure 2-7. Print window

Following are descriptions of Print window choices:

This section
Sends the contents of the current window to be printed.

Marked sections
Sends the sections whose titles were marked in the Contents window to be printed. Before selecting this choice, the user must select **Contents** and mark the titles to be printed. If no titles are marked, **Marked sections** is dimmed.

Sections are marked with the mouse by pressing and holding the Ctrl key then clicking mouse button 1. Sections are marked with the keyboard by using the cursor keys to highlight the item and then pressing the spacebar. The same key sequences are used to unmark the selection.

All sections
Sends the entire online document or help library to be printed.

Index
Sends the help-index or online document index to be printed.

Contents
Sends the help library or online-document contents list to be printed.

Graphics Text
Displays a check mark if a plotter or PostScript** printer are installed on the OS/2 operating system. Removing the check mark could result in unpredictable printing results.

This choice will not have a check mark if any other printer was installed on the OS/2 operating system. Mark the box to print your text in *graphics mode.*

The text that the user selects to be printed is sent to the print spooler so the user can continue to work. This choice does not print artwork or special highlighted text.

Bookmark: This choice displays the window shown in Figure 2-8.

** Trademark of Adobe Systems Incorporated

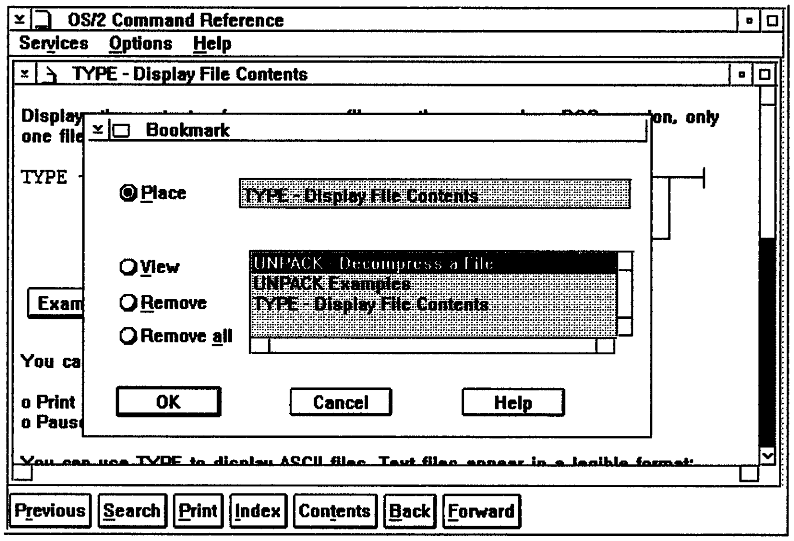


Figure 2-8. Bookmark Window

Place
Saves the user's place in the document being viewed.

View
Redisplays a specific place that was marked.

Remove
Deletes one marked place.

Remove all
Deletes all marked places.

New window: This choice opens a new window for the currently selected item (for example, table of content entry, hypertext link, index list, or search list) so the user can see more than one topic displayed at the same time. The user can resize the new window so the previous window also can be seen, then resize both to view both windows.

New windows are related to online documents. If the user minimizes the document, the new windows are saved. If the user closes the document, the new windows are closed.

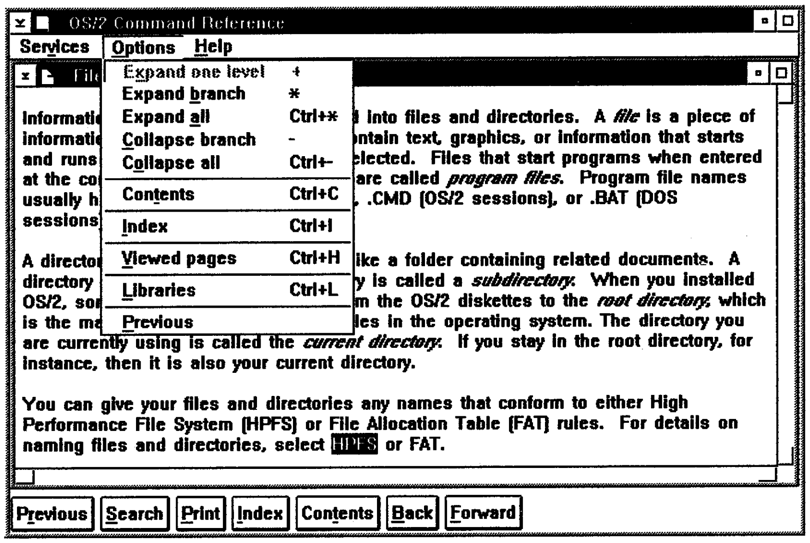
Copy: Copies the window currently being viewed to the system clipboard. The user can then select **Paste** from the menu of the OS/2 System Editor (or any other editor with this capability) to view or edit the information.

Copy to file: Copies the information to the file, TEXT.TMP. This file is placed in the current directory. If TEXT.TMP already exists, it is replaced.

Append to file: Copies the information to the file, TEXT.TMP. This file is placed in the current directory. If TEXT.TMP already exists, the new information is added to the existing information.

Options Menu

Selecting **Options** displays the menu shown in Figure 2-9.



Informati into files and directories. A *file* is a piece of
informati ntain text, graphics, or information that starts
and runs lected. Files that start programs when entered
at the co are called *program files*. Program file names
usually h , .CMD (OS/2 sessions), or .BAT (DOS
sessions

A directo like a folder containing related documents. A
directory y is called a *subdirectory*. When you installed
OS/2, so m the OS/2 diskettes to the *root directory*, which
is the ma les in the operating system. The directory you
are currently using is called the *current directory*. If you stay in the root directory, for
instance, then it is also your current directory.

You can give your files and directories any names that conform to either High
Performance File System (HPFS) or File Allocation Table (FAT) rules. For details on
naming files and directories, select HPFS or FAT.

Figure 2-9. Options Menu

The first five choices are active when the Contents window is active. If the Contents window is not active, these choices are dimmed. These choices control how the table of contents will be displayed.

A tree-structured table of contents is created if more than one heading level is specified with IPF heading tags when the windows are created. (For a description of heading tags, see "Headings" on page 3-4.)

If there are additional entries under a heading, + appears to the left of the entry. When an entry is expanded one level, the next level of entries subordinate to the selected entry is displayed, and the + is replaced by a –. The user can click on the + or – symbols to expand or contract the contents.

Expand one level: Expands the first level subordinate to the selected entry.

Expand branch: Expands all levels subordinate to the selected entry.

Expand all: Displays the entire tree structure of the contents.

Collapse branch: Contracts all levels subordinate to the selected entry.

Collapse all: Displays only the highest level entries in the contents.

Contents: Lists the table of contents for the document you are viewing. You can select from the topics shown. This choice has the same function as the **Contents** push button.

Index: Displays an alphabetic list of the topics in the document. You can select from the index entries shown. This choice has the same function as the **Index** push button.

Viewed pages: This selection displays a list of all windows viewed during the current session. Window titles are listed in the order that windows were viewed. If a window is viewed more than once, its title appears as many times as it was viewed.

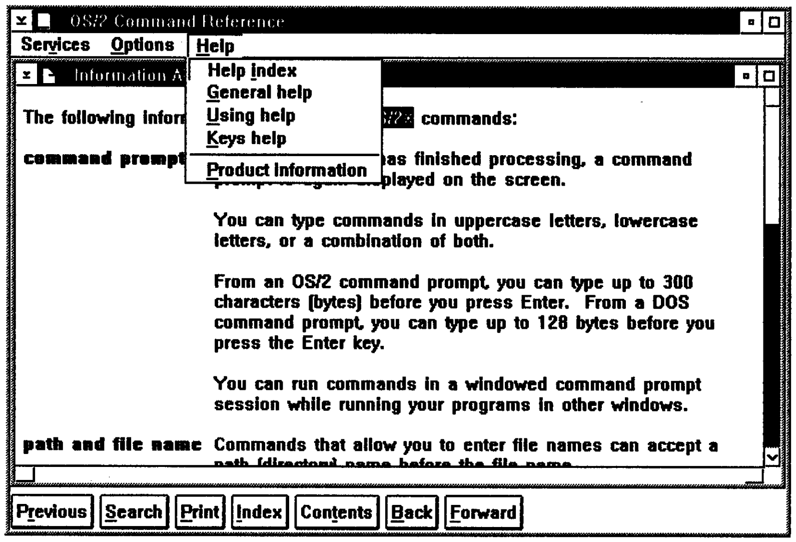
The maximum number of entries that can appear in the Viewed Pages window is 50.

Libraries: Displays a list of online libraries that are available. The libraries are identified by their directory path and file name.

Previous: Displays the previously viewed window. Each time **Previous** is selected, the previously viewed window is displayed, until the first window viewed in the current session is shown. Selecting **Previous** again from a help window, ends the current session. This choice has the same function as when the user selects the **Previous** push button; or presses the Esc key.

Help Menu

Selecting **Help** displays the menu shown in Figure 2-10.



The following infor commands:

command prompt as finished processing, a command played on the screen.

You can type commands in uppercase letters, lowercase letters, or a combination of both.

From an OS/2 command prompt, you can type up to 300 characters (bytes) before you press Enter. From a DOS command prompt, you can type up to 128 bytes before you press the Enter key.

You can run commands in a windowed command prompt session while running your programs in other windows.

path and file name Commands that allow you to enter file names can accept a

Figure 2-10. Help Menu

Help index: Displays an alphabetic list of topics for which help information is provided.

General help: Displays general information describing the user interface and how to access it.

Using help: Displays a list that describes the different help information that is available to users of the OS/2 operating system.

Keys help: Displays information describing key assignments for the application.

Note: Providing Keys help is the responsibility of the application programmer. A simple help window can be created that lists each key combination assigned to an application function, and a brief description of what the function does.

Product Information: Displays copyright information. Providing product information also is the responsibility of the application programmer.

Tutorial: This choice is included in the Help menu if your application tells IPF that it has created a Presentation Manager tutorial application. This choice has the same function as the **Tutorial** push button.

When the user selects **Tutorial**, IPF sends a message to the application that the selection has been made. The application then starts the tutorial.

Push Buttons

Push buttons provide users with a fast and easy way to access commonly used IPF functions. When a user selects a push button, the action represented is carried out immediately. IPF provides the following set of push buttons:

Previous: This push button lets the user see information from the previously viewed window. This is the same function as when the user selects **Options** then **Previous** from the menu bar, or presses the Esc key.

Search: This push button displays a window that lets the user search for a word or phrase. This is the same function as when the user selects **Services** then **Search** from the menu bar.

Print: This push button displays a window that lets the user print one or more topics. This is the same function as when the user selects **Services** then **Print** from the menu bar.

Index: This push button displays an alphabetic list of the index topics in a help library or an online document. This is the same function as when the user selects **Options** then **Index** from the menu bar.

Contents: This push button displays the Contents window. This is the same function as when the user selects **Options** then **Contents** from the menu bar.

Back: This push button displays the previous page in the table of contents hierarchy.

Forward: This push button displays the next page in the table of contents hierarchy.

Tutorial: This push button is included only if a tutorial was specified in your application. This is the same function as when the user selects **Help** then **Tutorial** from the menu bar.

Chapter 3. Starting with the Tag Language

As the author of online information, you can use the IPF tag language to define various characteristics of text format. You also can use tags to define characteristics of the window in which the text is displayed.

There are 45 tags (excluding symbols) that make up the IPF tag language. The tags are *mnemonic*, making it easy to associate them with their functions. However, before you can begin to use this language, you need to familiarize yourself with the elements that make up the syntax of the tags, and special rules that govern the use of the tags.

Syntax Conventions

Each tag must start with a colon (:) and end with a period (.). (The period is also known as a *delimiter*.) For example, the tag for a paragraph is:

:p.

A tag indicates how the text that immediately follows it is to be processed. In the following example, the text immediately after the paragraph tag (**:p.**) is the actual text that is displayed in the window, and it will begin a new paragraph.

:p. There are fewer than 1200 manatees...

End Tags

Some tags require end tags. An end tag is **e** immediately followed by the tag. For example, the end tag for the **:userdoc.** tag is:

:euserdoc.

Most of the tags that have end tags affect text format or appearance. The end tag tells the IPF compiler to end the operation associated with the tag. If you forget an end tag, the compiler displays an error message.

Nested Tags

Nested tags are tags within other tags. For example, a common way of presenting information is in a list form; a tag begins the list, another tag identifies each list item, and yet another tag ends the list. An example of the tagging for a simple list follows:

:sl.
:li. List item 1
:li. List item 2
:li. List item 3
:esl.

The list-item tag (**:li.**) is required for each item in the list. The **:li.** tags are *nested* between the **:sl.** tag and the **:esl.** tag.

Note: After paragraph and heading tags, you will probably use list tags most often. IPF provides general-purpose lists (simple, unordered, and ordered), and special-purpose lists (definition and parameter).

Text Strings

Some tags have text strings associated with them. The string can immediately follow the tag, or it can start the line immediately following the tag. For example, the tagging for the title bar of a window is **:h1.** (one of the heading tags) and a text string, which is called a *title string*. You can enter it like this:

:h1. Save the Manatee

or like this:

:h1.
Save the Manatee

Attributes

A tag also can have one or more *attributes*. An attribute contains additional information about a tag's operation. The attribute has a name, which may have a value or keyword assigned to it.

In the following example, the attribute **res=** specifies a window identifier.

:h1 res=001. Save the Manatee

In this case, **001** is the assigned value. The value assigned to a **res=** attribute must be unique for each heading tag. This value also will be the identifier for linking to the heading from elsewhere in the information. The concept of linking is described in "Hypertext Links" on page 5-2.

Notice that the period follows the attribute, not the heading tag. The period always follows the last attribute in the tag.

You can specify many attributes in one tag, and they can extend over several lines. However, you cannot split an attribute. For example, you cannot put the **res=** attribute of the heading tag on one line, and its value, **001**, on the next line.

Some attributes are optional and have a default (an assumed value) if they are not included with the tag; other attributes are required. Tag attributes can be specified in any order.

As mentioned, some attributes are required. For example, if you are creating a help library, the **res=** attribute of a heading tag is required as a window identifier (see "Window Identifiers" on page 5-1).

An attribute also can have a keyword associated with it. For example, an attribute of the **:color.** tag is **fc=** (foreground color), which is used to specify the color of the text. Its value can be equal to any of the following keywords:

- DEFAULT
- BLUE
- CYAN
- GREEN
- NEUTRAL
- RED
- YELLOW.

Not all attributes have values or keywords. For example, if you want a simple list with no blank lines between the list items, add the **compact** attribute to the simple-list tag (**:sl.**). In the following example, notice the **compact** attribute stands by itself:

:sl compact.
:li. List item 1
:li. List item 2
:li. List item 3
:esl.

Attribute Values with Blank Spaces: If an attribute value includes blank spaces, the value must be enclosed in single quotes. For example:

:font facename='Tms Rmn'.

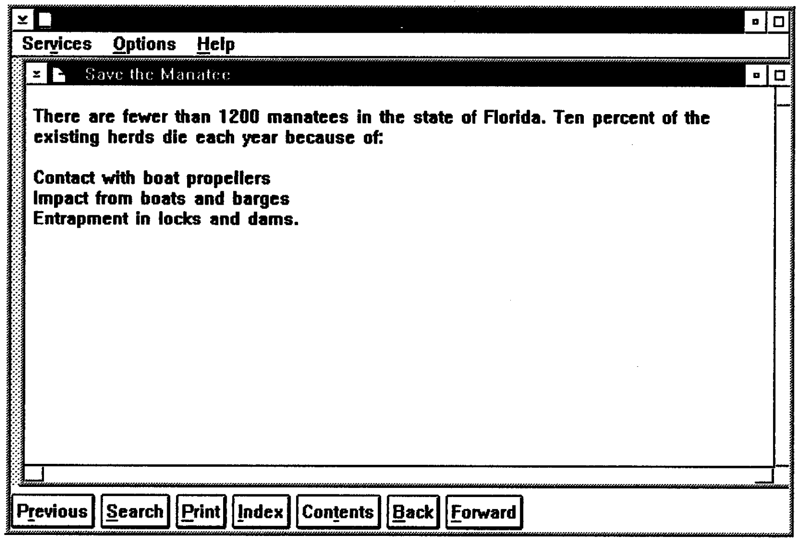
Notice that the value has initial capitals. For this particular case, they are required; otherwise, the IPF compiler will not recognize them as valid values.

Using some of the tags described thus far, you could produce a source file like this:

```
:userdoc.
:h1 res=001.Save the Manatee
:p.
There are fewer than 1200 manatees in the state of Florida.
Ten percent of the existing herds die each year
because of:
:sl compact.
:li.Contact with boat propellers
:li.Impact from boats and barges
:li.Entrapment in locks and dams.
:esl.
:euserdoc.
```

The output produced from the source file is an OS/2 standard window.

The menu-bar choices, **Services, Options,** and **Help** are provided automatically by IPF. The title-bar line, "Save the Manatee," is generated by the **:h1.** tag. The viewing area of the window displays the formatted information.



There are fewer than 1200 manatees in the state of Florida. Ten percent of the existing herds die each year because of:

Contact with boat propellers
Impact from boats and barges
Entrapment in locks and dams.

The best way to learn about tags is to study the examples provided in the following sections, then create some windows of your own.

Symbols

You use symbols to produce characters that cannot be entered from the keyboard. A symbol begins with an ampersand (&) and is followed by the symbol name and a period. For example, to produce a square bullet, which looks like this:

■

Enter the symbol like this:

&sqbul.

If you want the ampersand character (&) to appear in text, define it as the symbol, **&amp**. Otherwise, the IPF compiler tries to interpret whatever text follows the ampersand character as the name of a symbol, and will return the error message, `Invalid symbol.`

Symbols are case-sensitive. That is, if you do not type them exactly as the appear in the symbols table (see Chapter 14, "Symbols") you could get either the message, `Invalid Symbol`, or a symbol different from the one you want.

Note: The symbols table is also available online when you install the Online Information component of the Developer's Toolkit for OS/2 2.0.

Headings

Perhaps the most versatile tag is the heading tag. Heading tags enable information to be displayed in windows, control entries in the Contents window, control placement of push buttons in a window, and define the shape and size of windows. With IPF, you can specify six levels of headings, **:h1** through **:h6**. For information about default heading levels that start a window and place entries in the table of contents window, see page 3-6.

Displaying Window Titles

Every heading tag that starts a window must have an associated text string. The text string becomes the window title and appears in the title bar of the window. The window title also becomes an entry in the Contents window, which lists the headings of all topics in an online document.

For a window that occupies the full width of the screen, the maximum length of a text string, including spaces and blanks, is 70 characters. A narrower window requires a shorter text string. The text string can be on the same line as the heading tag, or at the beginning of the next line.

The following example shows the tagging for the first three heading levels, with a paragraph following each heading.

```
:userdoc.
:title.An Online Document
:h1.First Heading Level
:p.
This window is defined by a first-level heading tag.
:h2.Second-Level Heading
:p.
This window is defined by a second-level heading tag.
:h3.Third-Level Heading
:p.
This window is defined by a third-level heading tag.
:euserdoc.
```

The Contents window for the formatted output shows the three heading-level entries.

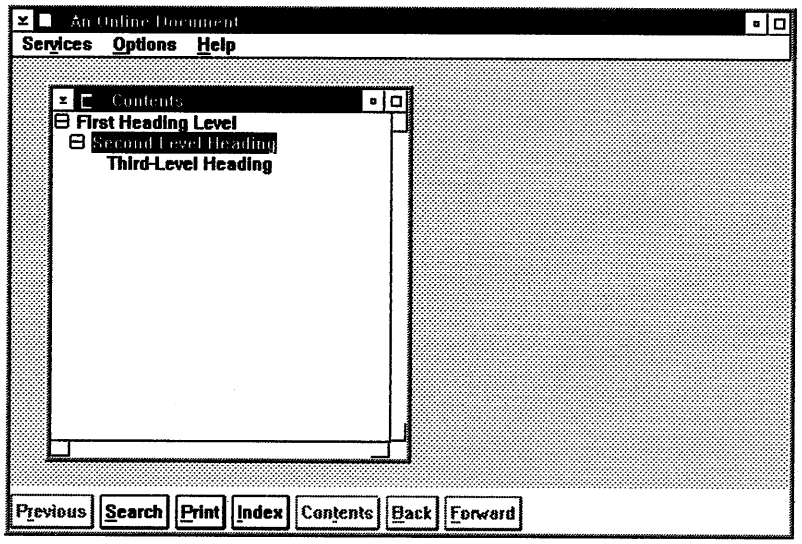


Figure 3-1. A Contents Window

Hiding Window Titles

If you do not want a title to appear in the Contents window, use the **hide** attribute. The heading definition would be entered like this:

```
:h3 hide.
Another Third-Level Heading
```

Note: You source file must contain at least one heading tag without the **hide** attribute.

Controlling Entries in the Contents Window

The following example shows some tagging that will control what entries appear in the Contents window, as well as what headings will start windows.

```
:userdoc.
:docprof toc=12.
:h1.Heading Levels
:h2.Second-Level Heading
:p.
This window is defined by a heading-level 2 tag.
:h2.Second-Level Heading
:p.
This window also is defined by a heading-level 2 tag.
:p.
:h3.Third-Level Heading
:p.
Because the :docprof. tag at the beginning of the file
specifies that only heading levels 1 and 2 can be entries in the
Contents window (toc=12), the preceding "Third-Level Heading"
and THIS text, which follows it, become part of the
window defined by the preceding heading-level 2 tag.
:h2 toc=123.Another Second-Level Heading
:p.
The heading-level 2 tag for this window contains
a toc=123 specification.
:h3.Third-Level Heading
:p.
Because the toc=123 in the preceding heading-level 2
tag overrides the toc=12 in the :docprof. tag, this
heading-level 3 tag defines a new window and creates a
Contents entry.
:euserdoc.
```

Unless otherwise specified, the default set of heading tags that create entries in the Contents window and define the start of windows are **:h1.**, **:h2.**, and **:h3.**. To change this default, specify a numeric sequence with the table of content attribute (**toc=**) of the **:docprof.** tag. The **:docprof.** tag controls the heading levels displayed in the Contents window. The sequence must begin with level 1 and cannot skip a level in the descending hierarchy. For example, the **:h4.**, **:h5.** and **:h6.** tags do not start separate windows, but control the appearance of the text of the window unless you specify:

> **docprof toc**=123456.

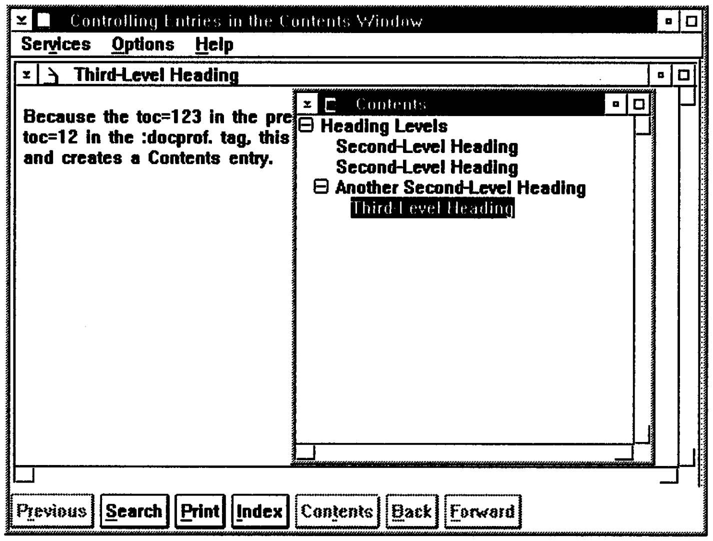
To specify that only heading levels 1 and 2 are to define windows and appear as entries in the Contents window, the following tag was used:

> **:docprof toc**=12.

The value specified for the **toc=** attribute remains in effect for all the heading definitions in the file. You can override it by specifying another value for the **toc=** attribute in a heading definition. The new value is then in effect for the rest of the headings in the file, or until overridden in another heading definition.

In the preceding example, the **toc=** attribute of the **:docprof.** tag is overridden by the **toc=** attribute of a heading tag.

The next example shows the results of the tagging. Notice the effect of including a heading level that is lower in the hierarchy than the range of heading levels specified with the **:docprof.** tag.



Because the toc=123 in the pre
toc=12 in the :docprof. tag, this
and creates a Contents entry.

Figure 3-2. Contents Window with Displayed Third-Level Heading Window

When the file is viewed, the **:h3.** title and the text following it are included as part of the window defined by the preceding **:h2.** tag.

Special Rules

Sequential Coding for Heading Tags: Headings for a series of windows must always start with **:h1.** and proceed in sequence. That is, you cannot have **:h1.** followed by **:h3.**. However, you can follow **:h3.** with **:h1.**.

Source File Size between Heading Tags: Do not exceed 16000 words, numbers, and punctuation marks between two consecutive heading tags in your source file. This includes blank spaces, but does not include commented lines (see "Comment" on page 3-15). If the source file exceeds this limit, the compiler will generate an error message. To correct the error, use another heading tag.

For more information about heading tags and attributes that define characteristics of windows, see Chapter 6, "Customizing Windows."

Push Buttons

Important

This section is for both the information author and the application programmer. Because push buttons must be considered when tagging all online information, an introduction to push buttons is necessary in this early chapter. No attempt is made to clarify the programming information; however, when possible, references are given to corresponding programming sections.

Push buttons provide users with a fast and easy way to access commonly used IPF tasks. When a user selects a push button, the action represented by the text on the push button is carried out immediately. Push buttons are displayed in a window called a *control area*. A control area can be defined within the IPF coverpage window, or the IPF text window (the child of the coverpage window), or both. For information about the IPF coverpage, see page 3-8. For information about the child of a coverpage, see "The Coverpage Window" on page 9-3.

IPF provides one set of push buttons for online documents and another set for help windows.

For online documents, the set of push buttons consists of:

Previous
Search
Print
Index
Contents
Back
Forward
Tutorial (only if a tutorial is available).

For help windows, the set of push buttons consists of

Previous
Search
Print
Index
Tutorial (only if a tutorial is available).

Figure 3-3 shows an online document with a set of push buttons in the control area of the the coverpage window (the default control area).

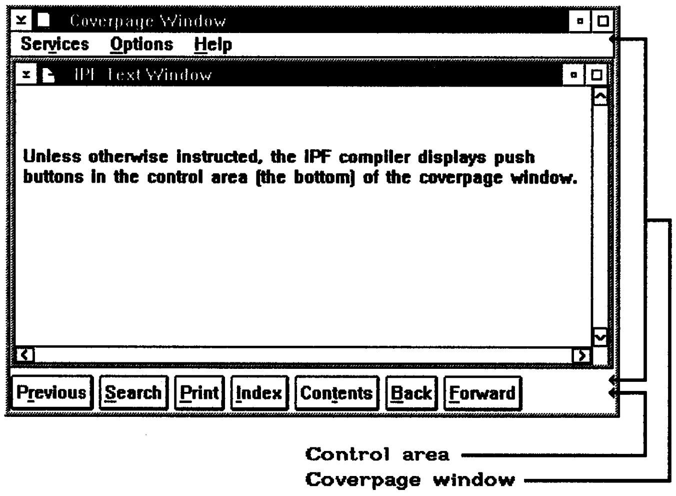
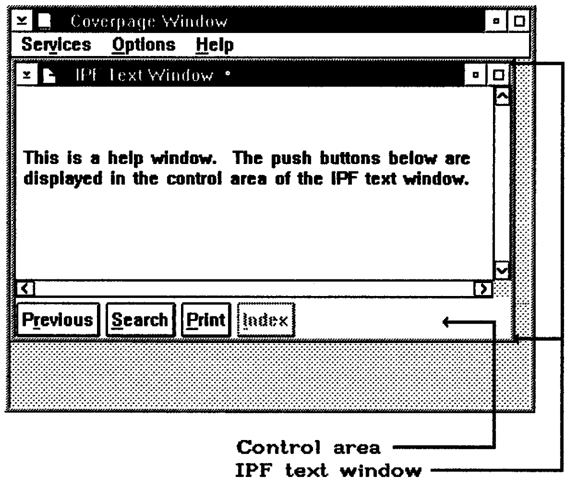

Unless otherwise instructed, the IPF compiler displays push buttons in the control area (the bottom) of the coverpage window.

Figure 3-3. The IPF default window for push buttons. These push buttons appear in the control area of the coverpage window.

Notice the difference in Figure 3-4 on page 3-9. This example shows a help window with a set of push buttons in the control area of the IPF text window.



This is a help window. The push buttons below are displayed in the control area of the IPF text window.

Figure 3-4. A help window. These push buttons were defined in the control area of the IPF text window. If the user changes the size of the window, the push buttons in the control area will wraparound onto the next line. The push buttons cannot be clipped or scrolled horizontally, because the control area is not part of the scrollable area of the IPF text window.

Tagging Example for the Default Set of Push Buttons

The following example shows the minimum tagging required for an online document that is to have a control area with the default set of push buttons displayed in the coverpage window.

```
:userdoc.
:title.Coverpage Window
:h1.IPF Text Window
:p.Text goes here.
:euserdoc.
```

Notice no extra tagging is necessary.

Specifying Push Buttons for the Control Area of a Window

The control area tag (**:ctrl.**) specifies where push buttons are to be displayed, and which push buttons you want displayed. When specifying a control area, always precede the tagging with **:docprof.**, then imbed **:ctrl.** between the control-area definition tag (**:ctrldef.**) and **:ectrldef.**. For example:

```
:docprof toc=123.
:ctrldef.
:ctrl.
:ectrldef.
```

Attribute Values for the Control Area of a Window

The **controls=** attribute of **:ctrl.** identifies the push buttons that you want in the control area of a window. Push buttons are displayed in the order in which they are defined. Values that can be specified are:

SEARCH	Specifies the "Search" push button. When selected, this push button displays a window that lets the user search for a word or phrase.
PRINT	Specifies the "Print" push button. When selected, this push button displays a window that lets the user print one or more topics.
INDEX	Specifies the "Index" push button. When selected, this push button displays an alphabetic list of the topics in the document.
CONTENTS	Specifies the "Contents" push button. When selected, this push button displays the Contents window.
ESC	Specifies the "Previous" push button. When selected, this push button lets the user see information from an earlier request.
BACK	Specifies the "Back" push button. When selected, this push button displays the previous page in the table of contents hierarchy.
FORWARD	Specifies the "Forward" push button. When selected, this push button displays the next page in the table of contents hierarchy.

Note: A value for the Tutorial push button is not provided because it is displayed automatically if a tutorial exists.

Both the **page** and **coverpage** attributes of **:ctrl.** affect where push buttons are displayed. For example, you use **page** to specify that push buttons are to be in the IPF text window; similarly, you use **coverpage** to specify that push buttons are to be in the IPF coverpage window.

A control area also can have a value associated with it. The **ctrlid=** attribute specifies the value, which can be either alpha or alphanumeric, and is referred to by a heading tag. In the following example, **ctrlid=** specifies a window identifier, and instructs the compiler to display the PREVIOUS, FORWARD, and BACK push buttons in the control area of the coverpage window:

```
:docprof toc=123.
:ctrldef.
:ctrl ctrlid=newl controls='ESC FORWARD BACK' coverpage.
:ectrldef.
```

Conversely, the following example shows the tagging for an online document that will display the PREVIOUS, FORWARD, and BACK push buttons in the control area of an IPF text window.

```
:docprof toc=123 ctrlarea=page.
:ctrldef.
:ctrl ctrlid=newl controls='ESC FORWARD BACK' page.
:ectrldef.
```

Notice the **:ctrlarea=page** attribute of **:docprof.**. When the IPF compiler encounters **ctrlarea=page.**, it defines the control area as the IPF text window and removes the push buttons from the control area of the coverpage window. You must ALWAYS specify the **:ctrlarea=** attribute in **:docprof.** when overriding the default control area in a window.

Other values for **:ctrlarea=** are:

coverpage	Identifies the control area as the bottom of the coverpage window. This is the default value.
both	Specifies both the control area within an IPF text window, and the coverpage window.
none	Specifies that you do not want a control area. (You do not want push buttons.)

You can define more than one control area with different sets of push buttons for the IPF text window; however, only one set of push buttons can be defined for the coverpage window.

Controlling the Display of Push Buttons in Designated Windows

Suppose your document consisted of 100 windows, and you wanted only one window to display push buttons in the control area of the IPF text window. The **ctrlarea=** attribute of a heading tag specifies which control area in a window you want to display push buttons. You would tag your source file as follows:

```
:docprof ctrlarea=none.
:
:h1 ctrlarea=page.One Window
```

When **ctrlarea=** is encountered in a heading tag, it overrides the **ctrlarea=** attribute specified by **:docprof.**

Disabling the Display of Push Buttons

The following example shows the minimum tagging for an online document without push buttons.

```
:userdoc.
:title.Coverpage Window Title
:docprof toc=123 ctrlarea=none.
:h1.IPF Text Window
:p.Text goes here.
:euserdoc.
```

Author-Defined Push Buttons

IPF also supports author-defined pushbuttons. For example, you can define a push button for "Examples" that can be included in the control area of a coverpage or IPF text window. When an author-defined push button is selected, the message HM_NOTIFY is sent to the application or communication object. It is the responsibility of the application or communication object to respond to this message. For information about communication objects, see "Application-Controlled Windows" on page 9-1.

The push button tag (**:pbutton.**) defines author-defined pushbuttons. This tag must be imbedded within the **:ctrldef.** and **:ectrldef.** tags, and it must precede the **:ctrl.** tag.

The following example shows how to override the default set of push buttons in the coverpage window with a set that consists of Search, Index, Previous, and Example.

```
:userdoc.
:docprof toc=123 dll='example.dll' objectname='xmpbutton'.
:ctrldef.
:pbutton id=xmp res=001 text='~Example'.
:ctrl ctrlid=newl controls='SEARCH INDEX ESC XMP' coverpage.
:ectrldef.
```

Notice that a dynamic link library (DLL) is required to support the function you want to provide with an author-defined push button. For more information, see ":pbutton (Push Button)" on page 13-55.

About the Tutorial Push Button

When the Tutorial push button is selected, the message HM_TUTORIAL is sent to the application or communication object. This is the same message that is sent when the **Tutorial** choice is selected from the Help pull-down, or when the **tutorial** attribute is specified with the heading tag.

The tutorial push button is included only if a tutorial was specified in the initialization structure (HMINIT) or with the tutorial attribute in a heading tag.

Indexing

IPF provides an index for both online documents and help windows from the following tags.

:i1.

:i2.

The **:i1.** tag creates a *primary entry*, which means the entry is at the first level. The **:i2.** tag provides a secondary entry to the primary one.

Index entries are imbedded in the text of a window. You should create at least one index entry for each window, using the **:i1.** tag. The text of an index entry must be on the same line as the tag.

You form an index for online documents and help windows the same way. For example, to create the index entry:

```
copy program
```

use the following tagging

```
:i1.copy program
```

To create two levels of index entries, you use the **:i1** tag with the **id=** attribute, and the **:i2.** tag, with the **refid=** attribute. Here is how to do it.

1. Create the primary index entry and give it an identifier; for example:

   ```
   :i1 id=prnt.printers and plotters
   ```

2. Create the secondary index entries that will be listed under the primary index entry, and refer to the identifier of the primary entry; for example:

   ```
   :i2 refid=prnt.change printer
   :i2 refid=prnt.add printer
   :i2 refid=prnt.printer properties
   ```

When an **:i1.** tag has an identifier that is referred to by **refid=** attributes of **:i2.** tags, the **:i1.** tag must precede the **:i2.** tags in the file. Index entries can be located in any of the windows defined in your source file; however, they cannot be in a footnote.

After your source file is compiled and the user selects **Index** from the **Options** menu, or the **Index** push button, the index entries look like this:

```
printers and plotters
   add printer
   change printer
   printer properties
```

Master Help Index

The Master Help Index is a collection of index entries from the OS/2 help-file library. Its primary purpose is to provide a quick way to help topics. With it, you can provide such features as:

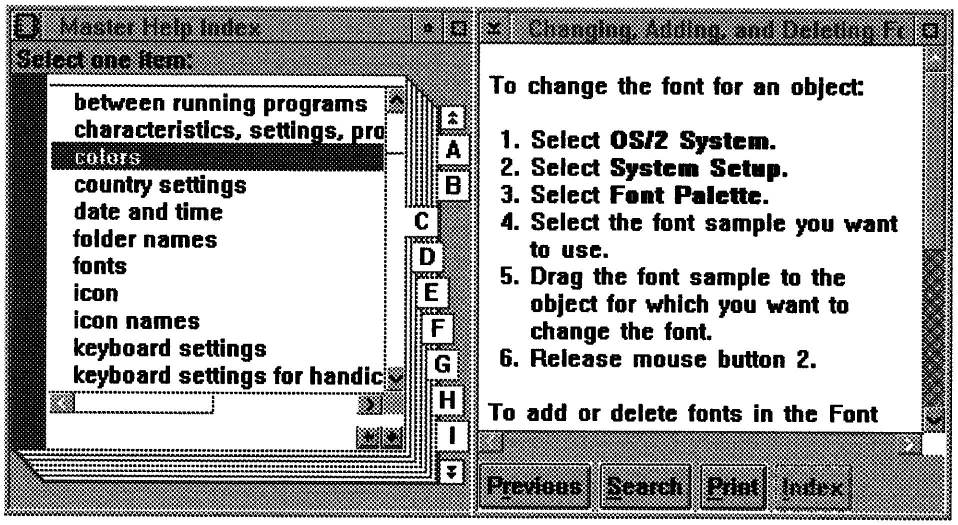
- A side-by-side window design that lets the user scan index entries on one side, then display the help-text information on the other side.
- A menu you can use to create a new Master Help Index or add index entries to the existing one. With this menu, the user can search the Master Help Index database, print help-text windows, or request assistance.

Master Help Index entries are global, which means they can be accessed by more than one application program, so system resources can be conserved.

Using the Master Help Index

When the user selects the Master Help Index from the Workplace, it opens to display an alphabetic list of entries within a bound notebook. Alphabetic tabs lay vertically along the right edge of the notebook. Selecting one of these tabs displays the index entries that match the letter of the tab; for example, if the user selects the "C" tab, the first entry beginning with the letter C is moved to the top of the list. Tabs are displayed only if an index entry exists with that letter. For example, if there is no index entry beginning with the letter "W," IPF does not create a "W" tab for the master index.

When the user double-clicks on an entry in the list, the associated help-text window appears next to the entries list. Figure 3-5 on page 3-14 shows an example of the Master Index window and the opened help-text window, "Changing and adding fonts."



between running programs
characteristics, settings, pro
colors
country settings
date and time
folder names
fonts
icon
icon names
keyboard settings
keyboard settings for handic

To change the font for an object:

1. Select **OS/2 System**.
2. Select **System Setup**.
3. Select **Font Palette**.
4. Select the font sample you want to use.
5. Drag the font sample to the object for which you want to change the font.
6. Release mouse button 2.

To add or delete fonts in the Font

Figure 3-5. The Master Help Index

Creating Entries for the Master Help Index

The **global** attribute of the **:i1.** and **:i2.** tags identifies index entries as candidates for the Master Help Index. Good candidates are pointers to procedural and conceptual topics. For example, a simple master index entry for conceptual information about batch files would look like this:

:i1 global.batch files, creating

When referring to an **:i1.** tag, use the **global** attribute in both the **:i1.** and **:i2.** tags. For example:

:i1 id=copy **global.**copying

:i2 refid=copy **global.**help topics

:i2 refid=copy **global.**document topics

When the IPF compiler encounters **global** attributes, it creates an alphabetic list, which can then be accessed by selecting **Master Help Index** from the Workplace.

Index-Synonyms

As a way of helping the user search for index entries by using synonyms, IPF provides the index-synonym tag (**:isyn.**). This tag requires the **root**= attribute. With these, you can specify synonyms that will be associated with primary index entries. The **:i1.** tags for these primary entries require a **roots**=' ' attribute that associates the entry with the synonyms.

For example, assume you have the following entries in your file:

:isyn root=copy.
copy copying duplicate duplicating
:isyn root=folder.
folder folders document documents
:i1 roots='copy folder'.
copying a document

The **roots**=' ' attribute of the **:i1.** tag associates "copying a document" with the synonyms of the **root**= attributes of the two **:isyn.** tags.

Now if a user, when requesting a search of the index, specifies any of the words in either of the two **:isyn.** entries, the search results will include all **:i1.** entries that contain the specified word, as well as any **:i1.** entries that have been associated with the word by a **roots=** attribute.

For example, the user enters "duplicating" in a search request. When the search is completed, one of the entries in the search results window is

```
copying a document
```

Control Words

In addition to tags, you can include *control words* in your source files to request special processing from the IPF compiler. A control word is placed at the beginning of a line, and starts with a period (.).

The IPF compiler recognizes the following control words:

.im *filename*	Imbed this file in the current file.
.*	Treat this line of text as a comment and do not interpret.
.br	Start a new line of text.

Imbed

The IPF compiler can produce a single output document by processing one master source file that *imbeds* other source files. The imbed control word (**.im**) sends a signal to the compiler to process each file in the sequence listed in the master file.

This process is most often associated with online documents. A portion of the master file for the online *IPF Reference* looks like this:

```
:userdoc.
⋮
.im ipfcch01.ipf
.im ipfcch02.ipf
.im ipfcch03.ipf
⋮
```

If you are imbedding files, the source file that begins with the **:userdoc.** tag is considered the master file. The imbedded files cannot have **:userdoc.** and **:euserdoc.**.

Comment

Occasionally, you might want to insert comments in your source file solely for the purpose of providing information. The .* enables you to do this. Any text on the same line as this control word is ignored by the compiler. For example, the compiler would recognize the following lines as comment lines and ignore them.

```
.**********************************
.* This file contains the
.* introduction to IPF.
.***********************************
```

Break

The break control word (**.br**) interrupts the display of text on a line, and continues it on the next line. The break control word must be the only entry on the line. For example, assume the source file has the following lines.

```
:p.These words
appear on
the same line.
.br
These words
.br
do not.
```

The output looks like this:

```
These words appear on the same line.
These words
do not.
```

If you enter text on the same line as the break control word, the IPF compiler ignores the break control word.

Chapter 4. Displaying Text and Graphics

Once you have defined your window, you need to consider the various ways text can be displayed. This chapter describes how you can use tags and symbols to:

- Highlight text
- Add notes, notices, and lists
- Define tables for a structured display of data
- Illustrate your text with examples, figures, and character graphics
- Control the formatting of lines of text
- Change the font and color of the displayed information
- Set the margins of the text
- Display art.

Highlighted Phrases

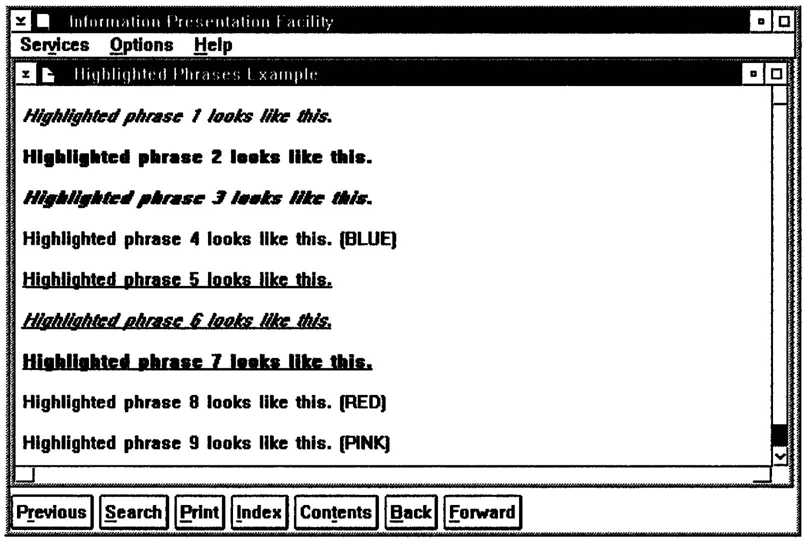
Text can be highlighted by using different type styles or color. There are nine highlighted-phrase tags you can use to emphasize text (**:hp1.** through **:hp9.**). Each tag requires a corresponding end tag (**:ehp1.** through **:ehp9.**).

In the following example, the highlighted phrases are shown as list items in a compact simple list.

Input Example

```
:sl compact.
:li.:hp1.Highlighted phrase 1 looks like this.:ehp1.
:li.:hp2.Highlighted phrase 2 looks like this.:ehp2.
:li.:hp3.Highlighted phrase 3 looks like this.:ehp3.
:li.:hp4.Highlighted phrase 4 looks like this.:ehp4. (BLUE)
:li.:hp5.Highlighted phrase 5 looks like this.:ehp5.
:li.:hp6.Highlighted phrase 6 looks like this.:ehp6.
:li.:hp7.Highlighted phrase 7 looks like this.:ehp7.
:li.:hp8.Highlighted phrase 8 looks like this.:ehp8. (RED)
:li.:hp9.Highlighted phrase 9 looks like this.:ehp9. (PINK)
:esl.
```

Figure 4-1 shows the output produced by these tags.



Highlighted phrase 1 looks like this.

Highlighted phrase 2 looks like this.

Highlighted phrase 3 looks like this.

Highlighted phrase 4 looks like this. (BLUE)

Highlighted phrase 5 looks like this.

Highlighted phrase 6 looks like this.

Highlighted phrase 7 looks like this.

Highlighted phrase 8 looks like this. (RED)

Highlighted phrase 9 looks like this. (PINK)

Figure 4-1. Highlighted Phrases. (See the online *IPF Reference* for the color representation of highlighted phrases 4, 8, and 9.)

The type styles displayed for highlighted phrases correspond to the typeface currently being used by IPF. You can change the typeface to Courier, Helvetica**, or Times New Roman** by using the **:font.** tag. See "Changing Fonts" on page 4-24.

Notes

To include notes in your information, you use a note tag: either **:note.** or **:nt.** (with its corresponding **:ent.**).

The one you use depends on whether your note consists of one paragraph or more than one.

:note.

Use **:note.** for single-paragraph notes. You do not need an end tag.

Following is an example of **:note.** and the resulting output.

Input Example

```
:note.Complete all entry fields before leaving
this window.  If you do not, all your information will be lost.
```

** Helvetica is a trademark of Linotype AG

** Times New Roman is a trademark of Monotype Corporation

Formatted Output

Note: Complete all entry fields before leaving this window. If you do not, all your information will be lost.

:nt.

Use **:nt.** to create notes with more than one paragraph. Remember to end the note with **:ent.**. In the following example, notice how the IPF compiler indents the text for the paragraphs in the note.

Input Example

```
:nt.Complete all entry fields before leaving
this window.  If you do not, all your information will be lost.
:p.If your information is lost, retype it in
the entry fields.
:ent.
```

Formatted Output

Note: Complete all entry fields before leaving this window. If you do not, all your information will be lost.

If your information is lost, retype it in the entry fields.

Another Name for a Note: Both **:nt.** and **:note.** provide the **text=** attribute, so you can substitute your own word or phrase for the word "Note." The following shows the use of this attribute:

Input Example

```
:note text='Reminder'.Complete all
entry fields before leaving this window.
```

Formatted Output

Reminder: Complete all entry fields before leaving this window.

Notices

Two tags enable you to include caution and warning notices in your information. Both tags require end tags.

:caution.

Use **:caution.** to alert users to a risk of possible damage to applications or data.

Input Example

```
:caution.
Be sure to save your data.  If you do not, all data will be lost.
:ecaution.
```

Formatted Output

CAUTION:
Be sure to save your data. If you do not, all data will be lost.

:warning.

Use **:warning.** to alert users to a more severe risk or possible error condition in the system.

Input Example

```
:warning.
The disk contains bad sectors.
:ewarning.
```

Formatted Output

Warning: The disk contains bad sectors.

Place the caution and warning statements before the help information to which they apply so the user is cautioned or warned in advance. You can use the **text=** attribute if you want to use words other than "Caution" and "Warning" with these notices.

Simple List

Simple lists are vertical arrangements of items without any symbol or character preceding the items in the list. Use simple lists when the order of the items are not important.

To create a simple list, use the simple-list tag (**:sl.**) to begin the list, and its corresponding end tag, **:esl.**. Identify each item in the list with a list-item tag (**:li.**).

Input Example

```
:p.Bring the following for lunch&colon.
:sl.
:li.Fruit
:li.Sandwich
:li.Drink
:esl.
```

Formatted Output

Bring the following for lunch:

Fruit

Sandwich

Drink

A Compact Simple List: Use the **compact** attribute to produce a list with no blank lines between the list items.

Input Example

```
:p.Bring the following for lunch&colon.
:sl compact.
:li.Fruit
:li.Sandwich
:li.Drink
:esl.
```

Formatted Output

Bring the following for lunch:

    Fruit
    Sandwich
    Drink

Nested Lists: A nested list is a list that is contained within another list. The following shows the tagging for a simple list nested within another simple list, and the resulting output.

Input Example

```
:p.Bring the following for lunch&colon.
:sl.
:li.Fruit, for example&colon.
:sl compact.
:li.Apple
:li.Orange
:li.Pear
:li.Banana
:esl.
:li.Sandwich
:li.A drink
:esl.
```

Formatted Output

Bring the following for lunch:

    Fruit, for example:

        Apple
        Orange
        Pear
        Banana

    Sandwich

    A drink

Unordered List

Unordered lists are vertical arrangements of items, with each item in the list preceded by a special character, usually the lowercase "o" (called a *bullet*).

Use unordered lists when the order of the items is not important.

To create an unordered list, use the unordered-list tag (**:ul.**) to begin the list, and **:eul.** to end it. Identify each item in the list with **:li.**.

Input Example

```
:ul.
:li.Information typed in Window A will be stored in the
STORES.DAT file in whatever directory you designate.
:li.Information typed in Window B will be stored in the
SALES.DAT file in the current directory.
:li.Information typed in Window C will be stored in the
LOSSES.DAT file in the C:\FINANCE directory.
:eul.
```

Formatted Output

o Information typed in Window A will be stored in the STORES.DAT file in whatever directory you designate.

o Information typed in Window B will be stored in the SALES.DAT file in the current directory.

o Information typed in Window C will be stored in the LOSSES.DAT file in the C:\FINANCE directory.

A Compact Unordered List: Use the **compact** attribute to produce a list with no blank lines between the list items.

