Edition Manager

This chapter describes how you can use the Edition Manager to allow your users to share and automatically update data from numerous documents and applications.

The Edition Manager is available only in System 7 or later. It can be used by many different applications located on a single disk or throughout a network of Macintosh computers. To test for the existence of the Edition Manager, use the `Gestalt` function, described in *Inside Macintosh: Operating System Utilities*.

Read the information in this chapter if you want your application's documents to share and automatically update data, or if you want to share and automatically update data with documents created by other applications that support the Edition Manager.

For example, a user might want to capture sales figures and totals from within a spreadsheet and then include this information in a word-processing document that summarizes sales for a given month. The Edition Manager establishes a connection between these two documents. When a user modifies the spreadsheet, the information in the word-processing document can be automatically updated to contain the latest changes. To accomplish this, both the spreadsheet application and the word-processing application must support the features of the Edition Manager.

To use this chapter, you should be familiar with sending and receiving high-level events, described in the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials.* Your application must also support Apple events to receive Apple events from the Edition Manager. See the following chapters in this book for detailed information on Apple events.

The Edition Manager provides you with the ability to

- capture data within a document and integrate it into another document
- modify information in a document and automatically update any document that shares its data
- share information between applications on the same computer or across a network of Macintosh computers

Building the capabilities of the Edition Manager into your program is similar to building cut-and-paste features into your program. Text, graphics, spreadsheet cells, database reports—any data that you can select, you can make accessible to other applications that support the Edition Manager. The next section provides an overview of the main elements of the Edition Manager. Following sections discuss how to implement these features in your application.

This chapter also describes an advanced feature that allows applications to share data directly from a file.

Introduction to Publishers, Subscribers, and Editions

A ***section*** is a portion of a document that shares its contents with other documents. The Edition Manager supports two types of sections: publishers and subscribers. A ***publisher*** is a section within a document that makes its data available to other documents or applications. A ***subscriber*** is a section within a document that obtains its data from other documents or applications.

Your application writes a copy of the data in each publisher to a separate file called an ***edition container.*** The actual data that is written to the edition container is referred to as the ***edition.*** Your application obtains the data for each subscriber by reading data from the edition container. Note that throughout this chapter, the term *edition* refers to the edition container and the data it contains.

You ***publish*** data when you want to make it available to other documents and applications. When data is published, it is stored in an edition container. You ***subscribe*** to data that a publisher makes available by reading an edition from its container.

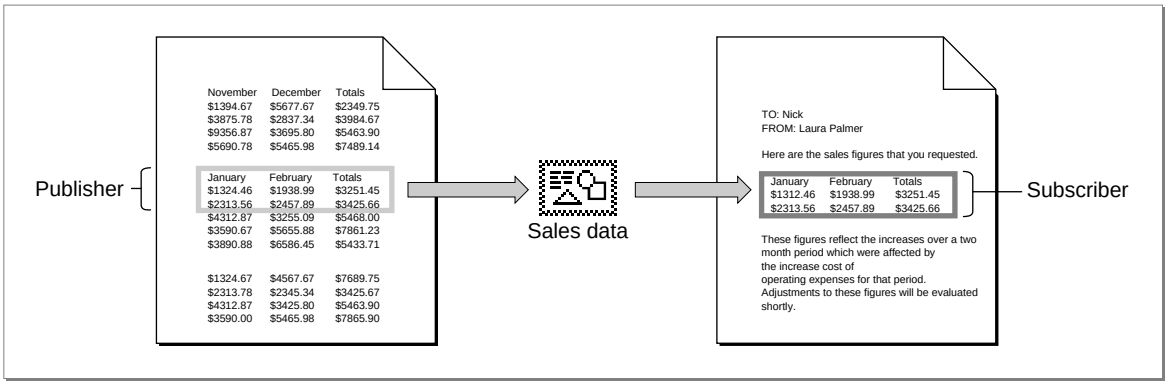
Note
Section and *edition container* are programmatic terms. You should not use them in your application or your documentation. Use *publishers, subscribers,* and *editions.* You should also refrain from using other terms such as *publication* or *subscription* to describe the dynamic sharing of information provided by the Edition Manager. Use the terms *publish* and *subscribe* to describe the Edition Manager features. ◆

Each edition has an icon that is visible from the Finder. Figure 2-1 shows the default edition icon.

Figure 2-1 The default edition icon

The name that the user specifies for the edition is located next to the edition icon. For information on providing icons for the editions created by your application, see the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials.* Figure 2-2 illustrates a document containing a single publisher, its corresponding edition, and a subscriber to the edition in another document.

Figure 2-2 A publisher, an edition, and a subscriber

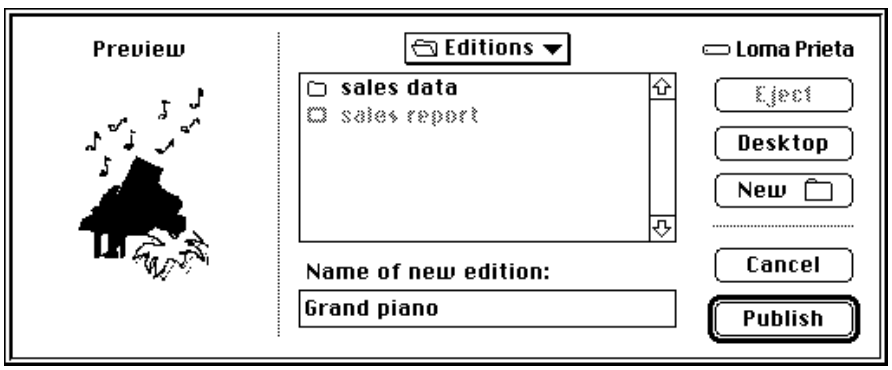


Note that the publisher and subscriber borders illustrated in Figure 2-2 may appear slightly different from the borders you see on the screen. Figure 2-6 on page 2-9 shows the publisher and subscriber borders as they appear onscreen.

Data always flows in one direction, from publisher to edition to subscriber. Documents that contain publishers and subscribers do not have to be open at the same time to share data. Whenever the user saves a document that contains a publisher, the edition changes to reflect the current data from the publisher. All subscribers update their contents from the edition. Any number of subscribers can subscribe to a single edition.

To create a publisher within a document, a user selects an area of the document to share and chooses Create Publisher from the Edit menu (see Figure 2-7 on page 2-10). Figure 2-3 shows the dialog box that your application should display when the user chooses Create Publisher.

Figure 2-3 The publisher dialog box



Your application provides a thumbnail sketch of the edition data, which the Edition Manager displays in the preview area of the publisher dialog box. Your preview of the edition in this dialog box should provide a visual cue about the type of information that the user has selected to publish.

A preview area also appears in the subscriber dialog box (see Figure 2-4). This preview, too, should provide a visual cue about the type of information the edition contains. For example, it should allow users to distinguish between text information and spreadsheet arrays.

The publisher dialog box uses the extended interface of the standard file dialog box that accompanies System 7. The user navigates through the contents of the disk using the mouse or keyboard.

A user can modify a publisher within a document just like any other portion of a document. As a default, each time a user saves a document containing a publisher, your application should automatically write the publisher's data to the edition. You also need to provide the user with the choice of sending new publisher data to an edition manually (that is, only at the user's specific request). You should provide these options by using the publisher options dialog box described later in "Using Publisher and Subscriber Options" beginning on page 2-43.

For example, one user may choose to update an edition automatically each time a document is saved. This update mode is useful for a user who creates a publisher within a spreadsheet application that records stock information. Each time the user updates the stock information and saves the spreadsheet, a new edition automatically becomes available to subscribers.

Another user may choose to update an edition only upon request. This update mode might be useful for a user who creates a publisher within a word-processing application for a quarterly sales report. The user incrementally updates the sales report throughout the entire quarter but does not want this information to be available to subscribers until the end of the quarter. Only at the end of each quarter does the user specifically request to update the edition and make it available to any subscribers.

To create a subscriber within a document, the user places the insertion point and chooses Subscribe To from the Edit menu. Figure 2-4 shows the dialog box that your application should display when the user chooses Subscribe To.

Figure 2-4 The subscriber dialog box

The subscriber dialog box also uses the extended interface of the standard file dialog box introduced with System 7. Initially, the dialog box should highlight the name of the last edition published or subscribed to. This allows a user to create a publisher and immediately subscribe to its edition.

A subscriber receives its data from a single edition. By default, your application should automatically update a document containing a subscriber whenever a new edition is available. You also need to provide the user with the choice of receiving the latest edition manually (that is, only when the user specifically requests it). You can provide these options by using the subscriber options dialog box described later in "Using Publisher and Subscriber Options" beginning on page 2-43.

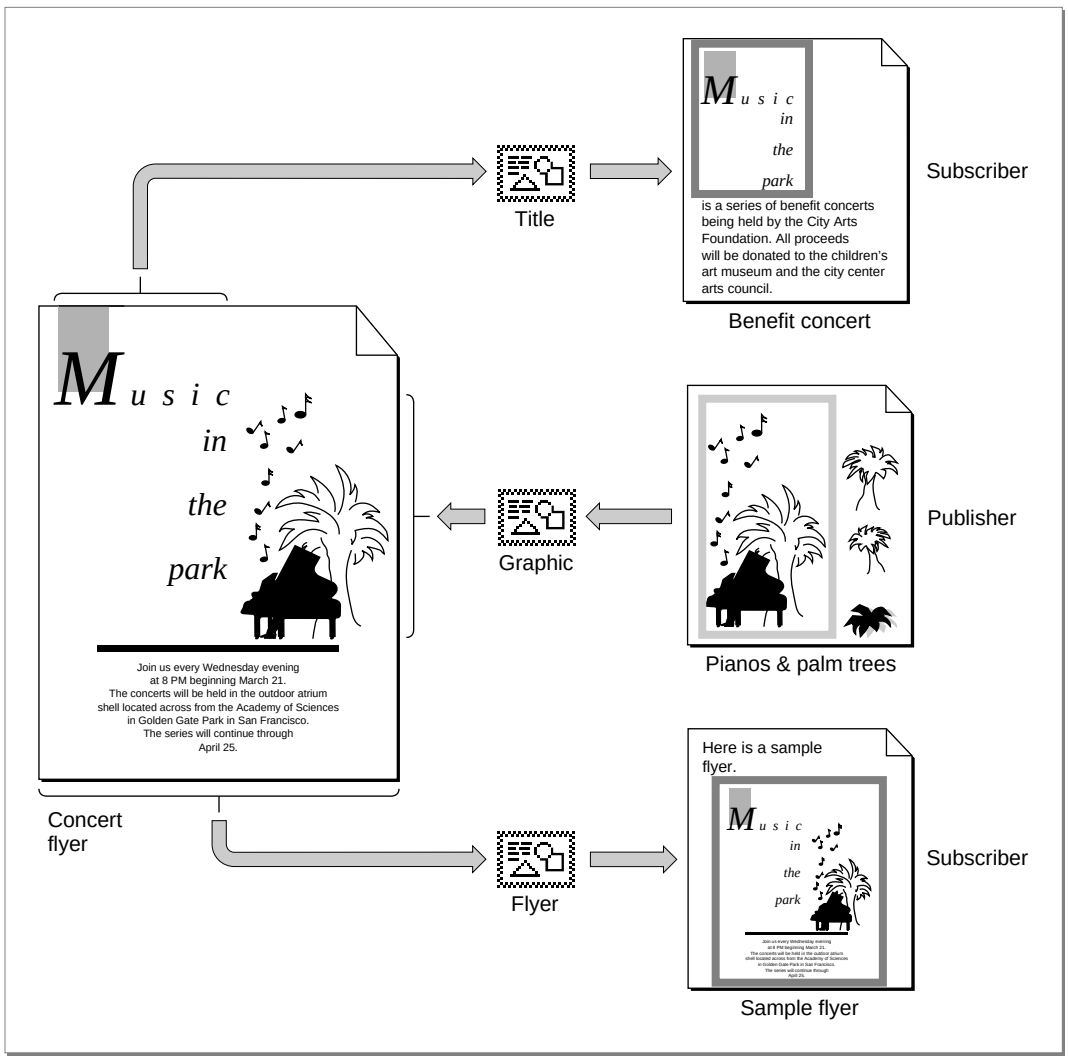
For example, one user may choose to receive new editions automatically as they become available. This update mode is useful for a user who subscribes to information from an edition that consists of daily sales figures. This user automatically acquires each version of the sales information as it becomes available.

Another user may choose to receive a new edition only upon request. This update mode is useful for a user who creates a subscriber to an edition that consists of graphics data (such as a company logo). The user may require only periodic versions of the logo and not need frequent updates. In this case, your application should update the subscriber with a new edition only when the user specifically requests it.

A user can select, cut, copy, or paste an entire subscriber. Although the contents of the subscriber as a whole can be modified, a user cannot edit portions of a subscriber. For example, a user can underline or italicize the entire subscriber text but cannot delete a sentence or rotate a single graphic object. This restriction protects the user from losing changes to a subscriber when a new edition arrives. Remember that, as a default, new editions should automatically update a subscriber. Any changes that a user made to the subscriber text would have to be reapplied by the user when the new edition arrives. See "Modifying a Subscriber" on page 2-59 for further information.

A single document can contain any number or combination of publishers and subscribers. Figure 2-5 shows an example of a document that contains two publishers and one subscriber (and their corresponding editions). Remember that data always flows in one direction, from publisher to edition to subscriber. The “Concert flyer” document contains a publisher that is subscribed to by the “Benefit concert” document. The “Concert flyer” document also subscribes to a portion of the “Pianos & palm trees” document. In addition, the “Concert flyer” document as a whole is subscribed to by the “Sample flyer” document.

Figure 2-5 A document and its corresponding editions

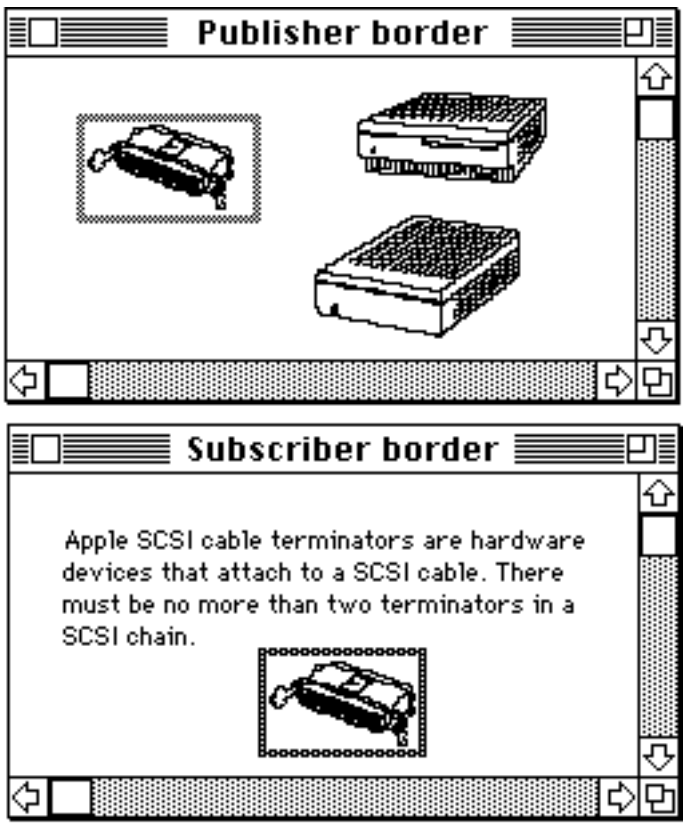


You should distinguish each selected publisher and subscriber within a document with a border. Display a publisher border as three pixels wide with 50 percent gray lines, and display a subscriber border as three pixels wide with 75 percent gray lines. A rectangle of one white pixel should separate the data from the border itself. Borders should be drawn *outside* the contents of publishers and subscribers so that data is not obscured. See Figure 2-6 for an illustration of the borders as they appear onscreen. See "Displaying Publisher and Subscriber Borders" on page 2-50 for detailed information on how to implement borders for specific applications.

Figure 2-6 shows a document containing a publisher and a document containing a subscriber, with borders displayed for each.

Borders for publishers and subscribers should behave like the borders of `'PICT'` graphics within a word-processing document. Your application should display a border whenever the user clicks within the content area of a publisher or a subscriber. Your application should hide the border whenever the user clicks outside the content area. See "Displaying Publisher and Subscriber Borders" on page 2-50 for detailed information on how to implement borders for specific applications.

Figure 2-6 Publisher and subscriber borders

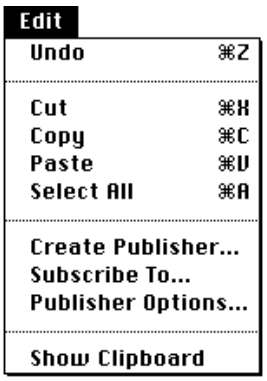


You also need to support the standard Edition Manager menu commands in the Edit menu. These menu items include

- Create Publisher…
- Subscribe To…
- Publisher/Subscriber Options…
- Show/Hide Borders (optional)
- Stop All Editions (optional)

Use a divider to separate the Edition Manager menu commands from the standard Edit menu commands Cut, Copy, and Paste. Figure 2-7 shows the standard Edition Manager menu commands.

Figure 2-7 Edition Manager commands in the Edit menu

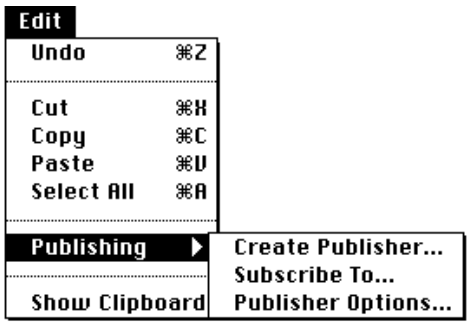


The Subscriber Options menu command should toggle with the Publisher Options menu command. When a user selects a subscriber and then accesses the menu bar, your application should adjust its menus so that the Subscriber Options menu command appears in the Edit menu. When a user selects a publisher and then accesses the menu bar, your application should adjust its menus so that the Publisher Options menu command appears in the Edit menu. In addition, you may support a Show Borders menu command that toggles with Hide Borders to display or hide all publishers and subscriber borders within documents. You may also support a Stop All Editions menu command to provide a method for temporarily suspending all update activity in a document. When the user chooses this command, you should place a checkmark next to it. You should also stop all publishers from sending data to editions and all subscribers from receiving new editions. When the user chooses this command again, remove the checkmark and update any subscribers that are set up to receive new editions automatically.

If you find that you need all of the available space in the Edit menu for your application's commands, you may create a hierarchical menu for the Edition Manager menu commands. If you choose to implement this structure, you should allow users to access the Edition Manager menu commands through a Publishing menu command in the Edit menu. Because this menu structure is not as accessible to users, you should implement it only if you have no other alternative.

Figure 2-8 shows the Edition Manager menu commands in a hierarchical menu structure.

Figure 2-8 Edition Manager commands under the Publishing menu command



For each publisher or subscriber within an open document, you must have a section record and an alias record. The section record contains a time stamp that records the version of the data that resides in the section. The section record also identifies the section as either a publisher or subscriber, and it establishes a unique identity for each publisher or subscriber. The section record does *not* contain the data within the section. The alias record is a reference to the edition container from the document that contains the corresponding publisher or subscriber section.

There are special options associated with publishers and subscribers within documents. Your application can use the publisher and subscriber options dialog boxes provided by the Edition Manager to make these choices available to the user. For example, a user can select Open Publisher within the subscriber options dialog box to access the document containing the publisher. Your application can also allow a user to cancel subscribers or publishers within documents, specify when to update an edition from a publisher, or specify when to update a subscriber with a new edition. These options are described in "Using Publisher and Subscriber Options" beginning on page 2-43.

About the Edition Manager

The next section discusses how to save, open, read, and write a document that shares data. In addition, it describes how to

- make data accessible to other applications
- integrate data into numerous documents
- set update options
- implement borders
- modify shared data
- customize dialog boxes
- subscribe to data in non-edition files

Using the Edition Manager

This section describes how your application can

- receive Apple events from the Edition Manager
- set up a section record and alias record for open documents containing sections
- save a document that contains sections
- open a document that contains sections
- read and write sections
- create a publisher within a document, create its edition container, and write data to it
- create a subscriber within a document and read its data from an edition

To begin, you must determine whether the Edition Manager is available on the system by using the `Gestalt` function with the `gestaltEditionMgrAttr` (`'edtn'`) selector. If the `response` parameter returns 1 in the bit defined by the `gestaltEditionMgrPresent` constant (bit 0), the Edition Manager is present.

If the Edition Manager is present, load it into memory using the `InitEditionPack` function. This function determines whether the machine has enough space in the system heap for the Edition Manager to operate.

```
err := InitEditionPack;
```

If the `InitEditionPack` function returns `noErr`, you have enough space to load the package. If you do not have enough space, the application can either terminate itself or continue with the Edition Manager functionality disabled.

Receiving Apple Events From the Edition Manager

Applications that use the Edition Manager must support Apple events. This requires that your application support the required Open Documents event and Apple events sent by the Edition Manager. See the chapter "Introduction to Apple Events" in this book for general information on Apple events.

Apple events sent by the Edition Manager arrive as high-level events. The `EventRecord` data type defines the event record.

```
TYPE EventRecord =
   RECORD
      what:       Integer;    {kHighLevelEvent}
      message:    LongInt;    {'sect'}
      when:       LongInt;
      where:      Point;      {'read', 'writ', 'cncl', 'scrl'}
      modifiers:  Integer;
   END;
```

The Edition Manager can send these Apple events with the event class and event ID as shown here:

- Section Read events (`'sect' 'read'`)
- Section Write events (`'sect' 'writ'`)
- Section Cancel events (`'sect' 'cncl'`)
- Section Scroll events (`'sect' 'scrl'`)

Each time your application creates a publisher or a subscriber, the Edition Manager registers its section. When an edition is updated, the Edition Manager scans its list to locate registered subscribers. For each registered subscriber that is set up to receive updated editions automatically, your application receives a Section Read event.

If the Edition Manager discovers that an edition file is missing while registering a publisher, it creates a new edition file and sends the publisher a Section Write event.

When you receive a Section Cancel event, you need to cancel the specified section. Note that the current Edition Manager does not send you Section Cancel events, but you do need to provide a handler for future expansion.

If the user selects a subscriber within a document and then selects Open Publisher in the subscriber options dialog box, the publishing application receives the Open Documents event and opens the document containing the publisher. The publishing application also receives a Section Scroll event. Scroll to the location of the publisher, display this section on the user's screen, and turn on its border.

See "Opening and Closing a Document Containing Sections" beginning on page 2-22 for detailed information on registering and unregistering a section and writing data to an edition. See "Using Publisher and Subscriber Options" beginning on page 2-43 for information on publisher and subscriber options.

After receiving an Apple event sent by the Edition Manager, use the Apple Event Manager to extract the section handle. In addition, you must also call the `IsRegisteredSection` function to determine whether the section is registered. It is possible (because of a race condition) to receive an event for a section that you recently disposed of or unregistered. One way to ensure that an event corresponds to a valid section is to call the `IsRegisteredSection` function after you receive an event.

```
err := IsRegisteredSection (sectionH);
```

Listing 2-1 illustrates how to use the Apple Event Manager and install an event handler to handle Section Read events. You can write similar code for Section Write events, Section Scroll events, and Section Cancel events.

Listing 2-1 Accepting Section Read events and verifying if a section is registered

```
{the following goes in your initialization code}
myErr := AEInstallEventHandler(sectionEventMsgClass {'sect'},
                               sectionReadMsgID {'read'},
                               @MyHandleSectionReadEvent, 0,
                               FALSE);

{this is the routine the Apple Event Manager calls when a }
{ Section Read event arrives}

FUNCTION MyHandleSectionReadEvent(theAppleEvent,
                                  reply: AppleEvent;
                                  refCon: LongInt): OSErr;
VAR
   myErr:      OSErr;
   sectionH:   SectionHandle;
BEGIN
   {get section handle out of Apple event message buffer}
   myErr := MyGetSectionHandleFromEvent(theAppleEvent, sectionH);
   IF myErr = noErr THEN
   BEGIN
      IF IsRegisteredSection(sectionH) = noErr THEN
         {if section is registered, read the new data}
         MyHandleSectionReadEvent := DoSectionRead(sectionH);
   END
   ELSE
      MyHandleSectionReadEvent := myErr;
END; {MyHandleSectionReadEvent}
```

```
{this routine reads in subscriber data and updates its display}
FUNCTION DoSectionRead(subscriber: SectionHandle): OSErr;
BEGIN
   {your code here}
END;  {DoSectionRead}

{this is part of your Apple event–handling code}
FUNCTION MyGetSectionHandleFromEvent(theAppleEvent: AppleEvent;
                                     VAR sectionH: SectionHandle)
                                     : OSErr;
VAR
   ignoreType:    DescType;
   ignoreSize:    Size;
BEGIN
   {parse section handle out of message buffer}
   MyGetSectionHandleFromEvent
      := AEGetParamPtr( theAppleEvent,    {event to parse}
                        keyDirectObject,  {look for direct }
                                          { object}
                        typeSectionH,     {want a SectionHandle}
                        ignoreType,       {ignore type it could }
                                          { get}
                        @sectionH,        {put SectionHandle }
                                          { here}
                        SizeOf(sectionH), {size of storage for }
                                          { SectionHandle}
                        ignoreSize);      {ignore storage it }
                                          { used}
END; {MyGetSectionHandleFromEvent}
```

In addition to the Section Read, Section Write, Section Cancel, and Section Scroll events, your application can also respond to the Create Publisher event. For more information on this event, as well as additional information on how to handle Apple events, see the chapter "Responding to Apple Events" in this book.

Creating the Section Record and Alias Record

Your application is responsible for creating a section record and an alias record for each publisher and subscriber section within an open document.

