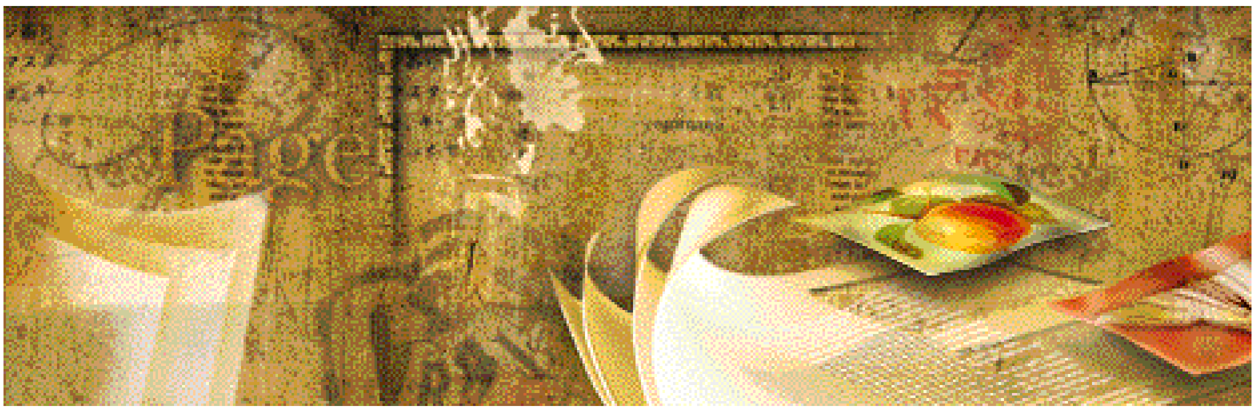
Adobe

Adobe PageMaker™ version 6.5



Software Development Kit

for Macintosh and Windows

Revised: 11 July 1996

Adobe PageMaker 6.5 Software Development Kit

Most of the material for this document was derived from work by xxx, xxx, xxx, and xxx. It was then compiled, edited, and reformatted into its current form by Brian Andrews.

Version History

1 July 1996	Brian Andrews	Version 6.5 – Preliminary release
11 July 1996	Brian Andrews	Version 6.5 – Final 6.5 release

PageMaker Plug-Ins Overview

Welcome to the PageMaker Plug-ins Software Development Kit. This kit provides documentation and sample code to help you write plug-ins for PageMaker 6.0.

The PageMaker plug-ins mechanism enables third-party developers and users to add to the capabilities of Adobe PageMaker. Using a command and query language to communicate with PageMaker, plug-ins can perform any action a user might do with the keyboard or mouse, extract information about the content and structure of publications, and more.

This chapter provides an overview of PageMaker plug-ins. It briefly describes the two types of plug-ins, what plug-ins can and cannot do, and how plug-ins interact with PageMaker.

What is a plug-in?

The plug-ins mechanism lets loadable modules or external applications control PageMaker and automate and extend PageMaker’s existing capabilities.

Plug-ins can take many forms. The two types of plug-ins are:

- Loadable plug-ins, which PageMaker loads dynamically and which appear on the PageMaker Plug-ins submenu.
- Stand-alone plug-ins, which communicate with PageMaker using either Apple Events or Windows Dynamic Data Exchange–DDE.

What can plug-ins do?

The range of possible plug-ins is vast. Plug-ins can:

- Provide new features, add special functions, or enhance existing functionality, such as equation editing, generating running headers and footers, or aligning columns.
- Automate common tasks, such as creating drop caps or converting typed fractions into typographer’s fractions.
- Streamline access to PageMaker’s existing functions, such as adding a palette or toolbar of certain PageMaker operations.
- Automate the production of boilerplate publications, such as updating a monthly report or converting existing publications to a new format.
- Improve the transfer of information between PageMaker and other applications, such as automatically extracting and formatting information from a database into a PageMaker publication, or creating an Adobe Persuasion presentation from a report.
- Organize and manage publications, linked files, and fonts, such as copying and compressing all files and fonts needed by a publication onto a single disk.

What plug-ins can’t do

While the plug-ins mechanism offers more than simple remote control of PageMaker, it does not provide a way to make PageMaker do something it doesn’t already do. You cannot create new tools for the toolbox or new kinds of page elements; nor can you change built-in components, such as the PageMaker composition algorithm, line and fill styles, or the way PageMaker draws to the screen and the printer. However, using existing PageMaker commands and the information gathered with queries, you can provide valuable, capability enhancing plug-ins for PageMaker users.

Who will use plug-ins?

Plug-ins are attractive to all sectors of the desk-top publishing market—both Macintosh and PC, high- and low-end. Numerous plug-ins are appropriate for a broad range of users, while others fulfill specific market needs in such areas as graphic arts, business communication, technical and scientific research, academia, training, multi-user environments, and, of course, the various types of publishing (magazine, newspaper, database, scientific, government, and catalog, among others).

What you need to develop a plug-in

The requirements vary depending on the type of plug-in you develop. However, in all cases, you need a solid understanding of how to use PageMaker and should be familiar with the C language.

Macintosh developers need

- Adobe PageMaker 6.0 for the Macintosh and Power Macintosh.
- Macintosh System 7.1 (or later).
- Metrowerks CodeWarrior (version CW7 or later).

Windows developers need

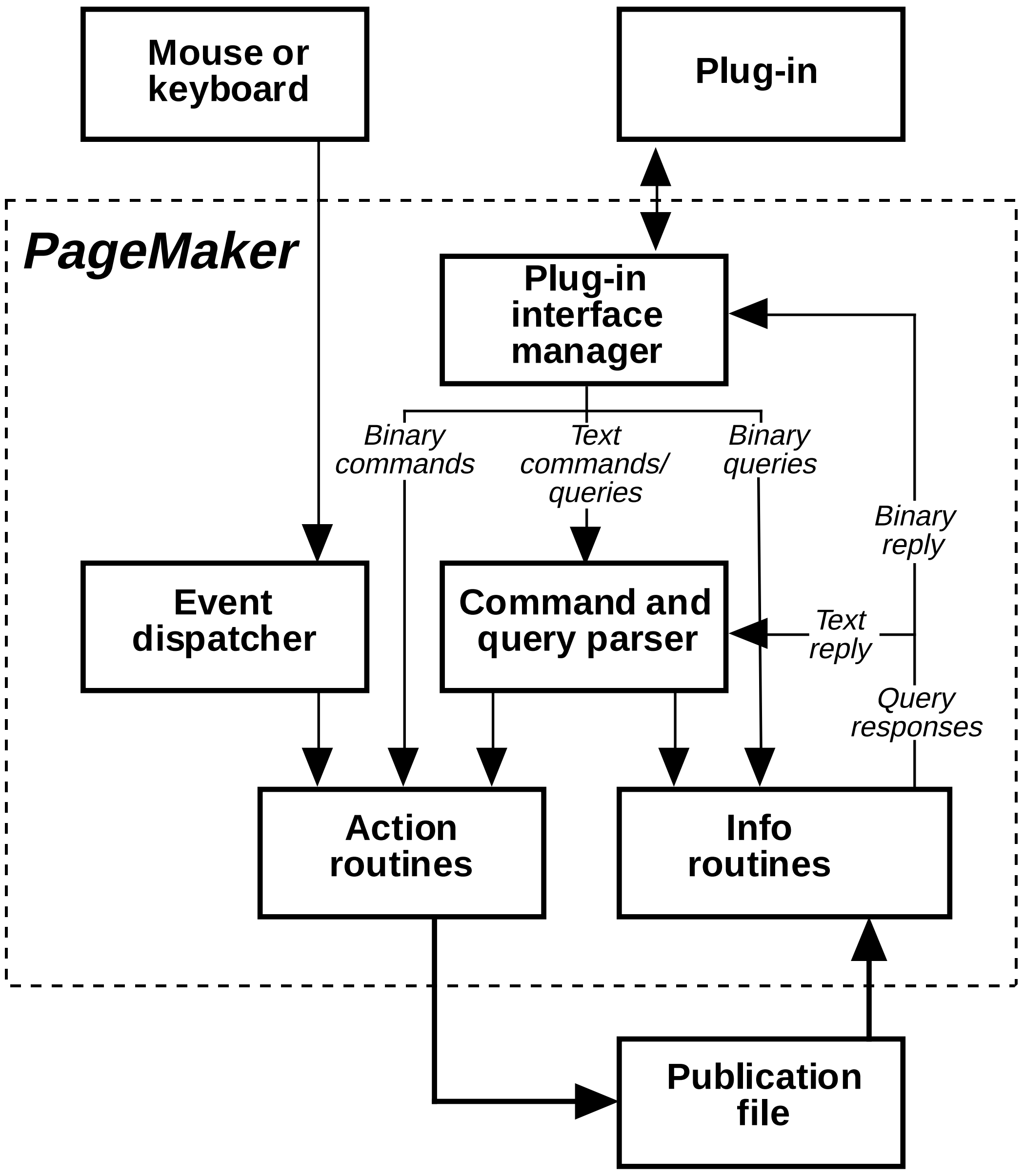
- Adobe PageMaker 6.0 for Windows
- Microsoft Windows 95
- Microsoft Visual C++, version 2.0 or later, and the SDK Guide for Microsoft Windows and Windows NT

Documentation, testing, distributing, and supporting

Each developer is responsible for documenting, testing, distributing, and supporting their plug-ins. Adobe does not have a program for providing these services.

How do plug-ins interact with PageMaker?

All plug-ins use the command and query language to control PageMaker. As indicated by the following diagram, a plug-in communicates with PageMaker through the plug-ins interface manager. Regardless of the type of plug-in—be it a separate application (using Dynamic Data Exchange or Apple Events) or a loadable module—all communication occurs through this manager.



The interface manager directs commands and queries to either the parser, or if in the binary format (more about this later), directly to the appropriate action or information routines. As shown, the action routines are the same routines PageMaker uses when you use the mouse or keyboard.

Binary query responses are sent back to the interface manager, which in turn sends the responses to the plug-in. Text query responses are sent through the parser before being sent to the interface manager.

How complex is the PageMaker plug-ins mechanism?

No knowledge of the PageMaker internals needed

You do not need to know anything about the PageMaker file format, subroutine libraries, or memory data structures to develop a plug-in. Because commands and queries are sent directly to PageMaker's existing action and information routines, PageMaker, not the plug-in, is responsible for the integrity of publications. In this way, you can focus your efforts on developing and testing your plug-in, not on preventing negative side effects your plug-in might have on a PageMaker publication.

PageMaker-based commands

The command and query language is straightforward and simple, based on the actual menu commands or mouse actions (open, paste, getfont, for example). The parameters and coordinates that accompany a command or query mimic the way a user works on a publication (column 1 top, guide 1, for example).

PC or Mac: The language is the same

With few exceptions, the command and query language is identical in the Windows and Macintosh environments, simplifying the development of plug-ins aimed for both platforms.

Command and query language

The command and query language is the key to the plug-ins interface:

Commands :	Invoke PageMaker menu commands and perform most user actions (for example, drawing, moving, or resizing objects; setting guides; and inserting text).
Queries:	Access information in PageMaker publications (for example, details about individual elements and their placement, the number of pages, the page size, the fonts used, and so on).

Two formats: text and binary

You can use two formats for commands or queries: ASCII text or binary (non-ASCII bindings). As shown in the previous diagram, the interface manager sends commands and queries in the text format through the parser, which in turn converts them into the binary format. Therefore, commands and queries in the binary format require less memory to process, and execute faster than the equivalent commands or queries in the text format.

Parameters

Many menu commands access dialog boxes, which solicit additional information from the user. Where appropriate, plug-ins commands require you to include this information in parameters and thereby bypass the dialog box. For example, to open the file income.pm5, the command is open "income.pm5".

To avoid long parameter lists for dialog boxes that contain numerous entries and choices, certain commands have been broken into several commands. For example, four script-language commands cover the File > Document Setup menu command: PageMargins, PageNumbers, PageOptions, and PageSize.

Defaults:	You must supply all parameters with a command unless specifically stated otherwise. The parser does not accept a command or query without the required parameters.

Coordinates

PageMaker accepts various types of page coordinates. You can specify a location on a page using precise numeric values, or you can specify a location in terms of column and ruler guides or placed elements (such as the bottom of a text block or the edge of a graphic)—the way a user works and thinks. For example, to place a logo at the intersection of the left edge of a column and the third horizontal guide, you could specify the x and y coordinates as column 2 left, guide 3. Or, to begin a story immediately following the last-drawn (top- or front most) object, you could specify the coordinates last left, last bottom.

Multiple commands but single queries

Multiple commands:	A plug-in may send multiple commands in the text format to PageMaker by separating each with a semicolon (;), carriage return, or carriage return/line feed pair (ASCII codes 0A or 0D).

Using the binary format, only one command or query may be sent at a time. However, the SDK includes macros to give these binary commands and queries a direct function-call feel.

Single queries: You should send only one query at a time. Although PageMaker accepts multiple queries, you'll receive a reply for the last query only, because each response overwrites the previous response.

Types of plug-ins

Plug-ins fall into two categories:

- Loadable plug-ins
- Stand-alone plug-ins

Loadable plug-ins

PageMaker automatically loads two types of plug-ins:

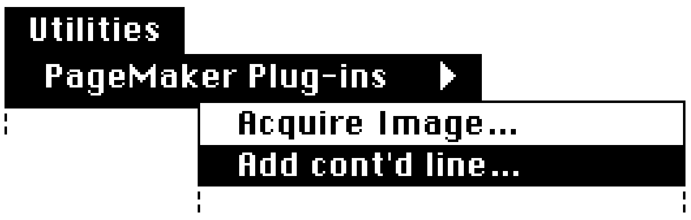
Menu plug-ins:	Appear on PageMaker’s plug-ins submenu (a hierarchical menu on the Utilities menu).
Function libraries:	Are invisible to the user but provide functions and utilities for other plug-ins.

You write loadable plug-ins in C, and compile and link the source files into a 68K-Windows DLL, a 32-bit Macintosh code resource, or a Power Macintosh shared library. Although you write these types of plug-ins in C, the plug-in uses the command and query language to communicate with PageMaker.

With few exceptions, the command and query language is identical in PageMaker for Windows and for the Macintosh. Therefore, you can develop a plug-in that can be marketed in both environments. By avoiding environment-specific language and compiler features and by conditionally isolating platform-dependent code, your plug-in source code can have a fair degree of portability.

Menu plug-ins

PageMaker lists menu plug-ins on the PageMaker Plug-ins submenu. A menu plug-in appears to users to be part of PageMaker and is invoked when a user selects it from the plug-ins submenu.



Function Libraries

PageMaker automatically loads plug-in function libraries and makes them available to other plug-ins, although the libraries remain hidden from the user. Function libraries can provide a variety of utilities, which can be used by any other plug-in.

Stand-alone plug-ins

Other applications can access PageMaker via DDE or Apple Events. Using the same language as loadable plug-ins, an application can send PageMaker commands and request information about a publication.

There are many potential uses for this kind of remote access of PageMaker, and using DDE or Apple Events, a stand-alone application can integrate the functionality of multiple applications. For example, you can write a stand-alone application that uses Apple events to open a database program and extract information, transfer the information to a graphics program, format the data into a chart, and then open PageMaker and place the chart into a publication. Or an application could automatically generate personalized letters from a database of addresses.

DDE

In Windows, an application initiates contact with PageMaker using DDE messages. In DDE parlance, PageMaker is the server and the application is the client. The DDE messages EXECUTE, REQUEST, and DATA establish contact between the applications and transmit the commands, queries, and responses. The application sends commands and parameters in EXECUTE messages. REQUEST messages carry queries, and PageMaker replies to a REQUEST using a DATA message.

Macintosh System 7 Apple Events

The Macintosh protocol using Apple Events is very similar to the protocol used on the PC. Rather than use a separate Apple Event ID for each command and query, a plug-in uses one of two Apple Event IDs to deliver commands or queries to PageMaker: DOSC and EVAL. DOSC can deliver multiple ASCII commands.

In addition, PageMaker supports the four “required” Apple Events: Open Application, Open File, Print, and Quit.

Where to go next

Use the following table to determine which chapters to refer to next:

To Write	Refer To
Loadable plug-ins	Chapter 2Writing Loadable Plug-Ins Chapter 3Required Routines for Loadable Plug-Ins Chapter 4Memory-Manager Routines Chapter 5Macros Chapter 6User Interface Design Guidelines Chapter 8Using Commands and Queries Chapter 9PageMaker Commands Chapter 10PageMaker Queries
Stand-alone plug-ins	Chapter 7Writing Stand-Alone Plug-Ins: Apple Events and DDE Chapter 8Using Commands and Queries Chapter 9PageMaker Commands Chapter 10PageMaker Queries

2 Writing Loadable Plug-Ins

This chapter discusses how to write loadable plug-ins. It describes the structure of a plug-in, details the flow of control between a plug-in and PageMaker, and presents implementation suggestions for writing both platform-specific plug-ins and portable plug-ins. What is a plug-in?

The plug-ins mechanism lets loadable modules or external applications control PageMaker and automate and extend PageMaker's existing capabilities.

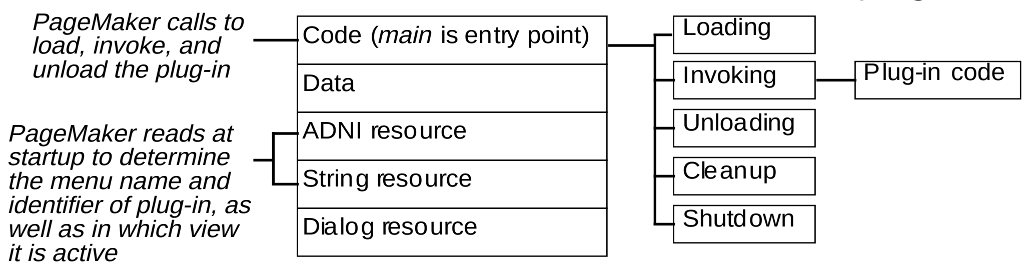
Structure of a plug-in

You write loadable plug-ins in C or C++ and compile and link them into either a 32-bit Windows DLL or a Macintosh fat-binary shared library. A plug-in must contain:

- A main routine, which serves as the single entry point for all calls from PageMaker to the plug-in.
- Registration and related string resources, which PageMaker reads at startup and uses to build the PageMaker Plug-ins submenu when first displayed.
- Five routines with operations for loading, invoking, unloading, cleanup, and shutdown (Chapter 3 describes these operations in more detail).
- Code for your plug-in.

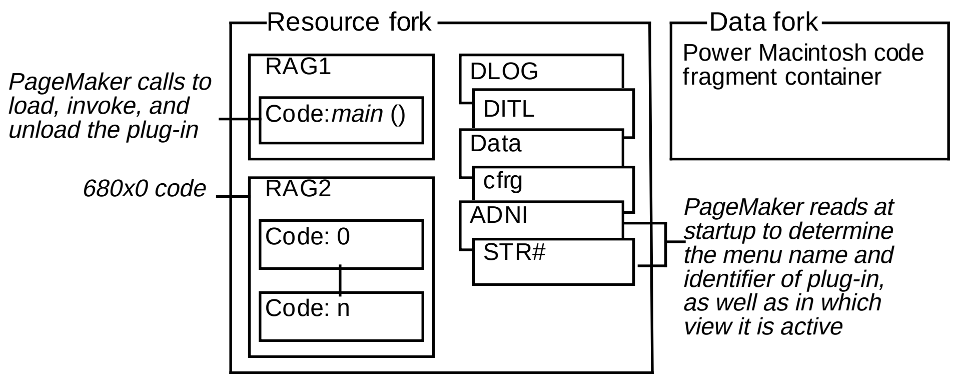
Windows DLL

The illustration below shows the basic structure of a plug-in DLL.

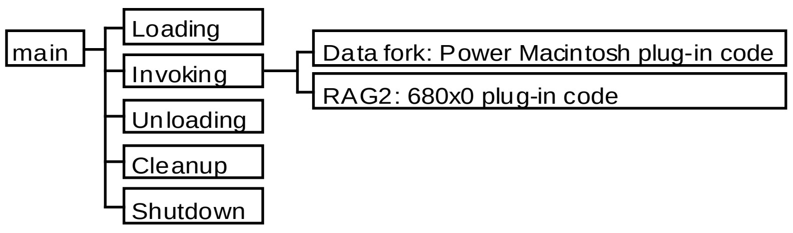

PageMaker calls to load, invoke, and unload the plug-in
PageMaker reads at startup to determine the menu name and identifier of plug-in, as well as in which view it is active
Code (main is entry point)
Data
ADNI resource
String resource
Dialog resource
Loading
Invoking
Unloading
Cleanup
Shutdown
Plug-in code

Macintosh shared library

A fat-binary shared library contains the code needed to run a plug-in in both PageMaker for the Macintosh and PageMaker for the Power Macintosh. As shown in the illustration below, the 68K code is in the RAG2 resource in the resource fork and the Power Macintosh code is in a code fragment container in the data fork.


Resource fork
Data fork
Power Macintosh code fragment container
PageMaker calls to load, invoke, and unload the plug-in
680x0 code
RAG1
Code:main ()
RAG2
Code: 0
Code: n
DLOG
DITL
Data
cfrg
ADNI
STR#
PageMaker reads at startup to determine the menu name and identifier of plug-in, as well as in which view it is active

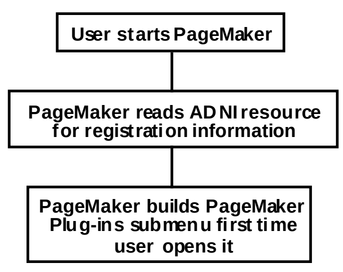
The main routine contains all the required operations. It determines the platform and jumps to the appropriate plug-in code.



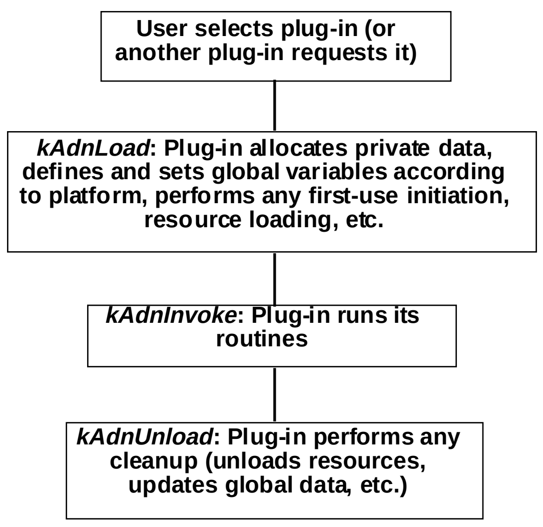
Flow of control

The following diagrams show the normal sequence of calls from PageMaker at startup, when a plug-in is invoked, when PageMaker is low on memory, and when PageMaker closes. Remember, main dispatches calls from PageMaker to the requested routine. The operation codes PageMaker sends during invocation, when low on memory, and prior to shutdown are included in the diagrams.