Input Example

```
:ul compact.
:li.Information typed in Window A will be stored in the
STORES.DAT file in whatever directory you designate.
:li.Information typed in Window B will be stored in the
SALES.DAT file in the current directory.
:li.Information typed in Window C will be stored in the
LOSSES.DAT file in the C:\FINANCE directory.
:eul.
```

Formatted Output

o Information typed in Window A will be stored in the STORES.DAT file in whatever directory you designate.
o Information typed in Window B will be stored in the SALES.DAT file in the current directory.
o Information typed in Window C will be stored in the LOSSES.DAT file in the C:\FINANCE directory.

Nested Unordered Lists: The following example contains two nested, unordered lists. Notice that a bullet (lowercase "o") precedes items in the first-level list and that a dash (—) precedes items in the second-level lists. The bullets and dashes alternate for each level of the list. That is, third-level list items would be preceded by bullets, fourth-level by dashes, and so on.

Input Example

```
:ul compact.
:li.C:REPORTS\SALES.89
:ul compact.
:li.FIRST.QTR
:li.SECOND.QTR
:li.THIRD.QTR
:li.FOURTH.QTR
:eul.
:li.C:REPORTS\SALES.90
:ul compact.
:li.FIRST.QTR
:li.SECOND.QTR
:li.THIRD.QTR
:li.FOURTH.QTR
:eul.
:eul.
```

Formatted Output

```
o C:REPORTS\SALES.89
  - FIRST.QTR
  - SECOND.QTR
  - THIRD.QTR
  - FOURTH.QTR
o C:REPORTS\SALES.90
  - FIRST.QTR
  - SECOND.QTR
  - THIRD.QTR
  - FOURTH.QTR
```

When nesting lists, make sure you end *each* list with an end-list tag.

Ordered List

Ordered lists are vertical arrangements of items, with each item in the list preceded by a number or letter. Use ordered lists when the sequence of the items is important, such as in a procedure.

To create an ordered list, use the ordered-list tag (**:ol.**) to begin the list, and **:eol.** to end it. Identify each item in the list with **:li.**.

Input Example

```
:ol.
:li.Open the diskette-drive door.
:li.Remove the diskette.
:li.Store the diskette in a safe place.
:eol.
```

Formatted Output

1. Open the diskette-drive door.

2. Remove the diskette.

3. Store the diskette in a safe place.

A Compact Ordered List: Use the **compact** attribute to produce a list with no blank lines between the list items.

Input Example

```
:ol compact.
:li.Open the diskette-drive door.
:li.Remove the diskette.
:li.Store the diskette in a safe place.
:eol.
```

Formatted Output

1. Open the diskette-drive door.
2. Remove the diskette.
3. Store the diskette in a safe place.

Nested Ordered Lists: The following example contains two nested, ordered lists. Notice that sequential numbers precede items in the first-level list, and sequential letters precede items in the second-level list. Numbers and letters alternate for each level of the list. That is, third-level list items would be preceded by numbers, fourth-level by letters, and so on.

Input Example

```
:ol.
:li.First item in the first-level list.
:li.Second item in the first-level list.
This item has a nested list within it.
:ol.
:li.First item in the second-level list.
:li.Second Item in the second-level list.
:eol.
:li.Third item in the first-level list.
:eol.
```

Formatted Output

1. First item in the first-level list.
2. Second item in the first-level list. This item has a nested list within it.
   a. First item in the second-level list.
   b. Second Item in the second-level list.
3. Third item in the first-level list.

When nesting lists, make sure you end *each* list with an end-list tag.

Definition List

A definition list is a special list that pairs a term and its description.

To create a definition list, use the definition-list tag (**:dl.**) to begin the list, and **:edl.** to end it. Identify each term in the list with a definition-term tag (**:dt.**) and each description with a definition-description tag (**:dd.**).

Column Width for Definition Terms: **:dl.** has several attributes that let you control the appearance of definition lists. The **tsize=** attribute specifies the width, in character spaces, for the term column. If **tsize=** is not specified, the default width for the term column is 10 character spaces.

Definition-List Headings: If you want headings for the columns of terms and definitions, use the definition-term heading tag (**:dthd.**) to identify the heading for the terms, and the definition-description tag (**:ddhd.**) to identify the heading for the definition descriptions.

Compact Definition List: The **compact** attribute produces a list with no blank lines.

The following example shows the tagging for a compact definition list with headings for the terms and descriptions. It also shows the use of the **tsize=** attribute.

Input Example

```
:dl compact tsize=13.
:dthd.:hp2.Key:ehp2.
:ddhd.:hp2.Purpose:ehp2.
:dt.Insert key
:dd.Switches between insert and replace modes.
:dt.Home key
:dd.Moves the cursor to the beginning of the current line.
:dt.End key
:dd.Moves the cursor to the end of the current line.
:edl.
```

Formatted Output

Key	**Purpose**
Insert key	Switches between insert and replace modes.
Home key	Moves the cursor to the beginning of the current line.
End key	Moves the cursor to the end of the current line.

Specifying where the Definition Descriptions Start: The **break=** attribute defines where the descriptions appear in relation to their terms:

break=none	Places the description on the same line as the term. This is the default. If the term is longer than the specified or default **tsize=** value, the term extends into the description column.
break=all	Places the description on the line below the term.
break=fit	Places the description on the line below the term *only* when the term is longer than the **tsize=** value.

The following example shows the tagging that starts the definition descriptions on the line below the term.

Input Example

```
:dl break=all tsize=3.
:dt.:hp2.Insert key:ehp2.
:dd.Switches between insert and replace modes.
:dt.:hp2.Home key:ehp2.
:dd.Moves the cursor to the beginning of the current line.
:dt.:hp2.End key:ehp2.
:dd.Moves the cursor to the end of the current line.
:edl.
```

Formatted Output

Insert key
  Switches between insert and replace modes.

Home key
  Moves the cursor to the beginning of the current line.

End key
  Moves the cursor to the end of the current line.

A definition description can apply to more than one definition term; that is, you can specify more than one **:dt.** in the sequence before specifying a matching **:dd.**.

The following example shows the tagging for a definition list with descriptions that apply to more than one term.

Input Example

```
:dl compact break=fit tsize=20.
:dthd.:hp2.Grocery Item:ehp2.
:ddhd.:hp2.Type:ehp2.
:dt.:hp2.Orange:ehp2.
:dt.:hp2.Apple:ehp2.
:dd.A fruit.
:dt.:hp2.Carrot:ehp2.
:dt.:hp2.Celery:ehp2.
:dd.A vegetable.
:edl.
```

Formatted Output

Grocery Item	**Type**
Orange	
Apple	A fruit.
Carrot	
Celery	A vegetable.

Parameter List

Parameter lists are similar to definition lists in appearance and the way you use tags to create them. The only difference between the two types of lists is that a parameter list cannot have headings.

The parameter-list tag (**:parml.**) begins the list; its corresponding **:eparml.** ends it. Identify each term in the list with a parameter-term tag (**:pt.**) and each description with a parameter-description tag (**:pd.**).

:parml. has the same attributes as **:dl.**. The **tsize=** attribute specifies the width for the term column. If **tsize=** is not specified, the default width is 10 character spaces.

Compact Definition List: The **compact** attribute produces a list with no blank lines.

Specifying where the Definition Descriptions Start: The **break=** attribute defines where the descriptions appear in relation to their terms:

break=none	Places the description on the same line as the term. This is the default. If the term is longer than the specified or default **tsize=** value, the term extends into the description column.
break=all	Places the description on the line below the term.
break=fit	Places the description on the line below the term *only* when the term is longer than the **tsize=** value.

Nested Parameter Lists: Like simple, unordered, and ordered lists, parameter lists can be nested.

Input Example

```
:parml compact tsize=3.
:pt.:hp2.KEYWORD-1:ehp2.
:pd.Is explained here.
:pt.:hp2.KEYWORD-2:ehp2.
:pd.Is explained here, and its nested subparameters:
:parml compact.
:pt.:hp2.SUBPARM1:ehp2.
:pt.:hp2.SUBPARM2:ehp2.
:pd.Are explained here.
:eparml.
:pt.:hp2.KEYWORD-3:ehp2.
:pd.Is explained here.
:eparml.
```

Formatted Output

KEYWORD-1
  Is explained here.
KEYWORD-2
  Is explained here, and its nested subparameters:
  SUBPARM1
  SUBPARM2
    Are explained here.
KEYWORD-3
  Is explained here.

A parameter description can apply to more than one parameter; that is, you can specify more than one **:pt.** in the sequence before specifying a matching **:pd.**.

The following example shows the tagging for a parameter list with descriptions that apply to more than one term.

Input Example

```
:parml compact tsize=3.
:pt.:hp2.KEYWORD-1:ehp2.
:pt.:hp2.KEYWORD-2:ehp2.
:pd.Is explained here.
:pt.:hp2.KEYWORD-3:ehp2.
:pd.Is not explained here.
:eparml.
```

Formatted Output

KEYWORD-1
KEYWORD-2
    Is explained here.
KEYWORD-3
    Is not explained here.

Tables

Table tags enable you to display text in an arrangement of rows and columns. The system font used to create tables is the monospace font.

The table tag (**:table.**) signals the start of the table. It requires a corresponding **:etable.** at the end of the table.

The row tag (**:row.**) specifies the start of each row in the table. Each row must have at least one column-entry tag (**:c.**). This tag specifies the text for each column in the table.

The **cols=' '** attribute of **:table.** specifies numeric values that represent the column widths, in character spaces, of each column in the table. The combined values cannot exceed 250 characters.

The number of columns in your table is determined by the number of column width values you have specified with the **cols=' '** attribute. For example, if you enter the values shown in the following, your table will have three columns, each of which will be eleven characters spaces wide.

cols='11 11 11'

A Table with Three Columns: The following is a simple example of a table with two rows and three columns:

Input Example

```
:table cols='13 13 13'.
:row.
:c.Column 1
:c.Column 2
:c.Column 3
:row.
:c.Row 1 Col 1
:c.Row 1 Col 2
:c.Row 1 Col 3
:row.
:c.Row 2 Col 1
:c.Row 2 Col 2
:c.Row 2 Col 3
:etable.
```

Formatted Output

Column 1	Column 2	Column 3
Row 1 Col 1	Row 1 Col 2	Row 1 Col 3
Row 2 Col 1	Row 2 Col 2	Row 2 Col 3

If you have more **:c.** tags following a **:row.** tag than you have column-width values, the extra column entries are placed in a new row, and the compiler returns an error message.

If you have fewer **:c.** tags than column-width values, the compiler does not consider this an error. Space is still allocated for the specified columns; however, only the columns for which you have provided entries will be filled.

Table Rules and Frames: The **rules=** attribute of **:table.** specifies whether the table will have vertical rules, horizontal rules, a combination of both, or no rules at all to delineate the items in the table. The values that you can specify for **rules=** are:

rules=both
rules=none
rules=horiz
rules=vert

A Table with Horizontal Rules

Input Example

```
:table rules=horiz cols='10 15 15'.
:row.
:c.SYMBOL
:c.ELEMENT
:c.CHARACTER
:row.
:c.&amp.bxas.
:c.box ascender
:c.&bxas.
:row.
:c.&amp.bxcr.
:c.box cross
:c.&bxcr.
:row.
:c.&amp.bxde.
:c.box descender
:c.&bxde.
:etable.
```

Formatted Output

SYMBOL	ELEMENT	CHARACTER
&bxas.	box ascender	┴
&bxcr.	box cross	┼
&bxde.	box descender	┬

If you do not specify the **rules=** attribute, your table will contain both vertical and horizontal rules (the default).

The **frame=** attribute of **:table.** specifies whether the table will have borders. The values that you can specify are:

frame=none
frame=rules
frame=box

If you specify **frame=none**, there will be no borders.

Specifying **frame=rules** results in a horizontal line at the top and bottom of the table.

If you specify **frame=box**, or do not specify the **frame=** attribute, the table is enclosed in a box.

A Table without a Frame: Here is the same table without a frame.

Example Input

```
:table rules=horiz frame=none cols='10 15 15'.
:row.
:c.SYMBOL
:c.ELEMENT
:c.CHARACTER
:row.
:c.&amp.bxas.
:c.box ascender
:c.&bxas.
:row.
:c.&amp.bxcr.
:c.box cross
:c.&bxcr.
:row.
:c.&amp.bxde.
:c.box descender
:c.&bxde.
:etable.
```

Formatted Output

SYMBOL	ELEMENT	CHARACTER
&bxas.	box ascender	┴
&bxcr.	box cross	┼
&bxde.	box descender	┬

Special Considerations: None of the text-formatting tags (for example, list tags) can be used in a table. You can use character-graphic symbols and highlighted-phrase tags. However, boldface and italic highlighting can cause vertical misalignment of column text and rules. To use boldface highlighting in tables and avoid word alignment problems, place the highlighted-phrase tags (**:hp2.** and **:ehp2.**) as shown in the example. The table rules as well as the text will be displayed in boldface.

Input Example

```
:hp2.
:table cols='11 11 11'.
:row.
:c.Row 1 Col 1
:c.Row 1 Col 2
:c.Row 1 Col 3
:row.
:c.Row 2 Col 1
:c.Row 2 Col 2
:c.Row 2 Col 3
:etable.
:ehp2.
```

Formatted Output

Row 1 Col 1	**Row 1 Col 2**	**Row 1 Col 3**
Row 2 Col 1	**Row 2 Col 2**	**Row 2 Col 3**

The above technique is practical only with **:hp2.**, and does not work for **:hp1.** or for the highlighted-phrase tags that change the color of text.

Unformatted Text in a Table Column: The text in table columns is formatted only once – at compile time. If you do not want the compiler to format the text in a column, enclose it with **:lines.** and **:elines.**, as shown in the following example.

Input Example

```
:table cols='10 10 15 10'.
:row.
:c.Spacecraft
:c.Date
:c.Astronauts
:c.Mission
:row.
:c.Apollo 11
:c.7-16-1969
:c.
:lines.
Neil Armstrong
Edwin Aldrin
Michael Collins
:elines.
:c.
First landing on
the moon.
:etable.
```

Formatted Output

Spacecraft	Date	Astronauts	Mission
Apollo 11	7-16-1969	Neil Armstrong Edwin Aldrin Michael Collins	First landing on the moon.

Plain Lines

Normally, lines of text that have no formatting tags are "wrapped" by IPF; that is, irregular lines in the source file become a continuous string, and one word follows another on a line until the line width of the current window is filled, a formatting tag is encountered, or the end of the window is reached.

The **:lines.** tag, and its corresponding end tag, enable you to control where lines break.

There are many ways to use **:lines.**; here we use it for a quotation.

Input Example

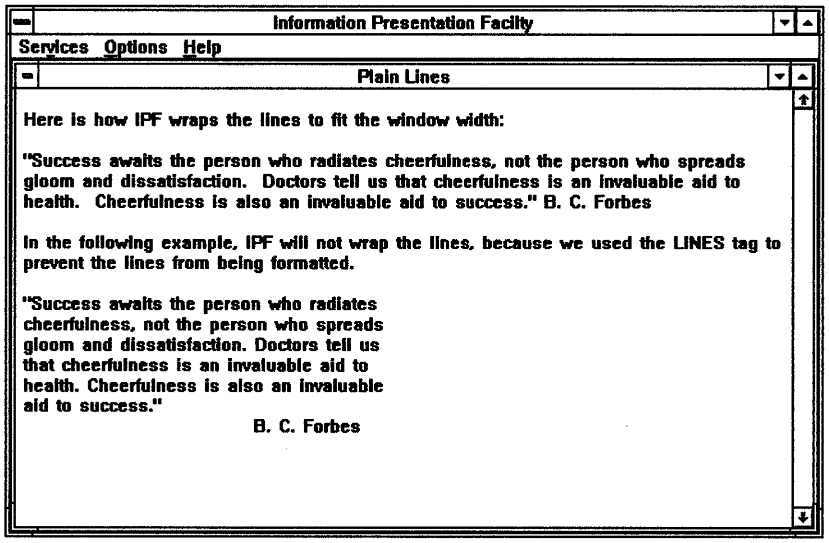
Here is how IPF wraps the lines to fit the window width&colon.

:p.&odq.Success awaits the person who radiates cheerfulness, not the person who spreads gloom and dissatisfaction. Doctors tell us that cheerfulness is an invaluable aid to health. Cheerfulness also is an invaluable aid to success.&cdq. B. C. Forbes

In the following example, IPF will not wrap the lines, because we used the LINES tag to prevent the lines from being formatted.

```
:lines.
&odq.Success awaits the person who radiates
cheerfulness, not the person who spreads
gloom and dissatisfaction. Doctors tell us
that cheerfulness is an invaluable aid to
health.  Cheerfulness also is an invaluable
aid to success.&cdq.
                              B. C. Forbes
:elines.
```

The quotation appears in two forms.



Here is how IPF wraps the lines to fit the window width:

"Success awaits the person who radiates cheerfulness, not the person who spreads gloom and dissatisfaction. Doctors tell us that cheerfulness is an invaluable aid to health. Cheerfulness is also an invaluable aid to success." B. C. Forbes

In the following example, IPF will not wrap the lines, because we used the LINES tag to prevent the lines from being formatted.

```
"Success awaits the person who radiates
cheerfulness, not the person who spreads
gloom and dissatisfaction. Doctors tell us
that cheerfulness is an invaluable aid to
health. Cheerfulness is also an invaluable
aid to success."
                    B. C. Forbes
```

Figure 4-2. Plain Lines Example

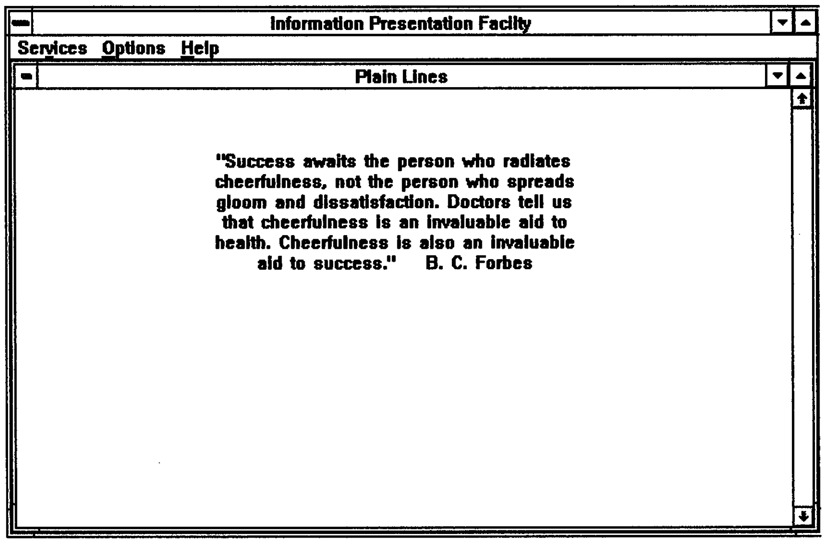
In the first case, IPF wraps the lines to fit the window width. In the second, IPF does not wrap the lines, because **:lines.** prevent them from being formatted. If a line of text were to exceed the width of the current window, it would be clipped. Also, when IPF encounters other tags between **:lines.** and **:elines.**, such as quotation tags, the tags are processed.

Aligned Lines: **:lines.** has an **align=** attribute, which you use to align text to the left, right, or center of the window.

Assume that in the previous example, the tag was:

```
:lines=center.
```

The output would be as shown here.



"Success awaits the person who radiates
cheerfulness, not the person who spreads
gloom and dissatisfaction. Doctors tell us
that cheerfulness is an invaluable aid to
health. Cheerfulness is also an invaluable
aid to success." B. C. Forbes

Figure 4-3. Text aligned in the center of the window.

Figures and Captions

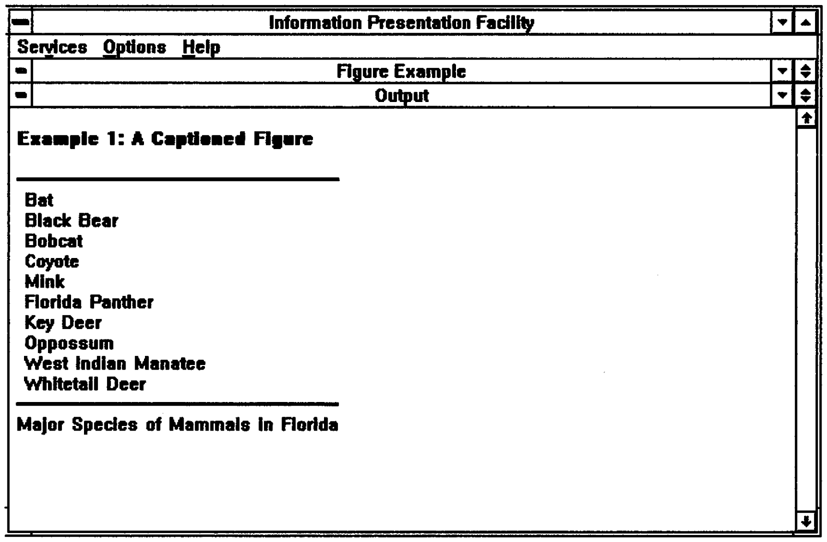
The figure tag (**:fig.**) is similar to **:lines.**. Both convey the same message: "Do not format the text that follows." Also, both tags have an end tag.

A Captioned Figure: Associated with **:fig.** is **:figcap.**, which enables you to place a descriptive sentence or caption above or below the text.

Input Example

```
:h4.Example 1&colon.  A Captioned Figure
:fig.
────────────────────────────
 Bat
 Black Bear
 Bobcat
 Coyote
 Mink
 Florida Panther
 Key Deer
 Oppossum
 West Indian Manatee
 Whitetail Deer
────────────────────────────
:figcap.Major Species of Mammals in Florida
:efig.
```

The formatted output looks like this:



Example 1: A Captioned Figure

Bat
Black Bear
Bobcat
Coyote
Mink
Florida Panther
Key Deer
Oppossum
West Indian Manatee
Whitetail Deer

Major Species of Mammals in Florida

Figure 4-4. Figure and Figure Caption

Textual Examples

One way of helping readers understand information is to use examples. The example tag (**:xmp.**) and its corresponding end tag (**:exmp.**) enable you to illustrate your information with textual examples by turning formatting off so that you can arrange text any way you want it. The text will be displayed in a monospace font. To change the monospace font, use **:font.** within **:xmp.**. For more information about **:font.**, see "Changing Fonts" on page 4-24.

Input Example

```
:xmp.
File   Edit   View   Options   Help

                 All
                 Some . . .
                 By . . .
:exmp.
```

Formatted Output

```
File   Edit   View   Options   Help

                 All
                 Some . . .
                 By . . .
```

Restriction: You cannot nest **:xmp.** within another **:xmp.**.

Character Graphics

If you want to include simple line drawings, use the character graphics tag (**:cgraphic.**) and its corresponding end tag (**:ecgraphic.**). Text within this tag is displayed in a monospace font. To change the monospace font, use **:font.** within **:cgraphic.**. For more information about **:font.**, see "Changing Fonts" on page 4-24. If text does not fit within the boundaries of a window, it is clipped, not wrapped.

Place the tags before and after the character graphic, as shown in the following example.

Input Example

```
:cgraphic.
┌──────────┬────┬─────────────┐
│File  Edit│View│Options  Help│
└──────────┼────┴────────┬────┘
           │All          │
           │Some . . .   │
           ├─────────────┤
           │By . . .     │
           └─────────────┘
:ecgraphic.
```

Formatted Output

```
┌──────────┬────┬─────────────┐
│File  Edit│View│Options  Help│
└──────────┼────┴────────┬────┘
           │All          │
           │Some . . .   │
           ├─────────────┤
           │By . . .     │
           └─────────────┘
```

Restriction: You cannot nest **:cgraphic.** within another **:cgraphic.**.

Changing Fonts

The **:font.** tag is used to change the current font within the text of the current window. When a heading tag that defines a new window is encountered, the font is reset to the system default font.

The font tag has three attributes: **facename=** and **size=** are required; **codepage=** is optional. If a code page value is not specified, the code page of the active system is used.

facename= specifies the name of the font you want to change to. Some of the common values for this attribute are:

Helv
Courier
default

size= specifies the height and width, in *points*, of the font you have selected. (A *point* is a typesetting measure equal to approximately 1/72 of an inch.) The value is expressed in the form, *HxW*. For example, suppose you want to change the current font to an 18-point-high by 10-point-wide Helvetica font. You would specify:

```
:font facename=Helv size=18x10.
```

You do not have to know exact point values. IPF uses a "best fit" method to select the font. If, in the example above, you had specified *20x12* as the size value, IPF would have selected *Helv 18x10* because it is the closest size to the one you specified.

Using **:font.**, you can make as many font changes within a window as you want. You can define highlighted phrases while a font tag is in effect, and the tagged text will be displayed in the font style corresponding to that typeface.

You can use **:font.** within the **:xmp.** and **:cgraphic.** tags to change the default system monospace font. To change the default system monospace font, specify the desired **facename=** and **size=** attribute.

The following resets the font to the default system proportional font.

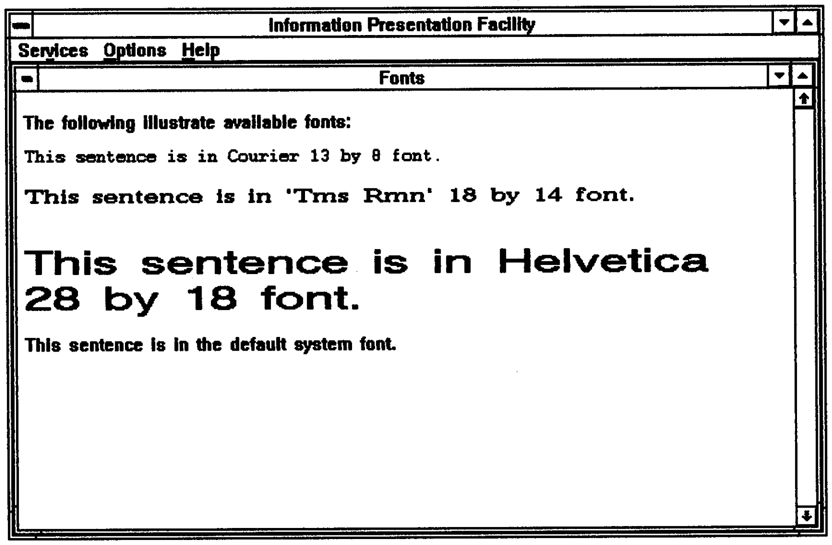
```
:font facename=default size=0x0.
```

In the following example, the font style is reset for each list item in the simple list.

Input Example

```
:p.The following illustrate available fonts&colon.
:sl.
:font facename=Courier size=13x8.
:li.This sentence is in Courier 13 by 8 font.
.*
:font facename='Tms Rmn' size=18x14.
:li.This sentence is in 'Tms Rmn' 18 by 14 font.
.*
:font facename=Helv size=28x18.
:li.This sentence is in Helvetica 28 by 18 font.
.*
:font facename=default size=0x0.
:li.This sentence is in the default system font.
:esl.
```

Here is the formatted output.



The following illustrate available fonts:

This sentence is in Courier 13 by 8 font.

This sentence is in 'Tms Rmn' 18 by 14 font.

This sentence is in Helvetica 28 by 18 font.

This sentence is in the default system font.

Figure 4-5. Example of the Font Tag

Changing Color

The color tag (**:color.**) with its attributes **fc=** and **bc=**, enables you to change the color of the text (foreground color) and the color of the area behind the text characters (background color).

Colors set with this tag remain in effect until others are specified, or until a heading definition is encountered.

To return to the system colors, specify:

```
:color fc=default bc=default.
```

In the following example, each of the first three color tags specifies different foreground and background colors. The last color tag returns the colors to the system colors.

Input Example

```
:ol.
:color fc=green bc=blue.
:li.Color the foreground green; color the background blue.
.*
:color fc=blue bc=red.
:li.Color the foreground blue; color the background red.
.*
:color fc=cyan bc=yellow.
:li.Color the foreground cyan; color the background yellow.
.*
:color fc=default bc=default.
:li.Return to the system colors.
:eol.
```

The formatted output is quite colorful (see the the online *IPF Reference*).

Margins

You can specify the boundaries of text in a window by using the margin tags. The left-margin tag (**:lm.**) specifies how many character spaces from the left border of the window the text is to start. The right-margin tag (**:rm.**) specifies how many character spaces from the right border the text is to end.

The **margin=** attribute sets the margin for the text. If none is specified on the **:lm.** or **:rm.** tag, the default is 1.

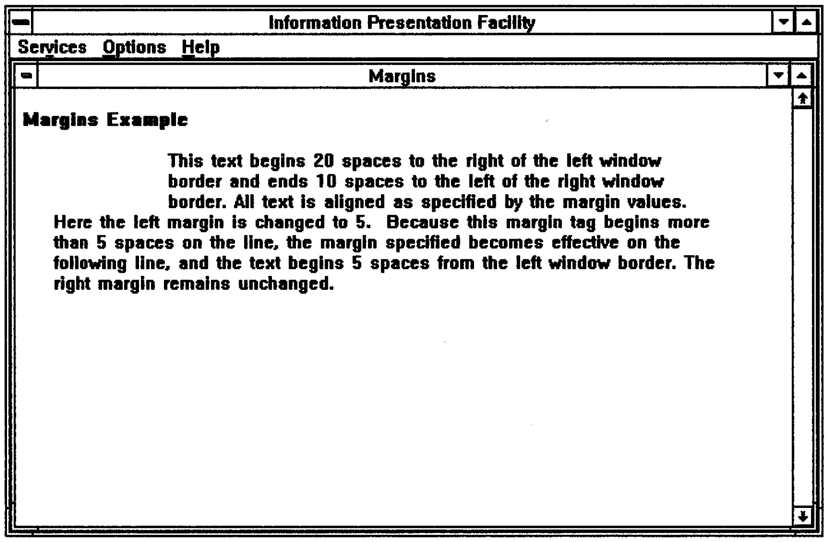
If the margin tag in a line begins beyond the specified boundary, the new margin becomes effective on the next line.

You can have multiple margin tags in your file. The specified margins remain in effect until they are reset.

Input Example

```
:p
:rm margin=10.
:lm margin=20.This text begins 20 spaces to the right
of the left window border and ends 10 spaces to the
left of the right window border.
All text is aligned as specified
by the margin values. :lm margin=5.Here the left margin
is changed to 5.  Because this margin tag begins
more than 5 spaces on the line, the margin specified
becomes effective on the following line, and the text
begins 5 spaces from the left window border.
The right margin remains unchanged.
```

Here is how the window looks:



Margins Example

This text begins 20 spaces to the right of the left window border and ends 10 spaces to the left of the right window border. All text is aligned as specified by the margin values.
Here the left margin is changed to 5. Because this margin tag begins more than 5 spaces on the line, the margin specified becomes effective on the following line, and the text begins 5 spaces from the left window border. The right margin remains unchanged.

Figure 4-6. Example of the Margin Tag

Bit Map and Metafile Graphics

In a previous topic, we discussed how you can use **:cgraphic.** to illustrate your text with character graphics. With **:artwork.**, you can illustrate your text with *bit-map* or *metafile* graphics. A bit map is a representation of an image, and can be created with such tools as the Icon Editor, which is available with the OS/2 2.0 Developer's Toolkit. Metafiles provide device independence — bit-maps do not. The bit map or metafile graphics reside in a file that must be specified with the **name=' '** attribute of **:artwork.**. This file is then loaded when you compile your source file with the IPF compiler.

The artwork tag has other attributes as well:

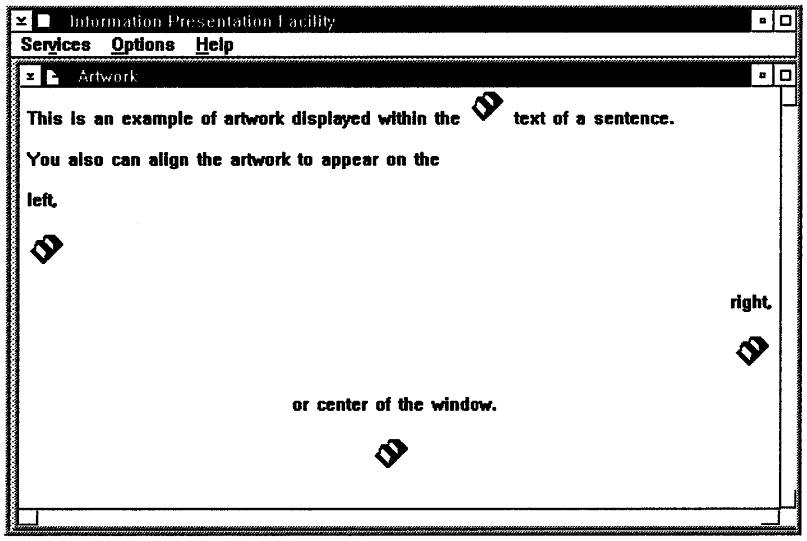
- The **align=** attribute enables you to position the graphic. The values are **left**, **right**, and **center**, and are with respect to the current margins.
- The **fit** attribute causes a bit map to be redrawn and scaled to fit the window.

  The ratio between the width and height of the window should be the ratio of the original width and height of the bit map or metafile; otherwise, the graphic might appear distorted.
- The **runin** attribute enables you to place a graphic within a line of text. For example, to include an icon within a line of text, the text and tag would be as follows:

Input Example

```
:p.This is an example of artwork displayed within the
:artwork runin name='BOOK.BMP'.
text of a sentence.
.**
:p.You can also align the artwork to appear on the
:lines align=left.
left,
:elines.
:artwork align=left name='BOOK.BMP'.
:lines align=right.
right,
:elines.
:artwork align=right name="BOOK.BMP'.
:lines align=center.
or center of the window.
:elines.
:artwork align=center name='BOOK.BMP'.
```

It would bring the artwork into the screen like this.



This is an example of artwork displayed within the text of a sentence.

You also can align the artwork to appear on the

left,

right,

or center of the window.

Figure 4-7. Example of the Artwork Tag

Chapter 5. Linking

Today, the computer's ability to link pieces of information gives the author flexibility in layering and structuring documents, and at the same time, provides cohesive information.

This chapter describes the tags that identify, associate, and link one window to another window. This chapter also describes the different types of linking available with IPF, and what to expect when using them.

Window Identifiers

The link tag (**:link.**) allows you to link to a heading, a footnote, an external database, or another application. The **reftype=** attribute is required with each link tag description. This attribute identifies the type of link you are defining.

The **res=** attribute and the value specified, identifies the window you are linking to. This attribute is the window identifier. A **res=** number must be in the range 1 through 64000. The same window identifier must be specified in the tagging of the window you are linking to in order for a hypertext link to exist (see "Hypertext Links" on page 5-2).

The IPF compiler recognizes links to headings (including hidden headings) only when the heading level is within the default range (**toc**=123) or specified range of heading levels. If you specify a window identifier for a level that is lower in the hierarchy than that recognized for contents entries, and then attempt to link to it, the compiler returns an error message. For example, suppose the default is in effect for contents entries; that is, only heading levels 1 through 3 cause entries in the Contents window. Also suppose your file contains the following heading definition:

```
:h4 res=050.Copy File
```

The heading "Copy File" appears in the same window as the preceding heading level 3. If you use this window identifier in a link definition to link to the heading from another window, the IPF compiler returns the error message, `No res for this reference`.

If you are creating windows for an online document (a .INF file), you can use the **res=**, **id=**, or **name=** attribute of the heading tag to specify window identifiers. An advantage of using either **id=** or **name=** is that you can specify both alphabetic and numeric characters, which can make the job of assigning and remembering window IDs easier. If you use one of these attributes, you must use the **refid=** attribute of **:link.** when defining a hypertext cross-reference to the window.

If you need to use both **res=** numbers and **id=** values, you can specify both in a window heading. For simplicity, you can assign the same number to both identifiers.

Note: If an OS/2 application needs to communicate with an IPF window, you must use the **res=** attribute as a window identifier.

Types of Links

Links are electronic pathways that connect one online element to another. With IPF, the user can be linked from one window to another by means of selectable text and graphic areas that the author defines. The user also can be linked to information in another IPF database.

Different types of links support document designs and information retrievability in various ways:

Hypertext Links
Selectable words or phrases that connect related information.

Hypergraphic Links
Selectable graphics that connect related information.

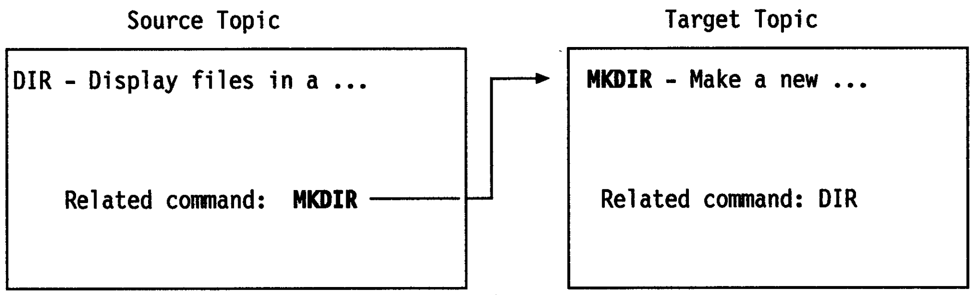
Automatic Links
Links that begin a chain reaction at the primary window. When the user selects the primary window, an automatic link is activated to display secondary windows.

External Links
Links that connect external online document files.

Hypertext Links

Hypertext is the linking of online information so the user can navigate from selectable text to related information. A hypertext link is the association between two topics. The *origin* of the link is the source topic; the *destination* is the target topic.

In the following example, the DIR command is the source topic; it describes the directory command. Within the DIR topic is a reference to the MKDIR command — the target topic.



You use **:link.** to establish a hypertext link between a topic in the source-topic window and a topic in the target-topic window. **:link.** enables you to create selectable, highlighted text in the source-topic window. When users select this text, they are linked to the window containing the target topic, and the linked window appears.

Consider the following example:

```
:link reftype=hd res=123.MKDIR:elink.
```

- **reftype=hd** indicates the hypertext phrase *MKDIR* is being linked to a heading in the target-topic window.

  Notice MKDIR is delimited by the period of the **:link.** tag and the colon of the **:elink.** tag.

- **res**=123 is the identifier of the target-topic window.

  The heading tag of the target-topic window must contain this identifier. The following is an example:

  ```
  :h2 res=123.MKDIR
  ```

For more information about hypertext links, see "Display Another Window of the Same Library" on page 5-6.

Hypergraphic Links

A hypergraphic link is similar to a hypertext link except that the user navigates from a selectable graphic instead of selectable text.

Bit Maps

Graphic illustrations are usually bit maps. Bit maps can be monochrome or color and can be created with the Presentation Manager Icon Editor, which is available in the Toolkit. The bit map resides in a separate file called by IPF at compile time.

The artwork tag (**:artwork.**) identifies the name of the bit-map; for example:

```
:artwork name='mybitmap.bmp'.
```

The **:artlink.** and **:eartlink.** tags define areas of the bit map that are selectable *hypergraphic*. This means the user can link from the artwork to additional information. If no **:artlink.** tag is used, no hypergraphic areas are defined.

If you want the entire bit map to be hypergraphic, the tagging is simple. You have only one art link, and you do not have to define the area. The following shows the tagging required to establish a link:

Input Example

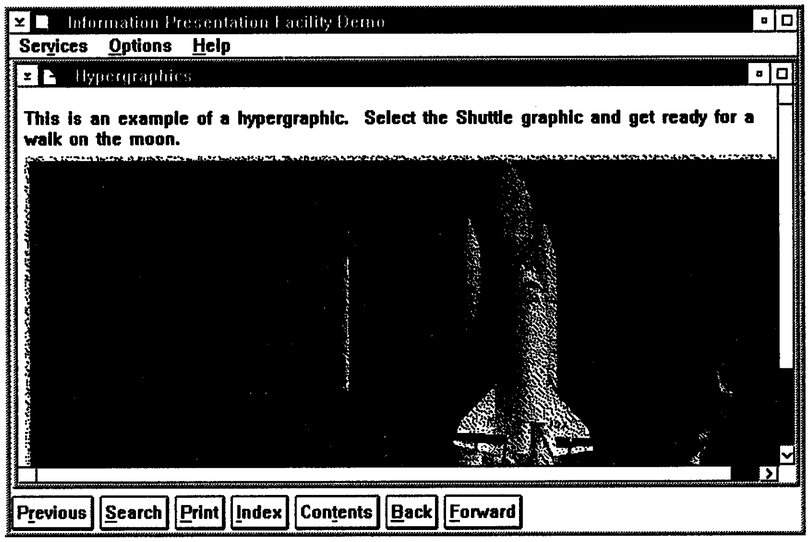
```
:p.This is an example of a hypergraphic.
Select the Shuttle graphic and get ready for a walk on
the moon.
:artwork name='shuttle.bmp'.
:artlink.
:link reftype=hd res=001.
:eartlink.
```

Notice there is no **:elink.** tag. Instead, there is an **:eartlink.** tag. An **:elink.** tag is required only to denote the end of a hypertext link.

You also need to specify the identifier in the tagging for the window you are linking to. For example:

```
:h1 res=001.Apollo 11
```

Here is the formatted result:



This is an example of a hypergraphic. Select the Shuttle graphic and get ready for a walk on the moon.

Figure 5-1. The entire bit map as a selectable hypergraphic area.

When the user double-clicks on the hypergraphic area, the window whose identifier is 001 ("Apollo 11") appears.

Metafiles

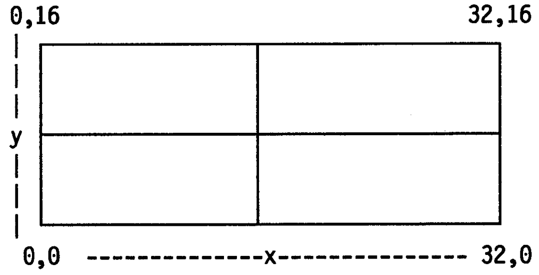
A *metafile* is another type of file in which graphics are stored. However, a metafile contains data generated from the Presentation Manager graphics (GPI) functions only. (For information about graphics functions, see the OS/2 2.0 *Programming Guide*, Volume 3.) IPF supports a metafile as a hypergraphic link only when the entire metafile is defined as a hypergraphic area.

The artwork tag identifies the file name of a metafile as follows:

```
:artwork name='myfile.met'.
```

Segmented Hypergraphics

You can divide your bit map into rectangular segments, make each segment selectable, and have each segment link to different information. You must define each segment in terms of values along the x and y axes. Values for x and y define the origin of the segment. The changes in x and y are given as values for cx and cy. The following is an example of a segmented bit map:



The following shows the tagging to establish a bit-map segment as a hypergraphic area:

```
:artwork name='show2.bmp'.
:artlink.
:link reftype=hd res=001 x=0 y=0 cx=16 cy=8.
:eartlink.
```

Automatic Links

Links also can be made automatically. An automatic link occurs when the user performs an action that selects a window in which a link is defined. For automatic links to occur, the **reftype=** attribute of the **:link.** tag must have a value of **hd**, **inform**, or **launch**. Automatic links allow you to:

- Display multiple windows when a heading or link definition is selected (**hd** attribute).
- Display multiple secondary windows within the coverpage of a primary window (**hd** attribute). For a tagging example, see Figure 6-12 on page 6-13.
- Send a message to the application when a window is displayed (**inform** attribute). For a tagging example, see Figure 9-7 on page 9-18.
- Start a Presentation Manager program when a window is displayed (**launch** attribute).

Automatic links can be associated with selectable links so that another action occurs in addition to the display of a linked window. For example, a Presentation Manager program can be started, or a message can be sent to the application program.

External Links

An external link is a link from a .HLP file to another .HLP file or from a .INF file to another .INF file.

If you are linking from one internal database to another, use the **res=** attribute. If you want to allow external databases to link to a window in your file, the window heading must contain the **global** attribute, and you must use the **id=** attribute as a window identifier.

For more information about external links, see "Display a Help Window from Another Help Library" on page 5-6.

What Linking Can Do

You now know that **:link.** makes text phrases and hypergraphic areas within a window selectable. When the user selects a hypertext or hypergraphic area, the following occurs, depending on the content of the **:link.** tag:

- Another window of the same library is displayed.
- Another window of a different library is displayed.
- A footnote window is displayed.
- A message is sent to the application program.
- Another application is started.

Display Another Window of the Same Library

When you want the user to link to another window in the current library, use the **reftype=hd** attribute with **:link.**. For example:

```
:link
reftype=hd res=21084.What Are Libraries For?
:elink.
```

The **hd** attribute tells the compiler to link to a heading in another window. The **res=** attribute value specifies the identification of the window being linked to.

The text "What Are Libraries For?" is uniquely highlighted in the window so that the user knows it is selectable. If the user selects it, the window containing the heading defined by **res=** 21084 appears.

Note: The highlighting of a hypertext phrase is done with a color selected by IPF and should not be confused with highlighted-phrase tags, which are used to change the type font. (See "Highlighted Phrases" on page 4-1 for an explanation of these tags.)

The tagging shown in Figure 5-2 on page 5-7 contains an example of the link tag. Also included is the tagging for the window being linked to.

Display a Window Linked to Another Database

You also can link a user to a window in another IPF .HLP or .INF file. You must specify the file name with the **database=** attribute. If the following were in the source file, selection of the hypertext link would cause the file, EDITOR.HLP to be loaded, and the window whose ID is 001 to be displayed.

```
:link reftype=hd database='editor.hlp' refid=001.
Editing Functions
:elink.
```

The heading definition in the other file must contain the **global** attribute. If the link to the file cannot be resolved, the hypertext phrase in the link is not highlighted. For example, if the .INF or .HLP file is not available, IPF will not highlight the linked phrase. If the .INF or .HLP file becomes available, IPF will dynamically highlight the phrase.

Display a Help Window from Another Help Library

If you are creating a window for a help library (a .HLP file), you must use the **res=** attribute to assign an identifier to each window. For example:

```
:h1 res=2001 id=2001 global.
Help for Copy
```

IPF uses the value specified for **res=** (any integer from 1 through 64,000) to associate a window with a user's request for help on a field or window of the application. If you use the **res=** attribute in a heading tag, you must also use it in a link tag when defining a hypertext cross-reference to the window. For example:

```
:link reftype=hd res=2001.
Help for Copy
:elink.
```

```
:**************************************************
:* In the following source, the text of the window
:* contains a heading tag with a window
:* identifier, a paragraph tag, and a hypertext
:* link to another window.
:**************************************************
:h1 res=21083.The Library Manager
:i1.object code libraries
:p.
The Library Manager (LIB) lets you create and maintain
libraries of object code.  A library is an organized
collection
of object code; that is, a library contains functions and data
that are already assembled or compiled and ready for linking
with your programs. See:
:link
reftype=hd res=21084.What Are Libraries For?
:elink.
:p.
LIB works with both DOS and OS/2 files.
:**************************************************
:* The following contains a heading tag with a
:* window identifier that matches the link-tag
:* res= attribute above.
:* This file also contains an unordered list.
:**************************************************
:h2 res=21084.What Are Libraries For?
:p.Programming libraries of object code are used:
:ul.
:li.To support high-level languages.
:p.Most compilers include libraries to perform standard
operations, such as input/output and floating-point mathematics.
:p.
When your program refers to a library routine, the
compiler and linker combine the library routine with your
program.
:li.To perform complex and specialized activities, such
as database management or advanced graphics.
:p.Compilers include libraries for specialized tasks. You
also can use a library from a third party software vendor.
:li.To support your own work.
:p.If you have created routines that you use with a
variety of programs, you might want to consolidate these routines
into a library.  You then can link to one library object module
rather than to a large group of object files.
:eul.
```

Figure 5-2. Example source for linking to another window.

Display a Footnote Window

A footnote window results when the user selects a hypertext phrase that is linked to a footnote tag (**:fn.**). The text between **:fn.** and **:efn.** is what appears in the footnote window. The following is an example of the tagging for the footnote text:

Input Example

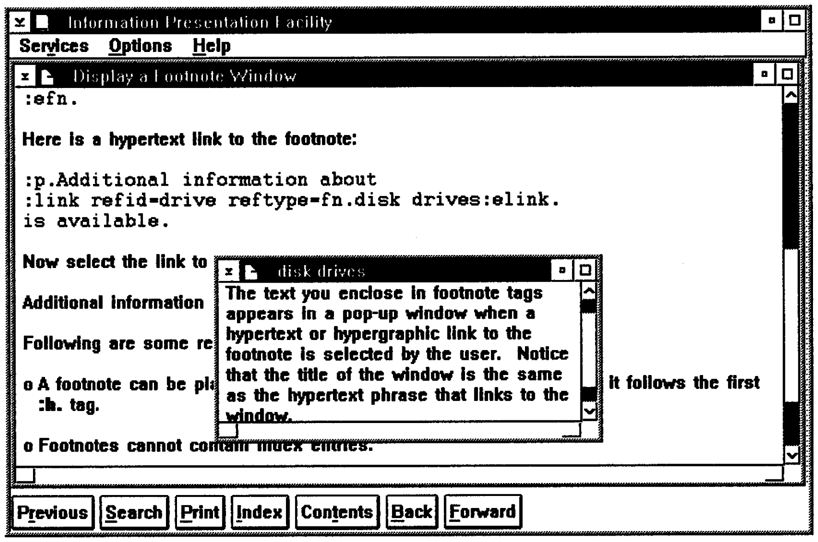
```
:fn id=drive.
:p.The text you enclose in footnote tags appears in a
small window when the user selects a hypertext or hypergraphic
link to the footnote.  Notice that
the title of the window is the same as the hypertext
phrase "disk drives" that links to the window.
:efn.
```

The **id=** attribute identifies the footnote for linking purposes.

In the **:link** tag, use **refid=** to refer to the footnote identifier (in this case, "drive"), and **reftype=fn** to indicate that the link is to a footnote, and to specify the title of the footnote window. The following is an example of the tagging for a link to the footnote:

```
:p.Additional information about
:link refid=drive reftype=fn.disk drives:elink.
is available.
```

Figure 5-3, shows the resulting footnote window.



```
:efn.
```

Here is a hypertext link to the footnote:

```
:p.Additional information about
:link refid=drive reftype=fn.disk drives:elink.
is available.
```

Now select the link to

Additional information

Following are some re

o A footnote can be pl **it follows the first**
:h. tag.

o Footnotes cannot co

The text you enclose in footnote tags appears in a pop-up window when a hypertext or hypergraphic link to the footnote is selected by the user. Notice that the title of the window is the same as the hypertext phrase that links to the window.

Figure 5-3. Footnote window.

