Data Access Manager

This chapter describes how your application can use the Data Access Manager to gain access to data in another application. It also tells you how to provide templates to be used for data transactions.

The Data Access Manager is available in System 7 and later versions. Use the Gestalt Manager to determine whether the Data Access Manager is present. To determine whether the Data Access Manager is available, use the `Gestalt` function with the `gestaltDBAccessMgrAttr` environmental selector. If the Data Access Manager is not available, the `Gestalt` function returns an error. For more information on the Gestalt Manager, see *Inside Macintosh: Operating System Utilities*.

The Data Access Manager allows your application to communicate with a database or other data source even if you do not know anything about databases in general or the specific data source with which the users of your software will be communicating. All your application needs are a few high-level Data Access Manager functions and access to a file called a ***query document.*** The query document, provided by another application, contains commands and data in the format appropriate for the database or other data source. The string of commands and data sent to the data source are referred to as a ***query.*** Note that a query does not necessarily extract data from a data source; it might only send data or commands to a database or other application.

The Data Access Manager makes it easy for your application to communicate with data sources. You need only add a menu item that opens a query document, using a few standard Data Access Manager functions to implement the menu selection. Users of your application can then access a database or other data source whenever they have the appropriate query documents. A user of a word-processing program might use this feature, for example, to obtain access to archived material, dictionaries in a variety of languages, or a database of famous quotations. A user of a spreadsheet program might use a query document to obtain tax records, actuarial tables, or other data. A user of an art or computer-aided design program might download archived illustrations or designs. And for the user of a database application for the Macintosh computer, the Data Access Manager can provide the resources and power of a mainframe database.

The Data Access Manager also provides a low-level interface for use by applications that are capable of creating their own queries and that therefore do not have to use query documents.

If your application uses only the high-level interface and relies on query documents created by other programs, then all the routines you need to know are described in this chapter. However, if you want to create a query document or an application that uses the low-level interface, then you must also be familiar with the command language used by the data server.

You need the information in this chapter if you want your application to access data in other applications or if you want to write a query document.

Note
The Data Access Manager makes it easy for your application to communicate with a database running on a remote computer, and this chapter generally assumes that you are using it for that purpose. However, there is no reason why the database could not be local—that is, running on the same computer as your application. To implement such a system, you would have to have a database that runs on a Macintosh computer and that has a command-language interface, plus a database extension that can use that command language. In most cases, it would be much simpler to run the database as a separate application and use the Clipboard to transfer data into and out of the database. ◆

Note also that the program containing the data need not be a database. With the appropriate database extension, your application could read data from a spreadsheet, for example, or any other program that stores data.

Apple Computer, Inc. provides a database extension that uses Data Access Language (DAL). A ***database extension*** provides an interface between the Data Access Manager and the database or other program that contains the data. If you want to write an application that uses the low-level interface to communicate with a Data Access Language server, or if you want to create a query document that uses Data Access Language, you must be familiar with that language. *Data Access Language Programmer's Reference*, available from APDA, fully describes this language.

About the Data Access Manager

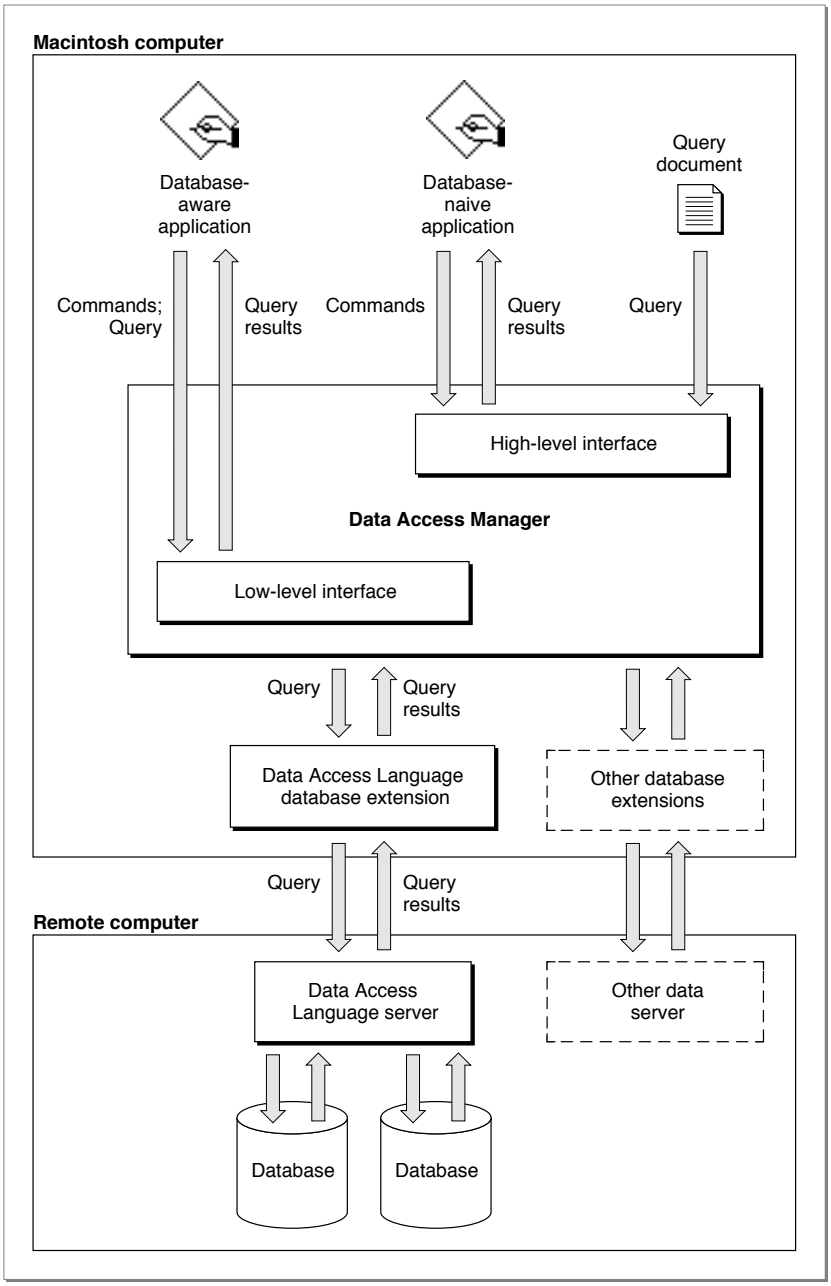
The Data Access Manager constitutes a standard interface that allows Macintosh applications to communicate with any number of databases or other data sources through a variety of data servers. As used in this chapter, a ***data server*** is the application that acts as an interface between the database extension on the Macintosh computer and the data source, which can be on the Macintosh computer or on a remote host computer. A data server can be a database server program, such as a Data Access Language server, which can provide an interface to a variety of different databases, or it can be the data source itself, such as a Macintosh application.

The Data Access Manager has two application interfaces: the high-level interface and the low-level interface. If the proper database extension and query documents are available in the user's system, you can use the high-level interface to communicate with a data source without having any knowledge of the command language that the data server uses. Even if you use the low-level interface, your application can isolate the user from any specific knowledge of the data source or the data server's command language.

This section presents an overview and description of the Data Access Manager, including diagrams and conceptual descriptions of the components and processes involved in using the high-level and low-level interfaces. Next, "Using the Data Access Manager" beginning on page 12-12 includes descriptions, flowcharts, and program fragments that provide a step-by-step guide to the use of the high-level and low-level interfaces. "Creating a Query Document" beginning on page 12-47 describes the contents and function of a query document. You do not have to read this section unless you are writing an application that creates query documents, although if you are using the high-level interface you might be interested to know just how a query document works.

Figure 12-1 illustrates connections between Macintosh applications and a database on a remote computer. The arrows in Figure 12-1 show the flow of information, not the paths of commands or control signals. See Figure 12-2 on page 12-8 and Figure 12-3 on page 12-10 for the sequences involved in sending and retrieving data.

Figure 12-1 A connection with a database



The High-Level Interface

As Figure 12-1 on page 12-6 shows, a database-naive application—that is, one that cannot prepare a query for a specific data server—uses the Data Access Manager's high-level routines to communicate with a data server. Because the application cannot prepare a query, it must use a query document to provide one. A query document can contain code, called a ***query definition function,*** that prompts the user for information and modifies the query before the Data Access Manager sends it to the data server. The exact format of a query definition function is described in "Writing a Query Definition Function" on page 12-52.

Note
The term *query* refers to any string of commands (and associated data) that can be executed by a data server. A query can send data to a data source, retrieve data from a data source, or reorganize the data in a data source. The Data Access Manager does not interpret or execute the query; it only implements the interface (sometimes called the *application program interface,* or API) that allows you to send the query to the data server. ◆

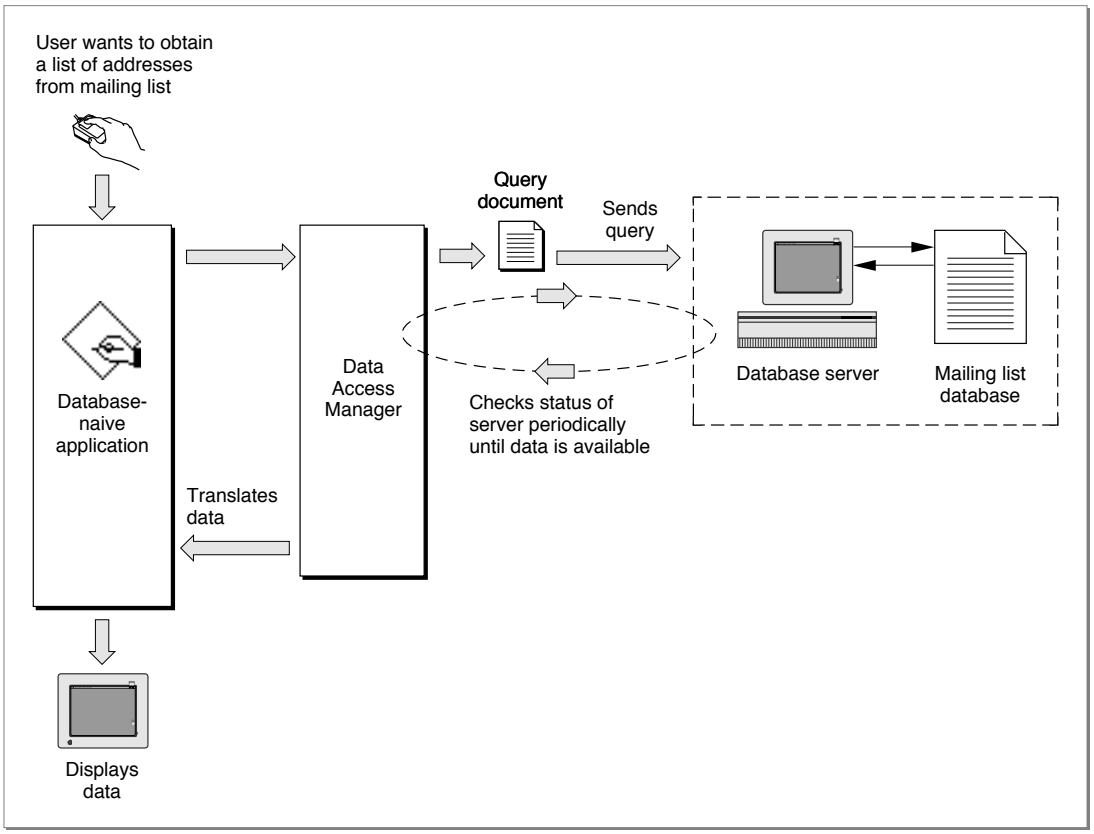
When you want to use the high-level routines to execute a query on a data server, you first select a query document or allow the user to select one. You use high-level routines to

- get the query from the query document
- execute the query definition function to modify the query
- send the query to the data server
- retrieve the results from any query that asks for information from the data source
- convert to text the results returned by a query

For example, suppose a company that makes rubber ducks has a database on a minicomputer that contains a mailing list of all its customers. The database has a Data Access Language interface, and the company's marketing manager has a Macintosh computer with an application that uses high-level Data Access Manager routines to communicate with the remote database server. As Figure 12-2 illustrates, the marketing manager must also have a query document, created by another application, that she can use to get an address from the mailing list on the remote minicomputer. The query document can be as complex or as simple as its creator cares to make it; in this example, the query document is designed specifically to obtain addresses from the rubber duck mailing list. The marketing manager might have several other query documents available as well: one to extract a mailing list for a specific zip code, one to list all of the customers who have made a purchase within the last year, and so on.

Notice that once the query document has sent the query to the data server, the Data Access Manager handles the data retrieval. Although query documents and high-level Data Access Manager routines make it very easy for you to *request* data from a data source, there is no way for a query document to verify that data *sent* to a data source has been successfully received. For that reason, it is recommended that you use the low-level interface to send data to a data source or update data in a data source.

Figure 12-2 Using high-level Data Access Manager routines



Sending a Query Through the High-Level Interface

To obtain a list of addresses from the mailing list, the marketing manager chooses the Open Query menu command from the File menu in her application. From the list of query documents displayed, she chooses one named Rubber Duck Address List.

The application calls the Data Access Manager function `DBGetNewQuery`, specifying the resource ID of the query (`'qrsc'`) resource in the Rubber Duck Address List query document. The `DBGetNewQuery` function creates a query record and a partial query from the information in the query resource. The partial query specifies the type of data (character strings) and the columns from which the data items should come (the name and address columns). The partial query lacks some specific data (the rows that should be searched) that is needed to complete the search criteria.

Next, the application calls the `DBStartQuery` function, which in turn calls the query definition function in the query document. The query definition function displays a dialog box that asks for the purchase dates to search. When the marketing manager types in the requested information and clicks OK, the query definition function adds the data to the partial query in memory. The query is now ready to be executed.

Next, the `DBStartQuery` function sends the query to the Data Access Language database extension, and the database extension sends the query over a communications network to the remote Data Access Language server. Finally, the `DBStartQuery` function commands the Data Access Language server to execute the query.

Retrieving Data Through the High-Level Interface

When the application is ready to retrieve the data that it requested from the database, the application calls the `DBGetQueryResults` function. This function determines when the data is available, retrieves it from the data server, and places the data in a record in memory. The application can then call the `DBResultsToText` function, which uses routines called ***result handlers*** to convert each data item to a character string. The `DBResultsToText` function passes to the application a handle to the converted data. The application then displays the list of customers for the marketing manager.

Data items and result handlers are described in "Processing Query Results" beginning on page 12-37.

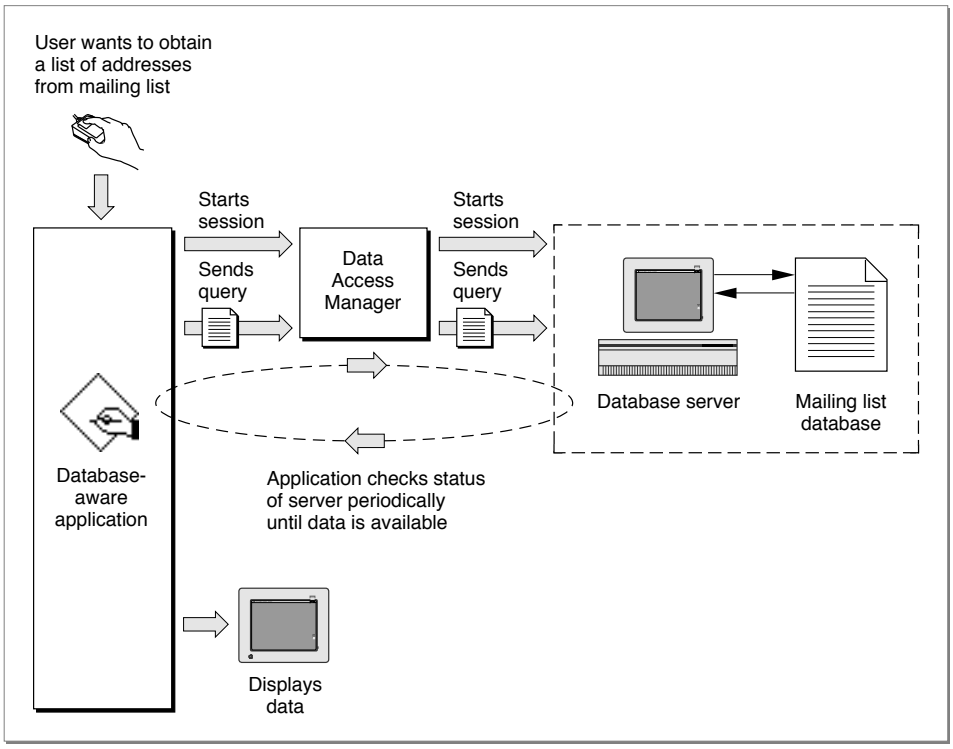
The Low-Level Interface

A database-aware application communicates through the low-level interface of the Data Access Manager. You can use the low-level interface to

- initiate communication with the data server, sending the user name, password, and other information to the data server
- send a query to the data server
- execute the query that you have sent to the data server
- halt execution of the query
- return status and errors from the data server
- send data to the data source
- retrieve data from the data source

For example, suppose once again that a company that makes rubber ducks has a mailing list of all of its customers in a database on a minicomputer, and the database has a Data Access Language interface. This time, suppose the Macintosh application the marketing manager is using calls low-level Data Access Manager routines to communicate with the remote database server. Figure 12-3 illustrates the use of the low-level interface. Notice that if you use the high-level interface (Figure 12-2), the query document and the Data Access Manager prepare the query, send the query, retrieve the query results, and translate the data for you. If you use the low-level interface, however, you must perform these functions yourself.

Figure 12-3 Using low-level Data Access Manager routines



Sending a Query Through the Low-Level Interface

To update the mailing list with a new address for customer Marvin M., the marketing manager enters the new address into her application. The application prepares a Data Access Language statement (a query) that specifies the type of data (a character string), the column into which the data item should go (the address column), the row to be modified (the Marvin M. row), plus the actual data the application wishes to send (Marvin M.'s address). The application then passes this query to the Data Access

Manager using the low-level interface. (The application can send the query in several pieces or all at once.) The Data Access Manager sends the query to the Data Access Language database extension in the Macintosh computer, and the database extension sends the query to the remote Data Access Language server.

Retrieving Data Through the Low-Level Interface

Once the query begins executing, the application can periodically check with the data server to determine whether the data is ready (Figure 12-3). When the data is available, the application must retrieve it one data item at a time. An application that uses the low-level interface must determine the data type of each data item, convert the data into a format that is meaningful to the user, and store the data in memory allocated by the application. Data types are described in "Getting Query Results" beginning on page 12-37.

Note that neither the Data Access Manager nor the DAL database extension reads, modifies, or acts on the query that an application sends to the data server. The data server does execute the query, causing the data source to accept new data or prepare data for the application. To use the low-level interface to communicate with a data server, your application must be capable of preparing a query that can be executed by the data server.

Comparison of the High-Level and Low-Level Interfaces

An application that uses the low-level interface to send a query to the data server must prepare the query, initiate communication with the data server, send the query to the data server, and execute the query. If it requested data to be returned, the application must determine when the data is ready, and retrieve the data one item at a time. Each step in this process requires calling one or more low-level routines.

The high-level interface between the Data Access Manager and the application, in contrast, consists of only a few routines, each of which might call several low-level routines to accomplish its tasks. For example, a single high-level function can call the query definition function, initiate communication with the data server, send the query to the data server, and execute the query.

Because the high-level interface is very easy to use and requires no specific knowledge of the data source or database server, you can add high-level data access to your application very easily. Then, whenever someone provides a query document for use with a specific data server, the user can take advantage of the data access capability included in your application. However, because there is no way for a query document to verify that data *sent* to a data source has been successfully received, it is recommended that you use the low-level interface to send data to a data source or update data in a data source.

Although in concept the low-level routines and high-level routines serve separate purposes, there is nothing to prevent you from using calls to both in a single application. For example, you might use low-level routines to send a query to a data server and high-level routines to read the results and convert them to text.

Using the Data Access Manager

There are at least three different ways in which you can use the Data Access Manager to communicate with a data source. You can

- use low-level interface routines to send queries and retrieve data from the data source. In this case, your application must be capable of preparing a query in a language appropriate for the data server.
- use high-level interface routines to send queries and retrieve data from the data source. In this case, you must have one or more query documents provided by another application.
- create your own query documents and use high-level interface routines to send queries and retrieve data from the data source. In this case, your application must be capable of preparing a query, but it can use the same query repeatedly once it has been prepared.

This section describes how to use the high-level and low-level interfaces to the Data Access Manager to send queries to a data server. This section also describes how to call Data Access Manager functions asynchronously, how to determine the status of the high-level functions at various points in their execution (and cancel execution if you so desire), how to obtain information about Data Access Manager sessions that are in progress, and how to retrieve query results and convert them to text.

Executing Routines Asynchronously

All of the Data Access Manager low-level routines and some of the high-level routines can execute asynchronously—that is, the routine returns control to your application before the routine has completed execution. Your application must call the Event Manager's `WaitNextEvent` function periodically to allow an asynchronous routine to complete execution.

Note
The database extension is responsible for implementing asynchronous execution of Data Access Manager routines. For example, if you call the `DBSend` function to send a query to a data server, and the database extension calls a device driver, the database extension can return control to your application as soon as the device driver has placed its routine in the driver input/output (I/O) queue. If you attempt to execute a routine asynchronously and the database extension that the user has selected does not support asynchronous execution, the routine returns a result code of `rcDBAsyncNotSupp` and terminates execution. ◆

All Data Access Manager routines that can execute asynchronously take as a parameter a pointer to a parameter block known as the *asynchronous parameter block.* If the value of this pointer is `NIL`, the function is executed synchronously—that is, the routine does not return control to your application until execution is complete.

General Guidelines for the User Interface

When you use the Data Access Manager to provide data access, you should keep two important principles in mind: keep the user in control, and provide feedback to the user.

Keep the User in Control

When designing a data access feature or application, keep in mind that the user should have as much access to the Macintosh computer's abilities as possible. Design your application so that most of the data access process happens in the background. Call the Data Access Manager asynchronously whenever the database extension you are using supports asynchronous calls. Because data retrieval queries can take minutes or even hours to complete, they should always run in the background.

After issuing a query, return control of the computer to users so that they may work on other tasks or switch to other applications while the query runs. Whenever a background task requires the user's attention, follow the suggestions in *Macintosh Human Interface Guidelines* regarding user notification. A background task should never take control from the user by posting an alert box in front of the active application's windows. Any message that you post should identify the query that requires attention. For example, an alert box might display the message "The query Get Employee Information was canceled because the connection was unexpectedly broken."