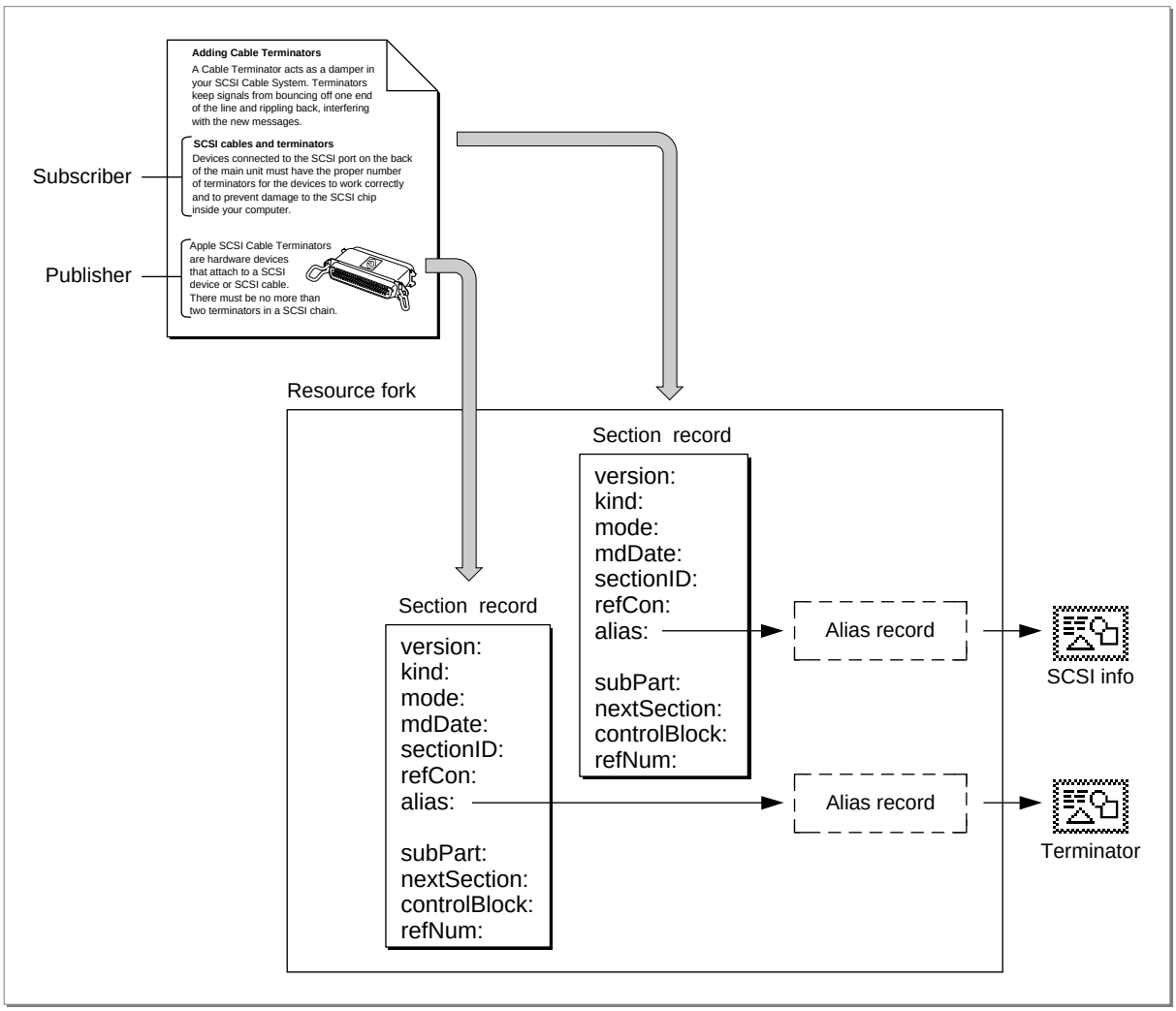
The section record identifies each section as a publisher or subscriber and provides identification for each section. The section record does not contain the data within the section; it describes the attributes of the section. Your application must provide its own method for associating the data within a section with its section record. Your application is also responsible for saving the data in the section.

The `alias` field of the section record contains a handle to its alias record. The alias record is a reference to the edition container from the document that contains the publisher or subscriber section. You should be familiar with the Alias Manager's conventions for creating alias records and identifying files, folders, and volumes to locate files that have been moved, copied, or restored from backup. For information on the Alias Manager, see *Inside Macintosh: Files.*

When a user saves a document, your application should store all section records and alias records in the document's resource fork. Corresponding section records and alias records should have the same resource ID.

Figure 2-9 shows a document containing a publisher and subscriber, and the corresponding section records and alias records.

Figure 2-9 A document with a publisher and subscriber and its resource fork



A section record contains information to identify the data contained within a section as a publisher or a subscriber, a time stamp to record the last modification of the section, and unique identification for each section. The `SectionRecord` data type defines the section record.

```
TYPE SectionRecord =
   RECORD
      version:       SignedByte;    {always 1 in 7.0}
      kind:          SectionType;   {publisher or subscriber}
      mode:          UpdateMode;    {automatic or manual}
      mdDate:        TimeStamp;     {last change in document}
      sectionID:     LongInt;       {application-specific, }
                                    { unique per document}
      refCon:        LongInt;       {application-specific}
      alias:         AliasHandle;   {handle to alias record}

      {The following fields are private and are set up by the }
      { RegisterSection function described later within this }
      { chapter. Do not modify the private fields.}

      subPart:       LongInt;       {private}
      nextSection:   SectionHandle; {private, do not use as a }
                                    { linked list}
      controlBlock:  Handle;        {may be used for comparison }
                                    { only}
      refNum:        EditionRefNum; {private}
END;
```

Field descriptions

`version` Indicates the version of the section record, currently $01.

`kind` Defines the section type as either publisher or subscriber with the `stPublisher` or `stSubscriber` constant.

`mode` Indicates if editions are updated automatically or manually.

`mdDate` Indicates which version (modification date) of the section's contents is contained within the publisher or subscriber. The `mdDate` field is set to 0 when you create a new subscriber section and to the current time when you create a new publisher. Be sure to update this field each time publisher data is modified. The section's modification date is compared to the edition's modification date to determine whether the section and the edition contain the same data. The section modification date is displayed in the publisher and subscriber options dialog boxes. See "Closing an Edition" on page 2-28 for detailed information.

`sectionID`	Provides a unique number for each section within a document. A simple way to implement this is to create a counter for each document that is saved to disk with the document. The counter should start at 1. The section ID is currently used as a tie breaker in the `GoToPublisherSection` function when there are multiple publishers to the same edition in a single document. The section ID should not be 0 or –1. See "Duplicating Publishers and Subscribers" on page 2-58 for information on multiple publishers.
`refCon`	Reference constant available for application-specific use.
`alias`	Contains a handle to the alias record for a particular section within a document.

Whenever the user creates a publisher or subscriber, call the `NewSection` function to create the section record and the alias record.

```
err := NewSection(container, sectionDocument, kind, sectionID,
                  initialMode, sectionH);
```

The `NewSection` function creates a new section record (either publisher or subscriber), indicates whether editions are updated automatically or manually, sets the modification date, and creates an alias record from the document containing the section to the edition container.

You can set the `sectionDocument` parameter to `NIL` if the current document has never been saved. Use the `AssociateSection` function to update the alias record of a registered section when the user names or renames a document by choosing Save As from the File menu. If you are creating a subscriber with the `initialMode` parameter set to receive new editions automatically, your application receives a Section Read event each time a new edition becomes available for this subscriber.

If an error is encountered, the `NewSection` function returns `NIL` in the `sectionH` parameter. Otherwise, `NewSection` returns a handle to the allocated section record in the `sectionH` parameter.

Set the `initialMode` parameter to the update mode for each subscriber and publisher created. You can specify the update mode using these constants:

```
CONST    sumAutomatic   = 0;     {subscriber receives new }
                                 { editions automatically}
         sumManual      = 1;     {subscriber receives new }
                                 { editions manually}
         pumOnSave      = 0;     {publisher sends new }
                                 { editions on save}
         pumManual      = 1;     {publisher does not send }
                                 { new editions until user }
                                 { request}
```

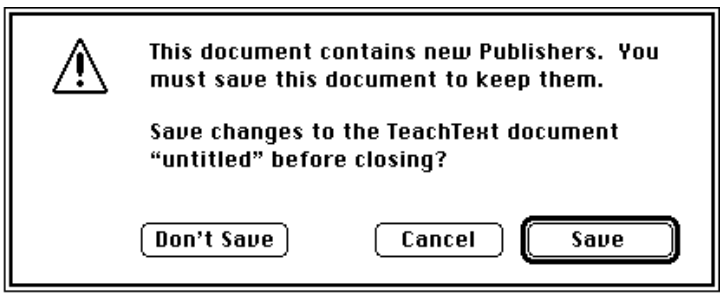
See "Using Publisher and Subscriber Options" beginning on page 2-43 for detailed information on update modes for publishers and subscribers. See Listing 2-4 beginning on page 2-33 for code that uses the `NewSection` function to create a publisher. See Listing 2-6 on page 2-40 for code that uses `NewSection` to create a subscriber.

Saving a Document Containing Sections

When saving a document that contains sections, you should write out each section record as a resource of type `'sect'` and write out each alias record as a resource of type `'alis'` with the same ID as the section record. See the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox* for detailed information on resources.

If a user closes a document that contains newly created publishers without attempting to save its contents, you should display an alert box similar to the one shown in Figure 2-10.

Figure 2-10 The new publisher alert box



If you keep the section records and alias records for each publisher and subscriber as resources, you can use the `ChangedResource` or `WriteResource` function. If you detach the section records and alias records from each section, you need to clone the handles and use the `AddResource` function. See the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox* for detailed information on the `ChangedResource`, `WriteResource`, and `AddResource` functions.

Use the `PBExchangeFiles` function to ensure that the file ID remains the same each time you save a document that contains sections. Saving a file typically involves creating a new file (with a temporary name), writing data to it, closing it, and then deleting the original file that you are replacing. You rename the temporary file with the original filename, which leads to a new file ID. The `PBExchangeFiles` function swaps the contents of the two files (even if they are open) by getting both catalog entries and swapping the allocation pointers. If the files are open, the file control block (FCB) is updated so that the reference numbers still access the same contents (under a new name). See *Inside Macintosh: Files* for detailed information on the `PBExchangeFiles` function.

Listing 2-2 illustrates how to save a file that contains sections. If the contents of a publisher have changed since the last save, the application-defined procedure `MySaveDocument` writes the publisher's data to its edition. It then writes out to the saved document the section records and alias records of all publishers and subscribers. `MySaveDocument` calls another application-defined routine, `MyGetSectionAliasPair`, to return a handle and resource ID to a section. As described earlier, you should write out the eligible section records and alias records as resources to allow for future compatibility. There are several different techniques for saving or adding resources; this listing illustrates one technique. The section handles are still valid after using the `AddResource` function because this listing illustrates just saving, not closing, the file.

Before you write out sections, you need to see if any publisher sections share the same control block. Publishers that share the same control block share the same edition.

If a user creates an identical copy of a file by choosing Save As from the File menu and does not make any changes to this new file, you simply use the `AssociateSection` function to indicate to the Edition Manager which document a section is located in.

Listing 2-2 Saving a document containing sections

```
PROCEDURE MySaveDocument(thisDocument: MyDocumentInfoPtr;
                         numberOfSections: Integer);
VAR
   aSectionH:        SectionHandle;
   copiedSectionH:   Handle;
   copiedAliasH:     Handle;
   resID:            Integer;
   thisone:          Integer;
   myErr:            OSErr;
BEGIN
   FOR thisone := 1 TO numberOfSections DO
   BEGIN
      aSectionH := MyGetSectionAliasPair(thisDocument, thisone,
                                         resID);
      IF (aSectionH^^.kind = stPublisher) &
         (aSectionH^^.mode = pumOnSave) &
         (MyCheckForDataChanged(aSectionH)) THEN
         DoWriteEdition(aSectionH);
   END; {end of for}
   {set the curResFile to the resource fork of thisDocument}
   UseResFile(thisDocument^.resForkRefNum);
   {write all section and alias records to the document}
   FOR thisone := 1 TO numberOfSections DO
   BEGIN
      {given an index, get the next section handle and resID }
      { from your internal list of sections for this file}
      aSectionH := MyGetSectionAliasPair(thisDocument, thisone,
                                         resID);
      {check for duplication of control block values}
      MyCheckForDupes(thisDocument, numberOfSections);
      {save section record to disk}
      copiedSectionH := Handle(aSectionH);
      myErr := HandToHand(copiedSectionH);
      AddResource(copiedSectionH, rSectionType, resID, '');
      {save alias record to disk}
      copiedAliasH := Handle(aSectionH^^.alias);
      myErr := HandToHand(copiedAliasH);
      AddResource(copiedSectionH, rAliasType, resID, '');
   END; {end of for}
   {write rest of document to disk}
END;
```

Opening and Closing a Document Containing Sections

When opening a document that contains sections, your application should use the `GetResource` function to get the section record and the alias record for each publisher and subscriber. Set the `alias` field of the section record to be the handle to the alias. See the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox* for detailed information on the `GetResource` function.

You also need to register each section using the `RegisterSection` function. The `RegisterSection` function informs the Edition Manager that a section exists.

```
err := RegisterSection(sectionDocument, sectionH,
                       aliasWasUpdated);
```

The `RegisterSection` function adds the section record to the Edition Manager's list of registered sections. This function assumes that the `alias` field of each section record is a handle to the alias record. The alias record is a reference to the edition container from the section's document. If the `RegisterSection` function successfully locates the edition container for a particular section, the section is registered through a shared control block. The control block is a private field in the section record.

If the `RegisterSection` function cannot find the edition container for a particular subscriber, `RegisterSection` returns the `containerNotFoundWrn` result code. If the `RegisterSection` function cannot find the edition container for a particular publisher, `RegisterSection` creates an empty edition container for the publisher in the last place the edition was located. The Edition Manager sends your application a Section Write event for that section.

When a user attempts to open a document that contains multiple publishers to the same edition, you should warn the user by displaying an alert box (see "Duplicating Publishers and Subscribers" on page 2-58).

When a user opens a document that contains a subscriber (with an update mode set to automatic), receives a new edition, and then closes the document without making any changes to the file, you should update the document and simply allow the user to close it. You do not need to prompt the user to save changes to the file.

When closing a document that contains sections, you must unregister each section (using the `UnRegisterSection` function) and dispose of each corresponding section record and alias record.

```
err := UnRegisterSection(sectionH);
```

The `UnRegisterSection` function removes the section record from the list of registered sections and unlinks itself from the shared control block.

Listing 2-3 illustrates how to open an existing file that contains sections. As described earlier, you should retrieve the section and alias resources, connect the pair through the `alias` field of the section record, and register the section with the Edition Manager. There are many different techniques for retrieving resources; this listing shows one technique. If an alias was out of date and was updated by the Alias Manager during the resolve, the Edition Manager sets the `aliasWasUpdated` parameter of the `RegisterSection` function to `TRUE`. This means that you should save the document. Additionally, your application must maintain its own list of registered sections for each open document that contains sections. You use this list to write out new editions for updated publishers within a document.

Listing 2-3 Opening a document containing sections

```
PROCEDURE MyOpenExistingDocument(thisDocument: MyDocumentInfoPtr);
VAR
   sectionH:            SectionHandle;
   aliasH:              AliasHandle;
   aliasWasUpdated:     Boolean;
   registerErr:         OSErr;
   resID:               Integer;
   theResType:          ResType;
   thisone:             Integer;
   numberOfSections:    Integer;
   aName:               Str255;
BEGIN
   UseResFile(thisDocument^.resForkRefNum);
   {find out the number of section resources}
   numberOfSections := Count1Resources(rSectionType);
   FOR thisone := 1 TO numberOfSections DO
   BEGIN
      sectionH := SectionHandle(Get1IndResource(rSectionType,
                                          thisone));
      IF sectionH = NIL THEN  {something could be wrong with }
         MySectionErr;        { the file, handle appropriately}
      {get resource ID of the section & use same ID for alias}
      GetResInfo(Handle(sectionH), resID, theResType, aName);
      {detaching is not necessary, but it is convenient}
      DetachResource(Handle(sectionH));
      {get the alias}
      aliasH := AliasHandle(Get1Resource(rAliasType, resID));
      IF aliasH = NIL THEN    {something could be wrong with }
         MyAliasErr;          { the file, handle appropriately}
      DetachResource(Handle(aliasH));
```

```
      {connect section and alias together}
      sectionH^^.alias := aliasH;
      {register the section}
      registerErr := RegisterSection(thisDocument^.fileSpec,
                                     sectionH, aliasWasUpdated);
      {The RegisterSection function may return an error if }
      { a section is not registered. This is not a fatal error. }
      { Continue looping to register remaining sections.}
      {add this section/alias pair to your internal bookkeeping}
      MyAddSectionAliasPair(thisDocument, sectionH, resID);
      IF aliasWasUpdated THEN
         {If alias has changed, make a note of this. }
         { It's important to know this when you save.}
         MyAliasHasChanged(sectionH);
   END; {end of FOR}
END;
```

Reading and Writing a Section

Your application writes publisher data to an edition. New publisher data replaces the previous contents of the edition, making the previous edition information irretrievable. Your application reads data from an edition for each subscriber within a document.

The following sections describe how to

- use different formats to write to or read from an edition
- open an edition to initiate writing or reading
- set a format mark
- write to or read from an edition
- close an edition after successfully writing or reading data

Formats in an Edition

You can write data to an edition in several different formats. These formats are the same as scrap format types. Scrap format types are indicated by a four-character tag.

Typically, when a user copies data, you identify the scrap format types and then write the data to the scrap. With the Edition Manager, when a user decides to publish data, you identify the format types and then write the data to an edition. You can write multiple formats of the same data.

For an edition, you should write your preferred formats first. In general, to write data to an edition, your application should use either `'TEXT'` format or `'PICT'` format. This allows your application to share data with most other applications. To subscribe to an edition, your application should be able to read both `'TEXT'` and `'PICT'` files. In addition, your application can write any other private formats that you want to support.

Scrap format types are described in the chapter "Scrap Manager" in *Inside Macintosh: More Macintosh Toolbox*.

A few special formats are defined as constants.

```
CONST kPublisherDocAliasFormat   = 'alis';{alias record from the }
                                         { edition to publisher}
      kPreviewFormat             = 'prvw';{'PICT' thumbnail }
                                         { sketch}
      kFormatListFormat          = 'fmts';{lists all available }
                                         { formats}
```

The `kPublisherDocAliasFormat` (`'alis'`) format is written by the Edition Manager. It is an alias record from the edition to the publisher's document. Appended to the end of the alias is the section ID of the publisher, which the Edition Manager uses to distinguish between multiple publishers to a single edition. You should discourage users from making multiple copies of the same publisher. See "Duplicating Publishers and Subscribers" on page 2-58 for detailed information.

In addition to writing a publisher's data to an edition in the `'TEXT'` format or `'PICT'` format, your application can also write data to an edition in the `kPreviewFormat` (`'prvw'`) format. If you provide a `'prvw'` format in an edition, the Edition Manager uses it to display a preview of the edition data in the preview area of the subscriber dialog box. The `'prvw'` format has the same format as a `'PICT'` file. To draw a preview in the `'prvw'` format, the Edition Manager calls `DrawPicture` with a rectangle of 120 by 120 pixels. (See *Inside Macintosh: Imaging* for more information about `DrawPicture`.) Your application should provide data in a `'prvw'` format so that the data displays well in a rectangle of this size. Your application can also use this preview to display subscriber data within a document (to save space).

If your application does not provide a preview in the `'prvw'` format for an edition, the Edition Manager attempts to provide a preview by using the edition's `'PICT'` format. To draw a preview in the `'PICT'` format, the Edition Manager examines the picture's bounding rectangle and calls `DrawPicture` with a rectangle that scales the picture proportionally and centers it in a 120-by-12-pixel area.

The `kFormatListFormat` (`'fmts'`) format is a virtual format that is read but never written. It is a list of all the formats and their lengths. Applications can use this format in place of the `EditionHasFormat` function (described in "Choosing Which Edition Format to Read" on page 2-41), which provides a procedural interface to determine which formats are available.

If your application can read two or more of the available formats, use `'fmts'` to determine the priority of these formats for a particular edition. The order of `'fmts'` reflects the order in which the formats were written.

The FormatsAvailable data type defines a record for the 'fmts' format.

```
TYPE FormatsAvailable = ARRAY[0..0] OF
   RECORD
      theType:    FormatType;    {format type for an edition}
      theLength:  LongInt;       {length of edition format }
                                 { type}
   END;
```

For example, an edition container may have a format type 'TEXT' of length 100, and a format type 'styl' of length 32. A subscriber to this edition can open it and then read the format type 'fmts' to list all available formats. In this example, it returns 16 bytes: 'TEXT' $00000064 'styl' $00000020.

Opening an Edition

For a publisher, use the OpenNewEdition function to initiate the writing of data to an edition. (Note that the edition container must already exist before you initiate writing; see "Creating the Edition Container" beginning on page 2-32.)

```
err := OpenNewEdition(publisherSectionH, fdCreator,
                      publisherSectionDocument, refNum);
```

The publisherSectionH parameter is the publisher section that you are writing to the edition. The fdCreator parameter is the Finder creator type of the new edition. (The edition container file already has a creator type; you can specify the same creator type or establish a new creator type for the edition.)

The publisherSectionDocument parameter specifies the document that contains the publisher. This parameter is used to create an alias from the edition to the publisher's document. If you pass NIL for publisherSectionDocument, an alias is not made in the edition file. The refNum parameter returns the reference number for the edition.

For a subscriber, use the OpenEdition function to initiate the reading of data from an edition.

```
err := OpenEdition(subscriberSectionH, refNum);
```

The subscriberSectionH parameter is a handle to the section record for a given section. The refNum parameter returns the reference number for the edition.

The user may rename or move the edition in the Finder. Before writing to or reading data from an edition, the Edition Manager verifies the name of the edition. This process is referred to as *synching* or synchronization. Synching ensures that the Edition Manager's existing edition names correspond to the Finder's existing edition names by updating the control block.

Format Marks

Each format has its own mark. The mark indicates the next position of a read or write operation. Initially, a mark automatically defaults to 0. After reading or writing data, the format mark is set past the last position written to or read from. The mark is similar to the File Manager's current read or write position marker for a data fork. Any time that an edition is open (after calling the `OpenEdition` or the `OpenNewEdition` function), any of the marks for each format can be queried or set.

To set the current mark for a section format to a new location, use the `SetEditionFormatMark` function.

```
err := SetEditionFormatMark(whichEdition, whichFormat,
                            setMarkTo);
```

To get the current mark for a format in an edition file, use the `GetEditionFormatMark` function.

```
err := GetEditionFormatMark(whichEdition, whichFormat,
                            currentMark);
```

Reading and Writing Edition Data

The Edition Manager allows you to read or write data a few bytes at a time (as with a data fork of a Macintosh file) instead of in one block (as with the Scrap Manager). You can read sequentially by setting the mark to 0 and repeatedly calling read, or you can jump to a specific offset by setting the mark there. The Edition Manager also adds the capability to stream multiple formats by keeping a separate mark for each format. This allows you to write a few bytes of one format and then write a few bytes of another format, and so forth.

Once you have opened the edition container for a particular publisher, you can begin writing data to the edition. Use the `WriteEdition` function to write publisher data to an edition.

```
err := WriteEdition(whichEdition, whichFormat, buffPtr, buffLen);
```

The `WriteEdition` function writes the specified format (beginning at the current mark for that format type) from the buffer pointed to by the `buffPtr` parameter up to `buffLen` bytes.

After you open the edition container for a subscriber and determine which formats to read, use the `ReadEdition` function to read edition data.

```
err := ReadEdition(whichEdition, whichFormat, buffPtr, buffLen);
```

The `ReadEdition` function reads the data with the specified format (`whichFormat`) from the edition into the buffer. The `ReadEdition` function begins reading at the current mark for that format and continues to read up to `buffLen` bytes. The actual number of bytes read is returned in the `buffLen` parameter. Once the `buffLen` parameter returns a value smaller than the value you have specified, there is no additional data to read, and the `ReadEdition` function returns a `noErr` result code.

Note
The Translation Manager (if it is available) attempts implicit translation under certain circumstances. For instance, it does so when your application attempts to read from an edition a format type that is not in the edition. In this case, the Translation Manager attempts to translate the data into the requested format. For more information, see the chapter "Translation Manager" in *Inside Macintosh: More Macintosh Toolbox*. ◆

Closing an Edition

When you are done writing to or reading data from an edition, call the `CloseEdition` function.

```
err := CloseEdition(whichEdition, successful);
```

Each time a user edits a publisher within a document, you must update the modification date in the section record (even if the data is not yet written). When the update mode is set to Manually, the user can compare the modification dates for a publisher and its edition in the publisher options dialog box. One modification date indicates when the publisher last wrote data to the edition, and the other modification date indicates when the publisher section was last edited.

If the `successful` parameter for a publisher is `TRUE`, the `CloseEdition` function makes the newly written data available to subscribers and sets the modification date in the `mdDate` field of the edition to correspond to the modification date of the publisher's section record. If the two dates differ, the Edition Manager sends a Section Read event to all current subscribers.

If the `successful` parameter for a subscriber is `TRUE`, the `CloseEdition` function sets the modification date of the subscriber's section record to correspond to the modification date of the edition.

If you cannot successfully read from or write data to an edition, set the `successful` parameter to `FALSE`. For a publisher, data is not written to the edition, but it should still be saved with the document that contains the section. When the document is next saved, data can then be written to the edition. See "Closing an Edition After Reading or Writing" on page 2-88 for additional information on the `CloseEdition` function.

Creating a Publisher

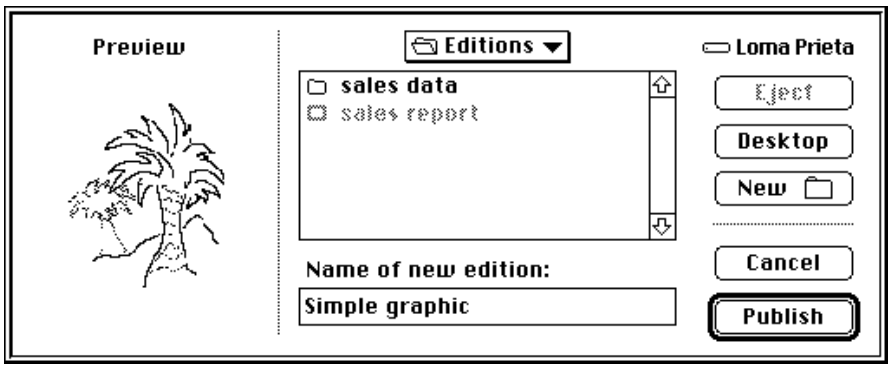
You need to support a Create Publisher menu command in the Edit menu. When a user selects a portion of a document and chooses Create Publisher from this menu, you should display the publisher dialog box on the user's screen. The Create Publisher menu command should remain dimmed until the user selects a portion of a document.