The following are some important points to remember about footnotes:

- A footnote can be placed anywhere in your source file, as long as it follows the first heading tag.
- Footnotes cannot contain index entries.
- Information in a footnote cannot be detected by a search.
- A footnote CANNOT be in a window that has a **split** attribute in its heading or link definition.

Send a Message to the Application

When the **reftype=inform** attribute is specified with **:link.**, a message is sent to the application. The **res=** attribute, instead of being a resource identifier for IPF (a window ID), is a resource identifier for the application. The value specified must be an integer. When the application receives the message, it can then perform an application-specific function.

For more information about how messages are sent to application windows using the **reftype=inform** attribute see "Using Communication Windows" on page 9-13.

Start an Application

The **reftype=launch** attribute of **:link.** causes IPF to start another Presentation Manager application. The **object=** attribute indicates the file specification of the application. The **data=** attribute specifies parameters associated with the application to be started.

You can use the **reftype=launch** attribute with **:link.** to start a tutorial.

Chapter 6. Customizing Windows

A window is an area of the screen with visible boundaries within which information is displayed. Often a single window uses the entire screen for its information. Because online information is best presented in small pieces, or units, most designs call for a multiple window format. This chapter explains how to size and position more than one window on a screen, and how to use attributes that enable IPF to open and close those windows. Before you begin this chapter, make sure you read about the OS/2 standard windows described in Chapter 2, "Viewing the User Interface for Online Information."

For a summary of attributes described in this chapter, see "Summary Tables of Attribute Values for Origin and Size" on page 6-22, "Summary Table for Heading Attributes" on page 6-23, and "Summary Table for Link Attributes" on page 6-24.

The Default Window

Both the heading tags (**:h*n*.**) and the link tag (**:link.**) have attributes that affect how windows look on a screen. For example, the attributes define:

- Window size and position
- Which window controls are provided to the user
- What windows are displayed.

You do not have to use all the attributes provided by a heading tag to define a window. The following is an example of the minimum tagging required for a window:

```
:h1 res=001.My First Window
:p.
Here is the text for the first window.
```

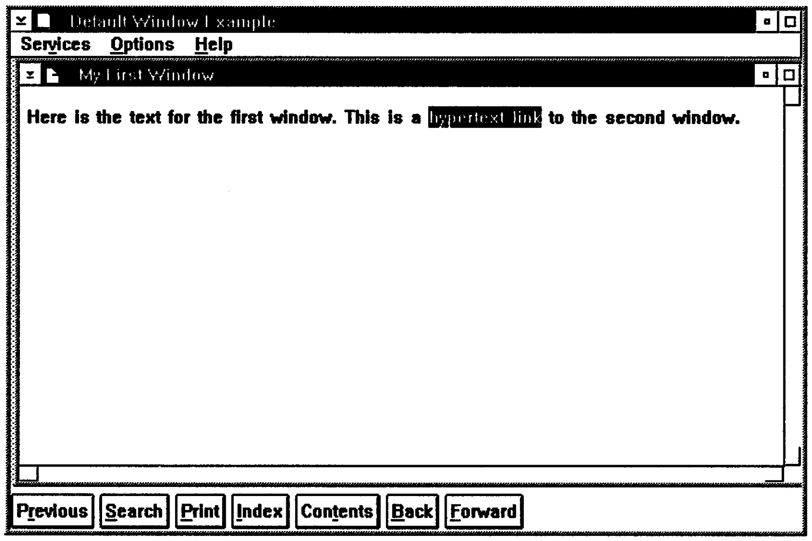
In this example, **:h1.** creates a level-1 entry in the Contents window and the title, "My First Window," in the title bar of the default window.

Figure 6-1 shows the tagging to produce the two default windows shown in Figure 6-2.

```
:h1 res=001.My First Window
:p.
Here is the text for the first window.
This is a
:link reftype=hd res=002.
hypertext link
:elink.
to the second window.
:h1 res=002.My Second Window
:p.
Here is the text for the second window.
This is a
:link reftype=hd res=001.
hypertext link
:elink. to the first window.
```

Figure 6-1. Tagging for two IPF default windows.

Figure 6-2 shows the compiled version of the tagging shown in Figure 6-1. "My First Window" is one of the default windows and is bounded by the window "Default Window Example." This window is called a *coverpage* and provides window controls for the user.



Here is the text for the first window. This is a hypertext link to the second window.

Figure 6-2. Example of an IPF default window.

The two windows each have a hypertext link. Selection of the hypertext link in "My First Window" causes the other default window "My Second Window" to display. Each default window has the same characteristics:

- Its size is 100% of the coverpage window.
- It provides window controls for the user:
  - Title bar with a title bar icon
  - Maximize and hide buttons
  - Vertical and horizontal scroll bars
  - Sizing borders
  - Push buttons.

Attribute Values for Window Controls

Both the heading tag and **:link.** have attributes that define window controls. Following are the names of the window-control attributes, and values you can specify (defaults are underscored):

titlebar=yes|sysmenu|minmax|<u>both</u>**|none**

scroll=horizontal|vertical|<u>both</u>**|none**

rules=border|<u>sizeborder</u>**|none**

You can eliminate window controls altogether by specifying:

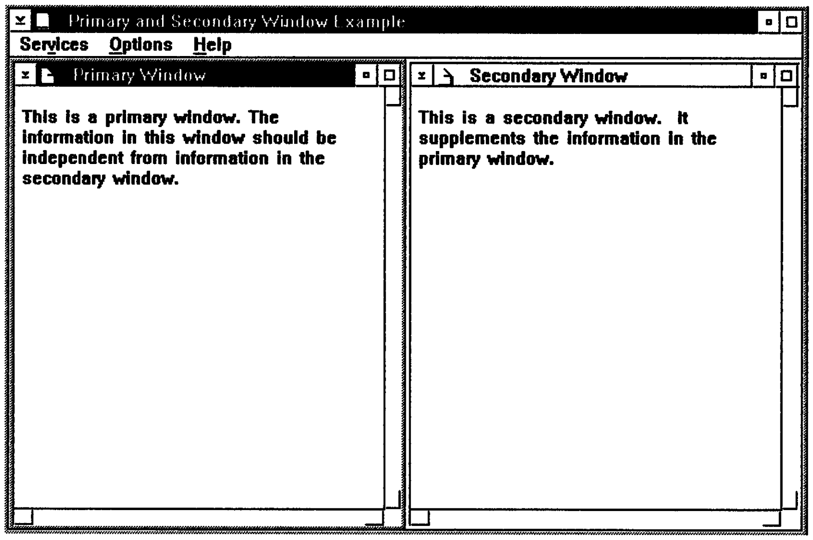
titlebar=none scroll=none rules=none

You then can substitute controls of your own. By eliminating borders around windows and using **:font.** to specify fonts, you can design a more sophisticated layout of text and graphics. The OS/2 system tutorial is an example of this.

For informaton about the tags that control the display of push buttons, see "Attribute Values for the Control Area of a Window" on page 3-10.

Multiple Windows

Windows can be considered to be subdivisions of the screen. They can be either primary or secondary windows. A primary window is where the main topic appears, or where the interaction between a user and an object or application takes place. A secondary window usually supplements the information in the primary window. It is closed when its primary window is closed. Figure 6-3 on page 6-4, shows a simple multiple-window design with a primary and secondary window.



This is a primary window. The information in this window should be independent from information in the secondary window.

This is a secondary window. It supplements the information in the primary window.

Figure 6-3. A primary and secondary window arrangement.

To create the two-window format shown in the figure, you must define the size of each window, then position them within the boundaries of the coverpage window. When defining window size, you specify horizontal and vertical areas of the window, using window coordinates.

Defining Window Origin and Size

Each window represents a rectangle with *x* and *y* coordinates. The x-axis is always horizontal; the y-axis is always vertical. The position where the values specified for x and y intersect is the window's origin. From this position, width and height are measured. Figure 6-4 shows the window coordinates of a primary and secondary window.

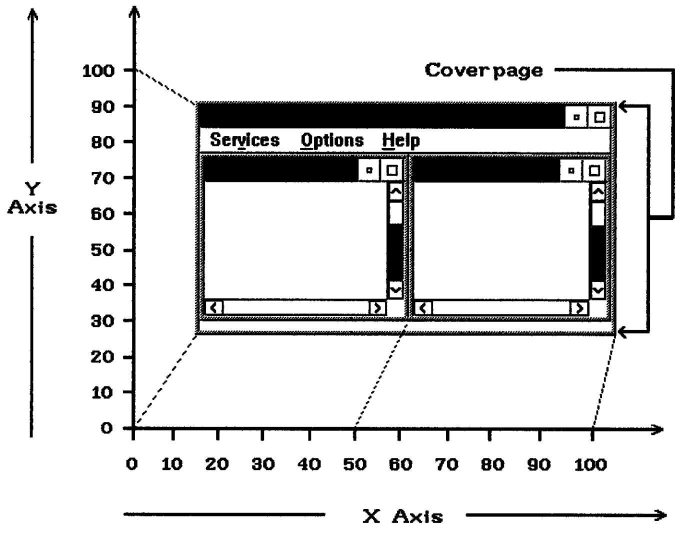


Figure 6-4. A window in relationship to its coordinates.

Attribute Values for Window Origin and Size

Both the heading tags and **:link.** have attributes that define window origin and size. The heading tag has four attributes:

x =	Specifies a point on the *x* axis. The x-axis runs horizontally from left to right.
y =	Specifies a point on the *y* axis. The y-axis runs vertically from bottom to top.
width =	Specifies the width (horizontal space) of the window.
height =	Specifies the height (vertical space) of the window.

The **:link.** tag also has four attributes:

vpx =	Specifies a point on the *x* axis. The x-axis runs horizontally from left to right.
vpy =	Specifies a point on the *y* axis. The y-axis runs vertically from bottom to top.
vpcx =	Specifies the width (horizontal space) of the window.
vpcy =	Specifies the height (vertical space) of the window.

Origin and size attributes also can be assigned values of the following types:

- Absolute
- Relative
- Dynamic.

Absolute Values

Absolute values are specified in characters, pixels, or points. The format for an absolute value is an integer followed by one of these letters:

c (characters)
Average character width of the default system font.

x (pixels)
Pixel size, dependent on the display adapter in use.

p (points)
Typesetting measure, equal to approximately 1/72 inch.

Relative Values

Relative values are specified as percentages of the display area of the coverpage window. The format for a relative value is an integer followed by the percent sign (%).

Dynamic Values

Dynamic values for x- and y-coordinates identify locations on the coverpage-window perimeter or its center. Values are **left** and **right** for x, **top** and **bottom** for y, and **center** for both.

Heading Definition Example

The window defined in the following example is a primary window; its origin is specified using dynamic values, and its width and height are specified as percentages of its coverpage window.

```
:h1 res=001
    x=left y=bottom width=50% height=100%
    group=1.Primary Window
```

For now, ignore "group = 1." We will explain it later.

The most practical values to use for window size and position are a combination of relative and dynamic values. Then, if the user resizes the coverpage window, IPF automatically resizes and repositions the windows relative to the new size and position of the coverpage window. If you use absolute values, the window might be clipped when the user resizes the coverpage window.

When defining window position and size, you cannot mix absolute values with dynamic or relative values for either of the following combinations of attributes:

x= and **width=**
y= and **height=**

If no values for x and y are specified, the origin of the window is 0,0. If you specify an origin other than 0,0, you also must specify width and height values. Negative values for these attributes are not allowed.

Origin and Size Example

The example of a source file shown in Figure 6-5, defines two windows. The origin and size attributes specified with the heading definitions place the windows adjacent to one another on the screen.

```
:h1.Origin and Size Window Example
:h2 res=003
    x=left y=bottom
    width=50% height=100%.
Primary Window
:p.
Here is the text for the primary window.  This is a
:link reftype=hd res=004.
hypertext link
:elink.
to the secondary window.
:h2 res=004
    x=right y=bottom
    width=50% height=100%.
Secondary Window
:p.
Here is the text for the secondary window.  This is a
:link reftype=hd res=003.
hypertext link
:elink.
to the primary window.
```

Figure 6-5. Sample source file for defining window origin and size.

The origin of the first window is the lower left-hand corner of the coverpage window. It occupies 50% of the width, but 100% of the height of the coverpage window on the left-hand side.

The origin of the second window is the lower right-hand corner of the coverpage window. It occupies 50% of the width, but 100% of the height of the coverpage window on the right-hand side.

Although these two windows occupy adjacent positions on the screen, you cannot display them both at the same time. To define separate windows, you must specify a *group* number in the heading definition.

Displaying Multiple Windows

To display more than one window on the screen, you must assign a unique group number to each window with the **group=** attribute. This attribute can be specified with **:link.** or the heading tag.

If you do not specify a group number, a value of 0 is assigned. (This is the default value and is reserved for use by IPF.) If another window is already opened with the number specified for **group=**, IPF swaps its image (places the image in the same window) for the one defined by the heading or link tag.

Note: If a group number is assigned in both a heading and a hypertext or an automatic link, the link group number overrides the heading group number. The numbers you can assign to **group=** are integers from 1 to 64 000.

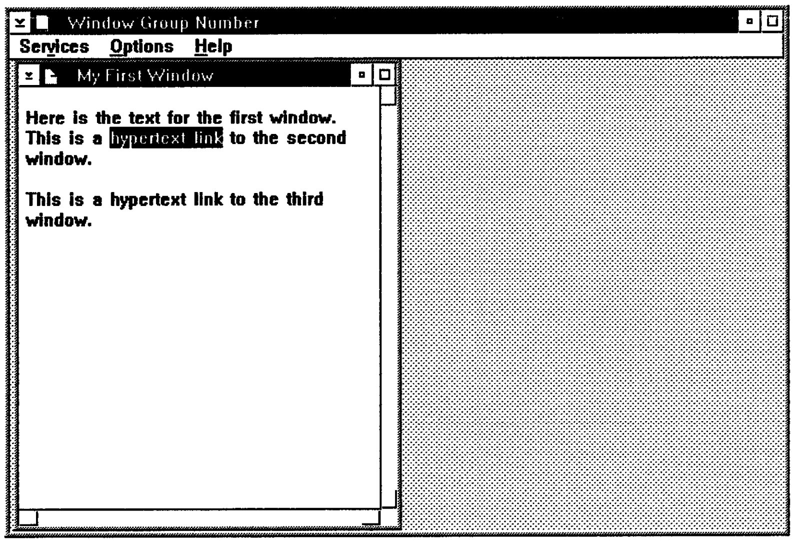
Compare the three heading definitions in Figure 6-6 on page 6-8. Notice that:

- The first and second windows have different group numbers and different positions.
- The second and third windows have the same group number.
- The second and third windows have the same size and position.

```
:h1 res=005
    x=left y=bottom
    width=50% height=100%
    group=1.
My First Window
:p.
Here is the text for the first window.
This is a
:link reftype=hd res=006.
hypertext link
:elink.
to the second window.
:p.
This is a
:link reftype=hd res=007.
hypertext link
:elink.
to the third window.
:h1 res=006
    x=right y=top
    width=50% height=100%
    group=2.
My Second Window
:p.
Here is the text for the second window.
This is a
:link reftype=hd res=005.
hypertext link
:elink.
to the first window.
:p.
This is a
:link reftype=hd res=007.
hypertext link
:elink.
to the third window.
:h1 res=007
    x=right y=top
    width=50% height=100%
    group=2.
My Third Window
:p.
Here is the text for the third window.
This is a
:link reftype=hd res=005.
hypertext link
:elink.
to the first window.
:p.
This is a
:link reftype=hd res=006.
hypertext link
:elink.
to the second window.
```

Figure 6-6. Source File for Window Group Number

Now assume that the source file shown in Figure 6-6 is compiled, and the user selects "My First Window" from the Contents window. The window in Figure 6-7 displays.

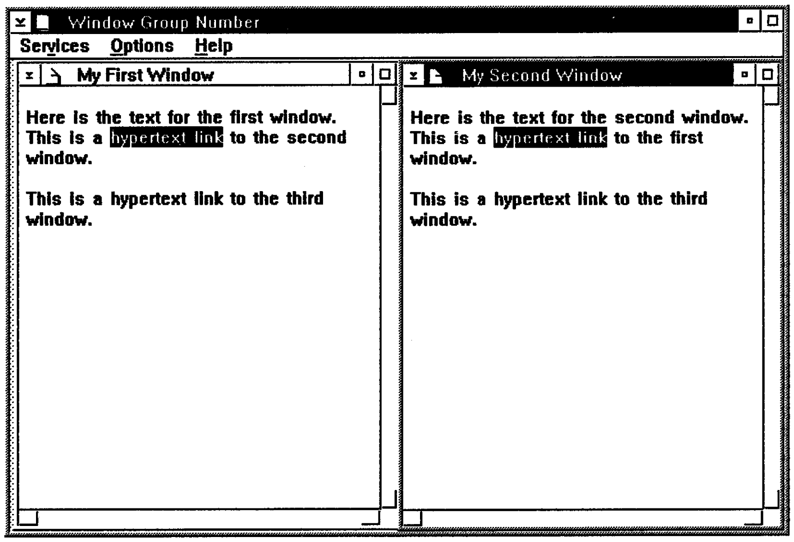


Here is the text for the first window. This is a hypertext link to the second window.

This is a hypertext link to the third window.

Figure 6-7. Multiple windows display with different group numbers.

If the user selects the hypertext link in this window, "My Second Window" will appear, as shown in Figure 6-8.



Here is the text for the first window. This is a hypertext link to the second window.

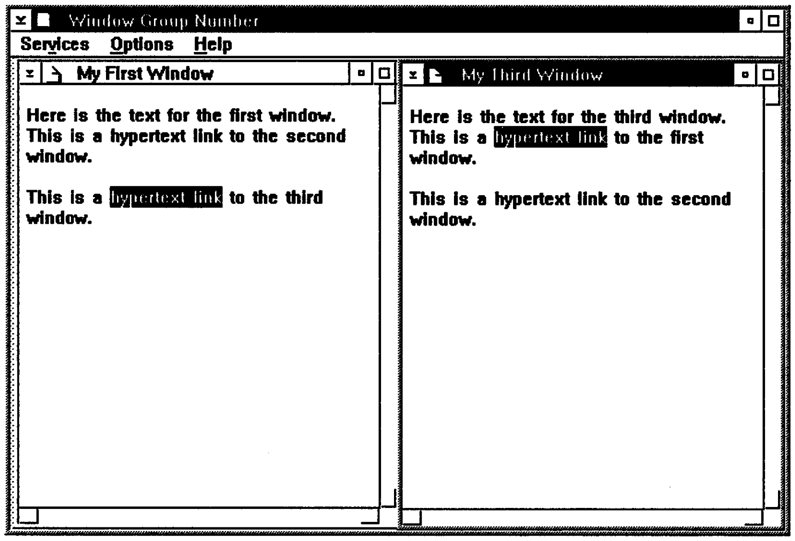
This is a hypertext link to the third window.

Here is the text for the second window. This is a hypertext link to the first window.

This is a hypertext link to the third window.

Figure 6-8. Multiple windows display with same group numbers.

The windows appear next to each other because their heading definitions specify different group numbers. If the user now selects the hypertext link in "My Second Window," the resulting screen will be as shown in Figure 6-9.



Here is the text for the first window.
This is a hypertext link to the second window.

This is a hypertext link to the third window.

Here is the text for the third window.
This is a hypertext link to the first window.

This is a hypertext link to the second window.

Figure 6-9. Compiled output of third window from group number.. "My Third Window" replaced "My Second Window" because it has the same group number as "My Second Window."

Preventing Image Swapping in Windows

The **group=** attribute opens a new window only if no other window with the same group number is already displayed. When a window is opened and a user selects another window with the same group number, IPF swaps its image in the already opened window. To prevent this, use the **viewport** attribute; it *always* opens a window.

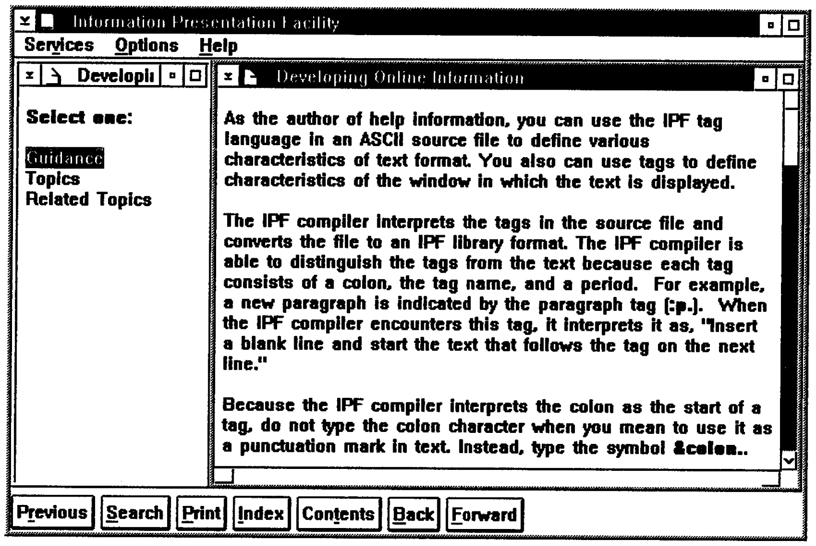
Suppose you have defined the following hypertext link to a window:

```
:link reftype=hd res=001.
      vpx=25% vpy=bottom
      vpcx=75% vpcy=100%
      viewport group=2.
Guidance
:elink.
```

When this window is displayed, if the user selects the same hypertext link, the same window will open. You cannot control how many times the user will select a hypertext link. If you do not want another window opened each time the user selects the same hypertext link, use the **group=** attribute instead of the **viewport** attribute. This eliminates the potential for the user to open multiple windows containing the same information.

Linking to a Window Automatically

As we have seen, one way to display a secondary window is to enable the user to select a hypertext link from one window to another. Another way is to link the user to the secondary window automatically. For example, in Figure 6-10 on page 6-11 the window on the right is displayed automatically when the user selects the window on the left (perhaps from the Contents window).


Information Presentation Facility
Services Options Help
Developli
Developing Online Information
Previous Search Print Index Contents Back Forward

Select one:

Guidance
Topics
Related Topics

As the author of help information, you can use the IPF tag language in an ASCII source file to define various characteristics of text format. You also can use tags to define characteristics of the window in which the text is displayed.

The IPF compiler interprets the tags in the source file and converts the file to an IPF library format. The IPF compiler is able to distinguish the tags from the text because each tag consists of a colon, the tag name, and a period. For example, a new paragraph is indicated by the paragraph tag (:p.). When the IPF compiler encounters this tag, it interprets it as, "Insert a blank line and start the text that follows the tag on the next line."

Because the IPF compiler interprets the colon as the start of a tag, do not type the colon character when you mean to use it as a punctuation mark in text. Instead, type the symbol &colon..

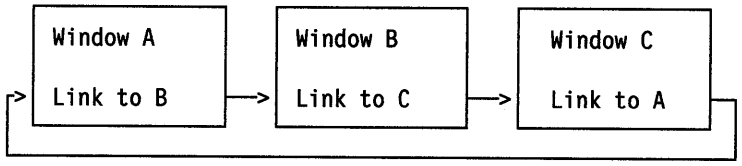
Figure 6-10. Example of a window displayed automatically. The window on the right was displayed automatically.

Auto Attribute: A window that starts the concurrent display of one or more other windows by automatic or hypertext links is referred to as the *owner* of the window chain. The **auto** attribute and the **reftype=hd** attribute indicate that a window is to be opened automatically whenever the owner window is opened. The **group=** attribute specifies the number of the window. (For more information about group numbers, see "Displaying Multiple Windows" on page 6-7.)

The **vpx**, **vpy**, **vpcx**, and **vpcy** attributes indicate the size and position of the window in relation to its coverpage window.

CAUTION:
When defining automatic links, you do not want to create an "infinite loop" by linking to the same window or group number more than once in a chain of links.

For example, suppose you create three windows, A, B, and C, that contain the following automatic links.



When the file containing these links is compiled, the IPF compiler does not return an error message because of the loop. Now suppose **Window A** is an entry in the Contents window and the user selects it. Windows A, B and C open and close uncontrollably until an error occurs and the process is terminated by the system.

Closing a Window Automatically

The **dependent** attribute causes the window to close automatically when the owner window is closed.

In the following example, the link at the end of the heading definition defines the owner window on the left. It links to the window on the right. Notice the link tag defining the automatic link does not require **:elink.**.

```
:h1 res=421
       x=left y=bottom
        width=25% height=100%
       group=1.
Developing Online Information
:link reftype=hd res=422
      vpx=right vpy=bottom
      vpcx=75% vpcy=100%
      auto dependent group=2.
⋮
:h1 res=422.Developing Online Information
```

Figure 6-11. Sample tagging defining an automatic link.

Tagging Example for Automatic Windows

The example shown in Figure 6-12 on page 6-13, defines two automatic window chains. A window chain has at least one owner window, and an owner window has one or more automatic or hypertext links to other windows in the chain. When an owner window closes, the windows in its chain that have specified the **dependent** attribute also close.

In "Example 1," (see Figure 6-13 on page 6-15) the only owner window in the chain is the first window (**res=008**). It contains links to three other automatic windows, which are referred to as *sibling windows* of the owner window.

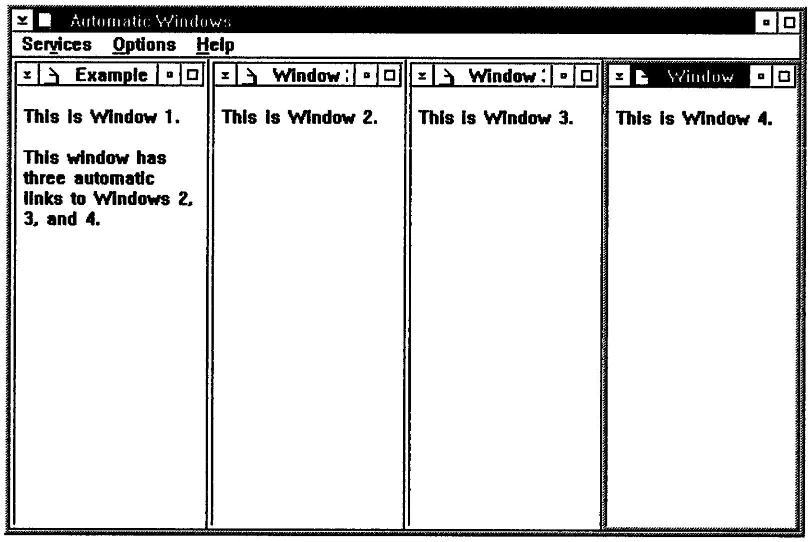
In "Example 2," (see Figure 6-14 on page 6-15) Windows 1 through 3 in the chain are owner windows. Window 1 owns all the windows in the chain and can close all of them. Window 2 also owns Windows 3 and 4. Window 3 also owns Window 4, the last window in the chain, which is displayed by means of a hypertext link in the text.

```
:h1.Automatic Windows
:h2 res=008
    x=left y=top width=25% height=100%
    scroll=none group=1 clear.
Example 1
:link reftype=hd res=009
      vpx=25% vpy=top vpcx=25% vpcy=100%
      group=2 auto dependent.
:link reftype=hd res=010
      vpx=50% vpy=top vpcx=25% vpcy=100%
      group=3 auto dependent.
:link reftype=hd res=011
      vpx=75% vpy=top vpcx=25% vpcy=100%
      group=4 auto dependent.
:p.
This is Window 1.
:p.
This window has three automatic links to
Windows 2, 3, and 4.
:h2 res=009
    x=25% y=top width=25% height=100%
    scroll=none hide.
Window 2
:p.
This is Window 2.
:h2 res=010
    x=50% y=top width=25% height=100%
    scroll=none hide.
Window 3
:p.
This is Window 3.
:h2 res=011
    x=75% y=top width=25% height=100%
    scroll=none hide.
Window 4
:p.
This is Window 4.
```

Figure 6-12 (Part 1 of 2). Sample tagging for automatic windows.

```
:h2 res=012
    x=left y=top width=25% height=100%
    scroll=none group=1 clear.
Example 2
:link reftype=hd res=013
      vpx=25% vpy=top vpcx=25% vpcy=100%
      group=2 auto dependent.
:p.
This is Window 1.
:p.
This window has an automatic link to
Window 2.
:h1 res=013
    x=25% y=top width=25% height=100%
    scroll=none hide.
Window 2
:link reftype=hd res=014
      vpx=50% vpy=top vpcx=25% vpcy=100%
      group=3 auto dependent.
:p.
This is Window 2.
:p.
This window has an automatic link to
Window 3.
:h1 res=014
    x=50% y=top width=25% height=100%
    scroll=none hide.
Window 3
:p.
This is Window 3.
:p.
This paragraph contains a
:link reftype=hd res=015
      vpx=75% vpy=top vpcx=25% vpcy=100%
      group=4 dependent.
hypertext link
:elink.
to Window 4.
:h1 res=015
    x=75% y=top width=25% height=100%
    scroll=none hide.
Window 4
:p.
This is Window 4.
```

Figure 6-12 (Part 2 of 2). Sample tagging for automatic windows.



This is Window 1.

This window has three automatic links to Windows 2, 3, and 4.

This is Window 2.

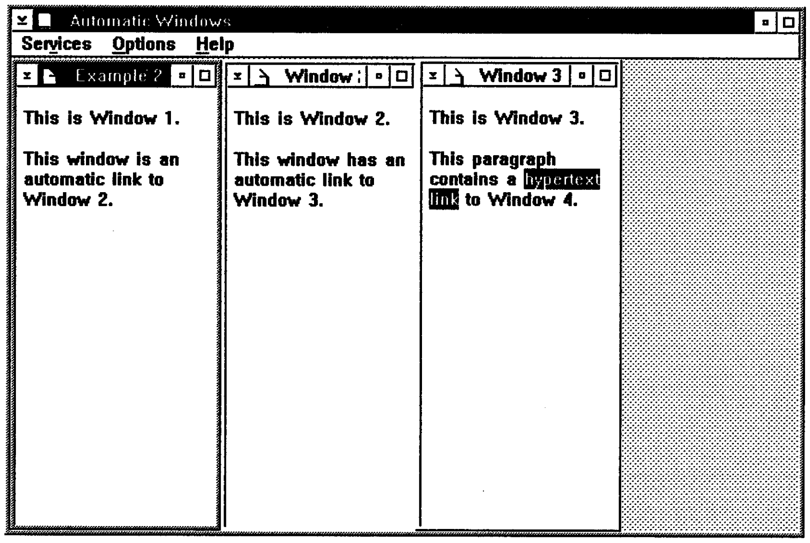
This is Window 3.

This is Window 4.

Figure 6-13. Example of four automatic windows.

When "Example 1" is selected from the Contents window, four windows are displayed in rapid succession. When Window 1 is closed, all four windows close.

Figure 6-14 shows the windows that are displayed when "Example 2" is selected from the Contents window.



This is Window 1.

This window is an automatic link to Window 2.

This is Window 2.

This window has an automatic link to Window 3.

This is Window 3.

This paragraph contains a hypertext link to Window 4.

Figure 6-14. Example of three automatic windows.

Notice Window 4 is not displayed. To display Window 4, you must select the hypertext link in Window 3.

Note: You can use the **viewport** attribute on an automatic link, because an automatic link is made only once.

Split Windows

A group of windows can be given the semblance of one window and yet offer the advantage of different windows; for example, text can be displayed next to an object the text describes. The author creates this effect by defining a window that consists of:

- **:h1.** or **:h2.** primary-window heading tags, followed by automatic links to secondary windows. (Text is not allowed.)
- **:h2.** secondary-window heading tags, each followed by text.

The primary window and its secondary windows must reside in the same file.

The position and size of the primary window determines the boundaries for its secondary windows. If the position and size of a secondary window are defined in absolute values that exceed the perimeter of the primary window, the secondary window is clipped. (When a window is clipped, part of it lies outside the window boundary and cannot be viewed.)

Sizes of secondary windows can be defined as percentages of the primary-window size. The minimum size of a secondary window (expressed in percentages) is zero height by zero width. Negative values for origin and position are not allowed.

Defining Split Windows

The primary window cannot have any text or graphics, only automatic links to each of its secondary windows. Each automatic link to a secondary window requires the **auto** and **split** attributes. The following is an example of the tagging for a primary window that contains a split window:

```
:h1 res=001 scroll=none.Primary Window A
:link reftype=hd res=002 auto split group=10
      vpx=left vpy=top vpcx=50% vpcy=100%
      scroll=none titlebar=none.
:link reftype=hd res=003 auto split group=11
      vpx=right vpy=top vpcx=50% vpcy=100%
      scroll=vertical titlebar=none.
```

The primary window does not have text and does not need a scroll bar; thus, the heading tag attribute is **scroll=none**. The primary window can define an overall title bar and disable the title bars in the secondary windows.

CAUTION:
When defining split windows, do not link to a footnote from a secondary-window.

For example, the text of a secondary window cannot have a link such as the following:

```
:link reftype=fn
      refid=001.
Display Pop-Up Window
:elink.
```

Tagging Example for Split Windows

The examples in Figure 6-15 on page 6-18, and Figure 6-18 on page 6-20, show the tagging for two different split-window arrangements.

In Figure 6-15 on page 6-18, "Primary Window A" (**res=016**) has automatic links to two secondary windows, (**res=017** and **res=018**). The tagging for **res=017** has two list items, each of which is a hypertext link. The first list item, "Ducks," links to **res=018**; the second item, "World," links to **res=019**.

The tagging for both **res=018** and **res=019** refer to bit-map files.

Notice that in "Primary Window A" the link tags for the secondary windows specify **titlebar=none**, but the heading tags for the secondary windows specify "Dummy" as title text. You must always provide IPF with a title string in a heading tag, even when you specify that the window will not have a title bar and will not have an entry in the Contents window because you have specified the **hide** attribute. The link tags for a hypertext link to a secondary window must specify the **split** attribute. If the **split** attribute is omitted, the window will not behave as a secondary window; that is, it will not close when the primary window is closed, and instead of moving when the primary window is moved, it will become obscured.

Hide, Noprint, and Nosearch Attributes: In the examples, each secondary window heading has the **hide**, **noprint**, and **nosearch** attributes. The **hide** attribute prevents an entry from appearing in the Contents window. You do not want a secondary window (in a split-window arrangement) to be displayed by itself; you want it displayed only when the Contents entry for its primary window is selected.

The **nosearch** attribute prevents the title string of the secondary window from being listed as an entry in the Search Results window. The Search option of IPF also searches the secondary window (for a word or phrase) because of the link definition in the primary window; however, only the title string of the primary window is returned in the Search Results window.

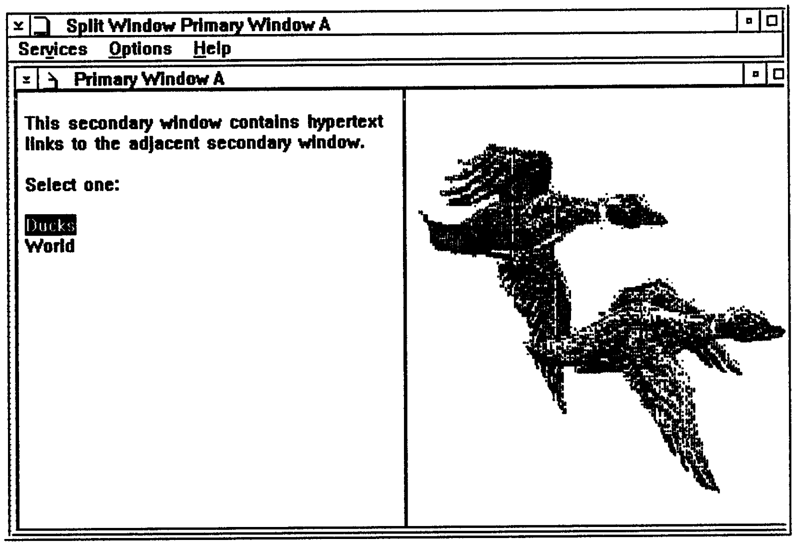
The Print option of IPF enables the user to print one or more topics, the index, or the table of contents. The **noprint** attribute in a primary-window heading prevents the contents of a secondary window from being printed. Secondary windows are printed as part of their primary window. The contents of secondary windows are printed only in the order in which the link definitions appear in the primary-window definition.

None of the primary-window heading tags specifies a group number with the **group=** attribute, so IPF assigns 0 (the default) as the group number of each. The **clear** attribute causes the screen to be cleared of windows before each split window is displayed.

```
:h1 res=016 scroll=none clear.
Primary Window A
:link reftype=hd res=017 auto split group=10
      vpx=left vpy=top vpcx=50% vpcy=100%
      rules=border scroll=none titlebar=none.
:link reftype=hd res=018 auto split group=11
      vpx=right vpy=top vpcx=50% vpcy=100%
      rules=border scroll=none titlebar=none.
:h2 res=017 hide nosearch noprint.Dummy
:p.
This secondary window contains hypertext links
to the adjacent secondary window.
:p.
Select one:
:sl compact.
:li.:link reftype=hd res=018 split group=11
          vpx=right vpy=top vpcx=50% vpcy=100%
          rules=border scroll=none titlebar=none.
Ducks
:elink.
:li.:link reftype=hd res=019 split group=11
          vpx=right vpy=top vpcx=50% vpcy=100%
          rules=border scroll=none titlebar=none.
World
:elink.
:esl.
:h2 res=018 hide nosearch noprint.Dummy
:artwork name='ducks.bmp' fit.
:h2 res=019 hide nosearch noprint.Dummy
:p.
:artwork name='world.bmp' fit.
```

Figure 6-15. Tagging for split window primary window A.

Here are both views of the compiled version of Primary Window A

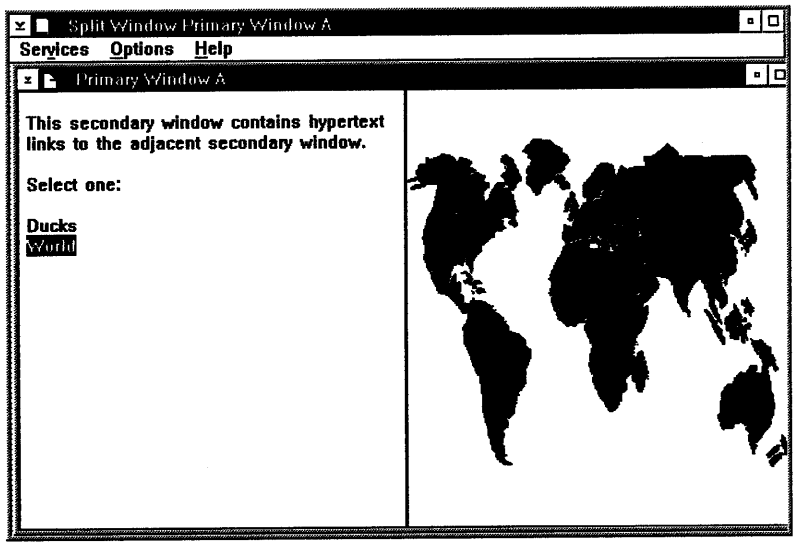


This secondary window contains hypertext links to the adjacent secondary window.

Select one:

Ducks
World

Figure 6-16. Example of a split window with an automatic link. The window on the right is displayed automatically when "Primary Window A" is selected from the Contents window.



This secondary window contains hypertext links to the adjacent secondary window.

Select one:

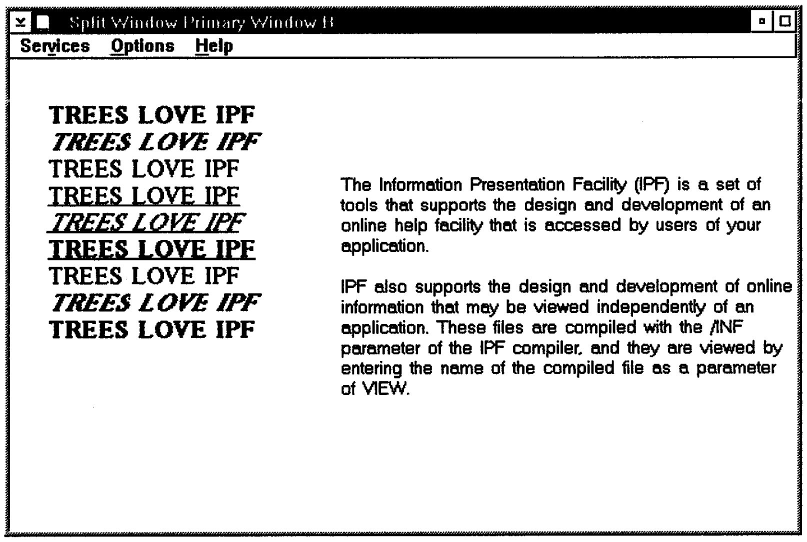
Ducks
World

Figure 6-17. Example of a split window with hypertext link. The window on the right is displayed when the second hypertext link is selected.

```
:h1 res=022 scroll=none titlebar=none rules=none clear.
Primary Window B
:link reftype=hd res=023 auto split group=10
      vpx=left vpy=top vpcx=40% vpcy=100%
      scroll=none titlebar=none rules=none.
:link reftype=hd res=024 auto split group=11
      vpx=right vpy=top vpcx=60% vpcy=20%
      scroll=none titlebar=none rules=none.
:link reftype=hd res=025 auto split group=12
      vpx=right vpy=bottom vpcx=60% vpcy=80%
      scroll=none titlebar=none rules=none.
:h2 res=023 hide nosearch noprint.Dummy
:lm margin=5.
:rm margin=2.
:p.
:font facename='Tms Rmn' size=24x12.
:color bc=green.:hp2.TREES LOVE IPF:ehp2.
:color bc=cyan.:hp3.TREES LOVE IPF:ehp3.
:color bc=green.:hp4.TREES LOVE IPF:ehp4.
:color bc=cyan.:hp5.TREES LOVE IPF:ehp5.
:color bc=green.:hp6.TREES LOVE IPF:ehp6.
:color bc=cyan.:hp7.TREES LOVE IPF:ehp7.
:color bc=green.:hp4.TREES LOVE IPF:ehp4.
:color bc=cyan.:hp3.TREES LOVE IPF:ehp3.
:color bc=green.:hp2.TREES LOVE IPF:ehp2.
:h2 res=024 hide nosearch noprint.Dummy
:p.
:h2 res=025 hide rules=none nosearch noprint.Dummy
:rm margin=3.
:font facename='Helv' size=18x9.
:p.
The Information Presentation Facility (IPF) is a set of tools
that supports the design and development of an online help
facility that is accessed by users of your application.
:p.
IPF also supports the design and development of online
information that may be viewed independently of an application.
These files are compiled with the /INF parameter of the IPF
compiler, and they are viewed by entering the name of
the compiled file as a parameter of the VIEW program.
```

Figure 6-18. Tagging for split window primary window B.

Here is the compiled version of Primary Window B.



TREES LOVE IPF
TREES LOVE IPF
TREES LOVE IPF
TREES LOVE IPF
TREES LOVE IPF
TREES LOVE IPF
TREES LOVE IPF
TREES LOVE IPF
TREES LOVE IPF

The Information Presentation Facility (IPF) is a set of tools that supports the design and development of an online help facility that is accessed by users of your application.

IPF also supports the design and development of online information that may be viewed independently of an application. These files are compiled with the /INF parameter of the IPF compiler, and they are viewed by entering the name of the compiled file as a parameter of VIEW.

Figure 6-19. Example of a split window without window controls. You cannot see the boundaries of the three windows because the window controls were eliminated. For a color representation of this screen, see the online *IPF Reference* available with the Toolkit.

Push Buttons for Split Windows: Be careful when using heading tags to define a control area for split windows. A control area cannot be defined in the secondary window heading tag of a split window. You must define the control area (the coverpage window) in the primary window heading tag. In the previous examples of split windows, the push button feature of IPF was disabled (see "Disabling the Display of Push Buttons" on page 3-11).

Summary Tables of Attribute Values for Origin and Size

The following tables summarize the attribute values that define a window's origin and size. Values shown in uppercase are keywords (words with special significance to IPF). Values shown in *lowercase italics* are to be substituted with your own values. Values are stacked when more than one value can be used with the attribute.

Heading (:h*n*) Tag	
Attribute = Value	**Description**
X=CENTER LEFT RIGHT	Specifies the location of the x-axis. The x-axis runs horizontally from left to right.
Y=CENTER TOP BOTTOM	Specifies the location of the y-axis. The y-axis runs vertically from bottom to top.
WIDTH=*an integer followed by the percent sign* (%)	Specifies the width (horizontal space) of a window.
HEIGHT=*an integer followed by the percent sign* (%)	Specifies the height (vertical space) of a window.

Link (:link.) Tag	
Attribute = Value	**Description**
VPX=CENTER LEFT RIGHT	Specifies the location of the x-axis. The x-axis runs horizontally from left to right. Overrides the x-axis attribute value specified by the heading tag.
VPY=CENTER TOP BOTTOM	Specifies the location of the y-axis. The y-axis runs vertically from bottom to top. Overrides the y-axis attribute value specified by the heading tag.
VPCX=*an integer followed by the percent sign* (%)	Specifies the width (horizontal space) of a window. Overrides the **width** attribute value specified by the heading tag.
VPCY=*an integer followed by the percent sign* (%)	Specifies the height (vertical space) of a window. Overrides the **height** attribute value specified by the heading tag.

Point to Remember: Origin and size attributes in a link tag override the origin and size attributes in a heading tag.

Summary Table for Heading Attributes

This table summarizes the heading attributes that support a multiple window format.

Heading (:h*n*) Tag	
Attribute = Value	**Function**
res = id = name = global tutorial	Define references to internal and external sources.
x = y = width = height =	Define the origin and size of a window in relation to its coverpage or primary window.
titlebar = scroll = rules =	Define the control the user has over the window.
viewport group = clear	Manage the display of information in multiple windows.
hide nosearch noprint	Restrict user retrieval of information.
toc =	Change heading levels appearing in the Contents window.

Summary Table for Link Attributes

This table summarizes the link attributes that support a multiple window format.

Link (:link.) Tag	
Attribute = Value	**Function**
reftype = res = refid = database = object = data =	Define references to internal and external sources.
vpx = vpy = vpcx = vpcy =	Define the origin and size of a window in relation to its coverpage or primary window.
titlebar = scroll = rules =	Define the control the user has over the window.
viewport group = dependent auto split	Manage the display of information in multiple windows.

Points to Remember: Link-tag attributes that have the same functions as those specified in a heading tag will override the heading-tag attributes. Although link-tag attributes have different names for x- and y-coordinates and window width and height, they provide the same functions.

Chapter 7. Compiling Source Files

This chapter explains how to prepare your source files so that they will be recognized by the IPF compiler. This chapter also shows you how to enter the compile command, how to interpret error messages, and how to view the compiled document. A section on national language support is also provided.

Source File Requirements

Using a single source file, you can produce a successful display of information with a limited number of tags. These tags are:

```
:userdoc.
:docprof.
:title.
:h1.
:p.
:euserdoc.
```

The **:userdoc.** tag is always the first item in your source file. It identifies the beginning of an IPF file. This tag is a signal to the IPF compiler to begin translating the tag language.

The **:euserdoc.** tag is required as the last line of the file to signal the end of the tagged document.

Place the **:docprof.** tag at the beginning of your source file after the **:userdoc.** tag and before any heading definitions. Use the function of the **toc** (table of contents) attribute of the **:docprof.** tag to control the heading levels displayed in the Content window. For example, if you want only heading levels 1 and 2 to appear, the tagging is:

:docprof toc=12.

If no **toc=** value is specified, heading level 1 through 3 appear in the Contents window.

Not to be confused with window titles, the text string specified with a **:title.** tag is placed into the title bar of an online document. When the online document is displayed, the title appears on the title line of the main window. The tagging looks like this:

:title. Endangered Mammals

The maximum length of a title string specified with a **:title.** tag is 47 characters, including spaces and blanks.

The title tag provides a name for the online document but is also used for titles of Help windows. The title appears in the title bar of the main window. You usually place the title tag after the **:docprof.** tag.

Every file must start with a **:h1.** tag. Heading level sequences must not skip a level in the heading hierarchy. For example, you cannot have a heading level 1 tag (**:h1.**) followed by a heading level 3 tag (**:h3.**).

You must have at least one paragraph tag (**:p.**) and associated text to display a window.

Figure 7-1 shows an IPF source file:

The source file contains a :userdoc.
tag, a :title. tag, a heading tag with a window
identifier, a :p.. tag, and the
:euserdoc. tag.

```
.*
:userdoc.
:title.Endangered Mammals
:h1 res=001.The Manatee
.*
:p.
The manatee has a broad flat tail and two flipper
like forelegs.  There are no back legs.
The manatee's large upper lip is split in two and
can be used like fingers to place food into the
mouth.  Bristly hair protrudes from its lips,
and almost buried in its hide are small eyes, with
which it can barely see.
.*
:euserdoc.
```

Figure 7-1. Source file structure

Source File Limits

- The maximum size of a line in an IPF source file is 255 characters. Each source file cannot exceed 64KB of data.
- The maximum number of fonts in a source file is 16.

Naming Conventions

It is a good idea to give your source files an extension of IPF, so they can be distinguished from your other files. For example:

```
MYHELP.IPF
```

The IPF compiler does not require an IPF file-name extension; however, if your file has an IPF file-name extension, you will not have to type the extension at compile time.

Naming Restriction

During the compilation process, IPF creates some files to hold data temporarily, and erases the files when it no longer needs them. The names of these files are:

filename.mdf
filename.clf
0000
2222

where *filename* is the name of your source file. Do not give your source file an MDF or CLF extension. Also, do not give your source file a name of 0000 or 2222.

Using a Base Source File

The IPF compiler can produce a single output document by processing multiple input files through one base source file. This process is most often associated with online documents. For example, the online *Information Presentation Facility Reference* has more than ten separate source files, but all the files were processed through one base file.

The **im.** (imbed) control word sends a signal to the compiler and tells it to process each file in the sequence listed in the base file.

A portion of the base file IPFCBASE.IPF for the online *Information Presentation Facility Reference* looks like this:

:userdoc.
⋮
.im ipfcch01.ipf
.im ipfcch02.ipf
.im ipfcch03.ipf
⋮

The placement of an imbedded file determines the order of entries in the table of contents.

Imbedded files cannot use the **:userdoc.** or **:euserdoc.** tags.

Note: When using a base source file to process multiple files, enter the base file name as the *filename* parameter of the IPFC command.