If your application allows more than one simultaneous connection to data sources or allows more than one query document to run, provide a modeless window that lists the open connections and queries, displays the status of each, and allows the user to cancel them if necessary.

Allow the user to limit the amount of disk space that must remain free after any transaction. For example, a user may wish to specify that 1 MB of space always be free. Cancel any transaction that would exceed the user's limit and notify the user.

Provide Feedback to the User

Keep the user informed about status, progress, and error conditions, and allow the user to cancel an interaction whenever possible. Inform the user before the application becomes modal and the computer becomes unavailable. Use the spinning beach ball cursor or the animated wristwatch cursor to indicate a process that takes several seconds to complete. Use a dialog box to indicate any process that lasts longer than a few seconds. For example, connecting to a remote database could take a couple of minutes. In this case include a Cancel button in the dialog box so that the user can cancel the operation. When possible, display a progress indicator to show how long a process lasts. Warn the user before doing anything potentially dangerous or irreversible, such as deleting all of a user's data files to replace them with data retrieved from a data source.

When a data retrieval query terminates prematurely, make the retrieved data available to the user but warn the user that it is incomplete. The user can then evaluate the partial data before deciding whether to run the query again.

Using the High-Level Interface

Use the high-level interface to the Data Access Manager if you want to use a query document to do the work of communicating with a data source. You can use the high-level interface to open a query document, execute the query definition function in the query document, establish communication (initiate a session) with a data server, send the query to the data server, execute the query, retrieve any data requested by the query, and convert the retrieved data to text. Although two or three high-level routines accomplish most of these tasks, you may need to call a few low-level routines as well to control a session with a data server.

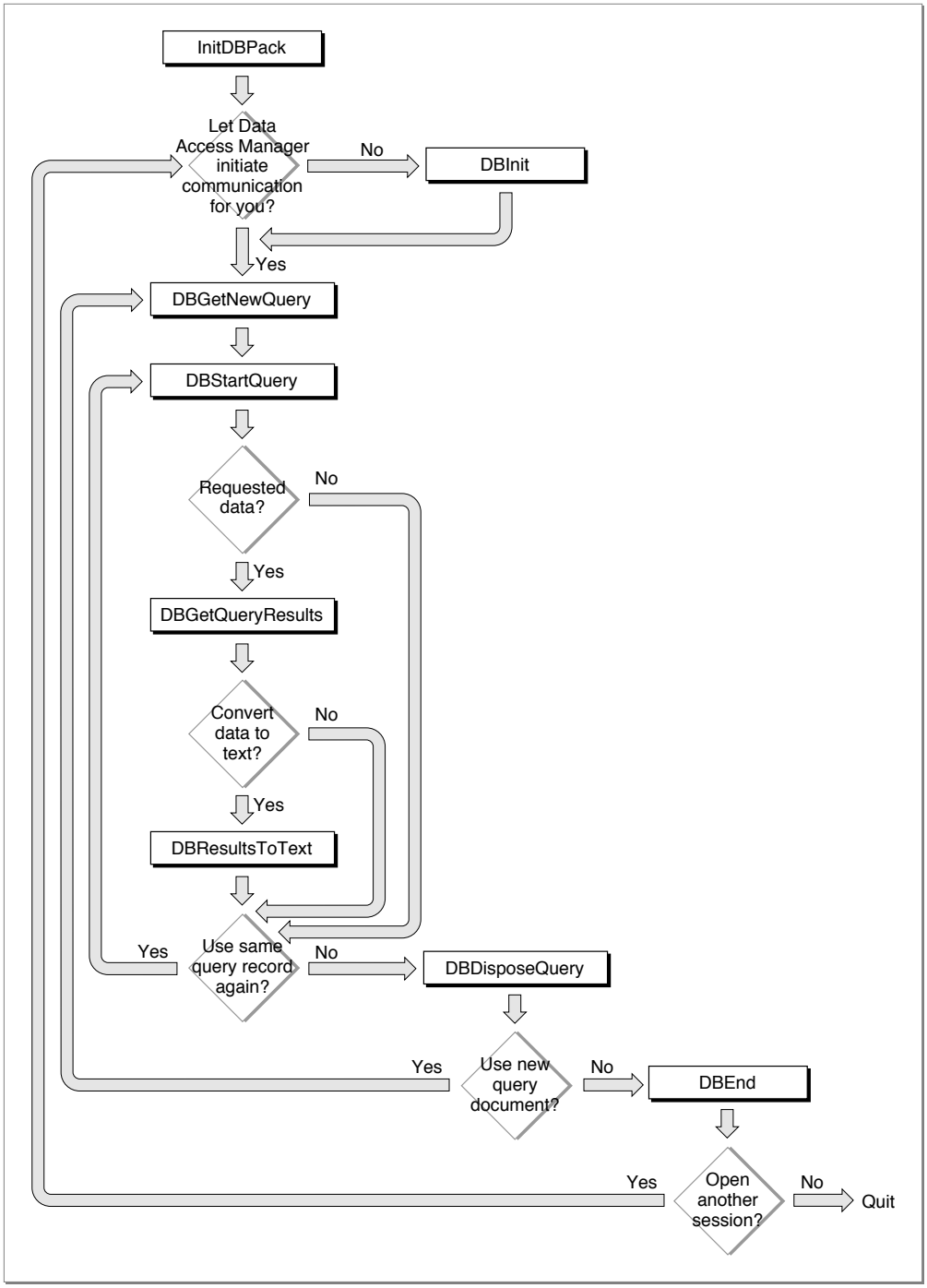
Applications that implement this type of data access must provide user control and feedback as described in "General Guidelines for the User Interface" on page 12-13. In addition, you should include an Open Query command in the File menu. The Open Query command is equivalent to the Open (file) command in meaning. When the user chooses this command, display an open file dialog box filtered to show only query documents (file type `'qery'`). The user can then select the desired query document. The query document contains the query to be sent to the data source. Depending on the type of query, the data source could receive information, send back information, report the status of the data source, or perform some other task.

Figure 12-4 is a flowchart of a typical session using the high-level interface.
As Figure 12-4 illustrates, you must follow this procedure to use the high-level interface:

1. Call the `InitDBPack` function to initialize the Data Access Manager.
2. Select the query document that you want to use and determine the resource ID of the `'qrsc'` resource in that query document. You can use any method you like to select the query document. One possibility is to use the `StandardGetFile` procedure to let the user select the query document. A query document should contain only one `'qrsc'` resource; you can then use the Resource Manager to determine the resource ID of the `'qrsc'` resource in the document that the user selected. For further information, see the description of the `StandardGetFile` procedure in the chapter "Standard File Package" in *Inside Macintosh: Files* and the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox*.
3. Call the `DBGetNewQuery` function. The `DBGetNewQuery` function creates in memory a data structure called a *query record* from the `'qrsc'` resource that you specify.
4. Call the `DBStartQuery` function specifying the handle to the query record that you created with the `DBGetNewQuery` function (step 3).

   You should also provide the `DBStartQuery` function with a handle to your status routine. A ***status routine*** is a routine that you provide to update windows, check the results of the low-level calls made by the `DBStartQuery` and `DBGetQueryResults` functions, and cancel execution of these functions when you consider it appropriate to do so.

Figure 12-4 A flowchart of a session using the high-level interface



The DBStartQuery function calls the query definition function (if any) referred to by the query record. The query definition function can prompt the user for information and modify the query record.

After the query definition function has completed execution, the DBStartQuery function calls your status routine so that you can update your windows if necessary. The DBStartQuery function then checks whether communication has been established with the data server. If not, it calls your status routine so that you can display a status dialog box and then calls the DBInit function to establish communication (initiate a session) with the data server. The DBStartQuery function obtains the values it needs for the DBInit function parameters from the query record. When the DBInit function completes execution, the DBStartQuery function calls your status routine again.

The DBInit function returns an identification number, called a ***session ID.*** This session ID is unique; no other current session, for any database extension, has the same session ID. You must specify the session ID any time you want to send data to or retrieve data from this session. If you prefer, you can use the DBInit function to establish communication before you call the DBStartQuery function. In that case, you must specify the session ID as an input parameter to the DBStartQuery function. See "Using the Low-Level Interface" beginning on page 12-28 for more information on using the DBInit function.

Once communication has been established, the DBStartQuery function calls the DBSend function to send the data server the query specified by the query record. When the DBSend function has completed execution, the DBStartQuery function calls your status routine. Finally, the DBStartQuery function uses the DBExec function to execute the query. The DBStartQuery function calls your status routine after the DBExec function has completed execution (that is, the query has started executing and the DBExec function has returned control to the DBStartQuery function) and again just before the DBStartQuery function completes execution.

5. If you requested data and want to know when the data is available, but do not want to retrieve the data immediately, you can call the DBState function. This function tells you when the data server has finished executing the query, but it does not retrieve the data. If you requested data and want to retrieve it as soon as it is available, you do not have to call the DBState function; go to step 6 instead.

   If you did not request data, you can use the DBState function to determine the status of the query. When the data server has finished executing the query, skip to step 8.

6. Call the DBGetQueryResults function. If the query has not finished executing, this function returns the rcDBExec result code. If the query has finished executing, the DBGetQueryResults function calls the DBGetItem function repeatedly until the data server has returned all of the data available.

   The DBGetQueryResults function puts the returned data into a record that contains handles to arrays that contain the data, the type of data in each column, and the length of each data item. The Data Access Manager allocates the memory for this data in the application heap.

The `DBGetQueryResults` function calls your status routine after it retrieves each data item. You can use this opportunity to display the data item for the user and to give the user the opportunity to cancel execution of the function. The `DBGetQueryResults` function also calls your status routine just before completing execution, so that you can dispose of any memory allocated by the status routine, remove any dialog box that you displayed, and update your windows if necessary.

To convert the returned data to text, go to the next step. If you do not want to convert the returned data to text, skip to step 9.

7. Call the `DBResultsToText` function. This function calls a result handler function for each data type. The result handler converts the data to text, places it in a buffer, and returns a handle to the buffer. Some result handlers are provided with the Data Access Manager; you can provide as many with your application as you wish. Result handlers are discussed in "Converting Query Results to Text" beginning on page 12-43.
8. If you are finished using the query record, call the `DBDisposeQuery` function to dispose of the query record and free all the memory associated with the query record. If you want to reuse the same query, return to step 5. You should close the query document when you are finished using it.

   If you want to use a new query document, return to step 3.
9. When you are finished using the data source, you must use the `DBEnd` function to terminate the session. You must call the `DBEnd` function after the `DBInit` function has returned a nonzero session ID, even if it also returned an error.

Listing 12-1 illustrates the use of the high-level interface. This code initiates a session with a remote database, lets the user select a query document to execute, opens the selected file, finds a `'qsrc'` resource, and creates a query record. Next, it executes the query, checks the status of the remote database server, retrieves the data when it's available, and converts this data to text. When the query has finished executing, the code disposes of the query record, ends the session, and closes the user-selected query document. In general, there's no reason why there can't be multiple sessions open at once. You can identify each session by its session ID. Listing 12-1 shows just one session.

Listing 12-1 assumes that you are using a database extension that supports asynchronous execution of Data Access Manager routines. This listing shows just one possible approach to sending a query and retrieving data asynchronously.

Listing 12-1 Using the high-level interface

```
PROCEDURE MyHiLevel(VAR rr: ResultsRecord; myTextHdl: Handle;
                  VAR thisSession: LongInt; VAR sessErr: OSErr);
TYPE
   {define a record to include space for the current value in }
   { A5 so a completion routine can find it}
   CRRec = RECORD
      QPB: DBAsyncParamBlockRec;       {the parameter block}
      appsA5: LongInt;                 {append A5 to the }
                                       { parameter block}
   END;
   CRRecPtr = ^CRRec;

VAR
   StartPB, GetQRPB:                              CRRec;
   SFR:                                           StandardFileReply;
   packErr, startQErr, getQErr, disposeQErr:      OSErr;
   getnewQErr, gStartQErr, gGetQRErr:             OSErr;
   endErr, fsopenErr, fscloseErr, resultsErr:     OSErr;
   gStart, gQueryResults:                         Boolean;
   qrscHandle:                                    Handle;
   rsrcID:                                        Integer;
   rsrcType:                                      ResType;
   rsrcName:                                      Str255;
   myQHandle:                                     QueryHandle;
   savedResFile:                                  Integer;
   typeList:                                      SFTypeList;
   fsRefNum:                                      Integer;

FUNCTION GetQPB: CRRecPtr;
   INLINE $2E88;                 {MOVE.L  A0,(SP)}

BEGIN
   gStart := FALSE;
   gQueryResults := FALSE;
   sessErr := noErr;             {assume everything went fine}
   packErr := InitDBPack;        {initialize the Data Access Mgr}
   {display a dialog box to let the user pick a query document}
   typeList[0] := 'qery';
   StandardGetFile(NIL, 1, typeList, SFR);
   IF SFR.sfGood = TRUE THEN
      fsopenErr := FSpOpenRF(SFR.sfFile, fsCurPerm, fsRefNum);
   IF (fsopenErr <> noErr) OR (SFR.sfGood = FALSE) THEN
```

```
BEGIN
   sessErr := fsopenErrOrUserCanceled;
   EXIT(MyHiLevel);
END;
savedResFile := CurResFile;   {save current resource file}
UseResFile(fsRefNum);         {get query info from here}
{a query document should have only one 'qrsc' resource}
qrscHandle := Get1IndResource('qrsc', 1);
IF ResError <> noErr THEN
BEGIN
   sessErr := ResError;
   EXIT(MyHiLevel);
END;
{get the resource ID of the 'qrsc' resource}
GetResInfo(qrscHandle, rsrcID, rsrcType, rsrcName);
{create a query record using the resource ID}
getnewQErr := DBGetNewQuery(rsrcID, myQHandle);
IF getnewQErr <> noErr THEN
BEGIN
   sessErr := getnewQErr;
   endErr := DBEnd(thisSession, NIL);
   EXIT(MyHiLevel);
END;
StartPB.QPB.completionProc := @MyStartCompRoutine;
StartPB.appsA5 := SetCurrentA5;     {save this for the }
                                    { completion routine}
{MyStartStatus is a status routine that handles messages sent }
{ by the DBStartQuery function when it calls a low-level }
{ function}
startQErr := DBStartQuery(thisSession, myQHandle,
                          @MyStartStatus, @StartPB);
IF startQErr <> noErr THEN
BEGIN
   sessErr := startQErr;
   IF thisSession <> 0 THEN
      endErr := DBEnd(thisSession, NIL);
   EXIT(MyHiLevel);
END;
WHILE NOT gStart DO  {while waiting for gStart to go TRUE, }
BEGIN                { MyGoDoSomething calls WaitNextEvent }
   MyGoDoSomething;  { to give other routines a chance to run}
END;     {while}
{the DBStartQuery call has completed}
```

```
IF gStartQErr <> noErr THEN
BEGIN
   sessErr := gStartQErr;
   IF thisSession <> 0 THEN
      endErr := DBEnd(thisSession, NIL);
   EXIT(MyHiLevel);
END;
GetQRPB.QPB.completionProc := @MyGetQRCompRoutine;
GetQRPB.appsA5 := SetCurrentA5;        {save this for the }
                                       { completion routine}
{MyGetQRStatus is a status routine that handles messages sent }
{ by the DBGetQueryResults function when it calls a low-level }
{ function.}
getQErr := DBGetQueryResults(thisSession, rr, kDBWaitForever,
                             @MyGetQRStatus, @GetQRPB);
IF getQErr <> noErr THEN
BEGIN
   sessErr := getQErr;
   endErr := DBEnd(thisSession, NIL);
   EXIT(MyHiLevel);
END;
WHILE NOT gQueryResults DO
BEGIN
   MyGoDoSomething;
END; {while}
{The DBGetQueryResults call has completed. Assuming the call }
{ completed successfully, you may want to convert the }
{ retrieved data to text, return memory you have borrowed, }
{ and end the session.}
IF gGetQRErr <> noErr THEN
BEGIN
   sessErr := gGetQRErr;
   endErr := DBEnd(thisSession, NIL);
   EXIT(MyHiLevel);
END;
{the data has been retrieved; convert it to text}
resultsErr := DBResultsToText(rr, myTextHdl);
{The current query is finished.  You can elect to execute }
{ the next 'qrsc' resource of the file, or select another }
{ query document.  This example just returns to the caller.}
disposeQErr := DBDisposeQuery(myQHandle);
UseResFile(savedResFile);{restore current resource file}
fscloseErr := FSClose(fsRefNum); {close the query document}
```

```
   IF fscloseErr <> noErr THEN
      DoError(fscloseErr);
   endErr := DBEnd(thisSession, NIL);
   IF endErr <> noErr THEN
      DoError(endErr);
   END;
```

Listing 12-2 shows the completion routines `MyStartCompRoutine` and `MyGetQRCompRoutine` used in Listing 12-1.

Listing 12-2 Two completion routines

```
PROCEDURE MyStartCompRoutine(aCRRecPtr: CRRecPtr);
VAR
   curA5:   LongInt;
BEGIN
   aCRRecPtr := GetQPB;                {get the param block}
   curA5 := SetA5(aCRRecPtr^.appsA5);  {set A5 to the app's A5}
   gStart := TRUE;                     {query has been started}
   gStartQErr := aCRRecPtr^.QPB.result;{send back result code}
   {do whatever else you want to do}
   curA5 := SetA5(curA5);{restore original A5}
END;  {MyStartCompRoutine}

PROCEDURE MyGetQRCompRoutine(aCRRecPtr: CRRecPtr);
VAR
   curA5:   LongInt;
BEGIN
   aCRRecPtr := GetQPB;                {get the param block}
   curA5 := SetA5(aCRRecPtr^.appsA5);  {set A5 to the app's A5}
   gQueryResults := TRUE;{query results are complete}
   gGetQRErr := aCRRecPtr^.QPB.result; {send back the result code}
   {do whatever else you want to do}
   curA5 := SetA5(curA5);              {restore original A5}
END;  {MyGetQRCompRoutine}
```

The next section provides information about status routines.

Writing a Status Routine for High-Level Functions

Both of the two main high-level functions, `DBStartQuery` and `DBGetQueryResults`, call low-level functions repeatedly. After each time they call a low-level function, these high-level functions call a routine that you provide, called a *status routine.* Your status routine can check the result code returned by the low-level function and can cancel execution of the high-level function before it calls the next low-level function. Your status routine can also update your application's windows after the `DBStartQuery` function has displayed a dialog box.

You provide a pointer to your status routine in the `statusProc` parameter to the `DBStartQuery` and `DBGetQueryResults` functions.

Here is a function declaration for a status routine:

```
FUNCTION MyStatusFunc (message: Integer; result: OSErr;
                       dataLen: Integer; dataPlaces: Integer;
                       dataFlags: Integer; dataType: DBType;
                       dataPtr: Ptr): Boolean;
```

Your status routine should return a value of `TRUE` if you want to continue execution of the `DBStartQuery` or `DBGetQueryResults` function, or a value of `FALSE` if you want to cancel execution of the function. In the latter case, the high-level function returns the `userCanceledErr` result code.

Note
If you call the `DBStartQuery` or `DBGetQueryResults` function asynchronously, you cannot depend on the A5 register containing a pointer to your application's global variables when the Data Access Manager calls your status routine. ◆

The `message` parameter tells your status routine the current status of the high-level function that called it. The possible values for the `message` parameter depend on which function called your routine.

The value of the `result` parameter depends on the value of the `message` parameter, as summarized in the following list:

Message	Result
`kDBUpdateWind`	0
`kDBAboutToInit`	0
`kDBInitComplete`	Result of `DBInit`
`kDBSendComplete`	Result of `DBSend`
`kDBExecComplete`	Result of `DBExec`
`kDBStartQueryComplete`	Result of `DBStartQuery`
`kDBGetItemComplete`	Result of `DBGetItem`
`kDBGetQueryResultsComplete`	Result of `DBGetQueryResults`

The dataLen, dataPlaces, dataFlags, dataType, and dataPtr parameters are returned only by the DBGetQueryResults function, and only when the message parameter equals kDBGetItemComplete. When the DBGetQueryResults function calls your status routine with this message, the dataLen, dataPlaces, and dataType parameters contain the length, decimal places, and type of the data item retrieved, respectively, and the dataPtr parameter contains a pointer to the data item.

The least significant bit of the dataFlags parameter is set to 1 if the data item is in the last column of the row. The third bit of the dataFlags parameter is set to 1 if the data item is NULL. You can use this information, for example, to check whether the data meets some criteria of interest to the user, or to display each data item as the DBGetItem function receives it. You can use the constants kDBLastColFlag and kDBNullFlag to test for these flag bits.

The DBGetQueryResults function returns a results record, which contains a handle to the retrieved data. The address in the dataPtr parameter points inside the array specified by this handle. Because the dataPtr parameter is not a pointer to a block of memory allocated by the Memory Manager, but just a pointer to a location inside such a block, you cannot use this pointer in any Memory Manager routines (such as the GetPtrSize function). Note also that you cannot rely on this pointer remaining valid after you return control to the DBGetQueryResults function.

The DBStartQuery function can send to your status routine the following constants in the message parameter:

```
CONST {DBStartQuery status messages}
      kDBUpdateWind           = 0;     {update windows}
      kDBAboutToInit          = 1;     {about to call DBInit}
      kDBInitComplete         = 2;     {DBInit has completed}
      kDBSendComplete         = 3;     {DBSend has completed}
      kDBExecComplete         = 4;     {DBExec has completed}
      kDBStartQueryComplete   = 5;     {DBStartQuery is about to }
                                       { complete}
```

DBStartQuery message constant	Meaning
kDBUpdateWind	The DBStartQuery function has just called a query definition function. Your status routine should process any update events that your application has received for its windows.
kDBAboutToInit	The DBStartQuery function is about to call the DBInit function to initiate a session with a data server. Because initiating the session might involve establishing communication over a network, and because in some circumstances the execution of a query can tie up the user's computer for some length of time, you might want to display a dialog box giving the user the option of canceling execution at this time.

