## 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.

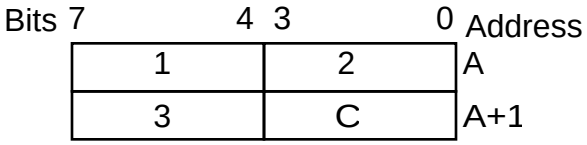
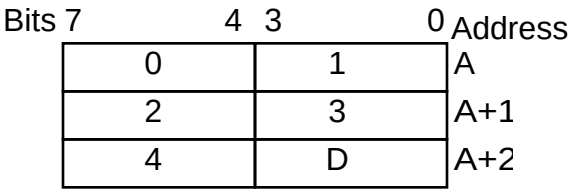
The following Table shows the data types recognized by the **Data Access Manager**. The writer of a database extension can define other data types to support specific data sources or data servers.

Each data type has a standard definition, shown in the Table below. 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.

**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. |
| 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 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).<br><br>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.<br><br>For example, the number +123 is represented as $123C<br><br>  <br><br><br>and the number -1234 is represented as $01234D.<br><br>  <br><br><br>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.<br><br>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. |
| 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 **ResultsRecord**. You must allocate the **ResultsRecord** data structure and pass this record to the **DBGetQueryResults** function. The **Data Access Manager** allocates the handles inside the **ResultsRecord**. When your application is finished using the results record, you must deallocate both the **ResultsRecord** and the handles inside the **ResultsRecord** .

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

## 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 7.0, Apple is providing system result handlers for the data types listed here. (These data types are described in the following Table.)

| 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 is providing the following three system result handlers that do not correspond to any 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.

```
pascal OSErr  MyResultHandler (DBType dataType, short theLen, short
thePlaces, short theFlags, Ptr theData, Handle theText );
```

The *dataType* parameter specifies the data type of the data item that the **DBResultsToText** function is passing to the result handler. The table above 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 **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 **Memory Manager**.

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 specifying that this routine is not a system result handler.

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

The Listing below shows a result handler that converts the integer data type to text.

```
// A result handler

// Assuming inclusion of <MacHeaders>

#include <DatabaseAccess.h>
#include <pascal.h>
#include <string.h>

pascal OSErr MyTypeIntegerHandler (DBType datatype, short theLen, Ptr
                              theData, Handle theText);

pascal OSErr MyTypeIntegerHandler (DBType datatype, short theLen, Ptr
                              theData, Handle theText)
{

    long        theInt;
    long        theTextLen;
    Str255      temp;
    Ptr         atemp1;
    long        atemp2;
    long        atemp3;


    BlockMove(theData, &theInt, sizeof(theInt));
```

```
    NumToString(theInt, temp);          // Convert to text

    theTextLen = GetHandleSize(theText);// Get current size of theText

    SetHandleSize(theText, theTextLen + (long) (temp[0]));
    // Grow text handle by length of temp

    if (MemErr)
        return MemErr;
    else {
        atemp1 = (Ptr) temp;
        atemp2 = (long) *theText + theTextLen;
        atemp3 = strlen (PtoCstr(temp));

        // use BlockMove to append text

        BlockMove(atemp1, (Ptr) atemp2, atemp3);
        return MemErr;
    }
}
```