Use the `NewPublisherDialog` function to display the publisher dialog box on the user's screen. This function is similar to the `CustomPutFile` procedure described in the chapter "Standard File Package" in *Inside Macintosh: Files*.

```
err := NewPublisherDialog(reply);
```

The dialog box contains space for a preview (a thumbnail sketch) of the edition and a space for the user to type the name of the edition in which to write the publisher data. Figure 2-11 illustrates a sample publisher dialog box.

Figure 2-11 A sample publisher dialog box



The `NewPublisherDialog` function displays the preview (provided by your application), displays a text box with the default name of the edition (provided by your application), and handles all user input until the user clicks Publish or Cancel.

You pass a new publisher reply record as a parameter to the `NewPublisherDialog` function.

```
TYPE NewPublisherReply =
   RECORD
      canceled:   Boolean;          {user clicked Cancel}
      replacing:  Boolean;          {user chose existing }
                                    { filename for an edition}
      usePart:    Boolean;          {always FALSE in version 7.0}
      preview:    Handle;           {handle to 'prvw', 'PICT', }
                                    { 'TEXT', or 'snd ' data}
      previewFormat:                {type of preview}
                  FormatType;
      container:  EditionContainerSpec;{initially, default name }
                                    { and location of edition; }
                                    { on return, edition name & }
                                    { location chosen by the }
                                    { user to publish data to}
   END;
```

You fill in the `usePart`, `preview`, `previewFormat`, and `container` fields of the new publisher reply record.

Always set the `usePart` field to `FALSE`. The `preview` field should contain either `NIL` or the data to display in the preview. The `previewFormat` field should contain `'PICT'`, `'TEXT'`, `'snd '`, or `'prvw'`.

Set the `container` field to be the default name and folder for the edition. The default name should reflect the data contained in the publisher. For example, if a user publishes a bar chart of sales information entitled “sales data,” then the default name for the edition could also be “sales data.” Otherwise, you should use the document name followed by a hyphen (-) and a number to establish uniqueness. For example, your default name could be “January Totals - 3.”

If the document has not been saved, the default name should be “untitled edition <*n*>” where *n* is a number to establish uniqueness. The default folder should be the same as the edition for the last publisher created in the same document. If this is the first publisher in the document, the default folder should be the same folder that the document is in.

The `canceled` field of the new publisher reply record indicates whether the user clicked Cancel. The `replacing` field indicates whether the user chose to replace an existing edition file. If `replacing` returns `FALSE`, call the `CreateEditionContainerFile` function to create an edition file.

The `container` field is of data type `EditionContainerSpec`.

```
TYPE EditionContainerSpec =
   RECORD
      theFile:       FSSpec;     {record that identifies the }
                                 { file to contain edition data}
      theFileScript: ScriptCode; {script code of filename}
      thePart:       LongInt;    {which part of file, }
                                 { always kPartsNotUsed}
      thePartName:   Str31;      {not used in version 7.0}
      thePartScript: ScriptCode; {not used in version 7.0}
   END;
```

The field `theFile` is a file system specification record, a data structure of type `FSSpec`. You identify the edition using a volume reference number, directory ID, and filename. When specifying an edition, follow the standard conventions described in *Inside Macintosh: Files*.

After filling in the fields of the new publisher reply record, pass it as a parameter to the `NewPublisherDialog` function, which displays the publisher dialog box.

```
err := NewPublisherDialog(reply);
```

After displaying the publisher dialog box, use the `CreateEditionContainerFile` function to create the edition container, and then use the `NewSection` function to create the section record and the alias record. See the next section, "Creating the Edition Container," and "Creating the Section Record and Alias Record" on page 2-15 for detailed information.

The following code segment illustrates how your application might respond to the user choosing the Create Publisher menu item. In this case, the code sets up the preview for the edition, sets the default name for the edition container, and calls an application-defined function (`DoNewPublisher`, shown in Listing 2-4 on page 2-33) to display the publisher dialog box on the user's screen. An application might call the `DoNewPublisher` function in response to the user's choosing Create Publisher from the Edit menu or in response to handling the Create Publisher event. The chapter "Responding to Apple Events" in this book gives an example of a handler for the Create Publisher event.

```
VAR
   thisDocument:        MyDocumentInfoPtr;
   promptForDialog:     Boolean;
   preview:             Handle;
   previewFormat:       FormatType;
   defaultLocation:     EditionContainerSpec;
   myErr:               OSErr;
```

```
BEGIN
   {Get a preview to show the user. The MyGetPreviewForSelection }
   { function returns a handle to the preview.}
   preview := MyGetPreviewForSelection(thisDocument);
   previewFormat := 'TEXT';
   defaultLocation := MyGetDefaultEditionSpec(thisDocument);
   promptForDialog := TRUE;
   myErr := DoNewPublisher(thisDocument, promptForDialog, preview,
                           previewFormat, defaultLocation);
END;
```

Creating the Edition Container

Use the `CreateEditionContainerFile` function to create an edition container to hold the publisher data.

```
err := CreateEditionContainerFile(editionFile, fdCreator,
                                  editionFileNameScript);
```

This function creates an edition container. The edition container is empty (that is, it does not contain any formats) at this time.

To associate an icon with the edition container, create the appropriate entries for the icon in your application's bundle. See the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials* for additional information. Depending on the contents of the edition, the file type will be `'edtp'` (for graphics), `'edtt'` (for text), or `'edts'` (for sound).

After creating the edition container, use the `NewSection` function to create the section record and alias record for the section.

Listing 2-4 illustrates how to create a publisher. The `DoNewPublisher` function shown in the listing is a function provided by an application. Note that an application might call the `DoNewPublisher` function in response to the user's choosing the Create Publisher command or in response to the Create Publisher event. The chapter "Responding to Apple Events" in this book gives an example of a handler for the Create Publisher event.

The parameters to the `DoNewPublisher` function include a pointer to information about the document, a Boolean value that indicates if the function should display the new publisher dialog box, the preview for the edition, the preview format, and an edition container.

The function displays the publisher dialog box if requested, letting the user accept or change the name of the edition and the location where the edition should reside. Use the `CreateEditionContainerFile` function to create the edition with the given name and location. Use the `NewSection` function to create a new section for the publisher.

After the section is created, you must write out the edition data. Be sure to add the newly created section to your list of sections for this document. There are several different techniques for creating publishers and unique IDs; this listing displays one technique.

After creating the edition container and creating a new section record, the `DoNewPublisher` function calls another application-defined routine, `DoWriteEdition`, to open the edition and write data to it.

Listing 2-4 Creating a publisher

```
FUNCTION DoNewPublisher(thisDocument: MyDocumentInfoPtr;
                        promptForDialog: Boolean;
                        preview: Handle;
                        previewFormat: FormatType;
                        editionSpec: EditionContainerSpec)
                        : OSErr;
VAR
   getLastErr, dialogErr:  OSErr;
   createErr, sectionErr:  OSErr;
   resID:                  Integer;
   thisSectionH:           SectionHandle;
   reply:                  NewPublisherReply;
BEGIN
   {set up info for new publisher reply record}
   reply.replacing := FALSE;
   reply.usePart := FALSE;
   reply.preview := preview;
   reply.previewFormat := previewFormat;
   reply.container := editionSpec;
   IF promptForDialog THEN
   BEGIN {user interaction is allowed}
      {display dialog box and let user select}
      dialogErr := NewPublisherDialog(reply);
      {dispose of preview data handle}
      DisposeHandle(reply.preview);
      IF dialogErr <> noErr THEN MyErrHandler(dialogErr);
      IF reply.canceled THEN
      BEGIN    {do nothing if user canceled}
         DoNewPublisher := userCanceledErr;
         EXIT(DoNewPublisher);
      END;
   END;  {of promptForDialog}
```

```
   IF NOT reply.replacing THEN
   BEGIN
      {if user isn't replacing an existing file, create a new one}
      createErr :=
         CreateEditionContainerFile(reply.container.theFile,
                                    kAppSignature,
                                    reply.container.theFileScript);
      IF createErr <> noErr THEN
      BEGIN
         DoNewPublisher := errAEPermissionDenied;
         EXIT(DoNewPublisher);
      END;
   END; {of not replacing}
   {Advance counter to make a new unique sectionID for this }
   { document. It is not required that you equate section IDs }
   { with resources.}
   thisDocument^.nextSectionID := thisDocument^.nextSectionID + 1;
   {create a publisher section}
   sectionErr := NewSection(reply.container,
                            thisDocument^.fileSpecPtr,
                            stPublisher,
                            thisDocument^.nextSectionID,
                            pumOnSave, thisSectionH);
   IF (sectionErr <> noErr) & (sectionErr <> multiplePublisherWrn)
         & (sectionErr <> notThePublisherWrn) THEN
      MyErrHandler(sectionErr);
   resID := thisDocument^.nextSectionID;
   {add this section/alias pair to app's internal bookkeeping}
   MyAddSectionAliasPair(thisDocument, thisSectionH, resID);
   {write out first edition}
   DoWriteEdition(thisSectionH);
   {Remember that the section and alias records need to be }
   { saved as resources when the user saves the document.}
   {set the function result appropriately}
   DoNewPublisher := MyGetLastError;
END;
```

Opening an Edition Container to Write Data

Several routines are required to write (publish) data from a publisher to an edition container. (For information on creating an edition container, see the previous section.) Before writing data to an edition, you must use the `OpenNewEdition` function. This function should be used only for a publisher within a document. Use this function to initiate the writing of data to an edition.

```
err := OpenNewEdition(publisherSectionH, fdCreator,
                      publisherSectionDocument, refNum);
```

A user may try to save a document containing a publisher that is unable to write its data to an edition—because another publisher (that shares the same edition) is writing, another subscriber (that shares the same edition) is reading, or a publisher located on another computer is registered to the section. In such a case, you may decide to refrain from writing to the edition so that the user does not have to wait. You should also refrain from displaying an error to the user. The contents of the publisher are saved to disk with the document. The next time that the user saves the document, you can write the publisher data to the edition. You should display an alert box to discourage users from making multiple copies of the same publisher and pasting them in the same or other documents (see "Duplicating Publishers and Subscribers" on page 2-58).

If a user clicks Send Edition Now within the publisher options dialog box (to write publisher data to an edition manually), and the publisher is unable to write its data to its edition (for any of the reasons outlined above), you should display an error message.

After you are finished writing data to an edition, use the `CloseEdition` function to close the edition.

Listing 2-5 illustrates how to write data to an edition. For an existing edition container, you must open the edition, write each format using the `WriteEdition` function, and close the edition using the `CloseEdition` function. This listing shows how to write text only. If the edition is written successfully, subscribers receive Section Read events.

Listing 2-5 Writing data to an edition

```
PROCEDURE DoWriteEdition(thePublisher: SectionHandle);
VAR
   eRefNum:       EditionRefNum;
   openErr:       OSErr;
   writeErr:      OSErr;
   closeErr:      OSErr;
   thisDocument:  MyDocumentInfoPtr;
   textHandle:    Handle;
BEGIN
   {find out which document this section belongs to}
   thisDocument := MyFindDocument(thePublisher);
   {open edition for writing}
   openErr := OpenNewEdition(thePublisher, kAppSignature,
                             thisDocument^.fileSpecPtr, eRefNum);
   IF openErr <> noErr THEN
      MyErrHandler(openErr);{handle error and exit}
   {get the text data to write}
   textHandle := MyGetTextInSection(thePublisher, thisDocument);
   {write out text data}
   HLock(textHandle);
   writeErr := WriteEdition(eRefNum, 'TEXT', textHandle^,
                            GetHandleSize(textHandle));
   HUnLock(textHandle);
   IF writeErr <> noErr THEN
   BEGIN
      {There were problems writing; simply close the edition. }
      { When successful = FALSE, the edition data <> section }
      { data. Note: this isn't fatal or bad; it just means }
      { that the data wasn't written and no Section Read events }
      { will be generated.}
      closeErr := CloseEdition(eRefNum, FALSE);
   END
   ELSE
   BEGIN
      {The write was successful; now close the edition. }
      { When successful = TRUE, the edition data = section data.}
      { This edition is now available to any subscribers. }
      { Section Read events will be sent to current subscribers.}
      closeErr := CloseEdition(eRefNum, TRUE);
   END;
END;
```

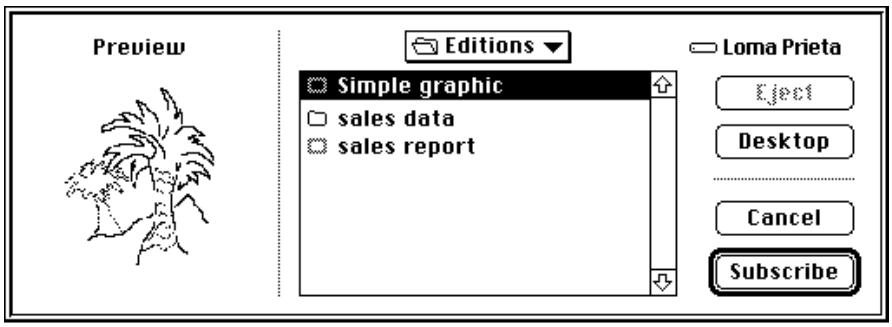
Creating a Subscriber

You need to create a Subscribe To menu command in the Edit menu. When a user chooses Subscribe To from this menu, your application should display the subscriber dialog box on the user's screen.

Use the `NewSubscriberDialog` function to display the subscriber dialog box on the user's screen. This function is similar to the `CustomGetFile` procedure described in the chapter "Standard File Package" in *Inside Macintosh: Files*.

To create a subscriber, you must get information from the user, such as the name of the edition being subscribed to. The dialog box displays a listing of all available editions and allows the user to see a preview (thumbnail sketch) of the edition selected. Figure 2-12 shows a sample subscriber dialog box.

Figure 2-12 A sample subscriber dialog box



The subscriber dialog box allows the user to choose an edition to subscribe to. The `NewSubscriberDialog` function handles all user interaction until a user clicks Subscribe or Cancel. When a user selects an edition container, the Edition Manager accesses the preview for the edition container (if it is available) and displays it.

You pass a new subscriber reply record as a parameter to the `NewSubscriberDialog` function.

```
TYPE  NewSubscriberReply =
      RECORD
         canceled:   Boolean;    {user clicked Cancel}
         formatsMask:SignedByte; {formats required}
         container:  EditionContainerSpec;{initially, default }
                                  { name & location of edition }
                                  { to subscribe to; on return, }
                                  { edition name & location }
                                  { chosen by the user}
      END;
```

The `canceled` field returns a Boolean value of `TRUE` if the user clicked Cancel. To indicate which edition format types (text, graphics, or sound) your application can read, you set the `formatsMask` field to one or more of these constants:

```
CONST kPICTformatMask   = 1;          {can subscribe to 'PICT'}
      kTEXTformatMask   = 2;          {can subscribe to 'TEXT'}
      ksndFormatMask    = 4;          {can subscribe to 'snd '}
```

To support a combination of formats, add the constants together. For example, a `formatsMask` of 3 displays both graphics and text edition format types in the subscriber dialog box.

The `container` field is of data type `EditionContainerSpec`. You must initialize the `container` field with the default edition volume reference number, directory ID, filename, and part. To do so, use the `GetLastEditionContainerUsed` function to obtain the name of the last edition displayed in the dialog box.

```
err := GetLastEditionContainerUsed(container);
```

This function returns the last edition container for which a new section was created using the `NewSection` function. If there is no last edition, or if the edition was deleted, `GetLastEditionContainerUsed` still returns the correct volume reference number and directory ID to use, but leaves the filename blank and returns the `fnfErr` result code.

The `container` field is of data type `EditionContainerSpec`.

```
TYPE EditionContainerSpec =
   RECORD
      theFile:          FSSpec;        {file containing edition }
                                       { data}
      theFileScript:    ScriptCode;    {script code of filename}
      thePart:          LongInt;       {which part of file, }
                                       { always kPartsNotUsed}
      thePartName:      Str31;         {reserved}
      thePartScript:    ScriptCode;    {reserved}
   END;
```

The field `theFile` is of type `FSSpec`. See *Inside Macintosh: Files* for further information on file system specification records.

After filling in the fields of the new subscriber reply record, pass it as a parameter to the `NewSubscriberDialog` function, which displays the subscriber dialog box.

```
err := NewSubscriberDialog(reply);
```

After displaying the subscriber dialog box, call the `NewSection` function to create the section record and the alias record. See "Creating the Section Record and Alias Record" beginning on page 2-15 for detailed information.

If the subscriber is set up to receive new editions automatically (not manually), the Edition Manager sends your application a Section Read event. Whenever your application receives a Section Read event, it should read the contents of the edition into the subscriber.

Listing 2-6 illustrates how to create a subscriber. As described earlier, you must set up and display the subscriber dialog box to allow the user to subscribe to any of the available editions. After your application creates a subscriber, your application receives a Section Read event to read in the data being subscribed to. Be sure to add the newly created section to your list of sections for this file. There are many different techniques for creating subscribers and unique IDs; this listing displays one technique.

Listing 2-6 Creating a subscriber

```
PROCEDURE DoNewSubscriber(thisDocument: MyDocumentInfoPtr);
VAR
   getLastErr:    OSErr;
   dialogErr:     OSErr;
   sectionErr:    OSErr;
   resID:         Integer;
   thisSectionH:  SectionHandle;
   reply:         NewSubscriberReply;
BEGIN
   {put default edition name into reply record}
   getLastErr := GetLastEditionContainerUsed(reply.container);
   {can subscribe to pictures or text}
   reply.formatsMask := kPICTformatMask + kTEXTformatMask;
   {display dialog box & let user select edition to subscribe to}
   dialogErr := NewSubscriberDialog(reply);
   IF dialogErr <> noErr THEN
      MyErrHandler(dialogErr);   {handle error and exit}
   IF reply.canceled THEN
      EXIT(DoNewSubscriber);     {do nothing if user canceled}
   {Advance counter to make a new unique sectionID for this }
   { document. It is not necessary to equate section IDs with }
   { resources.}
   thisDocument^.nextSectionID := thisDocument^.nextSectionID + 1;
   {create a subscriber section}
   sectionErr := NewSection(reply.container,
                            thisDocument^.fileSpecPtr,
                            stSubscriber,
                            thisDocument^.nextSectionID,
                            sumAutomatic, thisSectionH);
   IF sectionErr <> noErr THEN
      MyErrHandler(sectionErr);{handle error and exit}
   resID := thisDocument^.nextSectionID;
   {add this section/alias pair to app's internal bookkeeping}
   MyAddSectionAliasPair(thisDocument, thisSectionH, resID);
   {Remember that you will receive a Section Read event to read }
   { in the edition that you just subscribed to because the }
   { initial mode is set to sumAutomatic.}
   {Remember that the section and alias records need to be saved }
   { as resources when the user saves the document.}
END;
```

Opening an Edition Container to Read Data

Before reading data from an edition, you must use the `OpenEdition` function. Your application should only use this function for a subscriber. Use this function to initiate the reading of data from an edition.

```
err := OpenEdition(subscriberSectionH, refNum);
```

As a precaution, you should retain the old data until the user can no longer undo. This allows you to undo changes if the user requests it.

Your application can supply a procedure such as `DoReadEdition` to read in data from the edition to a subscriber. When your application opens a document containing a subscriber that is set up to receive new editions automatically, the Edition Manager sends you a Section Read event if the edition has been updated. The Section Read event supplies the handle to the section that requires updating. Listing 2-7, shown in the next section, provides an example of reading data from an edition.

Choosing Which Edition Format to Read

After your application opens the edition container for a subscriber, it can look in the edition for formats that it understands. To accomplish this, use the `EditionHasFormat` function.

```
err := EditionHasFormat(whichEdition, whichFormat, formatSize);
```

The `EditionHasFormat` function returns the `noTypeErr` result code if a requested format is not available. If the requested format is available, this function returns the `noErr` result code, and the `formatSize` parameter contains the size of the data in the specified format or `kFormatLengthUnknown` (–1), which signifies that the size is unknown.

Note
The Translation Manager (if it is available) attempts implicit translation under certain circumstances. For instance, it does so when your application attempts to read from an edition a format type that is not in the edition. In this case, the Translation Manager attempts to translate the data into the requested format. For more information, see the chapter "Translation Manager" in *Inside Macintosh: More Macintosh Toolbox*. ◆

After your application opens the edition container and determines which formats it wants to read, call the `ReadEdition` function to read in the edition data. See "Reading and Writing Edition Data" on page 2-27 for detailed information.

After you have completed writing the edition data into the subscriber section, call the `CloseEdition` function to close the edition. See "Closing an Edition" on page 2-28 for detailed information.

Listing 2-7 illustrates how to read data from an edition. As described earlier, you must open the edition, determine which formats to read, use the `ReadEdition` function to read in data, and then use the `CloseEdition` function to close the edition. This listing shows how to read only text.

Listing 2-7 Reading in edition data

```
PROCEDURE DoReadEdition(theSubscriber: SectionHandle);
VAR
   eRefNum:          EditionRefNum;
   openErr:          OSErr;
   readErr:          OSErr;
   closeErr:         OSErr;
   thisDocument:     MyDocumentInfoPtr;
   textHandle:       Handle;
   formatLen:        Size;
BEGIN
   {find out which document this section belongs to}
   thisDocument := MyFindDocument(theSubscriber);
   {open the edition for reading}
   openErr := OpenEdition(theSubscriber, eRefNum);
   IF openErr <> noErr THEN
      MyErrHandler(openErr);  {handle error and exit}
   {look for 'TEXT' format}
   IF EditionHasFormat(eRefNum, 'TEXT', formatLen) = noErr THEN
   BEGIN
      {get the handle of location to read to}
      textHandle := MyGetTextInSection(theSubscriber,
                                       thisDocument);
      SetHandleSize(textHandle, formatLen);
      HLock(textHandle);
      readErr := ReadEdition(eRefNum, 'TEXT', textHandle^,
                             formatLen);
      MyUpdateSubscriberText(theSubscriber, textHandle, readErr);
      HUnLock(textHandle);
      IF readErr = noErr THEN
      BEGIN
         {The read was successful; now close the edition. When }
         { successful = TRUE, the section data = edition data.}
         closeErr := CloseEdition(eRefNum, TRUE);
         EXIT(DoReadEdition);
      END;
   END;  {of EditionHasFormat}
```

```
   {'TEXT' format wasn't found or read error; just close }
   { the edition. FALSE tells the Edition Manager that your }
   { application did not get the latest edition.}
   closeErr := CloseEdition(eRefNum, FALSE);
END;
```

Using Publisher and Subscriber Options

You can allow users to set several special options associated with publishers and subscribers. To set these preferences, users change settings in two dialog boxes provided by the Edition Manager: publisher options and subscriber options. To make these dialog boxes available to the user, provide a command in the Edit menu that toggles between Publisher Options (when the user has selected a publisher within a document) and Subscriber Options (when a user has selected a subscriber within a document).

When a user chooses one of these menu commands, you need to display the appropriate dialog box. Use the `SectionOptionsDialog` function to display the publisher options or subscriber options dialog box on the user's screen.

```
err := SectionOptionsDialog(reply);
```

Each dialog box contains information regarding the section and its edition. Figure 2-13 shows the publisher options dialog box with the update mode set to On Save.

Figure 2-13 The publisher options dialog box with update mode set to On Save

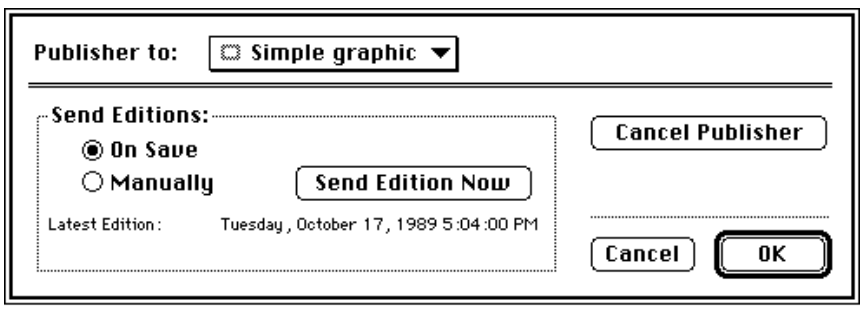
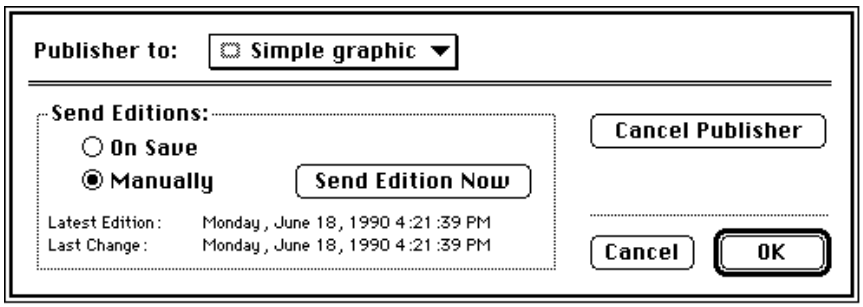


Figure 2-14 shows the publisher options dialog box with the update mode set to Manually.

Figure 2-14 The publisher options dialog box with update mode set to Manually



As a shortcut for the user, you should display the publisher options dialog box when the user double-clicks a publisher section in a document.

Figure 2-15 shows the subscriber options dialog box with the update mode set to Automatically.

Figure 2-15 The subscriber options dialog box with update mode set to Automatically

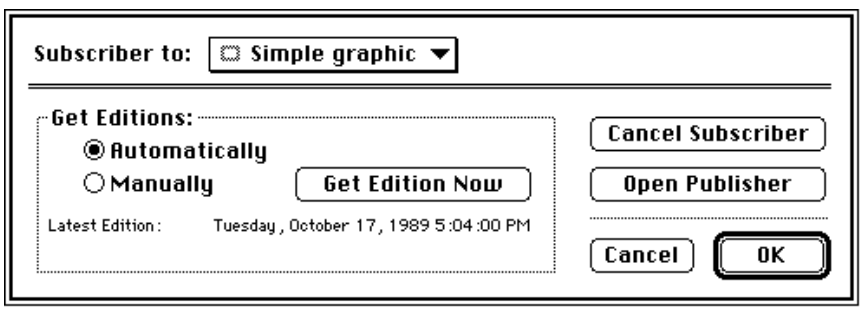
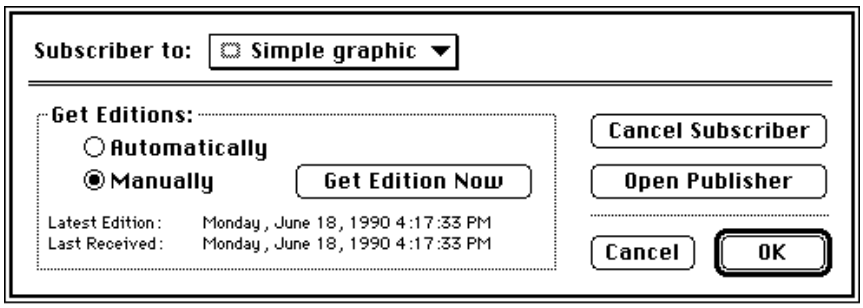


Figure 2-16 shows the subscriber options dialog box with the update mode set to Manually.