DBStartQuery message constant	Meaning (continued)

continued

kDBInitComplete	The DBInit function has completed execution. When the DBStartQuery function calls your status routine with this message, the result parameter contains the result code returned by the DBInit function. If the DBInit function returns the noErr result code, the DBStartQuery function calls the DBSend function next. If the DBInit function returns any other result code, you can display a dialog box informing the user of the problem before returning control to the DBStartQuery function. The DBStartQuery function then returns an error code and stops execution.
kDBSendComplete	The DBSend function has completed execution. When the DBStartQuery function calls your status routine with this message, the result parameter contains the result code returned by the DBSend function. If the DBSend function returns the noErr result code, the DBStartQuery function calls the DBExec function next. If the DBSend function returns any other result code, you can display a dialog box informing the user of the problem before returning control to the DBStartQuery function. The DBStartQuery function then returns an error code and stops execution.
kDBExecComplete	The DBExec function has completed execution. When the DBStartQuery function calls your status routine with this message, the result parameter contains the result code returned by the DBExec function. If the DBExec function returns the noErr result code, the DBStartQuery function returns control to your application next. If the DBExec function returns any other result code, you can display a dialog box informing the user of the problem before returning control to the DBStartQuery function. The DBStartQuery function then returns an error code and stops execution.
kDBStartQueryComplete	The DBStartQuery function has completed execution and is about to return control to your application. The function result is in the result parameter passed to your status routine. Your status routine can use this opportunity to perform any final tasks, such as disposing of memory that it allocated or removing from the screen any dialog box that it displayed.

The DBGetQueryResults function can send to your status routine the following constants in the message parameter:

```
CONST {DBGetQueryResults status messages}
      kDBGetItemComplete         = 6;  {DBGetItem has completed}
      kDBGetQueryResultsComplete = 7;  {DBGetQueryResults has }
                                       { completed}
```

DBGetQueryResults message constant	Meaning
kDBGetItemComplete	The DBGetItem function has completed execution. When the DBGetQueryResults function calls your status routine with this message, the result parameter contains the result code returned by the DBGetItem function. The DBGetQueryResults function also returns values for the dataLen, dataPlaces, dataType, dataFlags, anddataPtr parameters, as discussed earlier in this section. For each data item that it retrieves, the DBGetQueryResults function calls the DBGetItem function twice: once to obtain information about the next data item and once to retrieve the data item. The DBGetQueryResults function calls your status routine only after calling the DBGetItem function to retrieve a data item. If your status routine returns a function result of FALSE in response to the kDBGetItemComplete message, the results record returned by the DBGetQueryResults function to your application contains data through the last full row retrieved. Data types and results records are described in "Getting Query Results" beginning on page 12-37.
kDBGetQueryResultsComplete	The DBGetQueryResults function has completed execution and is about to return control to your application. The function result is in the result parameter passed to your status routine. Your status routine can use this opportunity to perform any final tasks, such as disposing of memory that it allocated or removing from the screen any dialog box that it displayed.

Listing 12-3 shows a status routine for the DBStartQuery function. This routine updates the application's windows in response to the kDBUpdateWind message,

displays a dialog box giving the user the option of canceling before the data access is initiated, and checks the results of calls to the `DBInit`, `DBSend`, and `DBExec` functions. If one of these functions returns an error, the status routine displays a dialog box describing the error.

Listing 12-3 A sample status routine

```
FUNCTION MyStartStatus(message: Integer; result: OSErr;
                       dataLen: Integer; dataPlaces: Integer;
                       dataFlags: Integer; dataType: DBType;
                       dataPtr: Ptr): Boolean;
VAR
   myString:   Str255;
   continue:   Boolean;
BEGIN
   continue := TRUE; {assume user wants to continue with query}
   CASE message OF
   kDBUpdateWind:    {a qdef function has just been called; }
      BEGIN          { handle activate and update events}
         MyDoActivate;  {find and handle activate events}
         MyDoUpdate;    {find and handle update events}
      END;  {kDBUpdateWind}
   kDBAboutToInit:   {about to initiate a session}
      BEGIN {MyDisplayDialog displays a dialog box. The value }
            { returned in the continue variable indicates }
            { whether DBStartQuery should continue.}
         myString := 'The Data Access Manager is about to open a
                     session.  This could take a while.  Do you
                     want to continue?';
         MyDisplayDialog(@myString, continue);
      END;  {kDBAboutToInit}
   kDBInitComplete:  {the DBInit function has completed execution}
      BEGIN
         IF result <> noErr THEN {if there's an error, }
         BEGIN                   { let the user know what it is}
            CASE result OF
            rcDBError:
            BEGIN
               myString := 'The Data Access Manager was unable to
                           open the session.  Please check your
                           connections and try again later.';
               MyDisplayString(@myString);
            END;  {rcDBError}
```

```
            rcDBBadDDev:
            BEGIN
               myString := 'The Data Access Manager cannot find
                           the database extension file it needs to
                           open a session.  Check with your system
                           administrator for a copy of the file.';
               MyDisplayString(@myString);
            END;  {rcDBBadDDev}
            OTHERWISE
            BEGIN
               myString := 'The Data Access Manager was unable to
                           open the session.  The error code
                           returned was';
               MyDisplayError(@myString, result);
            END; {of otherwise}
         END;  {of CASE result}
         END; {of result <> noErr}
      END; {kDBInitComplete}
   kDBSendComplete:  {the DBSend function has completed execution}
      BEGIN
         {if there's an error, let the user know what it is}
         IF result <> noErr THEN
         BEGIN
            IF result = rcDBError THEN
            BEGIN
               myString := 'An error occurred while the Data
                           Access Manager was trying to send the
                           query. Please try again later.';
               MyDisplayString(@myString);
            END
            ELSE
            BEGIN
               myString := 'An error occurred while the Data
                           Access Manager was trying to send the
                           query. The error code returned was';
               MyDisplayError(@myString, result);
            END;
         END;  {of result <> noErr}
      END;  {kDBSendComplete}
   kDBExecComplete:  {the DBExec function has completed execution}
      BEGIN
         IF result <> noErr THEN {if there's an error, }
         BEGIN                   { let the user know what it is}
```

```
            IF result = rcDBError THEN
            BEGIN
               myString := 'The Data Access Manager was unable to
                        execute the query. There may be a problem
                        with the query document or the database.
                        Check with your system administrator.';
               MyDisplayString(@myString);
            END
            ELSE
            BEGIN
               myString := 'An error occurred while the Data
                        Access Manager was trying to execute the
                        query.  The error code returned was';
               MyDisplayError(@myString, result);
            END;
         END;  {of result <> noErr}
      END;  {kDBExecComplete}
   kDBStartQueryComplete:{the DBStartQuery function is about }
      BEGIN             { to return control to your application}
         {clean up memory and any dialog boxes left on the screen}
         MyCleanUpWindows;
      END;  {kDBStartQueryComplete}
   END; {CASE message}
   MyStartStatus := continue;
END;
```

Using the Low-Level Interface

You can use the low-level interface to establish communication (initiate a session) with a data server, send a query to the data server, execute the query, and retrieve any data requested by the query. You call one or more low-level routines to accomplish each of these tasks.

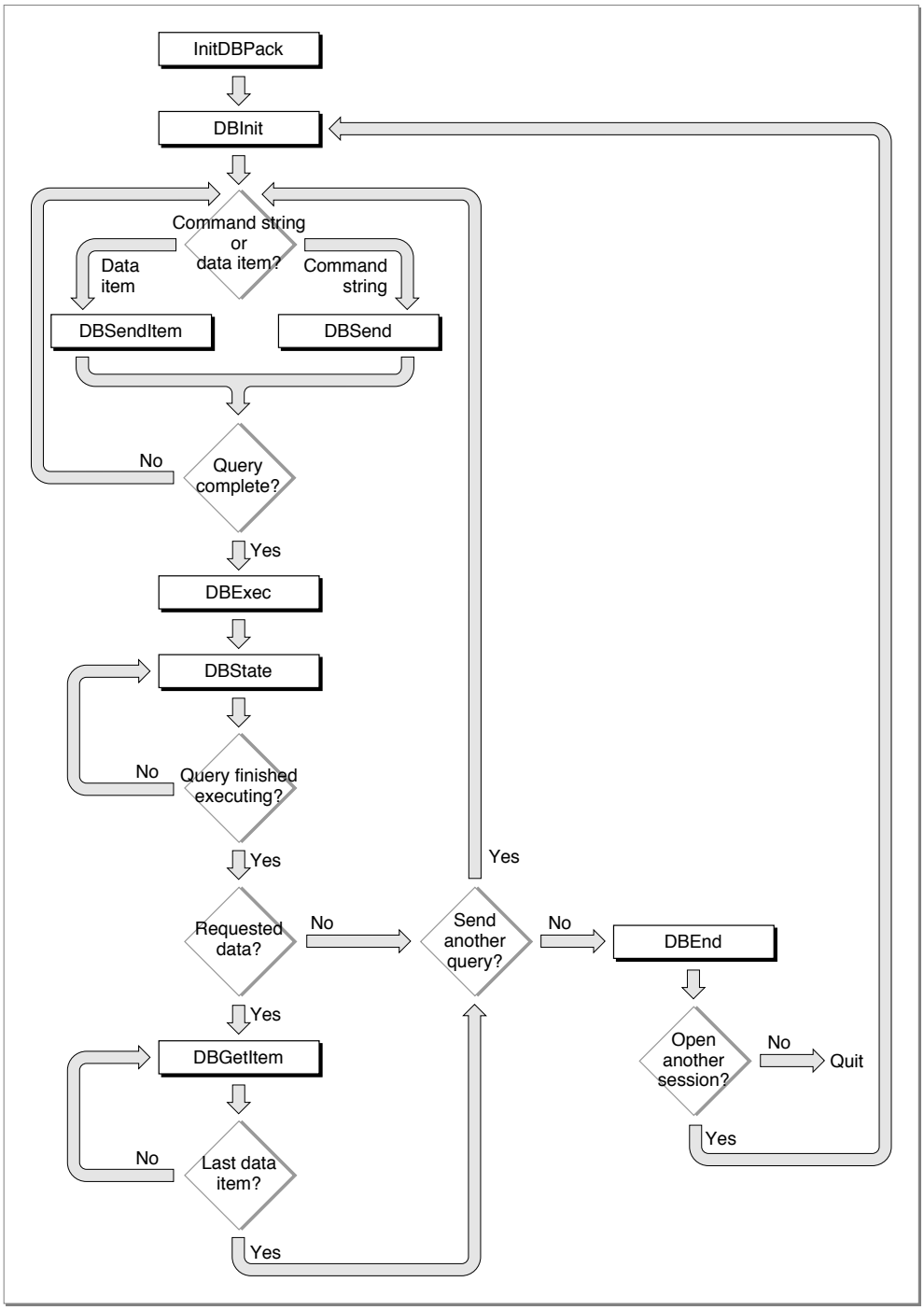
Applications that implement this type of data access must provide user control and feedback, as described in "General Guidelines for the User Interface" on page 12-13. When the data source is ready to return data, you can retrieve it all and then display it to the user, or you can display the data as it arrives. If the data arrives slowly, it's best to display it one record at a time as it arrives. This way the user can preview the data, decide if it's the desired information, and cancel the query if not.

Figure 12-5 is a flowchart of a typical session using the low-level interface. As Figure 12-5 illustrates, you must follow this procedure to use the low-level interface:

1. Call the `InitDBPack` function to initialize the Data Access Manager.
2. Call the `DBInit` function to establish communication with the data server. The `DBInit` function returns an identification number, called a *session ID*. This session ID

is unique; no other current session, for any database extension, has the same session ID.

Figure 12-5 A flowchart of a session using the low-level interface



You must specify the session ID any time you want to send data to or retrieve data from this session.

The `DBInit` function requires as input parameters the name of the database extension and character strings for the host system, user name, password, and connection string. All of these parameters depend on the user and the user's computer system, including the specific database extension, host computer, data server, and database management software in use. You will not know the user name and password when you are writing an application, and you might not know the values of any of these parameters. Therefore, you must display a dialog box that prompts the user for the necessary information.

Depending on the database extension you are using, the `DBInit` function might return a session ID of zero if it fails to initiate a session, or it might return a nonzero session ID and a result code other than `noErr`. In the latter case, you can pass the session ID to the `DBGetErr` function to determine the cause of the error. If the `DBInit` function returns a nonzero session ID and a result code other than `noErr`, you must call the `DBEnd` function before making another attempt to open the session.

3. Prepare a query, and send it to the data server by calling the `DBSend` and `DBSendItem` functions one or more times.

   An application that uses the low-level interface must be capable of creating a query for the data server in the language and format required by that data server.

   The `DBSend` function sends a query or a portion of a query to the data server. The data server appends this portion of the query to any portion you sent previously. Because the Data Access Manager and data server do not modify the string you send in any way, they do not insert any delimiter between fragments of queries that you send to the data server. If you want a blank or a semicolon to be included between query fragments, or if you want to use return characters to divide the query into lines of text, you must include them in the character string that you send with the `DBSend` function. The data string that you send with the `DBSend` function can be any length up to 64 KB.

   The `DBSendItem` function sends a single data item to the data server. Use the `DBSendItem` function to send data items to the data source in the same format as they are retrieved from the data source by the `DBGetItem` function. You must specify the data type as an input parameter and, for any data type that does not have an implied length, you must specify the length as well. The database extension or the data server (depending on how the system is implemented) converts the data item to a character string and appends it to the query, just as a query program fragment is appended to the query by the `DBSend` function.

   You can call the `DBSend` and `DBSendItem` functions as many times as you wish to send your query to the data server.

   Listing 12-4 sends the Data Access Language query fragment "`print 451+222;`" to the Data Access Language server.

Listing 12-4 Sending a query fragment

```
FUNCTION MySendFragment(sessID: LongInt): OSErr;
VAR
   value1:                        LongInt;
   value2:                        LongInt;
   text1, text2, text3:           Str15;
   text1Ptr, text2Ptr, text3Ptr: Ptr;
   rc:      OSErr;
BEGIN
   text1 := 'print ';
   value1 := 451;
   text2 := '+';
   value2 := 222;
   text3 := ';';
   MySetTextPtrs(text1, text1Ptr, text2, text2Ptr,
                 text3, text3Ptr);
   rc := DBSend (sessID, text1Ptr, LENGTH(text1), NIL);
   IF rc = noErr THEN
      rc := DBSendItem (sessID, typeInteger, 0, 0, 0,
                        Ptr(ORD(@value1)), NIL);
   IF rc = noErr THEN
      rc := DBSend (sessID, text2Ptr, LENGTH(text2), NIL);
   IF rc = noErr THEN
      rc := DBSendItem (sessID, typeInteger, 0, 0, 0,
                        Ptr(ORD(@value2)), NIL);
   IF rc = noErr THEN
      rc := DBSend (sessID, text3Ptr, LENGTH(text3), NIL);
   MySendFragment := rc;
END;
```

4. Use the `DBExec` function to initiate execution of the query.

   Depending on the way the system you are using is implemented, the `DBExec` function might return control to your application as soon as the query has begun execution.

5. Use the `DBState` function to determine the status of the data source.

   The `DBState` function tells you when the data server has finished executing the query you just sent. If you have requested data, the data server stores the data you requested but does not send it to your application until you request it explicitly. The `DBState` function tells you when the data is available; if data is available, go to step 6. If you wish to send another query, return to step 3. If you are finished using the data source, skip to step 7.

6. Call the `DBGetItem` function repeatedly to retrieve the data.

   The `DBGetItem` function retrieves the next data item from the data server. You can also use this function to obtain information about the next data item without retrieving the data. When you use the `DBGetItem` function to retrieve a data item, you must specify the location and size of the buffer into which the function is to place that item. If you know beforehand what kind of data to expect, you can allocate a buffer of the exact size you need. If you do not know what type of data to expect, you can first call the `DBGetItem` function with a `NIL` pointer to the data buffer. The `DBGetItem` function then returns information about the next data item without actually retrieving it. You can then allocate the appropriate buffer and call `DBGetItem` again.

   Alternatively, to avoid calling `DBGetItem` twice for each data item, you can allocate a buffer that you expect to be of sufficient size for any data item and call the `DBGetItem` function. If the buffer is not large enough for the data item, the `DBGetItem` function returns the `rcDBError` result code, but still returns information about the data item. You can then allocate the necessary buffer, call the `DBUnGetItem` function to go back one data item, and call the `DBGetItem` function again to retrieve the data item a second time.

   The `DBGetItem` function includes a `timeout` parameter that you can use to specify the maximum amount of time that the database extension should wait to receive results from the data server before canceling the command. If the database extension you are using does not support asynchronous execution of routines, you can use the `timeout` parameter to return control to your application while a query is executing. To use the `timeout` parameter in this way, call the `DBGetItem` function periodically, specifying a brief period of time for the `timeout` parameter. Your application can then retrieve the next data item as soon as execution of the query is complete without having to call the `DBState` function to determine when data is available. The `DBGetItem` function ignores the `timeout` parameter if you make an asynchronous call to this function.

7. When you are finished using the data source, you must use the `DBEnd` function to terminate the session. You must call the `DBEnd` function after the `DBInit` function has returned a nonzero session ID, even if it also returned an error.

The procedure in Listing 12-5 uses the low-level interface to send a Data Access Language routine to the Data Access Language server on a remote computer and then retrieves the results. The procedure initiates a session with a remote database and calls the `MySendFragment` routine (Listing 12-4) to send a query. Next, it executes the query, checks the status of the remote database server, and retrieves the data when it's available. This example retrieves only one data item. To retrieve more than one data item, put the data-retrieval code in a loop.

Listing 12-5 assumes that the database extension does not support asynchronous execution of Data Access Manager routines. For an example of asynchronous execution of routines, see Listing 12-1 beginning on page 12-18.

Listing 12-5 Using the low-level interface

```
PROCEDURE MyLoLevel(VAR thisSession: LongInt; VAR sessErr: OSErr);
VAR
   theDDevName:                          Str63;
   theHost, theUser:                     Str255;
   thePasswd, theConnStr:                Str255;
   packErr, initErr, sendErr, execErr: OSErr;
   stateErr, getErr, endErr:             OSErr;
   myTimeout:                            LongInt;
   myType:                               DBType;
   len, places, flags:                   Integer;
   myBuffer:                             Ptr;
   myDataInfo:                           Boolean;
   myDataReturned:                       Boolean;
BEGIN
   sessErr := noErr;       {assume everything went fine}
   packErr := InitDBPack;  {init the Data Access Mgr}
   {Set up values for theDDevName, theHost, theUser, thePasswd, }
   { and theConnStr. You can display a dialog box prompting }
   { the user to supply some of these parameters.}
   theDDevName := 'DAL';
   theHost := 'The Host System Name';
   theUser := 'Joe User';
   thePasswd := 'secret';
   theConnStr := 'extra stuff as needed';
   initErr := DBInit(thisSession, theDDevName, theHost, theUser,
                     thePasswd, theConnStr, NIL);
   IF initErr <> noErr THEN
   BEGIN
      sessErr := initErr;
      IF thisSession <> 0 THEN endErr := DBEnd(thisSession, NIL);
      EXIT(MyLoLevel);
   END;
   {send a query or query fragment to the remote data server}
   sendErr := MySendFragment(thisSession);
   {If there's an error, then probably something went wrong with }
   { DBSend or DBSendItem.  Don't forget to end the session.}
   IF sendErr <> noErr THEN
   BEGIN
      sessErr := sendErr;
      endErr := DBEnd(thisSession, NIL);
      EXIT(MyLoLevel);
   END;
```

```
{The query has been sent.  This example assumes that }
{ the query will return data.}
execErr := DBExec(thisSession, NIL);
IF execErr = noErr THEN
BEGIN
   stateErr := rcDBExec;
   WHILE (stateErr = rcDBExec) DO
   BEGIN             {while waiting for stateErr <> rcDBExec, }
      MyGoDoSomething; { let other apps run}
      stateErr := DBState(thisSession, NIL);
   END;
   {DBState returned a result code other than rcDBExec. }
   { If it's rcDBValue, there are results to retrieve. }
   { Otherwise, it's probably an error.}
   IF stateErr = rcDBValue THEN
   BEGIN
      {call DBGetItem once to get info on the data item and }
      { call DBGetItem a second time to get the data item}
      myTimeout := 2*60;   {2*60 ticks = 2 seconds}
      myType := DBType(typeAnyType);
      myDataInfo := FALSE;
      WHILE NOT myDataInfo DO
      BEGIN
         getErr := DBGetItem(thisSession, myTimeout, myType,
                              len, places, flags, NIL, NIL);
         {If you timed out, then give up control.  When }
         { control returns, continue getting the info.}
         IF getErr = rcDBBreak THEN MyGoDoSomething
         ELSE IF (getErr = noErr) OR (getErr = rcDBValue) THEN
            myDataInfo := TRUE
         ELSE
         BEGIN
            sessErr := getErr;
            endErr := DBEnd(thisSession, NIL);
            EXIT(MyLoLevel);
         END;
      END; {while}
      {At this point, you may want to examine the info }
      { about the data item before calling DBGetItem a }
      { second time to actually retrieve it.}
      {MyGimmeSpace returns a pointer to where you want }
      { the data item to go.}
      myBuffer := MyGimmeSpace(len);
```

```
         myDataReturned := FALSE;
         WHILE NOT myDataReturned DO
         BEGIN
            getErr := DBGetItem(thisSession, myTimeout, myType,
                                len, places, flags, myBuffer,
                                NIL);
            {If you timed out, then give up control.  When }
            { control returns, continue getting the data.}
            IF getErr = rcDBBreak THEN MyGoDoSomething
            ELSE IF (getErr = noErr) OR
                  (getErr = rcDBValue) THEN myDataReturned := TRUE
            ELSE
            BEGIN
               sessErr := getErr;
               endErr := DBEnd(thisSession, NIL);
               EXIT(MyLoLevel);
            END;
         END; {while}
      END
      ELSE sessErr := stateErr;
   END
   ELSE sessErr := execErr;
   endErr := DBEnd(thisSession, NIL);
END;
```

Note that, even if you are using the low-level interface to send queries to the data server, you might want to use the high-level functions to retrieve data and convert it to text.

Getting Information About Sessions in Progress

If your application is only one of several on a single Macintosh computer connected to data servers, you can use the `DBGetConnInfo` and `DBGetSessionNum` functions to obtain information about the sessions in progress. If you know the session ID (which is returned by the `DBInit` function when you open a session), you can use the `DBGetConnInfo` function to determine the database extension being used, the name of the host system on which the session is running, the user name and connection string used to initiate the session, the time at which the session started, and the status of the session. The status of the session specifies whether the data server is executing a query or waiting for another query fragment, whether there is output data available, and whether execution of a query ended in an error.

If you do not know the session ID, or if you want to get information about all open sessions, you can specify a database extension and a session number when you call the `DBGetConnInfo` function. Although there can be only one active session with a given session ID, session numbers are unique only for a specific database extension. Because the database extension assigns session numbers sequentially, starting with 1, you can call the `DBGetConnInfo` function repeatedly for a given database extension, incrementing the session number each time, to obtain information about all sessions open for that database extension. Your application need not have initiated the session to obtain information about it in this fashion.