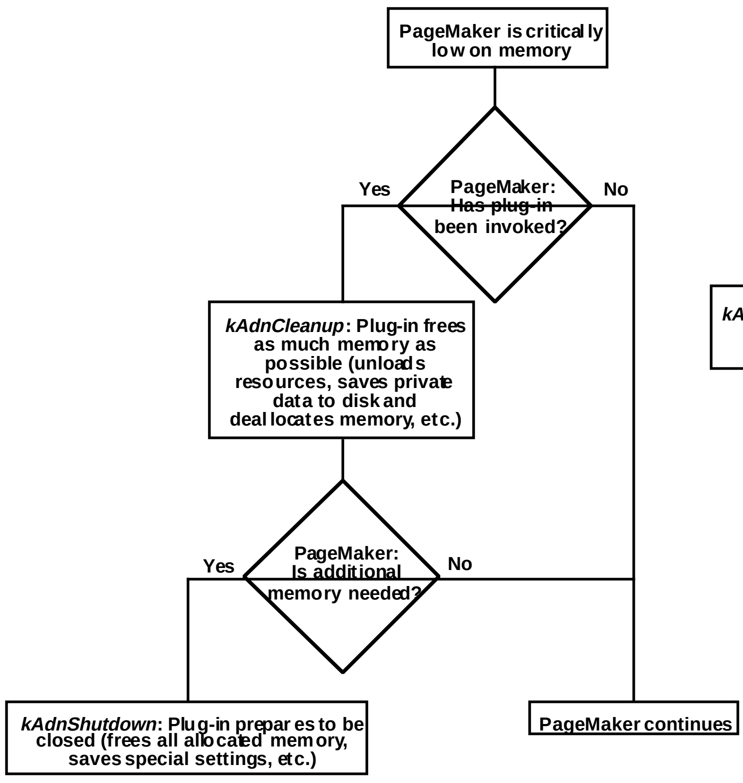
Startup


User starts PageMaker
PageMaker reads ADNI resource for registration information
PageMaker builds PageMaker Plug-ins submenu first time user opens it

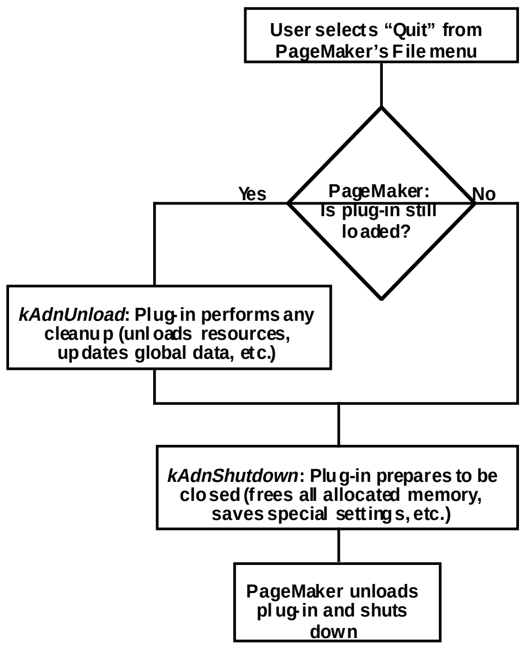
Invocation


User selects plug-in (or another plug-in requests it)
kAdnLoad: Plug-in allocates private data, defines and sets global variables according to platform, performs any first-use initiation, resource loading, etc.
kAdnInvoke: Plug-in runs its routines
kAdnUnload: Plug-in performs any cleanup (unloads resources, updates global data, etc.)

Low memory


PageMaker is critically low on memory
PageMaker: Has plug-in been invoked?
Yes
No
kAdnCleanup: Plug-in frees as much memory as possible (unloads resources, saves private data to disk and deallocates memory, etc.)
PageMaker: Is additional memory needed?
Yes
No
kAdnShutdown: Plug-in prepares to be closed (frees all allocated memory, saves special settings, etc.)
PageMaker continues

Shutdown


User selects "Quit" from PageMaker's File menu
PageMaker: Is plug-in still loaded?
Yes
No
kAdnUnload: Plug-in performs any cleanup (unloads resources, updates global data, etc.)
kAdnShutdown: Plug-in prepares to be closed (frees all allocated memory, saves special settings, etc.)
PageMaker unloads plug-in and shuts down

Data flow

A plug-in transfers data to or from PageMaker at various times: at startup, when invoked, and when sending commands or receiving query responses. To simplify development and reduce parameter traffic on the stack, a common parameter block structure is used for all data transfer. PageMaker initializes and allocates the parameter block and passes a pointer to the block every time PageMaker contacts the plug-in. During certain types of transfers, some fields may be unused or optional, while others may contain data or pointers to other data structures.

The parameter block and the parameters used in each transfer are described in detail in Chapter 3.

How PageMaker recognizes a plug-in

For PageMaker to recognize a loadable plug-in, the plug-in file must follow the guidelines noted below.

Macintosh

PageMaker for the Macintosh requires that a plug-in:

- Be a shared library with 680x0 code in the resource fork and with Power Macintosh code in the data fork.
- Have the filename extension .add (e.g., Plug_in.add).
- Be in the Plug-ins folder in this location:

  HD:PageMaker 6.0:RSRC:Plugins

  where HD is the drive on which you installed PageMaker and its files. Do not move the Plug-ins folder from this location.
- Have the main routine in a segment that has a RAG1 resource type.
- Have the main routine declared with the Pascal calling convention.
- Have required registration information in the ADNI resource (see “Registration,” which follows).

Windows

PageMaker for Windows requires that a plug-in:

- Be a DLL.
- Have the filename extension .add (e.g., Plug_in.add).
- Be in the Plug-ins folder in this location:

  C:\PM6\rsrc\<language>\plugins

  where C: is the drive on which you installed PageMaker and its files. Do not move the Plug-ins folder from this location.
- Have the main routine declared with the Pascal calling convention and be exported (either explicitly with the _export keyword or in the module definition file).
- Have required registration information in the ADNI resource (see “Registration,” which follows).

Registration

Registration information is stored in the ADNI resource. PageMaker reads the ADNI resource of each plug-in at start-up. Briefly, PageMaker needs to know:

- The name that should appear in the PageMaker Plug-ins submenu
- The plug-in internal ID
- Whether the plug-in should appear in the PageMaker Plug-ins submenu (a plug-in can be a library of functions used by other plug-ins and not appear in the menu)
- In what state the plug-in is active: in layout view, in story editor, and when no publications are open

PC: Creating the registration resource

The file, frame.rc, contains the definitions, declarations, and structures you need to create the ADNI resource. When PageMaker is started, it reads this data to register the plug-in.

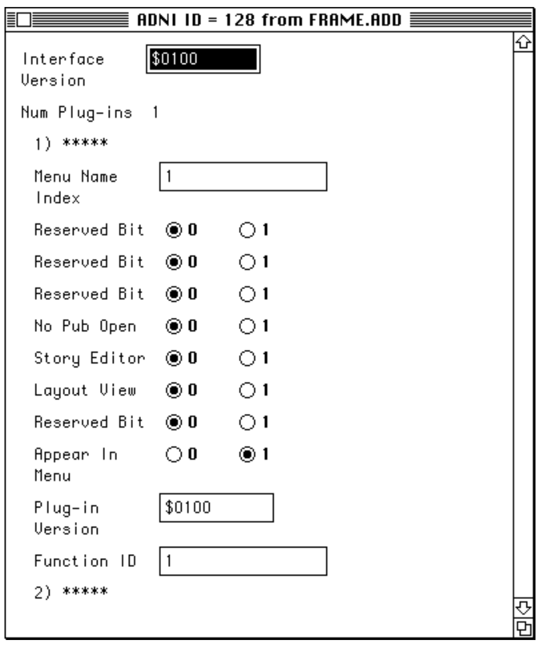
Note: When you open FRAME.RC in the MS Visual C++ Integrated Development Environment, change the Open As option in the Open dialog box from Auto to Text. This allows you to edit the registration resource more easily. Otherwise, it is translated into hex.

Example:

```
*frame.rc - Plug-in program initialization resource

*

************************************************/

// Codes for plug-in resources:

#define RI_REGINFO 101

#define ADNI 301


// Declaration of the RI_REGINFO ADNI resource.

RI_REGINFO ADNI MOVEABLE //101 301

// Header:

BEGIN

0L      // reserved (LONG)

0x0200  // Interface version, must be 2.0 for PM 6 (WORD)

1       // Num of plug-ins in DLL, we recommend 1 per DLL(SHORT)


// Definition of plug-in number 1:

1L      // String id# for menu name (DWORD)

0x1DL   // Mask: does plug-in appear in menu and in what state?

        // layout, story editor, when no publications are open

        // 0x00 for don't appear in submenu

        // 0x01 (plus any value below) for show on submenu

        // 0x04 for active on submenu in layout view

        // 0x08 for active on submenu in story editor

        // 0x10 for active on submenu when no publications are open
        // 0x1DL to appear in menu in all states
0x0203  // Plug-in version, for example 2.3 (WORD)
0       // reserved (WORD)
1L      // plug-in number (first plug-in in DLL is 1) (LONG)
0L      // reserved (LONG)
// Define any other plug-ins in same DLL here (not recommended)
END

// For each plug-in in DLL, include a string resource for its menu name
// The numbers of the string resource should correspond
// with the plug-in string id# numbers in the ADNI resource above
STRINGTABLE
BEGIN
1, "Plug-in Framework..."
// If you have more than one plug-in in DLL (not recommended),
// add more numbers and names here
END
```

Macintosh: Creating the registration resource with ResEdit

The ResEdit template, ADNI Template.rsrc, simplifies the creation of the ADNI resource.



ADNI ID = 128 from FRAME.ADD

Interface Version: $0100

Num Plug-ins: 1

1) *****

Menu Name Index: 1

Reserved Bit: [✓ 0] 1

Reserved Bit: [✓ 0] 1

Reserved Bit: [✓ 0] 1

No Pub Open: [✓ 0] 1

Story Editor: [✓ 0] 1

Layout View: [✓ 0] 1

Reserved Bit: [✓ 0] 1

Appear In Menu: 0 [✓ 1]

Plug-in Version: $0100

Function ID: 1

2) *****

To use this template

1. Start ResEdit (version 2.1 or later) and open ADNI Template.rsrc.
2. Copy the TMPL record.
3. Open the ResEdit Preferences file (located in System Folder:Preferences).
4. Paste the TMPL record into the ResEdit Preferences file.
5. Close both files.
6. Open the plug-in resource file.
7. Choose Create New Resource from the Resource menu and select ADNI in the Select New Type dialog box.
8. Enter $0200 for the interface version.

9 Select “1)*****” and choose Insert New Fields from the Resource menu.

10 Fill in the fields as described below:

Information	Description
Menu Name Index	The index number of the plug-in menu name as found in the string resource. You store the menu name in an indexed string resource (STR#).
No Pub Open	Whether or not the plug-in is active when no publications are open in PageMaker: 1 for active, 0 for disabled.
Story Editor	Whether or not the plug-in is active in story editor: 1 for active, 0 for disabled.
Layout View	Whether or not the plug-in is active in layout view of PageMaker: 1 for active, 0 for disabled.
Appear in Menu	Whether or not the plug-in appears in the PageMaker plug-ins submenu: 1 to appear in menu, 0 to not display in menu.
Plug-in Version	The version and revision number of the plug-in. Do not include a decimal point; PageMaker assumes a decimal point separates the numbers. For example, enter $0101 for version 1.01. (PageMaker lists all the installed plug-ins and their version numbers when you hold down either the Command or Control key while selecting “About PageMaker” from the Apple menu.)
Plug-in ID	The identifier PageMaker should use when calling the plug-in (e.g., for invocation or shutdown). The plug-in assigns the ID number.

Note: The IDs of ADNI and STR# must match.

Macintosh: Assigning resource type

The resource type of a Macintosh plug-in should be RAG1. Only the segment containing main should have this resource type. We recommend that the 680x0 code be in RAG2.

Single entry point: main

All calls from PageMaker to your plug-in go through the main() entry point. Typically, your code should dispatch the call based on the opcode field, which contains one of five values: kPMLoad, kPMInvoke, kPMUnload, kPMCleanup, or kPMShutdown. These opcodes are discussed in more detail in Chapter 3.

For Macintosh plug-ins, the main() routine must be 680x0 code in a RAG1 resource, regardless of whether PageMaker is running on a 680x0 or Power Macintosh system.

The sample code resource Main.RAG1.rsrc in the RAG1 Main folder provides a standard mechanism for handling both 680x0 and Power Macintosh native code execution. It loads either the RAG2 code resource (680x0) or the Power Macintosh code fragment and passes the parameter block through to the appropriate code. It also frees the code resources or code fragments when a kPMCleanup or kPMShutdown call is received from PageMaker. All the sample programs use this standard format for creating fat plug-ins.

Pascal calling convention

Main must be declared with the Pascal calling convention. In Windows, this function must be exported (either explicitly with the "_export" keyword or in the module definition file).

General guidelines

The following guidelines discuss displaying dialog boxes, handling errors, using global variables, and managing memory.

Dialog boxes or windows

PageMaker dialog boxes and alerts suppressed. When PageMaker passes control to the plug-in during invocation, and the plug-in begins sending commands to PageMaker, normal PageMaker dialog boxes, error messages, and alerts are not displayed. Command parameters (and sometimes supplemental commands) supply all the information normally provided in PageMaker dialog boxes. Therefore, when sending commands, a plug-in must both display its own dialog boxes if it needs the user to specify values for a command, as well as display its own error messages. (For more information about error conditions, see “Error and status codes,” which follows.)

Close your own. If your plug-in opens any dialog boxes or windows, make sure it closes and disposes of them before it returns from any invocation calls.

Plug-ins are modal. Therefore, on the Macintosh, floating, non-modal dialog boxes, palettes, and windows may not work properly from plug-ins. Plug-ins on the PC, however, do not have this restriction because event dispatching is handled differently on this platform.

Error and status codes

Error and status codes are defined in PageMakerCQErrs.h. PageMaker and plug-ins return these codes to report error conditions or the status (success) of an operation. What displays an error message, PageMaker or the plug-in, depends upon when an error occurs:

- If the plug-in is controlling PageMaker (sending it commands and queries), it is up to the plug-in to display an error message to the user.
- If a plug-in has returned control to PageMaker, PageMaker displays an error message if necessary.

Errors from PageMaker while executing commands and queries

To determine if an error has occurred while PageMaker executes a command or query, a plug-in should check the return code of the operation sending the command or query. If the code indicates that the command or query failed, the plug-in can query PageMaker for the error code or string. (See “GetLastErrorStr” or “GetLastError” in Chapter 10.)

Returning errors to PageMaker

Upon returning from loading, invoking, unloading, cleanup, or shutdown, the plug-in can return a non-zero error code and pass a handle to an error string in the hszErrMessage field of the parameter block. PageMaker displays plug-in errors during these operations.

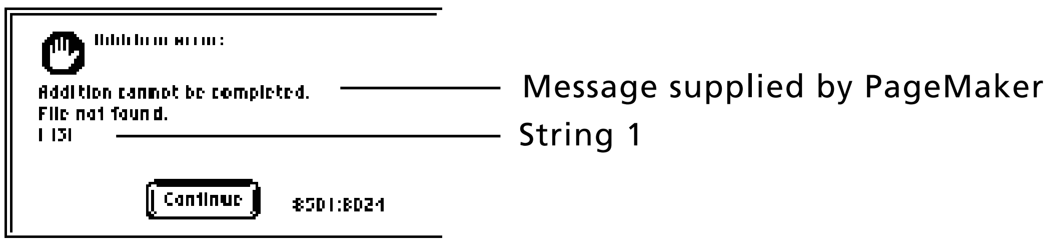
Format of the error string. The plug-in can optionally supply two messages. They must be separated by a null character. Regardless of whether the error string contains one or two messages, it must end with two null characters. For example:

```
<String1><NULL><NULL>
```

or

```
    <String1><NULL><String2><NULL><NULL>
```

If the plug-in supplies one error message, PageMaker displays “Plug-in error: Plug-in cannot be completed” followed by the plug-in message. For example:



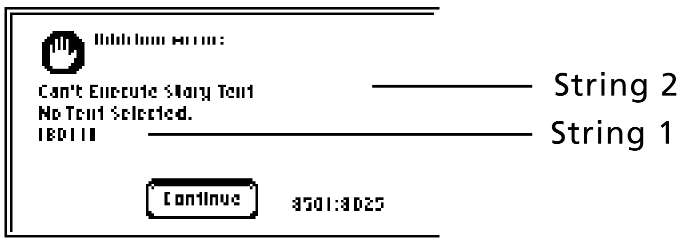
[illegible]:

Addition cannot be completed.
File not found.
1 131

Continue

8501:8024

If the plug-in supplies two messages (as indicated by the null character separating them), PageMaker displays them in reverse order, substituting the generic “Plug-in cannot be completed” with the second message. That is, PageMaker displays “Plug-in error:”, the second message (<String2>), followed by the first message (<String1>). For example:



[illegible]:

Can't Execute Story Text
No Text Selected.
IBDIII

Continue

8501:8025

Global variables

Global variables are not persistent; PageMaker generally unloads a plug-in after each invocation. For data a plug-in needs to maintain between invocations, it should use a private memory block, which the plug-in allocates when first loaded and deallocates when PageMaker shuts down. (See the description of pluginData in “Loading” and “Shutting down” in Chapter 3.)

Memory management

PageMaker and plug-ins share the same stack. Keep this in mind when using local variables or nested procedures. Because the stack is shared by PageMaker and a plug-in, executing a plug-in from within a plug-in further reduces the available space on the stack.

A plug-in should allocate only the memory it needs to avoid unnecessary purging of PageMaker code segments from the heap while the plug-in is running.

Binary versus text format for commands and queries

Sending plug-in commands and queries in the binary format (and receiving query replies as binary data, the default) uses less stack space than sending them as text. PageMaker must parse commands and queries sent in the text format into binary packets, whereas commands and queries in the binary format bypass the parser, thus eliminating the need for PageMaker to load the parser.

Palettes and windows

To maximize the memory available to the plug-in, close all PageMaker palettes and windows that aren’t needed by the plug-in. Be sure to return PageMaker to its previous state when the plug-in is through running.

How a user can maximize memory

A user can maximize the available memory for a plug-in by:

- (Macintosh only) Allocating more memory to PageMaker using the Get Info dialog box.
- (Macintosh only) Minimizing the size of the disk cache.
- (PC only) Closing other applications in Windows.

Cleanup

It is critical that a plug-in clean up any memory it allocates and any resources it loads before closing. PageMaker does not check for memory left behind by a plug-in, and such memory can quickly consume available heap space.

A plug-in is closed after the plug-in responds to the PageMaker kPMUnload message. On the Macintosh, plug-in code and resources are flushed from the heap at that time. Any heap objects allocated are placed in the PageMaker heap and must be flushed by the plug-in.

Under Windows, the plug-in DLL uses PageMaker stack space but may have its own local heap as well. The plug-in must clean up any objects allocated in the local and global heap, including dialog boxes, palettes, windows, or other system resources.

Adobe Memory Manager

Adobe has supplied a library of optional routines for memory management. These routines are optimized for use by PageMaker and work in both the Windows and Macintosh environments. Using them can help you write dual-platform plug-in code more quickly. In particular, we strongly recommend that you use the Adobe Memory-Manager routines MMAlloc and MMFree, instead of the standard C routines malloc and free. The Adobe Memory-Manager routines are described in Chapter 4, “Memory-Manager routines.”

Writing portable plug-ins

A portable plug-in is a plug-in that you need only recompile to use in another environment. Because the plug-in interface and the command and query language are nearly identical in both Windows and on the Macintosh, it is possible to write a plug-in that you can compile in both environments.

Portable code

You write portable code to be independent from its environment. You must make no assumptions about the compiler, the processor, or the byte order:

- Avoid using inherently non-portable language features.
- Keep compiler dependencies out of the source code as much as possible. Hide these dependencies in a header file.
- Conditionally compile or isolate platform-specific code. For clarity, clearly label any platform-dependent code, for example:

```
#ifdef WINDOWS

…

#endif /* WINDOWS */


#ifdef MACINTOSH

…

#endif /* MACINTOSH */
```

- Use the Adobe Memory-Manager routines and Adobe versions of the C run-time library routines when possible. These routines help insulate code from environment dependencies and help reduce the size of your plug-in. (See Chapter 4, “Memory-Manager routines,” for more information.)
- Keep processor differences in mind and implement byte-order swapping. (See “Motorola and Intel processor differences,” which follows.)

Examples: For examples of portable code, see Frame.C, included on disk in this SDK.

Motorola and Intel processor differences

To make software run on both the Motorola and Intel families of microprocessors, you must be aware of some of the differences between the processor families.

Byte order: One major difference is that the order of bytes in a word, and of words in a long word, is different between the processor families. Although the bit order in a byte is the same, the high-order byte comes first in the 680x0 and PowerPC (Motorola) family, and the low-order byte comes first in the 80x86 family (Intel).

Floats: Be aware that floating-point values not only have a different byte order but also have different formats that may be compiler-dependent.

C-bit field types: The order of bits in C-bit field types is both platform- and compiler-specific. In plug-in, some compilers will automatically pad a group of bit fields to a word boundary, while others will stop at a byte boundary. For portability, pad all bit field groups out to a word boundary.

Binary files:	Because of the difference in byte order between the processor families, binary files that contain SHORTs and LONGs are not compatible when transported between the Macintosh and the PC. Code that directly reads binary files must be able to process reversed byte order. (SHORT and LONG are data types defined in aldtypes.h.)

Macintosh development specifics

FAT binary: One executable, two platforms

PageMaker plug-ins for the Macintosh are fat-binary shared libraries that can run in both PageMaker for the Macintosh and PageMaker for the Power Macintosh in native mode. While a fat plug-in can be larger than a single platform plug-in, it lets you distribute and maintain one file for all Macintosh customers.

Use the sample code in the Frame folder on disk as the framework for your plug-in. As illustrated at the beginning of this chapter, a fat plug-in consists of three basic parts. The Frame sample divides the plug-in code into three projects, which correspond to the three parts of a plug-in:

- Main.µ , which creates RAG1, a 68K code resource. RAG1 contains only the main function and serves as the entry point to the plug-in from PageMaker. (Main.µ is in the RAG1 Main folder.)
- Frame.68k.µ, which creates RAG2, a 68K code resource. RAG2 contains the 68K executable code for the plug-in.
- Frame.PPC.µ, which creates a code fragment container in the data fork of the file. The code fragment container holds the native Power Macintosh executable code of the plug-in.

To build the final plug-in:

1. Make the 68k project. This creates a resource file containing the RAG1 and RAG2 resources as well as any plug-in specific resources.
2. Make the PPC project. This builds the fat plug-in.

When PageMaker loads the plug-in, it uses main as its entry point. The loading section of main determines the version, 680x0 or PowerPC, of both PageMaker and the computer. The invocation section of main jumps to either the 680x0 code in RAG2 (if PageMaker for the Macintosh is running), or the Power Macintosh code in the data fork (if PageMaker for the Power Macintosh is running).

Macintosh initialization calls

Because a plug-in is loaded by PageMaker, it need not make most Macintosh Toolbox initialization calls and should not make the following calls:

InitDialogs	InitResources	SetApplLimit
InitFonts	InitWindows	SetGrowZone
InitGraf	MaxApplZone	TEInit
InitMenus	RsrcZoneInit	

We also recommend you do not patch any traps (especially _LoadSeg, _UnloadSeg, or the Memory-Manager routines).

Do not use the C routine atexit( )

Do not call the C routine atexit( ) or use routines such as the file calls fopen, __open, and freopen that call atexit( ) indirectly. Although you can link the code segments, PageMaker will crash upon exiting if the plug-in has been loaded anytime during the session. The atexit( ) routine installs an exit-handler, but since the plug-in code has already been unloaded when PageMaker exits, the routine no longer exists.

Debugging

Refer to the documentation included with Metrowerks CodeWarrior for information on debugging shared libraries.

Editing the ‘vers’ resource

We strongly recommend that all plug-in libraries include a ‘vers’ resource that contains the library version and copyright information. This information is displayed when you select the plug-in file at the desktop and choose “Get Info...” from the Apple menu.

Windows development specifics

Microsoft Windows restrictions

On the PC, a plug-ins library is a standard Windows DLL and therefore must follow all the guidelines outlined for DLLs in the SDK Guide for Microsoft Windows and Windows NT.

Warning level 2 (Microsoft Visual C++ only). We highly recommend that you compile your code using at least warning level 2 (/W2) of the Microsoft Visual C++ compiler. Warning level 2 performs type checking on function arguments if you provide prototypes with argument types.

PC module-definition file

Every plug-in DLL compiled with Microsoft Visual C++, version 2, must have a module-definition file. The module-definition file is a text file that defines the contents and system requirements of a Windows DLL.

The module-definition file for a plug-in generally contains:

- A LIBRARY statement that gives the name of the plug-in module.
- A DESCRIPTION statement that describes the plug-in.
- A HEAPSIZE statement that defines the initial size of the plug-in's heap in bytes.
- A CODE statement that defines the memory attributes of the plug-in's code segments. This should be set to "MOVEABLE," "DISCARDABLE," and "LOADONCALL."
- A DATA statement that defines the memory attributes of the plug-in's data segment. This should be set to "MOVEABLE," "SINGLE," and "PRELOAD."
- An EXPORTS statement that defines the names of each function to be exported to PageMaker (if they aren't already marked for export in the source file).

Note: List the function names only; do not attach ordinal numbers. If you include an ordinal number with the function name, Windows may not keep the function name resident. The function names must be resident, since PageMaker calls the functions by name.

Also note: Both main and the Windows entry procedure, DllMain, must be exported functions (either explicitly with the _export keyword or in the module definition file).