Starting the IPF Compiler

You can start the IPF compiler and specify all input from the command line. An example of the syntax follows:

```
IPFC filename [/INF] [/S] [/X] [/W] [> messageoutputfilename]
```

where:

filename
Specifies the name of your IPF source file or base file.

If you do not give a file-name extension, the IPF compiler uses .IPF by default. If your file has a file-name extension other than IPF, include that file-name extension in the command line.

/INF
Compiles the source file as an online document.

If this parameter is not included, the default is to compile the source file as a help library, whose extension is .HLP.

/S
Suppresses the performance of the Search function. This parameter increases compression of compiled data by about 10% to further reduce the storage it requires.

/X
: Generates and displays a cross-reference list.

/W*n*
: Generates and displays a list of error messages. The *n* indicates the level of error messages you want to receive. Values you can specify for *n* are 1, 2, or 3. For more information, see "Interpreting IPFC Error Messages" on page 7-5.

messageoutputfilename
: Specifies the name of the file where error and cross reference messages are sent. If you do not specify this parameter, messages generated by **/X** and **/W*n*** are sent to the display screen.

Compiling Help Files

To compile a source file that is intended as a help-text window, use the IPFC command without the /INF option. For example:

```
IPFC myhelp.hlp
```

Compiling with International Language Considerations

The following parameters provide international language support:

/COUNTRY=*nnn* (*nnn* is the 3-digit country code)

/CODEPAGE=*nnn* (*nnn* is the 3-digit code page)

/LANGUAGE=*xxx* (*xxx* is a 3-letter identifier that indicates an international languages file is to be used).

An example of the command-line syntax follows:

```
IPFC myfile.txt /INF /COUNTRY=033 /CODEPAGE=437 /LANGUAGE=FRA
```

For more information, see "National Language Support" on page 7-7.

Viewing an Online Document

If you want to see your formatted online document, you can use the VIEW command to display it.

An online document has an extension of INF. It can be viewed by entering its name as a parameter to the VIEW command; for example:

```
VIEW myfile
```

You do not need to include the INF file extension.

Note: You cannot use VIEW to display help-text windows for application programs.

Where IPFC Files are Stored

When you first install the Toolkit, the following environment variable is placed into the CONFIG.SYS file:

```
IPFC=C:\TOOLKT20\IPFC
```

The IPFC environment variable identifies the directory in which data files needed by the IPF compiler are stored.

When you first install the system, the following environment variable is placed into the CONFIG.SYS file:

HELP=C:\OS2\HELP

The HELP environment variable identifies the location of .HLP libraries.

BOOKSHELF=C:\OS2\BOOK

The BOOKSHELF environment variable identifies the location of online documents and is used by VIEW.

Concatenating .INF Files

Concatenation of .INF files is useful when you have a large amount of information that cannot be compiled as one file that fits on a diskette. If you want to concatenate files, you must use the **res=** attribute for window identifiers.

After you have created your .INF files, use the SET command to set an environment variable equal to a string that consists of the .INF file names, for example:

```
SET PROGREF=PRINTRO.INF+PRCP.INF+PRWIN.INF+PRDATA.INF
```

When you specify the environment value as a parameter to the VIEW program, VIEW displays the online information. Headings from the different files are displayed in the contents window in the order the files are concatenated for the environment variable.

Interpreting IPFC Error Messages

The **/W***n* parameter of the IPFC command determines the levels of error messages that will be displayed. Following are the values that you can specify for *n*:

Value	Meaning
1	Returns only warning level 1 messages. Warning level 1 messages are the most severe.
2	Returns warning level 1 and 2 messages.
3	Returns all three warning levels of messages. This is the default. Warning level 3 messages are the least severe.

When IPF compiles your file, it generates and displays the error messages. If no errors are found, IPF tells you that compiling has been completed and no errors were found.

You may prefer to redirect error messages from the screen to an error file. You could enter the IPFC command like this:

```
IPFC myhelp /w3 > myhelp.err
```

If you have also requested that a cross-reference list be created by specifying the **/X** parameter, it will be included in the MYHELP.ERR file.

For a list of error messages that the IPF compiler returns, see Appendix A, "Compiler Error Messages."

Differences between .HLP and .INF Files

	Help Libraries	**Online Documents**
IPFC Command Syntax:	IPFC filename	IPFC filename /INF
Compiled File Extensions:	.HLP	.INF
Environment Variables Used by the VIEW Program:		BOOKSHELF= defines the location of .INF files
Environment Variables Used by IPF for Help Windows:	HELP= defines the location of .HLP libraries.	
Cause of Interface Display:	An application user's request for help.	Entering the file name as a parameter to the VIEW utility.
Initial Size of Main Window:	35% of screen (default)	85% of screen (default)
Initial Contents of Main Window:	Response to help request	Contents window (default)
Main Window Title:	Defined by the programmer in the HELPINIT structure.	Defined by the :title. tag, which is placed on the line following the :userdoc. tag.
External Links: (See global attribute of heading tag and database attribute of :link. tag.)	.HLP files can link to other .HLP files and also to .INF files.	.INF files can link only to other .INF files.
To View Concatenated Files:	Specify a string of .HLP files in the HELPINIT structure.	Set an environment variable equal to a string of .INF file names.

Note: Defaults may be overridden by objects that are displayed in application-controlled windows.

National Language Support

The following parameters provide national language support (NLS):

/COUNTRY
/CODEPAGE
/LANGUAGE

If you do not specify these parameters, the default for /COUNTRY and /CODEPAGE are the values specified in your CONFIG.SYS file. If you do not request that an NLS file be used, the titles for the tags listed in the parameter description are shown in United States English.

Country Code Pages

The following table lists the 3-digit country code for the /COUNTRY parameter of the IPFC command. The third column lists the numeric identifiers of code pages supported:

COUNTRY	COUNTRY CODE	CODE PAGES
Australia	061	437, 850
Belgium	032	437, 850
Canadian English	001	437, 850
Canadian French	002	863, 850
Chinese	088	938, 437, 850
Denmark	045	865, 850
Finland	358	437, 850
France	033	437, 850
Germany	049	437, 850
Italy	039	437, 850
Japan	081	932, 437, 850
Korea	082	934, 437, 850
Latin America	003	437, 850
Netherlands	031	437, 850
Norway	047	865, 850
Portugal	351	860, 850
Spain	034	437, 850
Sweden	046	437, 850
Switzerland	041	437, 850
United Kingdom	044	437, 850
United States	001	437, 850

The following table lists the 3-letter identifier for the /LANGUAGE parameter of the IPFC command:

ID	LANGUAGE	NLS FILE	APS FILE
CHT	Chinese	IPFCHT.NLS	APSYM938.APS
DAN	Danish	IPFDAN.NLS	APSYMBOL.APS
DEU	German	IPFDEU.NLS	APSYMBOL.APS
ENG	English UK	IPFENG.NLS	APSYMBOL.APS
ENU	English US	IPFENU.NLS	APSYMBOL.APS
ESP	Spanish	IPFESP.NLS	APSYMBOL.APS
FIN	Finnish	IPFFIN.NLS	APSYMBOL.APS
FRA	French	IPFFRA.NLS	APSYMBOL.APS
FRC	Canadian French	IPFFRC.NLS	APSYMBOL.APS
ITA	Italian	IPFITA.NLS	APSYMBOL.APS
JPN	Japanese	IPFJPN.NLS	APSYM932.APS
KOR	Korean	IPFKOR.NLS	APSYM934.APS
NLD	Dutch	IPFNLD.NLS	APSYMBOL.APS
NOR	Norwegian	IPFNOR.NLS	APSYMBOL.APS
PTG	Portuguese	IPFPTG.NLS	APSYMBOL.APS
SVE	Swedish	IPFSVE.NLS	APSYMBOL.APS
UND	User defined	IPFUND.NLS	APSYMBOL.APS

Note: You must specify the appropriate symbols file, otherwise the system defaults to APSYMBOL.APS. However, double-byte character set (DBCS) countries (Japan, Korea, and China), MUST copy the appropriate symbol file to APSYMBOL.APS. The IPF compiler will not recognize a file by any other name. For a list of symbols, see Chapter 14, "Symbols."

Part 2: Programmer's Guide

Information Presentation Facility

Chapter 8. Enabling Help for Applications

While running an application the user sometimes requires help. For example, the user may need assistance in making a choice, recalling the name of an application command or the use of a function key, or locating information.

Using IPF, you can develop a user interface that provides general help for application windows, and contextual help for fields within windows.

Implementing the IPF user interface when creating helps for an application requires two different development efforts:

- Developing the programming code that communicates with IPF and the Presentation Manager to display help windows.
- Developing a library of help information that IPF refers to in response to a user request.

This section will concentrate on the first development effort: writing the programming code that enables communication between IPF and the Presentation Manager.

Developing the Application Code

Use the following steps to develop the application code that adds help to your application.

1. Set up the help table and help subtable, and include the help constants defined in PMHELP.H.
2. Initialize the HELPINIT structure with a call to DosLoadModule.
3. Create a help instance.
4. Associate the help instance with the application window chain.
5. Respond to messages for menu bar choices.
6. End the help instance.

The following sections describe how to implement each of these steps. Some of the steps are illustrated with example code from the JIGSAW sample program provided in the \TOOLKT20\C\SAMPLES\JIGSAW subdirectory.

Setting Up the IPF Help Tables

Two table structures in application memory or in resource files (.RC file-name extension), identify window resources in the IPF library. The help table associates each application window with its corresponding help subtable and the window identifier of its extended help window. The help subtable associates each entry field, menu item and push button within an application window with the window identifier (ID) of its help window. The address of the help table is passed to the application during initialization of the IPF initializing structure (HELPINIT).

When the user requests help on a field, menu bar, or push button in the application window, IPF uses the help subtable associated with the field to find the window ID of the contextual help window for the field. The help subtable also can store optional entries relating to application-specific information.

The maximum size of the help table is 64KB. The number of help subtables is limited to 16,000.

Help table and help subtable structures are contained in the PMHELP.H in the \TOOLKT20\C\OS2H subdirectory.

IPF supports two methods of defining help tables and help subtables. They can be allocated in memory, or they can be defined as resources. In either case, the information passed to IPF is identical.

Defining Help Tables in Memory

By defining help tables and subtables in memory, you can dynamically change a single entry in the help table. You can add a new window ID to be associated with a field, or add fields that are to be associated with existing windows.

After the help table structure is initialized, the application can pass IPF the address in memory of the new help table, either by sending the HM_CREATE_HELP_TABLE message from its window procedure, or by calling **WinCreateHelpTable**.

When defining help tables in memory, the data structures in PMHELP.H are used. The help table contains the structure for each application window. This structure holds the following information:

- Application window ID
- Address of the window's subtable
- Window ID of the window's extended help window.

These entries are integers. The last entry in the list contains a NULL for each entry type, to indicate the end of the list. The following is an example of a help table for an application.

```
HELPSUBTABLE       table1, table2, table3,
                   table4, table5;

HELPTABLE     helpTableEntry [] =
{
  APP_WIND_1,  &table1, idExtHelp1,
  APP_WIND_2,  &table2, idExtHelp2,
  APP_WIND_3,  &table3, idExtHelp3,
  APP_WIND_4,  &table4, idExtHelp4,
  APP_WIND_5,  &table5, idExtHelp5,
  0,           NULL,    NULL
};
```

The **help subtable** contains the structure defined in the PMHELP.H file for each application window. This structure holds the following information for each field in the application window:

- Field ID from which user requests help
- Window ID of the contextual help window associated with the field
- Optional, application-specific integers.

The last entry in the structure contains the word length for each field entry. The minimum number of words is two, which is the default. The following is an example of a help subtable for an application window that has six fields.

```
HELPSUBTABLE        HelpSubTable [] =
{
  2,
  FIELD_ID_1, IDRES_HELP1,
  FIELD_ID_2, IDRES_HELP2,
  FIELD_ID_3, IDRES_HELP3,
  FIELD_ID_4, IDRES_HELP4,
  FIELD_ID_5, IDRES_HELP5,
  FIELD_ID_6, IDRES_HELP6,
  0,          0
};
```

Defining Help Tables as Resources

If help tables are defined as resources, they can be bound to the application's executable file, or they can reside in a dynamic link library (DLL).

If help tables are defined as resources in a dynamic link library, the application must call DosLoadModule to load the DLL before it calls WinCreateHelpInstance. When the application calls WinCreateHelpInstance, it passes the handle to the DLL and the resource ID of the help table in the HELPINIT structure.

The application can load a new help table residing in the DLL by either sending the HM_LOAD_HELP_TABLE message from its window procedure, or by calling **WinLoadHelpTable**. The application passes the handle to the DLL and the resource ID of the new help table.

A HELPTABLE resource contains a HELPITEM entry for each application window, dialog, and message box for which help is provided.

Each entry of a HELPTABLE resource contains:

- HELPITEM keyword
- Application window ID
- ID of the HELPSUBTABLE resource
- Window ID of the extended help window.

A HELPSUBTABLE resource contains an entry for each item that can be selected in an application window. Each of these items is assumed to be a child window of the application window identified in the HELPTABLE resource. The HELPSUBTABLE should contain a single SUBITEMSIZE and a HELPSUBITEM for each control, child window, and menu item.

Each entry of a HELPSUBTABLE resource contains:

- HELPSUBITEM keyword
- Field ID
- Window ID of the field's help window (corresponds to the resource number specified in the heading tag of the help-text window)
- Optional, application-defined integers.

The integer ID of the field can be a control, menu item, or message box ID. The ID specified must be unique within the table. An ID of hex FFFF is reserved for use by IPF.

The optional integers value allows the writer of the resource script file to append additional integers to the end of each HELPSUBITEM for application-specific use.

The SUBITEMSIZE keyword is used to identify the size in words of each HELPSUBITEM. All entries must be the same length. If this value is specified, it must be greater than or equal to 2. If this value is not specified, it defaults to 2.

All referenced HELPSUBTABLE resources must reside in the same .RES file as the HELPTABLE resource.

Following is an example of .RC source file for defining a HELPTABLE and its related HELPSUBTABLE resources.

```
HELPTABLE TABLE_1
BEGIN
HELPITEM parentwindow1, SUBTABLE_1,
  extendedhelppanel1
HELPITEM parentwindow2, SUBTABLE_2,
  extendedhelppanel2
END

HELPSUBTABLE SUBTABLE_1
[SUBITEMSIZE subitemsize1]
BEGIN
  HELPSUBITEM FIELD_ID1, helppanel1 [,
    integer1, ...n]
  HELPSUBITEM FIELD_ID2, helppanel2 [,
    integer1, ...n]
END

HELPSUBTABLE SUBTABLE_2
[SUBITEMSIZE subitemsize2]
BEGIN
  HELPSUBITEM FIELD_ID3, helppanel3 [,
    integer1, ...n]
  HELPSUBITEM FIELD_ID4, helppanel4 [,
    integer1, ...n]
END
```

Initializing the HELPINIT Structure

Before you call WinCreateHelpInstance, you must allocate memory for and initialize the HELPINIT structure. This structure defines values that IPF needs to create the help instance. Some of the values can be changed by your application after initialization.

The HELPINIT structure and the help table structures referred to by IPF during help processing are contained in the PMHELP.H file. The PMHELP.H file also contains the error codes returned in the event of an unsuccessful call. You include this file in your source code by using the INCL_WINHELP define statement. The following shows the HELPINIT structure.

```
typedef struct _HELPINIT /* hinit */
{

ULONG          cb;
ULONG          ulReturnCode;
PSZ            pszTutorialName;
PHELPTABLE     phtHelpTable;
HMODULE        hmodHelpTableModule;
HMODULE        hmodAccelActionBarModule;
ULONG          idAccelTable;
ULONG          idActionBar;
PSZ            pszHelpWindowTitle;
ULONG          fShowPanelId;
PSZ            pszHelpLibraryName;
} HELPINIT;
```

Following are descriptions of the HELPINIT structure fields.

Field Name	**Description**
cb	The length of the initialization structure. This value can be use to identify the version of IPF being used.
ulReturnCode	The IPF return code
pszTutorialName	A pointer to a tutorial name, if one exists. If this value is NULL, either the application help interface does not include a tutorial, or the tutorial is referenced from a help window. If this value in not NULL, IPF provides a **Tutorial** choice in the help pull-down. If the user selects the **Tutorial** choice in the pull-down, IPF sends the HM_TUTORIAL message to the application so that it can start the tutorial.
phtHelpTable	A pointer to the help table.
hmodHelpTableModule	The name of the resource file that indexes the dynamic link library that contains the help table and its corresponding subtables. If the help table is not being accessed through a dynamic link library, this value is 0.
hmodAccelActionBarModule	The name of the dynamic link library that contains the modified menu bar. If you do not have a modified menu bar, this value is 0.
idAccelTable	The name of the accelerator table if you are using a modified menu bar; otherwise, this value is 0.
idActionBar	The identity of the menu bar (action bar) template. If you are not modifying the menu bar, this value is 0.
pszHelpWindowTitle	A pointer to the name of the title for the main help window. This name can be changed after initialization by sending the message HM_SET_HELP_WINDOW_TITLE.
fShowPanelId	A flag used to append the window ID to the beginning of the help window title in the title bar of the help window. If this flag is set to CMIC_SHOW_PANEL_ID, the window IDs are displayed. If this flag is set to CMIC_HIDE_PANEL_ID or to 0, the window IDs are not displayed.

This flag is useful during the development stages of the help interface.

After initialization, this flag can be toggled with the HM_SET_SHOW_PANEL_ID message.

pszHelpLibraryName The help library names of the .HLP files containing the help windows. These .HLP files are created by the IPF compiler. When IPF needs to search for a help window, it looks for these library names in the path set by the HELP environment variable. If IPF cannot find a library name in this path, it then searches the current directory.

After initialization, help library names can be specified with the HM_SET_HELP_LIBRARY_NAME message. If multiple libraries are specified, library names must be separated by a blank space.

The following shows how the help facility for JIGSAW is initialized. Notice that hmodAccelActionBarModule, idAccelTable, and idActionBar have values set to 0; this is because JIGSAW uses a standard menu bar.

```
VOID HelpInit (VOID)
{
   HELPINIT hini;

   /* if we return because of an error, Help will be disabled */
   fHelpEnabled = FALSE;

   /* initialize help init structure */
   hini.cb = sizeof (HELPINIT)
   hini.ulReturnCode = 0L;

   hini.pszTutorialName = (PSZ)NULL /*if tutorial added, add name here*/

   hini.phtHelpTable = (PHELPTABLE)MAKELONG(JIGSAW_HELP_TABLE, 0xFFFF);
   hini.hmodHelpTableModule = (HMODULE)0;
   hini.hmodAccelActionBarModule = (HMODULE)0;
   hini.idAccelTable = 0;
   hini.idActionBar = 0;

   if (!WinLoadString (habMain,
            (HMODULE)0,
            IDS_HELPWINDOWTITLE,
            HELPLIBRARYNAMELEN,
            (PSZ)szWindowTitle))
   {
     MessageBox (habMain, IDS_CANNOTLOADSTRING, MB_OK | MB_ERROR, FALSE);
     return;
   }

   hini.pszHelpWindowTitle = (PSZ)szWindowTitle;

   /*  if debugging, show panel ids; else, don't   */
#ifdef DEBUG
   hini.fShowPanelId = CMIC_SHOW_PANEL_ID;
#else
   hini.fShowPanelId = CMIC_HIDE_PANEL_ID;
#endif

   if (!WinLoadString (habMain,
           (HMODULE)0,
           IDS_HELPLIBRARYNAME,
           HELPLIBRARYNAMELEN,
           (PSZ)szLibName))
   {
   MessageBox (habMain, IDS_CANNOTLOADSTRING, MB_OK | MB_ERROR, FALSE);
   return;
   }

   hini.pszHelpLibraryName = (PSZ)szLibName;
```

Creating the Help Instance

The WinCreateHelpInstance call passes the HELPINIT structure defined in the PMHELP.H include file to the Presentation Manager. WinCreateHelpInstance returns a handle to the help instance, which you must store in a HWND variable for use with the rest of the application programming interface (API) function calls associated with IPF.

IPF responds to the WinCreateHelpInstance call by installing its help hook and initializing for help processing.

The following shows how the help instance is created for JIGSAW.

```
   /*  Creating help instance */
   hwndHelpInstance = WinCreateHelpInstance (habMain, &hini);

   if (!hwndHelpInstance || hini.ulReturnCode)
   {
   MessageBox (hwndFrame,
         IDS_HELPLOADERROR,
         MB_OK | MB_ERROR,
         TRUE);
   return;
   }
```

Associating the Instance with the Window Chain

After an application creates a help instance, it must associate the instance with the application window chain by calling **WinAssociateHelpInstance.** IPF uses the active window handle passed by this call to index into the help table to find the help window that should be displayed for the application window.

An IPF instance can be associated with any application window that has a frame. Once the association of an IPF instance with the application window chain is made, help can be requested for any application window in the chain.

The following shows how the help instance is associated with the application window chain for JIGSAW.

```
/*  associate the help instance with the main frame  */
if (!WinAssociateHelpInstance (hwndHelpInstance, hwndFrame))
{
MessageBox (hwndFrame,
      IDS_HELPLOADERROR,
      MB_OK | MB_ERROR,
      TRUE);
return;
}

/*  IPF is successfully initialized; set flag to TRUE  */
fHelpEnabled = TRUE;
```

Ending the Help Instance

To end the current help instance, the application calls **WinDestroyHelpInstance,** passing the handle of the help instance that is to be ended.

The parameter *hwndHelpInstance* is the handle to the IPF instance returned from the WinCreateHelpInstance call.

The following shows how the help instance is terminated by JIGSAW.

```
VOID DestroyHelpInstance (VOID)
{
   if (hwndHelpInstance)
   {
     WinDestroyHelpInstance (hwndHelpInstance);
   }
}
```

Responding to Messages for Menu Bar Choices

IPF communicates with the active window. This communication is accomplished with messages. The application may need to do some of its own processing in response to these messages.

Processing "Using help" Requests

When the user selects "Using help" from the help pull-down menu, a WM_COMMAND is sent to the application's window procedure.

If the application has created its own 'Using help' window, it responds by sending the HM_REPLACE_USING_HELP message with the help-window ID. If the application chooses to use the "Using help" window provided by IPF, it responds by sending the HM_DISPLAY_PANEL with NULL in both parameters.

Current CUA guidelines recommend applications use "Using help;" however, IPF continues to support the "Help for help" window.

Processing a "Keys Help" Request

When the user selects 'Keys help' from the help pull-down, an HM_KEYS_HELP message is sent by the application to IPF. In response, IPF sends an HM_QUERY_KEYS_HELP message to the application. The application returns the window ID of the keys help window.

Processing Help Requests for a Child Window

In the Presentation Manager, parent and child windows are active at the same time. Therefore, when a help instance is associated with a window, its descendants are included in the association. However, only the parent window is the active *help* window.

Note: Do not confuse child windows with dialog, message boxes, and other windows which the application may own but are actually children of the desktop.

For IPF to process help requests for a child window, an application must send IPF HM_SET_ACTIVE_WINDOW messages to set the active help window. Until this happens, IPF continues to satisfy help requests for the child window from the help subtable for the parent window.

The HM_SET_ACTIVE_WINDOW message should be sent by ALL windows in response to the WM_ACTIVATE and WM_INITMENU messages as shown in the following example.

```
switch( usMsg )
{
                          .
                          .
                          .
case WM_ACTIVATE:
   if( SHORT1FROMMP( mp1 ) )
   {
      /*
       * Set active help window to this window's parent when
       * activated
       */
      WinSendMsg( WinQueryHelpInstance( hWnd ),
                  HM_SET_ACTIVE_WINDOW,
                  WinQueryWindow( hWnd, QW_PARENT ),
                  WinQueryWindow( hWnd, QW_PARENT ) );
   }
   else
   {
      /*
       * clear active help window when this window is
       * deactivated - necessary for message box help, etc.
       * to work properly.
       */
      WinSendMsg( WinQueryHelpInstance( hWnd ),
                  HM_SET_ACTIVE_WINDOW,
                  NULL,
                  NULL );
   }
   break;

case WM_INITMENU:
   /*
    * Set active window to this window's parent here so that
    * the menu id will be found in the proper subtable.
    * Activation and deactivation of the help window will
    * take care of setting the help window back to the
    * active window.
    */
   WinSendMsg( WinQueryHelpInstance( hWnd ),
               HM_SET_ACTIVE_WINDOW,
               WinQueryWindow( hWnd, QW_PARENT ),
               WinQueryWindow( hWnd, QW_PARENT ) );
   break;

                          .
                          .
                          .
}
```

When No Help Is Available

A user may request help by pressing F1 when the cursor is positioned on an item for which no field-level help is available. In such a case, IPF sends the HM_HELPSUBITEM_NOT_FOUND message to the application. To display the extended help window, the application then can either return FALSE or ignore the message. If the application returns TRUE, there is no response to the user request.

Help Window Resources

You can define the following window resources for the help interface:

- Help pull-down
- Help push button
- Command entry field
- Customized menu bar.

Help Pull-Down

CUA guidelines recommend that all application windows with menu bars include a help pull-down menu. The help application menu bar choice and corresponding pull-down menu is defined in your resource file. The following example shows how to define the help pulldown.

```
MENU IDR_MAIN PRELOAD
BEGIN
    SUBMENU "~File",                        IDM_FILE
    BEGIN
        MENUITEM "~Open...",                IDM_LOAD
    END

    SUBMENU "~Options",                     IDM_OPTIONS
    BEGIN
        SUBMENU "Size",                     IDM_SIZE_MENU
        BEGIN
            MENUITEM "Small",               IDM_SIZE_SMALL,   0, MIS_TEXT
            MENUITEM "Medium",              IDM_SIZE_MEDIUM,  0, MIS_TEXT
            MENUITEM "Large",               IDM_SIZE_LARGE,   0, MIS_TEXT
            MENUITEM "Full Size",           IDM_SIZE_FULL,    0, MIA_CHECKED
        END

        MENUITEM "~Jumble!",                IDM_JUMBLE
    END

    SUBMENU "~Help",                        IDM_HELP,            MIS_TEXT
    BEGIN
        MENUITEM "Help ~index",             IDM_HELPINDEX,       MIS_TEXT
        MENUITEM "~General help",           IDM_HELPEXTENDED,    MIS_TEXT
        MENUITEM "~Using help",             IDM_HELPHELPFORHELP, MIS_TEXT
        MENUITEM     SEPARATOR
        MENUITEM "~Product information",    IDM_HELPABOUT,       MIS_TEXT
    END
END
```

Help Push Button

If your application has a dialog or message area, you may want to include the Help push button in the bottom area of the secondary application window (dialog box). To define the Help push button, use the Presentation Manager button style BS_HELP and BS_NOPOINTERFOCUS.

The BS_HELP style causes the Presentation Manager to call IPF when the user selects this Help push button. The BS_NOPOINTERFOCUS style enables the Presentation Manager to determine the field for which the user requested help.

Command Entry Field

An entry field is a control window that enables users to enter text. A command entry field is used for typing commands, and may be programmed to accept entries recognized by the application.

For example, a command entry field might be used in an interpreter with a Presentation Manager interface. The field would accept a request from the user and execute it. Similarly, a command entry field might be used in an editor in a "command mode" to accept advanced instructions not associated with any editing keys. Any time the user has a limited number of correct responses, a command entry field may be appropriate.

Help windows for application commands can be associated with a command entry field by imbedding the index command tag (**:icmd.**) with a command name in the window that describes the command.

When the cursor is positioned in the associated entry field and the user presses F1 or selects the **Help** push button, titles of windows that contain these tags are displayed in alphabetic order in a list box window.

For a complete description on entry fields, see the *Programming Guide, Volume 2.*

A Customized Menu Bar

A Help menu bar template is shipped with the Toolkit. The template is in the HMTAILOR.RC file. Included in the template is the Help menu pull-down. You can customize the menu bar by adding pull-downs and choices to the Help menu bar template.

When a menu bar or pull-down choice you have added is selected by the application user, IPF sends the HM_ACTION_BAR_COMMAND message to the currently active application window. The low-order word of *param1* contains the command value of the selected item. The command values of the actions added by the application must be between hex 7F00 and hex 7FFF for its commands.

The accelerator table maps function keys to commands on help windows. This table is also contained in the HMTAILOR.RC file. If you add a choice to the menu bar that maps to a key on the keyboard, you must also add an entry to the accelerator table for that choice. IPF functions depend on the entries that already exist in the shipped accelerator table. They must not be altered. The command value specified in the accelerator table entry must be the same command value that was specified for the associated action in the menu bar template.

If the HMTAILOR.RC file is changed, you must compile it using the resource compiler and attach it to the executable file. If the executable file is a DLL, you must call DosLoadModule to load it before calling WinCreateHelpInstance. Identify the handle to IPF in the *hmodAccelActionBarModule* field in the initialization structure. When this field is 0, IPF uses the default menu bar.

The HMTAILOR.RC file includes the HMTAILOR.H file.

Note: When modifying the menu bar, define IDM_HM_MENU and IDD_COVERPAGE_HM_ACCEL in your help header (.H) file. Also, add the IDs in the idActionBar and idAccelTable fields in the HELPINIT structure.

Chapter 9. Expanding the Scope of IPF

Rather than have IPF display information that has been interpreted by the IPF compiler, you can expand IPF by having the application call a routine in a dynamic link library (DLL). Help information can be customized by hooking a piece of Presentation Manager code into the IPF help facility. This means that help information can include function simulation, user interaction, animation, and audio and video presentations.

An IPF window can be controlled by IPF or by an IPF communication object written by a programmer. The IPF communication object determines what is displayed in an application-controlled window. However, the use of IPF communication objects is not limited to creating application-controlled windows; it also can change the function and size of the IPF coverpage window (see "The Coverpage Window" on page 9-3).

IPF communication objects also can request IPF to display the Table of Contents, the Help Index, the Master Help Index, or the list of Viewed Pages. IPF communication code can subclass any window by knowing its handle and installing a help hook.

IPF also provides dynamic data formatting routines. This function enables the application to establish a dialog with the user by formatting text responses in a window.

Application-Controlled Windows

IPF handles the formatting and display of text and graphic information within its windows. *IPF-controlled* windows are defined in the tagged source file with a heading tag or **:link.**. These windows are IPF-controlled because IPF provides the window procedures that control them. The content and presentation of information in an IPF-controlled window is limited by the functions of a standard OS/2 window.

To create IPF-controlled windows, an author requires only tagging skills. However, to create *application-controlled* windows, an author requires both tagging and programming skills.

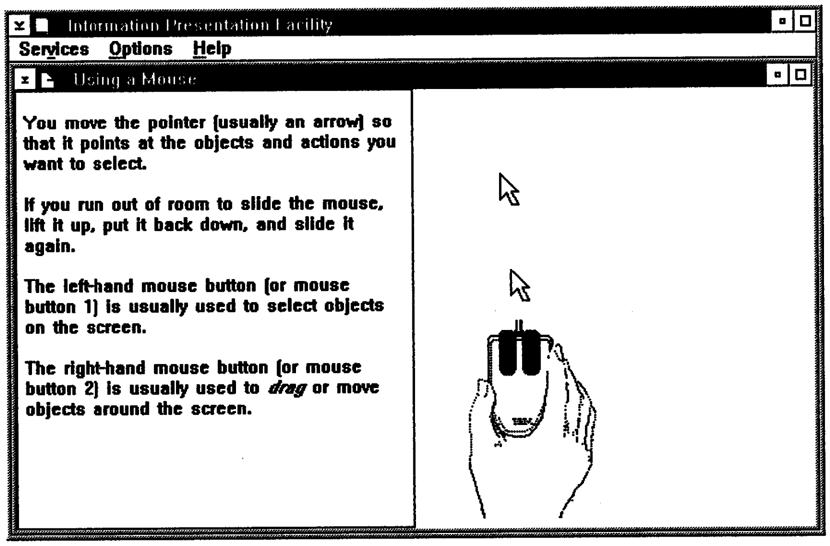
Application-controlled windows are defined in the tagged source file with the application-controlled window tag (**:acviewport.**). With this tag, a window is controlled by a program that has been written and compiled into the form of a dynamic link library (DLL). When an IPF window is displayed at execution time and **:acviewport.** is encountered, IPF passes control to the entry point in the DLL specified by the **objectname=' '** attribute of **:acviewport.**. At this point, the DLL takes control and executes the instructions specified in the source file. When the call returns to IPF, IPF sizes and positions the window on the screen as defined in the heading tag or **:link.** (see Chapter 6, "Customizing Windows").

Figure 9-1 shows the tagging to produce a split window. In this example, the contents of the left window are IPF-controlled. The contents of the right window are defined and controlled by the IPF communication object **IPFMain** which resides in IPF.DLL.

```
:userdoc.
:title.Information Presentation Facility
:docprof ctrlarea=none.
:h1 res=016 scroll=none clear.Using a Mouse
.*
:link reftype=hd res=017 auto split
      vpx=left vpy=top vpcx=50% vpcy=100%
      rules=border scroll=none titlebar=none.
.*
:acviewport dll='ipf'
                       objectname='IPFMain' objectid=1
                       vpx=right vpy=top vpcx=50% vpcy=100%.
.*
:h2 res=017.Using a Mouse
.*
:p.You move the pointer (usually an arrow) so that it
points at the objects and actions you want to select.
:p.If you run out of room to slide the mouse, lift it up, put it
back down, and slide it again.
:p.The left-hand mouse button (or mouse button 1) is
usually used to select objects on the screen.
:p.The right-hand mouse button (or mouse button 2) is
usually used to :hp1.drag:ehp1. or move objects around the screen.
:euserdoc.
```

Figure 9-1. Tagging for an IPF- and an application-controlled window.

Figure 9-2 shows the compiled version of the tagging shown in Figure 9-1.



You move the pointer (usually an arrow) so that it points at the objects and actions you want to select.

If you run out of room to slide the mouse, lift it up, put it back down, and slide it again.

The left-hand mouse button (or mouse button 1) is usually used to select objects on the screen.

The right-hand mouse button (or mouse button 2) is usually used to *drag* or move objects around the screen.

Figure 9-2. The left window is IPF-controlled. The right window displays an animated mouse whose activity is controlled by a routine in a DLL.

In the previous example, IPF processes **:acviewport.** as follows:

1. It loads IPF.DLL and calls the procedure IPFMain.
2. IPFMain creates a window and registers it with IPF as an object communication window.

3. When the call to IPFMain returns to IPF, IPF gives instructions to display the animated mouse.

IPFMain, IPF.DLL, and the bit maps used for the animated mouse are provided in the IPF sample program available with the Toolkit.

Communication Objects

The flexibility of IPF communication objects makes it a powerful tool for customizing IPF windows. However, before using a communication object, you must understand:

- The components of an IPF coverpage window.
- Communication with IPF.

The Coverpage Window

When an application requests that IPF create a help instance, IPF creates a *coverpage* frame window. The coverpage window is the window in which the application's help information is displayed. The coverpage window title is the help window title defined in the HELPINIT structure:

```
}
hini.pszHelpWindowTitle = (PSZ)szWindowTitle;
```

The IPF text window is a child of the coverpage window. When IPF receives a request to display an application-controlled window, it displays the application-controlled window as a child of the coverpage window. Figure 9-3 shows a coverpage window and its child window.

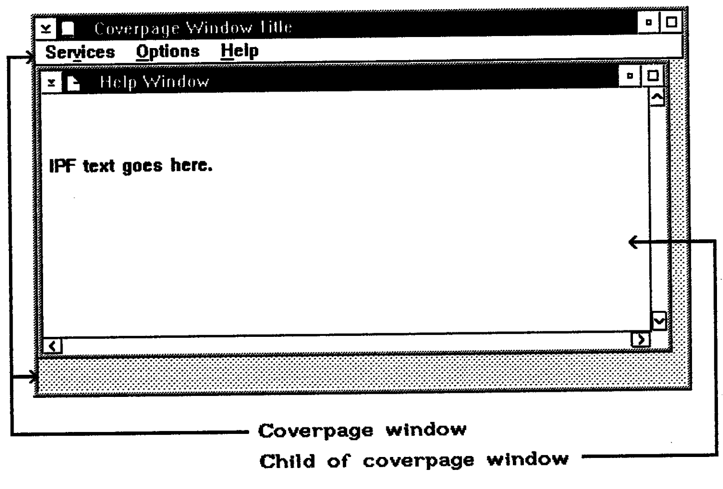


Figure 9-3. A coverpage window and a child window

Changing the Size of the Coverpage Window

IPF communication objects can change the look and function of the coverpage and its child windows. A communication object can change the size of the coverpage and the coverpage menu. This is accomplished by including the **dll=' '** and **objectname=' '** attributes in the **:docprof.** tag at the beginning of the tagged source file for the help information. When IPF processes **:docprof.**, the specified DLL is loaded and the specified communication object is called.

Communication with IPF

Information is passed between IPF and a communication object in two ways:

- Through the parameters to the IPF communication object
- Through messages between the IPF communication object or any windows it creates, and the IPF windows for which the communication object has access.

Communication Object Parameters: When IPF calls an IPF communication object, it passes two parameters to the communication object: a pointer to an ACVP data structure and a pointer to the object information specified by the **objectinfo=''** attribute of **:acviewport.**.

The ACVP data structure is defined in the PMHELP.H header file in the Toolkit and includes the following elements:

```
ULONG  cb;          /* length of data structure                  */
HAB    hAB;         /* anchor block handle                       */
HMQ    hmq;         /* handle to message queue                   */
ULONG  ObjectID;    /* ObjectID attribute as specified in
                    /* acviewport tag                            */
HWND   hWndParent;  /* handle to acviewport parent window        */
HWND   hWndOwner;   /* handle to acviewport owner window         */
HWND   hWndACVP;    /* handle to acviewport                      */
```

IPF supplies all but the last piece of information in this data structure for the communication object. If the communication object creates an application-controlled window, it must place the handle to that window in the last element of the data structure before returning to IPF. IPF uses the handle to size and position the window.

The following example contains an IPF communication object template that shows how an IPF communication object returns the window handle to IPF.

```
#define INCL_WIN
#include <os2.h>

MRESULT EXPENTRY MyObject(PACVP pACVP, PCH ObjectInfo)
       {

HWND hwndMyACVP;             /* handle to the application-controlled */
                             /* window that this procedure creates   */
       .
       .                     /* create the application-controlled    */
       .                     /* window                               */
       .
pACVP->hWndACVP=hwndMyACVP;  /* return the application-controlled    */
                             /* window handle to IPF through the     */
                             /* ACVP data structure                  */
return 0;                    /* return to IPF                        */
       }
```

IPF passes the value of **objectid=''** attribute of **:acviewport.** to the communication object through the ACVP data structure. When an IPF communication object supports multiple **:acviewport.** tags within a document, the **objectid=''** attribute is used to indicate which function the communication object executes when called as a result of a specific **:acviewport.** tag.

Messages between IPF and the Communication Object

IPF and its communication object also communicate through window messages. IPF communication objects, and windows that they create, can send messages to IPF and IPF windows for which the communication object can get a handle. Similarly, IPF can send messages to any window that the communication object creates.

Any message that an application can send to IPF also can be sent by IPF communication objects. The OS/2 2.0 operating system broadens the list of messages to include:

HM_CONTROL
This message is sent to the application or the communication object by IPF prior to the addition of a push button in the control area of a window.

HM_INFORM
This message is sent by IPF and notifies the application that a user has selected a hypertext field that was specified with the **reftype=inform** attribute of the **:link.** tag.

HM_INVALIDATE_DDF_DATA
This message is sent by the application and informs IPF that previous dynamic data formatting (DDF) information is no longer valid.

HM_NOTIFY
This message is sent by IPF and notifies the application or communication object that an event has occurred that the application may be interested in controlling.

HM_QUERY
This message is sent by the application and notifies IPF that the application requires IPF-specific information.

HM_QUERY_DDF_DATA
This message is sent by IPF and notifies the communication object window that IPF has encountered the dynamic data formatting tag (**:ddf.**).

HM_SET_COVERPAGE_SIZE
This message is sent by the application and informs IPF to set the size of the coverpage window.

HM_SET_OBJCOM_WINDOW
This message is sent by the application and informs IPF to identify the communication object to which the HM_INFORM and HM_QUERY_DDF_DATA messages are sent.

HM_SET_USERDATA
This message is sent by the application and informs IPF to store data in the IPF data area.

HM_UPDATE_OBJCOM_WINDOW_CHAIN
This message is sent to the currently active communication object by the communication object who wants to withdraw from the communication chain.

For a complete list, see Chapter 12, "Help Manager Messages."

The Communication Chain

When creating a communication object, always use HM_UPDATE_OBJCOM_WINDOW_CHAIN. This ensures that your application interacts well with other applications.

Communication objects are "inserted" in a daisy chain when they are created; upon termination, IPF sends the HM_UPDATE_WINDOW_CHAIN message to the currently active communication object. It is the responsibility of the communication object that receives the message, to close the gap in the daisy chain. The responsible communication object does this by checking to see if *param1* is equal to the handle that was received from a HM_SET_OBJCOM_WINDOW message issued when the communication chain started. The handle is stored as a variable. If the handle is equal to *param1*, then a communication object is being removed from the chain and the communication chain must replace the handle in the variable with the handle in param2. If the handle is not equal to *param1*, then the responsible communication object must send the message to the handle in the variable so that the communication chain is updated.

In other words, each communication object in the chain knows only one other communication object — the communication object handle returned by HM_SET_OBJCOM_WINDOW. When this one known communication object is removed from the communication chain, the only way to inform the application is to send the message HM_UPDATE_OBJCOM_WINDOW_CHAIN to the active communication object. Be sure to store the return value sent by HM_SET_OBJCOM_WINDOW.

Writing the Communication Object Code

An IPF communication object can be structured in many ways. Its content depends on the function being implemented. Application-controlled windows typically simulate activity that might or might not require user interaction.

An example of a communication object is provided in the IPF sample program (available in the Toolkit), and is shown in Figure 9-2 on page 9-2. The program contains two procedures:

IPFMain (Figure 9-4) registers a window class for the application-controlled window, creates an instance of the class, and registers it with IPF as a communication object.

IPFWinProc (Figure 9-5) provides the animation in the application-controlled window. IPFWinProc is called by IPFMain procedure.

```
#define INCL_WIN
#define INCL_GPI
#define INCL_DOS
#define INCL_DOSMODULEMGR
#define LINT_ARGS
#define DINCL_32

#include <OS2.H>
#include "IPF.H"

#define COM_HWND           4  /* Used in WinSetWindowULong            */
#define FRAMES             5  /* Number of frames in animation sequence */
#define BEEP_WARN_FREQ    60  /* Frequency of warning beep            */
#define BEEP_WARN_DUR    100  /* Duration of warning beep             */

USHORT   IPFClassRegistered = 0;        /* IPF class registered flag     */
HWND     hwndClient;                    /* Handle to the client window   */
HWND     hwndPrevious;                  /* Handle to the previous active */
                                        /* object communication window   */
HWND     hwndLatest;                    /* Handle to the latest active   */
                                        /* object communication window   */

MRESULT EXPENTRY IPFMain (PACVP pACVP, PCH Parameter);
MRESULT EXPENTRY IPFWinProc (HWND hwnd, USHORT msg, MPARAM mp1, MPARAM mp2);
VOID Error (PCH str);

MRESULT EXPENTRY IPFMain (PACVP pACVP, PCH Parameter)
{

    HWND  hwndParent;          /* Handle of parent window in IPF            */
    HWND  hwndFrame;           /* Handle to the frame                       */
    ULONG WinStyle;            /* window style for creating frame           */
    ULONG CtrlData;            /* control data for creating frame           */

    Parameter; /* Warning Level 3 Avoidance */

/** 1) Initialize **/
    if (!IPFClassRegistered)
    {
        if (!WinRegisterClass (pACVP->hAB,
                               "CLASS_IPF",
                               (PFNWP) IPFWinProc,
                               CS_SYNCPAINT | CS_SIZEREDRAW | CS_MOVENOTIFY,
                               8))
        {
            DosBeep (BEEP_WARN_FREQ, BEEP_WARN_DUR);
            exit (TRUE);
        }
        IPFClassRegistered = 1;
    }
    WinStyle = 0L;
    CtrlData = 0L;
```

Figure 9-4 (Part 1 of 2). IPFMain from IPF.C Sample Program

```
    if (!(hwndFrame = WinCreateStdWindow (pACVP->hWndParent,
                                          WinStyle,
                                          &CtrlData,
                                          "CLASS_IPF",
                                          "IPF",
                                          0L,
                                          0L,
                                          0L,
                                          &hwndClient
                                          )))
    {
        Error ("Cannot create window");
        return (MRESULT) TRUE;
    }

/** 2) Process **/

    pACVP->hWndACVP = hwndFrame;

    hwndParent = pACVP->hWndParent;

    hwndPrevious = WinSendMsg (pACVP->hWndParent,
                               HM_SET_OBJCOM_WINDOW,
                               (MPARAM) hwndFrame,
                               NULL);

    hwndLatest = WinSendMsg (pACVP->hWndParent,
                             HM_QUERY,
                             MPFROM2SHORT (NULL, HMQW_OBJCOM_WINDOW),
                             NULL);

    if (hwndFrame != hwndLatest)
    {
        Error ("Cannot set object communication window");
        return (MRESULT) TRUE;
    }

/** 3) Finish **/

    if (!WinSetWindowULong (hwndClient, COM_HWND, (ULONG) hwndPrevious))
    {
        Error ("Cannot save handle into reserved memory");
        return (MRESULT) TRUE;
    }
    return (MRESULT) FALSE;
}
```

Figure 9-4 (Part 2 of 2). IPFMain from IPF.C Sample Program

```
MRESULT EXPENTRY IPFWinProc (HWND hwnd, USHORT msg, MPARAM mp1, MPARAM mp2)
{

    static HAB      Hhab;          /* anchor block handle                  */
    static HBITMAP  hbm [5];       /* array of bitmap handles              */
    static HPS      hps;           /* presentation space                   */
    static POINTL   ptl;           /* pointl                               */
    static HMODULE  hModule;       /* to get bitmaps from DLL resource     */
    static SHORT    index;         /* index to current bitmap to display   */
    static LONG     cxClient,
                    cyClient;      /* window size                          */
           BOOL     rValue=TRUE;   /* FALSE if the message was acted       */
                                   /* upon successfully                    */

/** 1) Initialize **/

    switch (msg)
    {
        case HM_UPDATE_OBJCOM_WINDOW_CHAIN:

            hwndPrevious = (HWND) WinQueryWindowULong (hwnd, COM_HWND);

            if (hwndPrevious == mp2)
            {
                 hwndPrevious = mp1;

               if (!WinSetWindowULong (hwndClient,
                                       COM_HWND,
                                       (ULONG) hwndPrevious))
                {
                    Error ("Cannot save handle into reserved memory");
                    break;
                }
            }
            else
            {
                if (hwndPrevious != NULL)
                {
                    WinSendMsg (hwndPrevious,
                                HM_UPDATE_OBJCOM_WINDOW_CHAIN,
                                (MPARAM) mp1,
                                (MPARAM) mp2);
                }
            }

            rValue = FALSE;
            break;
```

Figure 9-5 (Part 1 of 4). IPFWinProc from IPF.C Sample Program

```
case WM_CREATE:

    if (DosLoadModule (NULL, 0L, "IPF", &hModule))
    {
        Error ("Cannot load module");
        break;
    }

    if (!(hps = WinGetPS(hwnd)))
    {
        Error ("Cannot get presentation space");
        break;
    }

    for (index = 0; index < FRAMES; index++)
    {
        if (!(hbm [index] = GpiLoadBitmap (hps,
                                           hModule,
                                           (USHORT)(IDB_FRAME1+index),
                                           cxClient,
                                           cyClient)))
        {
            Error ("Cannot load bitmap");
            return (MRESULT) rValue;
        }
    }

    WinReleasePS (hps);

    index = 0;

    if (!(Hhab = WinQueryAnchorBlock (hwnd)))
    {
        Error ("Cannot retrieve anchor block handle");
        break;
    }

    if (!WinStartTimer (Hhab, hwnd, ID_TIMER, 150))
    {
        Error ("Cannot start timer");
        break;
    }

    rValue = FALSE;
    break;
```

Figure 9-5 (Part 2 of 4). IPFWinProc from IPF.C Sample Program

```
        case WM_TIMER:

            if (index++ == FRAMES-1)
            {
                index = 0;
            }

            WinInvalidateRect (hwnd, NULL, FALSE);

            rValue = FALSE;
            break;

/** 2) Process **/

        case WM_PAINT:

            if (!(hps = WinBeginPaint (hwnd, NULL, NULL)))
            {
                Error ("Cannot set presentation space for drawing");
                break;
            }

            if (!WinDrawBitmap (hps,
                                hbm [index],
                                NULL,
                                &ptl,
                                CLR_NEUTRAL,
                                CLR_BACKGROUND,
                                DBM_NORMAL))
            {
                Error ("Cannot draw bitmap");
                break;
            }

            WinEndPaint (hps);

            rValue = FALSE;
            break;

        case WM_SIZE:

            cxClient = SHORT1FROMMP (mp2);
            cyClient = SHORT2FROMMP (mp2);

            rValue = FALSE;
            break;
```

Figure 9-5 (Part 3 of 4). IPFWinProc from IPF.C Sample Program

```
/** 3) Finish **/

       case WM_CLOSE:

           WinDestroyWindow (WinQueryWindow (hwnd, QW_PARENT));

           rValue = FALSE;
           break;

       case WM_DESTROY:

           WinStopTimer (Hhab, hwnd, ID_TIMER);

           for (index = 0; index < 8; index++)
           {
               GpiDeleteBitmap (hbm [index]);
           }

           hwndPrevious = (HWND) WinQueryWindowULong (hwnd, COM_HWND);

           hwndLatest = WinSendMsg (hwnd,
                                    HM_QUERY,
                                    MPFROM2SHORT (NULL, HMQW_OBJCOM_WINDOW),
                                    NULL);

           WinSendMsg (hwndLatest,
                       HM_UPDATE_OBJCOM_WINDOW_CHAIN,
                       (MPARAM) hwndPrevious,
                       (MPARAM) WinQueryWindow (hwnd, QW_PARENT));

           DosFreeModule (hModule);

           rValue = FALSE;
           break;

       default:

           rValue = TRUE;
           break;

   }

   return (rValue ? WinDefWindowProc (hwnd, msg, mp1, mp2) : 0L);

}
```

Figure 9-5 (Part 4 of 4). IPFWinProc from IPF.C Sample Program

Using Communication Windows

To position windows or graphics within an application-controlled window, the IPF communication object requires a communication object window. For example, an application-controlled window can be used to represent the Workplace, with an interactive, simulated application window positioned on the Workplace. However, the code implementation of this poses a dilemma. Because IPF sizes and positions the application-controlled window AFTER returning from the call to a communication object, the communication object cannot size and position the simulated application window until after it has created the window and returned control to IPF.

