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 (**General Guidelines for the User Interface**). 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 choose the desired query document. The query document sends the query 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.

The following Figure is a flowchart of a typical session using the high-level interface.

As illustrated, you must follow this procedure to use the high-level interface:

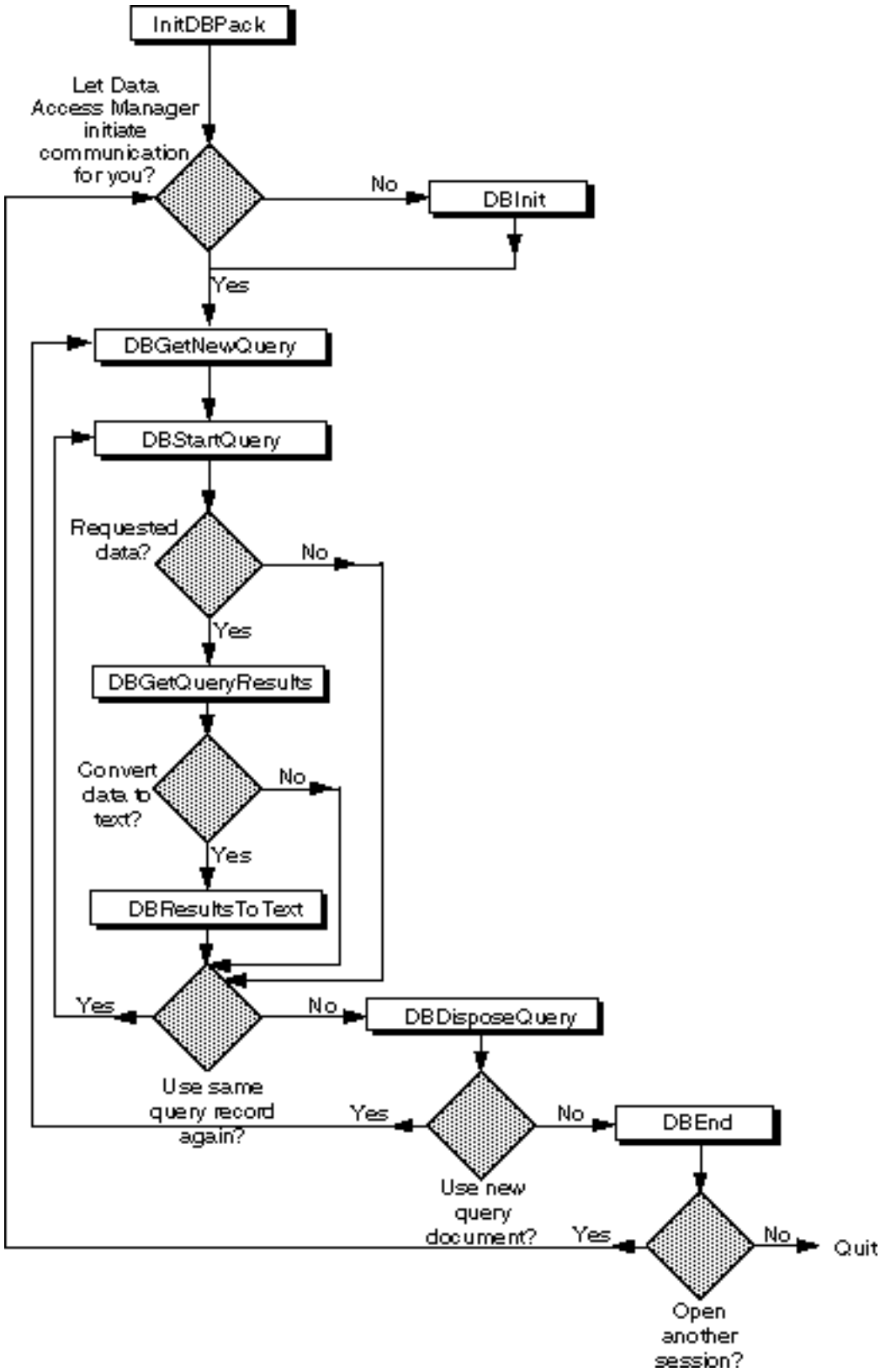
1. Call the **InitDBPack** function to initialize the **Data Access Manager**.

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. The **StandardGetFile** procedure is described in the **File Manager** , and in the **Resource Manager**.

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 that can 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.



A flowchart of a session using the high-level interface

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 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 to see if communication has been established with the data server. If not, it calls your status routine so that you can put up 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 the section entitled, **Using the Low-Level Interface** for more information on using the **DBInit** function.