Figure 2-16 The subscriber options dialog box with update mode set to Manually



As a shortcut for the user, you should display the subscriber options dialog box when the user double-clicks a subscriber section in a document.

You pass a section options reply record as a parameter to the `SectionOptionsDialog` function.

```
TYPE SectionOptionsReply =
   RECORD
      canceled:   Boolean;          {user clicked Cancel}
      changed:    Boolean;          {changed section record}
      sectionH:   SectionHandle;    {handle to the specified }
                                    { section record}
      action:     ResType;          {action codes}
   END;
```

Set the `sectionH` parameter to the handle to the section record for the section the user selected.

Upon return of the `SectionOptionsDialog` function, the `canceled` and `changed` fields are set. If the `canceled` field is set to `TRUE`, the user clicked Cancel. Otherwise, this field is set to `FALSE`. If the `changed` field is set to `TRUE`, the section record is changed. For example, the user may have changed the update mode.

The `SectionOptionsDialog` function returns in the `action` parameter the code for one of five user actions. The function dismisses the publisher and subscriber options dialog boxes after the user clicks a button.

- Action code is `'read'` for a click of the Get Edition Now button.
- Action code is `'writ'` for a click of the Send Edition Now button.
- Action code is `'goto'` for a click of the Open Publisher button.
- Action code is `'cncl'` for a click of the Cancel Publisher or Cancel Subscriber button.
- Action code is `'    '` ($20202020) for a click of the OK button.

Listing 2-8 shows an example of how your application can respond to the action codes received from the section options reply record. You can use several different techniques for this purpose; this listing shows one technique.

Listing 2-8 Responding to action codes

```
PROCEDURE DoOptionsDialog(theSection: SectionHandle);
VAR
   reply:            SectionOptionsReply;
   theEditionInfo:   EditionInfoRecord;
   action:           ResType;
   sodErr, geiErr:   OSErr;
   gpiErr, gpsErr:   OSErr;

BEGIN
   reply.sectionH := theSection;
   {display options dialog box}
   sodErr := SectionOptionsDialog(reply);
   {determine what the user did and handle appropriately}
   IF reply.canceled THEN  {user selected the Cancel button}
      EXIT(DoOptionsDialog);
   IF reply.changed THEN
      {the section record has changed; make note of this}
      MySectionHasChanged(theSection);
      {if you customize, you may want to do some }
      { post-processing now}
   {get the action code}
   action := reply.action;
   IF (action = 'read') THEN
   BEGIN    {user selected Get Edition Now button}
      DoReadEdition(theSection);
      EXIT(DoOptionsDialog);
   END;
```

```
   IF (action = 'writ') THEN
   BEGIN    {user selected Send Edition Now button}
      DoWriteEdition(theSection);
      EXIT(DoOptionsDialog);
   END;
   IF (action = 'goto') THEN
   BEGIN    {user selected Open Publisher button}
      geiErr := GetEditionInfo(theSection, theEditionInfo);
      IF geiErr <> noErr THEN
         MyErrHandler(geiErr);{handle error and exit}
      gpsErr := GotoPublisherSection(theEditionInfo.container);
      IF gpsErr <> noErr THEN
         MyErrHandler(gpsErr);{handle error and exit}
      EXIT(DoOptionsDialog);
   END;
   IF (action = 'cncl') THEN
   BEGIN {User selected Cancel Publisher or Cancel Subscriber }
      { button. Call the UnRegisterSection function and dispose }
      { of the section record and alias record.}
      EXIT(DoOptionsDialog);
   END;
END;
```

The following sections describe the features of the publisher and subscriber options dialog boxes.

Publishing a New Edition While Saving or Manually

By default, your application should write publisher data to an edition each time the user saves the document and the contents of the publisher differ from the latest edition. In the publisher options dialog box, the user can choose to write new data to an edition each time the document is saved (by clicking On Save) or only upon the user's specific request (by clicking Manually).

When the update mode is set to manual, a user must click the Send Edition Now button in the publisher options dialog box to write publisher data to an edition. When a user clicks this button, the section options reply record contains the action code `'writ'`. In this case, you should write out the new edition. Writing to an edition manually is useful when a user tends to save a document numerous times while revising it.

Each time the user saves the document, check the update mode of the publisher section. If the publisher section sends its data to an edition when the document is saved, check whether the publisher data has changed since it was last written to the edition. If so, write the publisher's data to the new edition.

In addition, you may also support a Stop All Editions menu command to provide a method for temporarily suspending all update activity. See "Introduction to Publishers, Subscribers, and Editions" beginning on page 2-4 for additional information.

Subscribing to an Edition Automatically or Manually

By default, your application should subscribe to an edition each time new edition data becomes available. In the subscriber options dialog box, the user can choose to read new data from an edition as the data is available (by clicking Automatically) or only upon the user's specific request (by clicking Manually).

When the update mode is set to manual, the user must click the Get Edition Now button in the subscriber options dialog box to receive new editions. When a user clicks this button, the section options reply record contains the action code `'read'`. In this case, you should read in the new edition. See "Opening an Edition Container to Read Data" beginning on page 2-41 for detailed information.

When the update mode is set to automatic, your application receives a Section Read event each time a new edition becomes available. In response, you should read the new edition data beginning with the `OpenEdition` function.

Your application does not receive Section Read events for subscribers that receive new editions manually.

You may also support a Stop All Editions menu command to provide a method for temporarily suspending all update activity. See "Introduction to Publishers, Subscribers, and Editions" beginning on page 2-4 for additional information.

Canceling Sections Within Documents

The option of canceling publishers and subscribers is available to the user through the Cancel Publisher and Cancel Subscriber buttons in the corresponding options dialog boxes. When the user clicks one of these buttons, the action code of the section options reply record is `'cncl'`. See "Relocating an Edition" on page 2-60 for additional information on canceling a section.

When a user cancels a section (either a publisher or subscriber) and then saves the document, or when a user closes an untitled document (which contains newly created sections) without saving it, you must unregister each corresponding section record and alias record using the `UnRegisterSection` function. In addition, you should also delete the section record and alias record using the `DisposeHandle` procedure. See *Inside Macintosh: Memory* for additional information on the `DisposeHandle` procedure.

When a user cancels a publisher section and then saves the document, or when a user closes an untitled document (which contains newly created publishers) without saving it, you must also delete any corresponding edition containers (in addition to deleting section records and alias records).

Do not delete an edition container file, section record, or alias record until the user saves the document; the user may decide to undo changes before saving the document.

To locate the appropriate edition container to be deleted (before you use the `UnRegisterSection` function), use the `GetEditionInfo` function.

```
err := GetEditionInfo(sectionH, editionInfo);
```

The `editionInfo` parameter is a record of data type `EditionInfoRecord`.

```
TYPE EditionInfoRecord =
   RECORD
      crDate:     TimeStamp;           {date edition container }
                                       { was created}
      mdDate:     TimeStamp;           {date of last change}
      fdCreator:  OSType;              {file creator}
      fdType:     OSType;              {file type}
      container:  EditionContainerSpec;{the edition}
   END;
```

The `GetEditionInfo` function returns the edition container as part of the edition information.

The `crDate` field contains the creation date of the edition. The `mdDate` field contains the modification date of the edition.

The `fdType` and the `fdCreator` fields are the type and creator of the edition file. The `container` field includes a volume reference number, directory ID, filename, script, and part number for the edition.

To remove the edition container, use the `DeleteEditionContainerFile` function.

```
err := DeleteEditionContainerFile(editionFile);
```

Locating a Publisher Through a Subscriber

The user can locate a publisher from a subscriber within a document by clicking the Open Publisher button in the subscriber options dialog box. As a shortcut, Apple suggests that you also allow the user to locate a publisher by selecting a subscriber in a document and pressing Option–double-click.

When the action code of the `SectionOptionsReply` record is `'goto'`, use the `GoToPublisherSection` function.

```
err := GoToPublisherSection(container);
```

The `GoToPublisherSection` function locates the correct document by resolving the alias in the edition, and it launches the document's application if necessary (the Edition Manager sends an Open Documents event). The Edition Manager then sends the publishing application a Section Scroll event. If the document containing the requested publisher is located on the same computer as its subscriber, the document opens and scrolls to the location of the publisher. If the document containing the requested publisher is located on a shared volume (using file sharing), the document opens and scrolls to the location of the publisher only if the user has privileges to open the document from the Finder.

You need to provide the `GoToPublisherSection` function with the edition container. To accomplish this, use the `GetEditionInfo` function. See the previous section, "Canceling Sections Within Documents," for information on the `GetEditionInfo` function.

Renaming a Document Containing Sections

If a user renames a document that contains sections by choosing Save As from the File menu, or if a user pastes a portion of a document that contains a section into another document, use the `AssociateSection` function.

Use the `AssociateSection` function to update the alias record of a registered section.

```
err := AssociateSection (sectionH, newSectionDocument);
```

The `AssociateSection` function internally calls the `UpdateAlias` function. It is also possible to update the alias record using the Alias Manager (see the chapter "Alias Manager" in *Inside Macintosh: Files* for additional information).

Displaying Publisher and Subscriber Borders

Each publisher and subscriber within a document should have a border that appears when a user selects the contents of these sections. You should display a publisher border as three pixels wide with 50 percent gray lines and a subscriber border as three pixels wide with 75 percent gray lines. Separate the contents of the section from the border itself with one pixel of white space. To create your borders, you should use patterns, not colors. Depending on the user's monitor type, colors may not be distinguishable.

In general, borders for publishers and subscribers should behave like the borders of `'PICT'` graphics in a word-processing document. A border should appear when the user clicks the content area of a publisher or a subscriber and disappear when the user clicks outside the content area of a section. You can also make all publisher and subscriber borders appear or disappear by implementing an optional Show/Hide Borders menu command.

Figure 2-17 displays the Edition Manager Show/Hide Borders menu command in the Edit menu.

Figure 2-17 Edit menu with Show/Hide Borders menu command

Depending on your application, you may choose to include resize handles or similar components in your borders. See "Object-Oriented Graphics Borders" on page 2-56 for an example of resize handles.

Whenever a user selects a portion of a publisher or sets the insertion point within a publisher, you should display the border as 50 percent gray. A user can copy the contents of a publisher or subscriber without copying the section itself by selecting the data, copying, and then pasting the data in a new location. A user can cut and paste a selection that contains an *entire* publisher or subscriber, but you should discourage users from making multiple copies of a publisher. See "Duplicating Publishers and Subscribers" on page 2-58 for detailed information.

When the user modifies a publisher, your application should grow or shrink its border to accommodate the new dimension of the section.

You should display only one publisher border within a document at a time. If a cursor is inserted within a publisher that is contained within a larger publisher, you should display only the smaller, internal publisher border. If it is absolutely necessary to display all section borders within a document at the same time, you can create a Show/Hide Borders menu item.

You do not need to provide support for publishers contained within other publishers. If you do not, you should dim the Create Publisher menu command (to indicate that it is not selectable) when a user attempts to create a publisher within an existing publisher.

Figure 2-18 shows the recommended border behavior for publishers. The top window shows a publisher with its borders displayed. The middle window shows how the borders look when a user selects some of the contents of a section. The bottom window shows how the borders look when a user selects data within a document that includes a publisher section.

Figure 2-18 Publisher borders

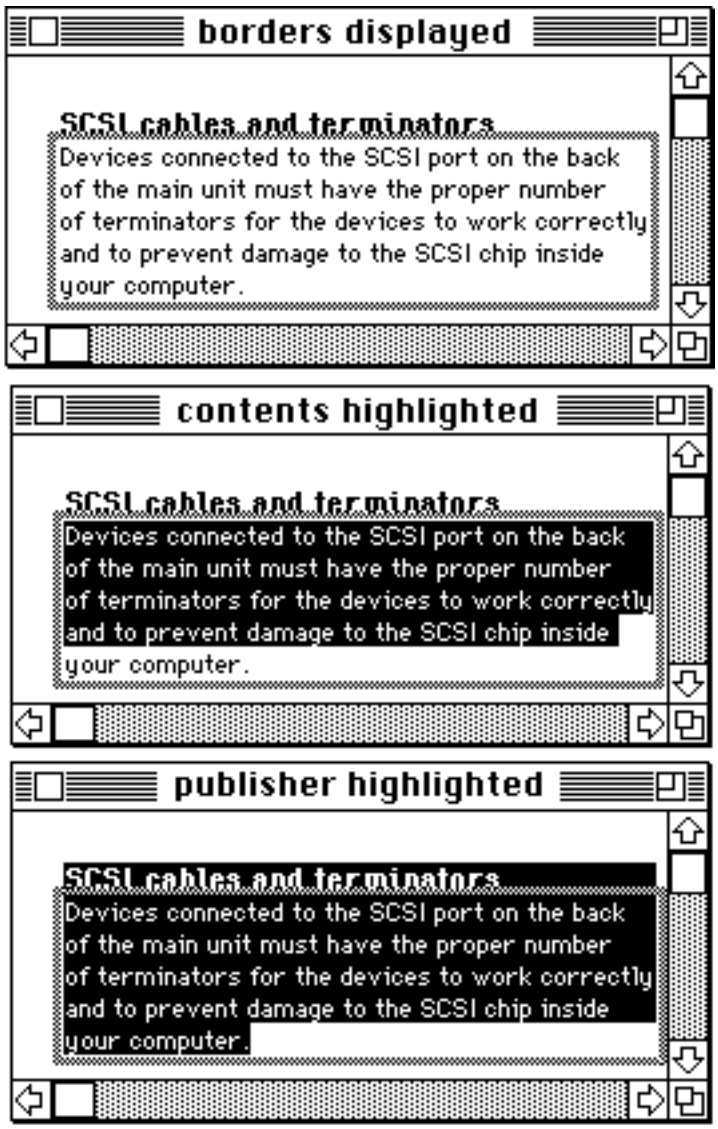
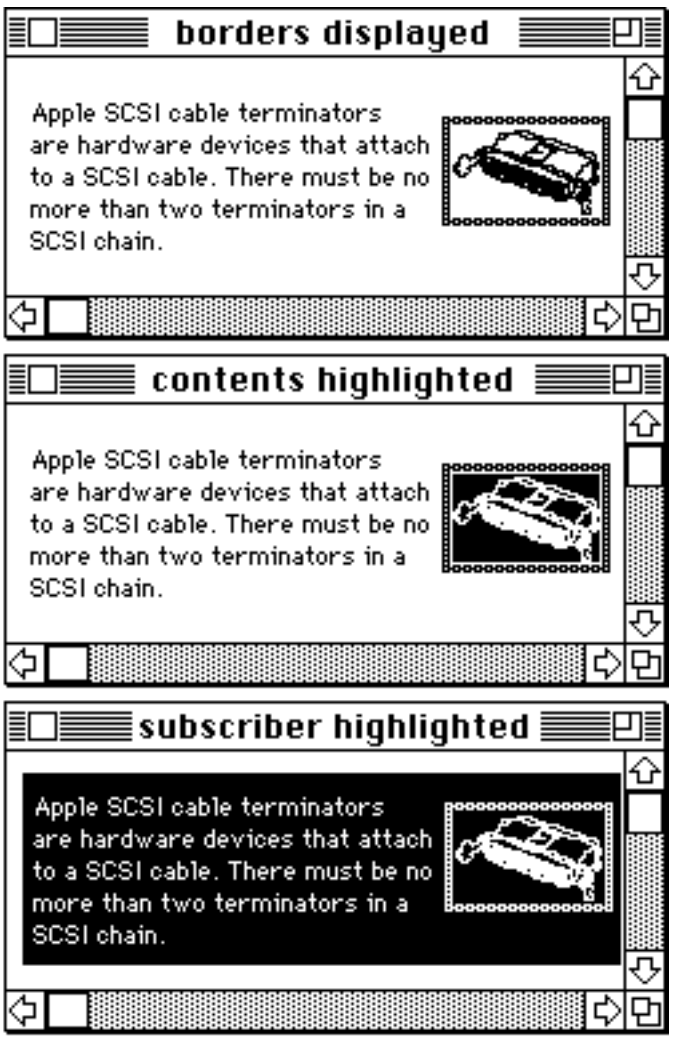


Figure 2-19 shows the recommended border behavior for subscribers. The top window shows a subscriber with its borders displayed. The middle window shows how the borders look when a user selects the contents of a section. The bottom window shows how the borders look when a user selects data within a document that includes a subscriber section.

Figure 2-19 Subscriber borders



If a user tries to select only a portion of a subscriber, you should highlight the entire contents of the subscriber. A user cannot edit the data in a subscriber. See “Modifying a Subscriber” on page 2-59 for detailed information.

If a user cancels a section using the publisher or subscriber options dialog box, your application should leave the contents of the section within the document, but you should be sure to remove the borders from this data, as it is no longer considered a section.

Generally, the appearance and function of publisher and subscriber borders should be the same across different applications. The following sections entitled "Text Borders," "Spreadsheet Borders," "Object-Oriented Graphics Borders," and "Bitmapped Graphics Borders" describe specialized features for publisher and subscriber borders in word-processing, spreadsheet, or graphics applications.

Text Borders

In word-processing documents, a publisher may contain other publishers. However, one publisher should not *overlap* another publisher. You should display only one publisher border at a time. If an insertion point is placed within a publisher that is encompassed by another larger publisher, you should display only the smaller internal publisher border.

In exceptional cases, it may be necessary to display more than one publisher or subscriber border at a time. For example, a publisher may consist of a paragraph that includes a marker for a footnote. The data contained within the footnote should also be considered part of the publisher. When a user selects the paragraph, you should simultaneously display a border around the footnote.

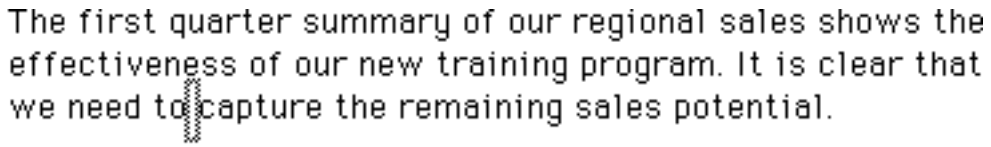
The border of a publisher that contains text should be located between characters within the text. The insertion point, when placed on such a boundary, should gravitate toward the publisher. That is, a click in front (to the left) of a publisher border should place the cursor inside the publisher, so that subsequent typing goes inside the publisher. Clicking at the end (to the right) of a publisher border should also place the cursor inside the publisher.

Whenever two separate borders are adjacent, the boundary click should go in between them. This is also true for a border that is next to other nontextual aspects of a document, such as `'PICT'` graphics or page breaks.

When a user removes information from a publisher that contains text data, you should resize the border so that it becomes smaller. When a user adds information to the publisher, you should enlarge the border to accommodate the new text. The insertion point should remain within the publisher.

If a user highlights the entire contents of a publisher and then chooses Cut from the Edit menu, you should not delete the publisher border within the document. The user may intend to delete the existing publisher data and replace it with new data, or the user may want to move the entire publisher and its data to a new location. Figure 2-20 shows this state.

Figure 2-20 A publisher with contents removed



You should leave the cursor inside the small publisher border for further typing. If the user inserts the cursor in a new location (instead of typing data inside the existing border), you need to remove the empty publisher border from the document to allow the user to move the publisher. This effectively deletes the publisher from the document. If the user pastes the publisher that is currently held in the scrap, you should re-create its border. If the user cuts or copies other data from the document before pasting the publisher from the scrap, the publisher should be removed from the scrap.

Spreadsheet Borders

Borders around spreadsheet data or other data in arrays should look and behave very much like text borders. Figure 2-21 shows a typical border within a spreadsheet document.

Figure 2-21 A publisher border within a spreadsheet document

	A	B	C	D	E
1		January	February	March	
2	Cogs	14890	17849	274945	
3	Sprockets	16494	184384	304890	
4	Widgets	3780	5839	7900	
5					

Note that the border goes below the column headers (A, B, C, D) and to the right of the row labels (1, 2, 3, 4)—it should not overlap these cell boundaries. The border at the bottom and the border on the right side can be placed within the adjacent cells (outside of the cells that constitute the publisher).

Unlike borders in word-processing applications, borders in spreadsheet documents (or other documents with array data) can overlap. That is, a user can select a row of cells to be a publisher and an overlapping column of those cells to be another publisher. You should never display more than one publisher border at a time. When a user selects a spreadsheet cell that is part of more than one publisher, you should display only the border of the publisher that was last edited. (This can be accomplished by comparing the modification dates of the publishers.)

If it is absolutely necessary to display all section borders within a document at the same time, you can create a Show/Hide Borders command in the Edit menu to toggle all borders on and off.

When data is added to or deleted from a publisher that consists of a spreadsheet cell or other array, you should resize its border to accommodate the addition or deletion of data. A publisher should behave like a named range in a spreadsheet. For example, if a user cuts a row within a publisher that consists of a named range in a spreadsheet, you should shrink the publisher data and its border correspondingly.

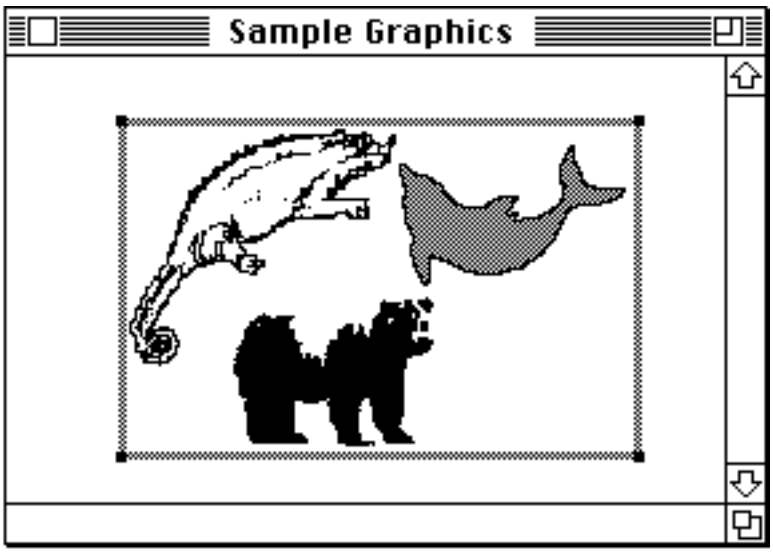
When a user cuts a publisher and its entire contents within a spreadsheet document, the entire section should be held in the scrap. Do not leave an empty publisher border in a spreadsheet (as recommended for text borders). If a user attempts to paste a copy of an existing publisher, you should warn the user by displaying an alert box (see "Duplicating Publishers and Subscribers" on page 2-58).

Object-Oriented Graphics Borders

In an object-oriented drawing application, the publisher border should fit just around the selected objects.

You can provide resize handles that appear with all drawing objects to allow the user to resize the border of a publisher. Figure 2-22 shows a publisher border with resize handles.

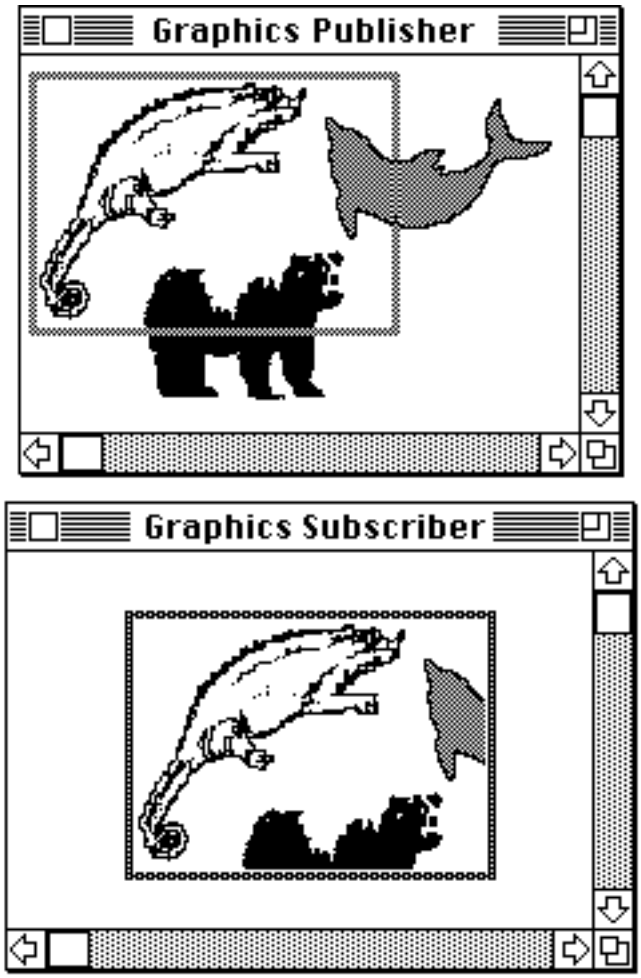
Figure 2-22 A publisher border with resize handles



An application can make publisher borders appear to float over the area the user publishes. The border acts like a clipping rectangle—anything within the border becomes the publisher. Figure 2-23 shows a publisher that contains clipped graphics and its subscriber in another application.

A user can create publishers and subscribers that overlap each other. Thus, borders may overlap and it may no longer be possible to turn on a particular border when the user clicks within a publisher. Drawing applications should provide a menu command, Show Borders, that toggles to Hide Borders. This command should allow users to turn all publisher and subscriber borders on or off.

Figure 2-23 A publisher and subscriber with clipped graphics



Bitmapped Graphics Borders

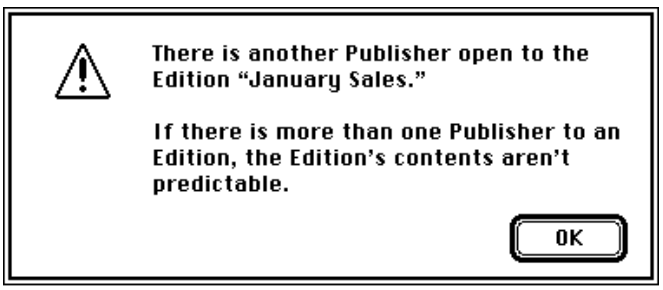
Creating a border around bitmapped graphics in applications is similar to doing so in object-oriented drawing applications. The border appears around the selected area. The user can create overlapping publishers and subscribers in bitmapped graphics applications. You need to provide a Show/Hide Borders command to allow users to turn all borders on and off.