The dilemma is resolved because the communication object can receive HM_INFORM messages after **:acviewport.** has been processed by IPF and the communication object has created an active communication object window. Upon receiving the HM_INFORM message from IPF, the window procedure can then create the simulated application window and position it within the application-controlled window.

The following C-language source code contains the communication object ComWindow that creates a communication window and processes messages from IPF.

```
#define INCL_WIN
#define INCL_DOS

#include <os2.h>

/* Define ID used with reftype = inform attribute in the link tag */
/* in tagged source for help information */

#define SIMULATE_APPWINDOW 1000

MRESULT EXPENTRY ComWindowProc (HWND hwnd, USHORT msg, MPARAM mp1, MPARAM mp2);
MRESULT EXPENTRY SimWindowProc (HWND hwnd, USHORT msg, MPARAM mp1, MPARAM mp2);

HWND hComWindow = NULL;
HWND hSimWindow = NULL;
HWND hComClientWindow;
HWND hSimClientWindow;
HWND PreviousComWindow;
HWND PreviousHwnd;

USHORT EXPENTRY ComWindow (pACVP, ObjectInfo)

PACVP pACVP;
PCH   ObjectInfo;
{

ULONG FrameFlags=0L;

/* Register class for communication window */

   WinRegisterClass (pACVP->hAB,
                     "CLASS_COMM",
                     (PFNWP)ComWindowProc,
                     CS_SYNCPAINT | CS_SIZEREDRAW | CS_MOVENOTIFY, 8);

/* Register class for simulated application window */

   WinRegisterClass (pACVP->hAB,
                     "CLASS_APPSIM",
                     (PFNWP)SimWindowProc,
                     CS_SYNCPAINT | CS_SIZEREDRAW | CS_MOVENOTIFY, 4);

/* Create the communication window */

hComWindow = WinCreateStdWindow (pACVP->hWndParent,
                                 0L,
                                 &FrameFlags,
                                 (PSZ)"CLASS_COMM",
                                 NULL,
                                 0L,
                                 (HMODULE)NULL,
                                 0L,
                                 (PHWND)&hComClientWindow);
```

Figure 9-6 (Part 1 of 4). Procedure for a communication object window.

```
/* Return handle of Communication frame window to IPF */

pACVP->hWndACVP=hComWindow;

/* Send a message to IPF indicating that it should talk to our */
/* communication window */

PreviousComWindow = WinSendMsg (pACVP->hWndParent,
                                HM_SET_OBJCOM_WINDOW,
                                (MPARAM)hComWindow,
                                (MPARAM)hComWindow);

/* Save handle of IPF's communication window in window word of the */
/* communication window */

WinSetWindowULong (hComClientWindow, 0L, (ULONG)PreviousComWindow);

return FALSE;
}
MRESULT EXPENTRY ComWindowProc (HWND hwnd, USHORT msg, MPARAM mp1, MPARAM mp2)
{
    HPS      hps;
    RECTL    Rect;

    ULONG FrameFlags = FCF_TITLEBAR   | FCF_SYSMENU |
                       FCF_SIZEBORDER | FCF_MINMAX  ;
    switch (msg)
    {

        case HM_INFORM:

            switch ((USHORT)mp1)
            {

                case SIMULATE_APPWINDOW:

                    /* create the application window */

                    hSimWindow = WinCreateStdWindow (hwnd,
                                                     WS_VISIBLE,
                                                     &FrameFlags,
                                                     (PSZ)"CLASS_APPSIM",
                                                     NULL,
                                                     0L,
                                                     (HMODULE)NULL,
                                                     0L,
                                                     (PHWND)&hSimClientWindow);
```

Figure 9-6 (Part 2 of 4). Procedure for a communication object window.

```
                    WinSetWindowText (hSimWindow, "Application X");

                    WinSendMsg (hSimWindow,
                                WM_SETICON,
                                WinQuerySysPointer (HWND_DESKTOP, SPTR_APPICON,
                                FALSE),  NULL);

                    /* get the size of the communication client window */

                    WinQueryWindowRect (hwnd, &Rect);

                    /* adjust the size of the application window within the  */
                    /* communication client window                           */

                    Rect.xLeft = Rect.xRight / 12;
                    Rect.yBottom = Rect.yTop / 5;
                    Rect.xRight = Rect.xLeft * 10;
                    Rect.yTop = Rect.yBottom * 3;

                    /* position the application window within the */
                    /* communication client window                */

                    WinSetWindowPos (hSimWindow, HWND_TOP,
                                     (SHORT)Rect.xLeft,
                                     (SHORT)Rect.yBottom,
                                     (SHORT)Rect.xRight,
                                     (SHORT)Rect.yTop,
                                     (SWP_SHOW | SWP_SIZE |
                                      SWP_MOVE | SWP_ACTIVATE));

                    return (MRESULT)TRUE;
            }

        case WM_PAINT:

            hps = WinBeginPaint (hwnd, (HPS)NULL, (PRECTL)NULL);
            WinQueryWindowRect (hwnd, &Rect);
            WinFillRect (hps, &Rect, CLR_RED);
            WinEndPaint (hps);
            break;

        case WM_CLOSE:

            WinDestroyWindow (WinQueryWindow (hwnd, QW_PARENT));
            return (MRESULT)TRUE;
```

Figure 9-6 (Part 3 of 4). Procedure for a communication object window.

```
        case WM_DESTROY:

            PreviousHwnd = (HWND)WinQueryWindowULong (hwnd, 0L);
            WinSendMsg (WinQueryWindow (hwnd, QW_PARENT),
                        HM_SET_OBJCOM_WINDOW,
                        PreviousHwnd,
                        NULL);
            break;
    }

    return (WinDefWindowProc (hwnd, msg, mp1, mp2));
}

/* Create the simulated frame window */

MRESULT EXPENTRY SimWindowProc(HWND hwnd, USHORT msg, MPARAM mp1, MPARAM mp2)
{
    HPS      hps;
    RECTL    Rect;

    switch (msg)
    {

        case WM_PAINT:

            hps = WinBeginPaint (hwnd, (HPS)NULL, (PRECTL)NULL);
            WinQueryWindowRect (hwnd, &Rect);
            WinFillRect (hps, &Rect, CLR_WHITE);
            WinEndPaint (hps);
            break;

        case WM_CLOSE:

            WinDestroyWindow (WinQueryWindow (hwnd, QW_PARENT));
            return (MRESULT)TRUE;

        case WM_DESTROY:

            PreviousHwnd = (HWND)WinQueryWindowULong (hwnd, 0L);
            WinSendMsg (WinQueryWindow (hwnd, QW_PARENT),
                        HM_SET_OBJCOM_WINDOW,
                        PreviousHwnd,
                        NULL);
            break;
    }

    return (WinDefWindowProc (hwnd, msg, mp1, mp2));
}
```

Figure 9-6 (Part 4 of 4). Procedure for a communication object window.

The following shows the tagging that communicates with the communication object through the **reftype=inform** attribute of **:link.**. The contents of the right window are defined by IPF. The contents of the left window are defined and controlled by the communication object **ComWindow** and resides in **INFORM.DLL**.

```
:userdoc.
:docprof ctrlarea=none.
:h1 id=examp5
                  scroll=none
                  x=left y=bottom width=100% height=100%.
Interacting with Application Windows on the Workplace
.*
:link reftype=hd refid=mytxt5
                 vpx=left vpy=bottom vpcx=50%
                 vpcy=100% titlebar=none scroll=none auto split.
.*
:acviewport dll='inform'
                      objectname='ComWindow'
                      objectid=1
                      objectinfo='optional'
                      vpx=right vpy=bottom
                      vpcx=50% vpcy=100%.
.*
.*
:link reftype=inform
                 res=1000 auto.
:h1 hide id=mytxt5.
My text
:p.
This window could contain an explanation of how to interact with
the application-controlled window displayed on the right.
:euserdoc.
```

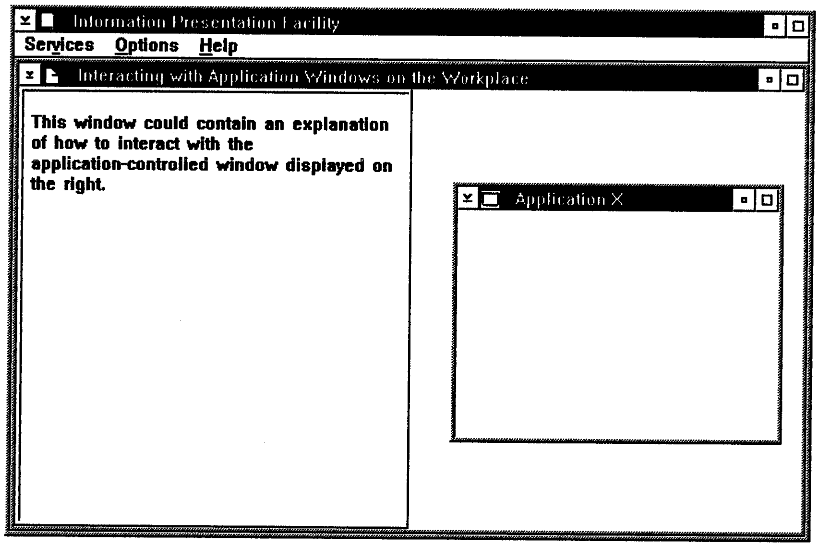
Figure 9-7. Tagging for a communication window

In the previous example, IPF processes **:acviewport.** as follows:

1. It loads INFORM.DLL and calls the procedure ComWindow.
2. ComWindow passes the value of **objectid=** and **objectinfo=**. These attributes are place holders for this example.
3. ComWindow creates a communication window that will receive the HM_INFORM messages from IPF when it processes the **reftype=inform** attribute of **:link.**.

   When the HM_INFORM message is sent to IPF, IPF creates and displays the simulated application window on the Workplace.

Figure 9-8 on page 9-19 displays the windows from the example in Figure 9-7.


Information Presentation Facility
Services Options Help
Interacting with Application Windows on the Workplace
Application X

This window could contain an explanation of how to interact with the application-controlled window displayed on the right.

Figure 9-8. An application-controlled window. The communication object window is a functioning frame window.

Communication windows also are useful when the same communication object is used to support multiple application-controlled windows in help information. For example, you can use the same IPF communication object to represent different simulated application windows from one window to another. Using the previous examples, this is accomplished in two steps.

- Add another **:h1.** window definition to the tagged source for the help information. A different number is specified in the **res=** attribute for the **:link.** tag that has the **reftype=inform** attribute (see Figure 9-7 on page 9-18).
- Add the corresponding **res=** number as another possible value of the parameter to the HM_INFORM message. It is processed accordingly by the communication object window procedure (see, Figure 9-6 on page 9-14).

Dynamic Data Formatting

Dynamic data formatting (DDF) allows you to incorporate text, bit maps, or metafiles in an IPF window at execution time. You can use dynamic data formatting facility in conjunction with the dynamic data format tag (**:ddf.**) The **:ddf.** tag functions as a request by IPF to the application for the DDF data, and a set of DDF application programming interface calls that provide primitives for formatting text. The DDF calls also allow you to incorporate bit maps and metafiles dynamically, and to specify a hypertext or inform link from DDF data to non-DDF data.

IPF has no knowledge of the DDF data it displays, other than that a block of data has been provided to it by the application program. Therefore, DDF data cannot be searched or printed. In effect, DDF is a specific extension of application-controlled windows. When the **:ddf.** tag is encountered at execution time, IPF sends the HM_QUERY_DDF_DATA message to the application window procedure with which the current instance of help is associated. IPF sends the message either by a WinAssociateHelpInstance request or a HM_SET_OBJCOM_WINDOW message.

DDF and Online Help Facilities

DDF data is treated differently for a help and an online document. In the case of a help facility, the HM_QUERY_DDF_DATA message must be processed in the application's window procedure. Within the processing for this message, you can turn on the number specified in the **res=** attribute of the **:ddf.** tag to allow for different processing based on which IPF window with a **:ddf.** tag is currently being displayed.

Therefore, in the case of dynamic data formatting within help, it is not necessary to specify an application-controlled window or a separate DLL. However, this would also work if the application-controlled window used the HM_SET_OBJCOM_WINDOW message to explicitly identify the entry point specified in the **dll=' '** and **objectname=' '** attributes of the **:acviewport.** tag as the proper window procedure where the HM_QUERY_DDF_DATA message is processed.

DDF and Online Documents

The situation is different with an online document. The VIEW program, which displays an online document, is not available for modification. Therefore, to display DDF data in an online document, the **:ddf.** tag must be specified within an application-controlled window. The window that actually specifies the **:ddf.** tag must be defined as a LINK AUTO SPLIT of the application-controlled window's parent window that is specified with a heading tag. The reason is based on the serialization sequence when IPF reads an .INF source file. For example, suppose the file is tagged as follows:

```
:h1 res=100 x=0 y=0 width=50% height=50%.DDF Parent
:acviewport dll='test.dll' objectname='someobject' objectid='1'.
:ddf res=100.
```

The HM_QUERY_DDF_DATA message will be sent to the window procedure of VIEW, which does not process it, and it will be lost. However, suppose the tagging sequence is as follows:

```
:h1 res=100 x=left y=top width=100% height=100% titlebar=both clear.Look here fir
:acviewport dll='flight' objectname='GetName' objectid='2'.
:link reftype=hd refid=ddf1 auto split.
:h1 id=ddf1 x=50% y=top width=50% height=100% hide.ddf1
:ddf res=100.
```

The HM_QUERY_DDF_DATA message will be sent to the "GetName" window procedure, which can initialize and process the DDF data. Therefore, to incorporate DDF data in an online document, you must write a DLL to handle the processing.

For information about the DDF calls, see Chapter 11, "Dynamic Data Formatting Functions."

Chapter 10. Window Functions

Following is a summary of the window function calls that you would use to interface with IPF.

WinAssociateHelpInstance
Associates the help instance with the application window chain.

WinCreateHelpInstance
Calls the IPF help hook so that IPF can handle help requests.

WinCreateHelpTable
Identifies or changes the pointer to the help table in application memory.

WinDestroyHelpInstance
Ends the window chain's association with the help instance.

WinLoadHelpTable
Identifies or changes the handle of the module that contains the help table resource.

WinQueryHelpInstance
Identifies the help instance associated with a particular application window chain.

WinAssociateHelpInstance

Purpose

This function associates a help instance with the application window chain.

Syntax

```
#define INCL_WINHELP   /* Or use INCL_WIN or INCL_PM */
```

```
BOOL  fSuccess = WinAssociateHelpInstance (hwndHelpInstance, hwndApp)
```

Parameters

hwndHelpInstance *(HWND)* - input
Handle returned by the WinCreateHelpInstance function.

hwndApp *(HWND)* - input
Handle of the application window with which to associate the help instance. The help instance is associated with the application window and any of its children or owned windows.

fSuccess *(BOOL)* - return
Success indicator.

Returns

WinAssociateHelpInstance returns the following values:

TRUE	Successful completion.
FALSE	Error has occurred.

When an error occurs, it is returned to the **ulReturnCode** parameter of the HELPINIT structure.

Example Code

This example shows a typical main function for an application which uses help. Following creation of the main application window, IPF is initialized and associated with the window. The help table is defined in the application's resources. When the window is destroyed, terminating the application, the help instance is also destroyed.

```
#define INCL=us.WIN
#include <os2.h>

#define IDHT_APPLICATION       100     /* id of HELP TABLE in resource file */

main( int argc, char *argvī┘, char *envpī┘ )
{
   HAB  hab = WinInitialize( 0 );
   HMQ  hmq = WinCreateMsgQueue( hab, 0 );
   HWND hwnd;
   HWND hwndClient;
   HWND hwndHelp;
   QMSG qmsg;
   ULONG flStyle;
   HELPINIT helpinit;

   /* Setup the help initialization structure */
   helpinit.cb = sizeof( HELPINIT );
   helpinit.ulReturnCode =  0L;
   helpinit.pszTutorialName =  (PSZ)NULL;
   /* Help table in application resource */
   helpinit.phtHelpTable = (PHELPTABLE)MAKEULONG( IDHT_APPLICATION, 0xffff );
   helpinit.hmodHelpTableModule = NULLHANDLE;
   /* Default action bar and accelerators */
   helpinit.hmodAccelActionBarModule = NULLHANDLE;
   helpinit.idAccelTable = 0;
   helpinit.idActionBar = 0;
   helpinit.pszHelpWindowTitle = "APPNAME HELP";
   helpinit.fShowPanelId = CMIC_SHOW_PANEL_ID;
   helpinit.pszHelpLibraryName = "APPNAME.HLP";
   /* Register the class */
   if( WinRegisterClass( ... ) )
   {
      /* create the main window */
      flStyle = FCF_STANDARD;
      hwnd = WinCreateStdWindow( ... );

      if( hwnd )
      {
         /* Create and associate the help instance */
         hwndHelp = WinCreateHelpInstance( hab, &helpinit );

         if( hwndHelp && WinAssociateHelpInstance( hwndHelp, hwnd ) )
         {
            /* Process messages */
            while( WinGetMsg( hab, &qmsg, NULLHANDLE, 0, 0 ) )
            {
               WinDispatchMsg( hab, &qmsg );
            } /* endwhile */
         }

         /* Remove help instance - note: add                      */
         /*     WinAssociateHelpInstance( NULLHANDLE, hwnd );      */
         /* to WM_DESTROY processing to remove the association. */
         WinDestroyHelpInstance( hwndHelp );
      }
   }

   /* finish the cleanup and exit */
   WinDestroyMsgQueue( hmq );
   WinTerminate( hab );
}
```

WinCreateHelpInstance

Purpose

This function calls the IPF help hook so that IPF can handle help requests.

Syntax

```
#define INCL_WINHELP   /* Or use INCL_WIN or INCL_PM */
```

```
HWND  hwndhelp = WinCreateHelpInstance (hab, phinitHMInitStructure)
```

Parameters

hab *(HAB)* **- input**
Handle of the application anchor block returned by the WinInitialize function.

phinitHMInitStructure *(HELPINIT)* **- input/output**
Pointer to the help initialization structure (HelpInit).

hwndhelp *(HWND)* **- return**
Handle to a help instance.

Returns

WinCreateHelpInstance returns the following values:

NULL	Error has occurred.
Other	Handle to help instance has been returned.

When an error occurs, it is returned to the **ulReturnCode** parameter of the HELPINIT structure.

Example Code

This example shows a typical main function for an application which uses help. Following creation of the main application window, IPF is initialized and associated with the window. The help table is defined in the application's resources. When the window is destroyed, terminating the application, the help instance is also destroyed.

```
#define INCL=us.WIN
#include <os2.h>

#define IDHT_APPLICATION        100     /* id of HELP TABLE in resource file */

main( int argc, char *argv[], char *envp[] )
{
   HAB  hab = WinInitialize( 0 );
   HMQ  hmq = WinCreateMsgQueue( hab, 0 );
   HWND hwnd;
   HWND hwndClient;
   HWND hwndHelp;
   QMSG qmsg;
   ULONG flStyle;
   HELPINIT helpinit;

   /* Setup the help initialization structure */
   helpinit.cb = sizeof( HELPINIT );
   helpinit.ulReturnCode =  0L;
   helpinit.pszTutorialName =  (PSZ)NULL;
   /* Help table in application resource */
   helpinit.phtHelpTable = (PHELPTABLE)MAKEULONG( IDHT_APPLICATION, 0xffff );
   helpinit.hmodHelpTableModule = NULLHANDLE;
   /* Default action bar and accelerators */
   helpinit.hmodAccelActionBarModule = NULLHANDLE;
   helpinit.idAccelTable = 0;
   helpinit.idActionBar = 0;
   helpinit.pszHelpWindowTitle = "APPNAME HELP";
   helpinit.fShowPanelId = CMIC_SHOW_PANEL_ID;
   helpinit.pszHelpLibraryName = "APPNAME.HLP";
   /* Register the class */
   if( WinRegisterClass( ... ) )
   {
      /* create the main window */
      flStyle = FCF_STANDARD;
      hwnd = WinCreateStdWindow( ... );

      if( hwnd )
      {
         /* Create and associate the help instance */
         hwndHelp = WinCreateHelpInstance( hab, &helpinit );

         if( hwndHelp && WinAssociateHelpInstance( hwndHelp, hwnd ) )
         {
            /* Process messages */
            while( WinGetMsg( hab, &qmsg, NULLHANDLE, 0, 0 ) )
            {
               WinDispatchMsg( hab, &qmsg );
            } /* endwhile */
         }

         /* Remove help instance - note: add                  */
         /*     WinAssociateHelpInstance( NULLHANDLE, hwnd ); */
         /* to WM_DESTROY processing to remove the association. */
         WinDestroyHelpInstance( hwndHelp );
      }
   }

   /* finish the cleanup and exit */
   WinDestroyMsgQueue( hmq );
   WinTerminate( hab );
}
```

WinCreateHelpTable

Purpose

This function identifies or changes a pointer to a help table in application memory.

Syntax

```
#define INCL_WINHELP   /* Or use INCL_WIN or INCL_PM */
```

```
BOOL  fSuccess = WinCreateHelpTable (hwndHelpInstance, phtHelpTable)
```

Parameters

hwndHelpInstance *(HWND)* **- input**
Handle of a help instance, returned by the WinCreateHelpInstance function.

phtHelpTable *(PHELPTABLE)* **- input**
Pointer to the help table allocated by the application.

fSuccess *(BOOL)* **- return**
Success indicator.

Returns

WinCreateHelpTable returns the following values:

TRUE	Successful completion.
FALSE	Error has occurred.

When an error occurs, it is returned to the **ulReturnCode** parameter of the HELPINIT structure.

Example Code

This example creates a help table in memory and passes the table to the IPF via WinCreateHelpTable. The help instance must have been created by WinCreateHelpInstance.

```
#define INCL_WINHELP
#include <os2.h>

/* defines for window id's, menu items, controls, panels, etc. should */
/* be inserted here or in additional include files.                   */
/* Subtable for the main window's help */
HELPSUBTABLE phtMainTableî ┘ = { 2,
                                  /* length of each entry */
                                  /* fill in one line for each menu item   */
                                  IDM_FILE,       PANELID_FILEMENU,
                                  IDM_FILENEW,    PANELID_FILENEW,
                                  IDM_FILEOPEN,   PANELID_FILEOPEN,
                                  IDM_FILESAVE,   PANELID_FILESAVE,
                                  IDM_FILESAVEAS, PANELID_FILESAVEAS,
                                  IDM_FILEEXIT,   PANELID_FILEEXIT };

/* Subtable for the dialog window's help */
HELPSUBTABLE phtDlgTableî ┘ = { 2,          /* length of each entry */
                                 /* fill in one line for each control */
                                 IDC_EDITFLD,    PANELID_DLGEDITFLD,
                                 IDC_OK,         PANELID_DLGOK,
                                 IDC_CANCEL,     PANELID_DLGCANCEL,
                                 IDC_HELP,       PANELID_HELP };

/* Help table for the applications context sensitive help */
HELPTABLE phtHelpTableî ┘ = { WINDOWID_MAIN, phtMainTable, PANELID_MAINEXT,
                               WINDOWID_DLG,  phtDlgTable,  PANELID_DLGEXT,
                               0,             NULL,            0 };

BOOL CreateHelpTable( HWND hWnd )
{
    BOOL bSuccess = FALSE;
    HWND hwndHelp;

    /* get the associated help instance */
    hwndHelp = WinQueryHelpInstance( hWnd );

    if( hwndHelp )
    {
        /* pass address of help table to the help manager */
        bSuccess = WinCreateHelpTable( hwndHelp, phtHelpTable );
    }

    /* return success indicator */
    return bSuccess;
}
```

WinDestroyHelpInstance

Purpose

This function ends a window chain's association with a help instance.

Syntax

```
#define INCL_WINHELP   /* Or use INCL_WIN or INCL_PM */
```

```
BOOL  fSuccess = WinDestroyHelpInstance (hwndHelpInstance)
```

Parameters

hwndHelpInstance *(HWND)* **- input**
Handle of the help instance to be destroyed. This is the handle returned by the WinCreateHelpInstance call.

fSuccess *(BOOL)* **- return**
Success indicator.

Returns

WinDestroyHelpInstance returns the following values:

TRUE	Successful completion.
FALSE	Error has occurred.

When an error occurs, it is returned to the **ulReturnCode** parameter of the HELPINIT structure.

Example Code

This example shows a typical main function for an application which uses help. Following creation of the main application window, IPF is initialized and associated with the window. The help table is defined in the application's resources. When the window is destroyed, terminating the application, the help instance is also destroyed.

```
#define INCL=us.WIN
#include <os2.h>

#define IDHT_APPLICATION        100     /* id of HELP TABLE in resource file */

main( int argc, char *argvī┘, char *envpī┘ )
{
   HAB  hab = WinInitialize( 0 );
   HMQ  hmq = WinCreateMsgQueue( hab, 0 );
   HWND hwnd;
   HWND hwndClient;
   HWND hwndHelp;
   QMSG qmsg;
   ULONG flStyle;
   HELPINIT helpinit;

   /* Setup the help initialization structure */
   helpinit.cb = sizeof( HELPINIT );
   helpinit.ulReturnCode =  0L;
   helpinit.pszTutorialName =  (PSZ)NULL;
   /* Help table in application resource */
   helpinit.phtHelpTable = (PHELPTABLE)MAKEULONG( IDHT_APPLICATION, 0xffff );
   helpinit.hmodHelpTableModule = NULLHANDLE;
   /* Default action bar and accelerators */
   helpinit.hmodAccelActionBarModule = NULLHANDLE;
   helpinit.idAccelTable = 0;
   helpinit.idActionBar = 0;
   helpinit.pszHelpWindowTitle = "APPNAME HELP";
   helpinit.fShowPanelId = CMIC_SHOW_PANEL_ID;
   helpinit.pszHelpLibraryName = "APPNAME.HLP";
   /* Register the class */
   if( WinRegisterClass( ... ) )
   {
      /* create the main window */
      flStyle = FCF_STANDARD;
      hwnd = WinCreateStdWindow( ... );

      if( hwnd )
      {
         /* Create and associate the help instance */
         hwndHelp = WinCreateHelpInstance( hab, &helpinit );

         if( hwndHelp && WinAssociateHelpInstance( hwndHelp, hwnd ) )
         {
            /* Process messages */
            while( WinGetMsg( hab, &qmsg, NULLHANDLE, 0, 0 ) )
            {
               WinDispatchMsg( hab, &qmsg );
            } /* endwhile */
         }

         /* Remove help instance - note: add                      */
         /*     WinAssociateHelpInstance( NULLHANDLE, hwnd );     */
         /* to WM_DESTROY processing to remove the association. */
         WinDestroyHelpInstance( hwndHelp );
      }
   }

   /* finish the cleanup and exit */
   WinDestroyMsgQueue( hmq );
   WinTerminate( hab );
}
```

WinLoadHelpTable

Purpose

This function identifies or changes the handle of a module containing a help table resource.

Syntax

```
#define INCL_WINHELP   /* Or use INCL_WIN or INCL_PM */
```

```
BOOL  fSuccess = WinLoadHelpTable (hwndHelpInstance, idHelpTable, Module)
```

Parameters

hwndHelpInstance *(HWND)* **- input**
Handle of the help instance. This is the handle returned by the WinCreateHelpInstance call.

idHelpTable *(USHORT)* **- input**
Help table identifier.

Module *(HMODULE)* **- input**
Handle of the module that contains the help table and help subtable resources.

fSuccess *(BOOL)* **- return**
Success indicator.

Returns

WinLoadHelpTable returns the following values:

TRUE	Successful completion.
FALSE	An error has occurred.

When an error occurs, it is returned to the **ulReturnCode** parameter of the HELPINIT structure.

Example Code

The following example loads a help table from RES.DLL using the module handle supplied by DosLoadModule. The module handle is passed to WinCreateHelpInstance and, with an application supplied help table identification (id), the help table is defined to the help manager instance. Notice the assignment of the hmodHelpTableModule field.

```
BOOL LoadHelpTable( HWND hWnd, USHORT usResource, PSZ pszModuleName )
{
    BOOL bSuccess = FALSE;
    HMODULE hmodule;
    HWND hwndHelp;
    PSZ  pszObjNameBuf[ 80 ];

    /* get the DLL loaded */
    if( !DosLoadModule( pszObjNameBuf, sizeof( pszObjNameBuf ),
                   pszModuleName, &hmodule ) )
    {
        /* get the associated help instance */
        hwndHelp = WinQueryHelpInstance( hWnd );

        if( hwndHelp )
        {
            /* pass address of help table to the help manager */
            bSuccess = WinLoadHelpTable( hwndHelp, usResource, hmodule );
        }
    }

    /* return success indicator */
    return bSuccess;
}
                      phinitHMInitStructure.hmodHelpTableModule);
```

WinQueryHelpInstance

Purpose

This function identifies the help instance that is associated with a particular application window chain.

Syntax

```
#define INCL_WINHELP   /* Or use INCL_WIN or INCL_PM */

HWND  hwndHelp = WinQueryHelpInstance (hwndApp)
```

Parameters

hwndApp *(HWND)* **- input**
Handle of the application window.

hwndHelp *(HWND)* **- return**
Help window handle.

Returns

WinQueryHelpInstance returns the following values:

NULL	No help instance is associated with the application window.
Other	A help window handle is returned.

Example Code

This example shows the use of the WinQueryHelpInstance call during the processing of a WM_INITMENU message in order to obtain the handle for sending an HM_SET_ACTIVE_WINDOW message.

```
#define INCL_WIN
#include <os2.h>

MRESULT wm_initmenu( HWND hWnd, ULONG ulMsg, MPARAM mp1, MPARAM mp2 )
{
   /* Send message to establish the current window's parent     */
   /* as the active help window.                                 */
   WinSendMsg( WinQueryHelpInstance( hWnd ),
               HM_SET_ACTIVE_WINDOW,
               (MPARAM)WinQueryWindow( hWnd, QW_PARENT ),
               (MPARAM)WinQueryWindow( hWnd, QW_PARENT ) );

   /* Pass message on for default processing                    */
   return WinDefWindowProc( hWnd, ulMsg, mp1, mp2 );
}
```

Chapter 11. Dynamic Data Formatting Functions

The application can also use the window to establish a dialog with the user, and format text responses in the window by calling dynamic data formatting (DDF) routines. The DDF functions provide limited formatting of text at run time.

Following is a summary of the DDF calls that you can use in your Presentation Manager application.

DdfBeginList
Begins a definition list in the DDF buffer.

DdfBitmap
Places a reference to a bit map in the DDF buffer.

DdfEndList
Terminates the definition list initialized by DdfBeginList.

DdfHyperText
Defines a hypertext link to another window.

DdfInform
Defines a hypertext inform link.

DdfInitialize
Initializes the IPF internal structures for a DDF facility and returns a DDF handle.

DdfListItem
Inserts a definition list entry item in the DDF buffer.

DdfMetafile
Places a reference to a metafile into the DDF buffer.

DdfPara
Creates a paragraph within the DDF buffer.

DdfSetColor
Sets the background and foreground colors of the displayed text.

DdfSetFont
Specifies a text font (Courier) in the DDF buffer.

DdfSetFontStyle
Specifies a text font (bold face) in the DDF buffer.

DdfSetFormat
Turns formatting off or on.

DdfSetTextAlign
Defines whether left, center, or right text justification is to be used when text formatting is off.

DdfText
Adds text to the DDF buffer.

DdfBeginList

Purpose

This function begins a definition list in the DDF buffer, and corresponds to the definition list tag (**:dl.**). Once this function is called, use of any DDF function other than DdfListItem, DdfSetColor, and DdfEndList may produce unpredictable results.

Syntax

```
#define INCL_DDF

APIRET = DdfBeginList (hddf, ulWidthDT, fBreakType,
                       fSpacing);


HDDF   hddf       /* Handle returned by
                         DdfInitialize         */

ULONG  ulWidthDT  /* Width of the definition
                         term                  */

ULONG  fBreakType /* Which line to start the
                         definition on         */

ULONG  fSpacing   /* Single or double line
                         spacing               */
```

Parameters

hddf *(HDDF)* **- input**
Handle to DDF returned by DdfInitialize

ulWidthDT *(ULONG)* **- input**
Width of the definition term.

fBreakType *(ULONG)* **- input**
Only the following constants may be specified:

HMBT_ALL	Start all definition descriptions on the next line, regardless of the actual lengths of definition terms.
HMBT_FIT	Start definition description on the next line only when the definition term is longer than the width specified.
HMBT_NONE	Do not start the definition description on the next line, even when the definition term is longer than the width specified.

fSpacing *(ULONG)* **- input**
Only the following constants may be specified:

HMLS_SINGLELINE	Do not insert a blank line between each definition description and the next definition term.
HMLS_DOUBLELINE	Insert a blank line between each definition description and the next definition term.

Returns

Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

Possible returns from WinGetLastError

1 HMERR_DDF_MEMORY
- Not enough memory is available.

7 HMERR_DDF_LIST_UNCLOSED
- An attempt was made to nest a list.

9 HMERR_DDF_LIST_BREAKTYPE
- The value for BreakType is not valid.

10 HMERR_DDF_LIST_SPACING
- The value for Spacing is not valid.

Example Code

After initializing a DDF buffer with DdfInitialize, the following example shows how to use DdfBeginList to indicate the beginning of a definition list in the DDF buffer. This function corresponds to **:dl.** For more information about initializing DDF, see "DdfInitialize" on page 11-16.

```
#define INCL_WINWINDOWMGR        /* General window management    */
#define INCL_WINMESSAGEMGR       /* Message management           */
#define INCL_DDF                 /* Dynamic Data Facility        */
#include <os2.h>
#include <pmhelp.h>

struct _LISTITEM         /* definition list                      */
{
   PSZ Term;
   PSZ Desc;
} Definition[2] = {{"MVS", "Multiple Virtual System"},
                   {"VM",  "Virtual Machine"}};
MRESULT WindowProc( HWND hwnd, ULONG ulMsg, MPARAM mp1, MPARAM mp2 )
{
   HWND   hwndParent;
   HWND   hwndInstance;
   HDDF   hDdf;             /* DDF handle                        */
   SHORT i;                /* loop index                         */
```

```
    switch( ulMsg )
    {
    case HM_QUERY_DDF_DATA:
        /* get the help instance */
        hwndParent = WinQueryWindow( hwnd, QW_PARENT );
        hwndParent = WinQueryWindow( hwndParent, QW_PARENT );
        hwndInstance = (HWND)WinSendMsg( hwndParent, HM_QUERY,
                                 MPFROMSHORT( HMQW_INSTANCE ), NULL );

        /* Allocate 1K Buffer (default)  */
        hDdf = DdfInitialize(
                   hwndInstance,  /* Handle of help instance */
                   0L,            /* Default buffer size     */
                   0L             /* Default increment       */
                   );

        if (hDdf == NULLHANDLE)   /* Check return code       */
        {
            return (MRESULT)FALSE;
        }

        /* begin definition list */
        if (!DdfBeginList(hDdf, 3L, HMBT_ALL, HMLS_SINGLELINE))
        {
            return (MRESULT)FALSE;
        }

        /* insert 2 entries into definition list */
        for (i=0; i < 2; i++)
        {
            if (!DdfListItem(hDdf, Definition[i].Term, Definition[i].Desc))
            {
               return (MRESULT)FALSE;
            }
        }

        /* terminate definition list */
        if (!DdfEndList(hDdf))
        {
            return (MRESULT)FALSE;
        }

        return (MRESULT)hDdf;
    }
}
```

DdfBitmap

Purpose

This function places a reference to a bit map in the DDF buffer.

The handle to the presentation space in which the bit map was created cannot be freed by the application while the window is displayed.

Syntax

```
#define INCL_DDF

APIRET = DdfBitMap (hddf, hbm, fAlign);


HDDF     hddf    /* Handle returned by
                       DdfInitialize          */

HBITMAP  hbm     /* Standard PM bit map handle    */

ULONG    fAlign  /* Alignment of the bit map      */
```

Parameters

hddf *(HDDF)* **- input**
Handle to DDF returned by DdfInitialize.

hbm *(HBITMAP)* **- input**
Standard Presentation Manager bit map handle.

fAlign *(ULONG)* **- input**
Any of the following values can be specified:

ART_LEFT	Left-justify the bit map.
ART_RIGHT	Right-justify the bit map.
ART_CENTER	Center the bit map.
ART_RUNIN	Allow the bit map to be reflowed with text.

Note: There is a (3-byte + size of HBITMAP structure) ESC code overhead in the DDF internal buffer for this function. There is a 1-byte ESC code overhead required for the Align flag.

Returns

Success indicator.

TRUE	Successful completion.
FALSE	Error occurred.

Possible returns from WinGetLastError

1 **HMERR_DDF_MEMORY**
- Not enough memory is available.

2 **HMERR_DDF_ALIGN_TYPE**
- The alignment type is not valid.

Example Code

After initializing a DDF buffer with DdfInitialize, the following example shows how to obtain a device context (DevOpenDC), create a presentation space (GpiCreatePS), and load a bit map (GpiLoadBitmap). The example then shows how to use DdfBitmap to place a reference to the bit map in the DDF buffer. For more information about initializing DDF, see "DdfInitialize" on page 11-16.

```
#define INCL_WINWINDOWMGR        /* General window management    */
#define INCL_WINMESSAGEMGR       /* Message management           */
#define INCL_GPICONTROL          /* Basic PS control             */
#define INCL_GPIBITMAPS          /* Bit maps and Pel Operations  */
#define INCL_GPIPRIMITIVES       /* Drawing Primitives/Attributes*/
#define INCL_DDF                 /* Dynamic Data Facility        */
#include <os2.h>
#include <pmhelp.h>

#define ACVP_HAB  12
#define BM_HPS    16
#define BM_HDC    20
#define BM_HWND   24
#define ID_LEFT   255

MRESULT WindowProc( HWND hwnd, ULONG ulMsg, MPARAM mp1, MPARAM mp2 )
{
    HWND    hwndParent;     /* parent window                          */
    HWND    hwndInstance;   /* help instance window                   */
    HDDF    hDdf;           /* DDF handle                             */
    HDC     hdc;            /* device context handle                  */
    HPS     hps;            /* presentation space handle              */
    HAB     hab;            /* anchor block handle                    */
    SIZEL   sizel = {0L,0L};/* size of new PS                         */
    HBITMAP hBitmap;        /* bit map handle                         */
    HMODULE hModule;        /* module handle                          */

    switch( ulMsg )
    {
    case HM_QUERY_DDF_DATA:
        hwndParent = WinQueryWindow( hwnd, QW_PARENT );
        hwndParent = WinQueryWindow( hwndParent, QW_PARENT );
        hwndInstance = (HWND)WinSendMsg( hwndParent, HM_QUERY,
                                 MPFROMSHORT( HMQW_INSTANCE ), NULL );

        /* Allocate 1K Buffer (default)  */
        hDdf = DdfInitialize(
                        hwndInstance,  /* Handle of help instance */
                        0L,            /* Default buffer size     */
                        0L             /* Default increment       */
                        );

        if (hDdf == NULLHANDLE)        /* Check return code       */
        {
            return (MRESULT)FALSE;
        }
```

```
/* get module handle for bit map */
DosGetModHandle("bitmap", &hModule);
if (hModule == NULLHANDLE)
{
    return (MRESULT)FALSE;
}
/* get hab for this window */
if ((hab = (HAB)WinQueryWindowULong(hwnd, ACVP_HAB)) == NULLHANDLE )
{
    return (MRESULT)FALSE;
}

/* create a device context */
if ((hdc = DevOpenDC(hab, OD_MEMORY, "*", 0L,
                     (PDEVOPENDATA)NULL, (HDC)NULL)) == NULLHANDLE )
{
   return (MRESULT)FALSE;
}

/* save hdc in reserved word */
WinSetWindowULong(hwnd, BM_HDC, (ULONG)hdc);

/* create a noncached micro presentation space */
/* and associate it with the window */
if ((hps = GpiCreatePS(hab, hdc, &sizel, PU_PELS | GPIF_DEFAULT
                       | GPIT_MICRO | GPIA_ASSOC)) == NULLHANDLE )
{
   return (MRESULT)FALSE;
}

/* save hps in reserved word */
WinSetWindowULong(hwnd, BM_HPS, (ULONG)hps);

/* Load the Bit map to display */
if ((hBitmap = GpiLoadBitmap(hps, hModule, ID_LEFT, 300L,
                             300L)) == NULLHANDLE )
{
    return (MRESULT)FALSE;
}

/* save bit map hwnd in reserved word */
WinSetWindowULong(hwnd, BM_HWND, (ULONG)hBitmap);

/* Display the bit map align left */
if (!DdfBitmap(hDdf, hBitmap, (ULONG)TA_LEFT))
{
    return (MRESULT)FALSE;
}

return (MRESULT)hDdf;
```

```
    case WM_CLOSE:
        /* release PS, DC, and bit map */
        GpiDestroyPS((HPS)WinQueryWindowULong(hwnd, BM_HPS));
        DevCloseDC((HDC)WinQueryWindowULong(hwnd, BM_HDC));
        GpiDeleteBitmap((HBITMAP)WinQueryWindowULong(hwnd, BM_HWND));
        WinDestroyWindow(WinQueryWindow(hwnd, QW_PARENT));
        return (MRESULT)TRUE;
    }
}
```

DdfEndList

Purpose

This function terminates the definition list initialized by DdfBeginList.

Syntax

```
#define INCL_DDF

APIRET = DdfEndList (hddf);


HDDF     hddf     /* Handle returned by
                        DdfInitialize    */
```

Parameters

hddf *(HDDF)* **- input**
Handle to DDF returned by DdfInitialize.

Returns

Success indicator.

TRUE	Successful completion.
FALSE	Error occurred.

Possible returns from WinGetLastError

8 HMERR_DDF_LIST_UNINITIALIZED
- No definition list has been initialized by DdfBeginList.

Example Code

After initializing a DDF buffer with DdfInitialize, the following example shows how to use DdfEndList to end a definition list in the DDF buffer. For more information about initializing DDF, see "DdfInitialize" on page 11-16.

```
#define INCL_WINWINDOWMGR        /* General window management    */
#define INCL_WINMESSAGEMGR       /* Message management           */
#define INCL_DDF                 /* Dynamic Data Facility        */
#include <os2.h>
#include <pmhelp.h>

struct _LISTITEM        /* definition list                       */
{
    PSZ Term;
    PSZ Desc;
} Definition[2] = {{"MVS", "Multiple Virtual System"},
                   {"VM",  "Virtual Machine"}};
MRESULT WindowProc( HWND hwnd, ULONG ulMsg, MPARAM mp1, MPARAM mp2 )
{
    HWND   hwndParent;
    HWND   hwndInstance;
    HDDF   hDdf;            /* DDF handle                          */
    SHORT i;               /* loop index                          */

    switch( ulMsg )
    {
    case HM_QUERY_DDF_DATA:
        /* get the help instance */
        hwndParent = WinQueryWindow( hwnd, QW_PARENT );
        hwndParent = WinQueryWindow( hwndParent, QW_PARENT );
        hwndInstance = (HWND)WinSendMsg( hwndParent, HM_QUERY,
                             MPFROMSHORT( HMQW_INSTANCE ), NULL );

        /* Allocate 1K Buffer (default)  */
        hDdf = DdfInitialize(
                   hwndInstance,  /* Handle of help instance */
                   0L,            /* Default buffer size     */
                   0L             /* Default increment       */
                   );

        if (hDdf == NULLHANDLE)           /* Check return code        */
        {
            return (MRESULT)FALSE;
        }

        /* begin definition list */
        if (!DdfBeginList(hDdf, 3L, HMBT_ALL, HMLS_SINGLELINE))
        {
            return (MRESULT)FALSE;
        }

        /* insert 2 entries into definition list */
        for (i=0; i < 2; i++)
        {
            if (!DdfListItem(hDdf, Definition[i].Term, Definition[i].Desc))
            {
               return (MRESULT)FALSE;
            }
        }

        /* terminate definition list */
        if (!DdfEndList(hDdf))
        {
            return (MRESULT)FALSE;
        }

        return (MRESULT)hDdf;
    }
}
```

DdfHyperText

Purpose

This function defines a hypertext link to another window.

Syntax

```
#define INCL_DDF

APIRET = DdfHyperText (hddf, pszText,
                       pszReference, fReferenceType);


HDDF   hddf           /* Handle returned by
                         DdfInitialize               */

PSZ    pszText        /* Hypertext phrase            */

PSZ    pszReference   /* Pointer to res number or
                         alphanumeric string         */

ULONG  fReferenceType /* Specifies linking using a
                        res number or alphanumeric ID */
```

Parameters

hddf *(HDDF)* **- input**
Handle to DDF returned by DdfInitialize.

pszText *(PSZ)* **- input**
Hypertext phrase.

pszReference *(PSZ)* **- input**
The value of this parameter depends on the value of ReferenceType:

- If ReferenceType is REFERENCE_BY_RES, this parameter must contain a pointer to a numeric string containing the res number; otherwise it will default to a res number of zero. Valid values are 1 - 64000; all other values are reserved.

- If ReferenceType is REFERENCE_BY_ID, this parameter contains a pointer to a string containing the alphanumeric identifier of the destination window.

fReferenceType *(ULONG)* **- input**
This parameter specifies whether you are linking using a resource identifier (res number), or an alphanumeric identifier.

REFERENCE_BY_RES to link using a resource identifier.
REFERENCE_BY_ID to link using an alphanumeric identifier.

Note: There is a 3-byte ESC code overhead in the DDF internal buffer for each word in the text buffer. There is a 1-byte ESC code overhead for each blank and for each newline character. If ReferenceType is REFERENCE_BY_ID, then there is a (3-byte + Reference length) ESC code overhead. For a ReferenceType of REFERENCE_BY_RES, the overhead is 5 bytes. Finally, there is a 3-byte ESC code overhead that is required for ending the hypertext link.

Returns

Success indicator.

TRUE	Successful completion.
FALSE	Error occurred.

Possible returns from WinGetLastError

1 HMERR_DDF_MEMORY
- Not enough memory is available.

6 HMERR_DDF_REFTYPE
- The reference type is not valid.

Example Code

After initializing a DDF buffer with DdfInitialize, the following example shows how to use DdfHyperText to create a hypertext link with another resource. For more information about initializing DDF, see "DdfInitialize" on page 11-16.

```
#define INCL_WINWINDOWMGR        /* General window management    */
#define INCL_WINMESSAGEMGR       /* Message management           */
#define INCL_DDF                 /* Dynamic Data Facility        */
#include <os2.h>
#include <pmhelp.h>

PSZ    Text = "This text is a HYPERTEXT message.\n"; /* hypertext
                                                        string  */
PSZ    ResID = "1";      /* Resource identifier              */

MRESULT WindowProc( HWND hwnd, ULONG ulMsg, MPARAM mp1, MPARAM mp2 )
{
    HWND   hwndParent;
    HWND   hwndInstance;
    HDDF   hDdf;            /* DDF handle                           */
    switch( ulMsg )
    {
    case HM_QUERY_DDF_DATA:
        /* get the help instance */
        hwndParent = WinQueryWindow( hwnd, QW_PARENT );
        hwndParent = WinQueryWindow( hwndParent, QW_PARENT );
        hwndInstance = (HWND)WinSendMsg( hwndParent, HM_QUERY,
                              MPFROMSHORT( HMQW_INSTANCE ), NULL );
        /* Allocate 1K Buffer (default)  */
        hDdf = DdfInitialize(
                    hwndInstance,  /* Handle of help instance */
                    0L,            /* Default buffer size     */
                    0L             /* Default increment       */
                    );
        if (hDdf == NULLHANDLE)    /* Check return code       */
        {
           return (MRESULT)FALSE;
        }
        /* create hypertext link with resource 1 */
        if (!DdfHyperText(hDdf, (PSZ)Text, ResID, REFERENCE_BY_RES))
        {
           return (MRESULT)FALSE;
        }

        return (MRESULT)hDdf;
    }
}
```

DdfInform

Purpose

This function defines a hypertext inform link, and corresponds to the **:link.** tag and its **reftype=inform** attribute.

Syntax

```
#define INCL_DDF

APIRET = DdfInform (hddf, pszText,
                    resInformNumber);


HDDF   hddf             /* Handle returned by
                           DdfInitialize          */

PSZ    pszText          /* Hypertext phrase       */

ULONG  resInformNumber  /* Res number associated
                           with the hypertext field */
```

Parameters

hddf *(HDDF)* **- input**
Handle to DDF returned by DdfInitialize.

pszText *(PSZ)* **- input**
Hypertext phrase.

resInformNumber *(ULONG)* **- input**
Res number associated with this hypertext field. Possible values are 1 to 64000; all other values are reserved.

Returns

Success indicator.

TRUE	Successful completion.
FALSE	Error occurred.

Possible returns from WinGetLastError

1 HMERR_DDF_MEMORY
- Not enough memory is available.

Example Code

After initializing a DDF buffer with DdfInitialize, the following example shows how to use DdfInform to create a hypertext inform link with another resource. This function corresponds to **:link.** and its **reftype=inform** attribute. For more information about initializing DDF, see "DdfInitialize" on page 11-16.

```
#define INCL_WINWINDOWMGR       /* General window management    */
#define INCL_WINMESSAGEMGR      /* Message management           */
#define INCL_DDF                /* Dynamic Data Facility        */
#include <os2.h>
#include <pmhelp.h>

PSZ    Text = "This text is a HYPERTEXT message.\n"; /* hypertext
                                                         string  */
MRESULT WindowProc( HWND hwnd, ULONG ulMsg, MPARAM mp1, MPARAM mp2 )
{
    HWND   hwndParent;
    HWND   hwndInstance;
    HDDF   hDdf;            /* DDF handle                          */

    switch( ulMsg )
    {
        case HM_QUERY_DDF_DATA:
            /* get the help instance */
            hwndParent = WinQueryWindow( hwnd, QW_PARENT );
            hwndParent = WinQueryWindow( hwndParent, QW_PARENT );
            hwndInstance = (HWND)WinSendMsg( hwndParent, HM_QUERY,
                                 MPFROMSHORT( HMQW_INSTANCE ), NULL );

        /* Allocate 1K Buffer (default)  */
        hDdf = DdfInitialize(
                        hwndInstance,  /* Handle of help instance */
                        0L,            /* Default buffer size     */
                        0L             /* Default increment       */
                        );

        if (hDdf == NULLHANDLE)        /* Check return code       */
        {
            return (MRESULT)FALSE;
        }

        /* create hypertext inform link with resource 1 */
        if (!DdfInform(hDdf, (PSZ)Text, 1L))
        {
            return (MRESULT)FALSE;
        }

        return (MRESULT)hDdf;
    }
}
```

DdfInitialize

Purpose

This function initializes the IPF internal structures for dynamic data formatting and returns a DDF handle. The application uses this handle to refer to a particular DDF window.