The module-definition file may have any name but must end with a .def file extension. The following example shows a typical module-definition file for a plug-in (in this case, for runscrip.add).

```
LIBRARY RUNSCRP

DESCRIPTION 'Run script plug-in'

HEAPSIZE8192

CODE MOVEABLE DISCARDABLE LOADONCALL

DATA MOVEABLE SINGLE PRELOAD

EXPORTS

    main

    DllMain
```

For detailed information on the module-definition file, see the section on creating module-definition files in the SDK Guide for Microsoft Windows and Windows NT

Project files

Project files for the sample plug-ins are included on disk in this SDK. Refer to these when you create your project file.

Sample plug-in routine

The following plug-in routines illustrate how you can mix queries and commands to solve a problem. Although the routine is not complete enough for real use, it will give you a sense of the language. A more extensive sample plug-in is included on disk.

```
PMErr View100(sPMParamBlockPtr lpParamBlk)
{
charbuff[100];
PBBinCommandByShortValue(lpParamBlk, pm_view, 100);
return CQ_SUCCESS;
}
PMErr RunStory(sPMParamBlockPtr lpParamBlk)
{
RC      rcVal = CQ_SUCCESS;
USHORTwStyle;
HANDLEhStory;
LPSTRlpStory;
SHORTarParms[2];    // parameters for query.
/* set up request and reply units to be the same.
*/
PBSetRequestUnits(lpParamBlk,kMUInches);
PBSetReplyUnits(lpParamBlk,kMUInches);
/* set up specifics for getStoryText query.
*/
arParms[0] = 0;
arParms[1] = 3;
if  (!(rcVal =PBBinQueryWithParms(lpParamBlk,pm_getstorytext,kRSPointer,
    arParms,sizeof(arParms),kRSHandle,NULL,MAXSTORYSIZE)) &&
        (hStory = PBGetReplyData(lpParamBlk)))
    {
        PBTextCommand(lpParamBlk, kRSHandle, (LPSTR)hStory);
    } else {
        /*
        **  So something else, no selection.
        */
    }
return (rcVal);
}
```

Required Routines for Loadable Plug-Ins

3

This chapter discusses the routines PageMaker requires of loadable plug-ins. It details the flow of data between a plug-in and PageMaker through the parameter block.

Data flow

A plug-in transfers data to or from PageMaker at various times: at startup, when invoked, when sending commands or receiving query responses, and at shutdown. To simplify development and reduce parameter traffic on the stack, a common parameter block structure is used for all data transfer. During certain types of transfers, some fields may be unused or optional, while others may contain data or pointers to other data structures.

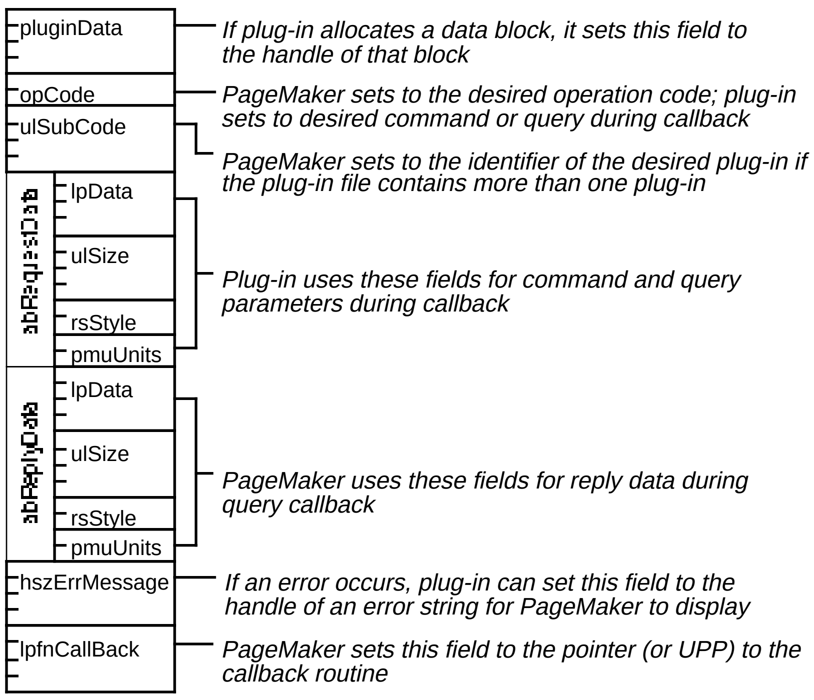
Allocated by PageMaker

PageMaker initializes and allocates the parameter block. Because PageMaker may use a different memory block with each invocation, the plug-in is passed a pointer to the parameter block every time PageMaker contacts the library.

Parameter block structure

The parameter block includes two argument blocks: one for plug-in parameters or parameters to PageMaker commands, and another for PageMaker replies to queries or functions.

The following diagram depicts the parameter block structure. Versions of this diagram accompany the description of each routine. The fields used in the routine are highlighted.


pluginData
If plug-in allocates a data block, it sets this field to the handle of that block
opCode
PageMaker sets to the desired operation code; plug-in sets to desired command or query during callback
ulSubCode
PageMaker sets to the identifier of the desired plug-in if the plug-in file contains more than one plug-in
pbRequestData
lpData
ulSize
rsStyle
pmuUnits
Plug-in uses these fields for command and query parameters during callback
pbReplyData
lpData
ulSize
rsStyle
pmuUnits
PageMaker uses these fields for reply data during query callback
hszErrMessage
If an error occurs, plug-in can set this field to the handle of an error string for PageMaker to display
lpfnCallBack
PageMaker sets this field to the pointer (or UPP) to the callback routine

Type definition for the parameter block

As specified in PageMakerTypes.h, here are the type definitions for the parameter block:

```
struct sPMParamBlock;  //so you can typedef the callback function ptr
#if defined(MACINTOSH)
    typedef pascal short (*fnPMCallbackProc)(struct sPMParamBlock *);
#else
    typedef short (pascal * fnPMCallbackProc)(struct sPMParamBlock *);
#endif
typedef struct sPMDataBlock
{
```

```
void *    pmData;  //A handle, pointer, or direct value
    unsigned long  pmDataSize;  //Size of handle or pointer
    unsigned short  pmStyle;  //Reference style
    unsigned short  pmUnits;  //For requesting reply units on text queries
} sPMDataBlock, * sPMDataBlockPtr;
typedef struct sPMParamBlock
{
void *    magicID;  // Reserved: plug-in should not touch this field
void *    pluginData;  //For storing plug-in data between invocations
    unsigned short  opCode;    // Used only in main() call from PageMaker
    unsigned long  subCode;  // For plug-in
    sPMDataBlock  requestData;
    sPMDataBlock  replyData;
    Handle    errMessage;  // Error text from plug-in or PM
    fnPMCallbackProc pmCallback;  // Callback function pointer
} sPMParamBlock, * sPMParamBlockPtr;
```

Inline macros

The PageMakerTypes.h file includes macro definitions to move data into and out of the correct fields of the parameter block and in and out of data buffers. Although we do not require that you use these macros, we recommend that you do. The macros should:

- Speed your development time, especially if you are writing a portable plug-in. (The macros can be used in both the PC and Macintosh environments.)
- Make your plug-ins easier to update. (Future versions of the plug-ins mechanism may use functions of the same names.)
- Reduce the potential for errors. (The macros have been tested and perform valuable tasks, such as clearing the parameter block when appropriate.)

These macros are described in Chapter 5.

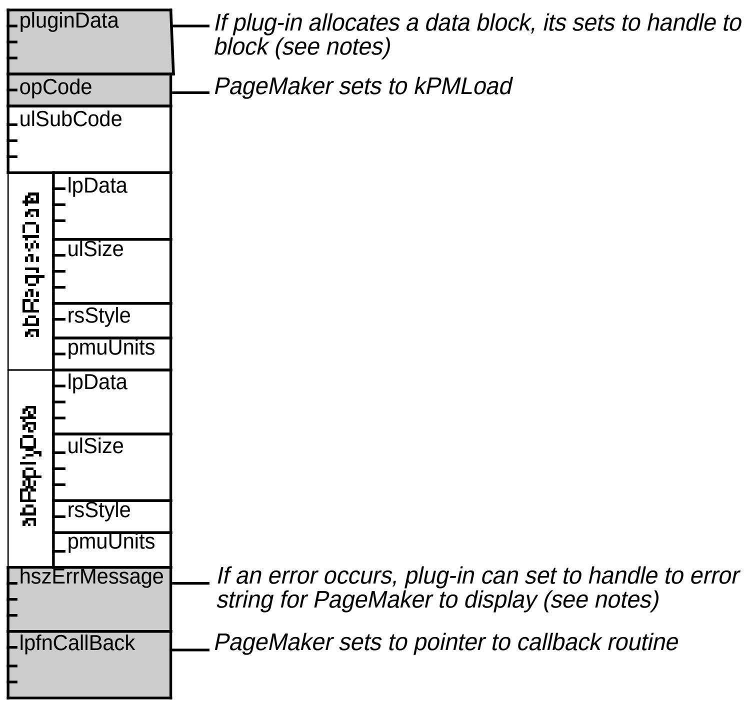
Loading

A plug-in loading routine performs any initialization needed for its first invocation and allocates memory for data (if needed).

Macintosh: The RAG1 main routine loads either the code fragment (Power Macintosh) or code resource (680x0). On the Power Macintosh, it also stores the connection ID and entry point for subsequent invocations.

Op code: kPMLoad: PageMaker sends kPMLoad when the user selects a plug-in from the PageMaker Plug-ins submenu (or from a script) or when another plug-in requests the plug-in. If the library is already open, PageMaker sends kPMInvoke instead.

Return code: The plug-in should return CQ_SUCCESS or a non-zero return code (as defined in PageMakerCQErrs.h).


pluginData
If plug-in allocates a data block, its sets to handle to block (see notes)
opCode
PageMaker sets to kPMLoad
ulSubCode
abRequestData
lpData
ulSize
rsStyle
pmuUnits
abReplyData
lpData
ulSize
rsStyle
pmuUnits
hszErrMessage
If an error occurs, plug-in can set to handle to error string for PageMaker to display (see notes)
lpfnCallBack
PageMaker sets to pointer to callback routine

pluginData: If the plug-in needs a persistent data block, it should allocate it during loading and set this field to the handle to the data block. PageMaker supplies this handle back to the plug-in on subsequent calls. Therefore, the plug-in should make sure the field is null before setting it.

hszErrMessage: If the plug-in returns a non-zero return code, it can set this to a handle to a string that describes why the requested operation failed. The string must end in two null characters, but it may consist of two parts separated by a null character. (See “Error and status codes” in Chapter 2 for a description of the format of this string.) If the plug-in fails for any reason during loading, use this string to describe the problem to the user.

If loading fails: If loading fails, PageMaker calls the unloading routine.

Example

```
PMErr kPMLoad(sPMParamBlockPtr lpParamBlk)
    {
```

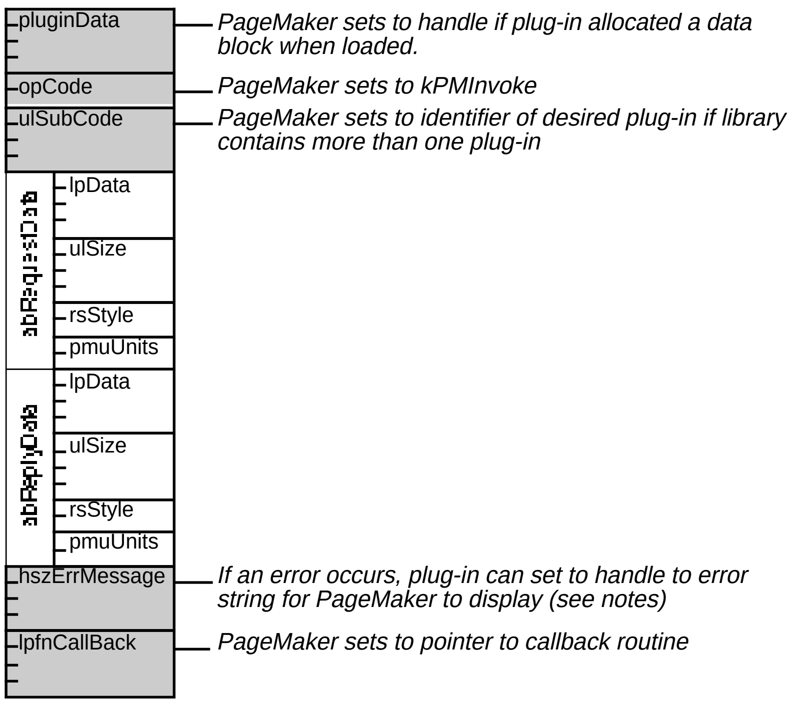
```
HANDLEhMyData = (HANDLE) lpParamBlk->pluginData;
short*ps;
if (!hMyData) {// hMyData will be NULL if
// plug-in has never been called.
//-----------------------------------------------------------
// Allocate pluginData:
// This maintains the last count of number of boxes across
// a page, used by the sample BoxTest routine's Dialog Box.
//-----------------------------------------------------------
hMyData = MMAlloc(SIZE_OF_DATA);
lpParamBlk->pluginData = hMyData;
//-----------------------------------------------------------
// Initialize the data handle.
//-----------------------------------------------------------
ps = (short*)MMLock(hMyData);
*ps = DEF_NUM_BOXES;// default number of boxes for dialog.
MMUnlock(hMyData);
}
return CQ_SUCCESS;
}
```

Invoking

A plug-in's invocation routine jumps to the plug-in's working code. On the Macintosh, the plug-in jumps to either RAG2 for 680x0 code or to the data fork for Power Macintosh code, depending upon the PageMaker version and the type of Macintosh.

Op code: kPMInvoke: PageMaker sends kPMInvoke when the user selects a plug-in from the PageMaker Plug-ins submenu (or from a script) or when another plug-in requests it. PageMaker always loads the plug-in before sending kPMInvoke unless the plug-in is already open.

Return code: The plug-in should return CQ_SUCCESS or a non-zero return code (as defined in PageMakerCQErrs.h).


pluginData
PageMaker sets to handle if plug-in allocated a data block when loaded.
opCode
PageMaker sets to kPMInvoke
ulSubCode
PageMaker sets to identifier of desired plug-in if library contains more than one plug-in
abRequestData
lpData
ulSize
rsStyle
pmuUnits
abReplyData
lpData
ulSize
rsStyle
pmuUnits
hszErrMessage
If an error occurs, plug-in can set to handle to error string for PageMaker to display (see notes)
lpfnCallBack
PageMaker sets to pointer to callback routine

hszErrMessage: If the plug-in returns a non-zero return code, it can set this to a handle to a string that describes why the requested operation failed. The string must end in two null characters, but it may consist of two parts separated by a null character. (See "Error and status codes" in Chapter 2 for a description of the format of this string.) If the plug-in fails for any reason during invocation, use this string to describe the problem to the user.

Example

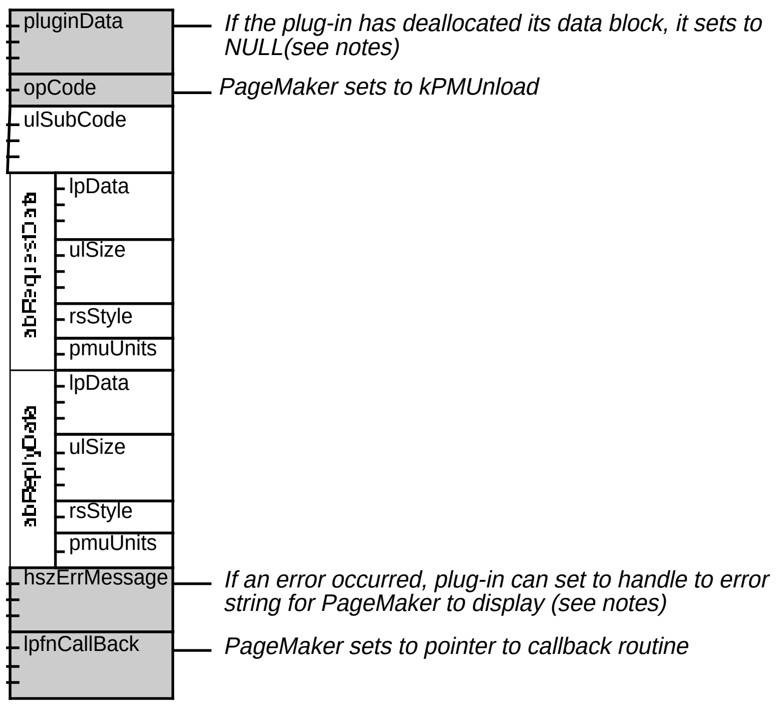
```
PMErr DoInvoke(sPMParamBlockPtr lpParamBlk){
    switch( PBGetId(lpParamBlk) )
    {
    case kPubInfo:
        PMErr = PlacePubInfo(lpParamBlk);
        break;
    case kView100:
        PMErr = View100(lpParamBlk);
        break;
    default:
        PMErr = RC_FAILURE;
        break;
    }
    return CQ_SUCCESS;}
```

Unloading

A plug-in’s unloading routine performs any cleanup required to minimize a plug-in’s memory footprint, such as updating global data, freeing any memory allocated (other than that pointed to by pluginData), and unloading resources and any additional code segments (Macintosh).

Op code: kPMUnload: PageMaker sends kPMUnload when the plug-in has returned from its invocation routine.

Return code: The plug-in should return CQ_SUCCESS or a non-zero return code (as defined in PageMakerCQErrs.h).


pluginData
If the plug-in has deallocated its data block, it sets to NULL(see notes)
opCode
PageMaker sets to kPMUnload
ulSubCode
abRequestData
lpData
ulSize
rsStyle
pmuUnits
abReplyData
lpData
ulSize
rsStyle
pmuUnits
hszErrMessage
If an error occurred, plug-in can set to handle to error string for PageMaker to display (see notes)
lpfnCallBack
PageMaker sets to pointer to callback routine

pluginData: If the plug-in previously allocated a private data block (the field contains a handle) and does not need it for subsequent calls this session, the library should deallocate the block and set this field to NULL.

hszErrMessage: If the plug-in returns a non-zero return code, it can set this to a handle to a string that describes why the requested operation failed. The string must end in two null characters, but it may consist of two parts separated by a null character. (See “Error and status codes” in Chapter 2 for a description of the format of this string.) If the plug-in fails for any reason during unloading, use this string to describe the problem to the user.

Example

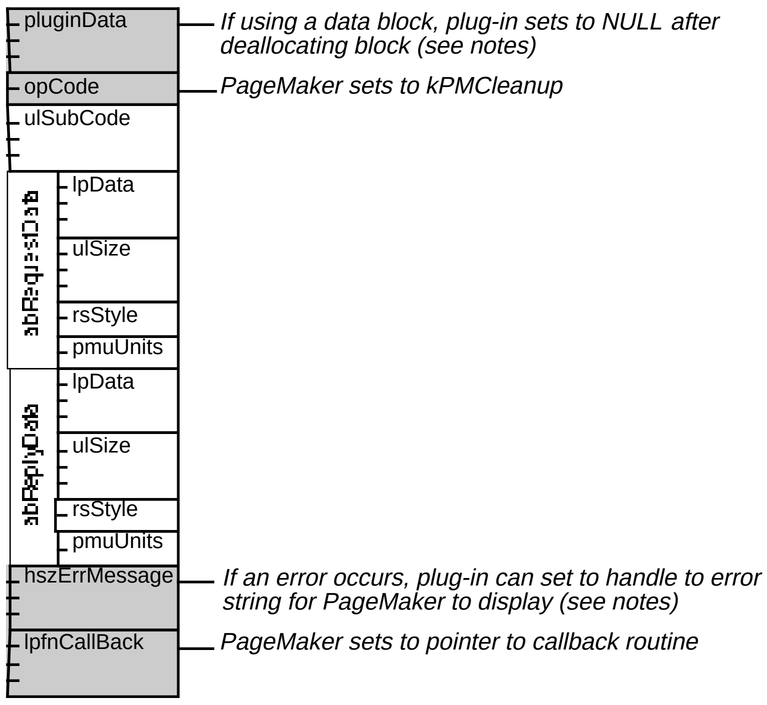
```
PMErr DoUnload(sPMParamBlockPtr lpParamBlk)
    {
    // Do any cleanup necessary before plug-in code segment is unloaded.
    // Update any global data, free memory allocated except pluginData
    // Unload resources, etc.
    return CQ_SUCCESS;
    }
```

Cleaning up

A plug-in's cleanup routine frees as much memory as it can, including any memory the plug-in allocated and any resources not needed immediately. (In future versions, PageMaker may cache plug-ins rather than unload them after each invocation and will use this routine to clear them from memory.)

Op code: kPMCleanup: PageMaker sends kPMCleanup when memory becomes critically low.

Return code: The plug-in should return CQ_SUCCESS or a non-zero return code (as defined in PageMakerCQErrs.h).


pluginData
If using a data block, plug-in sets to NULL after deallocating block (see notes)
opCode
PageMaker sets to kPMCleanup
ulSubCode
sbRequestData
lpData
ulSize
rsStyle
pmuUnits
sbReplyData
lpData
ulSize
rsStyle
pmuUnits
hszErrMessage
If an error occurs, plug-in can set to handle to error string for PageMaker to display (see notes)
lpfnCallBack
PageMaker sets to pointer to callback routine

pluginData: If the plug-in previously allocated a private data block (the field contains a handle) and does not need it for subsequent calls this session, the library should deallocate the block and set this field to NULL.

hszErrMessage: If the plug-in returns a non-zero return code, it can set this to a handle to a string that describes why the requested operation failed. The string must end in two null characters, but it may consist of two parts separated by a null character. (See "Error and status codes" in Chapter 2 for a description of the format of this string.) If the plug-in fails for any reason during cleanup, use this string to describe the problem to the user.

Minimizing private memory block: To minimize the size of the private memory block, a plug-in can write data to the resource fork (Macintosh), data fork (Macintosh), or private configuration file (Windows and Macintosh).

Example

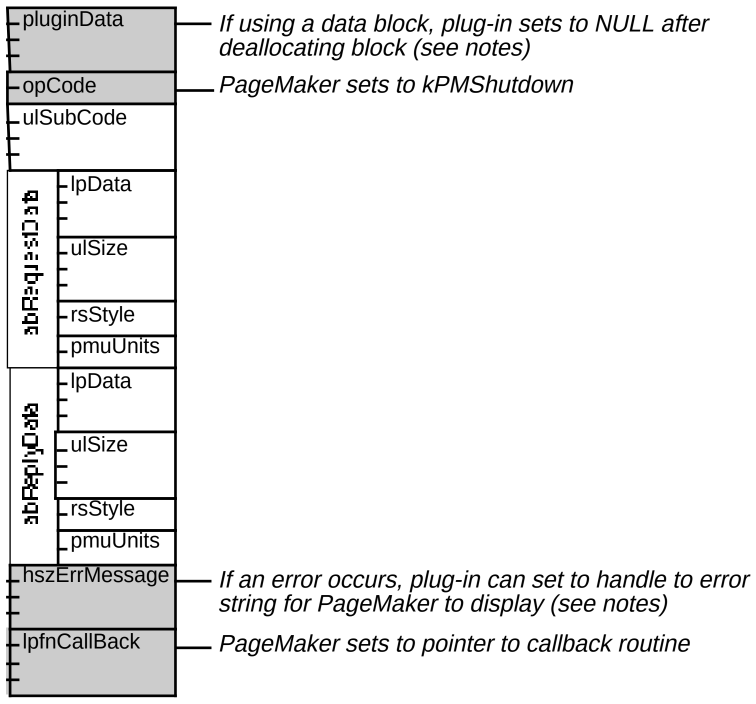
```
PMErr DoCleanup(sPMParamBlockPtr lpParamBlk)
    {
    // Free any memory allocated, resources, and additional memory
    // If PM still doesn't have enough memory, PM procedes with Shutdown
    // Save data to config file & free
    return CQ_SUCCESS;
    }
```

Shutting down

A plug-in’s shutdown routine deallocates memory it has used and saves any settings or values in the private memory block if appropriate.

Op code: kPMShutdown: PageMaker shuts down all plug-ins when the user selects “Quit” from the PageMaker File menu.