Once communication has been established, the **DBStartQuery** function calls the **DBSend** function to send the query specified by the query record to the data server, and, when the **DBSend** function has completed execution, 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 on 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 on 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 under **Converting Query Results to Text** in the section entitled **Processing Query Results** .

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.

The Listing below illustrates the use of the high-level interface. This code sample 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 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. The following code example shows just one session.

The Listing below also 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  Using the high-level interface
// Assuming inclusion of <MacHeaders>

#include <DatabaseAccess.h>

// Define a record to include space for the current value in
// A5 so a completion routine can find it.

typedef struct {
    DBAsyncParamBlockRec   QPB;    // The parameter block
```

```
    long                    appsA5;     // Append A5 to the
                                        //parameter block
    } CRRec, *CRRecPtr;

CRRecPtr GetQPB (void)
    = {0x2E88};             // MOVE.L A0, (SP)

void MyHiLevel (ResultsRecord *rr, Handle myTextHdl, long *thisSession,
                OSErr *sessErr);
pascal void MyStartCompRoutine (CRRecPtr aCRRecPtr);
pascal Boolean MyStartStatus (short message, OSErr result, short dataLen,
                              short dataPlaces, short dataFlags,
                              DBType dataType, Ptr dataPtr);

pascal void MyGetQRCompRoutine(CRRecPtr aCRRecPtr);

pascal Boolean MyGetQRStatus (short message, OSErr result, short dataLen,
                              short dataPlaces, short dataFlags,
                              DBType dataType, Ptr dataPtr);
void GoDoSomething (void);
void DoError (OSErr myErr);

OSErr  gStartQErr, gGetQRErr;
Boolean     gStart, gQueryResults;

void MyHiLevel (ResultsRecord *rr, Handle myTextHdl, long *thisSession,
                OSErr *sessErr)
{
    CRRec               startPB, getQRPB;
    StandardFileReply   SFR;
    OSErr               packErr, startQErr, getQErr, disposeQErr, getNewQErr,
                        endErr, fsOpenErr, fsCloseErr, resultsErr;
    Handle          qrscHandle;
    short               rsrcId;
    ResType             rsrcType;
    Str255              rsrcName;
    QueryHandle         myQHandle;
    short               savedResFile;
    SFTypeList          typeList;
    short               fsRefNum;

    gStart = FALSE;
    gQueryResults = FALSE;
    sessErr = noErr;        // Assume everything went fine
    packErr = InitDBPack();   // Initialize the Data Access Manager

    // Display a dialog box to let the user pick a query document.

    typeList[0] = 'qery';
    StandardGetFile(nil, 1, typeList, &SFR);
    if (!SFR.sfGood)
        return;
    fsOpenErr = FSpOpenRF(&(SFR.sfFile), fsCurPerm, &fsRefNum);

    if (fsOpenErr) {
        *sessErr = fsOpenErr;
```

```
    return;
}
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);

// There shouldn't be an error unless there really isn't a
// 'qrsc' resource in the file the user selected.
if (ResError()) {
    *sessErr = ResErr;
    return;
}

// 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) {
    *sessErr = getNewQErr;
    endErr = DBEnd(*thisSession, nil);
    return;
}

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,
                    (DBAsyncParmBlkPtr) &startPB);

if (startQErr) {
    *sessErr = startQErr;
    if (*thisSession)
        endErr = DBEnd(*thisSession, nil);
        return;
}

while (!gStart)
    // While waiting for gStart to go TRUE, the routine GoDoSomething
    // calls WaitNextEvent to give other routines a chance to run.

    GoDoSomething ();

// The DBStartQuery call has completed.
if (gStartQErr) {
    *sessErr = gStartQErr;
    if (thisSession)
        endErr = DBEnd(*thisSession, nil);
    return;
```

```
    }
    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, (DBAsyncParmBlkPtr) &getQRPB);
    if (getQErr) {
        *sessErr = getQErr;
        endErr = DBEnd(*thisSession, nil);
        return;
    }

    while (!gQueryResults)
        GoDoSomething();

    // 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) {
        *sessErr = gGetQRErr;
        endErr = DBEnd(*thisSession, nil);
        return;
    }

    // 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) {
        DoError(fsCloseErr);
    }

    endErr = DBEnd(*thisSession, nil);
    if (endErr) {
        DoError(endErr);
    }

}


// The following two routines illustrate one way to implement a
// completion routine.

pascal void MyStartCompRoutine (CRRecPtr aCRRecPtr)
```

```
{
    long    curA5;

    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 the result code

    // Do whatever else you want to do.

    curA5 = SetA5(curA5); // Restore original A5
}                           // MyStartCompRoutine


pascal void MyGetQRCompRoutine(CRRecPtr aCRRecPtr)
{
    long    curA5;

    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
}                           // MyGetQRCompRoutine
```