At initialization, the default for dynamic data display is that text is aligned on the left, and formatting is turned on.

Syntax

```
#define INCL_DDF

HDDF = DdfInitialize (hwndHelpInstance, cbBuffer,
                      ulIncrement);


HWND   hwndHelpInstance      /* Handle to help
                             instance           */

ULONG  cbBuffer          /* Initial DDF buffer
                            length              */

ULONG  ulIncrement       /* Amount by which to
                            increment buffer size
                            when necessary      */
```

Parameters

hwndHelpInstance *(HWND)* **- input**
Handle of a help instance.

cbBuffer *(ULONG)* **- input**
Initial length of internal buffer where DDF information is to be stored. If this field is NULL, a default value of 1K is defined. The maximum value is 60K.

ulIncrement *(ULONG)* **- input**
Amount by which to increment the buffer size, if necessary. If this field is NULL, a default value of 256 bytes is defined. The maximum value is 60K.

Returns

A handle to DDF (HDDF) is returned if initialization was successful. Otherwise, the value returned is:

NULL
- An error has occurred because of insufficient memory or incorrect instance.

Example Code

The following example shows how to initialize and use the DDF facility for displaying an online document. Two functions are defined: SampleObj, creates a window that displays the online information and specifies the second function, SampleWindowProc, as the corresponding window procedure. These functions are compiled into a dynamic link library (DLL) and exported, so that IPF can invoke them when it encounters **:ddf.** and **:acviewport.** during execution.

:acviewport. specifies the name of the DLL and the SampleObj function. When IPF calls SampleObj, it initializes an application-controlled window with SampleWindowProc as the window procedure and returns the window handle. Later, when IPF encounters **:ddf.**, it sends SampleWindowProc an HM_QUERY_DDF_DATA message. At this point, before calling any of the DDF functions, DdfInitialize must first be called to initiate a DDF buffer, after which the other DDF functions are called to display the online information.

```
#define INCL_WINWINDOWMGR         /* General window management    */
#define INCL_WINMESSAGEMGR        /* Message management           */
#define INCL_WINDIALOGS           /* Dialog boxes                 */
#define INCL_DDF                  /* Dynamic Data Facility        */
#define INCL_32
#include <os2.h>
#include <pmhelp.h>

#define COM_HWND 4                  /* window word offsets        */
#define PAGE_HWND 8
#define ACVP_HAB  12

USHORT DdfClass = FALSE;

MRESULT EXPENTRY SampleWindowProc(HWND hWnd, ULONG Message,
                                MPARAM lParaml, MPARAM lParam2);

USHORT APIENTRY SampleObj(PACVP pACVP, PCH Parameter)
{
HWND DdfHwnd;            /* Client window handle                */
HWND DdfCHwnd;           /* Child window handle                 */
HWND PreviousHwnd;       /* Handle for setting comm window active */
```

```
/* register DDF Base class if not registered already */
if (!DdfClass)
{
     if (!WinRegisterClass(
               pACVP->hAB,   /* Anchor block handle          */
               "CLASS_Ddf",  /* Application window class name */
                             /* Address of window procedure   */
               SampleWindowProc,
                             /* Window class style            */
               CS_SYNCPAINT | CS_SIZEREDRAW | CS_MOVENOTIFY,
               20))          /* Extra storage                 */
    {
       return TRUE;
    }
    DdfClass = TRUE;
}

/*  create standard window  */
if (!(DdfHwnd = WinCreateStdWindow(
                pACVP->hWndParent, /* ACVP is parent       */
                0L,                /* No class style       */
                NULL,              /* Frame control flag   */
                "CLASS_Ddf",    /* Window class name    */
                NULL,              /* No title bar         */
                0L,                /* No special style     */
                0L,                /* Resource in .EXE     */
                0,                 /* No window identifier */
                &DdfCHwnd )))      /* Client window handle */
{
    return FALSE;
}

/* store the frame window handle in ACVP data structure */
pACVP->hWndACVP = DdfHwnd;

/* set this window as active communication window */
PreviousHwnd = (HWND)WinSendMsg(pACVP->hWndParent,
                       HM_SET_OBJCOM_WINDOW,
                       MPFROMHWND(DdfHwnd), NULL);

/* save returned communication hwnd in reserved word */
WinSetWindowULong(DdfCHwnd, COM_HWND, (ULONG)PreviousHwnd);

/* save anchor block handle in reserved word */
WinSetWindowULong (DdfCHwnd, ACVP_HAB, (ULONG)pACVP->hAB);
```

```
    return FALSE;
} /* SampleObj */


MRESULT EXPENTRY SampleWindowProc(HWND hWnd, ULONG Message,
                                  MPARAM lParam1, MPARAM lParam2)
{
    HWND   hwndParent;      /* parent window                        */
    HWND   hwndInstance;    /* help instance window                 */
    HDDF   hDdf;            /* DDF handle                           */
    ULONG  DdfID;           /* DDF resource id                      */

    switch (Message)
    {
    case HM_QUERY_DDF_DATA:
        WinSetWindowULong(hWnd, PAGE_HWND, LONGFROMMP(lParam1));
        DdfID = LONGFROMMP(lParam2);
        hwndParent = WinQueryWindow(hWnd, QW_PARENT);
        hwndParent = WinQueryWindow(hwndParent, QW_PARENT);
        hwndInstance = (HWND)WinSendMsg(hwndParent, HM_QUERY,
                                  MPFROMSHORT(HMQW_INSTANCE), NULL);

        /* Allocate 1K Buffer (default)  */
        hDdf = DdfInitialize(
                        hwndInstance,  /* Handle of help instance */
                        0L,            /* Default buffer size     */
                        0L             /* Default increment       */
                        );

        if (hDdf == NULLHANDLE)        /* Check return code       */
        {
            return (MRESULT)FALSE;
        }

        return (MRESULT)hDdf;

    default:
        return (WinDefWindowProc(hWnd, Message, lParam1, lParam2));
    }
} /* SampleWindowProc */
```

DdfListItem

Purpose

This function inserts a definition list entry item in the DDF buffer, and corresponds to a combination of the definition term tag (**:dt.**) and definition define tag (**:dd.**). The handle to the presentation space in which the bit map was created cannot be freed by the application while the window is displayed.

Syntax

```
#define INCL_DDF

APIRET = DdfListItem (hddf, pszTerm,
                      pszDescription);


HDDF  hddf           /* Handle returned by
                          DdfInitialize        */

PSZ   pszTerm        /* Term portion of the
                          definition list entry */

PSZ   pszDescription /* Description portion of
                          the definition list
                          entry.               */
```

Parameters

hddf *(HDDF)* **- input**
Handle to DDF returned by DdfInitialize

pszTerm *(PSZ)* **- input**
Term portion of the definition list entry.

pszDescription *(PSZ)* **- input**
Description portion of the definition list entry.

Note: There is a (3-byte + size of HBITMAP structure) ESC code overhead in the DDF internal buffer for this function. There is a 1-byte ESC code overhead required for the Align flag.

Returns

Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

Possible returns from WinGetLastError

1 **HMERR_DDF_MEMORY**
- Not enough memory is available.

8 **HMERR_DDF_LIST_UNINITIALIZED**
- No definition list has been initialized by DdfBeginList.

Example Code

After initializing a DDF buffer with DdfInitialize, the following example shows how to begin a definition list and use DdfListItem to insert list entries in the DDF buffer. This function corresponds to a combination of **:dt.** and **:dd.**. For more information about initializing DDF, see "DdfInitialize" on page 11-16.

```
#define INCL_WINWINDOWMGR       /* General window management    */
#define INCL_WINMESSAGEMGR      /* Message management           */
#define INCL_DDF                /* Dynamic Data Facility        */
#include <os2.h>
#include <pmhelp.h>

struct _LISTITEM        /* definition list                      */
{
    PSZ Term;
    PSZ Desc;
} Definition[2] = {{"MVS", "Multiple Virtual System"},
                   {"VM",  "Virtual Machine"}};
MRESULT WindowProc( HWND hwnd, ULONG ulMsg, MPARAM mp1, MPARAM mp2 )
{
    HWND   hwndParent;
    HWND   hwndInstance;
    HDDF   hDdf;            /* DDF handle                       */
    SHORT i;               /* loop index                        */

    switch( ulMsg )
    {
    case HM_QUERY_DDF_DATA:
        /* get the help instance */
        hwndParent = WinQueryWindow( hwnd, QW_PARENT );
        hwndParent = WinQueryWindow( hwndParent, QW_PARENT );
        hwndInstance = (HWND)WinSendMsg( hwndParent, HM_QUERY,
                                 MPFROMSHORT( HMQW_INSTANCE ), NULL );

        /* Allocate 1K Buffer (default)  */
        hDdf = DdfInitialize(
                   hwndInstance,  /* Handle of help instance */
                   0L,            /* Default buffer size     */
                   0L             /* Default increment       */
                   );

        if (hDdf == NULLHANDLE)   /* Check return code       */
        {
            return (MRESULT)FALSE;
        }

        /* begin definition list */
        if (!DdfBeginList(hDdf, 3L, HMBT_ALL, HMLS_SINGLELINE))
        {
            return (MRESULT)FALSE;
        }
        /* insert 2 entries into definition list */
        for (i=0; i < 2; i++)
        {
           if (!DdfListItem(hDdf, Definition[i].Term, Definition[i].Desc))
           {
              return (MRESULT)FALSE;
           }
        }

        /* terminate definition list */
        if (!DdfEndList(hDdf))
        {
           return (MRESULT)FALSE;
        }

        return (MRESULT)hDdf;
    }
}
```

DdfMetafile

Purpose

This function places a reference to a metafile into the DDF buffer.

Syntax

```
#define INCL_DDF

APIRET = DdfMetafile (hddf, hmf,
                      prclRect);


HDDF    hddf          /* Handle returned by
                         DdfInitialize         */

HMF     hmf           /* Handle of the metafile
                         to display            */

PRECTL  prclRect      /* Size of the rectangle
                         in which the metafile
                         will be displayed     */
```

Parameters

hddf *(HDDF)* **- input**
Handle to DDF returned by DdfInitialize.

hmf *(HMF)* **- input**
The handle of the metafile to display.

prclRect *(PRECTL)* **- input**
NULL - fit metafile to window

If not NULL, contains the size of the rectangle in which the metafile will be displayed. The aspect ratio of the metafile is adjusted to fit this rectangle.

Note: There is a 3-byte ESC code overhead in the DDF internal buffer for this function. There also is a MetaFilename length overhead. Finally, the Rect variable requires an additional 16 bytes of overhead in the DDF internal buffer.

Returns

Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

Possible returns from WinGetLastError.

1 HMERR_DDF_MEMORY
- Not enough memory is available.

Example Code

After initializing a DDF buffer with DdfInitialize, and loading a metafile with GpiLoadMetaFile, the following example shows how to use DdfMetafile to place a reference to the metafile in the DDF buffer. For more information about initializing DDF, see "DdfInitialize" on page 11-16.

```
#define INCL_WINWINDOWMGR       /* General window management    */
#define INCL_WINMESSAGEMGR      /* Message management           */
#define INCL_DDF                /* Dynamic Data Facility        */
#define INCL_GPIMETAFILES       /* MetaFiles                    */
#include <os2.h>
#include <pmhelp.h>

#define MF_HWND   0
#define ACVP_HAB  4

MRESULT WindowProc( HWND hwnd, ULONG ulMsg, MPARAM mp1, MPARAM mp2 )
{
    HWND   hwndParent;
    HAB    hab;
    HWND   hwndInstance;    /* help instance window                 */
    HDDF   hDdf;            /* DDF handle                           */
    HMF    hwndMetaFile;    /* metafile handle                      */

    switch( ulMsg )
    {
    case HM_QUERY_DDF_DATA:
        /* get the help instance */
        hwndParent = WinQueryWindow( hwnd, QW_PARENT );
        hwndParent = WinQueryWindow( hwndParent, QW_PARENT );
        hwndInstance = (HWND)WinSendMsg( hwndParent, HM_QUERY,
                                 MPFROMSHORT( HMQW_INSTANCE ), NULL );

        /* Allocate 1K Buffer (default)  */
        hDdf = DdfInitialize(
                    hwndInstance,  /* Handle of help instance */
                    0L,            /* Default buffer size     */
                    0L             /* Default increment       */
                    );

        if (hDdf == NULLHANDLE)    /* Check return code       */
        {
            return (MRESULT)FALSE;
        }

        /* get hab for this window */
        if ((hab = (HAB)WinQueryWindowULong(hwnd, ACVP_HAB)) == NULLHANDLE)
        {
           return (MRESULT)FALSE;
        }

        /* Load the Metafile to display */
        if ((hwndMetaFile = GpiLoadMetaFile(hab, "SAMP.MET")) == NULLHANDLE)
        {
            return (MRESULT)FALSE;
        }
        /* Save MetaFile hwnd in reserved word */
        WinSetWindowULong(hwnd, MF_HWND, hwndMetaFile);

        if (!DdfMetafile(hDdf, hwndMetaFile, NULL))
        {
           return (MRESULT)FALSE;
        }

        return (hDdf);

    case WM_CLOSE:
        GpiDeleteMetaFile((HMF)WinQueryWindowULong(hwnd, MF_HWND));
        WinDestroyWindow(WinQueryWindow(hwnd, QW_PARENT));

        return (MRESULT)TRUE;
    }
    return WinDefWindowProc( hwnd, ulMsg, mp1, mp2 );
}
```

DdfPara

Purpose

This function creates a paragraph within the DDF buffer, and corresponds to the paragraph tag (**:p.**). This function places a reference to a bit map in the DDF buffer.

Syntax

```
#define INCL_DDF

APIRET = DdfPara (hddf);


HDDF  hddf  /* Handle returned by DdfInitialize */
```

Parameters

hddf *(HDDF)* **- input**
Handle to DDF returned by DdfInitialize.

Note: There is a 1-byte ESC code overhead in the DDF internal buffer for this function.

Returns

Success indicator.

TRUE	Successful completion.
FALSE	Error occurred.

Possible returns from WinGetLastError

1 HMERR_DDF_MEMORY
- Not enough memory is available.

Example Code

After initializing a DDF buffer with DdfInitialize, the following example shows how to use DdfPara to create a paragraph in the DDF buffer. This function corresponds to **:p.**. For more information about initializing DDF, see "DdfInitialize" on page 11-16.

```
#define INCL_WINWINDOWMGR        /* General window management    */
#define INCL_WINMESSAGEMGR       /* Message management           */
#define INCL_DDF                 /* Dynamic Data Facility        */
#include <os2.h>
#include <pmhelp.h>

MRESULT WindowProc( HWND hwnd, ULONG ulMsg, MPARAM mp1, MPARAM mp2 )
{
   HWND   hwndParent;
   HWND   hwndInstance;    /* help instance window                */
   HDDF   hDdf;            /* DDF handle                          */

    switch( ulMsg )
```

```
     {
     case HM_QUERY_DDF_DATA:
         /* get the help instance */
         hwndParent = WinQueryWindow( hwnd, QW_PARENT );
         hwndParent = WinQueryWindow( hwndParent, QW_PARENT );
         hwndInstance = (HWND)WinSendMsg( hwndParent, HM_QUERY,
                                  MPFROMSHORT( HMQW_INSTANCE ), NULL );

         /* Allocate 1K Buffer (default)  */
         hDdf = DdfInitialize(
                     hwndInstance,  /* Handle of help instance */
                     0L,            /* Default buffer size     */
                     0L             /* Default increment       */
                     );

         if (hDdf == NULLHANDLE)    /* Check return code       */
         {
             return (MRESULT)FALSE;
         }

         /* create paragraph in DDF buffer */
         if( !DdfPara( hDdf ) )
         {
            return (MRESULT)FALSE;
         }

         /* Change to large (100 x 100 dimensions) Courier font */
         if( !DdfSetFont( hDdf, "Courier", 100L, 100L ) )
         {
            return (MRESULT)FALSE;
         }

         /* make the font BOLDFACE */
         if( !DdfSetFontStyle( hDdf, FM_SEL_BOLD ) )
         {
            return (MRESULT)FALSE;
         }

         /* make the text display as BLUE on a PALE GRAY background */
         if( !DdfSetColor( hDdf, CLR_PALEGRAY, CLR_BLUE ) )
         {
            return (MRESULT)FALSE;
         }

         /* Write data into the buffer */
         if (!DdfText(hDdf, "Sample Text"))
         {
            return (MRESULT)FALSE;
         }

         return (MRESULT)hDdf;
     }
     return WinDefWindowProc( hwnd, ulMsg, mp1, mp2 );
}
```

DdfSetColor

Purpose

This function sets the background and foreground colors of displayed text.

Syntax

```
#define INCL_DDF

APIRET = DdfSetColor (hddf, clrfBackColor,
clrfForColor);


HDDF    hddf           /* Handle returned by
                          DdfInitialize      */

COLOR   fBackColor   /* Background color    */

COLOR   fForColor    /* Foreground color    */
```

Parameters

hddf *(HDDF)* **- input**
Handle to DDF returned by DdfInitialize.

fBackColor *(COLOR)* **- input**
Specifies the desired background color.

fForColor *(COLOR)* **- input**
Specifies the desired foreground color.

The following color value constants may be used for the foreground and background colors:

CLR_DEFAULT - used to set IPF default text color
CLR_BLACK
CLR_BLUE
CLR_RED
CLR_PINK
CLR_GREEN
CLR_CYAN
CLR_YELLOW
CLR_BROWN
CLR_DARKGRAY
CLR_DARKBLUE
CLR_DARKRED
CLR_DARKPINK
CLR_DARKGREEN
CLR_DARKCYAN
CLR_PALEGRAY
CLR_UNCHANGED

Note: There is a 4-byte ESC code overhead in the DDF internal buffer for the foreground color, and a 4-byte overhead for the background color, with this function.

Returns

Success indicator.

TRUE	Successful completion.
FALSE	Error occurred.

Possible returns from WinGetLastError

1 **HMERR_DDF_MEMORY**
 - Not enough memory is available.

3 **HMERR_DDF_BACKCOLOR**
 - The background color is not valid.

4 **HMERR_DDF_FORCOLOR**
 - The foreground color is not valid.

Example Code

After initializing a DDF buffer with DdfInitialize, the following example shows how to use DdfSetColor to set the foreground and background color for text in the DDF buffer. For more information about initializing DDF, see "DdfInitialize" on page 11-16.

```
#define INCL_WINWINDOWMGR       /* General window management    */
#define INCL_WINMESSAGEMGR      /* Message management           */
#define INCL_DDF                /* Dynamic Data Facility        */
#include <os2.h>
#include <pmhelp.h>

MRESULT WindowProc( HWND hwnd, ULONG ulMsg, MPARAM mp1, MPARAM mp2 )
{
   HWND   hwndParent;
   HWND   hwndInstance;    /* help instance window                    */
   HDDF   hDdf;            /* DDF handle                              */

    switch( ulMsg )
    {
    case HM_QUERY_DDF_DATA:
        /* get the help instance */
        hwndParent = WinQueryWindow( hwnd, QW_PARENT );
        hwndParent = WinQueryWindow( hwndParent, QW_PARENT );
        hwndInstance = (HWND)WinSendMsg( hwndParent, HM_QUERY,
                                 MPFROMSHORT( HMQW_INSTANCE ), NULL );

        /* Allocate 1K Buffer (default)  */
        hDdf = DdfInitialize(
                    hwndInstance,  /* Handle of help instance */
                    0L,            /* Default buffer size     */
                    0L             /* Default increment       */
                    );

        if (hDdf == NULLHANDLE)    /* Check return code       */
        {
            return (MRESULT)FALSE;
        }
        /* create paragraph in DDF buffer */
        if( !DdfPara( hDdf ) )
        {
           return (MRESULT)FALSE;
        }

        /* Change to large (100 x 100 dimensions) Courier font */
        if( !DdfSetFont( hDdf, "Courier", 100L, 100L ) )
        {
           return (MRESULT)FALSE;
        }

        /* make the font BOLDFACE */
        if( !DdfSetFontStyle( hDdf, FM_SEL_BOLD ) )
        {
           return (MRESULT)FALSE;
        }

        /* make the text display as BLUE on a PALE GRAY background */
        if( !DdfSetColor( hDdf, CLR_PALEGRAY, CLR_BLUE ) )
        {
           return (MRESULT)FALSE;
        }

        /* Write data into the buffer */
        if (!DdfText(hDdf, "Sample Text"))
        {
           return (MRESULT)FALSE;
        }

        return (MRESULT)hDdf;
    }
    return WinDefWindowProc( hwnd, ulMsg, mp1, mp2 );
}
```

DdfSetFont

Purpose

This function specifies a text font in the DDF buffer.

Syntax

```
#define INCL_DDF

APIRET = DdfSetFont (hddf, pszFaceName, ulWidth,
                     ulHeight);


HDDF    hddf         /* Handle returned by
                          DdfInitialize       */

PSZ     pszFaceName  /* Font name             */

ULONG   ulWidth      /* Font width in points  */

ULONG   ulHeight     /* Font height in points */
```

Parameters

hddf *(HDDF)* **- input**
Handle to DDF returned by DdfInitialize.

pszFaceName *(PSZ)* **- input**
This parameter can be specified in two ways:

- An ASCIIZ string specifying the font name.
- "NULL" or "DEFAULT" to specify the default font.

ulWidth *(ULONG)* **- input**
Font width in points. A point is approximately 1/72 of an inch.

ulHeight *(ULONG)* **- input**
Font height in points.

Returns

Success indicator.

TRUE	Successful completion.
FALSE	Error occurred.

Possible returns from WinGetLastError

1 HMERR_DDF_MEMORY
- Not enough memory is available.

Example Code

After initializing a DDF buffer with DdfInitialize, the following example shows how to use DdfSetFont to specify Courier as the text font used in the DDF buffer. For more information about initializing DDF, see "DdfInitialize" on page 11-16.

```
#define INCL_WINWINDOWMGR        /* General window management    */
#define INCL_WINMESSAGEMGR       /* Message management           */
#define INCL_DDF                 /* Dynamic Data Facility        */
#include <os2.h>
#include <pmhelp.h>

MRESULT WindowProc( HWND hwnd, ULONG ulMsg, MPARAM mp1, MPARAM mp2 )
{
   HWND   hwndParent;
   HWND   hwndInstance;    /* help instance window                  */
   HDDF   hDdf;            /* DDF handle                            */

    switch( ulMsg )
    {
    case HM_QUERY_DDF_DATA:
        /* get the help instance */
        hwndParent = WinQueryWindow( hwnd, QW_PARENT );
        hwndParent = WinQueryWindow( hwndParent, QW_PARENT );
        hwndInstance = (HWND)WinSendMsg( hwndParent, HM_QUERY,
                                 MPFROMSHORT( HMQW_INSTANCE ), NULL );

        /* Allocate 1K Buffer (default)  */
        hDdf = DdfInitialize(
                    hwndInstance,  /* Handle of help instance */
                    0L,            /* Default buffer size     */
                    0L             /* Default increment       */
                    );

        if (hDdf == NULLHANDLE)    /* Check return code       */
        {
            return (MRESULT)FALSE;
        }

        /* create paragraph in DDF buffer */
        if( !DdfPara( hDdf ) )
        {
           return (MRESULT)FALSE;
        }
        /* Change to large (100 x 100 dimensions) Courier font */
        if( !DdfSetFont( hDdf, "Courier", 100L, 100L ) )
        {
           return (MRESULT)FALSE;
        }
        /* make the font BOLDFACE */
        if( !DdfSetFontStyle( hDdf, FM_SEL_BOLD ) )
        {
           return (MRESULT)FALSE;
        }
        /* make the text display as BLUE on a PALE GRAY background */
        if( !DdfSetColor( hDdf, CLR_PALEGRAY, CLR_BLUE ) )
        {
           return (MRESULT)FALSE;
        }

        /* Write data into the buffer */
        if (!DdfText(hDdf, "Sample Text"))
        {
           return (MRESULT)FALSE;
        }

        return (MRESULT)hDdf;
    }
    return WinDefWindowProc( hwnd, ulMsg, mp1, mp2 );
}
```

DdfSetFontStyle

This function specifies a text font style in the DDF buffer.

Syntax

```
#define INCL_DDF

APIRET = DdfSetFontStyle (hddf, fFontStyle);


HDDF    hddf        /* Handle returned by
                         DdfInitialize     */

ULONG   fFontStyle  /* Font style          */
```

Parameters

hddf *(HDDF)* **- input**
Handle to DDF returned by DdfInitialize.

fFontStyle *(ULONG)* **- input**
Any of the following values can be specified:

FM_SEL_ITALIC
FM_SEL_BOLD
FM_SEL_UNDERSCORE

These values can be "ORed" together to combine different font styles.

A value of NULL for this parameter sets the font style back to the default font style.

Note: There is a 4-byte ESC code overhead in the DDF internal buffer for FontStyle.

Returns

Success indicator.

TRUE	Successful completion.
FALSE	Error occurred.

Possible returns from WinGetLastError

1 HMERR_DDF_MEMORY
- Not enough memory is available.

5 HMERR_DDF_FONTSTYLE
- The font style is not valid.

Example Code

After initializing a DDF buffer with DdfInitialize, the following example shows how to use DdfSetFontStyle to specify a bold face text font style in the DDF buffer. For more information about initializing DDF, see "DdfInitialize" on page 11-16.

```
#define INCL_WINWINDOWMGR        /* General window management    */
#define INCL_WINMESSAGEMGR       /* Message management           */
#define INCL_DDF                 /* Dynamic Data Facility        */
#include <os2.h>
#include <pmhelp.h>

MRESULT WindowProc( HWND hwnd, ULONG ulMsg, MPARAM mp1, MPARAM mp2 )
{
   HWND   hwndParent;
   HWND   hwndInstance;     /* help instance window                 */
   HDDF   hDdf;             /* DDF handle                           */

    switch( ulMsg )
    {
    case HM_QUERY_DDF_DATA:
        /* get the help instance */
        hwndParent = WinQueryWindow( hwnd, QW_PARENT );
        hwndParent = WinQueryWindow( hwndParent, QW_PARENT );
        hwndInstance = (HWND)WinSendMsg( hwndParent, HM_QUERY,
                                 MPFROMSHORT( HMQW_INSTANCE ), NULL );

        /* Allocate 1K Buffer (default)  */
        hDdf = DdfInitialize(
                    hwndInstance,  /* Handle of help instance */
                    0L,            /* Default buffer size     */
                    0L             /* Default increment       */
                    );

        if (hDdf == NULLHANDLE)    /* Check return code       */
        {
            return (MRESULT)FALSE;
        }

        /* create paragraph in DDF buffer */
        if( !DdfPara( hDdf ) )
        {
           return (MRESULT)FALSE;
        }
        /* Change to large (100 x 100 dimensions) Courier font */
        if( !DdfSetFont( hDdf, "Courier", 100L, 100L ) )
        {
           return (MRESULT)FALSE;
        }
        /* make the font BOLDFACE */
        if( !DdfSetFontStyle( hDdf, FM_SEL_BOLD ) )
        {
           return (MRESULT)FALSE;
        }
        /* make the text display as BLUE on a PALE GRAY background */
        if( !DdfSetColor( hDdf, CLR_PALEGRAY, CLR_BLUE ) )
        {
           return (MRESULT)FALSE;
        }

        /* Write data into the buffer */
        if (!DdfText(hDdf, "Sample Text"))
        {
           return (MRESULT)FALSE;
        }

        return (MRESULT)hDdf;
    }
    return WinDefWindowProc( hwnd, ulMsg, mp1, mp2 );
}
```

DdfSetFormat

Purpose

This function is used to turn formatting off or on. It corresponds to the **:lines.** tag.

Syntax

```
#define INCL_DDF

APIRET = DdfSetFormat (hddf, fFormatType);


HDDF   hddf          /* Handle returned by
                          DdfInitialize      */

ULONG  fFormatType   /* Turns formatting on
                          or off             */
```

Parameters

hddf *(HDDF)* **- input**
Handle to DDF returned by DdfInitialize.

fFormatType *(ULONG)* **- input**
Only the following constants may be used in this parameter:

TRUE	Turn formatting on.
FALSE	Turn formatting off.

Note: If formatting is ON, there is a 3-byte ESC code overhead in the DDF internal buffer for this function. Otherwise, there is a 4-byte ESC code overhead.

Returns

Success indicator.

TRUE	Successful completion.
FALSE	Error occurred.

Possible returns from WinGetLastError

TRUE
- Formatting was on.

FALSE
- Formatting was off.

DdfSetFormat also returns the following value:

1 HMERR_DDF_MEMORY
- Not enough memory is available.

Example Code

After initializing a DDF buffer with DdfInitialize, the following example shows how to use DdfSetFormat to turn off formatting for text in the DDF buffer. This corresponds to the **:lines.** tag. For more information about initializing DDF, see "DdfInitialize" on page 11-16.

```
#define INCL_WINWINDOWMGR       /* General window management    */
#define INCL_WINMESSAGEMGR      /* Message management           */
#define INCL_GPIPRIMITIVES      /* Drawing Primitives/Attributes*/
#define INCL_DDF                /* Dynamic Data Facility        */
#include <os2.h>
#include <pmhelp.h>


MRESULT WindowProc( HWND hwnd, ULONG ulMsg, MPARAM mp1, MPARAM mp2 )
{
   HWND   hwndParent;
   HWND   hwndInstance;    /* help instance window                 */
   HDDF   hDdf;            /* DDF handle                           */

    switch( ulMsg )
    {
    case HM_QUERY_DDF_DATA:
        /* get the help instance */
        hwndParent = WinQueryWindow( hwnd, QW_PARENT );
        hwndParent = WinQueryWindow( hwndParent, QW_PARENT );
        hwndInstance = (HWND)WinSendMsg( hwndParent, HM_QUERY,
                                MPFROMSHORT( HMQW_INSTANCE ), NULL );

        /* Allocate 1K Buffer (default)  */
        hDdf = DdfInitialize(
                    hwndInstance,  /* Handle of help instance */
                    0L,            /* Default buffer size     */
                    0L             /* Default increment       */
                    );

        if (hDdf == NULLHANDLE)    /* Check return code       */
        {
           return (MRESULT)FALSE;
        }

        /* left justify text when formatting is OFF */
        if (!DdfSetTextAlign(hDdf, TA_LEFT))
        {
           return (MRESULT)FALSE;
        }

        /* turn formatting OFF */
        if (!DdfSetFormat(hDdf, FALSE))
        {
           return (MRESULT)FALSE;
        }

        if (!DdfText(hDdf,
                "Format OFF: This text should be Left Aligned!\n"))
        {
           return (MRESULT)FALSE;
        }

        return (MRESULT)hDdf;
     }
     return WinDefWindowProc( hwnd, ulMsg, mp1, mp2 );
}
```

DdfSetTextAlign

Purpose

This function is used to specify left, center, or right text justification.

Syntax

```
#define INCL_DDF

APIRET = DdfSetTextAlign (hddf, fAlign);


HDDF   hddf     /* Handle returned by
                     DdfInitialize              */

ULONG  fAlign   /* Text alignment specification  */
```

Parameters

hddf *(HDDF)* **- input**
Handle to DDF returned by DdfInitialize.

fAlign *(ULONG)* **- input**
Only the following constants may be used:

TA_LEFT	Left-justify text.
TA_RIGHT	Right-justify text.
TA_CENTER	Center text.

Note: It should be called before DdfSetFormat is called to turn off text formatting, and should not be called again until formatting is turned back on. Note that leading and trailing spaces are not stripped from the text as a result of this alignment.

Returns

Success indicator.

TRUE	Successful completion.
FALSE	Error occurred.

Possible returns from WinGetLastError

2 HMERR_DDF_ALIGN_TYPE
- The alignment type is not valid.

Example Code

After initializing a DDF buffer with DdfInitialize, the following example shows how to use DdfSetTextAlign to specify left justified text in the DDF buffer when formatting is OFF. For more information about initializing DDF, see "DdfInitialize" on page 11-16.

```
#define INCL_WINWINDOWMGR        /* General window management    */
#define INCL_WINMESSAGEMGR       /* Message management           */
#define INCL_GPIPRIMITIVES       /* Drawing Primitives/Attributes*/
#define INCL_DDF                 /* Dynamic Data Facility        */
#include <os2.h>
#include <pmhelp.h>


MRESULT WindowProc( HWND hwnd, ULONG ulMsg, MPARAM mp1, MPARAM mp2 )
{
   HWND   hwndParent;
   HWND   hwndInstance;     /* help instance window                    */
   HDDF   hDdf;             /* DDF handle                              */

    switch( ulMsg )
    {
    case HM_QUERY_DDF_DATA:
        /* get the help instance */
        hwndParent = WinQueryWindow( hwnd, QW_PARENT );
        hwndParent = WinQueryWindow( hwndParent, QW_PARENT );
        hwndInstance = (HWND)WinSendMsg( hwndParent, HM_QUERY,
                                MPFROMSHORT( HMQW_INSTANCE ), NULL );

        /* Allocate 1K Buffer (default)  */
        hDdf = DdfInitialize(
                    hwndInstance,  /* Handle of help instance */
                    0L,            /* Default buffer size     */
                    0L             /* Default increment       */
                    );

        if (hDdf == NULLHANDLE)    /* Check return code       */
        {
           return (MRESULT)FALSE;
        }
```

```
         /* left justify text when formatting is OFF */
         if (!DdfSetTextAlign(hDdf, TA_LEFT))
         {
            return (MRESULT)FALSE;
         }
         /* turn formatting OFF */
         if (!DdfSetFormat(hDdf, FALSE))
         {
            return (MRESULT)FALSE;
         }

         if (!DdfText(hDdf,
                 "Format OFF: This text should be Left Aligned!\n"))
         {
            return (MRESULT)FALSE;
         }

         return (MRESULT)hDdf;
      }
      return WinDefWindowProc( hwnd, ulMsg, mp1, mp2 );
}
```

DdfText

Purpose

This function adds text to the DDF buffer.

Syntax

```
#define INCL_DDF

APIRET = DdfText (hddf, pszText);


HDDF   hddf      /* Handle returned by
                      DdfInitialize          */

PSZ    pszText   /* Pointer to the text buffer
                      to be formatted        */
```

Parameters

hddf *(HDDF)* **- input**
Handle to DDF returned by DdfInitialize.

pszText *(PSZ)* **- input**
Pointer to the text buffer to be formatted.

Note: There is a 3-byte ESC code overhead in the DDF internal buffer for each word in the text buffer. There is a 1-byte ESC code overhead for each blank and for each newline character.

Returns

Success indicator.

TRUE	Successful completion.
FALSE	Error occurred.

Example Code

After initializing a DDF buffer with DdfInitialize, the following example shows how to use DdfText to place text in the buffer. For more information about initializing DDF, see "DdfInitialize" on page 11-16.

```
#define INCL_WINWINDOWMGR        /* General window management    */
#define INCL_WINMESSAGEMGR       /* Message management           */
#define INCL_DDF                 /* Dynamic Data Facility        */
#include <os2.h>
#include <pmhelp.h>

MRESULT WindowProc( HWND hwnd, ULONG ulMsg, MPARAM mp1, MPARAM mp2 )
{
   HWND   hwndParent;
   HWND   hwndInstance;    /* help instance window                */
   HDDF   hDdf;            /* DDF handle                          */

    switch( ulMsg )
    {
    case HM_QUERY_DDF_DATA:
        /* get the help instance */
        hwndParent = WinQueryWindow( hwnd, QW_PARENT );
        hwndParent = WinQueryWindow( hwndParent, QW_PARENT );
        hwndInstance = (HWND)WinSendMsg( hwndParent, HM_QUERY,
                                MPFROMSHORT( HMQW_INSTANCE ), NULL );

        /* Allocate 1K Buffer (default)  */
        hDdf = DdfInitialize(
                    hwndInstance,  /* Handle of help instance */
                    0L,            /* Default buffer size     */
                    0L             /* Default increment       */
                    );

        if (hDdf == NULLHANDLE)    /* Check return code       */
        {
            return (MRESULT)FALSE;
        }

        /* create paragraph in DDF buffer */
        if( !DdfPara( hDdf ) )
        {
           return (MRESULT)FALSE;
        }
        /* Change to large (100 x 100 dimensions) Courier font */
        if( !DdfSetFont( hDdf, "Courier", 100L, 100L ) )
        {
           return (MRESULT)FALSE;
        }
        /* make the font BOLDFACE */
        if( !DdfSetFontStyle( hDdf, FM_SEL_BOLD ) )
        {
           return (MRESULT)FALSE;
        }

        /* make the text display as BLUE on a PALE GRAY background */
        if( !DdfSetColor( hDdf, CLR_PALEGRAY, CLR_BLUE ) )
        {
           return (MRESULT)FALSE;
        }

        /* Write data into the buffer */
        if (!DdfText(hDdf, "Sample Text"))
        {
           return (MRESULT)FALSE;
        }

        return (MRESULT)hDdf;
    }
    return WinDefWindowProc( hwnd, ulMsg, mp1, mp2 );
}
```

Chapter 12. Help Manager Messages

The following is a summary of the messages sent by IPF and the application in response to user help requests.

HM_ACTIONBAR_COMMAND
This message is sent by IPF and notifies the application that a user has selected a tailored menu bar item.

HM_CONTROL
This message is sent to the application or the communication object by IPF prior to the addition of a push button in the control area of a window.

HM_CREATE_HELP_TABLE
This message is sent by the application and informs IPF to use the new help table indicated by this address in memory.

HM_DISMISS_WINDOW
This message is sent by the application and informs IPF to remove the active help window.

HM_DISPLAY_HELP
This message is sent by the application and informs IPF to display a specific help window.

HM_ERROR
This message notifies the application of an error caused by user interaction.

HM_EXT_HELP
This message is sent by the application and informs IPF to display the extended help window for the active application window.

HM_EXT_HELP_UNDEFINED
This message is sent by IPF and notifies the application that an extended help window has not been defined.

HM_GENERAL_HELP
This message is sent by the application and informs IPF to display the general help window for the active application window.

HM_GENERAL_HELP_UNDEFINED
This message is sent by IPF and notifies the application that a general help window has not been defined.

HM_HELP_CONTENTS
This message is sent by the application and informs IPF to display the Contents window.

HM_HELP_INDEX
This message is sent by the application and informs IPF to display the help index window.

HM_HELPSUBITEM_NOT_FOUND
This message is sent by IPF and notifies the application that a user has requested help on a field but that IPF cannot find a related entry in the help subtable.

HM_INFORM
This message is sent by IPF and notifies the application that a user has selected a hypertext field that was specified with the **reftype=inform** attribute of the **:link.** tag.

HM_INVALIDATE_DDF_DATA
This message is sent by the application and informs IPF that previous dynamic data formatting (DDF) information is no longer valid.

HM_KEYS_HELP
This message is sent by the application and informs IPF to display the keys help window.

HM_LOAD_HELP_TABLE
This message is sent by the application and provides IPF with the module handle that contains the help table, the help subtable, and the identity of the help table.

HM_NOTIFY
This message is sent by IPF and notifies the application or communication object that an event has occured that the application may be interested in controlling.

HM_QUERY
This message is sent by the application and notifies IPF that the application requires IPF-specific information.

HM_QUERY_DDF_DATA
This message is sent by IPF and notifies the communication object that IPF has encountered the dynamic data formatting tag (**:ddf.**).

HM_QUERY_KEYS_HELP
This message is sent by IPF and notifies the application that a user has requested keys help for a function.

HM_REPLACE_HELP_FOR_HELP
This message is sent by the application and informs IPF to display the application-defined Help for Help window instead of the IPF Help for Help window.

HM_REPLACE_USING_HELP
This message is sent by the application and informs IPF to display the application-defined Using help window instead of the IPF Using help window.

HM_SET_ACTIVE_WINDOW
This message is sent by the application and enables the application to change the active application window with which the IPF help window is associated.

HM_SET_COVERPAGE_SIZE
This message is sent by the application and informs IPF to set the size of the coverpage window (the window within which all other IPF windows are displayed).

HM_SET_HELP_LIBRARY_NAME
This message is sent by the application and informs IPF to replace the list of help libraries specified in the initialization structure with a new list.

HM_SET_HELP_WINDOW_TITLE
This message is sent by the application and informs IPF to change the text of a help window title.

HM_SET_OBJCOM_WINDOW
This message is sent by the application and informs IPF to identify the communication object to which the HM_INFORM and HM_QUERY_DDF_DATA messages are sent.

HM_SET_SHOW_PANEL_ID
This message is sent by the application and informs IPF to display or hide window IDs for each help window.

HM_SET_USERDATA
This message is sent by the application and informs IPF to store data in the IPF data area.

HM_TUTORIAL
This message is sent by IPF and notifies the application when the user selects **Tutorial** choice from the Help menu bar.

HM_UPDATE_OBJCOM_WINDOW_CHAIN
This message is sent to the currently active communication object by the communication object who wants to withdraw from the communication chain.

A detailed description of the parameters and returns for these messages follows.

HM_ACTIONBAR_COMMAND

This message is sent by IPF and notifies the current active application window that a user has selected a customized menu bar item.

Parameters

param1

idCommand *(USHORT)*
Identity of the menu bar item that was selected.

param2 *(ULONG)*
Reserved

0	Reserved value, zero.

Returns

flreply *(ULONG)*
Reserved

0	Reserved value, zero.

HM_CONTROL

This message is sent by IPF to the child of the coverpage window (see "The Coverpage Window" on page 9-3) to add a *control* in the control area of a window. If an application wants to filter any of the controls, it can subclass the child of the coverpage window and intercept this message. If the application does not intercept this message, IPF adds the control to the control area.

Parameters

param1

reserved *(HIUSHORT)*

controlres *(LOUSHORT)*
The res number of the control that was selected. For author-defined push buttons, this is the res identification number that was specified with the push button tag (**:pbutton.**). For the default push buttons, this is the res identification number defined in the PMHELP.H file.

param2 *(BIT32)*
Reserved.

Returns

flreply *(ULONG)*
Reserved

0	Reserved value, zero.

HM_CREATE_HELP_TABLE

This message is sent by the application to give IPF a new help table.

Parameters

param1

HELPTABLE *(PHELPTABLE)*
A pointer to a help table structure.

param2 *(ULONG)*
Reserved.

0	Reserved value, zero.

Returns

reply

ulreturnValue (ULONG)
Return code.

0	The procedure was successfully completed.
Other	See the values of the **ulErrorCode** parameter of the HM_ERROR message.

HM_DISMISS_WINDOW

This message tells IPF to remove the active help window.

If the user requests help from a primary or secondary window, and then interacts with the primary or secondary window without leaving help, the currently displayed help window might not be appropriate for the application window. This message gives the application the ability to remove that help window.

Parameters

param1 *(ULONG)*
Reserved.

0	Reserved value, zero.

param2 *(ULONG)*
Reserved.

0	Reserved value, zero.

Returns

reply

ulreturnValue *(ULONG)*
Return code.

0	The help window was successfully removed.
Other	There was no associated help window. See also the values of the **ulErrorCode** parameter of HM_ERROR message.

HM_DISPLAY_HELP

This message tells IPF to display a specific help window.

Parameters

param1

This parameter depends on the value of the **usTypeFlag** parameter.

For a value of the **usTypeFlag** parameter of HM_RESOURCEID.

HelpPanelId *(PIDENTITY)*
Identity of the help window.

This points to a USHORT data type.

For a value of the **usTypeFlag** parameter of HM_PANELNAME.

HelpPanelName *(PSTRL)*
Name of the help window.

This points to a PSZ data type.

param2

usTypeFlag *(USHORT)*
Flag indicating how to interpret the first parameter.

HM_RESOURCEID	Indicates that *param1* points to the identity of the help window.
HM_PANELNAME	Indicates that *param1* points to the name of the help window.

Returns

reply

ulreturnValue *(ULONG)*
Return code.

0	The window was successfully displayed.
Other	See the values of the **ulErrorCode** parameter of the HM_ERROR message.

HM_ERROR

This message notifies the application of an error caused by a user interaction.

There is no other way to communicate the error to the application since the user initiated communication, not the application. Other errors caused when the application sends a message to IPF are returned as the *flreply* parameter of the message.

IPF does not display any error messages to the user. Instead, IPF sends or returns all error notifications to the application so that it can display its own messages. This procedure ensures a consistent message interface for all user messages.

Parameters

param1

ulErrorCode *(ULONG)*
Error code.

A constant describing the type of error that occurred. The application can also receive some of these error constants in the *flreply* parameter of messages it has sent to the help manager.

The error constants are:

HMERR_LOAD_DLL
The application is unable to load the resource dynamic link library (DLL).

HMERR_NO_FRAME_WND_IN_CHAIN
There is no frame window in the window chain from which to find or set the associated help instance.

HMERR_INVALID_ASSOC_APP_WND
The application window handle specified on the WinAssociateHelpInstance call is not a valid window handle.

HMERR_INVALID_ASSOC_HELP_INST
The help instance handle specified on the WinAssociateHelpInstance call is not a valid window handle.

HMERR_INVALID_DESTROY_HELP_INST
The window handle specified as the help instance to destroy is not of the help instance class.

HMERR_NO_HELP_INST_IN_CHAIN
The parent or owner chain of the application window specified does not have an associated help instance.

HMERR_INVALID_HELP_INSTANCE_HDL
The handle specified to be a help instance does not have the class name of an IPF help instance.

HMERR_INVALID_QUERY_APP_WND
The application window specified on a WinQueryHelpInstance call is not a valid window handle.

HMERR_HELP_INST_CALLED_INVALID
The handle of the help instance specified on a call to IPF does not have the class name of an IPF help instance.

HMERR_HELPTABLE_UNDEFINE
The application did not provide a help table for context-sensitive help.

HMERR_HELP_INSTANCE_UNDEFINE
The help instance handle specified is invalid.

HMERR_HELPITEM_NOT_FOUND
Context-sensitive help was requested but the ID of the main help item specified was not found in the help table.

HMERR_INVALID_HELPSUIBITEM_SIZE
The help subtable item size is less than 2.

HMERR_HELPSUIBITEM_NOT_FOUND
Context-sensitive help was requested but the ID of the help item specified was not found in the help subtable.

HMERR_INDEX_NOT_FOUND
The index is not in the library file.

HMERR_CONTENT_NOT_FOUND
The library file does not have any content.

HMERR_OPEN_LIB_FILE
The library file cannot be opened.

HMERR_READ_LIB_FILE
The library file cannot be read.

HMERR_CLOSE_LIB_FILE
The library file cannot be closed.

HMERR_INVALID_LIB_FILE
Improper library file provided.

HMERR_NO_MEMORY
Unable to allocate the requested amount of memory.

HMERR_ALLOCATE_SEGMENT
Unable to allocate a segment of memory for memory allocation requests from IPF.

HMERR_FREE_MEMORY
Unable to free allocated memory.

HMERR_PANEL_NOT_FOUND
Unable to find the requested help window.

HMERR_DATABASE_NOT_OPEN
Unable to read the unopened database.

param2 *(ULONG)*
Reserved.

0	Reserved value, zero.

Returns

flreply *(ULONG)*
Reserved.

0	Reserved value, zero.

HM_EXT_HELP

When IPF receives this message, it displays the extended help window for the active application window.

Parameters

param1 *(ULONG)*
Reserved.

0	Reserved value, zero.

param2 *(ULONG)*
Reserved.

0	Reserved value, zero.

Returns

reply

ulreturnValue *(ULONG)*
Return code.

0	The extended help window was successfully displayed.
Other	See the values of the **ulErrorCode** parameter of the HM_ERROR message.

HM_EXT_HELP_UNDEFINED

This message is sent to the application by IPF to notify it that an extended help window has not been defined.

When the extended help window is requested, IPF searches the help table for its identity. If the extended help window identity associated with the current active window is zero, IPF sends this message to the application to notify it that an extended help window has not been defined. The application can then:

- Ignore the request for help and not display a help window
- Display its own window
- Use the HM_DISPLAY_HELP message to tell IPF to display a particular window.

Parameters

param1 *(ULONG)*
Reserved.

0	Reserved value, zero.

param2 *(ULONG)*
Reserved.

0	Reserved value, zero

Returns

flreply *(ULONG)*
Reserved.

0	Reserved value, zero.

HM_GENERAL_HELP

When IPF receives this message, it displays the general help window for the active application window.

Parameters

param1 *(ULONG)*
Reserved.

0	Reserved value, zero.

param2 *(ULONG)*
Reserved.

0	Reserved value, zero.

Returns

reply

ulreturnValue *(ULONG)*
Return code.

0	The general help window was successfully displayed.
Other	See the values of the **ulErrorCode** parameter of the HM_ERROR message.

HM_GENERAL_HELP_UNDEFINED

This message is sent to the application by IPF to notify it that a general help window has not been defined.

When the general help window is requested, IPF searches the help table for its identity. If the general help window identity associated with the current active window is zero, IPF sends this message to the application to notify it that a general help window has not been defined. The application can then:

- Ignore the request for help and not display a help window
- Display its own window
- Use the HM_DISPLAY_HELP message to tell IPF to display a particular window.

Parameters

param1 *(ULONG)*
Reserved.

0	Reserved value, zero.

param2 *(ULONG)*
Reserved.

0	Reserved value, zero.

Returns

flreply *(ULONG)*
Reserved.

0	Reserved value, zero.

HM_HELP_CONTENTS

When IPF receives this message, it displays the Contents window.

Parameters

param1 *(ULONG)*
Reserved.

0	Reserved value, zero.

param2 *(ULONG)*
Reserved.

0	Reserved value, zero.

Returns

reply

ulreturnValue *(ULONG)*
Return code.

0	The Contents window was successfully displayed.
Other	See the values of the **ulErrorCode** parameter of the HM_ERROR message.

HM_HELP_INDEX

When IPF receives this message, it displays the help index window.

Parameters

param1 *(ULONG)*
Reserved.

0	Reserved value, zero.

param2 *(ULONG)*
Reserved.