Return code: The plug-in can return any return code because PageMaker ignores the value and proceeds with shutdown.


pluginData
opCode
ulSubCode
pbRequestData
lpData
ulSize
rsStyle
pmuUnits
pbReplyData
lpData
ulSize
rsStyle
pmuUnits
hszErrMessage
lpfnCallBack
If using a data block, plug-in sets to NULL after deallocating block (see notes)
PageMaker sets to kPMShutdown
If an error occurs, plug-in can set to handle to error string for PageMaker to display (see notes)
PageMaker sets to pointer to callback routine

pluginData: If the plug-in previously allocated a private data block (the field contains a handle) and does not need it for subsequent calls this session, the library should deallocate the block and set this field to NULL.

hszErrMessage: If the plug-in returns a non-zero return code, it can set this to a handle to a string that describes why the requested operation failed. The string must end in two null characters, but it may consist of two parts separated by a null character. (See “Error and status codes” in Chapter 2 for a description of the format of this string.) If the plug-in fails for any reason during cleanup, use this string to describe the problem to the user.

Example

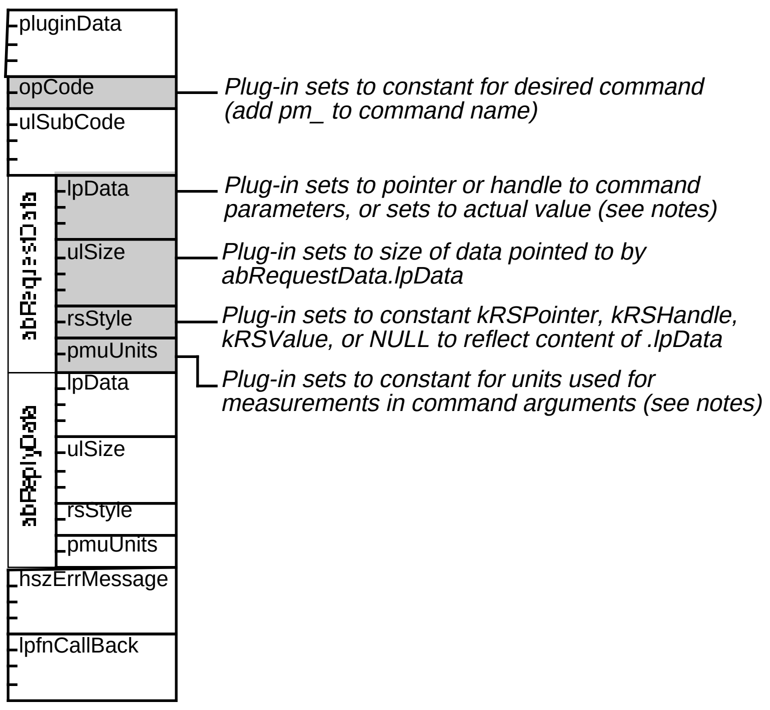
```
// PageMaker is shutting down. Save necessary info to disk.
// Free all allocated memory, including pluginData or
// memory in Global Heap. Last call before plug-in is closed.
    PMErr DoShutdown(sPMParamBlockPtr lpParmBlk)
    {
    HANDLE  hMyData = PBGetPluginData(lpParmBlk);
    if (hMyData)
        MMFree(hMyData);  // Free private data handle:

    return CQ_SUCCESS;
    }
```

Sending commands to PageMaker

When a plug-in sends commands to PageMaker it sets the parameters noted below.

A plug-in sends commands before it has returned from its invocation routine.

Return code: PageMaker returns CQ_SUCCESS or a non-zero return code. Return codes are defined in PageMakerCQErrs.h. If the return code indicates PageMaker has encountered an error, the plug-in can query PageMaker for the last error string and display that message to the user. (See “Error and status codes” in Chapter 2 and “GetLastErrorStr” in Chapter 10.)


pluginData
opCode
ulSubCode
abRequestData
lpData
ulSize
rsStyle
pmuUnits
abReplyData
lpData
ulSize
rsStyle
pmuUnits
hszErrMessage
lpfnCallBack
Plug-in sets to constant for desired command (add pm_ to command name)
Plug-in sets to pointer or handle to command parameters, or sets to actual value (see notes)
Plug-in sets to size of data pointed to by abRequestData.lpData
Plug-in sets to constant kRSPointer, kRSHandle, kRSValue, or NULL to reflect content of .lpData
Plug-in sets to constant for units used for measurements in command arguments (see notes)

abRequestData.lpData: The plug-in sets this field to a pointer or handle to the command arguments if any, or if the arguments fit within the four-byte field (ULONG), the actual values. If the command does not require any parameters, the plug-in should set the field to NULL. PageMaker ignores the remaining fields if abRequestData.lpData is NULL.

abRequestData.pmuUnits: If using the text format, the plug-in sets this field to the measure unit constant for coordinates and values in the command arguments. Measurements that accompany binary commands must be specified in twips (PageMaker’s internal measurement system). Unit constants for data in the text format are:
kMUNull
kMUInches
kMUDecInches
kMUCentimeters
kMUPicas
kMUPoints

If measurements are in the publication default units, use kMUNull.

Deallocating buffers: The plug-in must deallocate any buffers it allocated.

Example

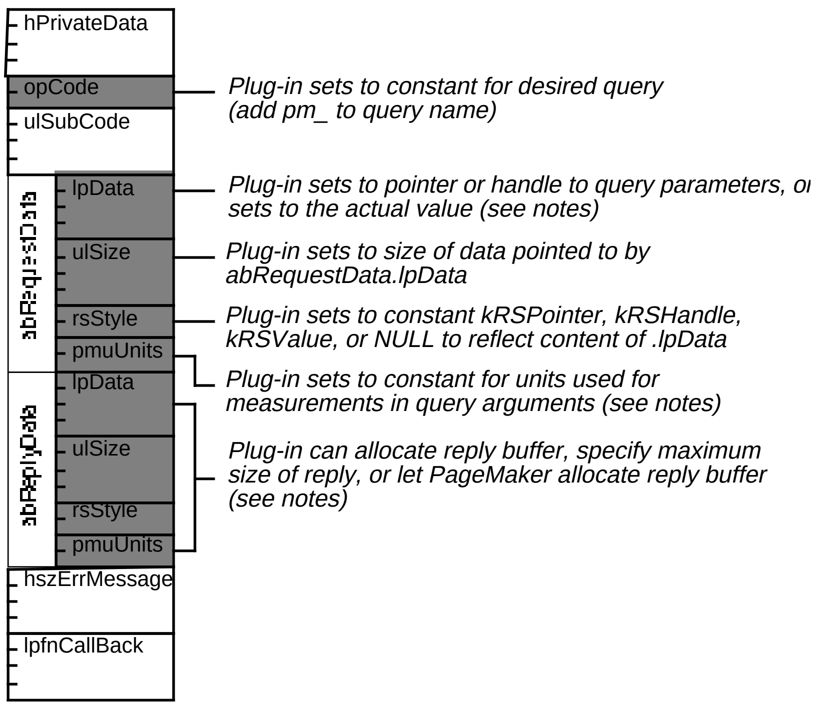
The macros used in this sample are described in Chapter 5, “Macros.”

```
/**[m************************************************
 *Box
 *
 *DESCRIPTION:
 *Draws a box. Deselects anything selected first to avoid adding
 *box to selection list. Uses binary format. INCH converts inches
 *to twips.
 *
 **m]************************************************/
PMErr Box(sPMParamBlockPtr lpParamBlk)
{
PMRect  rBox;

PBBinCommand(lpParamBlk,pm_deselect,kRSNull,NULL,NULL);
                                                // deselect anything selected.
PBBinCommand(lpParamBlk,pm_box,kRSPointer,
 PMSetRect(&rBox, (2*INCH),(3*INCH),(4*INCH),(5*INCH)), sizeof(PMRect));
PBBinCommand(lpParamBlk,pm_color,kRSPointer,"Red",4);
                                                // 4 = strlen("Red")+1
return CQ_SUCCESS;
}
```

Sending queries to PageMaker

When a plug-in sends a query to PageMaker, it needs to include the parameters noted below. A plug-in sends queries before it has returned from its invocation routine.

Return code: PageMaker returns CQ_SUCCESS or a non-zero return code. Return codes are defined in PageMakerCQErrs.h. If the return code indicates PageMaker has encountered an error, the plug-in can query PageMaker for the last error string and display that message to the user. (See "Error and status codes" in Chapter 2 and "GetLastErrorStr" in Chapter 10.)


hPrivateData
opCode
ulSubCode
abRequestData
lpData
ulSize
rsStyle
pmuUnits
abReplyData
lpData
ulSize
rsStyle
pmuUnits
hszErrMessage
lpfnCallBack
Plug-in sets to constant for desired query (add pm_ to query name)
Plug-in sets to pointer or handle to query parameters, or sets to the actual value (see notes)
Plug-in sets to size of data pointed to by abRequestData.lpData
Plug-in sets to constant kRSPointer, kRSHandle, kRSValue, or NULL to reflect content of .lpData
Plug-in sets to constant for units used for measurements in query arguments (see notes)
Plug-in can allocate reply buffer, specify maximum size of reply, or let PageMaker allocate reply buffer (see notes)

Notes:

abRequestData.lpData: The plug-in sets this field to a pointer or handle to the query arguments if any, or if the arguments fit within the four-byte field (ULONG), the actual values. If the query does not require any parameters, the plug-in sets the field to NULL, thus indicating that PageMaker can ignore the remaining fields.

abRequestData.pmuUnits: If using the text format, the plug-in sets this field to the measure unit constant for coordinates and values in the query arguments. Measurements that accompany binary queries must be specified in twips (PageMaker's internal measurement system). Unit constants for data in the text format are:
kMUNull
kMUInches
kMUDecInches
kMUCentimeters
kMUPicas
kMUPoints

If measurements are in the publication default units, use kMUNull.

abReplyData.lpdata: A plug-in can pre-allocate a buffer for the reply, let PageMaker allocate a buffer, or have PageMaker set this field to the actual value. How you specify one of these options is described in the table that follows.

If the plug-in has pre-allocated a buffer, PageMaker leaves this field unchanged. If the field is null, PageMaker supplies a handle or pointer, as indicated in the table.

abReplyData.ulSize: A plug-in can set this field to one of the following (also see the table that follows):

- The size of the buffer the plug-in allocated for reply information (if abReplyData.lpData points to a buffer)
- The maximum size of the reply data the plug-in can accept
- NULL if the plug-in allows PageMaker to allocate the reply data. In this case, PageMaker sets the field to the size of the reply data.

abReplyData.rsStyle: A plug-in can set this to the constant kRSPointer, kRSHandle, or kRSValue, depending upon if it has pre-allocated a buffer, wants PageMaker to allocate a buffer, or wants PageMaker to set abReplyData.lpData to the query value. See the table that follows.

Because it is often necessary for PageMaker to resize a buffer to accommodate query results, we recommend the plug-in pass a handle rather than a pointer, allowing PageMaker to resize the buffer as needed. Use kRSPointer only to point to buffers in your local stack frame or in the global heap.

abReplyData.pmuUnits: When using the text format, the plug-in can specify the measurement units PageMaker should use for the reply values. See abRequestData.pmuUnits for the unit constants. When using the binary format, PageMaker uses twips only for reply values.

PageMaker leaves this field unchanged. For queries in the text format, if the plug-in does not specify units, PageMaker uses the publication default units.

Table of abReplyData fields: The following table summarizes how a plug-in can set the abReplyData fields:

.lpData	.rsStyle	.ulSize	Description
Pointer/ handle	Required	Required	PageMaker should use this reply buffer.
NULL	kRSHandle*	NULL	PageMaker should allocate a buffer of this style.
NULL	kRSHandle*	Value	PageMaker should allocate a buffer of this style no larger than lpReplyData.ulSize.

.lpData	.rsStyle	.ulSize	Description
NULL	NULL	Value	PageMaker should allocate a buffer no larger than lpReplyData.ulSize.
NULL	NULL	NULL	PageMaker should allocate the buffer as needed.
NULL	kRSValue	NULL	PageMaker should return the reply data in the four-byte abReplyData.lpData field.

* A plug-in must always request a handle when it needs PageMaker to allocate the reply buffer. The plug-in, however, is responsible for deallocating the buffer.

Getting replies in the text format: The binary format is the default reply format for text queries, and the only format for replies from binary queries. When using the text format, you can request that the replies be in the text format by including a flag, kXRFText, with the reply style constant in abReplyData.rsStyle. (kXRFText is defined in PageMakerTypes.h.) For example:

```
flag = kRSHandle | kXRFText// Pass back by handle as ASCII
text
```

If you include kXRFText with a binary query, PageMaker ignores your request. (Note that in cases where the actual return value of a binary query is text—such as a publication name, a font name, or story text—PageMaker returns a null-terminated string.) The examples below illustrate how to specify formats for return values.

```
rcVal = PBTextQuery(lpParamBlk, // Gets object list for
current page

kRSPointer, “getobjectlist”,// By default, return is in
binary

kRSHandle, NULL, MAXSIZE);// format for text and binary
queries.


rcVal = PBTextQuery(lpParamBlk,// Gets object list for
current page

kRSPointer, “getobjectlist”,

kRSHandle | kRFBinary,// specifies binary format for
results.

NULL, MAXSIZE);


rcVal = PBTextQuery(lpParamBlk,// Gets object list for
current page

kRSPointer, “getobjectlist”,

kRSHandle | kXRFText, // specifies text format for results

NULL, MAXSIZE);


rcVal = PBBinQuery(lpParamBlk,// Binary reply is always in
twips

pm_getobjectlist,

kRSHandle | kXRFText,// PageMaker ignores kXRFText flag

NULL, MAXSIZE);
```

Deallocating memory: The plug-in must deallocate any buffers that it or PageMaker allocated to hold query results.

Buffer too small: If the reply buffer the plug-in allocated wasn't large enough for the reply data, PageMaker returns the failure code RC_BUFFSIZE_TOO_SMALL.

Important! Pointer versus handle: In general, we suggest you allocate memory on the heap. Only allocate memory on the stack for data that is small and needed temporarily. Allocate memory on the heap for data shared by plug-ins or that needs to be retained.

Example

The following sample code demonstrates the various ways to request query results and provide query parameters. It places the acquired text as a new story. (The macros used in this sample are described in Chapter 5, "Macros.")

```
/**[m******************************************************
 *RunQTest
 *DESCRIPTION
 *This routine tests the different ways to request query results and
 *provide command or query parameters. In addition, commands may
 *use the ...CommandByShortValue or ...CommandByLongValue macros
 *as a convenience. Queries without parameters can use the PBBinQuery
 *macro as well, the parameters would be the same as those used here
 *for reply packets.
 *
 **m]******************************************************/
RC  RunQTest(sPMParamBlockPtr lpParamBlk)
{
HANDLE h=MMAlloc(STORYSIZE);
HANDLE myHandle; // static handle
HANDLE hMyParms;
    LPSTR psz;
SHORT arParms[2]; // parameters for query.
    RC rcVal = CQ_SUCCESS;
PMPoint ptResult; // point to top, left of new results story.
char buff[STORYSIZE]; // static buffer.


SetHandle(&h, "Reply Style Variation Tests:\n---------------------------
\n\n");
/* set up request and reply units to be the same.
*/
PBSetRequestUnits(lpParamBlk,kMUInches);
PBSetReplyUnits(lpParamBlk,kMUInches);
/* set up specifics for getStoryText query.
*/
//-----------------------------------------------------------
// This test suite tests only reply buffers, but uses all
// reference styles. The first sample is the recommended
// way to use queries and parameters.
//-----------------------------------------------------------
// Buffer not supplied, Handle, size not given:
HStrCat(&h, "4. Buffer not supplied, Handle, size not given: ");
rcVal = PBBinQueryWithParms(lpParamBlk,pm_getstorytext,kRSPointer,arParms,
    sizeof(arParms),kRSHandle,NULL,NULL);
myHandle = PBGetReplyData(lpParamBlk);
if (!rcVal && myHandle) {
    DoSuccess(&h, MMLock(myHandle));
    MMUnlock(myHandle);
} else {
    DoFailure(&h,rcVal);
}
if (myHandle) MMFree(myHandle);

// Buffer supplied, pointer.
HStrCat(&h, "1. Buffer supplied, pointer: ");
arParms[0] = 0;
arParms[1] = 3;
rcVal = PBBinQueryWithParms(lpParamBlk,pm_getstorytext,kRSPointer,arParms,
    sizeof(arParms),kRSPointer,buff,STORYSIZE);
if (!(rcVal)) {
    DoSuccess(&h,buff);
} else {
    DoFailure(&h,rcVal);
}

// Buffer supplied, handle.
HStrCat(&h, "2. Buffer supplied, handle: ");
myHandle=MMAlloc(STORYSIZE);
rcVal = PBBinQueryWithParms(lpParamBlk,pm_getstorytext,kRSPointer,arParms,
    sizeof(arParms),kRSHandle,myHandle,STORYSIZE);
if (!(rcVal) ) {
    DoSuccess(&h, MMLock(myHandle));
    MMUnlock(myHandle);
} else {
    DoFailure(&h,rcVal);
}
if (myHandle) MMFree(myHandle);
}
// Buffer not supplied, Handle, size given:
```

```
HStrCat(&h, "3. Buffer not supplied, Handle, size given: ");
rcVal = PBBinQueryWithParms(lpParamBlk,pm_getstorytext,kRSPointer,arParms,
    sizeof(arParms),kRSHandle,NULL,STORYSIZE);
myHandle = PBGetReplyData(lpParamBlk);
if (!rcVal && myHandle) {
    DoSuccess(&h, MMLock(myHandle));
    MMUnlock(myHandle);
} else {
    DoFailure(&h,rcVal);
}
if (myHandle) MMFree(myHandle);

// Buffer not supplied, no style, size given:
HStrCat(&h, "5. Buffer not supplied, no style, size given: ");
rcVal = PBBinQueryWithParms(lpParamBlk,pm_getstorytext,kRSPointer,arParms,
    sizeof(arParms),NULL,NULL,MAXSTORYSIZE);
myHandle = PBGetReplyData(lpParamBlk);
if (!rcVal && myHandle) {
    DoSuccess(&h, MMLock(myHandle));
    MMUnlock(myHandle);
} else {
    DoFailure(&h,rcVal);
}
if (myHandle) MMFree(myHandle);

// Buffer not supplied, no style, size not given:
HStrCat(&h, "6. Buffer not supplied, no style, size not given: ");
rcVal = PBBinQueryWithParms(lpParamBlk,pm_getstorytext,kRSPointer,arParms,
    sizeof(arParms),NULL,NULL,NULL);
myHandle = PBGetReplyData(lpParamBlk);
if (!rcVal && myHandle) {
    DoSuccess(&h, MMLock(myHandle));
    MMUnlock(myHandle);
} else {
    DoFailure(&h,rcVal);
}
if (myHandle) MMFree(myHandle);
//-----------------------------------------------------------
// These tests try the query with reference style #6, using
// each reference style for the query parameters.
//-----------------------------------------------------------
HStrCat(&h, "\nParameter Style Variation Tests:\n-----------------------------
---\n\n");
HStrCat(&h, "7. By Value, 2 Shorts: ");
```

```
rcVal =
PBBinQueryWithParms(lpParamBlk,pm_getstorytext,kRSValue,MAKELONG(arParms[1],
    arParms[0]),sizeof(LONG),NULL,NULL,NULL);
myHandle = PBGetReplyData(lpParamBlk);
if (!rcVal && myHandle) {
    DoSuccess(&h, MMLock(myHandle));
    MMUnlock(myHandle);
} else {
    DoFailure(&h,rcVal);
}
if (myHandle) MMFree(myHandle);

HStrCat(&h, "8. By Pointer: ");
rcVal = PBBinQueryWithParms(lpParamBlk,pm_getstorytext,kRSPointer,arParms,
    sizeof(arParms),NULL,NULL,NULL);
myHandle = PBGetReplyData(lpParamBlk);
if (!rcVal && myHandle) {
    DoSuccess(&h, MMLock(myHandle));
    MMUnlock(myHandle);
} else {
    DoFailure(&h,rcVal);
}
if (myHandle) MMFree(myHandle);

HStrCat(&h, "9. By Handle: ");
hMyParms = MMAlloc(sizeof(arParms));
psz = MMLock(hMyParms);
((short*)psz)[0] = arParms[0];
((short*)psz)[1] = arParms[1];
MMUnlock(hMyParms);
rcVal = PBBinQueryWithParms(lpParamBlk,pm_getstorytext,kRSHandle,hMyParms,
    sizeof(arParms),NULL,NULL,NULL);
myHandle = PBGetReplyData(lpParamBlk);
if (!rcVal && myHandle) {
    DoSuccess(&h, MMLock(myHandle));
    MMUnlock(myHandle);
} else {
    DoFailure(&h,rcVal);
}
if (myHandle) MMFree(myHandle);
MMFree(hMyParms);

//---------------------------------------------------------
// The test results are placed as a new story.
```

```
//
// This deselects the current selection (story text), so
// the selection list is clear and the new story will be
// the only thing selected.
//----------------------------------------------------------
    // deselect original text.
PBBinCommand(lpParamBlk,pm_deselect,kRSNull,NULL,NULL);
PBBinCommand(lpParamBlk,pm_newstory,kRSPointer,PMSetPoint(&ptResult,0,0),
 sizeof(PMPoint));
    // set font to 14.0 points.
PBBinCommandByShortValue(lpParamBlk,pm_size,140);
PBBinCommand(lpParamBlk,pm_textenter,kRSHandle,h,MMSize(h));

MMFree(h);  // free the handle that contained the results.
return CQ_SUCCESS;
}
```

Memory-Manager Routines for Loadable Plug-ins

This chapter describes the Adobe Memory-Manager routines for allocating, referencing, locking, and freeing memory.

Why use these routines?

In this version of PageMaker, we do not require you to use our memory-management routines in a loadable plug-in. However, you may be required to use these routines in future versions. Although optional, these routines can save you time in both developing and testing your plug-in. They have been used extensively in our products and are reliable.

In addition, these routines work in both the Windows and Macintosh environments. Therefore, using them can really help you if you develop portable plug-ins (plug-ins that you can compile and use in both environments).

Where to find the routines

This SDK includes the header and source files your plug-in needs to use the Adobe Memory-Manager routines: PageMakerMemory.h and util.c.

Adobe Memory-Manager routines

The following list briefly describes the Memory-Manager routines you may want to use in your plug-in. The Memory Manager provides routines for allocating relocatable memory.

MMAlloc:	Allocates relocatable memory and returns a handle to it. (In Windows, memory is allocated from the global heap.) Use this routine to allocate memory that is retained between invocations of a plug-in, such as memory for user preferences. (Equivalent to Macintosh NewHandle.)
MMFree:	Frees memory allocated with MMAlloc. (Equivalent to Macintosh DisposHandle.)
MMGetPointer:	Gets pointer to locked memory block.
MMLock:	Locks handle in memory and returns a pointer to it.
MMResizeHandle:	Changes size of a relocatable memory block.
MMUnlock:	Unlocks locked memory.

MMAlloc

```
HANDLE MMAlloc(dwSize)
```

Allocates a block of global memory and returns a handle to the memory. On the PC, the memory is allocated from the global heap. The block is not initialized. (On the Macintosh, this routine is equivalent to the NewHandle routine.)

Type	Parameter	Description
DWORD	dwSize	Amount of memory to be allocated

Return value: A handle to the allocated memory or NULL if the routine fails.

Relocatable memory: Memory allocated with MMAlloc is relocatable. The handle is always valid and there is no loss of data when a block is moved because the master pointer is updated if the block is relocated.

Locking the handle: To access the allocated memory, lock the memory using MMLock.

Private data: Plug-in libraries should use this routine to allocate global memory for the private data that PageMaker maintains after the plug-in is unloaded.

Freeing memory: Use MMFree to free memory allocated with MMAlloc.

Handling errors: If the operating system fails to allocate memory, the plug-in should return RC_NSF_MEMORY to PageMaker.

Example

The following example uses MMAlloc, MMLock, and MMUnlock.

```
PMErr DoLoad(sPMParamBlockPtr lpParamBlk)
{
HANDLE hMyData = PBGetPluginData(lpParamBlk);
short *ps;
if (!hMyData) {   // is NULL if plug-in has never been called
//---------------------------------------------------------
// Allocate pluginData:
// Maintains last count of boxes across a page, used by BoxTest example.
//---------------------------------------------------------
hMyData = MMAlloc(SIZE_OF_DATA);
PBSetPluginData(lpParamBlk, hMyData);
//---------------------------------------------------------
// Initialize the data handle.
//---------------------------------------------------------
ps = (short*)MMLock(hMyData);
*ps = DEF_NUM_BOXES;  // default number of boxes for dialog.
MMUnlock(hMyData);
        }
    return CQ_SUCCESS;
    }
```

MMFree

```
void MMFree(hMem)
```

Frees a movable block of global memory allocated by MMAlloc. The block must be unlocked to be freed. (On the Macintosh, this routine is equivalent to the DiposHandle routine.)