The `DBGetSessionNum` function returns the session number when you specify the session ID. You can use this function to determine the session numbers for the sessions opened by your own application. You might want this information, for example, so you can distinguish your own sessions from those opened by other applications when you use the `DBGetConnInfo` function to get information about all open sessions.

Processing Query Results

You can use the low-level function `DBGetItem` to retrieve a single data item returned by a query, or you can use the high-level function `DBGetQueryResults` to retrieve all of the query results at once. If you use the `DBGetQueryResults` function, you can then use the `DBResultsToText` function to convert the results to ASCII text. The `DBResultsToText` function calls routines called *result handlers*, which are installed in memory by applications or by system extensions (files containing `'INIT'` resources). This section discusses the use of the `DBGetItem` and `DBGetQueryResults` functions and describes how to write and install a result handler.

Getting Query Results

The `DBGetItem` function retrieves a single data item that was returned by a data source in response to a query. When you call the `DBGetItem` function, you specify the data type to be retrieved. If you do not know what data type to expect, you can specify the `typeAnyType` constant for the `dataType` parameter, and the data server returns the next data item regardless of data type. It also returns information about the data item, including data type and length.

If you do not know the length of the next data item, you can specify `NIL` for the `buffer` parameter in the `DBGetItem` function, and the data server returns the data type, length, and number of decimal places without retrieving the data item. The next time you call the `DBGetItem` function with a nonzero value for the buffer parameter, the function retrieves the data item.

If you want to skip a data item, specify the `typeDiscard` constant for the `dataType` parameter. Then the next time you call the `DBGetItem` function, it retrieves the following data item.

You should use the `DBGetItem` function if you want complete control over the retrieval of each item of data. If you want the Data Access Manager to retrieve the data for you, use the `DBGetQueryResults` function instead.

Table 12-1 shows the data types recognized by the Data Access Manager. You use a constant to specify each data type, as follows:

```
CONST {data types}
      typeAnyType    =  0;       {can be any data type}
      typeNone       = 'none';   {no more data expected}
      typeBoolean    = 'bool';   {Boolean}
      typeSMInt      = 'shor';   {short integer}
      typeInteger    = 'long';   {integer}
      typeSMFloat    = 'sing';   {short floating point}
      typeFloat      = 'doub';   {floating point}
      typeDate       = 'date';   {date}
      typeTime       = 'time';   {time}
      typeTimeStamp  = 'tims';   {date and time}
      typeChar       = 'TEXT';   {character}
      typeDecimal    = 'deci';   {decimal number}
      typeMoney      = 'mone';   {money value}
      typeVChar      = 'vcha';   {variable character}
      typeVBin       = 'vbin';   {variable binary}
      typeLChar      = 'lcha';   {long character}
      typeLBin       = 'lbin';   {long binary}
      typeDiscard    = 'disc';   {discard next data item}
      typeUnknown    = 'unkn';   {result handler for unknown }
                                 { data type}
      typeColBreak   = 'colb';   {result handler for column break}
      typeRowBreak   = 'rowb';   {result handler for end of line}
```

The writer of a database extension can define other data types to support specific data sources or data servers.

Each data type has a standard definition, shown in Table 12-1. For example, if the `DBGetItem` function returns the `typeInteger` constant for the `dataType` parameter, you know that the data item represents an integer value and that a 4-byte buffer is necessary to hold it. Similarly, if you are using the `DBSendItem` function to send to the data server a data item that you identify as `typeFloat`, the data server expects to receive an 8-byte floating-point value.

Notice that some of these data types are defined to have a specific length (referred to as an *implied length*), and some do not. The `len` parameter of the `DBSendItem` and `DBGetItem` functions indicates the length of an individual data item. The `DBGetQueryResults` function returns a handle to an array of lengths, decimal places, and flags in the `colInfo` field of the results record. The `typeAnyType`, `typeColBreak`, and `typeRowBreak` constants do not refer to specific data types, and therefore the length specification is not applicable for these constants.

Table 12-1 Data types defined by the Data Access Manager

Constant	Length	Definition
`typeAnyType`	NA	Any data type (used as an input parameter to the `DBGetItem` function only; never returned by the function).
`typeNone`	0	Empty.
`typeBoolean`	1 byte	`TRUE` (`1`) or `FALSE` (`0`).
`typeSMInt`	2 bytes	Signed integer value.
`typeInteger`	4 bytes	Signed long integer value.
`typeSMFloat`	4 bytes	Signed floating-point value.
`typeFloat`	8 bytes	Signed floating-point value.
`typeDate`	4 bytes	Date; a long integer value consisting of a year (most significant 16 bits), month (8 bits), and day (least significant 8 bits).
`typeTime`	4 bytes	Time; a long integer value consisting of an hour (0–23; most significant 8 bits), minute (8 bits), second (8 bits), and hundredths of a second (least significant 8 bits).
`typeTimeStamp`	8 bytes	Date and time. A long integer date value followed by a long integer time value.
`typeChar`	Any	Fixed-length character string, not `NULL` terminated. The length of the string is defined by the specific data source.

Table 12-1 Data types defined by the Data Access Manager (continued)

Constant	Length	Definition
`typeDecimal`	Any	Packed decimal string. A contiguous string of 4-bit nibbles, each of which contains a decimal number, except for the low nibble of the highest-addressed byte (that is, the last nibble in the string), which contains a sign. The value of the sign nibble can be 10, 12, 14, or 15 for a positive number or 11 or 13 for a negative number; 12 is recommended for a positive number and 13 is recommended for a negative number. The most significant digit is the high-order nibble of the lowest-addressed byte (that is, the first nibble to appear in the string). The total number of nibbles (including the sign nibble) must be even; therefore, the high nibble of the highest-addressed byte of a number with an even number of digits must be 0. For example, the number +123 is represented as $123C. Bits 7 4 3 0 Address 1 2 A 3 C A+1 The number –1234 is represented as $01234D. Bits 7 4 3 0 Address 0 1 A 2 3 A+1 4 D A+2 The length of a packed decimal string is defined as the number of bytes, including any extra leading 0 and the sign nibble. A packed decimal string can have from 0 to 31 digits, not including the sign nibble. In addition to the length of a packed decimal string, each data item has an associated value that indicates the number of digits that follow the decimal place. The `places` parameter in the `DBGetItem` and `DBSendItem` functions indicates the number of decimal places in an individual data item. The `DBGetQueryResults` function returns the number of decimal places.
`typeMoney`	Any	Same as `typeDecimal`, but always has two decimal places.
`typeVChar`	Any	Variable-length character string, `NULL` terminated.

continued

Table 12-1 Data types defined by the Data Access Manager (continued)

Constant	Length	Definition
typeVBin	Any	Not defined. Reserved for future use.
typeLChar	Any	Not defined. Reserved for future use.
typeLBin	Any	Not defined. Reserved for future use.
typeDiscard	NA	Do not retrieve the next data item (used as an input parameter to the DBGetItem function only; never returned by the function).
typeUnknown	NA	A dummy data type for the result handler that processes any data type for which no other result handler is available (used as an input parameter to the DBInstallResultHandler, DBRemoveResultHandler, and DBGetResultHandler functions only; never returned by the DBGetItem function).
typeColBreak	NA	A dummy data type for the result handler that the DBGetQueryResults function calls after each item that is not the last item in a row (used as an input parameter to the DBInstallResultHandler, DBRemoveResultHandler, and DBGetResultHandler functions only; never returned by the DBGetItem function).
typeRowBreak	NA	A dummy data type for the result handler that the DBGetQueryResults function calls at the end of each row (used as an input parameter to the DBInstallResultHandler, DBRemoveResultHandler, and DBGetResultHandler functions only; never returned by the DBGetItem function).

The DBGetQueryResults function retrieves all of the data that was returned by a data source in response to a query, unless insufficient memory is available to hold the data, in which case it retrieves as many complete rows of data as possible. The DBGetQueryResults function stores the data in a structure called a *results record.* You must allocate the results record data structure and pass this record to the DBGetQueryResults function. The Data Access Manager allocates the handles inside the results record. When your application is finished using the results record, you must deallocate both the results record and the handles inside the results record.

The results record is defined by the `ResultsRecord` data type.

```
TYPE ResultsRecord =
   RECORD
      numRows:    Integer;          {number of rows retrieved}
      numCols:    Integer;          {number of columns per row}
      colTypes:   ColTypesHandle;   {type of data in each column}
      colData:    Handle;           {array of data items}
      colInfo:    ColInfoHandle;    {info about each data item}
   END;
```

The `numRows` field in the results record indicates the total number of rows retrieved. If the `DBGetQueryResults` function returns a result code other than `rcDBValue`, then not all of the data actually returned by the data source was retrieved. This could happen, for instance, if the user's computer does not have sufficient memory space to hold all the data. In this case, your application can make more space available (by writing the data in the data record to disk, for example) and then call the `DBGetQueryResults` function again to complete retrieval of the data.

Note
The `DBGetQueryResults` function retrieves whole rows only; if it runs out of space in the middle of a row, it stores the partial row in a private buffer so that the data in the results record ends with the last complete row. Because the last partial row is no longer available from the data server, you cannot start to retrieve data with the `DBGetQueryResults` function and then switch to the `DBGetItem` function to complete the data retrieval. ◆

The `numCols` field indicates the number of columns in each row of data.

The `colTypes` field is a handle to an array of data types, specifying the type of data in each column. The number of elements in the array is equal to the value in the `numCols` field. Table 12-1 beginning on page 12-39 shows the standard data types.

The `colData` field is a handle to the data retrieved by the `DBGetQueryResults` function.

The `colInfo` field is a handle to an array of records of type `DBColInfoRecord`, each of which specifies the length, places, and flags for a data item. There are as many records in the array as there are data items retrieved by the `DBGetQueryResults` function. Here is the `DBColInfoRecord` type definition:

```
TYPE DBColInfoRecord =
   RECORD
      len:        Integer;       {length of data item}
      places:     Integer;       {places for decimal and }
                                 { money data items}
      flags:      Integer;       {flags for data item}
   END;
```

The `len` field indicates the length of the data item. The `DBGetQueryResults` function returns a value in this field only for those data types that do not have implied lengths; see Table 12-1 beginning on page 12-39.

The `places` field indicates the number of decimal places in data items of types `typeMoney` and `typeDecimal`. For all other data types, the `places` field returns 0.

The least significant bit of the `flags` field is set to 1 if the data item is in the last column of the row. The third bit of the `flags` field is 1 if the data item is `NULL`. You can use the constants `kDBLastColFlag` and `kDBNullFlag` to test for these flag bits.

Converting Query Results to Text

The `DBResultsToText` function provided by the high-level interface converts the data retrieved by the `DBGetQueryResults` function into strings of ASCII text. This function makes it easier for you to display retrieved data for the user.

For the `DBResultsToText` function to convert data of a specific type to text, either the application or the system software must have a routine called a *result handler.* With System 7, Apple Computer, Inc., provides system result handlers for the data types listed here. (These data types are described in Table 12-1 beginning on page 12-39.)

Data type	Constant	Data type	Constant
Boolean	`typeBoolean`	Time	`typeTime`
Short integer	`typeSMInt`	Date and time	`typeTimeStamp`
Integer	`typeInteger`	Character	`typeChar`
Short floating point	`typeSMFloat`	Decimal number	`typeDecimal`
Floating point	`typeFloat`	Money value	`typeMoney`
Date	`typeDate`	Variable character	`typeVChar`

Note
Apple's system result handler for the variable character (`typeVChar`) data type strips trailing spaces from the character string. ◆

In addition to the result handlers for these standard data types, Apple provides the following three system result handlers, which correspond to no specific data type:

Data type	Constant
Unknown	`typeUnknown`
Column break	`typeColBreak`
End of line	`typeRowBreak`

The `typeUnknown` result handler processes any data type for which no other result handler is available. The `DBResultsToText` function calls the `typeColBreak` result handler after each item that is not the last item in a row. This result handler does not correspond to any data type, but adds a delimiter character to separate columns of text. The default `typeColBreak` result handler inserts a tab character. Similarly, the `DBResultsToText` function calls the `typeRowBreak` result handler at the end of each row of data to add a character that separates the rows of text. The default `typeRowBreak` result handler inserts a return character. Your application can install your own `typeColBreak` and `typeRowBreak` result handlers to insert whatever characters you wish—or to insert no character at all, if you prefer.

You can install result handlers for any data types you know about. When you call the `DBInstallResultHandler` function, you can specify whether the result handler you are installing is a system result handler. A ***system result handler*** is available to all applications that use the system. All other result handlers (called ***application result handlers***) are associated with a particular application. The `DBResultsToText` function always uses a result handler for the current application in preference to a system result handler for the same data type. When you install a system result handler for the same data type as an already installed system result handler, the new result handler replaces the old one. Similarly, when you install an application result handler for the same data type as a result handler already installed for the same application, the new result handler replaces the old one for that application.

Result handlers are stored in memory. The Data Access Manager installs its system result handlers the first time the Macintosh Operating System loads the Data Access Manager into memory. You must reinstall your own application result handlers each time your application starts up. You can also install your own system result handlers each time your application starts up, or you can provide a system extension (that is, a file with an `'INIT'` resource) that installs system result handlers each time the user starts up the system.

Here is a function declaration for a result handler function:

```
FUNCTION MyResultHandler (dataType: DBType;
                          theLen, thePlaces, theFlags: Integer;
                          theData: Ptr; theText: Handle): OSErr;
```

The `dataType` parameter specifies the data type of the data item that the `DBResultsToText` function is passing to the result handler. Table 12-1 beginning on page 12-39 describes the standard data types.

The parameters `theLen` and `thePlaces` specify the length and number of decimal places of the data item that the `DBResultsToText` function wants the result handler to convert to text.

The parameter `theFlags` is the value returned for the `flags` parameter by the `DBGetItem` function. If the least significant bit of this parameter is set to 1, the data item is in the last column of the row. If the third bit of this parameter is set to 1, the data item is `NULL`. You can use the constants `kDBLastColFlag` and `kDBNullFlag` to test for these flag bits.

The parameter `theData` is a pointer to the data that the result handler is to convert to text.

The parameter `theText` is a handle to the buffer that is to hold the text version of the data. The result handler should use the Memory Manager's `SetHandleSize` function to increase the size of the buffer as necessary to hold the new text, and append the new text to the end of the text already in the buffer. The `SetHandleSize` function is described in the chapter "Memory Manager" in *Inside Macintosh: Memory.*

If the result handler successfully converts the data to text, it should return a result code of 0 (`noErr`).

You can use the `DBInstallResultHandler` function to install a result handler and the `DBRemoveResultHandler` function to remove an application result handler. You can install and replace system result handlers, but you cannot remove them.

The following line of code installs an application result handler. The first parameter (`typeInteger`) specifies the data type that this result handler processes. The second parameter (`MyTypeIntegerHandler`) is a pointer to the result handler routine. The last parameter (`FALSE`) is a Boolean value specifying that this routine is not a system result handler.

```
err := DBInstallResultHandler
                        (typeInteger,@MyTypeIntegerHandler,FALSE);
```

Listing 12-6 shows a result handler that converts the integer data type to text.

Listing 12-6 A result handler

```
FUNCTION MyTypeIntegerHandler(datatype: DBType; theLen: Integer;
                              theData: Ptr;
                              theText: Handle): OSErr;
VAR
   theInt:        LongInt;
   theTextLen:    LongInt;
   temp:          Str255;
   atemp1:        Ptr;
   atemp2:        LongInt;
   atemp3:        LongInt;
BEGIN
   BlockMove(theData, @theInt, sizeof(theInt));
   NumToString(theInt, temp);                {convert to text}
   theTextLen := GetHandleSize(theText);     {get current size }
                                             { of theText}
                                             {size text handle}
   SetHandleSize(theText, theTextLen + LongInt(LENGTH(temp)));
   IF (MemError <> noErr) THEN
      MyTypeIntegerHandler := MemError
   ELSE
   BEGIN
      atemp1 := Ptr(ORD(@temp));
      atemp2 := LongInt(theText^) + theTextLen;
      atemp3 := LongInt(LENGTH(temp));
      {use BlockMove to append text}
      BlockMove(P2CStr(atemp1), Ptr(atemp2), atemp3);
      MyTypeIntegerHandler := MemError;
   END;
END;
```

Creating a Query Document

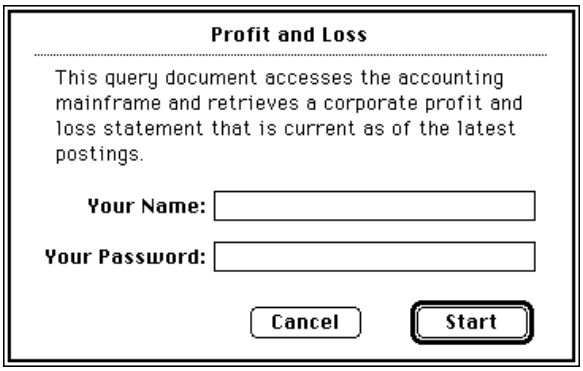
A query document is a file of type `'qery'` that contains a `'qrsc'` resource and one or more `'wstr'` resources, and may contain a `'qdef'` resource plus other resources. Query documents make it possible for you to write applications that can communicate with data servers without requiring familiarity with the command language used by the data server. Because a query document is most useful if it can be used by many different applications, no query document should depend on the presence of a particular application in order to function.

An application can call the `DBGetNewQuery` function to convert a `'qrsc'` resource into a ***query record*** in memory. A query record specifies connection information and also contains a handle to an array of queries; each query can be either a complete query or a template for a query. If the `'wstr'` resource is a template, it contains the commands and data necessary to create a query, without any information that the user must add just before the query is sent. The `'qdef'` resource contains a query definition function, which can modify the query record and, if necessary, fill in the query template to create a complete query. The `DBStartQuery` function sends the query pointed to by a query record to a data server. The following sections describe the contents of a query document, describe query records, and define the `'qrsc'`, `'wstr'`, and `'qdef'` resources.

User Interface Guidelines for Query Documents

All query documents should behave in fundamentally the same way. They should be self-explanatory and should never execute a query without an explicit command from the user. When your application opens a query document, the query document should display a dialog box with enough information about the query so that the user can decide if it's the right query. The dialog box should describe the purpose of the query, what kind of data it transfers and in which direction, the type of data source it accesses, and any warnings or instructions. The dialog box can describe how the user interprets the data, such as the name of each field in a record. Figure 12-6 shows an example of a query document dialog box.

Figure 12-6 A query document dialog box



This dialog box should allow the user to cancel the request for data. In addition, it may be useful to allow the user to set parameters with text boxes, checkboxes, or radio buttons. For example, a query to a database of financial information could provide a list of these options: a trial balance, profit-and-loss statements, or net worth reports. Save the last set of user-specified parameters with the query document. This way the user can review the parameters used to generate the data or use the same parameters the next time.

Once a query starts running, it must be able to complete its task without user intervention. If a query must run modally (that is, it must run to completion before returning control to the user), display a dialog box that shows the query's progress and be sure to return control to the user as soon as possible. The philosophy of this process is similar to that of receiving electronic mail—that is, inform the user when the information arrives, but let the user decide when to read it.

Whenever possible, query documents should check that data is compatible before transmitting it to a data source. Establish a connection with a data source only after you have checked the data.

Contents of a Query Document

The query document must contain

- one `'qrsc'` resource, as defined in the next section, "Query Records and Query Resources"
- one `'STR#'` resource that contains the name of the database extension to be used, plus any host, user name, password, and connection string needed for the `DBInit` function
- one or more `'wstr'` resources containing queries—that is, strings of commands and data that the `DBSend` function sends to the data server and that the `DBExec` function executes

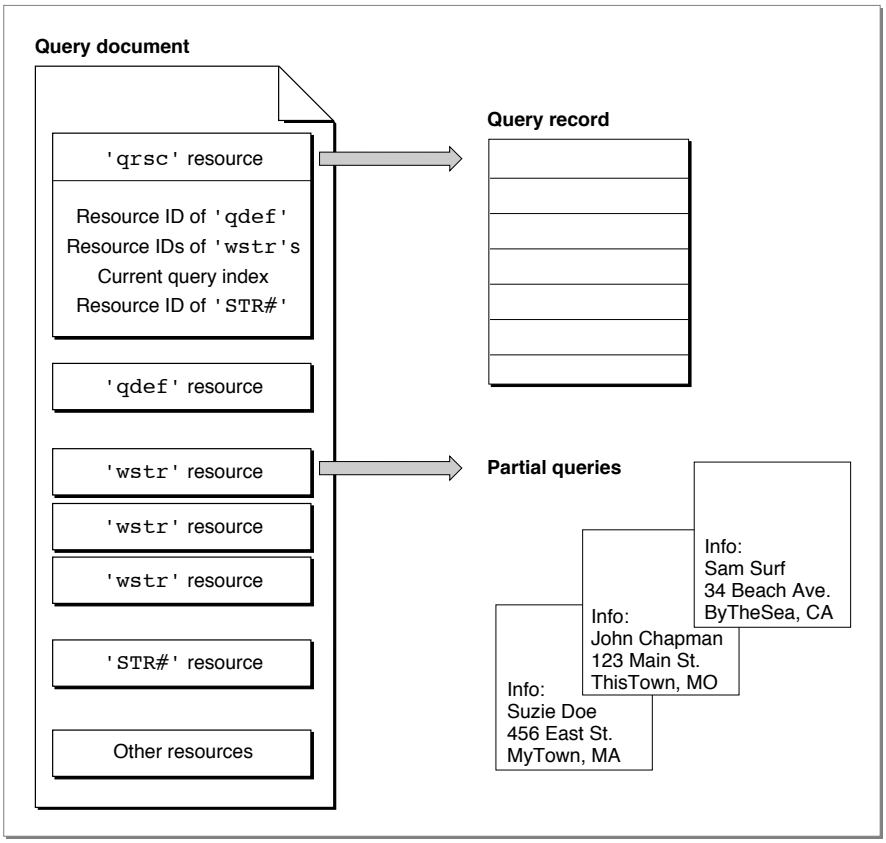
A `'wstr'` resource consists of a 2-byte length field followed by a character string. (The *w* in `'wstr'` refers to the length word as opposed to the length byte used in an `'STR '` resource.) Each `'wstr'` resource contains one query (or one query template, to be modified by the query definition function before it is sent to the data server). The `'qrsc'` resource includes an array that lists the resource ID numbers of all of the `'wstr'` resources in the query document and an index into the array that specifies which one of the `'wstr'` resources should be sent to the data server.

In addition, the query document may contain

- a `'qdef'` resource that contains a query definition function
- any resources needed by the query definition function, such as `'DLOG'` and `'DITL'` resources (which support dialog boxes)
- resources to support an icon (to replace the default icon that the Finder uses for files of type `'qery'`); see the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials* for more information on icon resources and for guidelines on designing icons

Figure 12-7 illustrates the relationship between the resources in a query document and the query record.