Duplicating Publishers and Subscribers

Whenever a user clicks a publisher or subscriber border, you should change the contents of the section to a selected state. You should discourage users from making multiple copies of a publisher and pasting them in the same or other documents, because the contents of the edition would be difficult or impossible to predict. Multiple copies of the same publisher also contain the same control block value. See "Creating and Registering a Section" on page 2-74 for detailed information on control blocks.

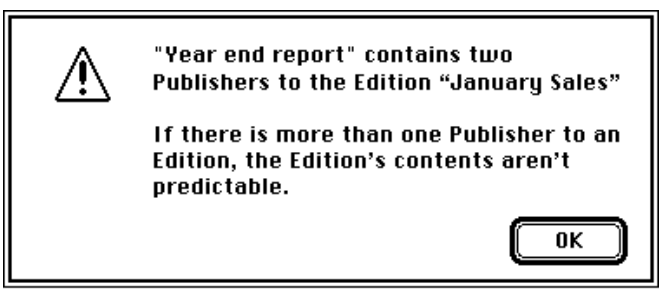
When a user attempts to create a copy of a publisher that already exists, you should display an alert box such as the one shown in Figure 2-24.

Figure 2-24 Creating multiple publishers alert box



When a user attempts to save a document that contains multiple copies of the same publisher, display an alert box such as the one shown in Figure 2-25.

Figure 2-25 Saving multiple publishers alert box



If a user decides to ignore your alert box, your application should still save the document, but you should continue to display this error message *every time* the user saves this document.

A user can modify the contents of any duplicate publisher, but the contents of the edition will be whichever publisher was the last to write.

When a user chooses to copy and paste or duplicate a section, use the `HandToHand` function (described in *Inside Macintosh: Memory*) to duplicate the section record and alias record. Set the `alias` field of the cloned section record to the handle of the cloned alias record and generate a unique section identification number for it. In addition, you should also place the section data, section record, and alias record in the scrap.

Use the `RegisterSection` function (described in "Opening and Closing a Document Containing Sections" on page 2-22) to register the cloned section's section record.

A user can select the *contents* of a publisher without selecting the border and copy just the data to a new location. In this case, the user has simply copied data (and not the publisher). Do not create a border for this data in the new location.

Modifying a Subscriber

When the user selects data or clicks the data area of a subscriber, you should highlight the entire contents of the subscriber using inverse video. Although you shouldn't allow a user to edit the information in a subscriber, you can allow a user to make global adornments to subscribers. In other words, users can change the font, size, or other characteristics of the *entire* subscriber. For example, a user might select a subscriber within a document and change all text from plain to bold. However, you should discourage users from modifying the individual elements contained within a subscriber—for example, by editing a sentence or rotating an individual graphic object.

Remember that each time a new edition arrives for a subscriber, any modifications that the user has introduced are overwritten. Global changes to a subscriber are much easier for your application to regenerate.

Note
Although adornments should be global and never partial, you may still need to give users the ability to select portions of a subscriber, for instance, when performing spell checking and search-and-replace operations. ◆

If you do allow a user to edit a subscriber section, provide an enable/disable editing option within the subscriber options dialog box using the `SectionOptionsExpDialog` function, described in "Customizing Dialog Boxes" beginning on the next page. When you allow a user to edit a subscriber, you should change the subscriber from a selected state to editable data.

Because a user can modify a publisher just like any other portion of a document, its subscriber may change in size as well as content. For example, a user may modify a publisher by adding two additional columns to a spreadsheet.

Relocating an Edition

In the Finder, users cannot move an edition across volumes. To relocate an edition, the user must first select its publisher and cancel the section (remember to remove the border). The user needs to republish and then select a new volume location for the edition. As a convenience for the user, you should retain the selection of all the publisher data after the user cancels the section to make it easy to republish the section.

Customizing Dialog Boxes

The expandable dialog box functions allow you to add items to the bottom of the dialog boxes, apply alternate mapping of events to item hits, apply alternate meanings to the item hits, and choose the location of the dialog boxes. See the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* and the chapter "Standard File Package" in *Inside Macintosh: Files* for additional information.

The expandable versions of these dialog boxes require five additional parameters. Use the `NewPublisherExpDialog` function to expand the publisher dialog box.

```
err := NewPublisherExpDialog (reply, where, expansionDITLresID,
                              dlgHook, filterProc, yourDataPtr);
```

Use the `NewSubscriberExpDialog` function to expand the subscriber dialog box.

```
err := NewSubscriberExpDialog (reply, where, expansionDITLresID,
                               dlgHook, filterProc, yourDataPtr);
```

Use the `SectionOptionsExpDialog` function to expand the publisher options and the subscriber options dialog boxes.

```
err := SectionOptionsExpDialog (reply, where, expansionDITLresID,
                                dlgHook, filterProc, yourDataPtr);
```

The `reply` parameter is a pointer to a `NewPublisherReply`, `NewSubscriberReply`, or `SectionOptionsReply` record, respectively.

You can automatically center the dialog box by passing (–1, –1) in the `where` parameter.

The `expansionDITLresID` parameter should contain 0 or a valid item list (`'DITL'`) resource ID. This integer is the resource ID of an item list whose items are appended to the end of the standard item list. The dialog items keep their relative positions, but they are moved as a group to the bottom of the dialog box. See the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for additional information on item lists.

The `filterProc` parameter should be a pointer to an expandable modal-dialog filter function or `NIL`. An expandable modal-dialog filter function is similar to a modal-dialog filter function or event filter function except that an expandable modal-dialog filter function accepts two extra parameters. The `ModalDialog` procedure calls the expandable modal-dialog filter function you provide in this parameter.

Providing a filter function enables you to map real events (such as a mouse-down event) to an item hit (such as clicking the Cancel button). For instance, you may want to map a keyboard equivalent to an item hit. See the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for information on the `ModalDialog` procedure.

The `dlgHook` parameter should be a pointer to an expandable dialog hook function or `NIL`. An expandable dialog hook function is similar to a dialog hook function except that an expandable dialog hook function accepts an additional parameter. The `NewSubscriberExpDialog`, `NewPublisherExpDialog`, and `SectionOptionsExpDialog` functions call your expandable dialog hook function after each call to the `ModalDialog` procedure. The dialog hook function should take the appropriate action, such as filling in a checkbox. The `itemOffset` parameter to the procedure is the number of items in the item list before the expansion dialog items. You need to subtract the item offset from the item hit to get the relative item number in the expansion dialog item list. The expandable dialog hook function should return as its function result the absolute item number.

When the Edition Manager displays subsidiary dialog boxes in front of another dialog box on the user's screen, your dialog hook and event filter functions should check the `refCon` field in the `WindowRecord` data type (from the `window` field in the `DialogRecord`) to determine which window is currently in the foreground. The main dialog box for the `NewPublisherExpDialog` and the `NewSubscriberExpDialog` functions contains the following constant:

```
CONST    sfMainDialogRefCon   = 'stdf';   {new publisher and }
                                          { new subscriber}
```

The main dialog box for the `SectionOptionsExpDialog` function contains the following constant:

```
CONST    emOptionsDialogRefCon   = 'optn';   {options dialog}
```

See "Summary of the Edition Manager" beginning on page 2-106 for additional constants.

The `yourDataPtr` parameter is reserved for your use. It is passed back to your dialog hook and event filter function. This parameter does not have to be of type `Ptr`—it can be any 32-bit quantity that you want. In Pascal, you can pass `yourDataPtr` in register A6 , and declare your dialog hook and modal-dialog filter as local functions without the last parameter. The stack frame is set up properly for these functions to access their parent local variables. See the chapter "Standard File Package" in *Inside Macintosh: Files* for detailed information.

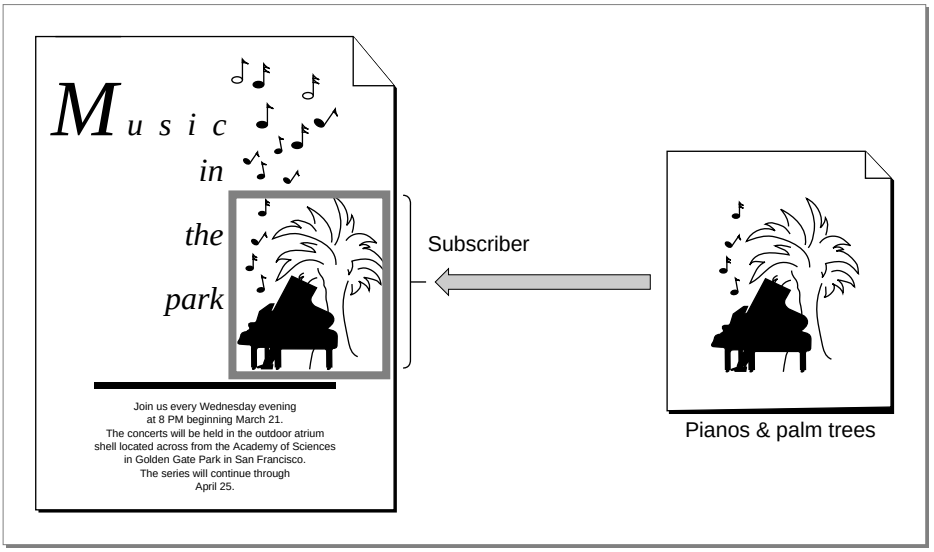
For the `NewPublisherExpDialog` and `NewSubscriberExpDialog` functions, all the pseudo-items for the Standard File Package—such as `sfHookFirstCall`(–1), `sfHookNullEvent`(100), `sfHookRebuildList`(101), and `sfHookLastCall`(–2)—can be used, as well as `emHookRedrawPreview`(150).

For the `SectionOptionsExpDialog` function, the only valid pseudo-items are `sfHookFirstCall`(–1), `sfHookNullEvent`(100), `sfHookLastCall`(–2), `emHookRedrawPreview`(150), `emHookCancelSection`(160), `emHookGoToPublisher`(161), `emHookGetEditionNow`(162), `emHookSendEditionNow`(162), `emHookManualUpdateMode`(163), and `emHookAutoUpdateMode`(164). See the chapter "Standard File Package" in *Inside Macintosh: Files* for information on pseudo-items.

Subscribing to Non-Edition Files

Using the Edition Manager, a subscriber can read data directly from another document, such as an entire `'PICT'` file, instead of subscribing to an edition. This feature is for advanced applications that can set up bottleneck procedures for reading. Figure 2-26 shows a document that is subscribing directly to a `'PICT'` file.

Figure 2-26 Subscribing directly to a `'PICT'` file



For each application, the Edition Manager keeps a pointer to a bottleneck function. The Edition Manager never opens or closes an edition container directly. Instead, the Edition Manager calls the current edition opener. The `InitEditionPack` function (described on page 2-74) sets up the current system opener function.

To override the standard opener function, create an opener function that contains the following parameters:

```
FUNCTION MyOpener (selector: EditionOpenerVerb;
                   VAR PB: EditionOpenerParamBlock): OSErr;
```

Your opener needs to know which formats the file contains and how the data is supposed to be read or written.

The opener function is passed an edition opener verb in the `selector` parameter, which identifies the action the opener function should perform. The opener can allocate a handle or pointer to contain information such as file reference numbers. This value is passed to the I/O routines in the `ioRefNum` field of the edition opener parameter block.

The `eoOpen` and `eoOpenNew` edition opener verbs (described in "Calling an Edition Opener" on page 2-64) return a pointer to a function to do the actual reading and writing.

The following sections describe

- how to get the current edition opener
- how to set your own edition opener
- how to call an edition opener
- the edition opener parameters

Getting the Current Edition Opener

When you want to get the current edition opener, use the `GetEditionOpenerProc` function.

```
err := GetEditionOpenerProc(opener);
```

The `opener` parameter returns a pointer to the current edition opener. A different current opener is kept for each application. One application's opener is never called by another application.

Setting an Edition Opener

You can provide your own edition opener. To do so, use the `SetEditionOpenerProc` function.

```
err := SetEditionOpenerProc(@MyOpener);
```

The `@MyOpener` parameter is a pointer to the edition opener function that you are providing. If you set the current opener to be a routine in your own code, be sure to call the `GetEditionOpenerProc` function first so that you can save the previous opener. If your opener is passed a selector that it does not understand, use the previous opener provided by the Edition Manager to handle it. See the next section for a list of selectors.

Calling an Edition Opener

You use the `CallEditionOpenerProc` function to call an edition opener. Since the Edition Manager is a package that may move, a real pointer cannot be safely returned for the standard opener and I/O routines. The system opener and the I/O routines are returned as a value that is not a valid address to a procedure. The `CallEditionOpenerProc` and `CallFormatIOProc` functions check for these values and call the system openers.

You should never assume that a value for a system opener is a fixed constant.

```
err := CallEditionOpenerProc (selector, PB, routine);
```

Set the `selector` parameter to one of the edition opener verbs. The edition opener verbs include

- `eoCanSubscribe`
- `eoOpen`
- `eoClose`
- `eoOpenNew`
- `eoCloseNew`

The `PB` parameter of the `CallEditionOpenerProc` function is an edition opener parameter block.

```
TYPE EditionOpenerParamBlock =
   RECORD
      info:          EditionInfoRecord;   {edition container to }
                                          { be subscribed to}
      sectionH:      SectionHandle;       {publisher or }
                                          { subscriber }
                                          { requesting open}
      document:      FSSpecPtr;           {document passed}
      fdCreator:     OSType;              {Finder creator type}
      ioRefNum:      LongInt;             {reference number}
      ioProc:        FormatIOProcPtr;     {routine to read }
                                          { formats}
      success:       Boolean;             {reading or writing }
                                          { was successful}
      formatsMask:   SignedByte;          {formats required to }
                                          { subscribe}
   END;
```

The `routine` parameter of the `CallEditionOpenerProc` function is a pointer to an edition opener function.

The following list shows which fields of the edition opener parameter block are used by the edition opener verbs:

Opener verb		Field	Description	Called by
`eoCanSubscribe`	→	`info`	Edition container to subscribe to.	`NewSubscriberDialog` function for a subscriber
	→	`formatsMask`	Formats required to subscribe.	
	←	Return value	A `noErr` code indicates that an edition container can be subscribed to. A `noTypeErr` code indicates that an edition container cannot be subscribed to.	
`eoOpen`	→	`info`	Edition container to open for reading.	`OpenEdition` and `GetStandardFormats` functions for a subscriber
	→	`sectionH`	Subscriber section requesting open or `NIL`.	
	←	`ioRefNum`	Reference number for use by I/O routine. Not the same as `EditionRefNum`.	
	←	`ioProc`	I/O routine to call to read formats.	
	←	Return value	A `noErr` code or appropriate error code.	
`eoClose`	→	`info`	Edition container to be closed for reading.	`CloseEdition` and `GetStandardFormats` functions for a subscriber
	→	`sectionH`	Subscriber section requesting close or `NIL`.	
	→	`ioRefNum`	Value returned by `eoOpen`.	
	→	`ioProc`	Value returned by `eoOpen`.	
	→	`success`	Success value passed to the `CloseEdition` function.	
	←	Return value	A `noErr` code or appropriate error code.	

continued

Opener verb		Field	Description	Called by (continued)
eoOpenNew	→	info	Edition container to open for writing.	OpenNewEdition function for a publisher
	→	sectionH	Publisher section requesting open or NIL.	
	→	document	Document pointer passed into the OpenNewEdition function.	
	→	fdCreator	The fdCreator passed into the OpenNewEdition function.	
	←	ioRefNum	Reference number for use by I/O routine. Not the same as EditionRefNum.	
	←	ioProc	I/O routine to call to write formats.	
	←	Return value	A noErr code or appropriate error code.	
eoCloseNew	→	info	Edition container to be closed after writing.	CloseEdition function for a publisher
	→	sectionH	Publisher section requesting close or NIL.	
	→	ioRefNum	Value returned by eoOpenNew.	
	→	ioProc	Value returned by eoOpenNew.	
	→	success	Success value passed to the CloseEdition function.	
	←	Return value	A noErr code or appropriate error code.	

As Listing 2-9 demonstrates, you install your own edition opener function by first saving the current opener and then installing your own opener. The listing also shows an edition opener, the MyEditionOpener function. When it receives the eoCanSubscribe opener verb, the MyEditionOpener function calls another application-defined routine, MyCanSubscribe. The Edition Manager sends your edition opener this verb to help it build the list of files displayed by the NewSubscriber function. The MyCanSubscribe function returns noErr if it can subscribe to the file; otherwise, it calls the original edition opener to handle the request.

Listing 2-9 Using your own edition opener function

```
VAR
   gOriginalOpener: EditionOpenerProcPtr;{global variable}

PROCEDURE MyInstallMyOpener;
BEGIN
   FailOSErr(GetEditionOpenerProc(gOriginalOpener));
   FailOSErr(SetEditionOpenerProc(@MyEditionOpener));
END; {MyInstallMyOpener}

FUNCTION MyEditionOpener (selector: EditionOpenerVerb;
                          VAR PB: EditionOpenerParamBlock)
                           : OSErr;
BEGIN
   WITH PB DO
   BEGIN
      CASE selector OF
         eoCanSubscribe:
            MyEditionOpener := MyCanSubscribe(PB);
         eoOpen:
            MyEditionOpener := MyEditionOpen(PB);
         eoClose:
            MyEditionOpener := MyEditionClose(PB);
         OTHERWISE
            {call the original edition opener}
            MyEditionOpener
               := CallEditionOpenerProc(selector, PB,
                                        gOriginalOpener);
      END; {of CASE}
   END; {of WITH}
END; {MyEditionOpener}

FUNCTION MyCanSubscribe (VAR PB: EditionOpenerParamBlock): OSErr;
BEGIN
   {check file type to see if it is a file you can emulate as an }
   { edition}
   IF PB.info.fdType = {for example}'PICT' THEN
      MyCanSubscribe := noErr
   ELSE {otherwise, let the saved edition opener decide}
         MyCanSubscribe := CallEditionOpenerProc(eoCanSubscribe,
                                                 PB, gOriginalOpener);
END; {MyCanSubscribe}
```

Opening and Closing Editions

Each time the Edition Manager opens or closes an edition container, it calls the current edition opener procedure and passes it an opener verb and a parameter block.

Your opener must be careful when closing documents since a document may already have been opened by another application. Be sure to use the Open/Deny modes whenever possible. Do not close a document if it was already open when your application opened it.

Listing Files That Can Be Subscribed To

The `NewSubscriberDialog` function calls the edition opener function and passes the `eoCanSubscribe` opener verb in the `selector` parameter to build the list of files that can be subscribed to. The preview in the subscriber dialog box is generated by calling the `GetStandardFormats` function (described in "Edition Container Formats" on page 2-101), which calls the format I/O procedure with the verbs `eoOpen`, `ioHasFormat`, `ioRead`, and then `eoClose`. See "Calling a Format I/O Function" on this page for detailed information on format I/O verbs.

Reading From and Writing to Files

The I/O procedure is a routine that actually reads and writes the data. It too has an interface of a selector and a parameter block.

To override the standard reading and writing functions, create an I/O function. Note that you also need to provide your own opener function to call your I/O function. See "Calling an Edition Opener" on page 2-64.

```
FUNCTION MyIO (selector: FormatIOVerb;
               VAR PB: FormatIOParamBlock): OSErr;
```

Calling a Format I/O Function

To indicate to the Edition Manager which format I/O function to use, use the `CallFormatIOProc` function.

```
err := CallFormatIOProc (selector, PB, routine);
```

Set the `selector` parameter to one of the format I/O verbs. The format I/O verbs include

- `ioHasFormat`
- `ioReadFormat`
- `ioNewFormat`
- `ioWriteFormat`

The `PB` parameter of the `CallFormatIOProc` function contains a format I/O parameter block.

```
TYPE FormatIOParamBlock =
   RECORD
      ioRefNum:      LongInt;       {reference number}
      format:        FormatType;    {edition format type}
      formatIndex:   LongInt;       {opener-specific enumeration }
                                    { of formats}
      offset:        LongInt;       {offset into format}
      buffPtr:       Ptr;           {data starts here}
      buffLen:       LongInt;       {length of data}
   END;
```

The `routine` parameter of the `CallFormatIOProc` function is a pointer to a format I/O function.

The following list shows which fields of `FormatIOParamBlock` are used by the format I/O verbs:

Format I/O verb		Parameter	Description	Called by
`ioHasFormat`	→	`ioRefNum`	I/O reference number returned by opener.	`EditionHasFormat`, `GetStandardFormats`, and `ReadEdition` functions
	→	`format`	Check for this format.	
	←	`formatIndex`	An optional enumeration of the supplied format.	
	←	`buffLen`	If found, return the length size or –1 if size is unknown.	
	←	Return value	A `noErr` or `noTypeErr` code.	

continued

Format I/O verb		Parameter	Description	Called by (continued)
ioReadFormat	→	ioRefNum	I/O reference number returned by opener.	ReadEdition and GetStandardFormats functions
	→	format	Get this format.	
	→	formatIndex	Value returned by ioHasFormat.	
	→	offset	Read format beginning from this offset.	
	→	buffPtr	Put data beginning here.	
	↔	buffLen	Specify buffer length to read, and return actual amount received.	
	←	Return value	A noErr code, or appropriate error code.	
ioNewFormat	→	ioRefNum	I/O reference number returned by opener.	SetEditionFormatMark and WriteEdition functions
	→	format	Create this format.	
	←	formatIndex	An optional enumeration of the supplied format.	
	←	Return value	A noErr code, or appropriate error code.	
ioWriteFormat	→	ioRefNum	I/O reference number returned by opener.	WriteEdition function
	→	format	Get this format.	
	→	formatIndex	Value returned by ioNewFormat.	
	→	offset	Write format beginning from this offset.	
	→	buffPtr	Get data beginning here.	
	↔	buffLen	Specify buffer length to write.	
	←	Return value	A noErr code or appropriate error code.	

The marks for each format are kept by the Edition Manager. The format I/O function only needs to be able to read or write, beginning at any offset. If you know that your application always reads an entire format sequentially, you can ignore the offset.

Edition Manager Reference

This section describes the data structures and routines that are specific to the Edition Manager. The "Data Structures" section describes the edition container record and the section record. The "Edition Manager Routines" section describes the routines your application can use to implement publish and subscribe features in your application.

Data Structures

This section describes the edition container record and the section record. See page 2-91 for a description of the new subscriber reply record, page 2-93 for a description of the new publisher reply record, page 2-95 for a description of the section options record, and page 2-99 for a description of the edition info record. For information on the edition opener parameter block and format I/O parameter block, see page 2-103 and page 2-104, respectively.

The Edition Container Record

An edition container record identifies a specific edition file. Many Edition Manager routines require an edition container record as a parameter. The `EditionContainerSpec` data type defines an edition container record.

```
TYPE EditionContainerSpec =
   RECORD
      theFile:          FSSpec;        {file containing edition }
                                       { data}
      theFileScript:    ScriptCode;    {script code of filename}
      thePart:          LongInt;       {which part of file, }
                                       { always kPartsNotUsed}
      thePartName:      Str31;         {reserved}
      thePartScript:    ScriptCode;    {reserved}
   END;
```

Field descriptions

`theFile` A file specificiation record that identifies the name and location of the edition file. Specify the file using the standard conventions for file specification records as described in the chapter "Introduction to File Management" in *Inside Macintosh: Files*.

`theFileScript` A script code that identifies the script in which the name of the document is to be displayed in the Finder. A script code of `smSystemScript` represents the default system script.

`thePart`	A value that must always be set to `kPartsNotUsed` in System 7.
`thePartName`	Reserved.
`thePartScript`	Reserved.

The Section Record

A section record identifies a specific publisher or subscriber section. It contains information to identify the section as a publisher or a subscriber, a time stamp to record the last modification of the section, and unique identification for each section. Many Edition Manager routines require a handle to a section record as a parameter. The `SectionRecord` data type defines a section record.

```
TYPE SectionRecord =
   RECORD
      version:       SignedByte;    {always 1 in 7.0}
      kind:          SectionType;   {publisher or subscriber}
      mode:          UpdateMode;    {automatic or manual}
      mdDate:        TimeStamp;     {last change in document}
      sectionID:     LongInt;       {application-specific, }
                                    { unique per document}
      refCon:        LongInt;       {application-specific}
      alias:         AliasHandle;   {handle to alias record}

      {The following fields are private and are set up by the }
      { RegisterSection function. Do not modify the private }
      { fields.}
      subPart:       LongInt;       {private}
      nextSection:   SectionHandle; {private, do not use as a }
                                    { linked list}
      controlBlock:  Handle;        {may be used for comparison }
                                    { only}
      refNum:        EditionRefNum; {private}
END;
```

Field descriptions

`version`	Indicates the version of the section record, currently $01.
`kind`	Defines the section type as either publisher or subscriber with the `stPublisher` or `stSubscriber` constant.
`mode`	Indicates if editions are updated automatically or manually.

`mdDate`	Indicates which version (modification date) of the section's contents is contained within the publisher or subscriber. The `mdDate` field is set to 0 when you create a new subscriber section and to the current time when you create a new publisher. Be sure to update this field each time publisher data is modified. The section's modification date is compared to the edition's modification date to determine whether the section and the edition contain the same data. The section modification date is displayed in the publisher and subscriber options dialog boxes. See "Closing an Edition" on page 2-28 for detailed information.
`sectionID`	Provides a unique number for each section within a document. A simple way to implement this is to create a counter for each document that is saved to disk with the document. The counter should start at 1. The section ID is currently used as a tie breaker in the `GoToPublisherSection` function when there are multiple publishers to the same edition in a single document. The section ID should not be 0 or –1. See "Duplicating Publishers and Subscribers" on page 2-58 for information on multiple publishers.
`refCon`	Reference constant available for application-specific use.
`alias`	Contains a handle to the alias record for a particular section within a document.

Whenever the user creates a publisher or subscriber, call the `NewSection` function (described on page 2-75) to create a section record and alias record.

Edition Manager Routines

This section describes the routines you use to

- initialize the Edition Manager
- create and register a section
- create and delete an edition container
- set and locate a format mark
- read in edition data
- write out edition data
- close an edition after reading or writing
- display dialog boxes
- locate a publisher and edition from a subscriber
- read and write non-edition files

Result codes appear at the end of each function where applicable. In addition to the specific result codes listed, you may receive errors generated by the Alias Manager, File Manager, and Memory Manager.