Type	Parameter	Description
HANDLE	hMem	A handle to the block of memory to be freed

Return value: None.

Unlocking handle: Unlock the handle before attempting to deallocate the memory. See example below.

Example

The following example uses MMLock, MMUnlock, and MMFree.

```
PMErr Box(sPMParamBlockPtr lpParamBlk)
{
RC rcVal;
short *spNumPages = 0;
HANDLE hNumPages = 0;
// First make sure there is a page to draw on.
rcVal = PBBinQuery(lpParamBlk, pm_getpages, kRSHandle, NULL, NULL);
hNumPages = PBGetReplyData(lpParamBlk);
spNumPages = (short*) MMLock(hNumPages);
//If there are one or more pages in the pub, post the sample dialog.
//Otherwise post an error.
if (*spNumPages)
rcVal = PostSampleDialog(lpParamBlk);
else
return Error(lpParamBlk, RC_FAILURE, “Error: no pages found.”,””);
MMUnlock(hNumPages);
MMFree(hNumPages);
return (rcVal);
}
```

MMGetPointer

```
LPSTR MMGetPointer(hLockedHandle)
```

Returns a pointer to a locked handle. This routine is the only safe way to get a pointer for a locked handle.

Type	Parameter	Description
HANDLE	hLocked	Handle Locked handle to the memory block for which a pointer is requested.

Return value: A pointer to the locked block or NULL if the routine failed.Use this function if the plug-in does not have the pointer returned by MMLock. The handle must be locked before calling MMGetPointer.

Example

Here is a sample use of this routine:

```
LPSTR lpData;
if (!(lpData = MMGetPointer (hMem)))
{
… /* failed */
}
```

MMLock

```
LPSTR MMLock(hMem)
```

Locks a relocatable block of global memory so it can be safely accessed.

Type	Parameter	Description
HANDLE	hMem	Handle to be locked

Return value: A pointer to the locked block or NULL if the routine failed.

Locking required: A plug-in must always lock a block of memory allocated with MMAlloc before accessing it.

Locking a locked block: If a plug-in locks an already locked block of memory, the block's state won't change: it remains locked.

Note: Windows keeps track of the number of times a block of memory is locked; the Macintosh does not. Therefore, to free a block of memory in Windows, a plug-in must unlock the block the same number of times it locked it. On the Macintosh, a plug-in need unlock a locked block only once to free it, regardless of the number of times it was locked.

Example

Here is a sample use of this routine (for a more complete example, see the sample code in the descriptions of MMAlloc and MMFree):

```
spNumPages = (short*) MMLock(hNumPages);
```

MMResizeHandle

```
PMErr MMResizeHandle(lphMem, dwSize)
```

Changes the size of an unlocked block of memory.

Type	Parameter	Description
LPHANDLE	lphMem	Pointer to the handle (LPHANDLE) of the block of memory that is to be locked
DWORD	dwSize	New size of block

Return value: CQ_SUCCESS or a non-zero value.

Important: Do not attempt to change the size of a locked block.

Resizes at high end: The block grows or shrinks at the high end. Any new space is not initialized.

Caution for Windows developers: This routine uses the Windows routine GlobalReAlloc. Therefore, the handle passed may change. MMResizeHandle will update the handle if it changes.

MMUnlock

```
void MMUnlock(hMem)
```

Unlocks a block that was locked by MMLock.

Type	Parameter	Description
HANDLE	hMem	Handle to be unlocked

Return value: None.

Lock count: Windows keeps track of the number of times a block of memory is locked; the Macintosh does not. Therefore, to free a block of memory in Windows, a plug-in must unlock the block the same number of times it locked it. On the Macintosh, a plug-in need unlock a locked block only once to free it, regardless of the number of times it was locked.

When writing portable code, it is better to pass handles to underlying service routines that lock and unlock memory blocks than to pass locked handles or pointers.

Example

Here is a sample use of this routine (for a more complete example, see the sample code in the descriptions of MMAlloc, MMFree, and MMResizeHandle):

```
MMUnlock(hError);
```

5

Macros

The macros included with this SDK move data into and out of the correct fields of the parameter block or data buffer. We have provided these macros to make it easier for you to develop a plug-in. In PageMakerUtils.c, a number of these macros have been replaced by actual functions of the same name, providing type safety, reduced chance for side-effect errors, and smaller executables.

This chapter documents only the most frequently used macros. Refer to PageMakerPBMacros.h for information about other macros included in this SDK.

Macro locations

All but two of the macros are included in PageMakerTypes.h, which you'll find in the Ink folder. The source code for both LPGetString() and LPPutString() is contained in CQUtils.c in the "Src" folder.

Finding descriptions

For easy reference, the macros are listed alphabetically in this chapter. The list below groups the macros by function:

- Move binary values in and out of data buffers:

  LPGetLong

  LPGetShort

  LPGetString

  LPPutLong

  LPPutShort

  LPPutString
- Move data in and out of the parameter block for loading, invoking, unloading, cleanup, and shutdown:

  PBGetID

  PBGetOpCode

  PBGetPluginData

  PBSetErrMessage

  PBSetPluginData
- Move data in the binary format in and out of the parameter block when sending commands and queries:

  PBBinCommand

  PBBinCommandByShortValue

  PBBinCommandByLongValue

  PBBinQuery

  PBBinQueryWithParms

  PBGetReplyData
- Move data in the text format in and out of the parameter block when sending commands and queries:

  PBGetReplyData

  PBTextCommand

  PBTextQuery

  PBSetReplyUnits

  PBSetRequestUnits
- Provide functionality to one of the other macros:

  PBClearReplyBlock

  PBClearRequestBlock

  PBSetOpCode

  PBSetReplyBlock

  PBSetRequestBlock

LPGetHandle

```
HANDLE LPGetHandle(&v,pSrc)
```

Extracts a handle from a data block and returns a value indicating the amount to increment the pointer.

Type	Parameter	Description
HANDLE	&v	Variable to contain the handle
long	pSrc	Pointer to data

Return value: Number of bytes copied [sizeof(HANDLE)].

See also: LPGetLong
LPGetShort
LPGetString
LPPutHandle
LPPutShort
LPPutLong
LPPutString

LPGetLong

```
short LPGetLong(&v,pSrc)
```

Gets a long value from a data block and returns a value indicating the amount to increment the pointer.

Type	Parameter	Description
long	&v	Variable to contain the long value
long	pSrc	Pointer to data

Return value: Number of bytes copied [sizeof(long)].

Using a handle: The pSrc parameter can be a handle, but you must lock the handle first and use its pointer in the macro.

See also: LPGetHandle
LPGetShort
LPGetString
LPPutHandle
LPPutShort
LPPutLong
LPPutString

short LPGetShort(&v,pSrc)

```
short LPGetShort(&v,pSrc)
```

Extracts a short value from a data block and returns a value indicating the amount to increment the pointer.

Type	Parameter	Description
short	&v	Variable to contain the short value
long	pSrc	Pointer to data

Return value: Number of bytes copied [sizeof(short)].

Using a handle: The pScr parameter can be a handle, but you must lock the handle first and use its pointer in the macro.

See also: LPGetHandle
LPGetLong
LPGetString
LPPutHandle
LPPutShort
LPPutLong
LPPutString

LPGetString

```
short LPGetString(pDst, pSrc, len)
```

Gets a string from a data block and returns a value indicating the amount to increment the pointer.

Type	Parameter	Description
LPSTR	pDst	Pointer to the destination string
LPVOID	pSrc	Pointer to data
short	len	Maximum length of data buffer

Return value: Number of bytes copied, including the null-terminator and any pad bytes necessary to align the next field with the word boundary.

Using a handle: The pSrc parameter can be a handle, but you must lock the handle first and use its pointer in the macro.

See also: LPGetHandle
LPGetLong
LPGetShort
LPPutHandle
LPPutLong
LPPutShort
LPPutString

LPPutHandle

```
HANDLE LPPutHandle(pDst,v,)
```

Places a handle in a data block and returns a value indicating the amount to increment the pointer.

Type	Parameter	Description
long	pDst	Pointer to data
HANDLE	v	Handle to append to data buffer

Return value: Number of bytes copied [sizeof(HANDLE)].

See also: LPGetHandle
LPGetLong
LPGetShort
LPGetString
LPPutShort
LPPutLong
LPPutString

LPPutLong

```
short LPPutLong(pDst, v)
```

Places a long value in a data block and returns a value indicating the amount to increment the pointer.

Type	Parameter	Description
long	pDst	Pointer to the destination buffer
long	v	Value to append to data buffer

Return value: Number of bytes copied [sizeof(long)].

Using a handle: The pDst parameter can be a handle, but you must lock the handle first and use its pointer in the macro.

See also: LPGetHandle
LPGetLong
LPGetShort
LPGetString
LPPutHandle
LPPutShort
LPPutString

LPPutShort

```
short LPPutShort(pDst, v)
```

Places a short value in a data block and returns a value indicating the amount to increment the pointer.

Type	Parameter	Description
long	pDst	Pointer to the destination buffer
short	v	Value to append to data buffer

Return value: Number of bytes copied [sizeof(short)].

Using a handle: The pDst parameter can be a handle, but you must lock the handle first and use its pointer in the macro.

See also: LPGetHandle
LPGetLong
LPGetShort
LPGetString
LPPutHandle
LPPutLong
LPPutString

LPPutString

```
short LPPutString(pDst, pSrc)
```

Places a string in a data block and returns a value indicating the amount to increment the pointer.

Type	Parameter	Description
LPVOID	pDst	Pointer to the data block
LPSTR	pSrc	Pointer to string to append to data block

Return value: Number of bytes copied, including the null-terminator and any pad bytes necessary to align the next field with the word boundary.

Using a handle: The pDst parameter can be a handle, but you must lock the handle first and use its pointer in the macro.

See also: LPGetHandle
LPGetLong
LPGetShort
LPGetString
LPPutHandle
LPPutLong
LPPutShort

PBBinCommand

```
PMErr PBBinCommand(lpPB,op,sy,hp,sz)
```

Issues a command to PageMaker using the binary format. Use this macro for commands that have more than one parameter.

Type	Parameter	Description
sPMParamBlockPtr	lpPB	Pointer to the parameter block
short	op	Command operation code, or binary ID (e.g., pm_open)
short	sy	Reference style of the hp parameter, either kRSPointer or kRSHandle
pointer or handle	hp	Pointer or handle (as specified by the sy parameter) to the values for the parameter block
short	sz	Size of data referenced by the hp parameter

Return value: CQ_SUCCESS or a non-zero return code.

Binary ID: The command operation code, or binary ID, is the command name preceded by pm_. For example, the binary ID for the Close command is pm_close. Each binary ID is also listed in the header file PageMakerCommands.h.

Example

```
/**[m****************************************************

    *  DESCRIPTION:

    Draws a box. Deselects anything selected first.

    **m]****************************************************/

    PMErr  Box(sPMParamBlockPtr lpParamBlk)

    {

    PMRect rBox;

    PBBinCommand(lpParamBlk,pm_deselect,kRSNull,NULL,NULL);

    // de-select anything selected.

    PBBinCommand(lpParamBlk,pm_box,kRSPointer,

        PMSetRect(&rBox, (2*INCH),(3*INCH),(4*INCH),(5*INCH)),
        sizeof(PMRect));

        PBBinCommand(lpParamBlk,pm_color,kRSPointer,”Red”,4);

        // 4 = strlen(“Red”)+1

        return CQ_SUCCESS;

    }
```

See also: PBBinCommandByShortValue
PBBinCommandByLongValue
PBBinQuery
PBTextCommand
PBTextQuery

PBBinCommandByShortValue

```
PMErr PBBinCommandByShortValue(lpPB,op,v)
```

Issues a command to PageMaker using the binary format. Use this macro only for commands that have a single, short parameter.

Type	Parameter	Description
sPMParamBlockPtr	lpPB	Pointer to the parameter block
short	op	Command operation code, or binary ID (e.g., pm_open)
short	v	Parameter

Return value: CQ_SUCCESS or a non-zero return code.

Binary ID: The command operation code, or binary ID, is the command name preceded by pm_. For example, the binary ID for the Close command is pm_close. Each binary ID is also listed in the header file PageMakerCommands.h.

Commands with varying number of parameters: Use this macro only for commands with a single, short parameter. Do not use it for commands for which the number of parameters can vary (e.g., Tabs). If a command generally has more than one parameter, PageMaker always expects a pointer or handle to the parameters, even if you specify the kRSValue constant in the abReplyData.rsStyle field.

Example

```
/**[m****************************************************
    *RedrawOn
    *DESCRIPTION:
    *This turns redraw back on. It will also update any outstanding
    *regions that need to be redrawn.
 **m]****************************************************/
    RC  RedrawOn(sPMParamBlockPtr lpParamBlk)
    {
    PBBinCommandByShortValue(lpParamBlk, pm_redraw, 1);
    return CQ_SUCCESS;
    }
```

See also: PBBinCommand
PBBinCommandByLongValue
PBBinQuery
PBTextCommand
PBTextQuery

PBBinCommandByLongValue

```
PMErr PBBinCommandByLongValue(lpPB,op,lv)
```

Issues a command to PageMaker using the binary format. Use this macro only for commands that have a single, long parameter.

Type	Parameter	Description
sPMParamBlockPtr	lpPB	Pointer to the parameter block
short	op	Command operation code, or binary ID (e.g., pm_open)
long	lv	Parameter

Return value: CQ_SUCCESS or a non-zero return code.

Binary ID: The command operation code, or binary ID, is the command name preceded by pm_. For example, the binary ID for the Close command is pm_close. Each binary ID is also listed in the header file PageMakerCommands.h.

Commands with varying number of parameters: Use this macro only for commands with a single, long parameter. Do not use it for commands for which the number of parameters can vary. If a command generally has more than one parameter, PageMaker always expects a pointer or handle to the parameters, even if you specify the kRSValue constant in the abReplyData.rsStyle field.

See also: PBBinCommand
PBBinCommandByShortValue
PBBinQuery
PBTextCommand
PBTextQuery

PBBinQuery

```
PMErr PBBinQuery(lpPB,op,rsy,r,rsz)
```

Issues a query to PageMaker using the binary format. Use this macro for queries that have no associated parameters.

Type	Parameter	Description
sPMParamBlockPtr	lpPB	Pointer to the parameter block
short	op	Query operation code, or binary ID (e.g., pm_open)
short	rsy	Reference style for the r parameter (if null, PageMaker allocates the buffer for the reply data)
pointer or handle	r	Pointer or handle (as specified by the rsy parameter) to the buffer for reply data (null if rsy is null)
short	rsz	Maximum size of reply data (NULL if r is a handle)

Return value: CQ_SUCCESS or a non-zero return code.

Binary ID: The query operation code, or binary ID, is the query name preceded by pm_. For example, the binary ID for the GetLineStyle query is pm_getlinestyle. Each binary ID is also listed in the header file PageMakerCommands.h.

See also: PBBinQueryWithParms
"Sending queries to PageMaker" in Chapter 3

PBBinQueryWithParms

```
PMErr PBBinQueryWithParms(lpPB,op,sy,hp,sz,rsy,r,rsz)
```

Issues a query to PageMaker using the binary format. Use this macro for queries that have parameters.

Type	Parameter	Description
sPMParamBlockPtr	lpPB	Pointer to the parameter block
short	op	Query operation code, or binary ID (e.g., pm_open)
short	sy	Reference style of the hp parameter, either kRSPointer or kRSHandle
pointer or handle	hp	Pointer or handle (as specified by the sy parameter) to the values for the parameter block
short	sz	Size of data referenced by the hp parameter
short	rsy	Reference style for the r parameter (if NULL, PageMaker allocates the buffer for the reply data)
pointer or handle	r	Pointer or handle (as specified by the rsy parameter) to the reply values (NULL if rsy is NULL)
short	rsz	Maximum size of reply data for pointer (NULL if r is a handle)

Return value: CQ_SUCCESS or a non-zero return code.

Binary ID: The query operation code, or binary ID, is the query name preceded by pm_. For example, the binary ID for the GetLineStyle query is pm_getlinestyle. Each binary ID is also listed in the header file PageMakerCommands.h.

See also: PBBinQuery
"Sending queries to PageMaker" in Chapter 3

PBClearReplyBlock

```
void PBClearReplyBlock(lpPB)
```

Clears the reply fields of the parameter block, namely: abReplyData.lpData, abReplyData.ulSize, abReplyData.reStyle, abReplyData.pmuUnits.

Type	Parameter	Description
sPMParamBlockPtr	lpPB	Pointer to the reply block

Return value: None

Used by other macros:
PBTextCommand
PBBinCommandByShortValue
PBBinCommandByLongValue
PBBinCommand

See also:
PBClearRequestBlock
PBTextCommand
PBBinCommandByShortValue
PBBinCommandByLongValue
PBBinCommand

PBClearRequestBlock

```
void PBClearRequestBlock(lpPB)
```

Clears the request fields of the parameter block, namely: abRequestData.lpData, abRequestData.ulSize, abRequestData.reStyle, abRequestData.pmuUnits.

Type	Parameter	Description
sPMParamBlockPtr	lpPB	Pointer to the parameter block

Return value: None

Used by other macros: PBBinQuery

See also: PBClearReplyBlock
PBBinQuery

PBGetPluginData

```
HANDLE PBGetPluginData(lpPB)
```

Gets the handle to the plug-in’s data from the pluginData field of the parameter block.

Type	Parameter	Description
sPMParamBlockPtr	lpPB	Pointer to the parameter block

Return value: Handle to the plug-in’s data.

Example

```
PMErr  DoLoad(sPMParamBlockPtr lpParamBlk)
    {
    HANDLE hMyData = PBGetPluginData(lpParamBlk);
    short *ps;
    if (!hMyData) { // NULL if plug-in's never been called.
        //----------------------------
        // Allocate pluginData:
        // Maintains last count of boxes across a page, used by BoxTest
        //example.
        //----------------------------
    hMyData = MMAlloc(SIZE_OF_DATA);
    PBSetPluginData(lpParamBlk, hMyData);
        //----------------------------
        // Initialize the data handle.
        //----------------------------
    ps = (short*)MMLock(hMyData);
    *ps = DEF_NUM_BOXES; // default number of boxes for dialog.
    MMUnlock(hMyData);
    }
    return CQ_SUCCESS;
    }
```

See also: PBSetPluginData
“Parameter block structure” in Chapter 3

PBGetID

```
ULONG PBGetID(lpPB)
```

Gets the plug-in identifier (function ID) from the ulSubCode field of the parameter block. Use this macro during invocation if the plug-in library contains more than one plug-in.

Type	Parameter	Description
sPMParamBlockPtr	lpPB	Pointer to the parameter block

Return value: The function ID of the requested plug-in.

Example

```
PMErr  DoInvoke(sPMParamBlockPtr lpParamBlk)
    {
    switch( PBGetId(lpParamBlk) )
    {
    case kPubInfo:
        PMErr = PlacePubInfo(lpParamBlk);
        break;
    case kView100:
         PMErr = View100(lpParamBlk);
        break;
    default:
        PMErr = RC_FAILURE;
         break;
    }
    return CQ_SUCCESS;
    }
```

See also: “Registration” in Chapter 2

PBGetOpCode

```
PMOPCODE PBGetOpCode(lpPB)
```

Gets the constant for the requested operation from the opCode field of the parameter block.

Type	Parameter	Description
sPMParamBlockPtr	lpPB	Pointer to the parameter block

Return value: One of the following constants:
kPMLoad
kPMInvoke
kPMUnload
kPMCleanup
kPMShutdown

Example

```
pascal PMErr main(sPMParamBlockPtr lpParamBlk)

    {

    PMErr rc;

    switch (PBGetOpCode(lpParamBlk)) {

        case kPMLoad:

             PMErr = DoLoad(lpParamBlk);

             break;

        case kPMUnload:

             PMErr = DoUnload(lpParamBlk);

             break;

        case kPMCleanup:

             PMErr = DoCleanup(lpParamBlk);

             break;

        case kPMShutdown:

             PMErr = DoShutdown(lpParamBlk);

             break;

        case kPMInvoke:

             PMErr = DoInvoke(lpParamBlk);

             break;

        default:

             PMErr = RC_FAILURE;

             break;

    }

    return rc;

    }
```

See also: PBSetOpCode

PBGetReplyData

```
LPVOID PBGetReplyData(lpPB)
```

Extracts the data reference from the reply section of the parameter block.

Type	Parameter	Description
sPMParamBlockPtr	lpPB	Pointer to parameter block

Return value: A pointer, handle, or value (as specified in the reference style in abReplyData.rsStyle)

lpData: abReplyData.lpdata can contain a value (if the data can fit in the 4-byte field) or a handle or pointer to the data.

PBSetErrMessage

```
void PBSetErrMessage(lpPB, hMsg)
```

Puts the handle to the plug-in's error-message string into the hszErrMessage field of the parameter block.

Type	Parameter	Description
sPMParamBlockPtr	lpPB	Pointer to the parameter block
HANDLE	hMsg	Handle to the null-terminated error-message string

Return value: None

Non-zero return code required: PageMaker displays the error message if the plug-in returns a non-zero return code from an operation. See "Error and status codes" in Chapter 2 for information on the format of the error string.

See also: "Parameter block structure" in Chapter 3

PBSetPluginData

```
void PBSetPluginData(lpPB, h)
```

Puts the handle to the plug-in’s data into the pluginData field of the parameter block.

Type	Parameter	Description
sPMParamBlockPtr	lpPB	Pointer to the parameter block
HANDLE	h	Handle for the plug-in’s data

Return value: None

Use only if plug-in needs data between use: Use this macro when the plug-needs to maintain data between invocations. The data is not preserved when PageMaker shuts down.

Example

```
PMErr DoLoad(sPMParamBlockPtr lpParamBlk)
{
    HANDLE hMyData = PBGetPluginData(lpParamBlk);
    short *ps;
    if (!hMyData) { // NULL if plug-in's never been called.
    //----------------------------
    // Allocate pluginData:
    // Maintains last count of boxes across a page, used by BoxTest
    //example.
    //----------------------------
    hMyData = MMAlloc(SIZE_OF_DATA);
    PBSetPluginData(lpParamBlk, hMyData);
    //----------------------------
    // Initialize the data handle.
    //----------------------------
    ps = (short*)MMLock(hMyData);
    *ps = DEF_NUM_BOXES; // default number of boxes for dialog.
        MMUnlock(hMyData);
    }
    return CQ_SUCCESS;
}
```

See also: PBGetPluginData
“Parameter block structure” in Chapter 3

PBSetOpCode

```
void PBSetOpCode(lpPB, op)
```

Puts the constant for the requested operation into the opCode field of the parameter block.

Type	Parameter	Description
sPMParamBlockPtr	lpPB	Pointer to the parameter block
short	op	Constant for the desired operation

Return value: None

Used by other macros:
PBBinCommand
PBBinCommandByShortValue
PBBinCommandByLongValue
PBBinQuery
PBBinQueryWithParms

See also:
PBGetOpCode
PBBinCommand
PBBinCommandByShortValue
PBBinCommandByLongValue
PBBinQuery
PBBinQueryWithParms

PBSetReplyBlock

```
void PBSetReplyBlock(lpPB, rs, pv, sz)
```

Puts the abReplyData values in the reply fields of the parameter block, specifically: abReplyData.lpData, abReplyData.ulSize, and abReplyData.reStyle.

Type	Parameter	Description
sPMParamBlockPtr	lpPB	Pointer to the parameter block
short	rs	Reference style of the pv parameter, either kRSPointer or kRSHandle
pointer or handle	pv	Pointer or handle (as specified by the rs parameter) to the reply data
short	sz	Maximum size of reply data

Return value: None

Used by other macros:
PBClearReplyBlock
PBTextQuery
PBBinQuery
PBBinQueryWithParms.

See also:
PBClearReplyBlock
PBSetReplyUnits
PBTextQuery
PBBinQuery
PBBinQueryWithParms
“Sending queries to PageMaker” in Chapter 3

PBSetReplyUnits

```
void PBSetReplyUnits(lpPB, u)
```

Puts the unit constant to be used for reply data into the abReplyData.pmuUnits field of the parameter block (for replies in the text format only).

Type	Parameter	Description
sPMParamBlockPtr	lpPB	Pointer to the parameter block
short	u	Unit constant for reply data: kMUNull kMUInches kMUDecInches kMUCentimeters kMUPicas kMUPoints

Return value: None

Units for text format only: PBSetReplyUnits sets the reply units for replies in the text format only. Binary replies are always in twips, PageMaker's internal measurement system.