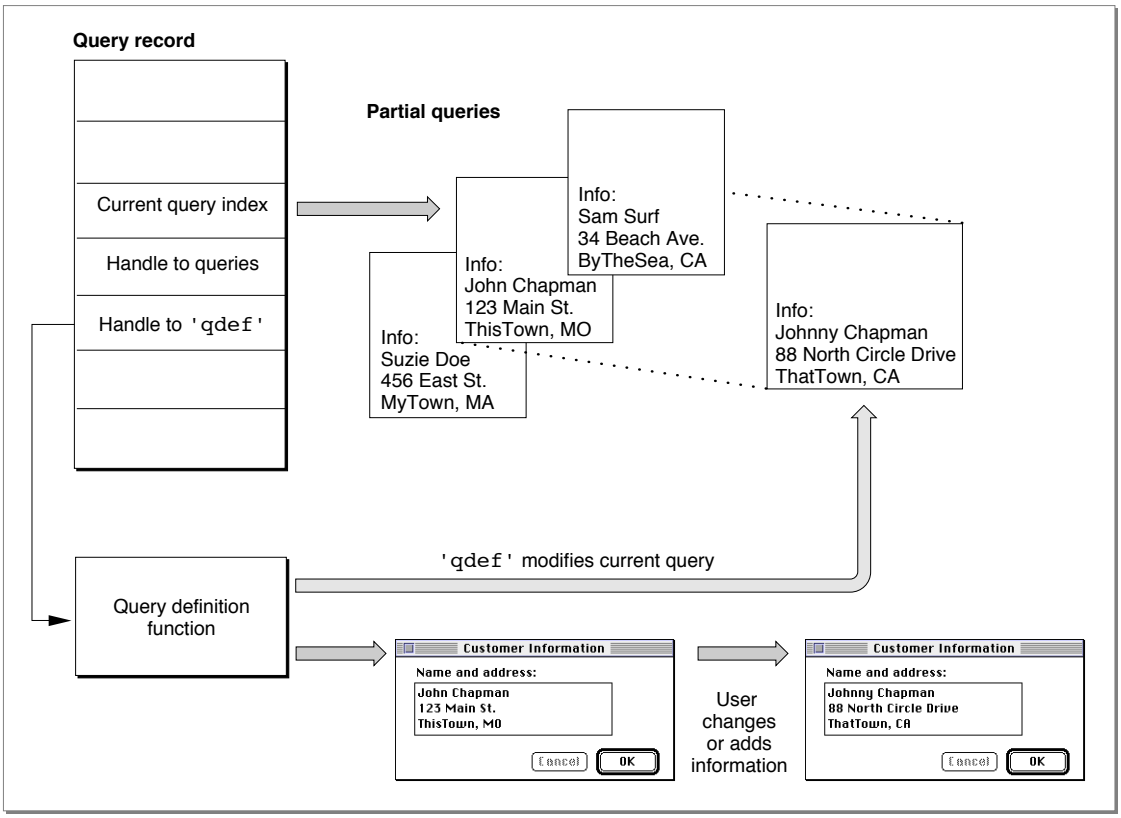
Figure 12-7 The relationship between resources in a query document and the query record



The query document in Figure 12-7 contains a `'qrsc'` resource that specifies the resource ID of a `'qdef'` resource, the resource IDs of three `'wstr'` resources, and the resource ID of an `'STR#'` resource. It also specifies which of the three `'wstr'` resources represents the current query.

The `DBGetNewQuery` function creates the query record and partial queries from this information. Your application can use the `DBStartQuery` function to send a query to a data server. The `DBStartQuery` function calls the query definition function referred to by the query record (if any). The query definition function can prompt the user for information and modify the query as needed. Figure 12-8 illustrates a query record that contains a handle to an array of queries, a handle to a query definition function, and an index that identifies the current query. The query definition function displays a dialog box and modifies the current query according to the user's input. Once the query definition function modifies the current query and returns, the `DBStartQuery` function sends the query to the data server.

Figure 12-8 The relationship between a query definition function and queries



Query Records and Query Resources

The `DBGetNewQuery` function converts the `'qrsc'` resource in the query document into a query record in memory. The query definition function can then modify the query record before the application sends the query to the data server. See "The Query Record" beginning on page 12-57 for a description of the query record. See "The Query Resource" beginning on page 12-91 for the format of a `'qrsc'` resource. The next section provides information about query definition functions.

Writing a Query Definition Function

Before the `DBStartQuery` function sends a query to a data server, it calls the query definition function specified by the `queryProc` field in the query record. The purpose of the query definition function is to modify the query and the query record before the query is sent to the data server. The query definition function can use dialog boxes to request information from the user. Because a query document is most useful if it can be used by many different applications, no query definition function should depend on the presence of a particular application.

If you want to include a query definition function, you must make it the first piece of code in a resource of type `'qdef'` in the query document.

Here is a function declaration for a query definition function.

```
FUNCTION MyQDef (VAR sessID: LongInt; query: QueryHandle): OSErr;
```

If the application has already initiated a session with the data server, the `DBStartQuery` function passes the session ID for that session in the `sessID` parameter to the query definition function. If the query definition function receives a 0 in this parameter, then the Data Access Manager has not initiated a session. In this case, the query definition function can return a 0 in the `sessID` parameter, or it can call the `DBInit` function to initiate a session and then return the session ID in this parameter.

If the query definition function returns a 0 in the `sessID` parameter, the `DBStartQuery` function calls the `DBInit` function and then calls the `DBSend` function to send a query to the data server. If the query definition function returns a session ID in this parameter, the `DBStartQuery` function calls the `DBSend` function immediately.

The `query` parameter to the query definition function specifies a handle to the query record. The query definition function can modify any of the fields in the query record, including the `currQuery` field that specifies which query is to be sent to the data server. In addition, the query definition function can modify an existing query or create a new query, adding the handle to the new query to the query list. Note that, because a query in memory consists only of a 2-byte length value followed by a character string, the query definition function has to know the exact contents and structure of a query in order to modify it.

The query definition function must return the `noErr` result code as the function result if the function executed successfully. If it returns any other value, the `DBStartQuery` function does not call the `DBSend` function. The query definition function can return any result code, including `noErr`, `userCanceledErr`, or `rcDBError`.

When the `DBStartQuery` function calls the query definition function, the current resource file is the file that contains the `'qrsc'` resource from which the Data Access Manager created the query record. When the query definition function returns control to the Data Access Manager, the current resource file must be unchanged. See the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox* for more information on the current resource file.

The query definition function can allocate memory and use the `dataHandle` field in the query record to store a handle to it. The query definition function must free any memory it allocates before terminating.

Listing 12-7 shows a query definition function that uses a dialog box to prompt the user for a user name and password and then modifies the query record accordingly.

Listing 12-7 A query definition function

```
FUNCTION MyQDef(VAR sessID: LongInt; query: QueryHandle): OSErr;
CONST
   myNameItem        =  7;
   myPassWordItem    =  8;
VAR
   myNumRes:         Integer;
   myResList:        ResListHandle;
   myResLPtr:        ResListPtr;
   myIndex:          Integer;
   myDialog:         DialogPtr;
   myDlogID:         Integer;
   itemType:         Integer;
   itemHName:        Handle;
   itemHPasswd:      Handle;
   itemBox:          Rect;
   mySTR:            ARRAY[1..2] OF Str255;
   itemHit:          Integer;
   myQErr:           OsErr;
BEGIN
   {If sessID = 0 no session has been initiated. Your qdef may }
   { optionally initiate a session, or it can let DBStartQuery }
   { take care of this.  In this example, the qdef doesn't }
   { check the sessID parameter.}
   HLock(Handle(query));
   myNumRes := query^^.numRes;
```

```
myResList := query^^.resList;
HLock(Handle(myResList));
myResLPtr := myResList^;
myIndex := 0;
{look for a 'DLOG' resource}
WHILE (myIndex < myNumRes) AND
      (myResLPtr^[myIndex].theType <> 'DLOG') DO
BEGIN
   myIndex := myIndex + 1;
END;
IF (myIndex < myNumRes) THEN  {found the 'DLOG' resource}
   myDlogID := myResLPtr^[myIndex].id
ELSE
BEGIN
   {The 'DLOG' wasn't found; exit with no error.  This }
   { is probably OK; it just means that the query }
   { and the query record don't get modified.}
   MyQDEF := noErr;
   HUnlock(Handle(query));
   HUnlock(Handle(myResList));
   EXIT(MyQDef);
END;
{found the 'DLOG' and its ID; now display the dialog box}
myDialog := GetNewDialog(myDlogID, Ptr(NIL), WindowPtr(-1));
SetPort(GrafPtr(myDialog));
REPEAT
   ModalDialog(@MyEventFilter, itemHit);
UNTIL ((itemHit = kOK) OR (itemHit = kCancel));
IF itemHit = kOK THEN
BEGIN
   {The user clicked the OK button. Update the user }
   { and password fields of the query record.}
   GetDialogItem(myDialog, myNameItem, itemType, itemHName,
                 itemBox);
   GetDialogItemText(itemHName, mySTR[1]);
   GetDialogItem(myDialog, myPassWordItem, itemType,
               itemHPasswd, itemBox);
   GetDialogItemText(itemHPasswd, mySTR[2]);
   {Now you can change the query record or the query itself. }
   { What you change is entirely up to you. In this example, }
   { the qdef changes only the user and password fields }
   { of the query record.}
   query^^.user := mySTR[1];
```

```
      query^^.password := mySTR[2];
      MyQDef := noErr;
   END
   ELSE
      MyQDef := userCanceledErr;
   HUnlock(Handle(query));
   HUnlock(Handle(myResList));
   DisposDialog(myDialog);
END;
```

Data Access Manager Reference

This section describes the data structures, routines, and resources that are specific to the Data Access Manager. The "Data Structures" section shows the data structures for the asynchronous parameter block, the results record, the query record, and the data item record. The "Data Access Manager Routines" section beginning on page 12-60 describes routines for using the high-level and low-level interfaces, including initializing the Data Access Manager, handling query documents and results, controlling sessions, sending and executing queries, retrieving results, and installing and removing result handlers. The "Resources" section beginning on page 12-91 describes the query resource, the query string resource, and the query definition function resource.

Data Structures

This section describes the data structures that you use to provide information to the Data Access Manager or that the Data Access Manager uses to provide information to your application.

You provide a pointer to an asynchronous parameter block as a parameter to the `DBStartQuery`, `DBGetQueryResults`, `DBInit`, `DBEnd`, `DBGetSessionNum`, `DBKill`, `DBSend`, `DBSendItem`, `DBExec`, `DBState`, `DBGetErr`, `DBBreak`, `DBGetItem`, and `DBUnGetItem` functions.

The query record specifies connection information and contains a handle to an array of one or more complete queries or query templates. The `DBGetNewQuery` function returns a handle to a query record, and you provide a handle to a query record as a parameter to the `DBStartQuery` and `DBDisposeQuery` functions.

You use the results record to store the data that was returned by a data source in response to a query. The results record is a parameter to the `DBGetQueryResults` and `DBResultsToText` functions.

The Asynchronous Parameter Block

Each Data Access Manager routine that can be called asynchronously (that is, that can return control to your application before it has completed execution) takes as a parameter a pointer to a parameter block known as the *asynchronous parameter block.* If you specify `NIL` for this parameter, the routine does not return control to your application until it has completed execution.

Note
The asynchronous parameter block is passed on to the database extension, which is responsible for implementing the asynchronous routine. If the database extension does not support asynchronous routines, the Data Access Manager returns the `rcDBAsyncNotSupp` result code and terminates execution of the routine. ◆

The `DBAsyncParamBlockRec` data type defines the asynchronous parameter block.

```
TYPE DBAsyncParamBlockRec =
   RECORD
      completionProc:   ProcPtr; {pointer to completion routine}
      result:           OSErr;   {result of call}
      userRef:          LongInt; {reserved for use by application}
      ddevRef:          LongInt; {reserved for use by database }
                                 { extension}
      reserved:         LongInt; {reserved for use by Data }
                                 { Access Manager}
   END;

   DBAsyncParmBlkPtr = ^DBAsyncParamBlockRec;
```

Field descriptions

`completionProc`
Points to a completion routine that the database extension calls when it has completed executing the asynchronous function. Before calling the completion routine, the Data Access Manager places a pointer to the asynchronous parameter block in the A0 register. If you do not want to use a completion routine, set this parameter to `NIL`.

`result`
Returns the result code for the called routine. The database extension sets this field to 1 while the routine is executing and places the result code in it when the routine completes. Your application can poll this field to determine when an asynchronous routine has completed execution.

`userRef`
Reserved for the application's use. Because the Data Access Manager passes a pointer to the parameter block to the completion routine, you can use this field to pass information to the completion routine.

ddevRef	Reserved for use by the database extension.
reserved	Reserved for use by the Data Access Manager.

The Query Record

The DBGetNewQuery function converts a 'qrsc' resource in a query document into a query record in memory and returns a handle to the query record. The query record specifies connection information and also contains a handle to an array of queries; each query can be either a complete query or a template for a query. The DBGetNewQuery function creates the queries from the 'wstr' resources stored in the query document.

The QueryRecord data type defines a query record.

```
TYPE QueryRecord =
   RECORD
      version:    Integer;          {query record format version}
      id:         Integer;          {resource ID of 'qrsc'}
      queryProc:  Handle;           {handle to qdef}
      ddevName:   Str63;            {name of database extension}
      host:       Str255;           {name of host computer}
      user:       Str255;           {name of user}
      password:   Str255;           {user's password}
      connStr:    Str255;           {connection string}
      currQuery:  Integer;          {index of current query}
      numQueries: Integer;          {number of queries in list}
      queryList:  QueryListHandle;  {handle to array of }
                                    { handles to text}
      numRes:     Integer;          {number of resources in list}
      resList:    ResListHandle;    {handle to array of resource }
                                    { list elements}
      dataHandle: Handle;           {handle to memory for qdef}
      refCon:     LongInt;          {reserved for use by app}
   END;
QueryPtr    = ^QueryRecord;         {pointer to query record}
QueryHandle = ^QueryPtr;            {handle to query record}
```

Field descriptions

version	The version number of the query record format. For the Data Access Manager released with System 7, the version number is 0.
id	The resource ID of the 'qrsc' resource from which the Data Access Manager created this query record.
queryProc	A handle to the query definition function that the DBStartQuery function calls. This handle is NIL if there is no query definition function—that is, if the DBStartQuery function should send the query specified by this query record to the data server without modifications.

ddevName	The database extension name used as a parameter to the DBInit function.
host	The name of the host computer system used as a parameter to the DBInit function.
user	The name of the user, used as a parameter to the DBInit function.
password	The user's password, used as a parameter to the DBInit function.
connStr	The connection string used as a parameter to the DBInit function.
currQuery	An index value from 1 through numQueries, indicating which element in the array of query handles represents the current query. The current query is the one actually sent to the data server. If the query document contains more than one 'wstr' resource, the query definition function can prompt the user to select a new current query and modify this field in the query record appropriately.
numQueries	The number of queries referred to by the queryList field.
queryList	A handle to an array of handles. Each handle in this array refers to a query. Each query is created from a 'wstr' resource in the query document and is stored in memory as a 2-byte length field followed by ASCII text. (The length does not include the 2 bytes of the length field.) The query definition function can create a new query. To add a new handle to the array of handles, use the Memory Manager's SetHandleSize function to increase the size of the array. Don't forget to change the value of the numQueries field as well.
numRes	The number of resources referred to by the resList field.
resList	A handle to an array of records of type ResListElem. Each record in the array contains the type and ID of a resource that is needed by the query definition function.

```
TYPE ResListElem =
   RECORD
      theType: ResType;    {resource type}
      id:      Integer;    {resource ID}
   END;
```

dataHandle	A handle to memory for use by the query definition function. When the Data Access Manager first creates the query record, it sets this field to NIL. The query definition function can allocate memory and place a handle to it in this field. The query definition function should dispose of any memory it allocates before it returns control to the Data Access Manager.
refCon	The query record's reference value. The application can use this field for any purpose.

The Results Record

The results record describes the data that was returned by a data source in response to a query. To get the results of a query, allocate a results record and pass this record to the `DBGetQueryResults` function. The Data Access Manager allocates the handles inside the results record. When your application is finished using the results record, you must deallocate both the results record and the handles inside the results record.

The results record is defined by the `ResultsRecord` data type.

```
TYPE ResultsRecord =
   RECORD
      numRows:    Integer;          {number of rows retrieved}
      numCols:    Integer;          {number of columns per row}
      colTypes:   ColTypesHandle;   {type of data in each column}
      colData:    Handle;           {array of data items}
      colInfo:    ColInfoHandle;    {info about each data item}
   END;
```

Field descriptions

`numRows` The total number of rows retrieved. If the `DBGetQueryResults` function returns a result code other than `rcDBValue`, then not all of the data actually returned by the data source was retrieved. This could happen, for instance, if the user's computer does not have sufficient memory space to hold all the data. In this case, your application can make more space available (by writing the data in the data record to disk, for example) and then call the `DBGetQueryResults` function again to complete retrieval of the data.

Note

The `DBGetQueryResults` function retrieves whole rows only; if it runs out of space in the middle of a row, it stores the partial row in a private buffer so that the data in the results record ends with the last complete row. Because the last partial row is no longer available from the data server, you cannot start to retrieve data with the `DBGetQueryResults` function and then switch to the `DBGetItem` function to complete the data retrieval. ◆

`numCols` The number of columns in each row of data.

`colTypes` A handle to an array of data types, specifying the type of data in each column. The number of elements in the array is equal to the value in the `numCols` field. Table 12-1 beginning on page 12-39 shows the standard data types.

`colData` A handle to the data retrieved by the `DBGetQueryResults` function.

`colInfo` A handle to an array of records of type `DBColInfoRecord`, each of which specifies the length, places, and flags for a data item. There are as many records in the array as there are data items retrieved by the `DBGetQueryResults` function. Here is the `DBColInfoRecord` type definition:

```
TYPE DBColInfoRecord =
   RECORD
      len:     Integer;    {length of data item}
      places:  Integer;    {places for decimal }
                           { and money data items}
      flags:   Integer;    {flags for data item}
   END;
```

The `len` field indicates the length of the data item. The `DBGetQueryResults` function returns a value in this field only for those data types that do not have implied lengths; see Table 12-1 on page 12-39 for a list of these data types.

The `places` field indicates the number of decimal places in data items of types `typeMoney` and `typeDecimal`. For all other data types, the `places` field returns 0.

The least significant bit of the `flags` field is set to 1 if the data item is in the last column of the row. The third bit of the `flags` field is 1 if the data item is `NULL`. You can use the constants `kDBLastColFlag` and `kDBNullFlag` to test for these flag bits.

Data Access Manager Routines

The Data Access Manager has high-level routines, low-level routines, and routines that manipulate result handlers. This section describes all of the Data Access Manager routines.

All of the low-level routines and some of the high-level routines accept a pointer to an asynchronous parameter block as a parameter. For these routines, see "The Asynchronous Parameter Block" beginning on page 12-56 for a description of the fields in the parameter block.

If you specify a nonzero value for the pointer to the asynchronous parameter block, the database extension executes the function asynchronously—that is, it returns control to the Data Access Manager before the routine has completed execution, and the Data Access Manager returns control to your application. If you specify `NIL` for this parameter, the database extension does not return control to your application until the routine has finished execution. Your application must call the Event Manager's `WaitNextEvent` function periodically to allow an asynchronous routine to complete execution. The `WaitNextEvent` function is described in the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials.*

You can tell when an asynchronous routine has completed execution and check the result code by looking at values in the asynchronous parameter block. You can use the `DBKill` function to cancel an asynchronous routine.

Note
A `noErr` result code returned by a routine that has been called asynchronously indicates only that the routine *began* execution successfully. You must check the `result` field of the asynchronous parameter block for the final result of the routine. ◆

Assembly-Language Note
You can invoke each of the Data Access Manager routines with a macro that has the same name as the routine, but preceded with an underscore; for example, the macro for the `DBInit` function is named `_DBInit`. Each of these macros places a routine selector in the D0 register and calls the trap `_Pack13`. The routine selectors are listed in each routine description and in "Assembly-Language Summary" beginning on page 12-104. ◆

Initializing the Data Access Manager

You must initialize the Data Access Manager before you can use it.

InitDBPack

Use the `InitDBPack` function to initialize the Data Access Manager.

```
FUNCTION InitDBPack: OSErr;
```

DESCRIPTION

The `InitDBPack` function initializes the Data Access Manager. You must call the `InitDBPack` function before you call any other Data Access Manager routines. If the Data Access Manager has already been initialized, the `InitDBPack` function returns the `noErr` result code but does nothing else.

The interface routine that implements the `InitDBPack` function includes a version number for the Data Access Manager. If the Data Access Manager is a different version from that specified by the interface routine, then the `InitDBPack` function returns the `rcDBWrongVersion` result code.

SPECIAL CONSIDERATIONS

The `InitDBPack` function may move or purge memory. You should not call this routine from within an interrupt, such as in a completion routine or a VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `InitDBPack` function are

Trap macro	Selector
`_InitDBPack`	`$0100`

RESULT CODES

`noErr`	0	No error
`rcDBWrongVersion`	–812	Wrong version number

High-Level Interface: Handling Query Documents

The high-level interface to the Data Access Manager allows applications to manipulate query documents and to get the results of the query provided by a query document. The use and contents of query documents are discussed in "Creating a Query Document" beginning on page 12-47. The routines described in this section create query records, dispose of query records, and use query documents to establish communication with and send queries to a data server. For a general discussion of the high-level interface, see "The High-Level Interface" beginning on page 12-7 . For instructions on using the high-level interface, refer to "Using the High-Level Interface" beginning on page 12-14.

DBGetNewQuery

You can use the `DBGetNewQuery` function to create a query record.

```
FUNCTION DBGetNewQuery (queryID: Integer;
                        VAR query: QueryHandle): OSErr;
```

`queryID` The resource ID of a `'qrsc'` resource.

`query` Returns a handle to the query record.

DESCRIPTION

The `DBGetNewQuery` function creates a query record from the specified `'qrsc'` resource. The resource file that contains the `'qrsc'` resource must remain open until after the `DBStartQuery` function has completed execution. If you do not already know the resource ID of the `'qrsc'` resource (for example, if you call the `StandardGetFile` procedure to let the user select the query document), you can use Resource Manager routines to determine the resource ID.