Initializing the Edition Manager

You use the `InitEditionPack` function to initialize the Edition Manager. Note that you should call this function only once.

InitEditionPack

Before calling the `InitEditionPack` function, be sure to determine whether the Edition Manager is available on your system by using the `Gestalt` function with the `gestaltEditionMgrAttr` (`'edtn'`) selector.

```
FUNCTION InitEditionPack: OSErr;
```

DESCRIPTION

The `InitEditionPack` function returns an error if the package could not be loaded into the system heap and properly initialized.

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Could not load package

Creating and Registering a Section

You use the `NewSection` function to create a new section (either publisher or subscriber) and alias record (which is a reference to the edition container from the document containing the publisher or subscriber section).

The `NewSection` function registers a section much as the `RegisterSection` function informs the Edition Manager about a section (except that the `NewSection` function does not resolve an alias to find the edition container).

When a section needs to be disposed of because the document containing the section is being closed or because the user has canceled the section, you need to call the `UnRegisterSection` function before disposing of the section.

Using the `IsRegisteredSection` function, your application must verify that each event received is for a registered section. This is necessary because your application may have just called `UnRegisterSection` while the event was already being held in the event queue.

If a user saves a document that contains sections under another name (using Save As) or pastes a portion of a document that contains a section into another document, use the `AssociateSection` function to update the section's alias record.

NewSection

Use the `NewSection` function to create a new section record and alias record for a new publisher or subscriber.

```
FUNCTION NewSection (container: EditionContainerSpec;
                     sectionDocument: FSSpecPtr;
                     kind: SectionType; sectionID: LongInt;
                     initialMode: UpdateMode;
                     VAR sectionH: SectionHandle): OSErr;
```

`container` The edition you want to publish or subscribe to.

`sectionDocument`
The volume reference number, directory ID, and filename of the document that contains a section. The `sectionDocument` parameter can be `NIL` if your current document has never been saved. If so, when the user finally saves the document, remember to call the `AssociateSection` function for each section to update its alias record.

`kind` The type of section (publisher or subscriber) being created.

`sectionID` A unique number for a section within a document. The `NewSection` function initializes the `sectionID` field of the new section record with the specified value. Do not use 0 or –1 for an ID number; these numbers are reserved. If your application copies a section, you need to specify a unique number for the copied section.

`initialMode`
The update mode for the section. For publishers this is either the `pumOnSave` or `pumManual` constant, and for subscribers it is either `sumAutomatic` or `sumManual`. A subscriber created with `sumAutomatic` mode automatically receives a Section Read event. To prevent this initial Section Read event, you should set the `initialMode` parameter to `sumManual` and then, when `NewSection` returns, set the `mode` field of the section record to `sumAutomatic`.

`sectionH` The `NewSection` function returns a handle to the allocated section record in this parameter. If an error occurs, `NewSection` returns `NIL` in this parameter.

DESCRIPTION

The `NewSection` function allocates two handles in the current zone: one handle for the section record and another handle for the alias record. Note that you are responsible for unregistering handles created by the Edition Manager.

Your application receives the `multiplePublisherWrn` result code if there is another registered publisher to the same edition. Your application receives the `notThePublisherWrn` result code if another publisher (to the same edition) was the last section to write to the edition. The `multiplePublisherWrn` result code takes priority over the `notThePublisherWrn` result code.

RESULT CODES

noErr	0	No error
editionMgrInitErr	–450	Manager not initialized
badSectionErr	–451	Not a valid section type
badSubPartErr	–454	Bad edition container spec
multiplePublisherWrn	–460	Already is a publisher
notThePublisherWrn	–463	Not the publisher

SEE ALSO

For information on the edition container record, see page 2-71. For information on the section record, see "The Section Record" beginning on page 2-72. For information on file specification records, see *Inside Macintosh: Files*. See Listing 2-4 on page 2-33 for an example that uses `NewSection` to create a publisher and Listing 2-6 on page 2-40 for an example that creates a subscriber using `NewSection`.

RegisterSection

When opening a document that contains sections, register each section using the `RegisterSection` function.

```
FUNCTION RegisterSection (sectionDocument: FSSpec;
                          sectionH: SectionHandle;
                          VAR aliasWasUpdated: Boolean): OSErr;
```

`sectionDocument`
The volume reference number, directory ID, and filename of the document that contains a section.

`sectionH` A handle to the section record for a given section.

`aliasWasUpdated`
A Boolean value that returns `TRUE` if the alias for the edition container subscribed to was out of date and was updated. This may occur if the edition file was moved to a new location or was renamed.

DESCRIPTION

The `RegisterSection` function adds the section record to the Edition Manager's list of registered sections and tries to allocate a control block. After calling the `RegisterSection` function, the `controlBlock` field of the section record contains either `NIL` or a valid control block.

For a subscriber, the `controlBlock` field contains `NIL` if the `RegisterSection` function could not locate the edition container being subscribed to. The `RegisterSection` function then returns either the `containerNotFoundWrn` or the `userCanceledErr` result code. For a publisher, if the `RegisterSection` function could not locate its corresponding edition container, the Edition Manager creates an

edition container in the last place the edition was located and creates a control block for it. If the `RegisterSection` function could not locate a publisher's corresponding edition container or its volume, the `controlBlock` field contains `NIL`. You should never re-register a section that is already registered.

Note that you can compare control blocks for individual sections. If two sections contain the same control block value, these sections publish or subscribe to the same edition (unless the control block is `NIL`). The Edition Manager keeps track of how many sections are referencing a control block to know when it can be deallocated. The control block maintains a count of how many sections are referencing it. Each time you use the `UnRegisterSection` function, the control block subtracts 1 from the number of sections. When the number of sections reaches 0, the control block is deallocated.

Your application receives the `multiplePublisherWrn` result code if there is another registered publisher to the same edition. Your application receives the `notThePublisherWrn` result code if another publisher (to the same edition) was the last section to write to the edition. The `multiplePublisherWrn` result code takes priority over the `notThePublisherWrn` result code.

RESULT CODES

`noErr`	0	No error
`userCanceledErr`	–128	User clicked Cancel in dialog box
`editionMgrInitErr`	–450	Manager not initialized
`badSectionErr`	–451	Not valid section type
`multiplePublisherWrn`	–460	Already is a publisher
`containerNotFoundWrn`	–461	Alias was not resolved
`notThePublisherWrn`	–463	Not the publisher

SEE ALSO

For information on the section record, see "The Section Record" beginning on page 2-72. For information on file specification records, see *Inside Macintosh: Files*. For additional information and an example of the use of `RegisterSection`, see "Opening and Closing a Document Containing Sections" beginning on page 2-22.

UnRegisterSection

When a section needs to be disposed of because the document containing the section is being closed or because the user has canceled the section, you need to call the `UnRegisterSection` function before disposing of the section.

```
FUNCTION UnRegisterSection (sectionH: SectionHandle): OSErr;
```

`sectionH` A handle to the section record for a given section.

DESCRIPTION

The `UnRegisterSection` function removes the section from the Edition Manager's list of registered sections. You can then dispose of the section record and alias record with standard Memory Manager and Resource Manager calls. Once unregistered, a section does not receive any events and cannot read or write any data. Depending on your Clipboard strategy, you may want to unregister sections that have been cut into the Clipboard.

RESULT CODES

`noErr`	0	No error
`fBsyErr`	–47	Section doing I/O
`editionMgrInitErr`	–450	Manager not initialized
`notRegisteredSectionErr`	–452	Not registered

IsRegisteredSection

Upon receiving a section event, your application must call the `IsRegisteredSection` function to verify that the event received is for a registered section. You must call `IsRegisteredSection` before handling a section event because your application may have just called `UnRegisterSection` while the event was already being held in the event queue.

```
FUNCTION IsRegisteredSection (sectionH: SectionHandle): OSErr;
```

`sectionH` A handle to the section record for a given section.

DESCRIPTION

The `IsRegisteredSection` function returns a result code (not a Boolean value) indicating whether the section is registered. A `noErr` result code indicates that a section is registered.

RESULT CODES

`noErr`	0	No error
`notRegisteredSectionErr`	–452	Not registered

SEE ALSO

For an example of the use of `IsRegisteredSection`, see Listing 2-1 on page 2-14.

AssociateSection

If a user saves a document that contains sections under another name (using Save As) or pastes a portion of a document that contains a section into another document, use the `AssociateSection` function to update the section's alias record.

```
FUNCTION AssociateSection (sectionH: SectionHandle;
                           newSectionDocument: FSSpecPtr): OSErr;
```

`sectionH` A handle to the section record for a given section.

`newSectionDocument`
The volume reference number, directory ID, and filename of the new document.

DESCRIPTION

The `AssociateSection` function calls `UpdateAlias` on the section's alias record.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter

SEE ALSO

For information on the `UpdateAlias` function, see the chapter "Alias Manager" in *Inside Macintosh: Files*.

Creating and Deleting an Edition Container

Each time a user creates a new publisher section within a document to an edition that does not already exist, you use the `CreateEditionContainerFile` function to create an empty edition container.

To remove the edition container, use the `DeleteEditionContainerFile` function.

CreateEditionContainerFile

You use the `CreateEditionContainerFile` function to create an empty edition container.

```
FUNCTION CreateEditionContainerFile
                      (editionFile: FSSpec; fdCreator: OSType;
                       editionFileNameScript: ScriptCode): OSErr;
```

`editionFile`
The volume reference number, directory ID, and filename for the edition container being created.

`fdCreator` The creator type for the edition.

`editionFileNameScript`
The script of the filename. (You can get this value from the `theFileScript` field of an edition container specification record.)

DESCRIPTION

The `CreateEditionContainerFile` function creates an empty edition container file (it does not contain any formats). This function sets the file type of the edition to `'edtu'`. As soon as you write data to the edition, the Edition Manager updates the type (to `'edtp'` for graphics, `'edtt'` for text, or `'edts'` for sound). If your application writes both `'TEXT'` and `'PICT'` formats to the edition, the Edition Manager sets the file type to the type that was written first. If your application has a bundle, you should designate an icon for the appropriate edition types that you can write.

RESULT CODES

`noErr`	0	No error
`dskFulErr`	–34	Disk is full
`nsvErr`	–35	No such volume
`ioErr`	–36	I/O error
`bdNamErr`	–37	Bad filename
`fnfErr`	–43	File not found
`dirNFErr`	–120	Directory not found
`editionMgrInitErr`	–450	Manager not initialized

SEE ALSO

For information on file specification records, see *Inside Macintosh: Files*. For an example of the use of `CreateEditionContainerFile`, see Listing 2-4 on page 2-33.

DeleteEditionContainerFile

If a user cancels a publisher section within a document or closes a document containing a newly created publisher without saving, you need to remove the edition container.

To locate the appropriate edition container to be deleted, use the `GetEditionInfo` function. You use the `UnRegisterSection` function (only after using the `GetEditionInfo` function) to unregister the section record and alias record of the publisher being canceled.

To remove the edition container, use the `DeleteEditionContainerFile` function.

```
FUNCTION DeleteEditionContainerFile (editionFile: FSSpec): OSErr;
```

`editionFile`
The volume reference number, directory ID, and filename for the edition container being deleted.

DESCRIPTION

If the user cancels a publisher, do not call the `DeleteEditionContainerFile` function until the user saves the document. This allows the user to undo changes and revert to the last saved version of the document.

The `DeleteEditionContainerFile` function deletes the edition container only if there is no registered publisher. You need to unregister a publisher before you can delete its corresponding edition container.

You should use the `DeleteEditionContainerFile` function even if there are subscribers to the edition. When a subscriber section tries to read in data, it receives an error if the edition container has been deleted.

RESULT CODES

`noErr`	0	No error
`nsvErr`	–35	No such volume
`ioErr`	–36	I/O error
`bdNamErr`	–37	Bad filename
`fnfErr`	–43	File not found
`dirNFErr`	–120	Directory not found
`editionMgrInitErr`	–450	Manager not initialized

SEE ALSO

See page 2-98 for detailed information on the `GetEditionInfo` function. See page 2-77 for information on the `UnRegisterSection` function. For information on file specification records, see *Inside Macintosh: Files*.

Setting and Getting a Format Mark

Use the `SetEditionFormatMark` function to set the current mark for a section format and the `GetEditionFormatMark` function to get the current mark for a particular format.

SetEditionFormatMark

A format mark indicates the next position of a read or write operation. Initially, a mark defaults to 0. After reading or writing data, the format mark is set past the last position written to or read from. To set the current mark for a given format, use the `SetEditionFormatMark` function.

```
FUNCTION SetEditionFormatMark (whichEdition: EditionRefNum;
                               whichFormat: FormatType;
                               setMarkTo: LongInt): OSErr;
```

`whichEdition`
The reference number for the edition.

`whichFormat`
The format type for the edition.

`setMarkTo` The offset for the next read or write for this format.

DESCRIPTION

The `SetEditionFormatMark` function sets the current mark for the specified format type according to the value of the `setMarkTo` parameter.

RESULT CODES

`noErr`	0	No error
`rfNumErr`	–51	Bad edition reference number
`noTypeErr`	–102	Unknown format (subscriber only)
`editionMgrInitErr`	–450	Manager not initialized

GetEditionFormatMark

Use the `GetEditionFormatMark` function to get the current mark for a particular format.

```
FUNCTION GetEditionFormatMark (whichEdition: EditionRefNum;
                               whichFormat: FormatType;
                               VAR currentMark: LongInt): OSErr;
```

`whichEdition`
The reference number for the edition.

`whichFormat`
The format type whose mark you want to get.

`currentMark`
The `GetEditionFormatMark` function returns the mark for the specified format in this parameter.

DESCRIPTION

If the edition does not support the format specified in the `whichFormat` parameter, you receive a `noTypeErr` result code.

RESULT CODES

`noErr`	0	No error
`rfNumErr`	–51	Bad edition reference number
`noTypeErr`	–102	Unknown format
`editionMgrInitErr`	–450	Manager not initialized

Reading in Edition Data

To initiate the reading of data from an edition (for a subscriber), use the `OpenEdition` function.

Use the `EditionHasFormat` function to learn in which formats the edition data is available.

Use the `ReadEdition` function to read data from an edition. This function reads from the current mark for the specified format.

OpenEdition

To initiate the reading of data from an edition (for a subscriber), use the `OpenEdition` function.

```
FUNCTION OpenEdition (subscriberSectionH: SectionHandle;
                      VAR refNum: EditionRefNum): OSErr;
```

`subscriberSectionH`
A handle to the section record for a given section.

`refNum` The `OpenEdition` function returns the reference number for the edition in this parameter.

DESCRIPTION

The `OpenEdition` function opens an edition for reading and returns a reference number that your application can use to refer to this edition in other Edition Manager routines. Multiple subscribers can each call the `OpenEdition` function simultaneously (each call returns a different reference number) and read data from a single edition. If a publisher (located on a different machine) is writing to an edition when you use the `OpenEdition` function, you receive an `flLckedErr` result code.

RESULT CODES

`noErr`	0	No error
`fnfErr`	–43	File not found
`flLckedErr`	–45	Publisher writing to an edition
`permErr`	–54	Not a subscriber
`editionMgrInitErr`	–450	Manager not initialized

SEE ALSO

For an example of the use of `OpenEdition`, see Listing 2-7 on page 2-42.

EditionHasFormat

Use the `EditionHasFormat` function to learn in which formats the edition data is available.

```
FUNCTION EditionHasFormat (whichEdition: EditionRefNum;
                           whichFormat: FormatType;
                           VAR formatSize: Size): OSErr;
```

`whichEdition`
The reference number for the edition.

`whichFormat`
The format type that you are requesting. For the `whichFormat` parameter, you should decide which formats to read in the same way that you do when reading data from the scrap. You can also get a list of all the available formats and their respective lengths by reading the `kFormatListFormat('fmts')` format.

`formatSize`
The `EditionHasFormat` function returns the format length in this parameter.

DESCRIPTION

If the requested format is available, the `EditionHasFormat` function returns `noErr`, and the `formatSize` parameter returns the size of the data in the specified format or `kFormatLengthUnknown` (–1), which signifies that the size is unknown. You should therefore continue to read the format until there is no more data.

Note
The Translation Manager (if it is available) attempts implicit translation under certain circumstances. For instance, it does so when your application attempts to read from an edition a format type that is not in the edition. In this case, the Translation Manager attempts to translate the data into the requested format. For more information, see the chapter "Translation Manager" in *Inside Macintosh: More Macintosh Toolbox*. ◆

RESULT CODES

`noErr`	0	No error
`rfNumErr`	–51	Bad edition reference number
`noTypeErr`	–102	Format not available
`editionMgrInitErr`	–450	Manager not initialized

SEE ALSO

For an example of the use of `EditionHasFormat`, see Listing 2-7 beginning on page 2-42. For information about the Translation Manager and Scrap Manager, see *Inside Macintosh: More Macintosh Toolbox*.

ReadEdition

Use the `ReadEdition` function to read data from an edition. This function reads from the current mark for the specified format.

```
FUNCTION ReadEdition (whichEdition: EditionRefNum;
                      whichFormat: FormatType; buffPtr: UNIV Ptr;
                      VAR buffLen: Size): OSErr;
```

`whichEdition`
The reference number for the edition.

`whichFormat`
The format type that you want to read.

`buffPtr` A pointer to the buffer into which you want to read the data.

`buffLen` The number of bytes that you want to read into the buffer. The `ReadEdition` function returns the actual number of bytes read in the `buffLen` parameter.

DESCRIPTION

The `ReadEdition` function reads data from the edition into the specified buffer. `ReadEdition` returns in the `buffLen` parameter the total number of bytes read into the buffer. If the `buffLen` parameter returns a value smaller than the value you have specified, there is no additional data to read, and the `ReadEdition` function returns a `noErr` result code. If you use the `ReadEdition` function after all data is read in, the `ReadEdition` function returns an `eofErr` result code.

You can read data from an edition while a publisher on the same machine is writing data to the same edition. The data that you are reading is the old edition (not the data that the publisher is writing). If the publisher finishes writing data before you are through reading the old edition data, the `ReadEdition` function returns an `abortErr` result code. If the `ReadEdition` function returns an `abortErr` result code, you should stop trying to read data and use the `CloseEdition` function with the `successful` parameter set to `FALSE`.

Note

The Translation Manager (if it is available) attempts implicit translation under certain circumstances. For instance, it does so when your application attempts to read from an edition a format type that is not in the edition. In this case, the Translation Manager attempts to translate the data into the requested format. For more information, see the chapter "Translation Manager" in *Inside Macintosh: More Macintosh Toolbox*. ◆

RESULT CODES

`noErr`	0	No error
`abortErr`	–27	Publisher has written a new edition
`ioErr`	–36	I/O error
`fnOpnErr`	–38	File not open
`eofErr`	–39	No more data of that format
`rfNumErr`	–51	Bad edition reference number
`noTypeErr`	–102	Format not available
`editionMgrInitErr`	–450	Manager not initialized

SEE ALSO

For an example of the use of `ReadEdition`, see Listing 2-7 beginning on page 2-42.

Writing out Edition Data

To initiate the writing of data from a publisher to its edition container, use the `OpenNewEdition` function. (To create an edition container, use the `CreateEditionContainerFile` function, as described on page 2-79.)

Use the `WriteEdition` function to write data to an edition.

OpenNewEdition

To initiate the writing of data from a publisher to its edition container, use the `OpenNewEdition` function.

```
FUNCTION OpenNewEdition (publisherSectionH: SectionHandle;
                         fdCreator: OSType;
                         publisherSectionDocument: FSSpecPtr;
                         VAR refNum: EditionRefNum): OSErr;
```

`publisherSectionH`
The publisher section that is writing to the edition.

`fdCreator` The Finder creator type of the new edition icon.

`publisherSectionDocument`
The document that contains the publisher. This parameter is used to create an alias from the edition to the publisher's document. If you pass `NIL` for `publisherSectionDocument`, an alias is not made in the edition file.

`refNum` The `OpenNewEdition` function returns the reference number for the edition in this parameter. You specify this reference number as a parameter for subsequent calls to `WriteEdition`, `SetEditionFormatMark`, and `CloseEdition` to specify which publisher is writing its data to an edition. If the edition cannot be opened for writing because there is another publisher writing to it, or because the file system does not allow writing, an error is returned and `OpenNewEdition` sets `refNum` to `NIL`.

DESCRIPTION

The `OpenNewEdition` function opens an edition for writing. The function returns an `flLckdErr` result code if there is a subscriber on another machine reading data from the same edition. The `OpenNewEdition` function returns a `permErr` result code if there is a registered publisher to that edition on another machine.

The Edition Manager allows two registered publishers that are located on the *same* machine to write to the same edition. Note that multiple publishers cannot write to the same edition simultaneously—only one publisher can write to an edition at a given time.

RESULT CODES

`noErr`	0	No error
`ioErr`	–36	I/O error
`flLckdErr`	–45	Edition in use by another section
`permErr`	–54	Registered publisher on another machine
`wrPermErr`	–61	Not a publisher
`editionMgrInitErr`	–450	Manager not initialized

SEE ALSO

For an example of the use of `OpenNewEdition`, see Listing 2-5 beginning on page 2-36.

WriteEdition

Use the `WriteEdition` function to write data to an edition. This function begins writing at the current mark for the specified format.

```
FUNCTION WriteEdition (whichEdition: EditionRefNum;
                       whichFormat: FormatType;
                       buffPtr: UNIV Ptr; buffLen: Size): OSErr;
```

`whichEdition`
The reference number for the edition.

`whichFormat`
The format type that you want to write.

`buffPtr` A pointer to the buffer containing the data to write to the edition.

`buffLen` The number of bytes that you want to write to the edition.

DESCRIPTION

The `WriteEdition` function writes the specified number of bytes to the edition. If the data cannot be entirely written to the edition, the `WriteEdition` function returns an error.

RESULT CODES

`noErr`	0	No error
`dskFulErr`	–34	Disk is full
`ioErr`	–36	I/O error
`rfNumErr`	–51	Bad edition reference number
`editionMgrInitErr`	–450	Manager not initialized

SEE ALSO

For an example that writes data to an edition, see Listing 2-5 beginning on page 2-36.

Closing an Edition After Reading or Writing

After finishing reading from or writing to an edition, use the `CloseEdition` function to close the edition.

CloseEdition

Use the `CloseEdition` function to close an edition after you finish reading from or writing to it.

```
FUNCTION CloseEdition (whichEdition: EditionRefNum;
                       successful: Boolean): OSErr;
```

`whichEdition`
The reference number for the edition.

`successful`
A value that indicates whether your application was successful (`TRUE`) or unsuccessful (`FALSE`) in reading from or writing data to the edition.

DESCRIPTION

When a subscriber successfully finishes reading data from the edition, the `CloseEdition` function takes the modification date of the edition file that you have read and puts it in the `mdDate` field of the subscriber's section record. This indicates that the data contained in the edition and the subscriber section within the document are the same.

When a subscriber is unsuccessful in reading data from an edition (because there is not enough memory, or you didn't find a format that you can read), set the `successful` parameter to `FALSE`. The `CloseEdition` function then closes the edition, but does not set the `mdDate` field. This implies that the subscriber is not updated with the latest edition.

When a publisher successfully finishes writing data to an edition, the `CloseEdition` function makes the data that the publisher has written to the edition available to any subscribers and sets the corresponding edition file's modification date (`ioFlMdDat`) to the `mdDate` field of the publisher's section record. The Edition Manager then sends a Section Read event to all current subscribers set to automatic update mode. At this point, the file type of the edition file is set based on the first known format that the publisher wrote.

When a publisher is unsuccessful in writing data to an edition, the `CloseEdition` function discards what the publisher has written to the edition. The data contained in the edition prior to writing remains unchanged, and Section Read events are not sent to subscribers.

RESULT CODES

`noErr`	0	No error
`ioErr`	–36	I/O error
`fnOpnErr`	–38	File not open
`rfNumErr`	–51	Bad edition reference number
`editionMgrInitErr`	–450	Manager not initialized

SEE ALSO

For an example of the use of `CloseEdition`, see Listing 2-5 beginning on page 2-36.

Displaying Dialog Boxes

The Edition Manager supports three dialog boxes: publisher, subscriber, and options dialog boxes. Your application can display simple dialog boxes that appear centered on the user's screen, or you can customize your dialog boxes.

Use the `GetLastEditionContainerUsed` function to get the default edition to display.

Use the `NewSubscriberDialog` function to display the subscriber dialog box on the user's screen and use the `NewPublisherDialog` function to display the publisher dialog box on the user's screen. Unlike the Standard File Package routines, the `NewPublisherDialog` and the `NewSubscriberDialog` functions allow you to specify the initial volume reference number and directory ID so that there can be one default location for editions for all applications.

You use the `SectionOptionsDialog` function to display the publisher options and subscriber options dialog boxes on the user's screen.

The `NewSubscriberExpDialog`, `NewPublisherExpDialog`, and `SectionOptionsExpDialog` functions are the same as the simple dialog functions but have five additional parameters.

GetLastEditionContainerUsed

Use the `GetLastEditionContainerUsed` function to get the default edition to display. This function allows a user to easily subscribe to the data recently published.

```
FUNCTION GetLastEditionContainerUsed
                  (VAR container: EditionContainerSpec): OSErr;
```

`container` If the `GetLastEditionContainerUsed` function locates the last edition for which a section was created, the `container` parameter contains its volume reference number, directory ID, filename, and part, and returns a `noErr` result code. (The last edition created is associated with the last time that your application or another application located on the same machine used the `NewSection` function.)

DESCRIPTION

If the last edition used is missing, the `GetLastEditionContainerUsed` function returns an `fnfErr` result code, but still returns the correct volume reference number and directory ID that you should use for the `NewSubscriberDialog` function.

Pass the information from the `GetLastEditionContainerUsed` function to the `NewSubscriberDialog` function.

RESULT CODES

`noErr`	0	No error
`fnfErr`	–43	Edition container not found
`editionMgrInitErr`	–450	Manager not initialized

SEE ALSO

For an example of the use of `GetLastEditionContainerUsed`, see Listing 2-6 beginning on page 2-40. For a description of the edition container record, see page 2-71. The `NewSubscriberDialog` function is described next.