Example

```
PMErr RunStory(sPMParamBlockPtr lpParamBlk)
    {
    RC      rcVal = CQ_SUCCESS;/* return code */
    USHORT wStyle;
    LPSTR lpStory;
    SHORT arParms[2];
    /* set up request and reply units to be the same.*/
    PBSetRequestUnits(lpParamBlk,kMUInches);
    PBSetReplyUnits(lpParamBlk, kMUInches);
    /* set up specifics for getStoryText query.*/
    arParms[0] = 0; /* get raw text */
    arParms[1] = 3; /* substitute for non-printing chars */
    if (!(PBBinQueryWithParms(lpParamBlk, pm_getstorytext, kRSPointer,
        arParms, sizeof(arParms), kRSPointer, NULL, MAXSTORYSIZE)) &&
      (lpStory = PBGetReplyData(lpParamBlk)))
        {
        /* use buffer supplied by PM to hold text but don't purge it.*/
        wStyle = ((PBGetReplyRefStyle(lpParamBlk)) & kRefStyleMask) |
            kRFCantPurge;
        PBSetRequestRefStyle(lpParamBlk, wStyle);
        }
    else
        {
```

```
        // do something else, no selection.
    }
/* send story back as text commands.*/
if (!rcVal)
    PBTextCommand(lpParamBlk, kRSPointer, lpStory);
return (rcVal);
}
```

See also: PBSetReplyBlock
“Sending queries to PageMaker” in Chapter 3

PBSetRequestBlock

```
void PBSetRequestBlock(lpPB, rs, pv, sz)
```

Puts the abReplyData values in the reply fields of the parameter block, specifically: abRequestData.lpData, abRequestData.ulSize, and abRequestData.reStyle.

Type	Parameter	Description
sPMParamBlockPtr	lpPB	Pointer to the parameter block
short	rs	Reference style of the pv parameter: kRSPointer or kRSHandle
pointer or handle	pv	Pointer or handle (depending upon the rs parameter) to the request data
short	sz	Size of the request data

Return value: None

Used by other macros: PBClearRequestBlock
PBBinCommand
PBBinCommandByShortValue
PBTextCommand
PBTextQuery

See also: PBClearRequestBlock
PBBinCommandByShortValue
PBBinCommand
PBSetRequestUnits
PBTextCommand
PBTextQuery
“Sending commands to PageMaker” in Chapter 3
“Sending queries to PageMaker” in Chapter 3

PBSetRequestUnits

```
void PBSetRequestUnits(lpPB, u)
```

Puts the unit constant to be used for request data into the abRequestData.pmuUnits field of the parameter block (for requests in the text format only).

Type	Parameter	Description
sPMParamBlockPtr	lpPB	Pointer to the parameter block
short	u	Unit constants for request data: kMUNull kMUInches kMUDecInches kMUCentimeters kMUPicas kMUPoints

Return value: None

Units for text format only: PBSetRequestUnits sets the request units for replies in the text format only. Binary requests are always in twips, PageMaker's internal measurement system.

Example

```
PMErr RunStory(sPMParamBlockPtr lpParamBlk)
    {
    RC      rcVal = CQ_SUCCESS;/* return code */
    USHORT  wStyle;
    LPSTR   lpStory;
    SHORT   arParms[2];
    /* set up request and reply units to be the same.*/
    PBSetRequestUnits(lpParamBlk,kMUInches);
    PBSetReplyUnits(lpParamBlk, kMUInches);
    /* set up specifics for getStoryText query.*/
    arParms[0] = 0; /* get raw text */
    arParms[1] = 3; /* substitute for non-printing chars */
    if (!(PBBinQueryWithParms(lpParamBlk, pm_getstorytext, kRSPointer,
        arParms, sizeof(arParms), kRSPointer, NULL, MAXSTORYSIZE)) &&
        (lpStory = PBGetReplyData(lpParamBlk)))
        {
        /* use buffer supplied by PM to hold text but don't purge it.*/
        wStyle = ((PBGetReplyRefStyle(lpParamBlk)) & kRefStyleMask) |
            kRFCantPurge;
        PBSetRequestRefStyle(lpParamBlk, wStyle);
        }
    else
            {
            // do something else, no selection.
        }
    /* send story back as text commands.*/
    if (!rcVal)
        PBTextCommand(lpParamBlk, kRSPointer, lpStory);
    return (rcVal);
    }
```

See also: PBSetRequestBlock
“Sending commands to PageMaker” in Chapter 3
“Sending queries to PageMaker” in Chapter 3

PBTextCommand

`PMErr PBTextCommand(lpPB, sy, tx)`

Issues a command to PageMaker using the text format.

Type	Parameter	Description
sPMParamBlockPtr	lpPB	Pointer to the parameter block
short	sy	Reference style of the tx parameter, either kRSPointer or kRSHandle
pointer or handle	tx	Pointer or handle (as specified by the sy parameter) to a null-terminated string containing the values for the parameter block

Return value: None

Example

```
PMErr RunStory(sPMParamBlockPtr lpParamBlk)
    {
    RC      rcVal = CQ_SUCCESS; /* return code */
    USHORT  wStyle;
    LPSTR   lpStory;
    SHORT   arParms[2];
    /* set up request and reply units to be the same.*/
    PBSetRequestUnits(lpParamBlk,kMUInches);
    PBSetReplyUnits(lpParamBlk, kMUInches);
    /* set up specifics for getStoryText query.*/
    arParms[0] = 0; /* get raw text */
    arParms[1] = 3; /* substitute for non-printing chars */
    if (!(PBBinQueryWithParms(lpParamBlk, pm_getstorytext, kRSPointer,
        arParms, sizeof(arParms), kRSPointer, NULL, MAXSTORYSIZE)) &&
        (lpStory = PBGetReplyData(lpParamBlk)))
        {
        /* use buffer supplied by PM to hold text but don't purge it.*/
        wStyle = ((PBGetReplyRefStyle(lpParamBlk)) & kRefStyleMask) |
            kRFCantPurge;
        PBSetRequestRefStyle(lpParamBlk, wStyle);
        }
    else
        {
            // do something else, no selection.
        }
    /* send story back as text commands.
     */
    if (!rcVal)
        PBTextCommand(lpParamBlk, kRSPointer, lpStory);
    return (rcVal);
}
```

See also: PBBinCommand
PBBinCommandByShortValue
PBBinCommandByLongValue
PBBinQuery
PBTextQuery

PBTextQuery

```
PMErr PBTextQuery(lpPB,sy,tx,rsy,r,rsz)
```

Issues a query to PageMaker using the text format.

Type	Parameter	Description
sPMParamBlockPtr	lpPB	Pointer to the parameter block
short	sy	Reference style for the tx parameter, either kRSPointer or kRSHandle
pointer or handle	tx	Pointer or handle (as specified by the sy parameter) to a null-terminated string that contains the values for the parameter block
short	rsy	Requested style for reply
pointer or handle	r	Pointer or handle to the reply data block
short	rsz	Maximum size of query reply data

Return value: None

Getting replies in the text format: The binary format is the default reply format for text queries, and the only format for replies from binary queries. When using the text format, you can request that the replies be in the text format by including a flag, kXRFText, with the reply style constant in abReplyData.rsStyle. (kXRFText is defined in PageMakerTypes.h.)

Example

```
flag = kRSHandle | kXRFText // Pass back by handle as ASCII text
```

If you include kXRFText with a binary query, PageMaker ignores your request. (Note that in cases where the actual return value of a binary query is text—such as a publication name, a font name, or story text—PageMaker returns a null-terminated string.) The examples below illustrate how to specify formats for return values.

```
rcVal = PBTextQuery(lpParamBlk,  //Gets object list for current page
kRSPointer, “getobjectlist”, //By default, return is in binary
kRSHandle, NULL, MAXSIZE);  //format for text and binary queries

rcVal = PBTextQuery(lpParamBlk, //Gets object list for current page
kRSPointer, “getobjectlist”,
kRSHandle | kRFBinary,  //specifies binary format for results
NULL, MAXSIZE);

rcVal = PBTextQuery(lpParamBlk, //Gets object list for current page
kRSPointer, “getobjectlist”,
kRSHandle | kXRFText,    //specifies text format for results
NULL, MAXSIZE);

rcVal = PBBinQuery(lpParamBlk, //Binary query reply is always in twips
 pm_getobjectlist,
 kRSHandle | kXRFText,  //PageMaker ignores kXRFText flag
 NULL, MAXSIZE);
```

See also: PBBinCommand
PBBinCommandByShortValue
PBBinCommandByLongValue
PBBinQuery
PBTextCommand
"Sending queries to PageMaker" in Chapter 3

6

User Interface Design Guidelines

In general, the user interface design of your plug-in is completely up to you. However, we recommend using PageMaker as a model. By basing your plug-in interface on the PageMaker interface, you visually tie your plug-in and its functions more closely to PageMaker and provide a more consistent working environment for users.

This chapter outlines some of the guidelines we used to develop the PageMaker interface. This chapter also provides some general development tips you may want to use, especially if your users may be working with multiple plug-ins.

For more interface development information, refer to *Inside Macintosh* by Apple Computer, Inc., and the “Windows User Interface Guidelines” in *SDK Guide for Microsoft Windows and Windows NT*.

General development tips

The following tips may help you design and implement a more efficient interface for your plug-in.

Simplify the plug-ins menu

Remember that the PageMaker Plug-ins menu can become quite long, depending on the number of plug-ins the user installs. Here are some tips to help keep the menu uncluttered:

- If you plan to write several related plug-ins, we recommend that you not list each plug-in separately on the plug-ins menu. Instead, combine them into a single plug-in and let the user access the various options in a dialog box. For example, if you plan to write a set of plug-ins that create automatic headers and footers, don't list "Automatic header…" and "Automatic footer…" on the PageMaker Plug-ins submenu. Instead, use just one command, "Automatic Headers and Footers…" that opens into a dialog box containing all available options for automating headers and footers.
- If you plan to write several plug-ins that perform unrelated functions, put them in separate plug-ins. That way, users can install only those plug-ins they need. (If all your plug-ins are in the same file, the user must install all the plug-ins.)

Create consistent icons

To maintain a consistent desktop interface with other plug-ins, base icons for your plug-ins on the PageMaker plug-ins icon.

Follow existing standards and usage

Take advantage of the user's knowledge of existing methods for performing operations and conventions for naming menu items.

- Support existing or de facto standard shortcuts:

Item	Windows	Macintosh
Cut	Control+X	Command+X
Copy	Control+C	Command+C
Paste	Control+V	Command+V
Undo	Control+Z	Command+Z
Cancel	Esc	Command+. (period)
Print	Control+P	Command+P
Quit	Control+Q	Command+Q
New	Control+N	Command+N
Open	Control+O	Command+O
Save	Control+S	Command+S
Place	Control+D	Command+D

Designing dialog boxes

As a rule, all dialog boxes must have:

- A title.
- An “OK” or action (for example, “Print”) button.
- A “Cancel” button.

In general, the rest of the dialog box design is up to you. However, to provide a consistent interface (and one with which the user is already familiar), we recommend that you make your plug-in dialog boxes look as similar to PageMaker’s as possible.

Common dialog boxes

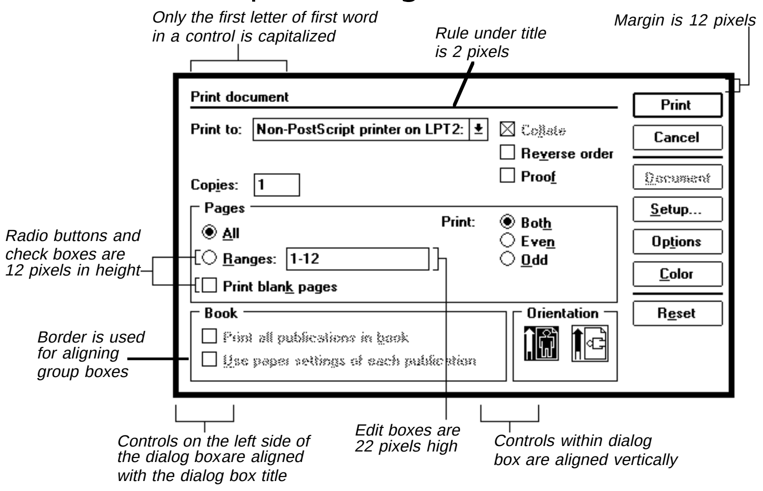
Where possible, use the common dialog boxes provided in Windows and on the Macintosh. Both platforms offer standard dialog boxes for opening and saving files.

Note: For Windows common dialog boxes, allow space for the button added when Workgroup for Windows is active.

PageMaker dialog box guidelines

Here are the guidelines we use for developing dialog boxes:

- Position dialog boxes and alerts 3/10 of the way down the Macintosh screen and 2/5 of the way down the PC screen.
- Use Chicago 12 and Geneva 9 for text on the Macintosh and Helv 8 and Helv 7 for text on the PC. Typically, the smaller font sizes are used for labels, group box titles, and message lines in complex dialog boxes. See the Links dialog box in PageMaker for an example.
- Capitalize only the first letter of the first word in a dialog box title, option name, control group name, or group box title.
- Run a 2-pixel rule from the left edge of the title to 18 pixels from the left edge of the OK button.
- Leave 6 pixels between the baseline of the title and the top of the rule, and 12 pixels between the bottom of the rule and the top of the first letter of the option immediately below.
- Leave a 12-pixel margin on all sides.


Only the first letter of first word in a control is capitalized
Rule under title is 2 pixels
Margin is 12 pixels
Radio buttons and check boxes are 12 pixels in height
Border is used for aligning group boxes
Controls on the left side of the dialog boxare aligned with the dialog box title
Edit boxes are 22 pixels high
Controls within dialog box are aligned vertically
Print
Cancel
Document
Setup...
Options
Color
Reset

Print document

Print to: Non-PostScript printer on LPT2:

☒ Collate
☐ Reverse order
☐ Proof

Copies: 1

Pages
● All
○ Ranges: 1-12
☐ Print blank pages

Print: ● Both ○ Even ○ Odd

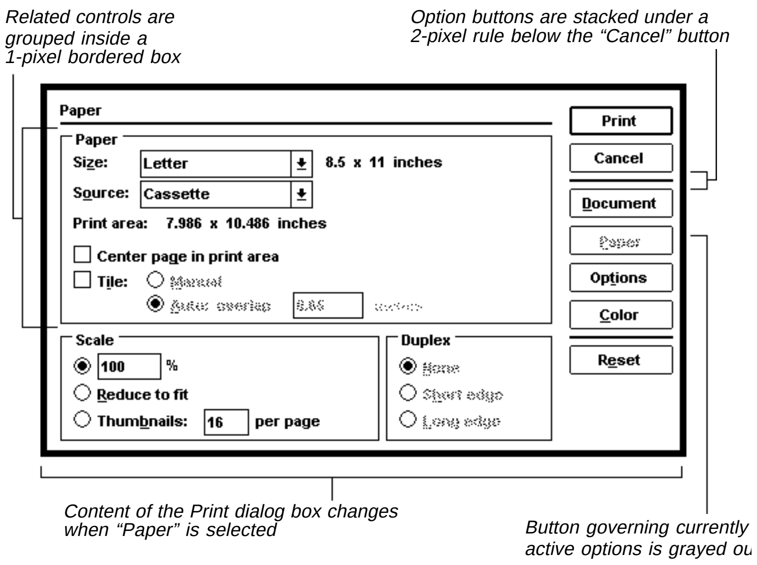
Book
☐ Print all publications in book
☐ Use paper settings of each publication

Orientation

About pixels

Pixels—as defined by the 13" Macintosh screen or a VGA screen for the PC—are used as the measurement system to describe spacing guidelines. The purpose in being so specific is to aid in determining control placement. However, perceived distance may be preferable to pixel specification, especially for higher resolution screens. So use what's reasonable; for example, 12 pixels or approximately 3/16 of an inch.

- Offset nested dialogs down and right from the dialog that invoked them. Typically, it's better to cover the parent dialog's OK and Cancel buttons (or similar controls) to minimize confusion for the user.
- Use selectable dialog boxes to present groups of related options that can't all fit in one dialog box. For example, the PageMaker 6.0 Print dialog box contains a list of push-buttons along the right side. Selecting a button causes the related items to appear in the dialog box.


Related controls are grouped inside a 1-pixel bordered box

Option buttons are stacked under a 2-pixel rule below the "Cancel" button

Paper

Paper
Size: Letter 8.5 x 11 inches
Source: Cassette
Print area: 7.986 x 10.486 inches
☐ Center page in print area
☐ Tile: ○ Manual
● Auto: overlap 0.65 inches

Scale
● 100 %
○ Reduce to fit
○ Thumbnails: 16 per page

Duplex
● None
○ Short edge
○ Long edge

Content of the Print dialog box changes when "Paper" is selected

Button governing currently active options is grayed ou

Creating and placing buttons

Here are the guidelines we use for button design:

- Center button names within the button, leaving at least 4 pixels between the button name and the left and right edges of the button.
- Make the "OK" or action button (for example, "Print") the default button (activated when the user presses the Return or Enter key). However, message boxes or alerts may designate "Cancel" as the default when the operation could cause data loss, for example, saving over an existing document.
- Use the initial letter in the button as its mnemonic when possible.
- Reinforce the default action button visually with a bold border. On the Macintosh, its border consists of a 1-pixel rule inside and a 2-pixel rule outside separated by a 2-pixel space; in Windows, its border is defined by the standard DEFPUSHBUTTON style.
- Set a 1-pixel border around standard (non-default) buttons.
- Make all buttons 19 pixels high, measured from the 1-pixel border.
- Position the "OK" or action button in the upper-right corner of the dialog box, directly above the "Cancel" button.
- Separate other buttons from the "Cancel" button by stacking them under a rule that runs the width of the buttons. Use the standard 1-pixel dotted rule on the Macintosh, and a 2-pixel line in Windows.
- Make all buttons the width of the widest non-default button.
- Align buttons along the right side with the main dismissal or "Cancel" button.
- Gray out the button currently governing the content of a selectable dialog box.

Radio buttons, check boxes, and edit boxes

The Windows and Macintosh platforms provide standard-sized controls for most dialog box options. If no standards are provided or if you may determine the size of options, use the following guidelines:

- Include a descriptive title above or to the side of all logical sets of controls (e.g., radio button groups).
- Organize related controls within titled boxes. Use a 1-pixel black border on the Macintosh, and the standard Windows group box on Windows.
- Make radio buttons 12 pixels in diameter and check boxes 12 pixels square.
- Size edit boxes to 22 pixels high and wide enough to accommodate the maximum number of characters the user can enter. When text in the box is highlighted, separate the edge of the highlight from the edit box border with 2 pixels of space.
- Use a 1-pixel border for radio buttons, check boxes, and edit boxes.

Option placement guidelines

Here are the guidelines we use for positioning controls within the dialog box:

- Align controls and control labels in the left portion of the dialog box with the title.
- Arrange groups horizontally.
- Align options along vertical lines.
- Run in the group name with its options, unless the group name is very long or the option names vary greatly in length. In that case, indent the option names under the group name.
- Left-align all options in a group.

Note: For boxed groups, use the border, rather than the boxed items, to left-align with the title and other controls.

Error messages and alerts

Error messages are posted when the user enters invalid data or when the settings in a dialog box are not within defined ranges. Try to give the user as much help in solving the problem as possible. Here are some tips for building dialogs that offer user feedback as well as conform to PageMaker standards:

- Trap errors when the user OKs the dialog box, not when the user is navigating between options within a dialog box. Though this is the preferred method, you may need to trap errors immediately in certain cases, for example, when values in controls are calculated based on settings in other controls.
- Construct error messages and alerts in two parts: a description of the problem or condition and a statement containing suggested corrective action or required state (e.g., “Cannot complete Drop-cap operation. Click text tool in a textblock and try again.”)
- Include a “Continue” button rather than an “OK” button when applicable. For example, for this message, “Cannot initialize language dictionaries. No languages supported for hyphenation,” “Continue” is more appropriate than “Cancel.”
- Return to a highlighted field containing the invalid value after the user accepts or cancels the error message.
- Select “Cancel” as the default for potentially destructive actions. For example, for this message “Delete page and all items on it?” with its options for “OK” and “Cancel,” “Cancel” is the default.
- List the acceptable range for illegal values or indicate the source of the problem in the error message text. For example, “-1 not a valid measurement. Enter a number between 1 and 16.”
- Adopt the measurement system in effect when the error is flagged. For example, an error message would read “Enter a number between 1 and 3 inches” when the selected measurement system is inches and “Enter a number between 30 and 600 mm” when in millimeters.
- Avoid using words like “fatal” or “bad.” Use “serious” or “important” instead.
- Point user to help when necessary. For example, when corrective action requires significant instruction, direct the user to the documentation or to online help.

Guidelines for making your product easy to localize

PageMaker is distributed in over 20 languages.

A successfully localized product provides more than the ability to translate text into other languages. Consideration of cultural differences up front can ease the localization process considerably. Fortunately, the fundamentals of planning for localizing Windows products are almost identical to planning for localizing Macintosh products, so you can transfer what you learn when localizing on one platform to the process on the other platform.

Localizing Windows plug-ins

If you plan to translate your Windows plug-in into different languages, you can help simplify localization by putting strings in a single resource file. Tracking the status of one file is much easier than keeping tabs on several files, each containing different groups of string definitions.

Note: Windows 3.1 and Windows 95 provide resources and language-sensitive functions that help ensure that your application behaves as expected in localized versions of Windows.

Organizing your resources

Keep functional code and strings separate. Hard coding strings makes it impossible to localize without generating a new executable. Instead, use resources for any information that needs to be modified. On the Macintosh, the .RSRC contains these items. For Windows applications, the .RC and .DLG files are used.

Content of resources

Some items requiring localization are obvious, for example, text in menus and dialog boxes. However, there are additional issues to consider, such as date format and icons for certain objects. You may discover others during your localization process. Apple has a “Internationalization Checklist” document for developers. In addition, the Windows SDK contains a section on the localization process. You may want to consult these documents for more information.

Here is a list of items that you should put in resources.

- All text visible on the screen, including menu items, dialog text, error messages, status line or help information, undo strings, titles and items listed in palettes, and conditionally displayed items for menus and dialog boxes (e.g., strings for the PageMaker “Place” dialog box change depending on the type of file selected).
- Quotation marks or other punctuation and special characters
- Shortcuts and accelerators

  Note: When providing translator tables for shortcuts, keep in mind that the international keyboards have keys in different places than the U.S. keyboards so that the locations of the VK_OEM keys may change depending on the keyboard layout chosen by the user.
- Number formats and separators (e.g., commas and decimal marks)
- Currency symbols
- Short and long formats for dates (e.g., 6/22/93 and June 16, 1993) and provision for European and Non-Gregorian calendars
- Time formats and descriptors (e.g., 12 hr, 24 hr, AM/PM)

- Address formats, including ZIP codes and phone numbers
- Lengths of strings and text resources
- Text translator tables for character, word, and phrases
- Units of measure
- Graphics and icons, including position and size. Items will change position and size as they get translated, so do not hard code the location or extent of any element in the window.
- Word and character boundaries

  Note: These delimiters are used in search and replace, sorting, word wrap, selection, backspacing and delete, and cut and paste operations. If your plug-in does not perform these operations, you won't need these items in the resource.

Providing for translated text

As you create dialog boxes, keep in mind that translated items will almost certainly grow in size, possibly in all directions. For example, diacritical marks are widely used outside the United States and may extend up to the ascent line, "Ü" and "É," or down to the descent line, "Ç."

Note: Potential grammar issues may affect the size of error messages, so keep them flexible.

- Design dialog boxes and other elements to give text room to grow up, down, and sideways. Use this rule of thumb to allocate extra space for strings:

For this number of English characters	Allocate this much additional space
1-10	200%
11-20	100%
21-30	80%
31-50	60%
51-70	40%
70 and above	30%

Note: Do not put text inside an icon. It's unlikely that translated text will fit within the confines of an icon.

Special considerations

Here are some other items to take into account.

- Determine appropriate settings and use them as the defaults. For example, page size should default to A4 in products intended for European English markets.
- Right-to-left and mixed-direction text can cause problems in justification, cursor positioning, highlighting.
- Restrict use of <Command><Space> (and arrow keys) for Command-key equivalents since these are used to select keyboards.