SPECIAL CONSIDERATIONS

The `DBGetNewQuery` function may move or purge memory. You should not call this routine from within an interrupt, such as in a completion routine or a VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `DBGetNewQuery` function are

Trap macro	Selector
`_DBGetNewQuery`	`$030F`

RESULT CODES

`noErr`	0	No error
`rcDBPackNotInited`	–813	The `InitDBPack` function has not yet been called

SEE ALSO

See Listing 12-1 beginning on page 12-18 for an example of the use of the `DBGetNewQuery` function. For a description of the query record, see page 12-57. For a description of the `'qrsc'` resource, see "The Query Resource" beginning on page 12-91. The `StandardGetFile` procedure is described in the chapter "Standard File Package" in *Inside Macintosh: Files*, and Resource Manager routines are described in the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox*.

DBDisposeQuery

When you are finished using a query record, call `DBDisposeQuery` to dispose of the query record.

```
FUNCTION DBDisposeQuery (query: QueryHandle): OSErr;
```

`query` A handle to the query record to dispose.

DESCRIPTION

The `DBDisposeQuery` function disposes of a query record and frees all the memory that the Data Access Manager allocated when it created the query record.

SPECIAL CONSIDERATIONS

The `DBDisposeQuery` function may move or purge memory. You should not call this routine from within an interrupt, such as in a completion routine or a VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `DBDisposeQuery` function are

Trap macro	Selector
`_DBDisposeQuery`	`$0210`

RESULT CODES

`noErr`	0	No error
`rcDBPackNotInited`	–813	The `InitDBPack` function has not yet been called

SEE ALSO

See Listing 12-1 beginning on page 12-18 for an example of the use of the `DBDisposeQuery` function in the high-level interface. For a description of the query record, see page 12-57.

DBStartQuery

Use the `DBStartQuery` function to initiate the process of sending a query to a data server.

```
FUNCTION DBStartQuery (VAR sessID: LongInt; query: QueryHandle;
                       statusProc: ProcPtr;
                       asyncPB: DBAsyncParmBlkPtr): OSErr;
```

`sessID` A session ID that identifies a session with the data server. If you specify 0 for this parameter, then the `DBStartQuery` function initiates a session and returns the session ID in the `sessID` parameter.

`query` A handle to a query record.

`statusProc`

A pointer to a status routine that your application can use to update its windows after the query definition function has completed execution. (The `DBStartQuery` function does not attempt to update your application's windows.) The `DBStartQuery` function also calls your status routine before it initiates a session with a data server, after it calls the `DBInit` function, after it calls the `DBSend` function, and after it calls the `DBExec` function. Status routines are discussed in "Writing a Status Routine for High-Level Functions" beginning on page 12-22.

`asyncPB` A pointer to an asynchronous parameter block. When specified, the `DBStartQuery` function calls the `DBInit`, `DBSend`, and `DBExec` functions asynchronously. As soon as the `DBInit` function has started execution, it returns control to your application. Your application must then call the Event Manager's `WaitNextEvent` function periodically to

allow these asynchronous routines to run, and it must check the `result` field of the asynchronous parameter block to determine when each routine has completed execution.

DESCRIPTION

The `DBStartQuery` function performs the following tasks, in the order specified:

1. It calls the query definition function (if any) pointed to by the query record. The query definition function modifies the query record and the query, usually by asking the user for input. The query definition function can display a dialog box that gives the user the option of canceling the query; if the user does cancel the query, the `DBStartQuery` function returns the `userCanceledErr` result code.
2. If you specify a nonzero value for the `statusProc` parameter, the `DBStartQuery` function calls your status routine with the `kDBUpdateWind` constant in the `message` parameter so that your application can update its windows.
3. If you specify a nonzero value for the `statusProc` parameter, the `DBStartQuery` function calls your status routine with the `kDBAboutToInit` constant in the `message` parameter so that your application can display a dialog box informing the user that a session is about to be initiated with a data server, and giving the user the option of canceling execution of the function.
4. If the `sessID` parameter is 0, the `DBStartQuery` function calls the `DBInit` function to initiate a session, and returns a session ID.
5. If you specify a nonzero value for the `statusProc` parameter and the `DBStartQuery` function calls the `DBInit` function, the `DBStartQuery` function calls your status routine with the `kDBInitComplete` constant in the `message` parameter and the result of the `DBInit` function in the function result.
6. The `DBStartQuery` function calls the `DBSend` function to send the query to the data server.
7. If you specify a nonzero value for the `statusProc` parameter, the `DBStartQuery` function calls your status routine with the `kDBSendComplete` constant in the `message` parameter and the result of the `DBSend` function in the `result` parameter.
8. The `DBStartQuery` function calls the `DBExec` function to execute the query.
9. If you specify a nonzero value for the `statusProc` parameter, the `DBStartQuery` function calls your status routine with the `kDBExecComplete` constant in the `message` parameter and the result of the `DBExec` function in the `result` parameter.
10. If you specify a nonzero value for the `statusProc` parameter, the `DBStartQuery` function calls your status routine with the `kDBStartQueryComplete` constant in the `message` parameter and the result of the `DBStartQuery` function in the `result` parameter.

SPECIAL CONSIDERATIONS

The `DBStartQuery` function may move or purge memory. You should not call this routine from within an interrupt, such as in a completion routine or a VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `DBStartQuery` function are

Trap macro	Selector
`_DBStartQuery`	`$0811`

RESULT CODES

`noErr`	0	No error
`userCanceledErr`	–128	User canceled the query
`rcDBError`	–802	Error initiating session, sending text, or executing query
`rcDBBadSessID`	–806	Session ID is invalid
`rcDBBadDDev`	–808	Couldn't find the specified database extension, or error occurred in opening database extension
`rcDBAsyncNotSupp`	–809	The database extension does not support asynchronous calls
`rcDBPackNotInited`	–813	The `InitDBPack` function has not yet been called

SEE ALSO

See "Using the High-Level Interface" beginning on page 12-14 for a general description of how the `DBStartQuery` function works in conjunction with other Data Access Manager routines. See Listing 12-1 beginning on page 12-18 for an example of the use of the `DBStartQuery` function. For a description of the query record, see page 12-57. For information on how to write a query definition function or status routine, see "Writing a Query Definition Function" beginning on page 12-52 and "Writing a Status Routine for High-Level Functions" beginning on page 12-22, respectively. Descriptions of the `DBInit`, `DBSend`, and `DBExec` functions begin on page 12-69, page 12-77, and page 12-79, respectively.

High-Level Interface: Handling Query Results

The high-level interface to the Data Access Manager allows applications to manipulate query documents and to get the results of the query provided by a query document. The high-level routines in this section retrieve query results and convert them to text.

DBGetQueryResults

You can use the `DBGetQueryResults` function to retrieve the results of a query.

```
FUNCTION DBGetQueryResults (sessID: LongInt;
                            VAR results: ResultsRecord;
                            timeout: LongInt; statusProc: ProcPtr;
                            asyncPB: DBAsyncParmBlkPtr): OSErr;
```

`sessID`	The session ID of the session from which you wish to retrieve results.
`results`	The results record, which contains handles to the retrieved data.
`timeout`	The value that the `DBGetQueryResults` function uses for the `timeout` parameter each time it calls the `DBGetItem` function. The `timeout` parameter specifies the maximum amount of time that the database extension should wait to receive results from the data server before canceling the `DBGetItem` function. Specify the `timeout` parameter in sixtieths of a second. To disable the timeout feature, set the `timeout` parameter to the `kDBWaitForever` constant. Some database extensions ignore the `timeout` parameter when you specify a nonzero value for the `asyncPB` parameter.
`statusProc`	A pointer to a status routine that you provide. The `DBGetQueryResults` function calls your status routine after it calls the `DBGetItem` function to retrieve a data item. When it calls the status routine, the `DBGetQueryResults` function provides the result of the `DBGetItem` function, the data type, the data length, the number of decimal places, the flags associated with the data item, and a pointer to the data item.
`asyncPB`	A pointer to an asynchronous parameter block. If specified, the `DBGetQueryResults` function calls the `DBGetItem` function asynchronously for each data item. As soon as the `DBGetItem` function has started execution, it returns control to your application. Your application must then call the Event Manager's `WaitNextEvent` function periodically to allow this asynchronous routine to run, and it must check the result field of the asynchronous parameter block to determine when the routine has completed execution.

DESCRIPTION

The `DBGetQueryResults` function retrieves the results returned by a query and places them in memory. If there is sufficient memory available, this function retrieves all of the results at once. If the `DBGetQueryResults` function runs out of memory, it places as much data as possible in memory, up to the last whole row. You can then make more memory available and call the `DBGetQueryResults` function again to retrieve more data.

You must allocate the results record and pass this record to the `DBGetQueryResults` function. The Data Access Manager allocates the handles inside the results record. When your application is finished using the results record, you must deallocate both the results record and the handles inside the results record.

The `DBGetQueryResults` function can be used to retrieve the results of any query, not only queries sent and executed by the `DBStartQuery` function.

SPECIAL CONSIDERATIONS

The `DBGetQueryResults` function may move or purge memory. You should not call this routine from within an interrupt, such as in a completion routine or a VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `DBGetQueryResults` function are

Trap macro	Selector
`_DBGetQueryResults`	`$0A12`

RESULT CODES

`noErr`	0	No error
`userCanceledErr`	–128	Function canceled by status routine
`rcDBValue`	–801	Data available
`rcDBError`	–802	Query execution ended in an error
`rcDBBreak`	–804	Function timed out
`rcDBExec`	–805	Query currently executing
`rcDBBadSessID`	–806	Session ID is invalid
`rcDBAsyncNotSupp`	–809	The database extension does not support asynchronous calls
`rcDBPackNotInited`	–813	The `InitDBPack` function has not yet been called

SEE ALSO

See Listing 12-1 beginning on page 12-18 for an example of the use of the `DBGetQueryResults` function. See page 12-56 for a description of the asynchronous parameter block. Descriptions of the `DBStartQuery` and `DBGetItem` functions begin on page 12-64 and page 12-84, respectively. For more information on results records, see "The Results Record" beginning on page 12-59 and "Getting Query Results" beginning on page 12-37. For more information on status routines, see "Writing a Status Routine for High-Level Functions" beginning on page 12-22.

DBResultsToText

After retrieving a results record from `DBGetQueryResults`, you can use the `DBResultsToText` function to convert the returned data to text.

```
FUNCTION DBResultsToText (results: ResultsRecord;
                          VAR theText: Handle): OSErr;
```

`results` The results record returned by the `DBGetQueryResults` function.

`theText` The `DBResultsToText` function returns a handle to the converted text in this parameter. This handle is allocated by the Data Access Manager.

DESCRIPTION

The `DBResultsToText` function calls result handlers to convert to text the data retrieved by the `DBGetQueryResults` function.

SPECIAL CONSIDERATIONS

The `DBResultsToText` function may move or purge memory. You should not call this routine from within an interrupt, such as in a completion routine or a VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `DBResultsToText` function are

Trap macro	Selector
`_DBResultsToText`	`$0413`

RESULT CODES

`noErr`	0	No error
`rcDBPackNotInited`	–813	The `InitDBPack` function has not yet been called

SEE ALSO

See Listing 12-1 begining on page 12-18 for an example of the use of the `DBResultsToText` function. See "Converting Query Results to Text" beginning on page 12-43 for a discussion of result handlers.

Low-Level Interface: Controlling the Session

The low-level interface to the Data Access Manager allows applications to open and close sessions with a data server, send and execute queries, retrieve query results, and obtain information about any current session.

DBInit

Use the `DBInit` function to initiate a session with a data server.

```
FUNCTION DBInit (VAR sessID: LongInt; ddevName: Str63;
                 host: Str255; user: Str255; password: Str255;
                 connStr: Str255;
                 asyncPB: DBAsyncParmBlkPtr): OSErr;
```

`sessID` The `DBInit` function returns the session ID in this parameter. This session ID is unique; no other current session, for any database extension, has the same session ID. You must specify the session ID any time you want to send data to or retrieve data from this session. Depending on the database extension you are using, the `DBInit` function might return a

session ID of 0 if it fails to initiate a session, or it might return a nonzero session ID and a result code other than `noErr`. In the latter case, you can pass the session ID to the `DBGetErr` function to determine the cause of the error.

`ddevName` A string of no more than 63 characters that specifies the name of the database extension. The name of the database extension is contained in the database extension file in a resource of type `'STR '` with a resource ID of 128. For the Data Access Language database extension provided by Apple, for example, this string is "`DAL`".

`host` The name of the host system on which the data server is located. This name depends on the manner in which the database extension initiates communication with the data server and how the system administrator has set up the computer system.

`user` The name of the user.

`password` The password associated with the user name.

`connStr` A string that is passed to the data server, which might pass it on to the database management software on the host computer. This string is necessary in some systems to complete log-on procedures.

`asyncPB` A pointer to an asynchronous parameter block. If you do not want to call the function asynchronously, set this parameter to `NIL`.

DESCRIPTION

You must initiate a session before you call any Data Access Manager function that requires a session ID as an input parameter. If the `DBInit` function returns a nonzero session ID, you must call the `DBEnd` function to terminate the session, even if the `DBInit` function also returns a result code other than `noErr`.

Because the high-level function `DBStartQuery` can call the `DBInit` function, you do not have to call the `DBInit` function if you have called the `DBStartQuery` function.

SPECIAL CONSIDERATIONS

The `DBInit` function may move or purge memory. You should not call this routine from within an interrupt, such as in a completion routine or a VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `DBInit` function are

Trap macro	Selector
`_DBInit`	`$0E02`

RESULT CODES

noErr	0	No error
rcDBError	–802	Error initiating session
rcDBBadDDev	–808	Couldn't find the specified database extension, or error occurred in opening database extension
rcDBAsyncNotSupp	–809	The database extension does not support asynchronous calls
rcDBPackNotInited	–813	The InitDBPack function has not yet been called

SEE ALSO

For a description of the asynchronous parameter block, see page 12-56. See Listing 12-5 beginning on page 12-34 for an example of the use of the DBInit function. See page 12-64 for a description of the DBStartQuery function. The DBEnd function is described next.

DBEnd

You must call the DBEnd function to terminate a session.

```
FUNCTION DBEnd (sessID: LongInt;
                asyncPB: DBAsyncParmBlkPtr): OSErr;
```

sessID The session ID that was returned by the DBInit function.

asyncPB A pointer to an asynchronous parameter block. If you do not want to call the function asynchronously, set this parameter to NIL.

DESCRIPTION

The DBEnd function terminates a session with a data server and terminates the network connection between the application and the host computer.

SPECIAL CONSIDERATIONS

The DBEnd function may move or purge memory. You should not call this routine from within an interrupt, such as in a completion routine or a VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the DBEnd function are

Trap macro	Selector
_DBEnd	$0403

RESULT CODES

`noErr`	0	No error
`rcDBError`	–802	Error ending session
`rcDBBadSessID`	–806	Session ID is invalid
`rcDBAsyncNotSupp`	–809	The database extension does not support asynchronous calls
`rcDBPackNotInited`	–813	The `InitDBPack` function has not yet been called

SEE ALSO

For a description of the asynchronous parameter block, see page 12-56.

DBGetConnInfo

The `DBGetConnInfo` function returns information about the specified session, including

- the version of the database extension
- the name of the host system on which the session is running
- the user name
- the connection string that was used to initiate communication
- the name of the network
- the time at which the session started, in ticks (sixtieths of a second)
- the status of the session

```
FUNCTION DBGetConnInfo (sessID: LongInt; sessNum: Integer;
                        VAR returnedID: LongInt;
                        VAR version: LongInt;
                        VAR ddevName: Str63;
                        VAR host: Str255; VAR user: Str255;
                        VAR network: Str255; VAR connStr: Str255;
                        VAR start: LongInt; VAR state: OSErr;
                        asyncPB: DBAsyncParmBlkPtr): OSErr;
```

`sessID` The session ID that was returned by the `DBInit` function. If you include a nonzero value for the `sessID` parameter when you call the `DBGetConnInfo` function, the function returns the name of the database extension in the `ddevName` parameter. If you use 0 for the `sessID` parameter and specify the database extension and session number instead (in the `ddevName` and `sessNum` parameters), the function returns the session ID in the `returnedID` parameter.

`sessNum` The session number of the session about which you want information. If you specify a nonzero session number, you must also provide the database extension in the `ddevName` parameter.

returnedID
: Returns the session ID if you specify the session number and the database extension.

version
: Returns the version number of the database extension that is currently in use.

ddevName
: A string of no more than 63 characters that specifies the name of the database extension. If you specify 0 for the session ID, you must include the name of the database extension as well as a session number. If you specify a valid session ID, then the DBGetConnInfo function returns the name of the database extension in the ddevName parameter. The name of the database extension is contained in the database extension file in a resource of type 'STR ' with a resource ID of 128. For the Data Access Language database extension provided by Apple, for example, this string is "DAL".

host
: Returns the host string used to initiate communication with the data server.

user
: Returns the user string used to initiate communication with the data server.

network
: Returns the name of the network through which the database extension is communicating with the data server. This parameter is an empty string if you are not communicating through a network.

connStr
: Returns the connection string used to initiate communication with the data server.

start
: Returns the time, in ticks (sixtieths of a second), at which this session was initiated.

state
: Returns one of the following values to provide information about the status of the session:

```
CONST noErr       =    0;  {no error--ready for more }
                           { text}
      rcDBValue   = –801;  {output data available}
      rcDBError   = –802;  {execution ended in an }
                           { error}
      rcDBExec    = –805;  {busy--currently executing }
                           { query}
```

asynchPB
: A pointer to an asynchronous parameter block. If you do not want to call the function asynchronously, set this parameter to NIL.

DESCRIPTION

You can use the DBGetConnInfo function to get information about a particular session, or you can call the function repeatedly, incrementing the session number each time, to get information about all of the sessions associated with a particular database extension.

The `sessID` parameter is the session ID that was returned by the `DBInit` function. The `sessNum` parameter is the session number of the session about which you want information. You can specify either the session ID or the session number when you call the `DBGetConnInfo` function. If you specify the `sessID` parameter, use 0 for the `sessNum` parameter. If you specify the `sessNum` parameter, then use 0 for the `sessID` parameter. If you specify the `sessNum` parameter, you must specify a value for the `ddevName` parameter as well. If you specify the session number and the database extension, then the `DBGetConnInfo` function returns the session ID in the `returnedID` parameter.

SPECIAL CONSIDERATIONS

The `DBGetConnInfo` function may move or purge memory. You should not call this routine from within an interrupt, such as in a completion routine or a VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `DBGetConnInfo` function are

Trap macro	Selector
`_DBGetConnInfo`	`$1704`

RESULT CODES

`noErr`	0	No error
`rcDBBadSessID`	–806	Session ID is invalid or database extension name is invalid
`rcDBBadSessNum`	–807	Invalid session number
`rcDBBadDDev`	–808	Couldn't find the specified database extension, or error occurred in opening database extension
`rcDBAsyncNotSupp`	–809	The database extension does not support asynchronous calls
`rcDBPackNotInited`	–813	The `InitDBPack` function has not yet been called

SEE ALSO

For a description of the asynchronous parameter block, see page 12-56. For more information on the use of the `DBGetConnInfo` function, see "Getting Information About Sessions in Progress" on page 12-36.

DBGetSessionNum

The `DBGetSessionNum` function returns a session number when you specify the session ID.

```
FUNCTION DBGetSessionNum (sessID: LongInt; VAR sessNum: Integer;
                          asyncPB: DBAsyncParmBlkPtr): OSErr;
```

`sessID` The session ID that was returned by the `DBInit` function.

`sessNum` Returns the session number of the session you specify with the `sessID` parameter. The session number is unique for a particular database extension, but the same session number might be in use for different database extensions at the same time.

`asyncPB` A pointer to an asynchronous parameter block. If you do not want to call the function asynchronously, set this parameter to `NIL`.

DESCRIPTION

You can use the `DBGetSessionNum` function to determine the session numbers for the sessions opened by your own application. You might want this information, for example, so you can distinguish your own sessions from those opened by other applications when you use the `DBGetConnInfo` function to get information about all open sessions.

SPECIAL CONSIDERATIONS

The `DBGetSessionNum` function may move or purge memory. You should not call this routine from within an interrupt, such as in a completion routine or a VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `DBGetSessionNum` function are

Trap macro	Selector
`_DBGetSessionNum`	`$0605`

RESULT CODES

`noErr`	0	No error
`rcDBBadSessID`	–806	Session ID is invalid
`rcDBAsyncNotSupp`	–809	The database extension does not support asynchronous calls
`rcDBPackNotInited`	–813	The `InitDBPack` function has not yet been called

SEE ALSO

A description of the asynchronous parameter block structure begins on page 12-56. The `DBInit` function description begins on page 12-69. A description of the `DBGetConnInfo` function begins on page 12-72.

DBKill

Use the `DBKill` function to cancel the execution of an asynchronous routine.

```
FUNCTION DBKill (asyncPB: DBAsyncParmBlkPtr): OSErr;
```

`asyncPB` A pointer to an asynchronous parameter block.

DESCRIPTION

The `DBKill` function cancels the execution of the asynchronous call specified by the `asyncPB` parameter.

SPECIAL CONSIDERATIONS

The `DBKill` function may move or purge memory. You should not call this routine from within an interrupt, such as in a completion routine or a VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `DBKill` function are

Trap macro	Selector
`_DBKill`	`$020E`

RESULT CODES

`noErr`	0	No error
`rcDBError`	–802	Error canceling routine
`rcDBBadAsynchPB`	–810	Invalid parameter block specified
`rcDBPackNotInited`	–813	The `InitDBPack` function has not yet been called

SEE ALSO

For a description of the asynchronous parameter block, see page 12-56.

Low-Level Interface: Sending and Executing Queries

The functions in this section send queries or portions of queries to the data server, execute queries that have been sent, return information about queries that have been sent, and halt execution of queries that are executing.

DBSend

You can use the `DBSend` function to send a query or a portion of a query to a data server.

```
FUNCTION DBSend (sessID: LongInt; text: Ptr; len: Integer;
                 asyncPB: DBAsyncParmBlkPtr): OSErr;
```

`sessID`	The session ID that was returned by the `DBInit` function.
`text`	A pointer to the query or query fragment that you want to send to the data server. The query or query fragment must be a character string.
`len`	The length of the character string. If the `len` parameter has a value of –1, then the character string is assumed to be `NULL` terminated (that is, the string ends with a `NULL` byte); otherwise, the `len` parameter specifies the number of bytes in the string.
`asyncPB`	A pointer to an asynchronous parameter block. If you do not want to call the function asynchronously, set this parameter to `NIL`.