NewSubscriberDialog

When a user chooses the Subscribe To menu command, your application should call the `NewSubscriberDialog` function to allow the user to choose an edition to subscribe to.

```
FUNCTION NewSubscriberDialog
                    (VAR reply: NewSubscriberReply): OSErr;
```

`reply` The new subscriber reply record. You specify a location to use as the default edition container in the `container` field of this record. You also specify in the `formatsMask` field which edition format types `NewSubscriberDialog` should display. The `NewSubscriberDialog` function returns information concerning the user's choice in the `canceled` and `container` fields of this record.

```
TYPE NewSubscriberReply =
   RECORD
      canceled:      Boolean;             {user canceled }
                                          { dialog box}
      formatsMask:   SignedByte;          {formats required}
      container:     EditionContainerSpec;{edition selected}
   END;
```

Field descriptions

`canceled` The `NewSubscriberDialog` function returns in this field a value that indicates whether the user canceled the dialog box. The function returns `TRUE` in the `canceled` field if the user canceled the dialog box. Otherwise, the function returns `FALSE` in this field and returns in the `container` field the edition container for the new subscriber.

`formatsMask`	The `formatsMask` field indicates which edition format type (text, graphics, and sound) to display within the subscriber dialog box. You can set the `formatsMask` field to the following constants: `kTEXTformatMask` (1), `kPICTformatMask` (2), or `ksndFormatMask` (4). To support a combination of formats, add the constants together. For example, a `formatsMask` of 3 displays both graphics and text edition format types in the subscriber dialog box.
`container`	The edition container of the last edition published or subscribed to. You provide in this parameter the location and filename to use as the default edition to subscribe to. If the user clicks the Subscribe button, `NewSubscriberDialog` returns `FALSE` in the `canceled` field and returns the selected edition container for the new subscriber in the `container` field.

DESCRIPTION

The `NewSubscriberDialog` function displays the subscriber dialog box on the user's screen. The `NewSubscriberDialog` function (which is based on the `CustomGetFile` procedure described in the chapter "Standard File Package" in *Inside Macintosh: Files*) switches to the volume reference number and directory ID and selects the filename of the edition container that you specified in the `container` field of the `reply` parameter. Use the `GetLastEditionContainerUsed` function to get the edition container of the last edition that was either published or subscribed to, then set the `container` field to this edition container. This allows the user to publish and then easily subscribe.

Note that if an edition does not contain either `'PICT'`, `'TEXT'`, or `'snd '` data, the `NewSubscriberDialog` function does not list the edition file in the new subscriber dialog box (unless you install an opener that can recognize the edition's data in response to the `eoCanSubscribe` verb).

RESULT CODES

`noErr`	0	No error
`editionMgrInitErr`	–450	Manager not initialized or could not load package
`badSubPartErr`	–454	Bad edition container spec

SEE ALSO

For an illustration of the new subscriber dialog box, see Figure 2-12 on page 2-37. For an example of the use of `NewSubscriberDialog`, see Listing 2-6 beginning on page 2-40. For a description of the edition container record, see page 2-71. For information on edition openers, see "Subscribing to Non-Edition Files" beginning on page 2-62.

NewPublisherDialog

When a user selects a portion of a document and then chooses the Create Publisher menu command, your application should call the `NewPublisherDialog` function to allow the user to choose a name and location of the edition to which your application writes the publisher data. Your application specifies a location and name to use as the default edition and provides a preview of the publisher data to the `NewPublisherDialog` function.

```
FUNCTION NewPublisherDialog
                (VAR reply: NewPublisherReply): OSErr;
```

`reply` A new publisher reply record. You specify a location to use as the default edition container in the `container` field of this record. You also specify information in the `usePart`, `preview`, and `previewFormat` fields. The `NewPublisherDialog` function returns information concerning the user's choice in the `canceled`, `replacing`, and `container` fields of this record.

```
TYPE NewPublisherReply =
   RECORD
      canceled:      Boolean;       {user canceled dialog box}
      replacing:     Boolean;       {user chose existing }
                                    { filename for an edition}
      usePart:       Boolean;       {always false in version 7.0}
      preview:       Handle;        {handle to 'prvw', 'PICT', }
                                    { 'TEXT', or 'snd ' data}
      previewFormat: FormatType;    {type of preview}
      container:     EditionContainerSpec;
                                    {edition chosen}
   END;
```

Field descriptions

`canceled` The `NewPublisherDialog` function returns in this field a value that indicates whether the user canceled the dialog box. The function returns `TRUE` in the `canceled` field if the user canceled the dialog box. The function returns `FALSE` in this field if the user clicked the Publish button and returns in the `container` field the edition container for the new publisher.

`replacing` The `NewPublisherDialog` function returns `TRUE` in the `replacing` field if the user chose an existing filename from the list of available editions and confirmed this replacement. If the value of the `replacing` field is `TRUE`, do not call the `CreateEditionContainerFile` function. If the value of this field and the `canceled` field is `FALSE`, you can call `CreateEditionContainerFile` to create a new edition container.

`usePart`	A value that must be set to `FALSE` before calling the `NewPublisherDialog` function.
`preview`	A handle to `'prvw'`, `'PICT'`, `'TEXT'`, or `'snd '` data. The `NewPublisherDialog` function displays this data in the preview area of the dialog box.
`previewFormat`	A value that indicates which type of data the handle in the `preview` field references.
`container`	An edition container record that specifies the volume reference number, directory ID, and filename to use as the default edition to publish the data to. The `NewPublisherDialog` function returns in this field the edition container that the user selected.

DESCRIPTION

The `NewPublisherDialog` function displays the new publisher dialog box on the user's screen. The `NewPublisherDialog` function (which is based on the `CustomPutFile` procedure described in the chapter "Standard File Package" in *Inside Macintosh: Files*) switches to the volume reference number and directory ID specified by the edition container, sets the editable text item to the filename specified by the edition container, and displays a preview of the publisher data in the new publisher dialog box. The `NewPublisherDialog` function handles all user interaction until the user clicks the Cancel or Publish button.

You should deallocate the handle referenced by the `preview` field to free up memory.

RESULT CODES

`noErr`	0	No error
`editionMgrInitErr`	–450	Manager not initialized or could not load package
`badSubPartErr`	–454	Bad edition container spec

SEE ALSO

For an illustration of the new publisher dialog box, see Figure 2-11 on page 2-29. For an example of the use of `NewPublisherDialog`, see Listing 2-4 beginning on page 2-33. For a description of the edition container record, see page 2-71.

SectionOptionsDialog

Use the `SectionOptionsDialog` function to display the publisher options and subscriber options dialog boxes on the user's screen.

```
FUNCTION SectionOptionsDialog
                    (VAR reply: SectionOptionsReply): OSErr;
```

`reply` The `reply` parameter contains a section options reply record. You specify a handle to the publisher's or subscriber's section record in the `sectionH` field of this record. The `SectionOptionsDialog` function returns information concerning the user's actions in the `canceled`, `changed`, and `action` fields.

```
TYPE SectionOptionsReply =
   RECORD
      canceled:   Boolean;          {user canceled dialog box}
      changed:    Boolean;          {changed the section record}
      sectionH:   SectionHandle;    {handle to the specified }
                                    { section record}
      action:     ResType;          {action codes}
   END;
```

Field descriptions

`canceled` The `SectionOptionsDialog` function returns in this field a value that indicates whether the user canceled the dialog box. The function returns `TRUE` in the `canceled` field if the user canceled the dialog box. Otherwise, the function returns `FALSE` in this field.

`changed` The `SectionOptionsDialog` function returns `TRUE` in this field if the user changed the section record. For example, the update mode may have changed. Otherwise, the function returns `FALSE` in this field.

`sectionH` A handle to the section record for the section the user selected.

`action` The `SectionOptionsDialog` function returns in this field the code for one of five user actions: action code `'read'` for user selection of the Get Edition Now button, action code `'writ'` for user selection of the Send Edition Now button, action code `'goto'` for user selection of the Open Publisher button, action code `'cncl'` for user selection of the Cancel Publisher or Cancel Subscriber button, or action code `'    '` ($20202020) for user selection of the OK button.

DESCRIPTION

The `SectionOptionsDialog` function displays the appropriate options dialog box for the specified section record. The function displays information about the subscriber or publisher, such as its latest edition and current update mode setting, and allows the user to perform various actions. The `SectionOptionsDialog` function handles all user interaction until the user selects a button. The function returns the user's action in the `action` field of the `reply` parameter; your application should then perform the corresponding action.

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Memory full

SEE ALSO

For illustrations of the section options dialog box, see Figure 2-13 through Figure 2-16 beginning on page 2-43. For an example of the use of `SectionOptionsDialog`, see Listing 2-8 beginning on page 2-46. For a description of the section record, see page 2-72.

NewSubscriberExpDialog, NewPublisherExpDialog, SectionOptionsExpDialog

The `NewSubscriberExpDialog`, `NewPublisherExpDialog`, and `SectionOptionsExpDialog` functions are the same as the simple dialog functions but have five additional parameters. These additional parameters allow you to add items to the bottom of the dialog boxes, apply alternate mapping of events to item hits, apply alternate meanings to the item hits, and choose the location of the dialog boxes.

```
FUNCTION NewSubscriberExpDialog
               (VAR reply: NewSubscriberReply; where: Point;
                expansionDITLresID: Integer;
                dlgHook: ExpDlgHookProcPtr;
                filterProc: ExpModalFilterProcPtr;
                yourDataPtr: UNIV Ptr): OSErr;

FUNCTION NewPublisherExpDialog
               (VAR reply: NewPublisherReply; where: Point;
                expansionDITLresID: Integer;
                dlgHook: ExpDlgHookProcPtr;
                filterProc: ExpModalFilterProcPtr;
                yourDataPtr: UNIV Ptr): OSErr;

FUNCTION SectionOptionsExpDialog
               (VAR reply: SectionOptionsReply; where: Point;
                expansionDITLresID: Integer;
                dlgHook: ExpDlgHookProcPtr;
                filterProc: ExpModalFilterProcPtr;
                yourDataPtr: UNIV Ptr): OSErr;
```

`reply` A new subscriber reply, new publisher reply, or section options reply record. You specify information in the fields of this record just as you do in the the corresponding fields of records used by `NewSubscriberDialog`, `NewPublisherDialog`, and `SectionOptionsDialog`.

`where` A point that specifies a location on the screen where the function displays the dialog box. You can automatically center the dialog box by passing (–1, –1) in the `where` parameter.

`expansionDITLresID`
A value of 0 or a valid item list (`'DITL'`) resource ID. This integer is the ID of a dialog item list whose items are appended to the end of the standard dialog item list. The dialog items keep their relative positions, but they are moved as a group to the bottom of the dialog box.

`dlgHook` A pointer to an expandable dialog hook function or `NIL`. An expandable dialog hook function is similar to a dialog hook function except that an expandable dialog hook function accepts an additional parameter. The `NewSubscriberExpDialog`, `NewPublisherExpDialog`, and `SectionOptionsExpDialog` functions call your expandable dialog hook function after each call to the `ModalDialog` procedure. The expandable dialog hook function should take the appropriate action, such as filling in a checkbox. The `itemOffset` parameter to the expandable dialog hook function is the number of items in the item list before your expansion dialog items. You need to subtract the item offset from the item hit to get the relative item number in the expansion item list. The expandable dialog hook function should return as its function result the absolute item number.

`filterProc`
A pointer to an expandable modal-dialog filter function or `NIL`. An expandable modal-dialog filter function is similar to a modal-dialog filter function or event filter function except that an expandable modal-dialog filter function accepts two extra parameters. The `ModalDialog` procedure calls the expandable modal-dialog filter function you provide in this parameter. An expandable modal-dialog filter function allows you to map real events (such as a mouse-down event) to an item hit (such as clicking a Cancel button). For instance, you may want to map a keyboard equivalent to an item hit.

`yourDataPtr`
Reserved for your use. It is passed back to your hook and event filter function. This parameter does not have to be of type `Ptr`—it can be any 32-bit quantity that you want. In Pascal, you can pass `yourDataPtr` in register A6, and declare your dialog hook and event filter as local functions without the last parameter. The stack frame is set up properly for these functions to access their parent local variables.

DESCRIPTION

The `NewPublisherExpDialog`, `NewSubscriberExpDialog`, and `SectionOptionsExpDialog` functions display the appropriate dialog box, handle user interaction, and call any functions you have provided in the `dlgHook` and `filterProc` parameters.

For the `NewPublisherExpDialog` and `NewSubscriberExpDialog` functions, all the pseudo-items for the Standard File Package such as `hookFirstCall`(–1), `hookNullEvent`(100), `hookRebuildList`(101), and `hookLastCall`(–2) can be used, as well as `hookRedrawPreview`(150).

For the `SectionOptionsExpDialog` function, the only valid pseudo-items are `hookFirstCall`(–1), `hookNullEvent`(100), `hookLastCall`(–2), `emHookRedrawPreview`(150), `emHookCancelSection`(160), `emHookGoToPublisher`(161), `emHookGetEditionNow`(162), `emHookSendEditionNow`(162), `emHookManualUpdateMode`(163), and `emHookAutoUpdateMode`(164).

If you provide an expandable dialog hook function, it must contain the following parameters:

```
FUNCTION MyExpDlgHook (itemOffset: Integer; itemHit: Integer;
                       theDialog: DialogPtr;
                       yourDataPtr: Ptr): Integer;
```

If you provide an expandable modal-dialog filter function, it must contain the following parameters.

```
FUNCTION MyExpModalFilter (theDialog: DialogPtr;
                           VAR theEvent: EventRecord;
                           itemOffset: Integer;
                           VAR itemHit: Integer;
                           yourDataPtr: Ptr): Boolean;
```

SEE ALSO

See the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for additional information on item lists. See the chapter "Standard File Package" in *Inside Macintosh: Files* for information on dialog hook and modal-dialog filter functions.

Locating a Publisher and Edition From a Subscriber

The `GetEditionInfo` function returns information about a section's edition such as its location, last modification date, creator, and type.

Once you locate a section's edition, you can use the `GoToPublisherSection` function to find the document containing the publisher.

GetEditionInfo

Use the `GetEditionInfo` function to obtain information about a section's edition, such as its location, last modification date, creator, and type.

```
FUNCTION GetEditionInfo
                     (sectionH: SectionHandle;
                      VAR editionInfo: EditionInfoRecord): OSErr;
```

`sectionH` A handle to the section record for a given section.

`editionInfo`
An edition information record. The `GetEditionInfo` function returns the public information contained in the section's control block.

DESCRIPTION

The Edition Manager ensures that the existing edition name corresponds to the Finder's existing edition name. If the `controlBlock` field of the section record is set to `NIL` or the edition cannot be located, the `GetEditionInfo` function returns an `fnfErr` result code.

The `GetEditionInfo` function returns information about the section's edition in a data structure of type `EditionInfoRecord`.

```
TYPE EditionInfoRecord =
   RECORD
      crDate:     TimeStamp;           {date edition container }
                                       { was created}
      mdDate:     TimeStamp;           {date of last change}
      fdCreator:  OSType;              {file creator}
      fdType:     OSType;              {file type}
      container:  EditionContainerSpec;{the edition}
   END;
```

Field descriptions

`crDate` The creation date of the edition.

`mdDate` The modification date of the edition.

`fdCreator` The creator of the edition file.

`fdType` The file type of the edition file.

`container` An edition container record, which specifies the volume reference number, directory ID, filename, script, and part number for the edition.

RESULT CODES

`noErr`	0	No error
`fnfErr`	–43	Not registered or file moved
`editionMgrInitErr`	–450	Manager not initialized

SEE ALSO

For an example of the use of `GetEditionInfo`, see Listing 2-8 beginning on page 2-46. For another use of this function, see "Canceling Sections Within Documents" beginning on page 2-48. For a description of the edition container record, see page 2-71.

GoToPublisherSection

When the user wants to locate the publisher for a particular subscriber (by clicking Open Publisher in the subscriber options dialog box), the `SectionOptionsDialog` function returns the action code `'goto'` in the `action` field of the section options reply record. When you receive this action code, you should open the document containing the publisher.

First, use the `GetEditionInfo` function to find the edition container. Then use the `GoToPublisherSection` function to open the document containing the publisher.

```
FUNCTION GoToPublisherSection
                          (container: EditionContainerSpec): OSErr;
```

`container` An edition container record, which specifies volume reference number, directory ID, and filename of the subscriber's edition. You obtain the edition container by calling the `GetEditionInfo` function.

DESCRIPTION

The `GoToPublisherSection` function resolves the alias in the edition to find the document containing its publisher. In general, this function internally uses the `GetStandardFormats` function to get the alias to the publisher document and then resolves the alias. It next sends the Finder an Apple event to open the document (which launches its application if necessary) and, after the publisher is registered, sends a Section Scroll event to the publisher.

As an optimization, if there is a registered publisher, the `GoToPublisherSection` function simply sends a Section Scroll event to the publisher.

If the edition does not contain an alias and there are no registered publishers, then the `GoToPublisherSection` function sends an Open Documents event to open the edition to the creating application.

If the edition container is not an edition file (as is the case when you are using bottlenecks to subscribe to non-edition files), the `GoToPublisherSection` function sends the Finder an Apple event to open that file.

RESULT CODES

`noErr`	0	No error
`fnfErr`	–43	File not found
`editionMgrInitErr`	–450	Manager not initialized
`badSubPartErr`	–454	Invalid edition container

SEE ALSO

For illustrations of the section options dialog box for subscribers, see Figure 2-15 on page 2-44 and Figure 2-16 on page 2-45. For an example of responding to the action code `'goto'`, see Listing 2-8 beginning on page 2-46. For a description of the edition container record, see page 2-71.

Edition Container Formats

The Edition Manager calls the `GetStandardFormats` function to get the alias used in the `GoToPublisherSection` function and to get the preview shown in the subscriber dialog box. You probably do not need to call this function directly.

GetStandardFormats

You probably do not need to call the `GetStandardFormats` function directly because the Edition Manager calls this function.

```
FUNCTION GetStandardFormats
              (container: EditionContainerSpec;
               VAR previewFormat: FormatType;
               preview, publisherAlias, formats: Handle): OSErr;
```

`container` An edition container record that specifies the edition volume reference number, directory ID, filename, and part.

`previewFormat`
The `GetStandardFormats` function returns in this parameter a handle to the first format of the requested format type that it finds in the edition.

`preview` A format type. The `GetStandardFormats` function looks for a format of the type specified in this parameter and returns in this parameter the format type of the first format that it finds. The function tries to find one of four formats: `'prvw'`, `'PICT'`, `'TEXT'`, or `'snd '`.

`publisherAlias`
The `publisherAlias` parameter reads the format `kPublisherDocAliasFormat` (`'alis'`).

`formats` The `formats` parameter reads the virtual format `kFormatListFormat` (`'fmts'`).

DESCRIPTION

You should pass in valid handles for the formats that you want and `NIL` for the formats that you don't want. The handles are resized to the size of the data.

If one of the requested formats cannot be found, `GetStandardFormats` returns a `noTypeErr` result code.

RESULT CODES

`noErr`	0	No error
`noTypeErr`	–102	Edition container not found
`editionMgrInitErr`	–450	Manager not initialized

Reading and Writing Non-Edition Files

The Edition Manager never opens or closes an edition container directly—it calls the current edition opener. See "Subscribing to Non-Edition Files" beginning on page 2-62 for additional information.

To override the standard opener function, create an opener function that contains the following parameters:

```
FUNCTION MyOpener (selector: EditionOpenerVerb;
                   VAR PB: EditionOpenerParamBlock): OSErr;
```

When this function is called by the Edition Manager, the `selector` parameter is set to one of the edition opener verbs (`eoOpen`, `eoClose`, `eoOpenNew`, `eoCloseNew`, `eoCanSubscribe`). The `PB` parameter contains an edition opener parameter block record.

Use the `GetEditionOpenerProc` function to locate the current edition opener and use the `SetEditionOpenerProc` function to provide your own edition opener.

Use the `CallEditionOpenerProc` function to call an edition opener and use the `CallFormatIOProc` function to call a format I/O function.

GetEditionOpenerProc

Use the `GetEditionOpenerProc` function to locate the current edition opener.

```
FUNCTION GetEditionOpenerProc
                     (VAR opener: EditionOpenerProcPtr): OSErr;
```

`opener` The `GetEditionOpenerProc` function returns a pointer to the current edition opener function in this parameter.

SetEditionOpenerProc

Use the `SetEditionOpenerProc` function to provide your own edition opener.

```
FUNCTION SetEditionOpenerProc
                     (opener: EditionOpenerProcPtr): OSErr;
```

`opener` A pointer to the edition opener function that you are providing.

CallEditionOpenerProc

Use the `CallEditionOpenerProc` function to call an edition opener.

```
FUNCTION CallEditionOpenerProc
                    (selector: EditionOpenerVerb;
                     VAR PB: EditionOpenerParamBlock;
                     routine: EditionOpenerProcPtr): OSErr;
```

`selector` An edition opener verb. When the `CallEditionOpenerProc` function is called by the Edition Manager, the `selector` parameter is set to one of the edition opener verbs (`eoOpen`, `eoClose`, `eoOpenNew`, `eoCloseNew`, `eoCanSubscribe`).

`PB` An edition opener parameter block.

`routine` A pointer to an edition opener function.

DESCRIPTION

The Edition Manager calls an edition opener function whenever it needs to open or close an edition. The Edition Manager passes an edition opener parameter block as one of the parameters to an edition opener function. The edition opener parameter block is defined by this structure:

```
TYPE EditionOpenerParamBlock =
   RECORD
      info:          EditionInfoRecord;   {edition container to }
                                          { be subscribed to}
      sectionH:      SectionHandle;       {publisher or }
                                          { subscriber }
                                          { requesting open}
      document:      FSSpecPtr;           {document passed}
      fdCreator:     OSType;              {Finder creator type}
      ioRefNum:      LongInt;             {reference number}
      ioProc:        FormatIOProcPtr;     {routine to read }
                                          { formats}
      success:       Boolean;             {reading or writing }
                                          { was successful}
      formatsMask:   SignedByte;          {formats required to }
                                          { subscribe}
   END;
```

To override the standard reading and writing functions, you should create an I/O function that contains the following parameters.

```
FUNCTION MyIO (selector: FormatIOVerb;
               VAR PB: FormatIOParamBlock): OSErr;
```

Set the `selector` parameter to one of the format I/O verbs (`ioHasFormat`, `ioReadFormat`, `ioNewFormat`, `ioWriteFormat`). The `PB` parameter contains a format I/O parameter block record.

SEE ALSO

See "Calling an Edition Opener" beginning on page 2-64 for additional information.

CallFormatIOProc

Use the `CallFormatIOProc` function to call a format I/O function.

```
FUNCTION CallFormatIOProc (selector: FormatIOVerb;
                           VAR PB:FormatIOParamBlock;
                           routine: FormatIOProcPtr): OSErr;
```

`selector` A format I/O verb (`ioHasFormat`, `ioReadFormat`, `ioNewFormat`, `ioWriteFormat`).

`PB` A format I/O parameter block record.

`routine` A pointer to a format I/O function.

DESCRIPTION

The Edition Manager calls a format I/O function whenever it needs to read from or write to an edition. The Edition Manager passes a format I/O parameter block as one of the parameters to a format I/O procedure. The format I/O parameter block is defined by this structure:

```
TYPE FormatIOParamBlock =
   RECORD
      ioRefNum:      LongInt;       {reference number}
      format:        FormatType;    {edition format type}
      formatIndex:   LongInt;       {opener-specific enumeration }
                                    { of formats}
      offset:        LongInt;       {offset into format}
      buffPtr:       Ptr;           {data starts here}
      buffLen:       LongInt;       {length of data}
   END;
```

SEE ALSO

See "Calling a Format I/O Function" beginning on page 2-68 for additional information.

Application-Defined Routines

Your application can provide an edition opener function, format I/O function, expandable dialog hook function, and expandable modal-dialog filter function. For the routine declarations of the edition opener and format I/O functions, see "Reading and Writing Non-Edition Files" beginning on page 2-102. For the routine declarations of the expandable dialog hook and expandable modal-dialog filter functions, see the description of `NewSubscriberExpDialog`, `NewPublisherExpDialog`, and `SectionOptionsExpDialog` beginning on page 2-96.