0	Reserved value, zero.

Returns

reply

ulreturnValue *(ULONG)*
Return code.

0	The help index window was successfully displayed.
Other	See the values of the **ulErrorCode** parameter of the HM_ERROR message.

HM_HELPSUBITEM_NOT_FOUND

IPF sends this message to the application when the user requests help on a field and it cannot find a related entry in the help subtable.

If FALSE is returned from this message, IPF displays the extended help window.

The application has the following options:

- Ignore the notification and not display help for that field or window
- Display its own window
- Use the HM_DISPLAY_HELP message to tell IPF to display a particular window.

Parameters

param1

usContext *(USHORT)*
The type of window on which help was requested.

HLPM_WINDOW	An application window.
HLPM_FRAME	A frame window.
HLPM_MENU	A menu window.

param2

sTopic *(USHORT)*
Topic identifier.

For a value of the **usContext** parameter of HLPM_WINDOW or HLPM_FRAME:

window	Identity of the window containing the field on which help was requested.
menu	Identity of the submenu containing the field on which help was requested.

sSubTopic *(USHORT)*
Subtopic identifier.

For a value of the **usContext** parameter of HLPM_WINDOW or HLPM_FRAME:

control	Control identity of the cursored field on which help was requested.
-1	No menu item was selected.
Other	Menu item identity of the currently selected submenu item on which help was requested.

Returns

reply
Informs IPF what should be done next.

fAction *(BOOL)*
Action indicator:

For a value of the **usContext** parameter of HLPM_WINDOW or HLPM_FRAME:

FALSE	Display the extended help window.
TRUE	Do nothing.

For a value of the **usContext** parameter of HLPM_MENU:

FALSE Display the extended help window.

HM_INFORM

This message is used by IPF to notify the application when the user selects a hypertext field that was specified with the **reftype=inform** attribute of the **:link.** tag.

Parameters

param1

idnum *(USHORT)*
Window identity.

The identity that is associated with the hypertext field.

param2 *(ULONG)*
Reserved.

0 Reserved value, zero.

Returns

flreply *(ULONG)*
Reserved.

0 Reserved value, zero.

HM_INVALIDATE_DDF_DATA

The application sends this message to IPF to indicate that the previous dynamic data formatting (DDF) information is no longer valid. When IPF receives this message, it discards the current DDF information and sends a new HM_QUERY_DDF_DATA message to the object communication window.

This message should be sent to the child of the coverpage window handle.

Parameters

param1 *(ULONG)*

rescount
The count of DDFs to be invalidated.

param2 *(PUSHORT)*

resarray
The pointer to an array of unsigned 16-bit (USHORT) integers that are the *res* numbers of DDFs to be invalidated.

Note: If both param1 and param2 are NULL, then all the DDFs in that window will be invalidated.

Returns

reply

ulreturnvalue *(ULONG)*
Return Code.

0	The procedure was successfully completed.
Other	See the values of the *errorcode* parameter of the HM_ERROR message.

HM_KEYS_HELP

This message is sent by the application and informs IPF to display the keys help window.

When IPF receives this message, it sends an HM_QUERY_KEYS_HELP message to the active application window. The active application window is the window that was specified when the last HM_SET_ACTIVE_WINDOW message was sent. If no HM_SET_ACTIVE_WINDOW message was issued, then the active application window is the window specified in the WinAssociateHelpInstance call.

The application must return one of the following:

- The identity of a keys help window in the **HelpPanel** parameter of the HM_QUERY_KEYS_HELP message.
- Zero, if no action is to be taken by IPF for keys help.

Parameters

param1 *(ULONG)*
Reserved.

0	Reserved value, zero.

param2 *(ULONG)*
Reserved.

0	Reserved value, zero.

Returns

reply

ulreturnValue *(ULONG)*
Return code.

0	The keys help window was successfully displayed.
Other	See the values of the **ulErrorCode** parameter of the HM_ERROR message.

HM_LOAD_HELP_TABLE

The application sends this message to give IPF the module handle that contains the help table, the help subtable, and the identity of the help table.

Parameters

param1

idHelpTable *(USHORT)*
Identity of the help table.

fsidentityflag *(USHORT)*
Help table identity indicator.

X'FFFF'	Reserved value.

param2

MODULE *(HMODULE)*
Resource identity.

Handle of the module that contains the help table and help subtable.

Returns

reply

ulreturnValue *(ULONG)*
Return code.

0	The procedure was successfully completed.
Other	See the values of the **ulErrorCode** parameter of the HM_ERROR message.

HM_NOTIFY

This message is sent to the application to notify it of events that the application would be interested in controlling.

This message is used by the application to sub-class and change the behavior or appearance of the help-window.

Parameters

param1

controlres *(HIUSHORT)*
The res number of the control that was selected. For author-defined push buttons, this is the res number that was specified with the push button tag (**:pbutton.**). For the default push buttons, this is the res number defined in the PMHELP.H file.

reserved *(HIUSHORT)*
Reserved for events other than CONTROL_SELECTED and HELP_REQUESTED.

0	Reserved value, zero.

event *(LOUSHORT)*
The type of event which has occurred.

CONTROL_SELECTED	A control was selected.
HELP_REQUESTED	Help was requested.
OPEN_COVERPAGE	The coverpage is displayed.
OPEN_PAGE	The child window of the coverpage is opened.
SWAP_PAGE	The child window of the coverpage is swapped.
OPEN_INDEX	The index window is displayed.
OPEN_TOC	The table of contents window is displayed.
OPEN_HISTORY	The history window is displayed.
OPEN_LIBRARY	The new library is opened.
OPEN_SEARCH_HIT_LIST	The search list is displayed.

param2 *(ULONG)*
Window handle of relevant window.

Returns

reply

fresult *(BOOL)*
Return code.

TRUE	IPF will not format the controls and re-size the window.
FALSE	IPF will process as normal.

HM_QUERY

This message is sent to IPF by the application to request IPF-specific information, such as the current Instance handle, the active communication object window, the active window, or the group number of the current window.

Parameters

param1

usreserved *(USHORT)*
Reserved

0	Reserved value, zero.

usmessageid *(USHORT)*
Specifies the type of window to query. The value can be any of the following constants:

HMQW_INDEX	The handle of the index window.
HMQW_TOC	The handle of the Table of Contents window.
HMQW_SEARCH	The handle of the Search Hitlist window.
HMQW_VIEWEDPAGES	The handle of the Viewed Pages window.
HMQW_LIBRARY	The handle of the Library List window.
HMQW_OBJCOM_WINDOW	The handle of the active communication window.
HMQW_INSTANCE	The handle of the help instance.
HMQW_COVERPAGE	The handle of the IPF multiple document interface parent window. It is where the secondary windows are contained within the parent window.
HMQW_VIEWPORT	The handle of the *viewport* window specified in the low order word of param1 and in param2.
HMQW_GROUP_VIEWPORT	The group number of the window whose handle is specified in param2.
HMQW_RES_VIEWPORT	The resource identification number of the window whose handle is specified in param2.
HMQW_ACTIVEVIEWPORT	The handle of the currently active window.
USERDATA	The previously stored user-data.

usselectionid *(USHORT)*
Specifies whether a res ID, ID number, or group number is being requested. The value can be any of the following constants:

HMQVP_NUMBER	A pointer to a USHORT that holds the res ID of the window.
HMQVP_NAME	A pointer to a null-terminated string that holds the ID of the window.
HMQVP_GROUP	The group number of the window.

param2 *(PVOID)*
Param2 depends on the value of *param1 messageid*:

If *param1 messageid* is HMQW_VIEWPORT, then *param2* is a pointer to the res number, ID, or group ID.

If *param1 messageid* is HMQW_GROUP_VIEWPORT, then *param2* is the handle of the viewport window for which the group number is requested.

If *param1 messageid* is HMQW_RES_VIEWPORT, then *param2* is the handle of the viewport for which the res number is requested.

Returns

reply

ulreturnvalue *(ULONG)*
Return value.

0	The procedure was not successfully completed.
Other	The handle *(HWND)*, group number *(USHORT)*, or res number *(USHORT)* of the window, or the user data *(USHORT)*, depending on the value of *param1 selectionid.*

HM_QUERY_DDF_DATA

This message is sent to the communication object window by IPF when it encounters the dynamic data formatting tag (**:ddf.**). Upon receiving this message, the communication object calls DdfInitialize to indicate the start of dynamic data formatting (DDF). Any combination of other DDF calls are then made to describe this data. When this is complete, the communication object finishes processing this message, indicating the DDF data is complete. After that time, the DDF handle received from DdfInitialize is considered invalid.

Parameters

param1 *(HWND)*

pageclienthwnd
The client handle of the page that contains the object communication window.

param2 *(ULONG)*

resid
The res ID associated with the DDF tag.

Returns

reply

hddfddfhandle *(HDDF)*
Return code.

0	An error has occurred in the application's DDF processing.
Other	The DDF handle to be displayed.

Note: Once this handle has been returned, the HDDF handle can no longer be used by the application.

HM_QUERY_KEYS_HELP

When the user requests the keys help function, IPF sends this message to the application.

The application responds by returning the identity of the requested keys help window. IPF then displays that help window.

Returning zero in the **usHelpPanel** parameter indicates that IPF should do nothing for the keys help function.

Parameters

param1 *(ULONG)*
Reserved.

0	Reserved value, zero.

param2 *(ULONG)*
Reserved.

0	Reserved value, zero

Returns

reply

usHelpPanel *(USHORT)*
The identity of the application-defined keys help window that is to be displayed.

0	Do nothing.
Other	The identity of the keys help window that is to be displayed.

HM_REPLACE_HELP_FOR_HELP

This message tells IPF to display the application-defined Help for Help window instead of the IPF Help for Help window. An application may prefer to provide information that is more specific to itself, rather than the more general help information that is provided in the IPF Help for Help window.

Parameters

param1

idHelpForHelpPanel *(USHORT)*
The identity of the application-defined Help for Help window.

0	Use the IPF Help for Help window.
Other	The identity of the application-defined Help for Help window.

param2 *(ULONG)*
Reserved.

0	Reserved value, zero.

Returns

flreply *(ULONG)*
Reserved.

0	Reserved value, zero.

HM_REPLACE_USING_HELP

This message tells IPF to display the application-defined Using help window instead of the IPF Using help window. An application may prefer to provide information that is more specific to itself, rather than the more general help information that is provided in the IPF Using help window. The guidelines that define the current CUA interface recommend the **Using help** choice be provided in a pull-down menu from the **Help** choice.

Parameters

param1

idUsingHelpPanel *(USHORT)*
The identity of the application-defined Using Help window.

0	Use the IPF Using Help window.
Other	The identity of the application-defined Using Help window.

param2 *(ULONG)*
Reserved.

0	Reserved value, zero.

Returns

flreply *(ULONG)*
Reserved.

0	Reserved value, zero.

HM_SET_ACTIVE_WINDOW

This message enables the application to change both the window with which IPF communicates and the window next to which the help window is to be positioned.

IPF normally communicates with the application window with which the IPF help instance has been associated, and the help window is positioned next to this same application window.

Parameters

param1

hwndActiveWindow *(HWND)*
The handle of the window to be made active.

Its window procedure receives all messages from IPF until the application changes the active window with another HM_SET_ACTIVE_WINDOW message.

param2

hwndRelativeWindow *(HWND)*
The handle of the window next to which the help window is to be positioned.

The handle of the application window next to which IPF will position a new help window.

HWND_PARENT	This IPF-defined constant tells IPF to trace the parent chain of the window that had the focus when the user requested help.
Other	Handle of the window next to which the help window is to be positioned.

If the **hwndactivewindow** parameter is zero, the **relativewindow** parameter is set to zero. That is, if the active window is NULL HANDLE, the relative window is not used.

Returns

reply

ulreturnValue *(ULONG)*
Return code.

0	The procedure was successfully completed.
Other	See the values of the **ulErrorCode** parameter of the HM_ERROR message.

HM_SET_COVERPAGE_SIZE

This message is sent to IPF by the application to set the size of the coverpage, the window within which all other IPF windows are displayed. The default size for the coverpage of a book is the full width of the screen, while the default size for a help file is one-half the width of the screen.

This message takes effect immediately, changing the size of the coverpage. If the coverpage is not currently open, the requested size is saved for the next open.

Parameters

param1 *(PRECTL)*

coverpagerectl
A PRECTL containing the size of the coverpage.

param2 *(ULONG)*
Reserved.

0	Reserved value, zero.

Returns

reply

ulreturnvalue *(ULONG)*
Return code.

0	The procedure was successfully completed.
Other	See the values of the **errorcode** parameter of the HM_ERROR message.

HM_SET_HELP_LIBRARY_NAME

This message identifies a list of help window library names to the IPF help instance.

Any subsequent communication to IPF with this message replaces the current list of names with the newly specified list.

When help is requested, IPF will search each library in the list for the requested help window.

Parameters

param1

HelpLibraryName *(PSTRL)*
Library name.

Pointer to a PSZ data type.

The string contains a list of help window library names that will be searched by IPF for the requested help window.

param2 *(ULONG)*
Reserved.

0	Reserved value, zero.

Returns

reply

ulreturnValue *(ULONG)*
Return code.

0	The newly specified library successfully replaced the current help window library name.
Other	See the values of the **errorcode** parameter of the HM_ERROR message.

HM_SET_HELP_WINDOW_TITLE

This message enable the application to change the text of a help window title.

Parameters

param1

HelpWindowTitle *(PSTRL)*
Help window title.

Pointer to a PSZ data type.

param2 *(ULONG)*
Reserved.

0	Reserved value, zero.

Returns

reply

ulreturnValue *(ULONG)*
Return code.

0	The window title was successfully set.
Other	See the values of the **errorcode** parameter of the HM_ERROR message.

HM_SET_OBJCOM_WINDOW

This message is sent to IPF by the application to identify the communication object window to which the HM_INFORM and HM_QUERY_DDF_DATA messages will be sent. This message is not necessary if the communication object does not expect to receive either of these messages.

HM_INFORM and HM_QUERY_DDF_DATA messages which are not processed must be passed to the previous communication object window which was returned when HM_SET_OBJECT_WINDOW was sent.

Parameters

param1 *(HWND)*

objcomhwnd
The handle of the communication object window to be set.

param2 *(ULONG)*
Reserved.

0 Reserved value, zero.

Returns

reply

hwndprevioushwnd *(HWND)*
The handle of the previous communication object window.

Note: It is important that the return value be stored and not discarded.

HM_SET_SHOW_PANEL_ID

This message tells IPF to display, hide, or toggle the window identity for each help window displayed.

Parameters

param1

fsShowPanelId *(USHORT)*
The show window identity indicator.

CMIC_HIDE_PANEL_ID	Sets the show option off and the window identity is not displayed.
CMIC_SHOW_PANEL_ID	Sets the show option on and the window identity is displayed.
CMIC_TOGGLE_PANEL_ID	Toggles the display of the window identity.

param2 *(ULONG)*
Reserved.

0	Reserved value, zero.

Returns

reply

ulreturnValue *(ULONG)*
Return code.

0	The show window identity indicator was successfully changed.
Other	See the values of the **errorcode** parameter of the HM_ERROR message.

HM_SET_USERDATA

The application sends this message to IPF to store data in the IPF data area.

Parameters

param1 *(ULONG)*
Reserved.

0	Reserved value, zero.

param2 *(VOID)*
4 byte user data area.

Returns

reply

ulreturn-value *(ULONG)*
Return code.

TRUE	The user data was successfully stored.
FALSE	The call failed.

HM_TUTORIAL

IPF sends this message to the application window when the user selects the Tutorial choice from a help window. The application then calls its own tutorial program.

Parameters

param1

TutorialName *(PSTRL)*
Default tutorial name.

This points to a PSZ data type.

This string contains the name of the default tutorial program specified in the IPF initialization structure. A tutorial name specified in the help window definition overrides this default tutorial program.

param2 *(ULONG)*
Reserved.

0 Reserved value, zero.

Returns

flreply *(ULONG)*
Reserved.

0 Reserved value, zero.

HM_UPDATE_OBJCOM_WINDOW_CHAIN

This message is sent to the currently active communication object by the communication object who wants to withdraw from the communication chain.

Parameters

param1 *(HWND)*
The handle of the object to be withdrawn from the communication chain.

param2 *(HWND)*
Window containing the handle of the object to be replaced.

The object that receives this message should check to see if the object handle returned from HM_SET_OBJCOM_WINDOW is equal to the handle in *param1*. If the handle is equal, then the handle in *param1* should be replaced by the handle in *param2*. If the handle is not equal *and* the handle previously received is not NULL HANDLE, then send HM_UPDATE_OBJCOM_WINDOW_CHAIN to that object.

Returns

flreply *(ULONG)*
Reserved.

0	Reserved value, zero.

Part 3: Tag Reference

Information Presentation Facility

Chapter 13. IPF Tag Reference

This section contains an alphabetic listing of the tags used by the IPF compiler to create online documents and Help windows. An IPF tag controls the format of the displayed output.

The syntax description of each tag includes the tag name, the element that the tag describes, the attributes of the tag, and the end tag. A tag begins with a colon (:) and ends with a period (.) Most tags have an end tag associated with them. An end tag has the same name as the tag, preceded by the letter "e." For example, the end tag for the :userdoc. tag is the :euserdoc. tag.

A tag may have one or more attributes associated with it. An attribute provides additional control information for the tag. An attribute can be followed by apostrophes or single quotation marks. This shows that the information needed contains special characters, and requires single quotation mark or apostrophe delimiters, for example,
:font facename='Tms Rmn'.

Notice that the period that ends the tag follows the attributes specified for the tag. If no attributes are specified, then the period immediately follows the tag name. For example, when the :note. tag does not have the **text=''** attribute specified, the period immediately follows the word :note.

Some tags are required to be in a specific order before the file can be compiled by the IPF compiler. The following example shows the minimum tags required to compile a file:

```
:userdoc.
:h1 id=example1.Tag Example 1
:p.This is the first tag example.
:euserdoc.
```

This section also describes control words used by the IPF compiler. Control words start with a period (.). A control word tells the IPF compiler about the statement that it is part of. For example, the imbed (.im) control word tells the IPF Compiler to include the specified file in the source file at this point.

.br (Break)

Purpose
Causes a break in a line of text.

Syntax

Control Word	Element	Attributes	End
.br	Break		

Attributes

None

Description
Use the **.br** control word to stop the display of text on a line, and continue it on the next line. The break control word must be the only statement on the line. If you enter text on the same line as the break control word, the IPF compiler ignores the break control word.

The break control word is especially useful before a line of text that contains a symbol.

Conditions

The **.br** control word must start in column 1, and be the only statement on the line.

Example

```
:p.These words
appear on
the same line.
.br
These words
.br
do not.
```

Output

```
These words
appear on the same line.
These words
do not.
```

For more information, see "Break" on page 3-16.

.* (Comment)

Purpose
Places a comment into a file.

Syntax

Control Word	Element	Attributes	End
.*	Comment		

Attributes
None

Description
The .* control word allows you to place a comment line into your file. The IPF compiler ignores any text on the same line as the comment control word, and does not display this text.

The comment control word must be the first statement on the line of text that you do not want displayed. Each comment control word must begin on a new line.

You can use comment control words to refer to items, to place notes into your file, or to prevent the display of an item.

No space is required between the comment control word and the text that follows it. Comment control words are used independently of IPF tags. They are not used between any IPF tags or with any IPF tag and its accompanying text or attributes.

Conditions
Do not use the comment control word:

- Within the IPF tag, that is, between the colon that starts the tag and the period that ends the tag.
- Between an IPF tag and its accompanying text or attributes.

Always start the comment control word in column 1.

Example

```
.* The comment control word must be the first statement on the line.
.* When the source file is compiled, the text on the
.* comment line is not displayed.
```

Output
When the file is compiled, the comment control word and the information following it on the comment line are not displayed.

.im (Imbed)

Purpose
Specifies that text or artwork files are to be included at process time.

Syntax

Control Word	Element	Attributes	End
.im	Imbed		

Attributes
None

Description
The **.im** control word enables you to include text or artwork files when you are ready to compile your file.

Conditions

- If the file to be included is not in the current directory, you must enter a complete file name.
- Imbedded files must not use the **:userdoc.** or **:euserdoc.** tags.

Always start the **.im** control word in column 1.

Example

```
:userdoc.
.im filename.ext
.im c:\main\filename.ext
:euserdoc.
```

Output
The text and art in the imbedded files are displayed when you access the compiled file.

:acviewport. (Application-Controlled Window)

Purpose

Enables an application to dynamically control what is displayed in an IPF window.

Syntax

Tag	Element	Attributes	End

:acviewport	Have IPF call a function in a dynamic-link module.	dll=' '
		objectname=' '
		objectinfo=' '
		objectid=' '	

	Define the window in which the function runs.	vpx= vpy= vpcx= vpcy=	

Attributes

dll=' '
Specifies a dynamic-link module for IPF to load so that an *object* (a function) in the module can be run in a window (an *application-controlled window*).

objectname=' '
Identifies the entry point of the object in the dynamic-link module. The value specified for this attribute is case sensitive.

objectinfo=' '
Identifies parameters to be passed to the object.

objectid=' '
Specifies an identifier that will associate the window with the object.

vpx=
vpy=
vpcx=
vpcy=

Define the location and size of the window. **vpx=** and **vpy=** are positions along the x (horizontal) and y (vertical) axes. The point where the values intersect represents the origin of the window. **vpcx=** and **vpcy=** represent changes along the x and y axes with respect to the origin.

These attributes can be expressed as absolute values, relative values, or dynamic values:

Absolute value:
A number followed by a letter, which indicates the unit of measure:

c	(Characters): Average character width of the default system font.
x	(Pixels): Dependent on the display adapter in use.
p	(Points): Typesetting measure; equal to approximately 1/72 inch.

Relative value:
A number followed by the percent sign (%), indicating a percentage of the parent-window width or height.

Dynamic value:
A term indicating a window coordinate location that is dependent on the current size and position of the parent window:

left | center | right
For *x* values, flush left with, in the center of, or flush right with the parent window.

top | center| bottom
For *y* values, at the top, center, or bottom of the parent window.

Description

:acviewport is used in either a help file or an online document file to specify that a window will be under the control of a routine that was written and compiled as part of a dynamic-link module. When an IPF window is selected·for display at run time, and **:acviewport** is encountered, IPF passes control to the entry point (**objectname=**) in the dynamic-link module. At this point, the routine in the module takes control. For more information, see Chapter 9, "Expanding the Scope of IPF" on page 9-1.

The definition for **:acviewport** follows a primary heading; for example:

```
:h2 res=2000
    x=left y=top width=100% height=100%
    scroll=none titlebar=both clear group=1.Information Windows
:acviewport dll='My_DLL' objectname='My_Routine' objectid='1'.
    vpx=right vpc=top vpcx=50% vpcy=100%
```

In the example, a window is displayed in the primary window indicated by the heading tag (**:h2**) and its attributes. The contents of the window are controlled by the object, My_Routine in the dynamic-link module, My_DLL.

When the user selects the primary window and **:acviewport** tag is encountered, IPF calls the object in the dynamic-link module and sizes the child window.

:artlink. (Art Link)

Purpose

Identifies link definitions for hypergraphic areas of a bit map or a metafile.

Syntax

Tag	Element	Attributes	End
:artlink.			:eartlink.

Attributes

None

Description

Use **:artlink** in conjunction with the artwork tag (**:artwork**) to indicate links to a bit map or segments of a bit map, or a metafile. The link definitions are specified by link tags (**:link** and follow **:artlink**, as in Example 1.

Example 1

```
:artlink.
:link reftype=hd res=001 x=0 y=0 cx=16 cy=8.
:link reftype=fn refid=afnr x=16 y=8 cx=16 cy=8.
:link reftype=inform res=0345 x=0 y=8 cx=16 cy=8.
:eartlink.
```

(For more information, see ":link. (Link)" on page 13-43.)

All of the above could be in a separate file, which would be identified by the **linkfile** attribute of the artwork tag, as in Example 2.

Example 2

```
:artwork name='mybitmap.bmp' linkfile='mylinks'.
```

In this example, MYBITMAP.BMP is the name of the file containing the bit map, and MYLINKS is the file consisting of the entries shown in Example 1.

If the artwork tag does not specify the attribute **linkfile=**, IPF looks for **:artlink** on the line immediately following **:artwork**, as shown in Example 3.

Example 3

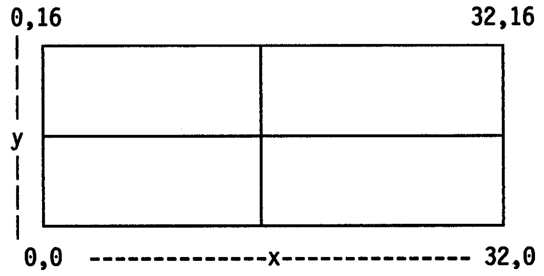
```
:artwork name='mybitmap.bmp'.
:artlink.
:link reftype=hd res=001.
:eartlink.
```

In this example, if the user clicks on the bit map associated with this art link, the window with the identifier, 001 is displayed.

If no **:artlink.** tag is found, no hypergraphic areas for the bit map are defined.

You can divide a bit map into rectangular segments, each of which is selectable and links to different information. For each segment, you need to define values for x, y, cx, and cy, which represent pixel values on the x and y axes. The x axis is always horizontal, and the y axis is always vertical; x and y define the origin of the segment, while cx and cy identify the changes in x and y. The value 0,0 indicates the origin of the bit map and is always the bottom left corner.

Following is an example of a segmented bit map.



Example 4 shows the tagging when the link is from a segmented bit map. The name of the segmented bit-map file is **show2.bmp**; the name of the file with the link information is **link.dat**.

Example 4

```
:artwork name='show2.bmp' linkfile='link.dat'.
```

The following information could be placed into LINK.DAT.

```
:artlink.
:link reftype=hd res=001 x=0 y=0 cx=16 cy=8.
:link reftype=fn refid=afnr x=16 y=8 cx=16 cy=8.
:link reftype=inform res=0345 x=0 y=8 cx=16 cy=8.
:link reftype=launch object='c:\os2\e.exe' data='c:\appsdir\tutor.dat'
      x=16 y=0 cx=16 cy=8.
:eartlink.
```

:artwork. (Artwork)

Purpose
Identifies a bit map to be placed into the user's file.

Tag	Element	Attributes	End
:artwork.	Artwork	name=''	
		align=	
		linkfile=''	
		runin	
		fit	

Attributes

name='filename.ext'
Identifies the file containing the bit map (artwork). This attribute is required and must specify a complete file name.

align=left|right|center
Specifies how the artwork is to align with the current margins. It can be to the left, to the right, or centered.

linkfile='filename.ext'
Identifies the file with the link definitions. This file begins with **:artlink** and ends with **:eartlink**. The **linkfile=** attribute enables you to link from whole or segmented bit maps. It can be omitted if the artwork file does not require links, or if the links are enclosed by **:artlink** and **:eartlink** immediately following the artwork tag.

runin
Specifies that the artwork is to be placed within the line of text. You enter **:artwork** and its attributes in the line of text where you want the artwork to appear.

fit
Causes the artwork to fill the window in which it is displayed. If the user resizes the window, IPF redisplays the bit map so that it fits the new window size.

When the initial size of the window is specified, the ratio between its width and height should be approximately the same as that of the bit map; otherwise, the artwork may appear distorted.

The **fit** attribute is most often used when artwork is to be displayed in a split window, where one window contains a bit map, and another contains text that is displayed beside the bit map.

If the artwork tag has **fit**, and you include text in the same window, the text will be displayed briefly, but will then be covered by the painting of the bit map in the window.

Description
Use **:artwork** to include bit maps, such as vectors and scanned images, in the text file. The artwork tag and its attributes enable you to merge whole or segmented bit maps and position them in the window. A bit map can be created by an application or by a bit-map editing tool, such as the Presentation Manager Icon Editor.

Conditions

- If a path name is not specified for either **name=** or **linkfile=**, IPF looks for the file in the current directory.
- If **linkfile=** is not specified, IPF looks for the artlink tag on the line immediately following the artwork tag.
- The artwork tag requires the **name=** attribute.

Example 1

This example shows how to include artwork that does not require a hypergraphic link. The artwork is to be placed within the line of text that contains the artwork tag.

```
Click on the :artwork name='gopi.art' runin. symbol to close the file.
```

Example 2

This example shows how to include artwork that fills the window in which it is displayed.

```
:artwork name='c:\main\world.bmp' fit.
```

:caution. (Caution)

Purpose
Alerts the user to a risk.

Tag	Element	Attributes	End
:caution.	Caution	text=''	:ecaution.

Attributes

text=''
Enables you to change CAUTION to different text.

Description
A caution message notifies the user of possible risks. It should precede the text to which it pertains so the user will see it first.

When **:caution** is encountered, **CAUTION** appears on the screen, and the caution text is displayed on the next line. A blank line is inserted before the caution message.

Conditions
None

Example 1

```
:caution.
These berries are wild.  Do not eat.
:ecaution.
```

Example 2

```
:caution text='Wild Berries:'.
These berries are wild.  Do not eat.
:ecaution.
```

Output

Example 1

CAUTION:
These berries are wild. Do not eat.

Example 2

Wild Berries:
These berries are wild. Do not eat.

:cgraphic. (Character Graphic)

Purpose
Defines a character graphic.

Syntax

Tag	Element	Attributes	End
:cgraphic.	Character graphic		:ecgraphic.

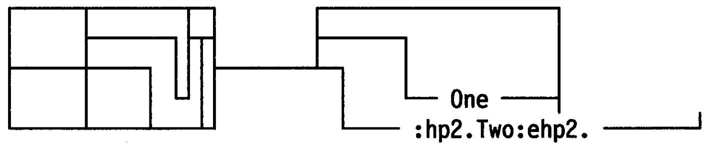
Attributes
None

Description
Character graphics are those you create with an ASCII editor. The **:cgraphic** tag indicates that a character graphic is to follow. Everything after the tag and before **:ecgraphic** will be in a monospace font. A blank line is inserted before and after the graphic.

Conditions
Text that does not fit in the display area of a window is clipped.

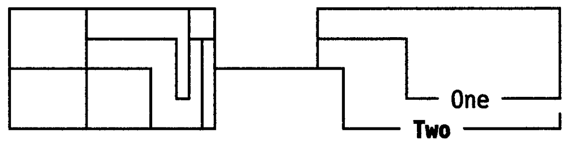
Example

```
:cgraphic.
```



```
:ecgraphic.
```

Output



:color. (Color)

Purpose
Changes the colors of the text and text background.

Syntax

Tag	Element	Attributes	End
:color.	Color	fc=	
		bc=	

Attributes

fc=
Enables you to change the color of the text. Text following this attribute appears in the color specified. The values that can be specified are:
- **default**
- **blue**
- **cyan**
- **green**
- **neutral**
- **red**
- **yellow**

bc=
Enables you to change the background color of the text. The screen colors remain the same. The values that can be specified are the same as those for **fc=**.

Description
:color and its attributes enable you to change the color of the text and the color of the text background. Colors set with this tag remain in effect until another color is specified or a heading definition is encountered.

To return to the system colors, use **fc=default** and **bc=default**.

Conditions
None

Example

```
:sl.
:color fc=green bc=blue.
:li.Color the foreground green; color the background blue.
.*
:color fc=blue bc=red.
:li.Color the foreground blue; color the background red.
.*
:color fc=cyan bc=yellow.
:li.Color the foreground cyan; color the background yellow.
.*
:color fc=default bc=.default.
:li.Return to the system colors.
:esl.
```

Output
The colors of the screen and text change to the specified colors.

:ctrl. (Control Area)

Purpose

Defines the contents of the control area.

Syntax

Tag	Element	Attributes	End
:ctrl	Control area	ctrlid= controls=' ' page coverpage	

Attributes

ctrlid=

Specifies the identification value for the control area. The identification value can be either alpha or alphanumeric, and is referenced by the heading tag.

controls=' '

Specifies the identification values of the push buttons that you want included in the control area of a window. Push buttons are displayed in the order in which they are defined. The values that can be specified are:

Search	Specifies the "Search" push button. This push button displays a window that lets the user search for a word or phrase.
Print	Specifies the "Print" push button. This push button displays a window that lets the user print one or more topics.
Index	Specifies the "Index" push button. This push button displays an alphabetic list to the topics in the document.
Contents	Specifies the "Contents" push button. This push button displays the Contents window.
Esc	Specifies the "Previous" push button. This push button lets the user see information from an earlier request.
Back	Specifies the "Back" push button. This push button displays the previous page in the table of contents hierarchy.
Forward	Specifies the "Forward" push button. This push button displays the next page in the table of contents hierarchy.

Note: An identification value for the Tutorial push button is not provided because it is displayed automatically if a tutorial exists.

If you are defining your own push buttons, use **id=** attribute of the push button tag (**:pbutton.**). See ":pbutton (Push Button)" on page 13-55.

The identification values for the predefined push buttons are defined in the PMHELP.H file (refer to the Developer's Toolkit for OS/2 2.0).

page

Specifies that a set of push buttons display in the control area area of an IPF text window (see page 3-9). You can use this attribute to override the default set of push buttons that display in the control area of an IPF text window.

coverpage

Specifies the set of push buttons that display in the control area of the the coverpage window (see page 3-8). The control area in the coverpage window is at the very bottom of a window. You can use this attribute to override the default set of push buttons that display in the coverpage window.

Example: The following tagging specifies the Previous, Forward, and Back push buttons display in the coverpage window:

```
:ctrl ctrlid=newl controls='ESC FORWARD BACK' coverpage.
```

Description

The control area tag (**:ctrl.**) specifies where push buttons are displayed, and which push buttons you want displayed. You can display push buttons in the *control areas* of coverpage window or an IPF text window.

The default control area for online documents and Help windows is the coverage page window, and the default push buttons that display are:

Online documents

Previous
Search
Print
Index
Contents
Back
Forward
Tutorial (only if a tutorial is available).

Help windows

Previous
Search
Print
Index
Tutorial (only if a tutorial is available).

You can define more than one control area with different sets of pushbutton for an IPF text window. However, only one set of pushbuttons can be defined for the coverpage window.

The default set of push buttons for an IPF text window can be overridden by defining a new default or by referring to the control area definition with the heading tag (see ":h1. through :h6. (Headings)" on page 13-27). For more information about push buttons, see "Push Buttons" on page 3-7.

Conditions

- The control area tag (**:ctrl.**) must be enclosed within the control area definition tag (**:ctrldef.**) and associated end tag (**:ectrldef.**) (see ":ctrldef (Control Area Definition)" on page 13-16).
- The **:ctrl.** tag must follow all push button tags (**:pbutton.**) (see ":pbutton (Push Button)" on page 13-55).

:ctrldef (Control Area Definition)

Purpose

Defines a control area.

Syntax

Tag	Element	Attributes	End
:ctrldef.	Control area definition	NONE	:ectrldef.

Attributes

None

Description

Use the **:ctrldef.** tag to define a control area and the contents of the control area. For tagging information about the control area of a window, see ":ctrl. (Control Area)" on page 13-14.

Conditions

- This tag should follow the **:docprof.** tag.
- The following tags are embedded within the **:ctrldef.** and **:ectrldef.** tags.
  - **:pbutton.**
  - **:ctrl.**

:ddf. (Dynamic Data Formatting)

Purpose

Display dynamically formatted text in an application-controlled window.

Syntax

Tag	Element	Attributes	End
:ddf.	Dynamic data formatting	res=	

Attributes

res=

Associates a location in a window with a request for specific information. The value is an integer from 1 to 64000.

Description

:ddf indicates that the application will provide dynamically formatted data.

When IPF encounters **:ddf**, it sends the message HM_QUERY_DDF_DATA to the OBJCOM window, and specifies the **res=** value. (The application identified the OBJCOM window by sending HM_SET_OBJCOM_WINDOW to IPF.) The OBJCOM code responds by initializing for dynamic data formatting and proceeding with a dynamic data-formatting routine, using dynamic data-formatting functions. For more information, see "Dynamic Data Formatting" on page 9-19.

:dl. (Definition List)

Purpose

Identifies a list of terms and definitions.

Syntax

Tag	Element	Attributes	End
:dl.	Definition list	compact tsize= break=	:edl.
:dthd.	Definition-term heading		
:ddhd.	Definition- description heading		
:dt.	Definition term		
:dd.	Definition description		

Attributes

compact
Causes the list to be formatted without a blank line between each term and description. If you omit this attribute, a blank line is inserted.

tsize=<u>10</u> | n
Defines the amount of space to allot for the terms and term headings. The default is 10 character units. If the value of **tsize=** exceeds the current size of the formatting area (the space between the current margins), the current formatting area size is assigned, and a warning message is issued.

break=<u>none</u> | fit | all
Controls the formatting of the terms and descriptions:

none	The description is on the same line as the term. If the length of the term exceeds the value specified by **tsize=**, the term extends into the description column, and the description starts one space after the term.
fit	The description is placed on the line below the term if the term is longer than the value specified by **tsize=**.
all	All descriptions are placed on the line below the term.

Conditions

- The term-heading tag (**:dthd.**) is paired with the description-heading tag (**:ddhd.**) and precedes the term and description tags (**:dt.** and **:dd.**).
- The term tag requires a description tag.

Example

```
:dl compact tsize=20.
:dthd.:hp2.Mammal:ehp2.
:ddhd.:hp2.Description:ehp2.
:dt.Florida Panther
:dd.Relative of the mountain lion or puma.
:dt.Key Deer
:dd.&odq.Toy&cdq. member of the whitetail deer family.
:dt.Manatee
:dd.Gentle giant sea cow.
:edl.
```

Output

Mammal	**Description**
Florida Panther	Relative of the mountain lion or puma.
Key Deer	“Toy” member of the whitetail deer family.
Manatee	Gentle giant sea cow.

:docprof. (Document Profile)

Purpose

Specifies the heading-level entries to be displayed in the Contents window.

Syntax

Tag	Element	Attributes	End
:docprof.	Contents window entries	toc=	

	Application-controlled window support	dll=' '
		objectname=' '
		objectinfo=' '	

	Push button support	ctrlarea=	

Attributes

toc=

Enables you to control the heading levels displayed in the table of contents. For example, if you want only heading levels 1 and 2 to appear, the tagging is:

```
:docprof toc=12.
```

If no **toc=** value is specified, heading levels 1 through 3 appear in the Contents window. Heading levels 4 through 6 appear as part of the text when the window is displayed.

If a heading tag also specifies a value for **toc=**, the heading-tag value overrides the **:docprof** value until either the end of the file is reached, or another heading **toc=** value is encountered.

dll=' '

Specifies the communication dynamic-link library that IPF loads so that a communication object in the library can be executed in an application-controlled window (see ":acviewport. (Application-Controlled Window)" on page 13-5). For author-defined push buttons, this is the communication object that will receive the HM_NOTIFY message. For a tutorial push button, this is the communication object that will receive the HM_TUTORIAL message.

objectname=' '

Identifies the entry point of the communication object in the dynamic-link library. The value for this attribute is case sensitive.

objectinfo=' '

Identifies parameters to be passed to the object.

ctrlarea=

defines the control areas in a window where you want to display push buttons. For more information on control areas in a window, see "Push Buttons" on page 3-7.

Possible values are:

page	Identifies the control area within the IPF text window.
coverpage	Identifies the control area as the bottom of the coverpage window. This is the default value.
both	Specifies that you want a control area in both the IPF text window, and the coverpage window.
none	Specifies that you do not want a control area. You do not want push buttons.

Description

:docprof is placed at the beginning of the file. It follows the title tag (**:title**), if a title is specified; otherwise, it follows the user-document tag (**:userdoc**).

The **:docprof** tag also provides application-controlled window support by loading any dynamic-link modules specified by **:acviewport** tags. It is possible to have multiple windows, multiple dynamic-link modules, and multiple entry points within a dynamic-link module. You also can use this tag to change the size and function of the coverpage and its client and control windows (see "The Coverpage Window" on page 9-3).

The **:docprof** tag defines the control area in a window where you want to display push buttons.

Conditions

None

Example

```
:userdoc.
:title.
:docprof toc=123 ctrlarea=none.
:euserdoc.
```

Output

When the user selects the + icon in the Contents window, heading levels 1 through 4 are displayed in a tree-structured format. There are no push buttons because of **ctrlarea = none**.

:fig. (Figure)

Purpose
Identifies a figure.

Syntax

Tag	Element	Attributes	End
:fig.	Any text		:efig.

Attributes
None

Description
:fig indicates that what follows is to be formatted exactly as it is entered. Text that exceeds the window area will be clipped.

The figure is displayed in proportional font, with a blank line preceding the text. Because proportional font is used, words will align, but letters and numbers may not.

Conditions
None

Example
:fig.

```
Area   Number  Classification        Code
-----------------------------------------
JOB1

          2      Full-time exempt       1A
          4      Part-time exempt       1B
          4      Full-time nonexempt    2A
          1      Part-time nonexempt    2B
          2      Supplemental           3A

-----------------------------------------
```

:efig.

Output

```
Area   Number  Classification        Code
-----------------------------------------
JOB1

          2      Full-time exempt       1A
          4      Part-time exempt       1B
          4      Full-time non-exempt   2A
          1      Part-time non-exempt   2B
          2      Supplemental           3A

-----------------------------------------
```

:figcap. (Figure Caption)

Purpose
Specifies a figure title.

Syntax

Tag	Element	Attributes	End
:figcap.	Any text		

Attributes
None

Description
:figcap is placed between **:fig** and **:efig**. The text of the caption goes on the same line as the tag, or on the next line.

Conditions

- Use **:figcap** either immediately after **:fig** or immediately before **:efig**.
- The text of the figure caption cannot contain tags or semicolons.

Example
:fig.

```
Area   Number  Classification        Code
-------------------------------------------
JOB2

          5     Full-time exempt       1A
          1     Part-time exempt       1B
          3     Full-time non-exempt   2A
          1     Part-time non-exempt   2B
          1     Supplemental           3A
-------------------------------------------
```

:figcap.Payroll Codes for Area JOB2 :efig.

Output

```
Area   Number  Classification        Code
-------------------------------------------
JOB2

          5     Full-time exempt       1A
          1     Part-time exempt       1B
          3     Full-time non-exempt   2A
          1     Part-time non-exempt   2B
          1     Supplemental           3A
-------------------------------------------
```

Payroll Codes for Area JOB2

:font. (Font)

Purpose

Changes the font to the specified typeface, size, and code page.

Syntax

Tag	**Element**	**Attributes**	**End**
:font.	Fonts	facename=	None
		size=	
		codepage=	

Attributes

facename=
Defines the typeface name of the font. Possible values are:
Courier
'Tms Rmn'
Helv
default

This attribute is required. If **default** is specified, the font is reset to the default system font.

Notice that **facename=** values have initial capitals. These are required; otherwise, the IPF compiler will not recognize them as valid values. No error message is returned when an invalid value for **facename=** is encountered.

size=h x w
Defines the average character height and width, in *points*, of the Presentation Manager image font. (A point is a typesetting measure that is equal to approximately 1/72 of an inch.)
Following are the Presentation Manager image fonts available on all system-supported display adapters:

Face	**Point Sizes**
Courier	8, 10, 12
Helv	8, 10, 12, 14, 18, 24
Tms Rmn	8, 10, 12, 14, 18, 24

The **size=** attribute is required. If the value is set to 0x0, the font is reset to the default system font.

codepage=
Specifies the code page to be used. This is a three-digit number. Possible values are:
437 - U.S. IBM PC
850 - Multilingual
860 - Portuguese
863 - Canadian French
865 - Nordic

See "Country Code Pages" on page 7-7 for a list of countries and their code pages.

The **codepage=** attribute is optional. If no code-page value is specified, the code page of the active system process is used.

Description

:font changes the current font for the text within the current window. When a heading tag defining a new window is encountered, the font resets to the default system font.

You can make as many font changes within a window as you want. If you define highlighted phrases while a font tag is in effect, the highlighted text will be displayed in the font style corresponding to the specified typeface.

When you specify height and width values for a valid font name, you do not have to know the exact point values. If no match is found for a specified font size, IPF uses a "best fit" method to select the font. For example, suppose you specify:

```
:font facename=Helv size=20x12.
```

IPF selects "Helv 18x10" because it is the closest match.

:fn. (Footnote)

Purpose

Identifies a pop-up window.

Syntax

Tag	Element	Attributes	End
:fn.	Pop-up	id=	:efn.

Attributes

id=

Specifies the ID of the footnote. It is used in conjunction with the link tag (see ":link. (Link)" on page 13-43).

Description

The footnote tag encloses information that will be displayed in a pop-up window when the user selects a hypertext link to the information. Footnotes can appear within paragraphs, lists, highlighted phrases, and artwork.

Conditions

- Index entries are not valid within a footnote.
- The *id=* attribute is required.
- One footnote must end before another begins.
- A footnote cannot be linked from a child window.
- Information in a footnote cannot be returned as the result of a search.

Example

The following shows how to enter the footnote ID (here "ddrive") and provide a link to the footnote.

```
:fn id=ddrive.
The information you place here appears in the pop-up window as a
footnote.  For example, you could enter additional information
about the disk drive in a footnote.
:efn.
```

To provide the link that allows the user to view the footnote pop-up, you could enter:

```
:p.Additional information about
:link refid=ddrive reftype=fn.disk drives:elink.
is available.
```

Output

When the information is displayed, *disk drives* is highlighted and clicking on *disk drives* pops up the footnote window.

:h1. through :h6. (Headings)

Purpose

Define window characteristics.

Syntax

Tag	Element	Attributes	End
:h1.—:h6.	Define cross references to internal and external sources.	res= id= name= global tutorial=' '	
	Define origin and size of windows with relation the primary window.	x= y= width= height=	
	Manage the display of information in multiple windows.	group= viewport clear	
	Define the user's control over the window.	titlebar= scroll= rules=	
	Restrict user retrieval of information.	nosearch noprint hide	
	Change heading levels that appear in the Contents window.	toc=	
	Define the control area of a window for displaying push buttons.	ctrlarea= ctrlrefid=	

Attributes

res=
id=
name=

Specify window identifiers.

If you are creating an .HLP file, **res=** is required and can be any integer from 1 through 64000. However, if you are creating an .INF file (compiled by specifying the /INF parameter with the IPFC command), you can use **res=**, **name=** or **id=**. With **name=** and **id=**, you can include alphabetic characters. You CANNOT use these attributes if you plan to concatenate .INF files. Instead, you must use **res=**. For more information see "Concatenating .INF Files" on page 7-5.

global
Indicates to IPF that the window can be a reference in an external database (another IPF .HLP or .INF file). A reference from one IPF database to another is made by specifying **reftype=database** and **object**='filename' with the link tag.

tutorial=' '
Specifies the file name of the tutorial and causes the tutorial choice to be added to the help pull-down when the window is displayed. When the user selects *Tutorial*, the HM_TUTORIAL message specifying the file name of the tutorial is sent to the application. An example of the tagging follows:

```
:h1 tutorial='example.exe'.Test Window
```

x=
y=
width=
height=

Define the size and position of a window. The **x=** and **y=** attributes are values along the x and y axes; they define the origin of the window. The x axis runs horizontally from left to right, and the y axis runs vertically from bottom to top. The position where the values specified for **x=** and **y=** intersect is the the origin of the window. (The 0,0 intersection is the bottom left corner of the parent window.) From this location, width and height are measured. For more information about window coordinates, see "Defining Window Origin and Size" on page 6-4.

Size and position attributes can be given in absolute, dynamic, or relative values:

Absolute value:
A number followed by a letter, which indicates the unit of measure:

c	(Characters): Average character width of the default system font.
x	(Pixels): Dependent on the display adapter in use.
p	(Points): Typesetting measure; equal to approximately 1/72 inch.

Relative value:
A number followed by the percent sign (%), indicating a percentage of the parent-window width or height.

Dynamic value:
A term indicating a window coordinate location that is dependent on the current size and position of the primary window:

center | left | right
For *x*= values: In the center of, flush left in, or flush right in the parent window.

center | top | bottom
For *y*= values: In the center of, at the top of, or at the bottom of the parent window.

Restrictions:
When defining window position and size, you cannot mix absolute values with dynamic or relative values for either of the following combinations of attributes:

The x coordinate and the width
The y coordinate and the height.

If no values for **x=** and **y=** are specified, the origin of the window is 0,0. If you specify values other than 0,0, you also must specify width and height values. Negative values for these attributes are not allowed.

group=
viewport
clear

The **group=** attribute enables you to assign the window a number from 1 through 64000. This associates the window with a heading definition and the IPFC information that follows it. If you do not provide a number with *group=*, IPF assigns the number 0.

A group number can be assigned to a viewport by a heading or link definition. For example, suppose you have a group number specified in a link definition, and another in the heading that the link refers to. If a user action causes the link definition to be selected, the link group number overrides the heading group number. However, if the user selects the heading from either the Contents window or the Index window, the heading group number takes effect.

IPF searches among the open windows to find one with a number matching the one specified with **group=**. If no match is found, IPF opens a new window. If a match IS found, the information associated with the group number is swapped with the information in the matched window.

The **viewport** attribute always opens a window. If you specify both **viewport** and **group=**, and a window with the specified group number is already open, IPF opens another window with the same group number. Thus, it is better that you do NOT specify the **viewport** attribute in a heading that will appear in the Contents window, unless you want your contents entries to always open separate windows.

The **clear** attribute causes IPF to close any open windows before opening a window to display the current window.

titlebar=yes | sysmenu | minmax | <u>both</u> **| none**
rules=border | <u>sizeborder</u> **| none**
scroll=horizontal | vertical | <u>both</u> **| none**

These attributes define Presentation Manager window controls and are used primarily when defining secondary windows. If none of these attributes are specified, the default is to open a window that has a title bar with title bar icon, hide button, maximize button; a sizing border; and vertical and horizontal scroll bars.

nosearch
noprint
hide

These attributes restrict information retrieval and are most often used in heading definitions for secondary windows.

The **nosearch** attribute in a secondary heading definition prevents the heading from being returned as an entry in the search-results window. This does not mean the secondary window is not searched. It is; however, only the primary heading definition that is returned. When the user selects the primary heading definition, the contents of the second window are displayed as part of the primary-window composition.

The **noprint** attribute in a secondary heading definition prevents the contents of a secondary window from being printed as a separate entity. Instead, secondary windows are printed as part of their primary window. The contents of secondary windows are printed in the order in which the link definitions are listed in the primary window.