DESCRIPTION

The `DBSend` function sends a query or a portion of a query to the data server. The data server appends this portion of the query to any portion you sent previously. Because the Data Access Manager does not modify the string you send in any way, it does not insert any delimiter between fragments of queries that you send to the data server. If you want a blank or a semicolon to be included between query fragments, or if you want to use return characters to divide the query into lines of text, you must include them in the character string that you send with this function.

The data server does not execute the query until you call the `DBExec` function.

SPECIAL CONSIDERATIONS

The `DBSend` function may move or purge memory. You should not call this routine from within an interrupt, such as in a completion routine or a VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `DBSend` function are

Trap macro	Selector
`_DBSend`	`$0706`

RESULT CODES

`noErr`	0	No error
`rcDBError`	–802	Error trying to send text
`rcDBBadSessID`	–806	Session ID is invalid
`rcDBAsyncNotSupp`	–809	The database extension does not support asynchronous calls
`rcDBPackNotInited`	–813	The `InitDBPack` function has not yet been called

SEE ALSO

For a description of the asynchronous parameter block, see page 12-56. See Listing 12-4 beginning on page 12-32 for an example of the use of the `DBSend` function in sending a query fragment. See page 12-79 for a description of the `DBExec` function.

DBSendItem

You can use the `DBSendItem` function to send to the data server the data that you wish to include in a query.

```
FUNCTION DBSendItem (sessID: LongInt; dataType: DBType;
                     len: Integer; places: Integer;
                     flags: Integer; buffer: Ptr;
                     asyncPB: DBAsyncParmBlkPtr): OSErr;
```

`sessID` The session ID that was returned by the `DBInit` function.

`dataType` The data type for the data item that you are sending to the data server.

`len` The length of the data item that you are sending to the data server. The database extension and data server ignore the `len` parameter if the data type has an implied length.

`places` The number of decimal places for the data item that you are sending to the data server. The database extension and data server ignore the `places` parameter for all values of the `dataType` parameter except `typeDecimal` and `typeMoney`.

`flags` Set the `flags` parameter to 0. There are no flags currently defined for the `DBSendItem` function.

`buffer` A pointer to the memory location of the data item that you want to send. When you use the `DBSendItem` function to send an item of data to a data server, the database extension and data server format the data according to the data type, length, and decimal places you specify, convert it to a character string, and append the data to the query.

`asyncPB` A pointer to an asynchronous parameter block. If you do not want to call the function asynchronously, set this parameter to `NIL`.

DESCRIPTION

The `DBSendItem` function sends a single data item to the data server. The database extension or the data server (depending on how the system is implemented) converts the data item to a character string and appends it to the query, just as the `DBSend` function appends a query program fragment to the query. The query is not executed until you call the `DBExec` function.

SPECIAL CONSIDERATIONS

The `DBSendItem` function may move or purge memory. You should not call this routine from within an interrupt, such as in a completion routine or a VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `DBSendItem` function are

Trap macro	Selector
`_DBSendItem`	`$0B07`

RESULT CODES

`noErr`	0	No error
`rcDBError`	–802	Error trying to send item
`rcDBBadSessID`	–806	Session ID is invalid
`rcDBAsyncNotSupp`	–809	The database extension does not support asynchronous calls
`rcDBPackNotInited`	–813	The `InitDBPack` function has not yet been called

SEE ALSO

For a discussion of data types, see "Getting Query Results" beginning on page 12-37. For a description of the asynchronous parameter block, see page 12-56. See Listing 12-4 beginning on page 12-32 for an example of the use of the `DBSendItem` function in sending a query fragment. See page 12-77 for a description of the `DBSend` function. The `DBExec` function is described next.

DBExec

The `DBExec` function initiates execution of a query that you have sent to a data server.

```
FUNCTION DBExec (sessID: LongInt;
                 asyncPB: DBAsyncParmBlkPtr): OSErr;
```

`sessID` The session ID that was returned by the `DBInit` function.

`asyncPB` A pointer to an asynchronous parameter block. If you do not want to call the function asynchronously, set this parameter to `NIL`.

DESCRIPTION

The `DBExec` function initiates execution of a query that you have sent to a data server. You can use the `DBState` function to determine the status of a query after you have initiated execution.

SPECIAL CONSIDERATIONS

The `DBExec` function may move or purge memory. You should not call this routine from within an interrupt, such as in a completion routine or a VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `DBExec` function are

Trap macro	Selector
`_DBExec`	`$0408`

RESULT CODES

`noErr`	0	No error
`rcDBError`	–802	Error trying to begin execution
`rcDBBadSessID`	–806	Session ID is invalid
`rcDBAsyncNotSupp`	–809	The database extension does not support asynchronous calls
`rcDBPackNotInited`	–813	The `InitDBPack` function has not yet been called

SEE ALSO

For a description of the asynchronous parameter block, see page 12-56. See Listing 12-5 beginning on page 12-34 for an example of the use of the `DBExec` function. Descriptions of the `DBSend` and `DBSendItem` functions begin on page 12-77 and page 12-78, respectively. The `DBState` function is described next.

DBState

You can use the `DBState` function to determine whether the data server has successfully executed a query and whether it has data available for you to retrieve.

```
FUNCTION DBState (sessID: LongInt;
                  asyncPB: DBAsyncParmBlkPtr): OSErr;
```

`sessID` The session ID that was returned by the `DBInit` function.

`asyncPB` A pointer to an asynchronous parameter block. If you do not want to call the function asynchronously, set this parameter to `NIL`.

DESCRIPTION

The `DBState` function returns a result code that indicates the status of the data server.

SPECIAL CONSIDERATIONS

The `DBState` function may move or purge memory. You should not call this routine from within an interrupt, such as in a completion routine or a VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `DBState` function are

Trap macro	Selector
`_DBState`	`$0409`

RESULT CODES

`noErr`	0	No error
`rcDBValue`	–801	Output data available
`rcDBError`	–802	Error executing function
`rcDBExec`	–805	Query currently executing
`rcDBBadSessID`	–806	Session ID is invalid
`rcDBAsyncNotSupp`	–809	The database extension does not support asynchronous calls
`rcDBPackNotInited`	–813	The `InitDBPack` function has not yet been called

SEE ALSO

For a description of the asynchronous parameter block, see page 12-56.

DBGetErr

The `DBGetErr` function retrieves error codes and error messages from a data server. You can use this function to obtain information when a low-level function returns the result code `rcDBError`.

```
FUNCTION DBGetErr (sessID: LongInt; VAR err1: LongInt;
                   VAR err2: LongInt; VAR item1: Str255;
                   VAR item2: Str255; VAR errorMsg: Str255;
                   asyncPB: DBAsyncParmBlkPtr): OSErr;
```

`sessID` The session ID that was returned by the `DBInit` function.

`err1` Returns the primary error code.

`err2` Returns the secondary error code.

`item1` Returns a string that describes the object of the error message.

item2	Returns a string that describes the object of the error message.
errorMsg	Returns the error message.
asyncPB	A pointer to an asynchronous parameter block. If you do not want to call the function asynchronously, set this parameter to NIL.

DESCRIPTION

If the DBState function returns the rcDBError result code, indicating that execution of a query ended in an error, the error information retuned by DBGetErr can help you debug the query. The meaning of each error code and error message returned by this function depends on the data server with which you are communicating; see the documentation for that data server for more information.

SPECIAL CONSIDERATIONS

The DBGetErr function may move or purge memory. You should not call this routine from within an interrupt, such as in a completion routine or a VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the DBGetErr function are

Trap macro	Selector
_DBGetErr	$0E0A

RESULT CODES

noErr	0	No error
rcDBError	–802	Error retrieving error information
rcDBBadSessID	–806	Session ID is invalid
rcDBAsyncNotSupp	–809	The database extension does not support asynchronous calls
rcDBPackNotInited	–813	The InitDBPack function has not yet been called

SEE ALSO

For a description of the asynchronous parameter block, see page 12-56.

DBBreak

You can use the DBBreak function to cancel a query—for example, if you determine that it is taking too long to complete execution.

```
FUNCTION DBBreak (sessID: LongInt; abort: Boolean;
                  asyncPB: DBAsyncParmBlkPtr): OSErr;
```

sessID	The session ID that was returned by the DBInit function.
abort	A Boolean value that indicates how DBBreak should cancel the query. Specify TRUE (nonzero) to cause the data server to halt any query that is executing and terminate the current session. Specify FALSE (0) to cause the data server to halt any query that is executing and reinitialize itself.
asyncPB	A pointer to an asynchronous parameter block. If you do not want to call the function asynchronously, set this parameter to NIL.

DESCRIPTION

The DBBreak function can halt execution of a query and reinitialize the data server, or it can unconditionally terminate a session with a data server.

SPECIAL CONSIDERATIONS

The DBBreak function may move or purge memory. You should not call this routine from within an interrupt, such as in a completion routine or a VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the DBBreak function are

Trap macro	Selector
_DBBreak	$050B

RESULT CODES

noErr	0	No error
rcDBError	–802	Error executing function
rcDBBadSessID	–806	Session ID is invalid
rcDBAsyncNotSupp	–809	The database extension does not support asynchronous calls
rcDBPackNotInited	–813	The InitDBPack function has not yet been called

SEE ALSO

For a description of the asynchronous parameter block, see page 12-56.

Low-Level Interface: Retrieving Results

The functions in this section allow you to retrieve a data item from the data server, to obtain information about the next data item, and to retrieve the same data item more than once.

DBGetItem

After you have executed a query and the `DBState` function returns the `rcDBValue` result code, indicating that data is available, you can use the `DBGetItem` function to retrieve the next data item. You can also use this function to obtain information about the next data item without retrieving the data.

```
FUNCTION DBGetItem (sessID: LongInt; timeout: LongInt;
                    VAR dataType: DBType; VAR len: Integer;
                    VAR places: Integer; VAR flags: Integer;
                    buffer: Ptr; asyncPB: DBAsyncParmBlkPtr)
                    : OSErr;
```

`sessID` The session ID that was returned by the `DBInit` function.

`timeout` The maximum amount of time that the database extension should wait to receive results from the data server before canceling the function. Specify the `timeout` parameter in ticks (sixtieths of a second). To disable the timeout feature, set the `timeout` parameter to the `kDBWaitForever` constant. If the timeout period expires, the `DBGetItem` function returns the `rcDBBreak` result code. The `DBGetItem` function ignores the `timeout` parameter if you call the function asynchronously.

One use for the `timeout` parameter is to call the `DBGetItem` function periodically with a short value set for this parameter in order to return control to your application while a query is executing. Your application can then retrieve the next data item as soon as execution of the query is complete without having to call the `DBState` function to determine when data is available.

`dataType` The data type that you expect the next data item to be. If the item is not of the expected data type, the database extension returns the `rcDBBadType` result code. If you want to retrieve the next data item regardless of type, set the `dataType` parameter to the `typeAnyType` constant. To skip the next data item, set the `dataType` parameter to the `typeDiscard` constant. The data server sets the `dataType` parameter to the actual type of the data item when it retrieves the data item or returns information about the data item.

`len` The length of the data buffer pointed to by the `buffer` parameter. If you use the `DBGetItem` function to obtain information only (by setting the `buffer` parameter to `NIL`), then the data server ignores the `len` parameter. The data server sets the `len` parameter to the actual length of the data item when it retrieves the data item or returns information about the data item.

`places` Returns the number of decimal places in data items of types `typeMoney` and `typeDecimal`. For all other data types, the data server returns 0 for the `places` parameter.

`flags`	If the least significant bit of the `flags` parameter is set to 1, the data item is in the last column of the row. If the third bit of this parameter is set to 1, the data item is `NULL`. You can use the constants `kDBLastColFlag` and `kDBNullFlag` to test for these flag bits.
`buffer`	A pointer to the location where you want the retrieved data item to be stored. You must ensure that the location you specify contains enough space for the data item that will be returned. To determine the data type, length, and number of decimal places of the next data item without retrieving it, specify `NIL` for the `buffer` parameter.
`asyncPB`	A pointer to an asynchronous parameter block. If you do not want to call the function asynchronously, set this parameter to `NIL`.

DESCRIPTION

The `DBGetItem` function retrieves the next data item from the data server. You can repeat the `DBGetItem` function as many times as is necessary to retrieve all of the data returned by the data source in response to a query.

SPECIAL CONSIDERATIONS

The `DBGetItem` function may move or purge memory. You should not call this routine from within an interrupt, such as in a completion routine or a VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `DBGetItem` function are

Trap macro	Selector
`_DBGetItem`	`$100C`

RESULT CODES

`noErr`	0	No error
`rcDBNull`	–800	The data item was `NULL`
`rcDBValue`	–801	Data available was successfully retrieved
`rcDBError`	–802	Error executing function
`rcDBBadType`	–803	Next data item not of requested data type
`rcDBBreak`	–804	Function timed out
`rcDBBadSessID`	–806	Session ID is invalid
`rcDBAsyncNotSupp`	–809	The database extension does not support asynchronous calls
`rcDBPackNotInited`	–813	The `InitDBPack` function has not yet been called

SEE ALSO

For a discussion of data types, see "Getting Query Results" beginning on page 12-37. To retrieve all of a query's data items at once, use the high-level function `DBGetQueryResults`; a description of that function begins on page 12-66. For a description of the asynchronous parameter block, see page 12-56. See Listing 12-5 beginning on page 12-34 for an example that illustrates the use of the `DBGetItem` function.

DBUnGetItem

The `DBUnGetItem` function reverses the effect of the last call to the `DBGetItem` function, in the sense that the next time you call the `DBGetItem` function it retrieves the same item a second time.

```
FUNCTION DBUnGetItem (sessID: LongInt;
                      asyncPB: DBAsyncParmBlkPtr): OSErr;
```

`sessID` The session ID that was returned by the `DBInit` function.

`asyncPB` A pointer to an asynchronous parameter block. If you do not want to call the function asynchronously, set this parameter to `NIL`.

DESCRIPTION

The `DBUnGetItem` function does not remove the just-retrieved data item from the input buffer. This function can reverse the effect of only one call to the `DBGetItem` function; you cannot use it to step back through several previously retrieved data items.

SPECIAL CONSIDERATIONS

The `DBUnGetItem` function may move or purge memory. You should not call this routine from within an interrupt, such as in a completion routine or a VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `DBUnGetItem` function are

Trap macro	Selector
`_DBUnGetItem`	`$040D`

RESULT CODES

`noErr`	0	No error
`rcDBError`	–802	Error executing function
`rcDBBadSessID`	–806	Session ID is invalid
`rcDBAsyncNotSupp`	–809	The database extension does not support asynchronous calls
`rcDBPackNotInited`	–813	The `InitDBPack` function has not yet been called

SEE ALSO

For a description of the asynchronous parameter block, see page 12-56. See page 12-84 for a description of the `DBGetItem` function.

Installing and Removing Result Handlers

The functions in this section install, remove, and return pointers to result handlers.

DBInstallResultHandler

The `DBInstallResultHandler` function installs a result handler for the data type specified by the `dataType` parameter. The result handler is then used by the `DBResultsToText` function to convert data of the specified type into a character string.

```
FUNCTION DBInstallResultHandler (dataType: DBType;
                                 theHandler: ProcPtr;
                                 isSysHandler: Boolean): OSErr;
```

`dataType` The type of result handler to install.

`theHandler`
A pointer to a result handler.

`isSysHandler`
A Boolean value that specifies whether the result handler is an application result handler—to be used only when the `DBResultsToText` function is called by the application that installed the result handler—or a system result handler—to be used by every application running on the system. If the `isSysHandler` parameter is `TRUE`, the result handler is a system result handler.

DESCRIPTION

When you install an application result handler, it replaces any result handler with the same name previously installed by that application. Similarly, when you install a system result handler, it replaces any existing system result handler with the same name. Before you temporarily replace an existing result handler, use the `DBGetResultHandler`

function to obtain a pointer to the present handler, and save the present result handler in your application's private storage. Then you can reinstall the original result handler when you are finished using the temporary one.

Because an application result handler is used in preference to a system result handler if both are available, you can temporarily replace a system result handler for purposes of your application by installing an application result handler for the same data type. You can then use the `DBRemoveResultHandler` function to remove the application result handler and return to using the system result handler whenever you wish.

SPECIAL CONSIDERATIONS

The `DBInstallResultHandler` function may move or purge memory. You should not call this routine from within an interrupt, such as in a completion routine or a VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `DBInstallResultHandler` function are

Trap macro	Selector
`_DBInstallResultHandler`	`$0514`

RESULT CODES

`noErr`	0	No error
`rcDBPackNotInited`	–813	The `InitDBPack` function has not yet been called

SEE ALSO

See page 12-68 for a description of the `DBResultsToText` function. For information on application and system result handlers, see "Converting Query Results to Text" beginning on page 12-43; that section also lists the data types for which Apple provides system result handlers. Listing 12-6 on page 12-46 shows a sample result handler. The `DBRemoveResultHandler` function is described on page 12-90, and the `DBGetResultHandler` function is described next.

DBGetResultHandler

The `DBGetResultHandler` function returns a pointer to a result handler for a specified data type.

```
FUNCTION DBGetResultHandler (dataType: DBType;
                             VAR theHandler: ProcPtr;
                             getSysHandler: Boolean): OSErr;
```

`dataType` The data type for which to install a result handler.

`theHandler`
Returns a pointer to the result handler.

`getSysHandler`
If you set the `getSysHandler` parameter to `FALSE` (`0`), the function returns a pointer to the current application result handler for the specified data type, or it returns `NIL` if there is no application result handler for that data type. If you set the `getSysHandler` parameter to `TRUE` (nonzero), the function returns a pointer to the current system result handler for the specified data type, or it returns `NIL` if there is no system result handler for that data type.

DESCRIPTION

You can use the `DBGetResultHandler` function to obtain a pointer to a result handler so that you can use it to convert to text an individual data item retrieved by the `DBGetItem` function. The `DBGetQueryResults` function automatically converts to text all of the data pointed to by the results record.

SPECIAL CONSIDERATIONS

The `DBGetResultHandler` function may move or purge memory. You should not call this routine from within an interrupt, such as in a completion routine or a VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `DBGetResultHandler` function are

Trap macro	Selector
`_DBGetResultHandler`	`$0516`

RESULT CODES

`noErr`	0	No error
`rcDBNoHandler`	–811	There is no handler for this data type installed for the current application
`rcDBPackNotInited`	–813	The `InitDBPack` function has not yet been called

SEE ALSO

The `DBGetQueryResults` function is described on page 12-66, and the `DBGetItem` function is described on page 12-84. See "Converting Query Results to Text" beginning on page 12-43 for a list of the data types for which Apple provides system result handlers. Listing 12-6 on page 12-46 shows a sample result handler.

DBRemoveResultHandler

You can use the `DBRemoveResultHandler` function to remove an application result handler.

```
FUNCTION DBRemoveResultHandler (dataType: DBType): OSErr;
```

`dataType` The type of result handler to remove.

DESCRIPTION

The `DBRemoveResultHandler` function removes from memory the specified application result handler. This function cannot remove a system result handler.

SPECIAL CONSIDERATIONS

The `DBRemoveResultHandler` function may move or purge memory. You should not call this routine from within an interrupt, such as in a completion routine or a VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `DBRemoveResultHandler` function are

Trap macro	Selector
`_DBRemoveResultHandler`	`$0215`

RESULT CODES

`noErr`	0	No error
`rcDBNoHandler`	–811	There is no handler for this data type installed for the current application
`rcDBPackNotInited`	–813	The `InitDBPack` function has not yet been called

SEE ALSO

For a discussion of result handlers, see "Converting Query Results to Text" beginning on page 12-43.

Application-Defined Routines

You can provide status functions, result handler functions, and query defintion functions for use with the Data Access Manager. For information on status functions, see "Writing a Status Routine for High-Level Functions" beginning on page 12-22. See "Processing Query Results" beginning on page 12-37 for information on result handlers. See "Writing a Query Definition Function" beginning on page 12-52 for information on query definition functions.

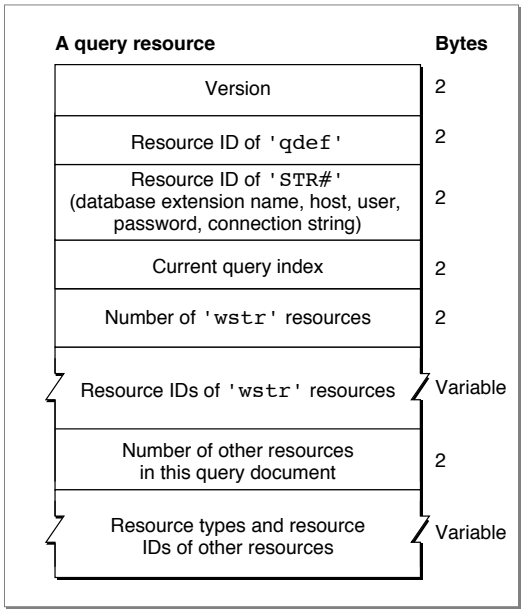
Resources

This section describes the query resource, the query string resource, and the query definition function resource. You use the query resource to provide information that the Data Access Manager uses to create a query record in memory. You use the query string resource to define individual queries. You use the query definition function to modify a query and the query record before the query is sent to the data server.

The Query Resource

Each query document should contain a single `'qrsc'` resource. Figure 12-9 shows the format of the `'qrsc'` resource.

Figure 12-9 Structure of a compiled query (`'qrsc'`) resource



A `'qrsc'` resource contains these elements:

- The version number of the `'qrsc'` format. For the Data Access Manager released with System 7, the version number is 0.
- The resource ID of the `'qdef'` resource containing the query definition function that the Data Access Manager is to call when it opens this `'qrsc'` resource. Use an ID of 0 if there is no query definition function for this resource—that is, if the Data Access Manager should send the query in this resource to the data server without modifications.

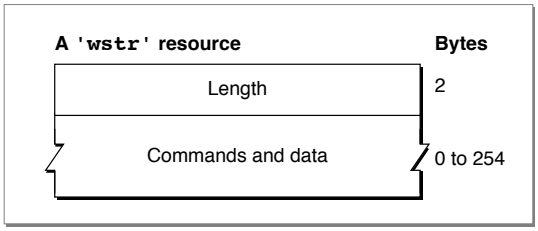
- The resource ID of an 'STR#' resource that contains five Pascal strings corresponding to some of the parameters used by the DBInit function. If the query definition function is going to prompt the user for the values of these parameters before entering them in the query record, they should be zero-length strings in the 'STR#' resource.
- An index value indicating which element in the array of 'wstr' IDs represents the current query. The current query is the one actually sent to the data server.
- The number of 'wstr' resources in the query document.
- An array of resource IDs of the 'wstr' resources in the query document. (The array elements are numbered starting with 1.) If the query document contains more than one 'wstr' resource, the query definition function can prompt the user to select the query to use and modify the current query field in the query record appropriately.
- The number of other resources in this query document.
- An array listing the resource types and IDs of all the resources in the query document other than the standard resources included in all query documents. The resources listed in this final array are those used by the query definition function. This list should include resources embedded in other resources, such as a 'PICT' resource that is included in a 'DITL' resource.