Summary of the Edition Manager

Pascal Summary

Constants

```
CONST
   {resource types}
   rSectionType                  = 'sect';   {resource type for a }
                                             { section}
   {section types}
   stSubscriber                  = $01;      {subscriber section type}
   stPublisher                   = $0A;      {publisher section type}
   {update modes}
   sumAutomatic                  = 0;        {subscriber receives new }
                                             { editions automatically}
   sumManual                     = 1;        {subscriber receives new }
                                             { editions manually}
   pumOnSave                     = 0;        {publisher sends new }
                                             { editions on save}
   pumManual                     = 1;        {publisher does not send }
                                             { new editions until user }
                                             { request}
   {edition container subpart number}
   kPartsNotUsed                 = 0;        {edition is the whole file}
   kPartNumberUnknown            = -1;       {not used in version 7.0}

   {preview size}
   kPreviewWidth                 = 120;      {preview width}
   kPreviewHeight                = 120;      {preview height}

   {special formats}
   kPublisherDocAliasFormat      = 'alis';   {alias record from the }
                                             { edition to publisher}
   kPreviewFormat                = 'prvw';   {'PICT' thumbnail sketch}
   kFormatListFormat             = 'fmts';   {list of all available }
                                             { formats and their sizes}
```

```
{bits for formatMask}
kPICTformatMask              = 1;        {graphics format}
kTEXTformatMask              = 2;        {text format}
ksndFormatMask               = 4;        {sound format}

{Finder types for edition files}
kPICTEditionFileType         = 'edtp';   {contains 'PICT', }
kTEXTEditionFileType         = 'edtt';   { 'TEXT', and }
ksndEditionFileType          = 'edts';   { 'snd ' file types}
kUnknownEditionFileType      = 'edtu';   {unknown file type}
{miscellaneous}
kFormatLengthUnknown         = -1;       {length of format unknown}

{message IDs for Apple events sent by the Edition Manager}
sectionEventMsgClass         = 'sect';   {Apple events sent by the }
                                         { Edition Manager}
sectionReadMsgID             = 'read';   {Section Read events}
sectionWriteMsgID            = 'writ';   {Section Write events}
sectionScrollMsgID           = 'scrl';   {Section Scroll events}
sectionCancelMsgID           = 'cncl';   {Section Cancel events}

{refCon field when displaying stacked dialog boxes}
sfMainDialogRefCon           = 'stdf';   {new publisher and }
                                         { new subscriber}
sfNewFolderDialogRefCon      = 'nfdr';   {new folder}
sfReplaceDialogRefCon        = 'rplc';   {replace dialog}
sfStatWarnDialogRefCon       = 'stat';   {warning dialog}
sfErrorDialogRefCon          = 'err ';   {error dialog}
emOptionsDialogRefCon        = 'optn';   {options dialog}
emCancelSectionDialogRefCon  = 'cncl';   {cancel section}
emGotoPubErrDialogRefCon     = 'gerr';   {locate publisher}

{pseudo-item hits for dialogHooks}
emHookRedrawPreview          = 150;      {for NewPublisher or }
                                         { NewSubscriber dialogs}
emHookCancelSection          = 160;      {for SectionOptions dialog}
emHookGoToPublisher          = 161;      {for SectionOptions dialog}
emHookGetEditionNow          = 162;      {for SectionOptions dialog}
emHookSendEditionNow         = 162;      {for SectionOptions dialog}
emHookManualUpdateMode       = 163;      {for SectionOptions dialog}
emHookAutoUpdateMode         = 164;      {for SectionOptions dialog}
```

Data Types

```
TYPE TimeStamp                  = LongInt;     {seconds since 1904}
   EditionRefNum                = Handle;      {for use in Edition I/O}
   UpdateMode                   = Integer;     {sumAutomatic, }
                                               { sumManual, }
                                               { pumOnSave, pumManual}
   SectionType                  = SignedByte;  {stSubscriber or }
                                               { stPublisher}
   FormatType                   = PACKED ARRAY[1..4] OF CHAR;
                                               {similar to ResType used }
                                               { by the Scrap Manager}

   SectionHandle                = ^SectionPtr;
   SectionPtr                   = ^SectionRecord;
   SectionRecord                =
   RECORD
      version:                  SignedByte;    {always 1 in version 7.0}
      kind:                     SectionType;   {publisher or subscriber}
      mode:                     UpdateMode;    {automatic or manual}
      mdDate:                   TimeStamp;     {last change to section}
      sectionID:                LongInt;       {application-specific, }
                                               { unique per document}
      refCon:                   LongInt;       {application-specific}
      alias:                    AliasHandle;   {handle to alias record}

      {The following fields are private and are set up by the }
      { RegisterSection function.}

      subPart:                  LongInt;       {private}
      nextSection:              SectionHandle; {private}
      controlBlock:             Handle;        {private}
      refNum:                   EditionRefNum; {private}
   END;

   EditionContainerSpecPtr =^EditionContainerSpec;
   EditionContainerSpec =
   RECORD
      theFile:                  FSSpec;        {file containing edition }
                                               { data}
      theFileScript:            ScriptCode;    {script code of filename}
      thePart:                  LongInt;       {which part of file, }
                                               { always kPartsNotUsed}
```

```
   thePartName:              Str31;         {reserved}
   thePartScript:            ScriptCode;    {reserved}
END;

FormatsAvailable = ARRAY[0..0] OF
RECORD
   theType:                  FormatType;    {format type for an }
                                            { edition}
   theLength:                LongInt;       {length of edition format }
                                            { type}
END;

EditionInfoRecord =
RECORD
   crDate:                   TimeStamp;     {date edition container }
                                            { was created}
   mdDate:                   TimeStamp;     {date of last change}
   fdCreator:                OSType;        {file creator}
   fdType:                   OSType;        {file type}
   container:                EditionContainerSpec;
                                            {the edition}
END;

NewPublisherReply =
RECORD
   canceled:         Boolean;             {user canceled dialog box}
   replacing:        Boolean;             {user chose existing }
                                          { filename for an edition}
   usePart:          Boolean;             {always FALSE in version 7.0}
   preview:          Handle;              {handle to 'prvw', 'PICT',}
                                          { 'TEXT', or 'snd ' data}
   previewFormat:    FormatType;          {type of preview}
   container:        EditionContainerSpec;
                                          {edition chosen}
END;

NewSubscriberReply =
RECORD
   canceled:         Boolean;             {user canceled dialog box}
   formatsMask:      SignedByte;          {formats required}
   container:        EditionContainerSpec;
                                          {edition selected}
END;
```

```
   SectionOptionsReply =
   RECORD
      canceled:         Boolean;             {user canceled dialog box}
      changed:          Boolean;             {changed the section }
                                             { record}
      sectionH:         SectionHandle;       {handle to the specified }
                                             { section record}
      action:           ResType;             {action codes}
   END;

   EditionOpenerVerb= (eoOpen, eoClose, eoOpenNew, eoCloseNew,
                       eoCanSubscribe);
   EditionOpenerParamBlock =
   RECORD
      info:             EditionInfoRecord;   {edition container to }
                                             { be subscribed to}
      sectionH:         SectionHandle;       {publisher or subscriber }
                                             { requesting open}
      document:         FSSpecPtr;           {document passed}
      fdCreator:        OSType;              {Finder creator type}
      ioRefNum:         LongInt;             {reference number}
      ioProc:           FormatIOProcPtr;     {routine to read formats}
      success:          Boolean;             {reading or writing was }
                                             { successful}
      formatsMask:      SignedByte;          {formats required to }
                                             { subscribe}
   END;

   FormatIOVerb = (ioHasFormat, ioReadFormat, ioNewFormat, ioWriteFormat);
   FormatIOParamBlock =
   RECORD
      ioRefNum:         LongInt;             {reference number}
      format:           FormatType;          {edition format type}
      formatIndex:      LongInt;             {opener-specific enumeration }
                                             { of formats}
      offset:           LongInt;             {offset into format}
      buffPtr:          Ptr;                 {data starts here}
      buffLen:          LongInt;             {length of data}
END;
```

Edition Manager Routines

Initializing the Edition Manager

```
FUNCTION InitEditionPack    : OSErr;
```

Creating and Registering a Section

```
FUNCTION NewSection         (container: EditionContainerSpec;
                             sectionDocument: FSSpecPtr; kind: SectionType;
                             sectionID: LongInt; initialMode: UpdateMode;
                             VAR sectionH: SectionHandle): OSErr;
FUNCTION RegisterSection    (sectionDocument: FSSpec;
                             sectionH: SectionHandle;
                             VAR aliasWasUpdated: Boolean)
                             : OSErr;
FUNCTION UnRegisterSection  (sectionH: SectionHandle): OSErr;
FUNCTION IsRegisteredSection
                            (sectionH: SectionHandle): OSErr;
FUNCTION AssociateSection   (sectionH: SectionHandle;
                             newSectionDocument: FSSpecPtr): OSErr;
```

Creating and Deleting an Edition Container

```
FUNCTION CreateEditionContainerFile
                           (editionFile: FSSpec; fdCreator: OSType;
                            editionFileNameScript: ScriptCode): OSErr;
FUNCTION DeleteEditionContainerFile
                           (editionFile: FSSpec): OSErr;
```

Setting and Getting a Format Mark

```
FUNCTION SetEditionFormatMark
                           (whichEdition: EditionRefNum;
                            whichFormat: FormatType;
                            setMarkTo: LongInt): OSErr;
FUNCTION GetEditionFormatMark
                           (whichEdition: EditionRefNum;
                            whichFormat: FormatType;
                            VAR currentMark: LongInt): OSErr;
```

Reading in Edition Data

```
FUNCTION OpenEdition         (subscriberSectionH: SectionHandle;
                              VAR refNum: EditionRefNum): OSErr;
FUNCTION EditionHasFormat    (whichEdition: EditionRefNum;
                              whichFormat: FormatType;
                              VAR formatSize: Size): OSErr;
FUNCTION ReadEdition         (whichEdition: EditionRefNum;
                              whichFormat: FormatType; buffPtr: UNIV Ptr;
                              VAR buffLen: Size): OSErr;
```

Writing out Edition Data

```
FUNCTION OpenNewEdition      (publisherSectionH: SectionHandle;
                              fdCreator: OSType;
                              publisherSectionDocument: FSSpecPtr;
                              VAR refNum: EditionRefNum): OSErr;
FUNCTION WriteEdition        (whichEdition: EditionRefNum;
                              whichFormat: FormatType; buffPtr: UNIV Ptr;
                              buffLen: Size): OSErr;
```

Closing an Edition After Reading or Writing

```
FUNCTION CloseEdition        (whichEdition: EditionRefNum;
                              successful: Boolean): OSErr;
```

Displaying Dialog Boxes

```
FUNCTION GetLastEditionContainerUsed
                             (VAR container: EditionContainerSpec): OSErr;
FUNCTION NewSubscriberDialog
                             (VAR reply: NewSubscriberReply): OSErr;
FUNCTION NewPublisherDialog  (VAR reply: NewPublisherReply): OSErr;
FUNCTION SectionOptionsDialog
                             (VAR reply: SectionOptionsReply): OSErr;
FUNCTION NewSubscriberExpDialog
                             (VAR reply: NewSubscriberReply; where: Point;
                              expansionDITLresID: Integer;
                              dlgHook: ExpDlgHookProcPtr;
                              filterProc: ExpModalFilterProcPtr;
                              yourDataPtr: UNIV Ptr): OSErr;
```

```
FUNCTION NewPublisherExpDialog
                            (VAR reply: NewPublisherReply; where: Point;
                             expansionDITLresID: Integer;
                             dlgHook: ExpDlgHookProcPtr;
                             filterProc: ExpModalFilterProcPtr;
                             yourDataPtr: UNIV Ptr): OSErr;
FUNCTION SectionOptionsExpDialog
                            (VAR reply: SectionOptionsReply; where: Point;
                             expansionDITLresID: Integer;
                             dlgHook: ExpDlgHookProcPtr;
                             filterProc: ExpModalFilterProcPtr;
                             yourDataPtr: UNIV Ptr): OSErr;
```

Locating a Publisher and Edition From a Subscriber

```
FUNCTION GetEditionInfo     (sectionH: SectionHandle;
                             VAR editionInfo: EditionInfoRecord): OSErr;
FUNCTION GoToPublisherSection
                            (container: EditionContainerSpec): OSErr;
```

Edition Container Formats

```
FUNCTION GetStandardFormats (container: EditionContainerSpec;
                             VAR previewFormat: FormatType;
                             preview, publisherAlias,
                             formats: Handle): OSErr;
```

Reading and Writing Non-Edition files

```
FUNCTION GetEditionOpenerProc
                            (VAR opener: EditionOpenerProcPtr): OSErr;
FUNCTION SetEditionOpenerProc
                            (opener: EditionOpenerProcPtr): OSErr;
FUNCTION CallEditionOpenerProc
                            (selector: EditionOpenerVerb;
                             VAR PB: EditionOpenerParamBlock;
                             routine: EditionOpenerProcPtr): OSErr;
FUNCTION CallFormatIOProc   (selector: FormatIOVerb;
                             VAR PB: FormatIOParamBlock;
                             routine: FormatIOProcPtr): OSErr;
```

Application-Defined Routines

```
FUNCTION MyExpDlgHook       (itemOffset: Integer; itemHit: Integer;
                             theDialog: DialogPtr;
                             yourDataPtr: Ptr): Integer;
```

```
FUNCTION MyExpModalFilter   (theDialog: DialogPtr;
                             VAR theEvent: EventRecord;
                             itemOffset: Integer; VAR itemHit: Integer;
                             yourDataPtr: Ptr): Boolean;
FUNCTION MyOpener           (selector: EditionOpenerVerb;
                             VAR PB: EditionOpenerParamBlock): OSErr;
FUNCTION MyIO               (selector: FormatIOVerb;
                             VAR PB: FormatIOParamBlock): OSErr;
```

C Summary

Constants

```
CONST
enum {
   /*resource types*/
   #define rSectionType           'sect'     /*resource type for a */
                                             /* section*/

   /*section types*/
   stSubscriber                 = 0x01,     /*subscriber section type*/
   stPublisher                  = 0x0A,     /*publisher section type*/

   /*update modes*/
   sumAutomatic                 = 0,        /*subscriber receives new */
                                            /* editions automatically*/
   sumManual                    = 1,        /*subscriber receives new */
                                            /* editions manually*/
   pumOnSave                    = 0,        /*publisher sends new */
                                            /* editions on save*/
   pumManual                    = 1,        /*publisher does not send */
                                            /* new editions until user */
                                            /* request*/

   /*edition container subpart number*/
   kPartsNotUsed                = 0,        /*edition is the whole file*/
   kPartNumberUnknown           = -1,       /*not used in version 7.0*/

   /*preview size*/
   kPreviewWidth                = 120,      /*preview width*/
   kPreviewHeight               = 120,      /*preview height*/
```

```
   /*special formats*/
   #define kPublisherDocAliasFormat 'alis'   /*alias record from the */
                                             /* edition to publisher*/
   #define kPreviewFormat           'prvw'   /*'PICT' thumbnail sketch*/
   #define kFormatListFormat        'fmts'   /*list of all available */
                                             /* formats and their sizes*/

   /*bits for formatMask*/
   kPICTformatMask               = 1,        /*graphics format*/
   kTEXTformatMask               = 2,        /*text format*/
   ksndFormatMask                = 4,        /*sound format*/

   /*Finder types for edition files*/
   #define kPICTEditionFileType     'edtp'   /*contains 'PICT', */
   #define kTEXTEditionFileType     'edtt'   /* 'TEXT', and */
   #define ksndEditionFileType      'edts'   /* 'snd ' file types*/
   #define kUnknownEditionFileType  'edtu'   /*unknown file type*/

   /*pseudo-item hits for dialogHooks*/
   emHookRedrawPreview           = 150,      /*for NewPublisher or */
                                             /* NewSubscriber dialogs*/
   emHookCancelSection           = 160,      /*for SectionOptions dialog*/
   emHookGoToPublisher           = 161,      /*for SectionOptions dialog*/
   emHookGetEditionNow           = 162,      /*for SectionOptions dialog*/
   emHookSendEditionNow          = 162,      /*for SectionOptions dialog*/
   emHookManualUpdateMode        = 163,      /*for SectionOptions dialog*/
   emHookAutoUpdateMode          = 164       /*for SectionOptions dialog*/
};
   /*edition opener verbs*/
enum {eoOpen, eoClose, eoOpenNew, eoCloseNew, eoCanSubscribe};

enum {
   /*refCon field when displaying stacked dialog boxes*/
   #define emOptionsDialogRefCon        'optn'     /*options dialog*/
   #define emCancelSectionDialogRefCon  'cncl'     /*cancel section*/
   #define emGotoPubErrDialogRefCon     'gerr'     /*locate publisher*/

   kFormatLengthUnknown          = -1        /*length of format unknown*/
};
   /*refCon field when displaying stacked dialog boxes*/
   #define sfMainDialogRefCon     'stdf'     {new publisher and }
                                             { new subscriber}
   #define sfNewFolderDialogRefCon'nfdr'     {new folder}
```

```
   #define sfReplaceDialogRefCon  'rplc'     {replace dialog}
   #define sfStatWarnDialogRefCon 'stat'     {warning dialog}
   #define sfErrorDialogRefCon    'err '     {error dialog}

   /*message IDs for Apple events sent by the Edition Manager*/
   #define sectionEventMsgClass   'sect'     /*Apple events sent by the */
                                             /* Edition Manager*/
   #define sectionReadMsgID       'read'     /*Section Read events*/
   #define sectionWriteMsgID      'writ'     /*Section Write events*/
   #define sectionScrollMsgID     'scrl'     /*Section Scroll events*/
   #define sectionCancelMsgID     'cncl'     /*Section Cancel events*/
```

Data Types

```
typedef unsigned long TimeStamp;            /*seconds since 1904*/
typedef Handle EditionRefNum;`              /*used in Edition I/O*/
typedef short UpdateMode;                   /*update mode: sumAutomatic, */
                                            /* sumManual, */
                                            /* pumOnSave, pumManual*/
typedef char SectionType;                   /*one byte, stSubscriber */
                                            /* or stPublisher*/
typedef unsigned long FormatType;           /*similar to ResType*/

struct SectionRecord {
   SignedByte version;                      /*always 1x01 in version 7.0*/
   SectionType kind;                        /*stPublisher or */
                                            /* stSubscriber*/
   UpdateMode mode;                         /*automatic or manual*/
   TimeStamp mdDate;                        /*last change to section*/
   long sectionID;                          /*application-specific, */
                                            /* unique per document*/
   long refCon;                             /*application-specific*/
   AliasHandle alias;                       /*handle to alias record*/
   long subPart;                            /*private*/
   struct SectionRecord **nextSection;      /*private*/
   Handle controlBlock;                     /*private*/
   EditionRefNum refNum;                    /*private*/
};

typedef struct SectionRecord SectionRecord;
typedef SectionRecord *SectionPtr, **SectionHandle;
```

```
struct EditionContainerSpec {
   FSSpec theFile;                          /*file containing */
                                            /* edition data*/
   ScriptCode theFileScript;                /*script code of filename*/
   long thePart;                            /*which part of file, */
                                            /* always kPartsNotUsed*/
   Str31 thePartName;                       /*reserved*/
   ScriptCode thePartScript;                /*reserved*/
};

typedef struct EditionContainerSpec EditionContainerSpec;
typedef EditionContainerSpec *EditionContainerSpecPtr;

struct EditionInfoRecord {
   TimeStamp crDate;                    /*date edition container */
                                        /* was created*/
   TimeStamp mdDate;                    /*date of last change*/
   OSType fdCreator;                    /*file creator*/
   OSType fdType;                       /*file type*/
   EditionContainerSpec container;      /*the edition*/
};

typedef struct EditionInfoRecord EditionInfoRecord;

struct NewPublisherReply {
   Boolean canceled;                    /*user canceled dialog box*/
   Boolean replacing;                   /*user chose existing */
                                        /* filename for an edition*/
   Boolean usePart;                     /*always FALSE in version */
                                        /* 7.0*/
   Handle preview;                      /*handle to 'prvw', 'PICT',*/
                                        /* 'TEXT', or 'snd ' data*/
   FormatType previewFormat;            /*type of preview*/
   EditionContainerSpec container;      /*edition chosen*/
};
typedef struct NewPublisherReply NewPublisherReply;

struct NewSubscriberReply {
   Boolean canceled;                    /*user canceled dialog box*/
   unsigned char formatsMask;           /*formats required*/
   EditionContainerSpec container;      /*edition selected*/
};
typedef struct NewSubscriberReply NewSubscriberReply;
```

```
struct SectionOptionsReply {
   Boolean canceled;                    /*user canceled dialog box*/
   Boolean changed;                     /*changed the section */
                                        /* record*/
   SectionHandle sectionH;              /*handle to the specified */
                                        /* section record*/
   ResType action;                      /*action codes*/
};

typedef struct SectionOptionsReply SectionOptionsReply;

typedef pascal Boolean (*ExpModalFilterProcPtr) (DialogPtr theDialog,
                                    EventRecord *theEvent, short itemOffset,
                                    short *itemHit, Ptr yourDataPtr);

typedef pascal short (*ExpDlgHookProcPtr) (short itemOffset, short itemHit,
                                       DialogPtr theDialog, Ptr yourDataPtr);

typedef unsigned char EditionOpenerVerb;

struct EditionOpenerParamBlock {
   EditionInfoRecord info;              /*edition container to */
                                        /* be subscribed to*/
   SectionHandle sectionH;              /*publisher or subscriber */
                                        /* requesting open*/
   FSSpecPtr document;                  /*document passed*/
   OSType fdCreator;                    /*Finder creator type*/
   long ioRefNum;                       /*reference number*/
   FormatIOProcPtr ioProc;              /*routine to read formats*/
   Boolean success;                     /*reading or writing was */
                                        /* successful*/
   unsigned char formatsMask;           /*formats required to */
                                        /* subscribe*/
};
typedef struct EditionOpenerParamBlock EditionOpenerParamBlock;

typedef pascal short (*EditionOpenerProcPtr) (EditionOpenerVerb selector,
                                           FormatIOParamBlock *PB);

enum {ioHasFormat, ioReadFormat, ioNewFormat, ioWriteFormat};
typedef unsigned char FormatIOVerb;
```

```
struct FormatIOParamBlock {
   long ioRefNum;                          /*reference number*/
   FormatType format;                      /*edition format type*/
   long formatIndex;                       /* opener-specific */
                                           /* enumeration */
                                           /* of formats*/
   unsigned long offset;                   /*offset into format*/
   Ptr buffPtr;                            /*data starts here*/
   unsigned long buffLen;                  /*length of data*/
};

typedef struct FormatIOParamBlock FormatIOParamBlock;

typedef pascal short (*FormatIOProcPtr) (FormatIOVerb selector,
                                         FormatIOParamBlock *PB);
```

Edition Manager Routines

Initializing the Edition Manager

```
pascal OSErr InitEditionPack(void)
```

Creating and Registering a Section

```
pascal OSErr NewSection     (const EditionContainerSpec *container,
                             const FSSpec *sectionDocument,
                             SectionType kind, long sectionID,
                             UpdateMode initialMode,
                             SectionHandle *sectionH);
pascal OSErr RegisterSection
                            (const FSSpec *sectionDocument,
                             SectionHandle sectionH,
                             Boolean *aliasWasUpdated);
pascal OSErr UnRegisterSection
                            (SectionHandle sectionH);
pascal OSErr IsRegisteredSection
                            (SectionHandle sectionH);
pascal OSErr AssociateSection
                            (SectionHandle sectionH,
                             const FSSpec *newSectionDocument);
```

Creating and Deleting an Edition Container

```
pascal OSErr CreateEditionContainerFile
                            (const FSSpec *editionFile, OSType fdCreator,
                             ScriptCode editionFileNameScript);
pascal OSErr DeleteEditionContainerFile
                            (const FSSpec *editionFile);
```

Setting and Getting a Format Mark

```
pascal OSErr SetEditionFormatMark
                            (EditionRefNum whichEdition,
                             FormatType whichFormat,
                             unsigned long setMarkTo);
pascal OSErr GetEditionFormatMark
                            (EditionRefNum whichEdition,
                             FormatType whichFormat,
                             unsigned long *currentMark);
```

Reading in Edition Data

```
pascal OSErr OpenEdition    (SectionHandle subscriberSectionH,
                             EditionRefNum *refNum);
pascal OSErr EditionHasFormat
                            (EditionRefNum whichEdition,
                             FormatType whichFormat,
                             Size *formatSize);
pascal OSErr ReadEdition    (EditionRefNum whichEdition,
                             FormatType whichFormat, void *buffPtr,
                             Size *buffLen);
```

Writing out Edition Data

```
pascal OSErr OpenNewEdition (SectionHandle publisherSectionH,
                             OSType fdCreator,
                             const FSSpec *publisherSectionDocument,
                             EditionRefNum *refNum);
pascal OSErr WriteEdition   (EditionRefNum whichEdition,
                             FormatType whichFormat, const void *buffPtr,
                             Size *buffLen);
```

Closing an Edition After Reading or Writing

```
pascal OSErr CloseEdition   (EditionRefNum whichEdition,
                             Boolean successful);
```

Displaying Dialog Boxes

```
pascal OSErr GetLastEditionContainerUsed
                            (EditionContainerSpec *container);
pascal OSErr NewSubscriberDialog
                            (NewSubscriberReply *reply);
pascal OSErr NewPublisherDialog
                            (NewPublisherReply *reply);
pascal OSErr SectionOptionsDialog
                            (SectionOptionsReply *reply);
pascal OSErr NewSubscriberExpDialog
                            (NewSubscriberReply *reply, Point where,
                             short expansionDITLresID,
                             ExpDlgHookProcPtr dlgHook,
                             ExpModalFilterProcPtr filterProc,
                             void *yourDataPtr);
pascal OSErr NewPublisherExpDialog
                            (NewPublisherReply *reply, Point where,
                             short expansionDITLresID,
                             ExpDlgHookProcPtr dlgHook,
                             ExpModalFilterProcPtr filterProc,
                             void *yourDataPtr);
pascal OSErr SectionOptionsExpDialog
                            (SectionOptionsReply *reply, Point where,
                             short expansionDITLresID,
                             ExpDlgHookProcPtr dlgHook,
                             ExpModalFilterProcPtr filterProc,
                             void *yourDataPtr);
```

Locating a Publisher and Edition From a Subscriber

```
pascal OSErr GetEditionInfo (const SectionHandle sectionH,
                             EditionInfoRecord *editionInfo);
pascal OSErr GoToPublisherSection
                            (const EditionContainerSpec *container);
```

Edition Container Formats

```
pascal OSErr GetStandardFormats
                            (const EditionContainerSpec *container,
                             FormatType *previewFormat,
                             Handle preview, Handle publisherAlias,
                             Handle formats);
```

Reading and Writing Non-Edition files

```
pascal OSErr GetEditionOpenerProc
                            (EditionOpenerProcPtr *opener);
pascal OSErr SetEditionOpenerProc
                            (EditionOpenerProcPtr opener);
pascal OSErr CallEditionOpenerProc
                            (EditionOpenerVerb selector,
                             EditionOpenerParamBlock *PB,
                             EditionOpenerProcPtr routine);
pascal OSErr CallFormatIOProc
                            (FormatIOVerb selector,
                             FormatIOParamBlock *PB,
                             FormatIOProcPtr routine);
```

Application-Defined Routines

```
pascal OSErr MyExpDlgHook   (short itemOffset, short itemHit,
                             DialogPtr theDialog,
                             Ptr yourDataPtr);
pascal OSErr MyExpModalFilter
                            (DialogPtr theDialog,
                             EventRecord *theEvent,
                             short itemOffset, short *itemHit,
                             Ptr yourDataPtr);
pascal OSErr MyOpener       (EditionOpenerVerb selector,
                             EditionOpenerParamBlock *PB);
pascal OSErr MyIO           (FormatIOVerb selector,
                             FormatIOParamBlock *PB);
```

Result Codes

noErr	0	No error
abortErr	–27	Publisher has written a new edition
dskFulErr	–34	Disk is full
nsvErr	–35	No such volume
ioErr	–36	I/O error
bdNamErr	–37	Bad filename
fnOpnErr	–38	File not open
eofErr	–39	No additional data in the format
fnfErr	–43	Edition container not found
flLckedErr	–45	Publisher writing to an edition
fBsyErr	–47	Section doing I/O
paramErr	–50	Invalid parameter
rfNumErr	–51	Bad edition reference number
permErr	–54	Not a subscriber

wrPermErr	–61	Not a publisher
noTypeErr	–102	Format not available
memFullErr	–108	Memory full
dirNFErr	–120	Directory not found
userCanceledErr	–128	User clicked Cancel in dialog box
editionMgrInitErr	–450	Manager not initialized or could not load package
badSectionErr	–451	Not a valid section type
notRegisteredSectionErr	–452	Not registered
badSubPartErr	–454	Bad edition container spec or invalid edition container
multiplePublisherWrn	–460	Already is a publisher
containerNotFoundWrn	–461	Alias was not resolved
notThePublisherWrn	–463	Not the publisher