Writing Stand-Alone Plug-ins: Apple Events and DDE

PageMaker can receive commands and queries via two System 7 Apple Events on the Macintosh and via Dynamic Data Exchange (DDE) under Windows on the PC. You use Apple Events and DDE to send commands and queries from a stand-alone plug-in. This chapter describes how to use Apple Events and DDE to communicate with PageMaker.

Note: We assume you're familiar with the way Apple Events and the Apple Event manager work. If you're not, refer to *Inside Macintosh, Interapplication Communication* by Apple Computer, Inc. before reading this chapter.

We also assume you are familiar with the way DDE works. If not, refer to the *SDK Guide for Microsoft Windows and Windows NT*.

About stand-alone plug-ins

A stand-alone plug-in is an independent application that sends commands and queries to PageMaker via Apple Events or Windows DDE. A stand-alone plug-in uses the same commands and queries other plug-ins use, the only difference is that you send them from an external source. This chapter does not attempt to tell you how to write a stand-alone plug-in. Instead, it tells you what you need to know to allow your application to communicate with PageMaker.

A major advantage of a stand-alone plug-in is that it lets you integrate the functionality of multiple applications. For example, you could write a stand-alone plug-in that automates the production of a catalog. It could open a database application, extract any new product descriptions from the catalog database, and then open PageMaker, locate the previous descriptions in the catalog, and replace both the descriptions and the images.

Using Apple Events to communicate with PageMaker

Unlike some applications, PageMaker does not distinguish between commands and queries at the Apple Event level. Therefore, you can use either of the two supported events—Do Script and Evaluate Expression—to send commands and queries to PageMaker. (PageMaker supports both events to provide compatibility with applications that cannot use these standard Apple Events interchangeably.)

Addressing events to PageMaker

In general, you can use several methods to address Apple Events to a specific application: the application signature, the session ID, the target ID, or the process serial number. The approach depends on the application you're using. To learn about each method and determine which best suits your needs, refer to Inside Macintosh, Interapplication Communication.

Note: PageMaker's application signature is "ALD6."

Required constants

To communicate with PageMaker using AESend, your plug-in must identify these constants:

Constant	Description
kAEMiscStdSuite = 'misc'	Miscellaneous standard suite
kAEDoScript = 'dosc'	Standard DoScript event (or you can identify kAEEvaluate below; both are not required)
kAEEvaluate = 'eval'	Standard Eval event (or you can identify kAEDoScript above; both are not required)
keyAEDirectParameter = '----'	Direct parameter of AEDescriptor
keyErrorNumber = 'erno'	Error number returned from PageMaker
keyErrorString = 'errs'	Error string returned from PageMaker
typeText = 'TEXT'	Raw text data
typeLongInteger = 'long'	Long integer

Sample code

To see how Apple Events send commands or queries to PageMaker, review the sample code included in this SDK.

Do script and Evaluate Expression Apple Events

Description

The Do script and Evaluate Expression events send both commands and queries to PageMaker.

Note: PageMaker does not distinguish between the Do Script and Evaluate Expression events. PageMaker recognizes both events to accommodate applications that require both.

Event	Do Script	Evaluate Expression
Event Class	kAEMiscStdSuite='misc'	kAEMiscStdSuite='misc'
Event ID	kAEDoScript='dosc'	kAEEvaluate='eval'
Keyword	Descriptor type	Description

Event	Do Script	Evaluate Expression
keyAEDirectParameter ='----'	typeText= 'TEXT'	Required; commands and/or query
Reply Keyword	Descriptor type	Description
keyAEDirectParameter='----'	typeText='TEXT'	PageMaker reply values from a query

Error	Description
errAECantCoerce	Could not coerce the object into the required type
errAENoSuchObject	Can't find the object referred to by the direct parameter
errAEFail	General failure of event
customErr	An error returned by PageMaker when a command or query failed

Sending commands and queries

When you send commands and queries to PageMaker via an Apple Event:

- Send commands and queries as plain text encapsulated in a descriptor. PageMaker 6.0 does not support file aliases and object specifiers.
- Send only one query per event. If you send more than one query per event, only the result of the last query is returned in the reply. You can send numerous commands in an event.
- Separate commands and queries with a semicolon (;) or new line character (carriage return). You need not terminate the set of commands and query with a null character, but can if you prefer.

Note: It is possible to send events to PageMaker faster than PageMaker can process them. In that case, events may be ignored. To circumvent the problem, you should always specify a kAEWaitReply in the sendMode parameter of AESend, even if your plug-in is only sending commands.

PageMaker’s reply to a query

PageMaker uses the reply Apple Event to provide query results or error information to the application sending the Apple Event. Replies may contain:

- A query reply (keyAEDirectParameter) if the plug-in sent a query.
- An error number (keyErrorNumber).
- An error string (keyErrorString) if PageMaker can generate a string for that error code.

Note: If an error occurs, the reply may also contain invalid data in keyAEDirectParameter. To verify whether the data is valid, make sure the reply contains keyAEDirectParameter and does not contain the keyErrorNumber parameter.

A successful query reply in keyAEDirectParameter:

- Is a null-terminated character string.
- Uses a comma to separate values.

HyperCard or SuperCard example

The following HyperCard or SuperCard example creates a simple utility that threads (joins) the text of two independent text blocks into one story and then replaces the second text block in its original position. The utility consists of a button, which sends the commands and queries to select, join, and replace the text, and a simple text field, where user instructions are displayed.

To use this threading utility, you should have a publication open in PageMaker with at least two stories on the page.

Note: You must use HyperCard version 2.1 or later. Also, program linking must be active in the Sharing Setup control panel for this example to work.

Stack or project script

The following function is the stack or project script for the utility:

```
--Threading utility courtesy David Butler

function sendQueryToPM pmscript

    global PMAPP

    -- put PageMaker name into variable PMAPP

    if PMAPP is empty then

        answer program "Select PageMaker from list on right:"

        if it is empty then exit sendQueryToPM

        put it into PMAPP

    end if

    request pmscript from program PMAPP

    return it

end sendQueryToPM
```

Button

The utility has one button named Thread. The script for the button is:

```
on mouseUp

-- Get coordinates of selected text block

-- Use coordinates later to place text back on page

    put sendQueryToPM("getobjectloc topleft") into TLCoords

    put sendQueryToPM("getobjectloc bottomright") into BRCoords


-- Highlight and cut text from second text block

-- Select first text block

    put sendQueryToPM("textedit;selectall;cut;select 1;") into reply


-- Get bottom corner of first text block

    put sendQueryToPM("getobjectloc bottomright") into BCd


-- Get last character of first text block

    put sendQueryToPM("textedit;textcursor +textblock;textselect -char") into
reply

    put sendQueryToPM("getstorytext 0 0") into reply


-- If last character is not a return, add one

    If character 2 of reply is return then
```

```
        put "textcursor +textblock;" into TxtSend
    else
        put "textcursor +textblock;textenter " & quote & return & quote & ";"
into TxtSend
    end if

-- Paste text and reposition text blocks
    put "paste;select 1;resize bottomright" && BCd & ";" after TxtSend
    put "placenext;place" && TLCoords & ";" after TxtSend
    put "resize topleft" && TLCoords & ";" after TxtSend
    put "resize bottomright" && BRCoords & ";" after TxtSend
    put sendQueryToPM(TxtSend) into reply
end mouseUp
```

Text field

The utility has one text field that contains the following instructions for the user:

To thread two text blocks, select the first text block and send it to the back. Then, select the second block and click Thread.

Using DDE to communicate with PageMaker

You can communicate with PageMaker by sending DDE messages directly to PageMaker from a stand-alone plug-in or from any application that supports DDE commands.

You can also create a stand-alone plug-in that calls the routines contained in the Windows 95 DDE Manager Library (DDEML). The examples in this chapter illustrate both methods of using DDE to communicate with PageMaker.

Using DDE messages

PageMaker recognizes the first four DDE messages listed below and sends the last two DDE messages:

WM_DDE_INITIATE:	Use this message to begin a conversation. PageMaker registers itself as "PageMaker." It responds to WM_DDE_INITIATE messages for "PageMaker" that use any topic name (including NULL).
WM_DDE_ EXECUTE:	Use this message to send commands. Results are processed as text strings.
WM_DDE_ REQUEST:	Use this message to send queries. Queries must use the CF_TEXT format. (For more information, see SDK Guide for Microsoft Windows and Windows NT.) Results are processed as text strings.
WM_DDE_ TERMINATE:	Use this message to end a conversation.
WM_DDE_DATA:	PageMaker uses this message to transmit the result of the query to the application that issued the WM_DDE_REQUEST.
WM_DDE_ACK:	PageMaker uses this message to acknowledge the receipt of a command.

Sending commands and queries to PageMaker

As described above, you use WM_DDE_EXECUTE to send commands and WM_DDE_REQUEST to send queries.

You should also follow these general guidelines when sending commands and queries to PageMaker:

- Separate multiple commands or queries within the message by new line characters or by semicolons (;) For example:

  ```
  new; close
  select(5,6)
  ```

- Send no more than one query per message. There can be only one text reply from a DDE query. If you send more than one query per event, only the result of the last query is returned in the reply.

Receiving replies from PageMaker

PageMaker uses WM_DDE_DATA to transmit the result of the query to the application that issued the WM_DDE_REQUEST. A query reply is a null-terminated character string. Items in a reply are separated by commas (,).

Example: Calling routines in DDE Manager Library

This example illustrates how to establish a callback routine, initiate a conversation, send a command and a query, and end a conversation with PageMaker by calling routines in the Windows 95 DDE Manager Library (DDEML).

The callback routine for DDEML

To use DDEML, you must first write a callback function. MyDdeCallback is registered on DdeInitialize and called during various DDEML functions. Remember to export this function in the .DEF file of your application.

```
 static PFNCALLBACK lpCallback; // Procedure instance for
// callback function.
HDDEDATA CALLBACK MyDdeCallback(UINT type, UINT fmt, HCONV hconv, HSZ
hsz1,
HSZ hsz2, HDDEDATA hData, DWORD dwData1, DWORD dwData2)
{
switch (type) {
default: // No need to handle
return 0; // any callback types.
}
}
```

Initiating a DDE conversation

StartPMConv initiates a DDE conversation with PageMaker and returns the handle that you use to send commands or queries to PageMaker. Once you initiate the conversation, you can send multiple commands and queries to PageMaker.

```
 HCONV StartPMConv(void)
 {
 DWORD dwInst = 0; // Instance identifier for DDEML.
 HSZ hszService; // String handle for PageMaker service name.
 HCONV hconv; // Handle to PageMaker DDE conversation.

 lpCallback = // Make procedure instance
 (PFNCALLBACK)MakeProcInstance( // for the callback function.
 (FARPROC)MyDdeCallback,
 hInst);

 DdeInitialize(&dwInst, // Initialize the DDEML.
 lpCallback,
 APPCLASS_STANDARD | APPCMD_CLIENTONLY,
 0);

 hszService = // Make a string handle for the
 DdeCreateStringHandle(dwInst, // service name.
 “PageMaker”, CP_WINANSI);
 hconv = DdeConnect(dwInst, // Establish a DDE conversation.
 hszService, 0, 0);
 DdeFreeStringHandle(dwInst, // Free the memory associated with
 hszService); // the service name.

 return hconv;
}
```

Sending a command to PageMaker

The following routine sends a ‘\0’ terminated string command to PageMaker.

```
 void SendPMCommand(HCONV hconv, LPSTR lpCmd)
 {
 DWORD dwResult; // DDE conversation status.
 DdeClientTransaction(lpCmd, // Send the command string
 lstrlen(lpCmd) + 1, // to PageMaker. hconv, 0, CF_TEXT,
 XTYP_EXECUTE, 1000,
 &dwResult);
 }
```

Sending a query to PageMaker

The following routine sends a query string to PageMaker and displays the results of the query in a Windows message box.

```
void SendPMQuery(HCONV hconv, LPSTR lpQuery)
{
HDDEDATA hReplyData;
HSZ hszQuery;
hszQuery = // Create a string handle
DdeCreateStringHandle(dwInst, // for the query.
lpQuery, CP_WINANSI);
hReplyData = // Send the command line argument
DdeClientTransaction(0, // to PageMaker as a query.
0, hconv, hszQuery,
CF_TEXT, XTYP_REQUEST,
1000, &dwResult);
DdeFreeStringHandle(dwInst, hszQuery);
if (hReplyData) {
DWORD dwReplySize;
LPSTR lpReply;
lpReply = // Get a pointer to the
DdeAccessData(hReplyData, // reply handle.
&dwReplySize);
MessageBox(0, lpReply, // Show it in a message “PM Query Result”, MB_OK); //
dialog box.
DdeUnaccessData(hReplyData); // Done with the data.
DdeFreeDataHandle(hReplyData);
} else {
MessageBox(0, “No reply from
PageMaker”, “Error”, MB_OK);

}

}
```

Ending the conversation

When the conversation is complete, call EndPMConv to terminate the DDE conversation.

```
void EndPMConv(HCONV hconv)

{

DdeDisconnect(hconv); // End conversation.

DdeUninitialize(dwInst); // Allow DDEML to clean up.

FreeProcInstance((FARPROC)lpCallback); // Free the procedure

}// instance for the

// callback function.
```

Example: Sending DDE messages

You can also communicate with PageMaker by sending DDE messages directly to PageMaker from a stand-alone plug-in or from within another application. This concept is illustrated by the following Visual Basic program.

This Visual Basic program creates a simple utility that threads (joins) the text of two independent text blocks into one story and then replaces the second text block in its original position. The utility form consists of a button, which sends the commands and queries to select, join, and replace the text, and a simple text window, where user instructions are displayed and replies from PageMaker are sent.

To use this threading utility, you should have a publication open in PageMaker with at least two stories on the page.

Declarations

Here are the "(general)" declarations for the utility:

```
REM Threading utility courtesy David Butler

REM Subroutine to keep utility on top

Declare Sub SetWindowPos Lib "User" (ByVal hWnd As Integer, ByVal
hWndInsertAfter As Integer, ByVal X As Integer, ByVal Y As Integer, ByVal cx
As Integer, ByVal cy As Integer, ByVal wFlags As Integer)


Const HWND_TOPMOST = -1

Const HWND_NOTOPMOST = -2

Const SWP_NOACTIVATE = &H10

Const SWP_SHOWWINDOW = &H40
```

Subroutines

Here are the subroutines used by the utility:

```
Sub Form_Load ()

REM Make window stay on top of PageMaker

SetWindowPos hWnd, HWND_TOPMOST, 0, 0, 0, 0, SWP_NOACTIVATE Or SWP_SHOWWINDOW


REM  Prevent utility from timing out if PageMaker is not running

Text1.LinkTimeout = -1
```

```
Text1.LinkTopic = "PageMaker|DDE_LINK"

REM Put help message in text window
UpdateStatus

End Sub

Sub RunScriptCommand (PM_Cmd As String)
Text1.LinkMode = 2

REM Send either commands or query based on first 3 characters
REM You can group commands, but must send queries one by one
REM Use Execute for commands, Request for queries

    If Left$(LCase$(PM_Cmd), 3) = "get" Then
    Text1.LinkItem = PM_Cmd
    Text1.LinkRequest
    Else
    Text1.LinkExecute PM_Cmd
    End If

End Sub

Sub UpdateStatus ()
REM Define help text to appear in text window
    Msg$ = "To thread two text blocks, select the first text block and send to
back. "
    Msg$ = Msg$ + "Then, select the second block and click Thread."
    Text1.Text = Msg$

End Sub
```

Text field

The utility has one text field with the MultiLine property set to True. The LinkClose procedure contains the following code:

```
Sub Text1_LinkClose ()

REM Let PageMaker finish before utility continues
REM This procedure is important for more complex scripts
DoEvents

End Sub
```

Command button

The utility has one command button with a caption of Thread. The subroutine for the button follows. Be careful to follow the PageMaker syntax correctly (for example, inserting a space between commands and parameters).

The sample code below sends several commands at a time. If it becomes necessary to troubleshoot a problem, you may want to send one command at a time.

```
Sub Command1_Click ()

REM Define a paragraph (carriage return) character
Cr$ = Chr$(34) + Chr$(13) + Chr$(10) + Chr$(34)

REM Get coordinates of selected text block
REM Use coordinates later to place text back on page
RunScriptCommand ("getobjectloc topleft")
TLCoord$ = Text1.Text
RunScriptCommand ("getobjectloc bottomright")
BRCoord$ = Text1.Text

REM Highlight and cut text in second text block
REM Then select first text block
RunScriptCommand ("textedit;selectall;cut;select 1;")

REM Get bottom corner of first text block
RunScriptCommand ("getobjectloc bottomright")
BCd$ = Text1.Text

REM Get last character of first text block
RunScriptCommand ("textedit;textcursor +textblock;textselect -char;")

REM If last character is not a return, add one
RunScriptCommand ("getstorytext 0 0")
If Asc(Mid$(Text1.Text, 2)) <> 13 Then
    Msg$ = "textcursor +textblock;textenter " & Cr$ & ";"
Else
    Msg$ = "textcursor +textblock;"
End If
REM Paste text and reposition text blocks
Msg$ = Msg$ + "paste;select 1;resize bottomright " + BCd$ + ";"
Msg$ = Msg$ + "placenext;place " + TLCoord$ + ";"
Msg$ = Msg$ + "resize topleft " + TLCoord$ + ";"
Msg$ = Msg$ + "resize bottomright " + BRCoord$ + ";"

RunScriptCommand (Msg$)

REM Put help message back in text window
UpdateStatus

End Sub
```

8 Using Commands and Queries

The PageMaker command and query language consists of simple commands and queries based on PageMaker menu commands or mouse actions.
This chapter describes how to use the command and query language to write plug-ins. For information on specific commands and queries, see Chapters 9 and 10.

Command and query language

In PageMaker's command and query language, commands are one-word equivalents of menu, keyboard, or mouse actions. For example, the commands for File > Open, Edit > Paste, and Type > Set Width are Open, Paste, and SetWidth, respectively.

Queries ask questions about the PageMaker publication and use the same one-word approach that commands use. Queries always begin with "get," such as GetColumnGuides or GetCropRect.

Using commands and queries is the easiest part of writing a plug-in. Just think through the logical sequence of the task you want to complete and find the appropriate command or query in Chapters 9 and 10. For commands, it's a good idea to do the steps using the PageMaker menus, and then jot down the commands as you do them.

The command and query language is simpler than most programming or macro languages. It includes no programming control structures (such as "if-then-else" statements) or loops (such as "repeat-until" statements).

Text versus binary format: to parse or not to parse

Plug-ins can send commands and queries either as text or in the binary format. (Internal scripts must always use text.) Because PageMaker does not need to parse binary code, binary commands and queries require less memory to process and execute faster than text commands and queries.

Query replies: if you want text, you gotta ask

Loadable plug-ins

For loadable plug-ins, the default format of query replies is the binary format, regardless of the format of the query from the plug-in. You can receive query replies as text only if you:

- Use the text format for the query, and
- Include the kXRFText flag in the abReplyData.rsStyle field of the parameter block (e.g., flag = kRSHandle | kRFText—see "Sending queries to PageMaker" in Chapter 3).

Binary queries, however, only return results in the binary format. If you specify the text format for binary query returns, your specification will be ignored. (Note that in cases where the actual return value is text—such as a publication name, a font name, or story text—the return value of a binary query is a null-terminated string.) For more information, see "Sending queries to PageMaker" in Chapter 3.

Stand-alone plug-ins

For plug-ins communicating with PageMaker using Apple Events or Window DDE messages, regardless of the format of the query, PageMaker returns the results of the query as a null-terminated string. For replies that include several values, PageMaker separates each value with a comma.

Binary format

Add PM_ to command and query names

When sending commands and queries in the binary format, add PM_ to the command or query name.

Even byte boundaries

Binary data must begin on even byte boundaries. With the exception of string data, all fields are evenly sized, so this is generally not a problem.

String data

Strings are variable-length text runs (character arrays). Therefore, to designate the end of the field, string values must be null-terminated. To start the field following a string on an even boundary, the pointer may need to skip a byte.

Routines to put and get data (loadable plug-ins)

To facilitate putting data in and retrieving data from the appropriate addresses, we recommend that you use a routine that automatically increments the pointer, adjusting the address as needed for string data to ensure that the pointer lands on an even byte. The sample routines below demonstrate one approach. The SDK includes macros to perform these functions as well. See Chapter 5, “Macros” for more information.

These routines copy data to the appropriate buffer and increment the pointer accordingly. The put routines need a pointer to the buffer pointer and the value (the text-buffer address in the case of string values). The get routines need a pointer to the buffer pointer and a pointer to the destination (the text-buffer address in the case of string values).

It is important that you use the correct macro for each command or query you send. For example, if you are using the binary format and the query requires parameters, you use PBBinQueryWithParms, not PBBinQuery. In the same way, if a command has several parameters, you use PBBinCommand, but if it has only one short parameter, you use PBBinByShortValue.

```
void CQPUTSHORT(short **, short);
void CQPUTSTRING(char **, char *);
void CQGETSHORT(short *, short **);
void CQGETSTRING(char **, char **);
void TestPut(LPPARAMBLOCK);
void CQPUTSHORT(dest,src)
short **dest;
short src;
{
    **dest = src;  /* copy the value */
    (*dest)++;  /* increment the pointer */
}
void CQPUTSTRING(dest,src)
char **dest;
char *src;
{
    char *dstptr = *dest;
    unsigned long  addr;
    while(*src) *dstptr++ = *src++;
    *dstptr++ = '\0';
    addr = (unsigned long) dstptr;
    if (addr & 1L) dstptr++;  /* pad if odd boundary */
    *dest = dstptr;
```

```
}
void CQGETSHORT(dest,src)
short *dest, **src;
{
    *dest = **src;  /* copy the value */
    (*src)++;  /* inc the pointer */
}
void CQGETSTRING(dest,src)
char **dest, **src;
{
    char *srcptr = *src;
    unsigned long  addr;
    *dest = *src;  /* get string address set up */
    while(*srcptr) srcptr++;  /* skip past the string */
    srcptr++;  /* skip null */
    addr = (unsigned long) srcptr;
    if (addr & 1L) srcptr++;  /* skip pad if odd boundary */
    *src = srcptr;
}
void TestPut(LPPARAMBLOCK lpParamBlk)
    {
    /*-----------  sample usage ----------- */
/****Adding publications to the book list*****/
short renumOption = 3; /* next even */
short numofPub = 3; /* number of publications */
char psnamebuf[20]; /* publication filenames */
char CommandPacket[100];
char *pPacket = CommandPacket;
char *pBook;
RC rc;
short j;
HANDLE hpx;
    /* some variable pPacket is a pointer to our output buffer */
CQPUTSHORT(&pPacket, renumOption); /* put copies into the buffer */
CQPUTSHORT(&pPacket,numofPub); /* put firstpage into the buffer */
    /* put in publication names */
strcpy (psnamebuf, "HardDisk:Pub 1");
CQPUTSTRING(&pPacket,psnamebuf); /* put the filename string */
strcpy (psnamebuf, "HardDisk:Pub 11");
CQPUTSTRING(&pPacket,psnamebuf); /* put the filename string */
strcpy (psnamebuf, "HardDisk:Pub 111");
CQPUTSTRING(&pPacket,psnamebuf); /* put the filename string */
    /* send out the command */
rc = PBBinCommand(lpParamBlk, pm_book, kRSPointer, CommandPacket,
        strlen(CommandPacket));
    pPacket = CommandPacket;
    for (j=0; j < sizeof(CommandPacket); j++)
        *pPacket++ = 0;
    pBook = CommandPacket;
rc = PBBinQuery(lpParamBlk, pm_getbook, kRSHandle, 0, 0);
    if (hpx = PBGetReplyData(lpParamBlk)) {
        if (pPacket = MMLock(hpx)) {
            CQGETSHORT(&renumOption,&pPacket);
            CQGETSHORT(&numofPub,&pPacket);
            for (j=0; j < numofPub; j++)
                CQGETSTRING(&pBook,&pPacket);
                    MMUnlock(hpx);
            }
            MMFree(hpx);
      }
      }
```

64K query return limit for Windows

The return from a query is limited to 64K. You can work around the 64K limit if your plug-ins requests a handle to the query results and then processes the text with a HUGE pointer.

Optional parameters not optional

When sending a binary packet, all parameters are required, so you should disregard the optional brackets in the command and query descriptions in Chapters 9 and 10.