When used in secondary heading definitions, **nosearch** and **noprint** merely prevent duplication of output (search results or printed copy). When used in regular heading definitions, they prevent retrieval of the information by the user. The only exception to this condition is if the user selects *This section* for either printing or searching.

The **hide** attribute prevents a heading level from appearing in the Contents window. However, there must be at least one heading level that is not hidden.

toc=

Specifies heading levels that are to be entries in the Contents window. When this attribute is encountered in a heading tag, the specified heading levels override any levels specified by **toc=** of the document-profile tag (**:docprof.**) until either the end-of-file is reached or another **toc=** attribute is encountered in a heading tag. If no document-profile tag exists, the heading levels that appear in the Contents window are levels 1, 2, and 3.

ctrlarea=

Specifies which control area in a window you want to display push buttons. When this attribute is encountered in a heading tag, it overrides the **ctrlarea** attribute specified by **:docprof.**. Possible values are:

page	Identifies the control area as the IPF text window.
none	Specifies that you do not want a control area.

For example: If your document consisted of 100 windows, and you wanted only one window to display push buttons in the IPF text window, you would tag your source file as follows:

```
:docprof ctrlarea=none.
⋮
:h1 ctrlarea=page.One Window
```

ctrlrefid=

Refers to the identification value (**id=**) specified by the control area tag (**:ctrl.**). This attribute specifies which control area you want to display for this heading. This attribute is used to overrides the default control area (the coverpage window).

Note: Be careful when using heading tags to define a control area for split windows. A control area cannot be defined in the secondary window heading tag of a split window. You must define the control area (the coverpage window) in the primary window heading tag.

:hide. (Hide)

Purpose

Controls display of IPF text and graphics to meet conditions set by the IPF_KEYS= environment variable.

Syntax

Tag	Element	Attributes	End
:hide.	Hide	key=	:ehide.

Attributes

key=' '

Defines the key that enables a user to view hidden information. You can specify one or more key names. Enclose each key name within apostrophes. When specifying more than one key name, insert a plus (+) sign after each name.

Text entered between the **:hide.** and **:ehide.** tags is only displayed when the **key=' '** attribute matches the entry specified by the user. Use the OS/2 environment variable SET IPF_KEYS= to specify the key name identified for the hidden information. When this feature is used in online documents, the SET_KEYS= line MUST be set in CONFIG.SYS. This feature cannot be altered on a session basis.

If this attribute is not specified, the information identified by the hide tag is displayed.

Description

:hide enables you to determine what text and graphics will be displayed within a window. This function is useful when you want to tailor the information you give to users; for example, if you want to display levels of information on the basis of a user's system configuration, you assign each level a value with the **key=** attribute. When a topic containing hide tags is selected for viewing, IPF will look for an environment variable called IPF_KEYS= to determine what level of information to show the user. If a match is found, the information within the hide tags is displayed; otherwise, the information is hidden from view.

The hide tag affects the display of compiled information. You can hide lines of text within the window, a word or a phrase within a line, or you can hide an instruction to display a bit map, as in the following example.

```
:hide key='level1'.
:artwork name='mybitmap.bmp'.
:ehide.
```

If the user's environment does not contain the key to display the hidden information, IPF wraps the text from the last character or formatting instruction on the line preceding **:hide** to the line following **:ehide**.

In some situations, the same user may need to view more than one level of hidden information. This can be accomplished by setting the IPF_KEYS= to concatenated values; for example:

```
SET IPF_KEYS=LEVEL1+LEVEL2
```

Take care that a window view does not contain an orphan tag. For example, you do not want to hide the information following a list item, unless you have alternate information to display, based on the setting of a key. In the case of an ordered list, which generates sequential numbers, you would not include a list item in the hidden information, unless it is the last item in the list.

Conditions

- You cannot nest one set of hide tags within another.
- You cannot include a heading tag that has a **res=** attribute within a set of hide tags.
- You cannot set IPF_KEYS= on a session basis.

Example

Suppose the following source has been compiled as part of a help library file:

```
:h1 res=001.Installation Procedure
:ol.
:li.
:hide key='usera'.
Instruction for User A.
:ehide.
:hide key='userb'.
Instruction for User B.
:ehide.
:li.
Shut down the system from the desk top.
:li.
Press Ctl+Alt+Del to restart the system.
:eol.
```

If the user's environment includes the setting, IPF_KEYS=USERA, the following is displayed:

1. Instruction for User A.
2. Shut down the system from the desk top.
3. Press Ctl+Alt+Del to restart the system.

:hp1. through :hp9. (Highlighted Phrase)

Purpose

Emphasize text by changing the font style or foreground color.

Syntax

Tag	Element	Attributes	End
:hp*n*.	Highlighting	None	:ehp*n*.

Description

Highlighted-phrase tags are useful for emphasizing words and phrases within text.

Font styles that are displayed for highlighted phrases correspond to the typeface currently being used by IPF. To change from the default system typeface to other typefaces, use **:font**. When you use either the example tag (**:xmp**) or the character-graphics tag (**:cgraphic**), the system monospace typeface is displayed.

Input

```
:sl compact.
:li.:hp1.Highlighted phrase 1 looks like this.:ehp1.
:li.:hp2.Highlighted phrase 2 looks like this.:ehp2.
:li.:hp3.Highlighted phrase 3 looks like this.:ehp3.
:li.:hp4.Highlighted phrase 4 looks like BLUE.:ehp4.
:li.:hp5.Highlighted phrase 5 looks like this.:ehp5.
:li.:hp6.Highlighted phrase 6 looks like this.:ehp6.
:li.:hp7.Highlighted phrase 7 looks like this.:ehp7.
:li.:hp8.Highlighted phrase 8 looks like RED.:ehp8.
:li.:hp9.Highlighted phrase 9 looks like PINK.:ehp9.
:esl.
```

System Default Font Output

Highlighted phrase 1 looks like this.
Highlighted phrase 2 looks like this.
Highlighted phrase 3 looks like this.
Highlighted phrase 4 looks like BLUE.
Highlighted phrase 5 looks like this.
Highlighted phrase 6 looks like this.
Highlighted phrase 7 looks like this.
Highlighted phrase 8 looks like RED.
Highlighted phrase 9 looks like PINK.

Conditions

You cannot nest highlighted-phrase tags.

:i1. and :i2. (Index)

Purpose

Place topics into the index.

Syntax

Tag	Element	Attributes	End
:i1.	Primary entry	id=	
		global	
		roots=''	
		sortkey=''	
:i2.	Secondary entry	refid=	
		global	
		sortkey=''	

Attributes

id=
Provides a cross-reference identifier for the secondary index tag (**:i2**). This attribute is optional and only valid when used with the primary index tag (**:i1**).

global
Specifies that the index entry appear in the OS/2 Master Help Index folder. Entries also appear in the component index. This attribute is only used in Help windows. Online document cannot use this attribute.

roots=*'root words'*
Specifies a list of root words that act as index entries to specified topics. These root words are associated with words defined with the index-synonym tag (**:isyn**). Root words can contain alphabetic and numeric characters, which can be entered in uppercase or lowercase. When entering a string of words, insert a blank space between each word, and enclose the string within apostrophes.

Root words do not appear in the index, so are not viewed by the user, and need not be translated. They are used to create a link between the primary index tag and the index-synonym tag. To enable the user to search for an index entry, use the index-synonym tag to map the root words associated with the entry to synonyms.

sortkey=*'sortkey-text'.index-text*
Specifies a character string that is used for sorting the entry in the index, and another character string that is displayed for the index entry.

The *sortkey-text* character string determines where this entry is placed in the index. The *index-text* character string is displayed for the index entry.

refid=
Provides a reference to the text associated with the primary index tag.

Description

You use the primary and secondary index tags to provide index entries to the information. The attributes associated with each index tag enable you to define related information. Index entries can be used throughout the file, but cannot be placed within a footnote.

The text of the index entry must be on the same line as the tag, and cannot contain other tags. The entry for each primary index entry within the window must be unique. That is, you cannot provide duplicate index entries within the same window. Secondary index entries must refer to an identifier specified for a primary index entry.

When the user selects *Help index* from the Help menu, an Index window is displayed for the help interface. When the user selects *Index* from the Options menu, an Index window is displayed for the online information interface. If the user enters a synonym that matches a root word, the index topics listed for the root word are displayed.

Conditions

- Index entries cannot appear in a footnote.
- When referencing the **:i1.** tag use the **global** attribute on both the **:i1.** and **:i2.** tag.

Example 1

This example shows how to tag your file to include primary and secondary index entries.

```
:i1 id=del.delete
:i2 refid=del.directories
:i2 refid=del.files
```

Output

The index will include the following entry:

```
delete
  directories
  files
```

Example 2

This example shows a file with the index-synonym tag (**:isyn**) and the **roots=** attribute.

```
:h1 id=copy03.Help for Copying
:isyn root=copy.copy copying duplicate duplicating
:isyn root=book.book manual draft manuscript
:isyn root=folder.folder folders document documents
:i1 roots='copy folder'.Copying a document
:i1 roots='book folder'.Test procedures
:p.When copying a file from the current directory to a new
directory, specify the following:
:ul.
:li.The file name
:li.The target directory
:li.The new file name and extension.
:eul.
```

Output

The index-synonym tag creates the following synonym table:

Root word	**Synonym words**
copy	copy copying duplicate duplicating
book	book manual draft manuscript
folder	folder folders document documents

The **roots=** attribute points to the root words, "copy" and "folder," and the list of associated synonyms. For example, if the user searches for "copy" or "folder," the "Copying a document" entry appears because "copy" and "folder," identified by the index **roots=** attribute, match the entries listed for the index synonym **root=** attribute.

A search for the synonym "duplicate" lists "Copying a document" as one of the index choices. A search for the synonym "manual" lists "Test procedures" as an index choice, and a search for "document" lists both "Copying a document" and "Test procedures."

Example 3

This example shows how to specify a sort key to change the location of the entry in the index.

```
:i1 sortkey='point sizes'.changing fonts
:i1.program header
:i1.parameter list
:i1.preface
```

Output

The index will include the entry "changing fonts" at the location where the term "point sizes" would appear in the sorting sequence of the index, as follows:

parameter list
changing fonts
preface
program header

:icmd. (Index Command)

Purpose
Identifies the help window that describes a command.

Syntax

Tag	Element	Attributes	End
:icmd.	Index command	*external command string*	

Attributes

external-command-string
Specifies the command for which help is being defined. The text can contain no other tags.

Description
The help information for a command is assumed to be in the help window in which the index-command tag (**:icmd**) is defined. If the help window provides help for more than one command, an index-command tag should be defined within the heading tag for each command.

The same external command string cannot be specified in more than one index-command tag of an index file; that is, only one help window can be designated as describing a command.

If the compiler finds the same external command string more than once (either from the same or different help windows), the duplicate occurrences are discarded, and a warning message is issued.

Note: The association with entry field and command names is a programming task. In addition, the application developer must define the field with which command windows are to be associated as a command entry field. For more information about programming a command entry field, see "Command Entry Field" on page 8-12.

Conditions
:icmd must follow a heading tag or another index tag.

Example
```
:h1 id=xhlp.Help for Copying
:icmd.Copying
:h1 res=129.Deleting Files
:icmd.Delete
```

Output
At execution time, the index entries enable the compiler to process command helps, create a list of commands for which help is available, and display the help window defined for any of those commands.

:isyn. (Index Synonym)

Purpose

Identifies synonyms and word variations for the help keywords.

Syntax

Tag	Element	Attributes	End
:isyn.	Index synonyms	root=	

Attributes

root=

Links synonyms and variations of words specified in a primary index tag.

To establish a link, specify the same word as specified in the *roots=* attribute of the primary index tag. Then add a period, repeat the root word, and add the list of synonyms and variations, separated by blanks. For example, assume that the value specified for the *roots=* attribute of the primary index tag is "copy." The entry for the index-synonym tag could be:

```
:isyn root=copy.copy copying duplicate duplicating
```

The words entered in the synonym list enable the user to search for terms that may not be in the Index list, and still receive the appropriate help. Lowercase and uppercase characters are treated the same.

Description

:isyn begins a list of synonyms or variations of a word specified by a primary index tag. The compiler uses this list to build a table that serves as a link to the primary index tags. Synonyms determine the topic entries displayed when the user enters words for a search of the index. The compiler matches the entered words with words in the table and links to the topics to be displayed.

The index-synonym tag can be placed within any window that contains related index entries identified by the index tag. The synonyms defined in a window can relate to many topics, and thus to many windows.

Synonyms defined with this tag do not appear in the index.

Conditions

A root word can contain only alphabetic and numeric characters.

Example

```
:h1 id=copy03.Help for Copying
:isyn root=copy.copy copying duplicate duplicating
:isyn root=folder.folder folders document documents
:i1 roots='copy folder'.Copying a document
:p.When copying a file from the current directory to a new
directory, specify the following:
:ul.
:li.The file name
:li.The target directory
:li.The new file name and extension
:eul.
```

Output

The index-synonym tag creates the following synonym table:

Root word	**Synonym words**
copy	copy copying duplicate duplicating
folder	folder folders document documents

The **roots=** attribute points to the root words, "copy" and "folder," and the list of associated synonyms. If the user searches for "copy" or "folder," the words will be displayed because of the matches between the **roots=** attribute of the primary index tag and the **root=** attribute of the index-synonym tag. However, a search for the synonym "duplicate" returns "Copying a document" as an index choice.

:li. (List Item)

Purpose
Identifies an item within a list.

Syntax

Tag	Element	Attributes	End
:li.	List item		

Attributes
None

Description
The format of the list items depends on the type of list: ordered, unordered, or simple. For example, if the list is an ordered list, a number precedes each list item. If the list is an unordered list, a bullet precedes each item. See ":ol. (Ordered List)" on page 13-51, ":sl. (Simple List)" on page 13-60, and ":ul. (Unordered List)" on page 13-65 for more information.

Conditions
None

Example
```
:p.To remove a diskette&colon.
:ol.
:li.Open the drive door.
:li.Remove the diskette.
:li.Put the diskette in a safe place.
:eol.
```

Output
To remove a diskette:

1. Open the drive door.
2. Remove the diskette.
3. Put the diskette in a safe place.

:lines. (Lines)

Purpose
Turns formatting off.

Syntax

Tag	Element	Attributes	End
:lines.	Lines	align=	:elines.

Attributes

align=left|right|center
Places the entered lines to the left in the window, to the right, or in the center.

Description
:lines specifies that the following text is to be formatted exactly as it is entered. The attributes enable you to align the text within the window. Text that is too long for the window is clipped.

Proportional fonts are used for formatting, so the text may not be displayed exactly as entered.

Conditions
None

Example 1
This example aligns text to the left.

```
:lines align=left.
The warehouse contained:
  12  desks
  28  chairs
  15  lamps
  39  typewriters
  11  pictures
:elines.
```

Example 2
This example aligns text to the right.

```
:lines align=right.
The warehouse contained:
  12  desks
  28  chairs
  15  lamps
  39  typewriters
  11  pictures
:elines.
```

Output

Example 1

```
The warehouse contained:
  12  desks
  28  chairs
  15  lamps
  39  typewriters
  11  pictures
```

Example 2

```
The warehouse contained:
               12  desks
              28  chairs
              15  lamps
        39  typewriters
            11  pictures
```

:link. (Link)

Purpose

Activates a link to additional information.

Syntax

Tag	Elements	Attributes	End

:link.	Link to more information	reftype=
		res=
		refid=
		database=' '
		object=' '
		data=' '
			:elink.

	Automatic linking	auto viewport dependent split group=	
	Define window position and size	vpx= vpy= vpcx= vpcy=	
	Define window controls	titlebar= scroll= rules=	

Attributes

reftype=

Defines the type of link. Possible values are **hd, fn, launch,** and **inform**.

reftype=hd

Links to a heading. The heading definition (or an overriding definition in the link) causes its information to be displayed in the current window or another window. The integer value of **refid=** identifies the ID of the heading. If the heading is in an external IPF database, its file name is specified with the **database=** attribute.

In the following example, selection of the hypertext link causes the external database, EDITOR.HLP, to be loaded, and the heading with the ID of 001 to be displayed.

```
:link reftype=hd refid=001.
     database='editor.hlp'.
Editing Functions
:elink.
```

The heading definition in the external database must contain the **global** attribute. If the link to the file cannot be resolved, the hypertext phrase in the link will not be highlighted.

reftype=fn

Links to a footnote. Its contents are displayed in a pop-up window in the current window. The **refid=** attribute specifies the ID of the footnote.

Restriction: A split window cannot contain a link to a footnote.

reftype=launch
Starts a Presentation Manager program. The file name of the program is specified with the **object=** attribute. Any parameters to the program are specified with **data=**. In the following example, the hypertext link starts the System Editor and opens the file, MYFILE, for editing.

```
:link reftype=launch
      object='c:\os2\e.exe'
      data='myfile'.
Start Editor
:elink.
```

reftype=inform
Causes a message to be sent to the application. The **res=** attribute is required and is an integer value that directs the application to perform some application-specific function. When using this attribute, DO NOT use **:elink.**. For example:

```
:link reftype=inform res=1000 auto.
```

auto
viewport
dependent
split
group=

With the **auto** attribute, you can define any of the link types described above, with the exception of a footnote link, as an automatic link.

The automatic-link definition follows a heading definition and is activated as soon as a reference to the heading definition is made. The reference can be made by the user selecting an IPF window entry (for example, the Contents window), or by a hypertext or hypergraphic link.

Following are the automatic-link actions that can be specified, and the attributes used:

- **Open a secondary window** when the heading that contains the link is referred to:

  `auto reftype=hd viewport dependent res=`

  Note the inclusion of the **dependent** attribute. Usually, the information in an automatic window is dependent on the information in its secondary window. Specifying **dependent** causes an automatic window to close when the user closes the window of the secondary that contains the automatic link.

- **Open secondary windows** when the heading of the primary window that contains the links is referred to:

  `auto reftype=hd split res=`

  Restriction: The primary heading cannot contain text or graphics; only links to its secondary headings. For more information, see "Split Windows" on page 6-16.

- **Start a Presentation Manager program** when the heading that contains the link is referred to:

  `auto reftype=launch object= data=`

- **Send the application a message** when the heading that contains the link is referred to:

  `auto reftype=inform res=`

To display more than one window on the screen, you must assign a unique group number to each window with the **group=** attribute. This attribute can be specified with **:link.** or the heading tag. For more information about group numbers, see "Displaying Multiple Windows" on page 6-7.

vpx=
vpy=
vpcx=
vpcy=

Define the size and position of the window. Any values specified by these attributes override size and position values specified by the attributes in a heading tag. (See ":h1. through :h6. (Headings)" on page 13-27 for details about these attributes.)

titlebar=yes | sysmenu | minmax | both | **none**
scroll=horizontal | vertical | both | **none**
rules=border | sizeborder | **none**

Define window controls. Any values specified by these attributes override window-control values specified by the attributes in a heading tag. (See ":h1. through :h6. (Headings)" on page 13-27 for details about these attributes.)

When **titlebar=yes** is specified the window displays a titlebar WITHOUT the system menu symbol, the hide button, and the maximize button.

:lm. (Left Margin)

Purpose

Sets the left margin of the text.

Syntax

Tag	Element	Attributes	End
:lm.	Left margin	margin=	

Attributes

margin=
Specifies where the left margin of the text is to begin. To set the margin for the current line, specify a number greater than the position of the cursor. For example, to set the left margin to 15, begin the left margin tag before space 15. Otherwise, the margin becomes effective on the next line.

Note: When counting character spaces, you are actually counting average character widths.

Description

Use the left-margin tag and the right-margin tag (**:rm**) to specify the boundaries of the text in the window.

When the text window is sized, the text area adjusts from the right to fit within the specified margin boundaries; that is, the right margin adjusts to the new window size. The left margin remains constant. If the window is sized smaller than the specified margins, the margins remain the same, and the text area is reduced to one character space.

You can place multiple margin tags in your file. The margins specified remain effective until they are reset. If no margin value is specified, the default is 1.

Conditions

None

Example

This example shows the use of both margin tags.

```
:p.
:rm margin=10.
:lm margin=20.This text begins 20 spaces to the
right of the left window border and ends 10 spaces to the
left of the right window border.
All text is aligned as specified
by the margin values. :lm margin=5.Here the left margin
is changed to 5.  Because this margin tag begins
more than 5 spaces on the line, the margin specified
becomes effective on the following line, and the text
begins 5 spaces from the left window border.
The right margin remains unchanged.
```

Output

This text begins 20 spaces to the right of the left window
border and ends 10 spaces to the left of the right window
border. All text is aligned as specified by the margin values.
Here the left margin is changed to 5. Because this margin tag begins more
than 5 spaces on the line, the margin specified becomes effective on the
following line, and the text begins 5 spaces from the left window border.
The right margin remains unchanged.

:lp. (List Part)

Purpose

Identifies an explanation within a list.

Syntax

Tag	Element	Attributes	End
:lp.	List part		

Attributes

None

Description

:lp. can be entered anywhere within the list. The text following the tag starts at the left margin of the current list item. It is not numbered or lettered. Using the list-part tag does not interrupt the sequence of the list.

Conditions

None

Example

```
:p.To remove a diskette&colon.
:ol.
:li.Open the drive door.
:lp.Before removing the diskette, make sure all drive activity
has stopped.
:li.Remove the diskette.
:li.Put the diskette in a safe place.
:eol.
```

Output

To remove a diskette:

1. Open the drive door.

Before removing the diskette, make sure all drive activity has stopped.

2. Remove the diskette.
3. Put the diskette in a safe place.

:note. (Note)

Purpose

Starts a note.

Syntax

Tag	Element	Attributes	End
:note.	Note	text=''	

Attributes

text=''
Enables you to change the name of the note.

Description

:note identifies a single-paragraph note. When the tag is encountered, a blank line is inserted, and the note starts at the left margin with **Note:** followed by two blank spaces. The start of another tag ends the note, so no end tag is needed.

When the tag is used within a list, the note aligns with the text of the items within the list.

Use the **text=' '** attribute to give the note a specific name.

Conditions

None

Example 1

```
:note.
This text appears within a note.
The word :hp2.Note:ehp2. aligns
with the text that precedes it.
```

Example 2

```
:note text='Text note:'.
The name of this note is :hp2.Text note:ehp2..
The name of the note replaces
the word :hp2.Note:ehp2..  The name of the note
aligns with the text that precedes it.
```

Output

Example 1

Note: This text appears within a note. The word **Note** aligns with the text that precedes it.

Example 2

Text note: The name of this note is **Text note**. The text for the note replaces the word **Note**. The name of the note aligns with the text that precedes it.

:nt. (Note)

Purpose

Starts a note that can have multiple paragraphs.

Syntax

Tag	Element	Attributes	End
:nt.	Note	text=''	:ent.

Attributes

text=''
Enables you to change the name of the note.

Description

:nt starts a new paragraph with **Note:** followed by two blank spaces and the first line of the text. The second and succeeding lines of text align with the first line, to the right of **Note:**.

Notes can be placed within lists and paragraphs. However, unlike the **:note.** tag, **:nt** requires an end tag.

You can use the **text=''** attribute to assign a specific name to the note.

Conditions

None

Example

```
:nt.
Use this tag to include paragraphs in a note.
You also can use it within
paragraphs and lists.
:p.End this tag before you begin another note
tag. :ent.
```

Output

Note: Use this tag to include paragraphs in a note. You also can use it within paragraphs and lists.

End this tag before you begin another note tag.

:ol. (Ordered List)

Purpose

Starts a sequential list of items or steps.

Syntax

Tag	Element	Attributes	End
:ol.	Ordered list	compact	:eol.

Attributes

compact
Causes the list to be formatted without a blank line between each list item. If you omit **compact**, a blank line appears between each list item.

Description

:ol. indicates the start of an ordered list. Items in the list are entered with the list-item tag (**:li.**). The output is an indented list with each item numbered. Use the list-part tag (**:lp.**) for paragraphs within the list.

Ordered lists can be nested or imbedded within other lists. When this is done, the first list has sequential numbers at the left margin, and the nested list has sequential *letters* indented two spaces. After the second list, the number-letter sequence repeats for each successive ordered list.

Be sure to end each list with the end-list tag.

Example

```
:p.To remove a diskette&colon.
:ol.
:li.Open the drive door&colon.
:ol compact.
:li.Remove two screws.
:li.Lift the door.
:eol.
:li.Remove the diskette.
:li.Put the diskette in a safe place.
:eol.
```

Output

To remove a diskette:

1. Open the drive door:

   a. Remove two screws.
   b. Lift the door.

2. Remove the diskette.

3. Put the diskette in a safe place.

:p. (Paragraph)

Purpose

Starts a new paragraph.

Syntax

Tag	Element	Attributes	End
:p.	Paragraph		

Attributes

None

Description

Each paragraph identified by a paragraph tag formats as an unindented block of text. Paragraphs placed within a list align with the text of the list. When paragraphs are placed within a note, the text of the paragraph aligns with the text of the note.

Conditions

None

Example

```
:p.Paragraph tags cause a blank line before the text.
When placed within a list or note, the text of the paragraph
aligns with the text of the list or note.
:ul.
:li.Paragraph tags
:p.Paragraph tags are flexible and can be used
with most tags.
:li.Note tags
:p.Note tags can include paragraphs.
:eul.
```

Output

Paragraph tags cause a blank line before the text. When placed within a list or note, the text of the paragraph aligns with the text of the list or note.

- Paragraph tags

  Paragraph tags are flexible and can be used with most tags.

- Note tags

  Note tags can include paragraphs.

:parml. (Parameter List)

Purpose

Starts a two-column list of parameter terms and descriptions.

Syntax

Tag	Element	Attributes	End
:parml.	Parameter list	tsize=	:eparml.
		break=	
		compact	
:pt.	Parameter term		
:pd.	Parameter definition		

Attributes

tsize=
Specifies the space allocated for the parameter term. The default is 10 character units.

break=all | fit | none
Controls the formatting of the parameter terms and descriptions:

break=all
Causes the description to begin on the line below the parameter term, next to the space allocated by **tsize=**. This is the default.

break=fit
Causes the parameter description to begin on the same line as the term, if the term has fewer characters than specified by **tsize=**. If the term has more characters, the description begins on the line below the term.

break=none
Causes the description to begin on the same line as the term. If the term has more characters than specified by **tsize=**, it continues into the description area. The description starts one space after the end of the term.

compact
Causes the list to be formatted without a blank line between each list item. If you omit **compact**, a blank line appears between each item.

Description

Parameter lists are similar to definition lists; they define terms and descriptions that format in two columns. The elements of the parameter-list tag are the parameter-term tag (**:pt.**) and the parameter-description tag (**:pd.**). The term tag identifies the term, and the definition tag identifies the description.

Parameter lists can occur anywhere in text; you can nest them within other lists, and you can nest other lists within parameter lists.

Conditions

- Each parameter-term tag requires a parameter-description tag.
- Each parameter list requires an end-parameter-list tag.

Example

```
:parml compact tsize=10 break=none.
:pt.Tree
:pd.Plant life in forest
:pt.Orange
:pd.Fruit on tree
:pt.Cow
:pd.Animal on farm
:eparml.
```

Output

```
Tree      Plant life in forest
Orange    Fruit on tree
Cow       Animal on farm
```

:pbutton (Push Button)

Purpose

Defines author-defined push buttons.

Syntax

Tags	Element	Attributes	End
:pbutton.	Author-defined pushbuttons	id= res= text=' '	

Attributes

id=

Specifies the identification value for a push button that you define. The identification value can be alpha or alphanumeric. This identification value is referenced by the control area tag (**:ctrl.**).

res=

Specifies the *resource* identification value for a push button that you define. This value is returned with the HM_NOTIFY and HM_CONTROL messages and can be any integer greater than 256 (0 to 256 are reserved for use by IPF).

text=' '

Specifies the text for the push button that you define. Define the mnemonic for the pushbutton by placing the tilde (~) character before the mnemonic character. For example:

```
:pbutton id=xmp res=300 text='~Example'.
```

Note: Make sure the mnemonic you specify for author-defined push buttons does not conflict with the mnemonics of the predefined set of pushbuttons, or with any of IPF's shortcut keys. See ":ctrl. (Control Area)" on page 13-14, for a description of the control area tag (**:ctrl.**) and a list of the predefined push buttons and their associated mnemonics.

Description

Use the push button tag (**:pbutton.**) to define author-defined pushbuttons. For more information, see "Author-Defined Push Buttons" on page 3-11.

:pd. (Parameter Description)

Purpose

Starts the description for a parameter term in a parameter list.

Syntax

Tag	Element	Attributes	End
:pd.	Parameter description		

Attributes

None

Description

The text that follows **:pd.** describes the term identified by **:pt**. The description formats in the right column, as defined by the values of **tsize=** and **break=**. For a description of **:parml.** see ":parml. (Parameter List)" on page 13-53.

A parameter list can have multiple parameter-term and parameter-description tags. However, each term tag requires a description tag.

Conditions

- The parameter-description tag follows the parameter-term tag.
- The parameter-description tag is valid only within a parameter list.

Example

```
:parml compact tsize=15 break=all.
:pt.Tree
:pd.Plant life in forest
:pt.Orange
:pd.Fruit on tree
:pt.Cow
:pd.Animal on farm
:eparml.
```

Output

Tree
                Plant life in forest
Orange
                Fruit on tree
Cow
                Animal on farm

:pt. (Parameter Term)

Purpose

Identifies a term in a parameter list.

Syntax

Tag	Element	Attributes	End
:pt.	Parameter description		

Attributes

None

Description

The term identified by **:pt.** formats in the left column. The **:pt.** tag requires a parameter-description tag (**:pd.**); the description formats in the right column.

Conditions

- The parameter-term tag requires a parameter-description tag.
- The parameter-term tag precedes the parameter-description tag.
- The parameter-term tag is valid only within a parameter list (for a description of **:parml.**, see ":parml. (Parameter List)" on page 13-53).

Example

```
:parml compact tsize=15 break=all.
:pt.Tree
:pd.Plant life in forest
:pt.Orange
:pd.Fruit on tree
:pt.Cow
:pd.Animal on farm
:eparml.
```

Output

```
Tree
                Plant life in forest
Orange
                Fruit on tree
Cow
                Animal on farm
```

:rm. (Right Margin)

Purpose

Sets the right margin of the text.

Syntax

Tag	Element	Attributes	End
:rm.	Right margin	margin=	

Attributes

margin=
Enables you to indicate the number of character spaces from the right border of the window the text is to end. For example, **margin=60** means that the text is to end 60 spaces from the right border.

Note: When counting character spaces, you are actually counting average character widths.

Description

Use **:rm** with the left-margin tag (**:lm**) to specify the boundaries of the text in the window. The left-margin tag specifies where the text is to start, and the right-margin tag specifies where it is to end.

You can enter margin tags at the beginning of the line of text or while you are entering the text. Margin tags that begin the line of text cause text on that line and the following lines to align with the values specified. Margins set while you enter text become effective on the current line or on the next line, depending on where the margin tag begins. For example, to set the right margin to 60 (that is, 60 spaces before the right border of the window), begin the right-margin tag at least 60 spaces to the left of the right border. When the file is displayed, the text entered after the margin tag aligns to the value specified on that line.

If the margin tag is started after the specified boundary, the margin becomes effective on the next line.

When the text window is sized, the text area adjusts from the right to fit within the specified margin boundaries; that is, the right margin adjusts to the window size. The left margin stays the same. If the window is sized smaller than the specified margins, the margins remain the same, and the text area is reduced to one character space. If no value is specified for **margin=**, the default for the right margin is 1.

You can place multiple margin tags in your file. The specified margins remain effective until they are reset.

Example

```
:lm margin=1.
:rm margin=44.
:p.In this
example, the left margin is 1.  The right margin
is 44.  The margins are set before the text;
therefore, when the file is displayed, the text
formats according to the margins set.
The text begins at space 2 and ends 44 spaces before
the right window border.  If the margin specified is
less than the current cursor position on the screen,
the margins set become effective on the following
line.  For example, if the current cursor position is
60 spaces to the left of the right window border and
you set the right margin to 50, the margin is
effective on the current line.  However, if the right
margin is set to 65, the margin becomes effective
on the next line.
:p.
:lm margin=5.
:rm margin=60.Here the left margin is set to 5
and the right margin is set to 60.  This means that
the left margin begins 5 spaces to the right of the
left border.  The right margin ends 60 spaces to the
left of the right border.
```

Output

In this example, the left margin is 1. The right margin is 44. The margins are set before the text; therefore, when the file is displayed, the text formats according to the margins set. The text begins at space 2 and ends 44 spaces before the right window border. If the margin specified is less than the current cursor position on the screen, the margins set become effective on the following line. For example, if the current cursor position is 60 spaces to the left of the right window border and you set the right margin to 50, the margin is effective on the current line. However, if the right margin is set to 65, the margin becomes effective on the next line.

Here the left margin is set to 5 and the right margin is set to 60. This means that the left margin begins 5 spaces to the right of the left border. The right margin ends 60 spaces to the left of the right border.

:sl. (Simple List)

Purpose

Starts a nonsequential list of items.

Syntax

Tag	Element	Attributes	End
:sl.	Simple list	compact	:esl.

Attributes

compact
Causes the list to be formatted without a blank line between each list item. If you omit **compact**, a blank line appears between each item.

Description

:sl. identifies items that do not require a sequential listing. Items in a simple list are not indented and do not have bullets, hyphens, or dashes preceding them. Simple lists can be nested within other lists. When nested, a simple list is indented four spaces to the right of the left margin of the list that contains it. Each list requires an end-list tag.

The simple-list tag requires the list-item tag (**:li.**) to identify items in the list. You can use the list-part tag (**:lp.**) to include paragraphs in the list.

Conditions: None

Example

```
:p.Bring the following for lunch&colon.
:sl.
:li.Fruit, for example&colon.
:sl compact.
:li.An apple
:li.An orange
:li.A pear
:li.A banana
:esl.
:li.Sandwich
:li.A drink, for example&colon.
:sl compact.
:li.A soda
:li.Juice
:li.Milk.
:esl.
:esl.
```

Output

Bring the following for lunch:

Fruit, for example:

```
An apple
An orange
A pear
A banana
```

Sandwich

A drink, for example:

```
A soda
Juice
Milk.
```

:table. (Table)

Purpose

Formats information as a table.

Syntax

Tag	Element	Attributes	End
:table.	Tables	cols=' '	:etable.
		rules=	
		frame=	
:row.	Rows	None	None
:c.	Columns	None	None

Attributes

cols=' '
Specifies the width, in character spaces, of each column; for example: **cols=**'10 15 20'.

rules=
Specifies whether the table will have horizontal and vertical rules. Following are the possible values and meanings:

both	Horizontal and vertical rules
horiz	Horizontal rules only
vert	Vertical rules only
none	No rules

Note: The default is **both**.

frame=
Specifies whether the table will have borders. Following are the possible values and meanings:

rules	A horizontal line at the top and bottom of the table
box.	A box around the table
none	No borders.

Note: The default is **box**.

The **:row.** tag specifies the start of each row in the table. The **:c.** tag specifies the text for each column entry in the table. The text provided with the **:c.** tag is formatted within the column. However, if a single word is longer than the specified width of the column, the word will be clipped.

Example

The following defines a table with three columns and two rows. The width of each column is 15, 20, and 25 character spaces.

```
:table cols='15 20 25' rules=both frame=box.
:row.
:c.Row 1 Col 1
:c.Row 1 Col 2
:c.Row 1 Col 3
:row.
:c.Row 2 Col 1
:c.Row 2 Col 2
:c.Row 2 Col 3
:etable.
```

Output

Row 1 Col 1	Row 1 Col 2	Row 1 Col 3
Row 2 Col 1	Row 2 Col 2	Row 2 Col 3

:title. (Title)

Purpose

Provides a name for the online document.

Syntax

Tag	Element	Attributes	End
:title.	Title		

Attributes

None

Description

The text that follows **:title.** provides a name for the online document. The title of an online document can contain up to 47 characters, including spaces and blanks. If the title exceeds 47 characters, the IPF Compiler displays an error message.

When you display the online document, the title appears on the title line of the main window. The title is limited to one line. Word wrapping does not occur in the title of an online document.

Conditions

Use the **:title.** tag only for the title of an online document. Do not use it for online help windows.

Example

```
:userdoc.
:title.Using the Information Presentation Facility
:h1 res=100.Creating an Index
:p.This section shows you how to create index entries.
:euserdoc.
```

Output

When you compile this file, "Using the Information Presentation Facility" is displayed on the title line of the main window of the online document.

"Creating an Index" is listed as an entry in the contents window. If you select "Creating an Index", the window with this heading and the accompanying text is displayed in the text information area, overlaying the contents window.

:ul. (Unordered List)

Purpose

Starts a list of nonsequential items.

Syntax

Tag	Element	Attributes	End
:ul.	Unordered list	Compact	:eul.

Attributes

compact
Causes the list to be formatted without a blank line between each list item. If you omit **compact**, a blank line appears between each item.

Description

:ul. indicates the start of a list of items that do not require sequential listing. The list-item tag (**:li.**) identifies the items within the list. The list-part tag (**:lp.**) is used to include paragraphs within the list.

Unordered list items are indented, and a bullet (lowercase "o") precedes each item. Unordered lists can be nested within other lists. If placed within an ordered list or a simple list, the nested list will be indented four spaces, and each item will be preceded by a bullet. If placed within another unordered list, the nested list will be indented four spaces, and each item will be preceded by a dash.

Conditions

None

Example

```
:p.Before leaving for the day remember to&colon.
:ul.
:li.Turn off the computer
:li.Turn off the lights&colon.
:ul compact.
:li.Ceiling
:li.Desk
:eul.
:li.Secure all equipment.
:eul.
```

Output

Before leaving for the day remember to:

- Turn off the computer
- Turn off the lights:
  - Ceiling
  - Desk
- Secure all equipment.

:userdoc. (User Document)

Purpose

Identifies the source file that is to be compiled.

Syntax

Tag	Element	Attributes	End
:userdoc.	User Document		:euserdoc.

Attributes

None

Description

:userdoc must be the first tag in the source file. It signals the compiler to begin compiling the tagged text that follows. All other tags that define how the text is to be formatted follow this tag.

The end-user-document tag (**:euserdoc**) identifies the end of the tagged text and the end of the source file. It must be the last tag in the source file.

Conditions

None

Example

```
:userdoc.
.
.
.
:euserdoc.
```

:warning. (Warning)

Purpose

Alerts the user of a risk or possible error condition.

Syntax

Tag	Element	Attributes	End
:warning.	Warning	text=''	:ewarning.

Attributes

text=''
Enables you to give a specific name to the warning notice.

Description

A warning notice alerts the user to a possible risk, such as an error condition in the system. It should appear before the text that it discusses. Use the **text=''** attribute to provide a specific name for the warning notice.

Conditions

None

Example 1

```
:warning.
The disk contains bad sectors.
:ewarning.
```

Example 2

```
:warning text='Bad disk:'.
The disk contains bad sectors.
:ewarning.
```

Output

Example 1

Warning: The disk contains bad sectors.

Example 2

Warning: The disk contains bad sectors.

:xmp. (Example)

Purpose

Turns formatting off.

Syntax

Tag	Element	Attributes	End
:xmp.	Example		:exmp.

Attributes

None

Description

Text entered between **:xmp** and **:exmp** is formatted as entered, in a monospace font. The text is indented two spaces from the left margin of the window. Lines that are too long to fit within the window are clipped.

Conditions

- An example cannot be placed within another example.
- An end-example tag is required.

Example

```
:xmp.
 #define INCL_WIN
 #include <os2.h>

 MRESULT EXPENTRY MyObject(PACVP pACVP, PCH ObjectInfo)
        {

 HWND hwndMyACVP;             /* handle to the application-controlled */
                              /* window that this procedure creates   */

:exmp.
```

Output

```
 #define INCL_WIN
 #include <os2.h>

 MRESULT EXPENTRY MyObject(PACVP pACVP, PCH ObjectInfo)
        {

 HWND hwndMyACVP;             /* handle to the application-controlled */
                              /* window that this procedure creates   */
```

Chapter 14. Symbols

This appendix discusses the symbols you can use to display special characters that you may want to include in your file. Symbols can be used to specify characters that are not on the keyboard. Each symbol represents a single character. When tagging your file to include symbols, begin each symbol with an ampersand (&) and end the symbol with a period (.). For example, to place a square bullet (■) in a file, you would enter:

```
&sqbul.
```

Symbols are case sensitive, that is uppercase characters produce different symbols than lowercase characters. Therefore, when tagging the file to include a symbol, enter the tag for the symbol exactly as it is shown in the symbols chart.

Note: All symbols in the following chart are also in the APSYMBOL.APS file. This file is in the C:/TOOLKT20/IPFC directory and can be edited with any text editor. However, some National Language code pages require a different symbols file. See "National Language Support" on page 7-7, for a list of these files.

Symbol	**Symbol Name**	**Character**
&aa.	a acute	á
&ac.	a circumflex	â
&ae.	a umlaut	ä
&Ae.	A umlaut	Ä
&ag.	a grave	à
&aelig.	ae ligature	æ
&AElig.	AE ligature	Æ
&Alpha.	Alpha	A
&amp.	Ampersand	&
&and.	and	∧
&angstrom.	angstrom	Å
&ao.	a overcircle	å
&Ao.	A overcircle	Å
&apos.	Apostrophe	'
&bx2022.	ASCII code 185	╣
&bx2020.	ASCII code 186	║
&bx0022.	ASCII code 187	╗
&bx2002.	ASCII code 188	╝
&bx2200.	ASCII code 200	╚
&bx0220.	ASCII code 201	╔
&bx2202.	ASCII code 202	╩
&bx0222.	ASCII code 203	╦
&bx2220.	ASCII code 204	╠
&bx0202.	ASCII code 205	═
&bx2222.	ASCII code 206	╬
&asterisk.	Asterisk	*
&atsign.	At sign	@
&bslash., &bsl.	Back slash	\
&Beta.	Beta	B

Symbol	**Symbol Name**	**Character**
&bxas., &bxbj.	box ascender	┴
&bxcr., &bxcj.	box cross	┼
&bxde., &bxtj.	box descender	┬
&bxh.	box horizontal	─
&bxll.	box lower-left	└
&bxlr.	box lower-right	┘
&bxri., &bxrj.	box right junction	┤
&bxul.	box upper-left	┌
&bxur.	box upper-right	┐
&bxv.	box vertical	│
&cc.	c cedilla	ç
&Cc.	C cedilla	Ç
&caret.	Caret symbol	^
&cdq.	Close double quote	”
&cdqf.	Close French double quote	»
&csq.	Close single quote	’
&comma.	Comma	,
&colon.	Colon	:
&dash.	Dash	–
&degree., &deg.	degree	°
&divide	divide	÷
&dollar.	Dollar sign	$
&dot.	dot	.
&darrow.	Down arrow	↓
&ea.	e acute	é
&Ea	E acute	É
&ec.	e circumflex	ê
&ee.	e umlaut	ë
&eg.	e grave	è
&emdash.	Em dash	—
&endash.	En dash	–
&eq., &equals., &eqsym.	Equal sign	=
&xclm., &xclam.	Exclamation point	!
&fnof.	function of	ƒ
&gtsym., &gt. &gesym.	Greater than	>
&house.	House	⌂
&hyphen.	Hyphen	-
&ia.	i acute	í
&ic.	i circumflex	î
&ie.	i umlaut	ï
&ig.	i grave	ì
&inve.	inverted exclamation mark	¡
&invq.	inverted question mark	¿
&larrow.	Left arrow	←
&lahead.	Left arrowhead	◄

Symbol	Symbol Name	Character
&lbrace., &lbrc.	Left brace	{
&lbracket. &lbrk.	Left bracket	[
&lpar. , &lparen.	Left parenthesis	(
&lnot.	logical not	¬
&mdash.	M dash	—
&minus.	Minus sign	−
&mu.	Mu	μ
&ndash.	N dash	–
&nt.	n tidle	ñ
&Nt.	N tidle	Ñ
&lnot., &notsym.	not symbol	¬
&numsign.	Number sign	#
&oa.	o acute	ó
&oc.	o circumflex	ô
&og.	o grave	ò
&oe.	o umlaut	ö
&Oe.	O umlaut	Ö
&frac14.	one fourth	¼
&frac12.	one half	½
&odq.	Open double quote	“
&odqf.	Open French double quote	«
&osq.	Open single quote	‘
&percent.	Percent	%
&per.	Period	.
&plus.	Plus sign	+
&plusmin., &pm.	plusminus	±
&Lsterling.	pound sterling	£
&rbl.	Required blank	
&rarrow.	Right arrow	→
&rahead.	Right arrowhead	►
&rbrace., &rbrc.	Right brace	}
&rbracket., &rbrk.	Right bracket	]
&rpar., &rparen.	Right parenthesis	)
&semi.	Semicolon	;
&box14.	shaded box 1/4 dots	░
&box12.	shaded box 1/2 dots	▒
&box34.	shaded box 3/4 dots	▓
&slash., &slr.	Slash	/
&BOX.	solid box	█
&BOXBOT.	solid box bottom half	▄
&splitvbar.	Split vertical bar (piping symbol)	¦
&sqbul.	square bullet	■
&sup2.	superscript 2	^{2}
&tilde.	Tilde	~
&ua.	u acute	ú

Symbol	Symbol Name	Character
&uc.	u circumflex	û
&ug.	u grave	ù
&ue.	u umlaut	ü
&Ue.	U umlaut	Ü
&us.	Underscore	_
&aus.	underscored a	ª
&ous.	underscored o	º
&ye.	y umlaut	ÿ

Appendix A. Compiler Error Messages

This appendix lists the error messages sent by the IPF compiler.

Description and Format of Error Messages

There are three types of error messages:

- *Warning Level 1.* They are the most severe.
- *Warning Level 2.* They are moderately severe.
- *Warning Level 3.* They are the least severe.

These error messages have the following format.

```
<C:\IPFC\YOURFILE.IPF:999> 124: Invalid tag in footnote [ ]
 └──┬───┘└─────┬───────┘ └┬┘ └┬┘ └──────┬─────────────┘ └┬┘
    │          │          │   │         │                │
    │          │          │   │         │          Optional error information.
    │          │          │   │         │          Tag, filename, etc.
    │          │          │   │         │
    │          │          │   │         └─ Error message
    │          │          │   │
    │          │          │   └─ Error code
    │          │          │
    │          │          └─ Line number error occurred at in source file
    │          │
    │          └─ Filename of source file
    │
    └─ Drive and path of source file
```

Warning Level 1 Messages

101 **Invalid document body**

Explanation: No userdoc/euserdoc match

102 **Invalid tag syntax**
103 **Missing hypertext information**
104 **Cannot hide parent head level**

Explanation: Preceding head level must be hidden

105 **Illegal context for tag**

Explanation: Tags are not properly matched, a tag is used incorrectly, or a tag is placed incorrectly.

106 **List start tag missing-tag ignored**
107 **List end tag not matched-tag ignored**
108 **Ignoring unmatched tag**
109 **Cannot open file**

Explanation: SYSTEM ERROR. Filename or path is incorrect, file doesn't exist, or other system problem.

110 **No id for this reference**
111 **No references to this footnote**
112 **No id for this footnote**
113 **No text found in tag**
114 **Page is too big**

Explanation: Panel is too big. Maximum size is 16,000 words and punctuation marks. (Note maximum size is language dependent.)

115 **Bitmap is too large or invalid and will be ignored**
116 **Cannot create panel(s)**
117 **Duplicate text in tag**
118 **Duplicate root word**
119 **Duplicate tag in tag file**
120 **Ignoring text before :h1 tag**
121 **Invalid head level**

Explanation: Head levels are not in consecutive order.

Example: If h1 and h3 are used and h2 is missing, this error will occur.

122 **Definition term or header not matched**
123 **Unexpected end of file**

Explanation: This may be caused by an ending tag not being found, a corrupted or truncated source file, or a control-Z character found before the true end of file.

124 **Invalid tag in footnote**
125 **Not enough memory**

Explanation: SYSTEM ERROR. Close some applications to free some memory.

126 **Cannot free memory**

Explanation: SYSTEM ERROR. System could not free memory.

127 **Cannot read file**

Explanation: SYSTEM ERROR. Source file may be corrupted.

128 **Invalid file type**

Explanation: File is corrupt or may not be an IPF tagged source file.

129 **Document is too big — unique words exceed 16,000**
130 **A DT tag is not defined**
131 **A PT tag is not defined**
132 **Cannot write to a file**

Explanation: SYSTEM ERROR. File system is full, out of disk space, diskette is write protected, etc.

133 **Attribute not defined**
134 **Tag not defined**
135 **Invalid bitmap format**

Explanation: File is not a valid PM format bitmap file.

137 **Cannot execute a program**

Explanation: SYSTEM ERROR. IPF could not execute a required program. Program may be missing, corrupt, or other system problem may exist.

138 **Cannot rename file**

Explanation: SYSTEM ERROR.

140 **Invalid country code, or codepage**
141 **Invalid language code**
142 **Cannot determine current working directory**

Explanation: SYSTEM ERROR.

143 **No valid COLS specification was given**
144 **Ignoring invalid tag in table cell**
145 **Ignoring text before :c tag**
146 **Extra cells will be placed in next table row**
147 **Missing ELINK tag inserted at end of table cell**
148 **Total table width exceeds limit of 250 characters**
149 **Cannot reopen. File is already opened**

Explanation: SYSTEM ERROR.

150 **Document has no vocabulary**
151 **No res for this reference**
152 **Duplicate tag in source file**

Warning Level 2 Messages

201 **Invalid tag**
202 **Invalid attribute**
203 **Invalid symbol**

Explanation: Invalid APS symbol; period missing after the APS symbol, symbol specified is not in the APSYMBOL.APS file, invalid APSYMBOL.APS file.

204 **Invalid macro**
205 **Text too long in tag**

Explanation: Heading and index tags have a maximum of 150 characters.

206 **Token is bigger than expected.**

Explanation: Maximum length of token is 255 characters. This error could be caused by a missing end period or quote character.

207 **Invalid attribute value**
208 **Missing tag**
209 **Attribute not matched**
210 **Text too long in macro expansion**

Explanation: Maximum 255 characters.

211 **Total number of fonts exceeds the limit of 14**
212 **Sub index cannot be global without global main index**

Warning Level 3 Messages

301 **Ignoring attribute**
302 **Duplicate ID**

Explanation: Cannot specify the same ID in the same panel or index.

303 **Duplicate symbol in symbol file**
304 **Duplicate res number**
305 **Parent panel cannot have its own text**

Index

S10G-6262-00

P10G6262