The Query String Resource

A query document must contain one or more query string resources of type 'wstr'. These 'wstr' resources contain individual queries—that is, strings of commands and data that the DBSend function sends to the data server and that the DBExec function executes.

A 'wstr' resource consists of a 2-byte length field followed by a character string. (The *w* in 'wstr' refers to the length word as opposed to the length byte used in an 'STR ' resource.) Each 'wstr' resource contains one query (or one query template, to be modified by the query definition function before it is sent to the data server). Figure 12-10 shows the structure of the 'wstr' resource.

Figure 12-10 Structure of a compiled query string ('wstr') resource



The 'qrsc' resource includes an array that lists the resource ID numbers of all of the 'wstr' resources in the query document and an index into the array that specifies which one of the 'wstr' resources should be sent to the data server.

The Query Definition Function Resource

A query document may contain a query definition function, which can modify the query record and, if necessary, fill in the query template to create a complete query.

If you want to include a query definition function, you must make it the first piece of code in a resource of type `'qdef'` in the query document.

Note that, because a query in memory consists only of a 2-byte length value followed by a character string, the query definition function has to know the exact contents and structure of a query in order to modify it. For a sample query definition function that uses a dialog box to prompt the user for a user name and password, see Listing 12-7 on page 12-53.

Summary of the Data Access Manager

Pascal Summary

Constants

```
CONST
      gestaltDBAccessMgrAttr     = 'dbac';    {Gestalt selector for }
                                              { Data Access Manager}
      {Gestalt selector response}
      gestaltDBAccessMgrPresent  = 0;         {TRUE if Data Access Manager }
                                              { is present}
      {DBStartQuery status messages}
      kDBUpdateWind              = 0;         {update windows}
      kDBAboutToInit             = 1;         {about to call DBInit}
      kDBInitComplete            = 2;         {DBInit has completed}
      kDBSendComplete            = 3;         {DBSend has completed}
      kDBExecComplete            = 4;         {DBExec has completed}
      kDBStartQueryComplete      = 5;         {DBStartQuery is about }
                                              { to complete}
      {DBGetQueryResults status messages}
      kDBGetItemComplete         = 6;         {DBGetItem has completed}
      kDBGetQueryResultsComplete = 7;         {DBGetQueryResults has }
                                              { completed data types}
      {data type codes}
      typeNone                   = 'none';    {no more data expected}
      typeDate                   = 'date';    {date}
      typeTime                   = 'time';    {time}
      typeTimeStamp              = 'tims';    {date and time}
      typeDecimal                = 'deci';    {decimal number}
      typeMoney                  = 'mone';    {money value}
      typeVChar                  = 'vcha';    {variable character}
      typeVBin                   = 'vbin';    {variable binary}
      typeLChar                  = 'lcha';    {long character}
      typeLBin                   = 'lbin';    {long binary}
      typeDiscard                = 'disc';    {discard next data item}
      typeBoolean                = 'bool';    {Boolean}
      typeChar                   = 'TEXT';    {character}
```

```
    typeSMInt                   = 'shor';   {short integer}
    typeInteger                 = 'long';   {integer}
    typeSMFloat                 = 'sing';   {short floating point}
    typeFloat                   = 'doub';   {floating point}
    {dummy data types for DBResultsToText}
    typeUnknown                 = 'unkn';   {result handler for unknown }
                                            { data type}
    typeColBreak                = 'colb';   {result handler for column }
                                            { break}
    typeRowBreak                = 'rowb';   {result handler for end of }
                                            { line}
    {any data type in DBGetItem}
    typeAnyType                 = 0;        {any data type}
    {infinite timeout value for DBGetItem}
    kDBWaitForever              = -1;       {infinite timeout value for }
                                            { DBGetItem}
    {flags for DBGetItem}
    kDBLastColFlag              = $0001;    {data item is last column }
                                            { of the row}
    kDBNullFlag                 = $0004;    {data item is NULL}
```

Data Types

```
TYPE DBType =        OSType;                {data type}

   DBAsyncParamBlockRec =                   {asynchronous parameter block}
   RECORD
      completionProc:   ProcPtr;            {pointer to completion routine}
      result:           OSErr;              {result of call}
      userRef:          LongInt;            {reserved for use by }
                                            { application}
      ddevRef:          LongInt;            {reserved for use by database }
                                            { extension}
      reserved:         LongInt;            {reserved for use by }
                                            { Data Access Mgr}
   END;
   DBAsyncParmBlkPtr = ^DBAsyncParamBlockRec;

   ResListElem =                            {resource list in QueryRecord}
   RECORD
      theType:          ResType;            {resource type}
      id:               Integer;            {resource ID}
   END;
```

```
ResListArray   = ARRAY[0..255] OF ResListElem;
ResListPtr     = ^ResListArray;
ResListHandle  = ^ResListPtr;

QueryRecord =
RECORD
   version:       Integer;          {query record format version}
   id:            Integer;          {resource ID of 'qrsc'}
   queryProc:     Handle;           {handle to query def proc}
   ddevName:      Str63;            {name of database extension}
   host:          Str255;           {name of host computer}
   user:          Str255;           {name of user}
   password:      Str255;           {user's password}
   connStr:       Str255;           {connection string}
   currQuery:     Integer;          {index of current query}
   numQueries:    Integer;          {number of queries in list}
   queryList:     QueryListHandle;  {handle to array of handles to text}
   numRes:        Integer;          {number of resources in list}
   resList:       ResListHandle;    {handle to array of resource list }
                                     { elements}
   dataHandle:    Handle;           {handle to memory for query def proc}
   refCon:        LongInt;          {reserved for use by application}
END;
QueryPtr    = ^QueryRecord;         {pointer to query record}
QueryHandle = ^QueryPtr;            {handle to query record}
{query list in QueryRecord}
QueryArray        = ARRAY[0..255] OF Handle;
QueryListPtr      = ^QueryArray;
QueryListHandle   = ^QueryListPtr;

{column types array in ResultsRecord}
ColTypesArray  = ARRAY[0..255] OF DBType;
ColTypesPtr    = ^ColTypesArray;
ColTypesHandle = ^ColTypesPtr;

DBColInfoRecord =                   {column info in ResultsRecord}
RECORD
   len:        Integer;             {length of data item}
   places:     Integer;             {places for decimal and money }
                                    { data items}
   flags:      Integer;             {flags for data item}
END;
```

```
ColInfoArray   = ARRAY[0..255] OF DBColInfoRecord;
ColInfoPtr     = ^ColInfoArray;
ColInfoHandle  = ^ColInfoPtr;

{structure of results returned by DBGetResults}
ResultsRecord =
RECORD
   numRows:       Integer;          {number of rows retrieved}
   numCols:       Integer;          {number of columns per row}
   colTypes:      ColTypesHandle;   {type of data in each column}
   colData:       Handle;           {array of data items}
   colInfo:       ColInfoHandle;    {DBColInfoRecord array--info about }
                                    { each data item}
END;
```

Data Access Manager Routines

Initializing the Data Access Manager

```
FUNCTION InitDBPack:        OSErr;
```

High-Level Interface: Handling Query Documents

```
FUNCTION DBGetNewQuery      (queryID: Integer; VAR query: QueryHandle)
                             : OSErr;
FUNCTION DBDisposeQuery     (query: QueryHandle): OSErr;
FUNCTION DBStartQuery       (VAR sessID: LongInt; query: QueryHandle;
                             statusProc: ProcPtr;
                             asyncPB: DBAsyncParmBlkPtr): OSErr;
```

High-Level Interface: Handling Query Results

```
FUNCTION DBGetQueryResults  (sessID: LongInt; VAR results: ResultsRecord;
                             timeout: LongInt; statusProc: ProcPtr;
                             asyncPB: DBAsyncParmBlkPtr): OSErr;
FUNCTION DBResultsToText    (results: ResultsRecord; VAR theText: Handle)
                             : OSErr;
```

Low-Level Interface: Controlling the Session

```
FUNCTION DBInit             (VAR sessID: LongInt; ddevName: Str63;
                             host: Str255; user: Str255; password: Str255;
                             connStr: Str255; asyncPB: DBAsyncParmBlkPtr)
                             : OSErr;
```

```
FUNCTION DBEnd              (sessID: LongInt;
                             asyncPB: DBAsyncParmBlkPtr): OSErr;
FUNCTION DBGetConnInfo      (sessID: LongInt; sessNum: Integer;
                             VAR returnedID: LongInt; VAR version: LongInt;
                             VAR ddevName: Str63; VAR host: Str255;
                             VAR user: Str255; VAR network: Str255;
                             VAR connStr: Str255; VAR start: LongInt;
                             VAR state: OSErr; asyncPB: DBAsyncParmBlkPtr)
                             : OSErr;
FUNCTION DBGetSessionNum    (sessID: LongInt; VAR sessNum: Integer;
                             asyncPB: DBAsyncParmBlkPtr): OSErr;
FUNCTION DBKill             (asyncPB: DBAsyncParmBlkPtr): OSErr;
```

Low-Level Interface: Sending and Executing Queries

```
FUNCTION DBSend             (sessID: LongInt; text: Ptr; len: Integer;
                             asyncPB: DBAsyncParmBlkPtr): OSErr;
FUNCTION DBSendItem         (sessID: LongInt; dataType: DBType;
                             len: Integer; places: Integer; flags: Integer;
                             buffer: Ptr; asyncPB: DBAsyncParmBlkPtr)
                             : OSErr;
FUNCTION DBExec             (sessID: LongInt; asyncPB: DBAsyncParmBlkPtr)
                             : OSErr;
FUNCTION DBState            (sessID: LongInt; asyncPB: DBAsyncParmBlkPtr)
                             : OSErr;
FUNCTION DBGetErr           (sessID: LongInt; VAR err1: LongInt;
                             VAR err2: LongInt; VAR item1: Str255;
                             VAR item2: Str255; VAR errorMsg: Str255;
                             asyncPB: DBAsyncParmBlkPtr): OSErr;
FUNCTION DBBreak            (sessID: LongInt; abort: Boolean;
                             asyncPB: DBAsyncParmBlkPtr): OSErr;
```

Low-Level Interface: Retrieving Results

```
FUNCTION DBGetItem          (sessID: LongInt; timeout: LongInt;
                             VAR dataType: DBType;
                             VAR len: Integer; VAR places: Integer;
                             VAR flags: Integer; buffer: Ptr;
                             asyncPB: DBAsyncParmBlkPtr): OSErr;
FUNCTION DBUnGetItem        (sessID: LongInt;
                             asyncPB: DBAsyncParmBlkPtr): OSErr;
```

Installing and Removing Result Handlers

```
FUNCTION DBInstallResultHandler
                            (dataType: DBType; theHandler: ProcPtr;
                             isSysHandler: Boolean): OSErr;
FUNCTION DBGetResultHandler (dataType: DBType; VAR theHandler: ProcPtr;
                             getSysHandler: Boolean): OSErr;
FUNCTION DBRemoveResultHandler
                            (dataType: DBType): OSErr;
```

Application-Defined Routines

```
FUNCTION MyStatusFunc       (message: Integer; result: OSErr;
                             dataLen: Integer; dataPlaces: Integer;
                             dataFlags: Integer; dataType: DBType;
                             dataPtr: Ptr): Boolean;
FUNCTION MyResultHandler    (dataType: DBType; theLen: Integer;
                             thePlaces: Integer; theFlags: Integer;
                             theData: Ptr; theText: Handle): OSErr;
FUNCTION MyQDef             (VAR sessID: LongInt;
                             query: QueryHandle): OSErr;
```

C Summary

Constants

```
enum {
   #define gestaltDBAccessMgrAttr   'dbac'   /*Gestalt selector for */
                                             /* Data Access Manager*/
   /*Gestalt selector response*/
   gestaltDBAccessMgrPresent     = 0         /*TRUE if Data Access Manager */
                                             /* is present*/
};
enum {                                       /*DBStartQuery status messages*/
   kDBUpdateWind                 = 0,        /*update windows*/
   kDBAboutToInit                = 1,        /*about to call DBInit*/
   kDBInitComplete               = 2,        /*DBInit has completed*/
   kDBSendComplete               = 3,        /*DBSend has completed*/
   kDBExecComplete               = 4,        /*DBExec has completed*/
   kDBStartQueryComplete         = 5         /*DBStartQuery is about */
                                             /* to complete*/
};
```

```
enum {
   /*DBGetQueryResults status messages*/
   kDBGetItemComplete            = 6,     /*DBGetItem has completed*/
   kDBGetQueryResultsComplete    = 7,     /*DBGetQueryResults has */
                                          /* completed data types*/
   /*data type codes*/
   #define typeNone              'none'   /*no more data expected*/
   #define typeDate              'date'   /*date*/
   #define typeTime              'time'   /*time*/
   #define typeTimeStamp         'tims'   /*date and time*/
   #define typeDecimal           'deci'   /*decimal number*/
   #define typeMoney             'mone'   /*money value*/
   #define typeVChar             'vcha'   /*variable character*/
   #define typeVBin              'vbin'   /*variable binary*/
   #define typeLChar             'lcha'   /*long character*/
   #define typeLBin              'lbin'   /*long binary*/
   #define typeDiscard           'disc'   /*discard next data item*/
   /*dummy data types for DBResultsToText*/
   #define typeUnknown           'unkn'   /*result handler for unknown */
                                          /* data type*/
   #define typeColBreak          'colb'   /*result handler for */
                                          /* column break*/
   #define typeRowBreak          'rowb'   /*result handler for */
                                          /* end of line*/
   /*any data type in DBGetItem*/
   #define typeAnyType        (DBType)0   /*any data type*/
   /*infinite timeout value for DBGetItem*/
   kDBWaitForever                = -1,    /*infinite timeout value for */
                                          /* DBGetItem*/
   /*flags for DBGetItem*/
   kDBLastColFlag                = 0x0001,/*data item is last column */
                                          /* of the row*/
   kDBNullFlag                   = 0x0004 /*data item is NULL*/
};
enum {
   /*more data type codes*/
   typeBoolean                = 'bool',   /*Boolean*/
   typeChar                   = 'TEXT',   /*character*/
   typeSMInt                  = 'shor',   /*short integer*/
   typeInteger                = 'long',   /*integer*/
   typeSMFloat                = 'sing',   /*short floating point*/
   typeFloat                  = 'doub'    /*floating point*/
};
```

Data Types

```
typedef OSType DBType;              /*data type*/

struct DBAsyncParamBlockRec {       /*asynchronous parameter block*/
   ProcPtr  completionProc;         /*pointer to completion routine*/
   OSErr    result;                 /*result of call*/
   long     userRef;                /*reserved for use by application*/
   long     ddevRef;                /*reserved for use by database */
                                    /* extension*/
   long     reserved;               /*reserved for use by */
                                    /* Data Access Manager*/
};
typedef struct DBAsyncParamBlockRec DBAsyncParamBlockRec;
typedef DBAsyncParamBlockRec *DBAsyncParmBlkPtr;

struct ResListElem {                /*resource list in QueryRecord*/
   ResType  theType;                /*resource type*/
   short    id;                     /*resource ID*/
};
typedef struct ResListElem ResListElem;
typedef ResListElem *ResLisPtr, **ResListHandle;

typedef Handle **QueryListHandle;
struct QueryRecord {                /*query record*/
   short             version;       /*query record format version*/
   short             id;            /*resource ID of 'qrsc'*/
   Handle            queryProc;     /*handle to query def proc*/
   Str63             ddevName;      /*name of database extension*/
   Str255            host;          /*name of host computer*/
   Str255            user;          /*name of user*/
   Str255            password;      /*user's password*/
   Str255            connStr;       /*connection string*/
   short             currQuery;     /*index of current query*/
   short             numQueries;    /*number of queries in list*/
   QueryListHandle   queryList;     /*handle to array of handles to text*/
   short             numRes;        /*number of resources in list*/
   ResListHandle     resList;       /*handle to array of resource list */
                                    /* elements*/
   Handle            dataHandle;    /*handle to memory for query def proc*/
   long              refCon;        /*reserved for use by application*/
};
typedef struct QueryRecord QueryRecord;
```

```
typedef QueryRecord *QueryPtr, **QueryHandle;

/*column types array in ResultsRecord*/
typedef Handle ColTypesHandle;

struct DBColInfoRecord {            /*column info in ResultsRecord*/
   short len;                       /*length of data item*/
   short places;                    /*places for decimal and money */
                                    /* data items*/
   short flags;                     /*flags for data item*/
};
typedef struct DBColInfoRecord DBColInfoRecord;
typedef Handle ColInfoHandle;

struct ResultsRecord {              /*results returned by DBGetResults*/
   short          numRows;          /*number of rows retrieved*/
   short          numCols;          /*number of columns per row*/
   ColTypesHandle colTypes;         /*type of data in each column*/
   Handle         colData;          /*array of data items*/
   ColInfoHandle  colInfo;          /*DBColInfoRecord array--info about */
                                    /* each data item*/
};
typedef struct ResultsRecord ResultsRecord;
```

Data Access Manager Routines

Initializing the Data Access Manager

```
pascal OSErr InitDBPack     (void);
```

High-Level Interface: Handling Query Documents

```
pascal OSErr DBGetNewQuery  (short queryID, QueryHandle *query);
pascal OSErr DBDisposeQuery (QueryHandle query);
pascal OSErr DBStartQuery   (long *sessID, QueryHandle query,
                             ProcPtr statusProc, DBAsyncParmBlkPtr asyncPB);
```

High-Level Interface: Handling Query Results

```
pascal OSErr DBGetQueryResults
                            (long sessID, ResultsRecord *results,
                             long timeout, ProcPtr statusProc,
                             DBAsyncParmBlkPtr asyncPB);
pascal OSErr DBResultsToText
                            (ResultsRecord *results, Handle *theText);
```

Low-Level Interface: Controlling the Session

```
pascal OSErr DBInit          (long *sessID, ConstStr63Param ddevName,
                              ConstStr255Param host, ConstStr255Param user,
                              ConstStr255Param passwd,
                              ConstStr255Param connStr,
                              DBAsyncParmBlkPtr asyncPB);
pascal OSErr DBEnd           (long sessID, DBAsyncParmBlkPtr asyncPB);
pascal OSErr DBGetConnInfo   (long sessID, short sessNum, long *returnedID,
                              long *version, Str63 ddevName, Str255 host,
                              Str255 user, Str255 network, Str255 connStr,
                              long *start, OSErr *state,
                              DBAsyncParmBlkPtr asyncPB);
pascal OSErr DBGetSessionNum
                             (long sessID, short *sessNum,
                              DBAsyncParmBlkPtr asyncPB);
pascal OSErr DBKill          (DBAsyncParmBlkPtr asyncPB);
```

Low-Level Interface: Sending and Executing Queries

```
pascal OSErr DBSend          (long sessID, char *text, short len,
                              DBAsyncParmBlkPtr asyncPB);
pascal OSErr DBSendItem      (long sessID, DBType dataType, short len,
                              short places, short flags, void *buffer,
                              DBAsyncParmBlkPtr asyncPB);
pascal OSErr DBExec          (long sessID, DBAsyncParmBlkPtr asyncPB);
pascal OSErr DBState         (long sessID, DBAsyncParmBlkPtr asyncPB);
pascal OSErr DBGetErr        (long sessID, long *err1, long *err2,
                              Str255 item1, Str255 item2, Str255 errorMsg,
                              DBAsyncParmBlkPtr asyncPB);
pascal OSErr DBBreak         (long sessID, Boolean abort,
                              DBAsyncParmBlkPtr asyncPB);
```

Low-Level Interface: Retrieving Results

```
pascal OSErr DBGetItem       (long sessID, long timeout, DBType *dataType,
                              short *len, short *places, short *flags,
                              void *buffer, DBAsyncParmBlkPtr asyncPB);
pascal OSErr DBUnGetItem     (long sessID, DBAsyncParmBlkPtr asyncPB);
```

Installing and Removing Result Handlers

```
pascal OSErr DBInstallResultHandler
                             (DBType dataType, ProcPtr theHandler,
                              Boolean isSysHandler);
```

```
pascal OSErr DBGetResultHandler
                            (DBType dataType, ProcPtr *theHandler,
                             Boolean getSysHandler);
pascal OSErr DBRemoveResultHandler
                            (DBType dataType);
```

Application-Defined Routines

```
pascal Boolean MyStatusFunc (short message, OSErr result, short dataLen,
                             short dataPlaces, short dataFlags,
                             DBType dataType, Ptr dataPtr);
pascal OSErr  MyResultHandler
                            (DBType dataType, short theLen,
                             short thePlaces, short theFlags, Ptr theData,
                             Handle theText);
pascal OSErr MyQDef         (long *sessID, QueryHandle query);
```

Assembly-Language Summary

Trap Macros

Trap Macros Requiring Routine Selectors

`_Pack13`

Selector	Routine
$0100	`InitDBPack`
$020E	`DBKill`
$0210	`DBDisposeQuery`
$0215	`DBRemoveResultHandler`
$030F	`DBGetNewQuery`
$0403	`DBEnd`
$0408	`DBExec`
$0409	`DBState`
$040D	`DBUnGetItem`
$0413	`DBResultsToText`
$050B	`DBBreak`
$0514	`DBInstallResultHandler`
$0516	`DBGetResultHandler`
$0605	`DBGetSessionNum`

Selector	Routine
$0706	DBSend
$0811	DBStartQuery
$0A12	DBGetQueryResults
$0B07	DBSendItem
$0E02	DBInit
$0E0A	DBGetErr
$100C	DBGetItem
$1704	DBGetConnInfo

Result Codes

noErr	0	No error
userCanceledErr	–128	User canceled the query
rcDBNull	–800	The data item was NULL
rcDBValue	–801	Data available or successfully retrieved
rcDBError	–802	Error executing function
rcDBBadType	–803	Next data item not of requested data type
rcDBBreak	–804	Function timed out
rcDBExec	–805	Query currently executing
rcDBBadSessID	–806	Session ID is invalid
rcDBBadSessNum	–807	Invalid session number
rcDBBadDDev	–808	Couldn't find the specified database extension, or error occurred in opening database extension
rcDBAsyncNotSupp	–809	The database extension does not support asynchronous calls
rcDBBadAsynchPB	–810	Invalid parameter block specified
rcDBNoHandler	–811	There is no handler for this data type installed for the current application
rcDBWrongVersion	–812	Wrong version number
rcDBPackNotInited	–813	The InitDBPack function has not yet been called