Deleting reply buffers

PageMaker always allocates a handle for reply results. If the query fails, PageMaker cannot delete the handle it allocated because it can no longer distinguish that handle from any handle you might have allocated. To ensure that the reply buffer is deleted whenever it’s appropriate, we recommend that you send queries using the following method:

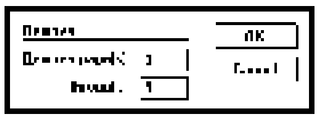
```
rcVal = PBBinQueryWithParms(lpParamBlk,pm_getstorytext,kRSPointer,
arParms, sizeof(arParms),kRSHandle,NULL,MAX_REPLY_SIZE);
myHandle = PBGetReplyData(lpParamBlk);
if (!rcVal && myHandle) {
CatHandle(&h, “Success:[“);
CatHandle(&h, MMLock(myHandle));
MMUnlock(myHandle);
lsprintf(buff, “]\n”);
CatHandle(&h, buff);
} else {
lsprintf(buff, “Failed:%d\n”,rcVal);
CatHandle(&h, buff);
}
if (myHandle) MMFree(myHandle);
```

Commands, queries, and parameters

To duplicate dialog box options or mouse actions, many commands and a few queries include parameters. (Dialog boxes are suppressed when you use commands.) Parameters correspond directly to the options in the related dialog box or to page locations or object handles normally specified by dragging and clicking the mouse. For example:

```
removepages 3 5
```

In this example, RemovePages corresponds to the Layout > Remove Pages menu command. The first parameter (3) corresponds to the first page in the range of pages to be removed, and the second parameter (5) corresponds to the last page in the range of pages to be removed. The parameters are identical to the dialog box options shown below:



The parameters 3 and 5 in the previous example work just like the options in this dialog box.

Multiple commands for single menu commands

To keep the number of parameters to a manageable level, menu commands that open complex dialog boxes may be represented by several commands. For example, four script-language commands represent the File > Document Setup menu command:

- PageMargins
- PageNumbers
- PageOptions
- PageSize

The command you use depends on which options you want.

Command and query syntax

The order in which you specify parameter values for a command or query is listed on the top line of the description with the command or query name. You must specify parameter values in this order for PageMaker to correctly interpret the command.

Commands, defaults, and preferences

Be aware of the possible conditions, preferences, and default settings in a publication and on the computer, such as:

- Installed fonts, filters, and plug-ins
- Whether or not an object is selected
- Whether or not a publication is open

Otherwise, under certain circumstances, your plug-in may not run correctly or may yield undesirable results.

As when using the menus or mouse, the effect of a command or the values a query returns depends on the current state of PageMaker and the publication. Here are the effects of commands in each state:

No publication is open

If no publication is open, many commands set PageMaker default values for new publications, and many queries return the PageMaker default settings. The default settings in existing publications are not affected.

Publication is open and no object is selected

If a publication is open and nothing is selected (neither text blocks nor graphics), many commands set the publication defaults and many queries return the publication defaults.

Publication is open and an object (text block or graphic) is selected

If a publication is open and objects are selected with the Select command or pointer tool, object-specific commands and queries apply to those selected objects.

Publication is open and text is highlighted with the text tool

If a publication is open and text is selected with the text tool, text-specific commands and queries apply only to those selected sections of the text; paragraph-specific commands and queries apply to all the paragraphs containing the selected text.

Publication is open and the insertion point is within a text block

If a publication is open and the insertion point is within a text block, text-specific commands and queries apply only to the next characters inserted; paragraph-specific commands and queries apply to the paragraph containing the cursor.

Deciphering PageMaker's replies to queries

PageMaker returns information from a query as a string of numbers or words separated by commas. To decipher the reply, match the values that PageMaker returns with the table listing reply values in the query description in Chapter 10. For example, let's say PageMaker returns these values to the GetRuleAbove query:

```
1,1,"Blue-green",1,0,0,0,0
```

You simply match each value with its corresponding reply value listed in the description of GetRuleAbove.

Command and query language rules (text format only)

When you send a command or query, PageMaker checks each statement to see if the statement meets the requirements of the language. This section describes the basic rules you must follow when using the text format for commands and queries.

1 **Type each command or query as one word.**

For example, type “lockguides,” not “lock guides;” type “getlinkinfo,” not “get link info.”

2 **Use a semicolon or a carriage return to separate the command or query and its parameters from the next command or query. The semicolon and carriage return also mark the end of a comment (see rule 7 below).**

For example, both of the following examples are acceptable:

```
selectall

delete
```

or

```
selectall; delete
```

3 **Use commas, spaces, tabs, or parentheses ( ) to separate parameters from one another.**

All of the following examples are acceptable:

```
resize righttop, 3.5i, 7i, 1,1
```

or

```
resize righttop 3.5i 7i 1 1
```

or

```
resize righttop (3.5i, 7i) (1, 1)
```

4 **Use the correct syntax.**

Parameter values must always follow the command or query in the order specified in this SDK.

5 **Don’t worry about case when entering commands, queries, and parameter keywords. They can all contain lowercase and capital letters.**

All the following examples are acceptable:

```
ManualKerning ApartFine
```

or

```
manualkerning apartfine
```

or

```
MANUALKERNING Apartfine
```

6 **Always match the case, as well as spelling and punctuation, of submenu, pop-up menu, and palette options, such as fonts, colors, master pages, dictionaries, and styles. These parameters appear in quotation marks.**

You must capitalize, spell, and punctuate the option name exactly as it appears on-screen.

Only the first example is acceptable:

```
Font “HelveticaNeue Condensed”
```

not

```
Font “helveticaneue condensed”
```

7 **Precede all comments with a double hyphen (--). Comments may be either on a line by themselves or on the same line as a command or query.**

The double hyphens designate the beginning of a comment; a semicolon or carriage return designate the end of the comment. The following are all correct examples of comments:

```
new 5 --Creates a new, 5 page publication

box (2,2) (5,5) --Draws a 3" square box
```

or

```
new 5  --Creates a new, 5 page pub;box (2,2) (5,5) --Draws a 3" box
```

or

```
--The following creates a new, 5-page publication and draws a 3" box

new 5

box (2,2) (5,5)
```

Remember: Never include a semicolon as part of your comments, even within quotation marks. If you do, PageMaker will assume the semicolon marks the end of the comment and will try to interpret the remainder of the comment as a command.

Specifying the measurement system

Several commands and queries use measurements or ruler coordinates as parameters.

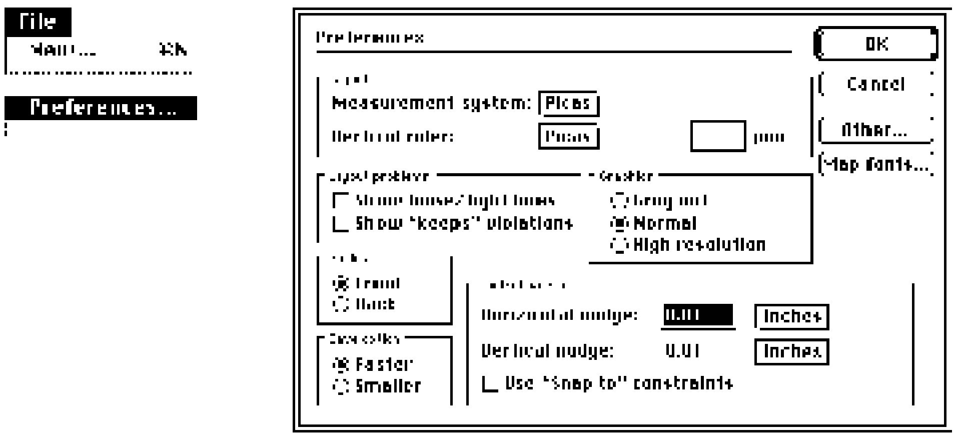
Binary format uses twips

When using the binary format, you cannot specify the measurement system you want to use or receive for measurements or coordinates. All measurements must be in twips, PageMaker's internal measurement system, regardless of the default measurement system in a publication. PageMaker also returns binary query data in twips.

Note: A twip is 1⁄20 of a PostScript point or 1⁄1440 of an inch. 240 twips equal 12 points (or 1 pica).

Text format

When using the text format, unless you specify another system, PageMaker uses the default measurement system (set in the Preferences dialog box) when interpreting the parameter values you specify or when returning query results.



Measurement system: Picas
Vertical ruler: Picas
Show "keeps" violations
Graphics: Normal
High resolution
Faster
Smaller
Horizontal nudge: 0.01 Inches
Vertical nudge: 0.01 Inches
Use "Snap to" constraints

The command and query language lets you either:

- Specify a system for an individual parameter, leaving the publication's default measurement system intact. This technique is useful if you don't know (or don't want to alter) the publication measurement system.
- Specify a new default measurement system for the publication (or, if no publications are open, for all future publications). This technique is useful if you want to use a particular measurement system for all commands and queries.

Specifying the measurement system for individual parameters

To override the default measurement system for a parameter, include a measurement abbreviation (from the table below) with the measurement or coordinate value. For example, "7i" in the command "move bottom, 7i" specifies a location 7 inches from the vertical ruler's zero point regardless of the default measurement system.

For example, if the publication measurement system is picas, PageMaker assumes the "7" in the following command means 7 picas:

```
move bottom 7
```

To specify inches instead of picas, add the inches abbreviation (listed in the following table) to the parameter:

```
move bottom, 7i
```

The following table lists the measurement abbreviations you can use to override the default measurement system:

System	Abbreviation	Example
Inches	i after	5.625i
Millimeters	m after	25m
Picas	p after	18p
Points	p before	p6
Picas and points	p between	18p6
Ciceros	c after number	5c

Note: Do not insert a space between the measurement and the abbreviation.

Changing the default measurement system

To change the measurement system for the publication, use the MeasureUnits command. The new measurement system becomes the default and remains in effect after the plug-in has run. PageMaker uses the default measurement system when interpreting measurements and coordinates in commands (unless overridden with a measurement abbreviation) and when returning measurements and coordinates from queries.

(For more information about MeasureUnits, see Chapter 9, “PageMaker commands.”)

Setting the zero point and specifying coordinates

For some commands and queries, you must use coordinates to specify locations on the page. When using the binary format, you specify coordinates in twips. When using the text format, you can specify coordinates either:

- Using numeric values (specified relative to the rulers' zero point), for example:

  ```
  move bottom, 7i
  ```

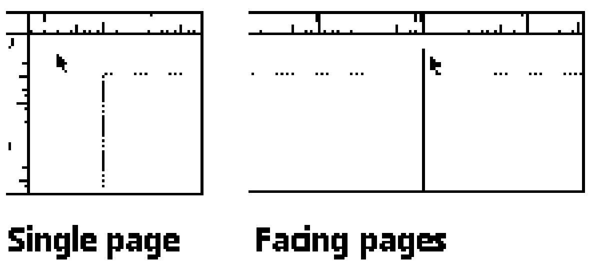
- In terms of page elements, such as columns, guides, and objects (more on these later in this section), for example:

  ```
  move bottom, column bottom
  ```

Whether using the binary or text format, numeric coordinates are specified relative to the ruler's zero point.

Setting the ruler's zero point

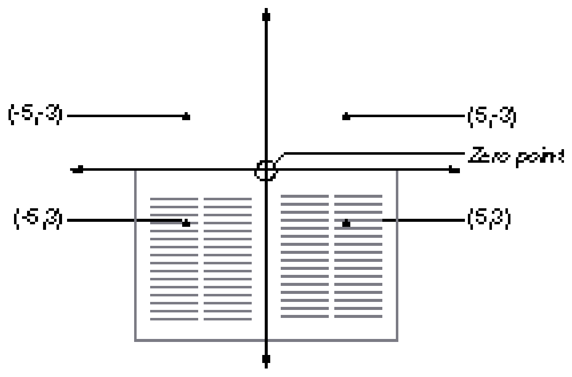
The default position of the zero point is at the upper-left corner of single pages and the upper-touching corners of two-page spreads, as shown in the following illustration:



The zero point is moveable and is often not in its default position. It's a good idea to explicitly set the zero point location to ensure that PageMaker places objects and guides where you want them, and to ensure that you understand the locations returned in query results. To position the zero point, use the ZeroPoint or ZeroPointReset command.

Using numeric coordinates

Numeric coordinates represent locations in relation to the PageMaker rulers. Coordinate values can be either negative or positive numbers, depending on the location of the rulers' zero point. However, unlike standard coordinates, PageMaker uses positive numbers to express locations down from the zero point. Locations above the zero point are expressed as negative numbers.



The following examples specify parameters using the numeric method:

Precise coordinate	Action
move top, 6i	Positions the selected object so its top edge is 6 inches below the zero point.
guidevert 4.25i	Creates a vertical ruler guide 4.25 inches to the right of the zero point.
deletevert 4.25i	Deletes the vertical guide that is located 4.25 inches to the right of the zero point.

Vertical coordinates do not use separate measurement system

Although PageMaker lets you specify a separate measurement system for the vertical ruler, all coordinates and measurements use the measurement system set in the Measurements In option (or with the cMeasurement parameter of the MeasureUnits command). For example, even if the vertical ruler is set to inches, PageMaker interprets any vertical coordinates or measurements using the default measurement system (which may not be inches). As mentioned previously, you can override the default system by including the abbreviation for the desired system with the value.

Specifying locations by page elements (text format only)

If you use the text format, you can specify locations in relation to elements on the page, rather than by precise numeric coordinates. The advantage of using page elements is that the locations remain valid even if you move the rulers' zero point, move an object, or change the publication page size or orientation.

To use this method, you refer to a column guide or object by the internal number PageMaker assigns it when it is first placed, typed, or drawn on the page.

- Columns are numbered from left to right on the specified page. You specify "leftpage" or "rightpage" (as shown in the table below) only when the publication has facing pages.
- Guides are numbered in the order in which they were placed on the page, regardless of their positions on the page. The first guide drawn is number one. If you delete a guide, PageMaker renumbers the remaining guides.
- Objects (text blocks and graphics) are numbered in the order in which they were first typed or drawn, regardless of position. The first object placed on the page is number one. Using the BringToFront, BringForward, SendBackward, or SendToBack commands changes its drawing order. If you delete an object, PageMaker renumbers the remaining objects.

The following table shows how to specify coordinates relative to columns, guides, and objects:

Location	x-coordinates	y-coordinates
Columns	column n left	column top column n right column bottom rightpage column n left leftpage column n left rightpage column n right leftpage column n right
Guides	guide n	guide n
Objects	last left	last top last right last bottom

Note:

- "n" in the column and guide references represents the column or guide number.
- "last" in the object descriptions refers to the edge of the last drawn object (the object with the highest drawing order).
- If you do not specify a location, PageMaker uses the right page by default.

Here are some examples of command parameter coordinates specified using page elements:

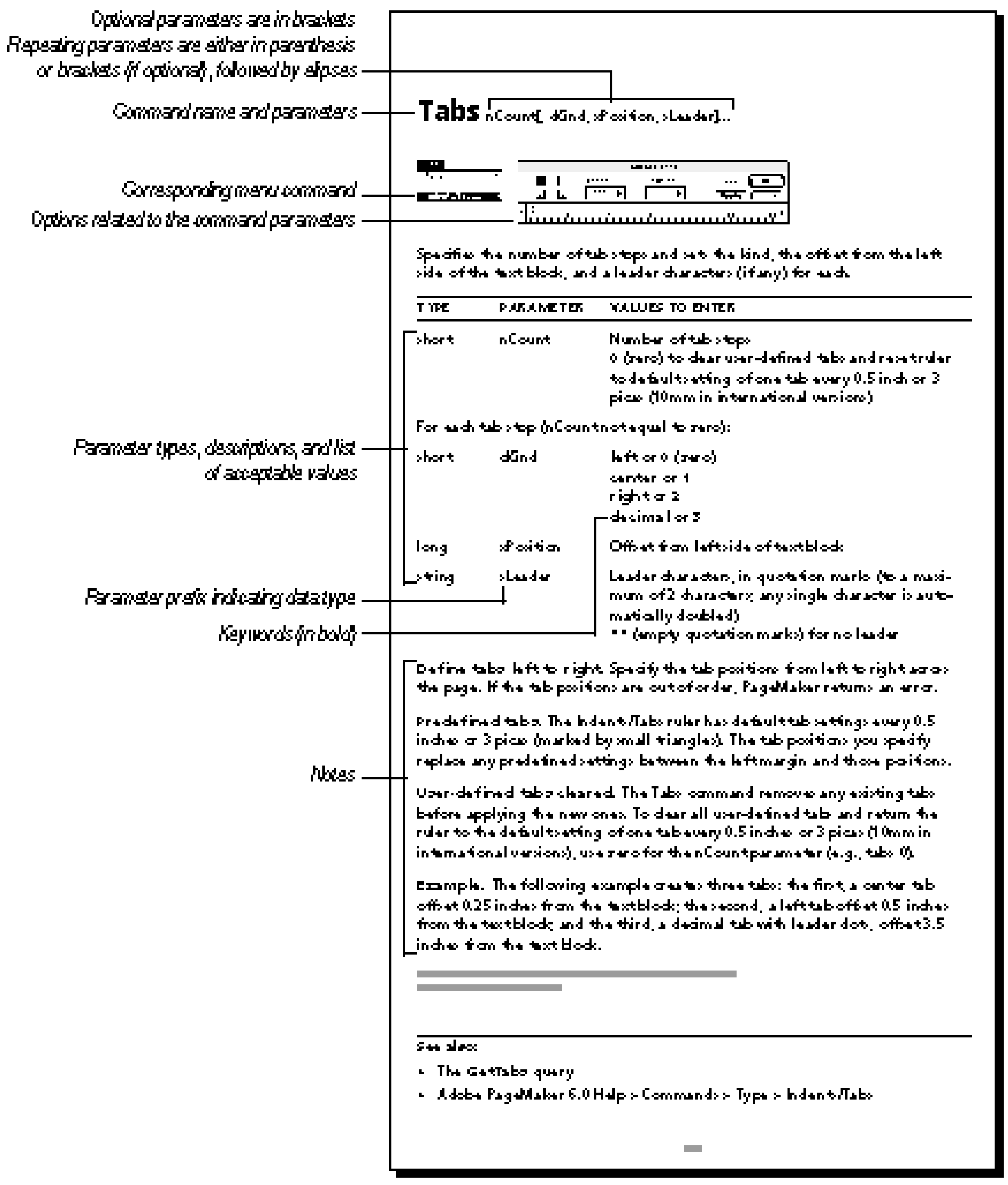
Command	Action
move left, guide 1	Positions the left edge of the selected object on the first guide drawn on the page.
deletevert guide 3	Deletes the third vertical ruler guide placed on the page (regardless of its location).
select (rightpage column 2 left, guide 2)	Selects the object that is on the right page, where the left side of the second column meets with second horizontal guide placed on the page.

Using the command and query reference

This section explains the content of the command and query descriptions in Chapters 9 and 10, and describes the conventions used in the command and query language and throughout the rest of this SDK.

Anatomy of the command and query descriptions

The commands are described in detail in Chapter 9; query descriptions are in Chapter 10. The illustration below shows an example of a command description and identifies the content:


Optional parameters are in brackets
Repeating parameters are either in parenthesis or brackets (if optional), followed by elipses
Command name and parameters
Corresponding menu command
Options related to the command parameters
Parameter types, descriptions, and list of acceptable values
Parameter prefix indicating data type
Keywords (in bold)
Notes

Tabs nCount[, dKind, xPosition, sLeader]...

Specifies the number of tab stops and sets the kind, the offset from the left side of the text block, and a leader characters (if any) for each.

TYPE	PARAMETER	VALUES TO ENTER
short	nCount	Number of tab stops 0 (zero) to clear user-defined tabs and reset ruler to default setting of one tab every 0.5 inch or 3 picas (10mm in international versions)
For each tab stop (nCount not equal to zero):		
short	dKind	left or 0 (zero) center or 1 right or 2 decimal or 3
long	xPosition	Offset from left side of text block
string	sLeader	Leader characters, in quotation marks (to a maximum of 2 characters; any single character is automatically doubled) "" (empty quotation marks) for no leader

Define tabs left to right. Specify the tab positions from left to right across the page. If the tab positions are out of order, PageMaker returns an error.

Predefined tabs. The Indents/Tabs ruler has default tab settings every 0.5 inches or 3 picas (marked by small triangles). The tab positions you specify replace any predefined settings between the left margin and those positions.

User-defined tabs cleared. The Tabs command removes any existing tabs before applying the new ones. To clear all user-defined tabs and return the ruler to the default setting of one tab every 0.5 inches or 3 picas (10mm in international versions), use zero for the nCount parameter (e.g., tabs 0).

Example. The following example creates three tabs: the first, a center tab offset 0.25 inches from the textblock; the second, a left tab offset 0.5 inches from the textblock; and the third, a decimal tab with leader dots, offset 3.5 inches from the text block.

See also:

- The GetTabs query
- Adobe PageMaker 6.0 Help > Commands > Type > Indents/Tabs

Parameter types

Just as a dialog box may include several types of options (pop-up menu, entry box, check boxes, radio buttons), the command and query language also requires different types of parameters:

- Numeric values
- Coordinates
- Filenames
- Submenu, pop-up menu, or palette choices
- Text

In the command and query descriptions in Chapters 9 and 10, each parameter name includes a lowercase prefix. The prefix indicates the type of value you can use or that PageMaker will return. The remainder of the name identifies the dialog box option, mouse action, object handle, and so forth, to which the parameter relates. The following table defines each parameter prefix and notes acceptable values:

Prefix	Data type	Description
b	boolean	Values are: true, on, 1 or false, off , 0. (In the binary format, use only 1 or 0.) Boolean parameters represent check boxes and options that you can turn on and off (such as the display of rulers, guides, or palettes). For example: `rulers on`

c	choice	`linestyle thindash`
		
		or
		
		`linestyle 13`
		
		Note: Do not enclose keywords in quotation marks.

d	decimal	Values are decimal numbers, generally accepted to one decimal point (for example, 6.2). In the binary format, you specify the value in the units of precision (in tenths of a percent or thousandths of a degree, for example). Decimal values specify point size, leading, page size, and so forth. For example: `size 13.5`

f	filename	Values are a filename. In the text format, the filename must appear in quotation marks. In the binary format, the parameter is a null-terminated string. For the best results, include the full path with the filename. Filenames are used with commands that refer to a file (such as, place, relink, or open). For example:
		
		`relink "MyDisk:Newsletter:Art:Chart.eps"`
		
		or
		
		`relink "c:\Newsltr\Art\Chart.eps"`
		
		Note: For dual-platform plug-ins: When referencing filenames, avoid upper-ASCII characters (character number 128 and up) . Although the first 128 characters are identical in the character sets used by Windows and the Macintosh, the upper-ASCII characters are not. This can cause a problem if the character on one platform maps to a character that is illegal for filenames on another.

Prefix	Data type	Description
n	number	Values are integers. Integers are used for page numbers, columns, new pages, and so forth. For example: `new 5`
s	string	Values are text. In the text format, the text must be in quotation marks (for example, “Blue-green”). In the binary format, the parameter is a null-terminated string. String parameters are used for entering text, page-number prefixes, and variable palette and submenu options, such as fonts, dictionaries, export filters, plug-ins, master pages, styles, and colors. For example: `font "Zapf Dingbats"` Where the string represents a variable palette, pop-up, or submenu option, you must capitalize, spell, and punctuate the option name exactly as it appears on-screen. **Note:** To include a quotation mark within the text, precede the quotation mark with a backslash, such as “\“Scripting is fun!\” said the script writer.” The quotation mark is the only character that requires special treatment. **Note:** For dual-platform plug-ins: While you can enclose strings in either typographer’s (curly) or straight quotation marks, we recommend that you use straight quotation marks if your plug-in is to be dual-platform.

x
y
	x-coordinate
	y-coordinate
		Values are coordinates. In the text format, you canspecify coordinates as either numeric locations (or offsets) or in reference to a guide, column edge, or the edge of the last object drawn. In the binary format, you specify numeric coordinates only, and only in twips. Coordinates identify a location on the page, an offset, or a relative position. (For details, “Setting the zero point and specifying coordinates” earlier in this chapter.) For example:
		`move lefttop 2p5 3p5`
		or
		`move lefttop (rightpage column 2 left, column top)`
		Note: For numeric coordinates: You generally specify numeric coordinates relative to the zero point. To ensure you know the location of the zero point, set it with either the ZeroPoint or ZeroPointReset commands. In the binary format, you specify numeric coordinates in twips. In the text format, you specify the coordinates using the current measurement system or by including the appropriate measurement identifier with the coordinate (such as, 3p6 for 3 picas, 6 points). See “Specifying the measurement system” earlier in this chapter.