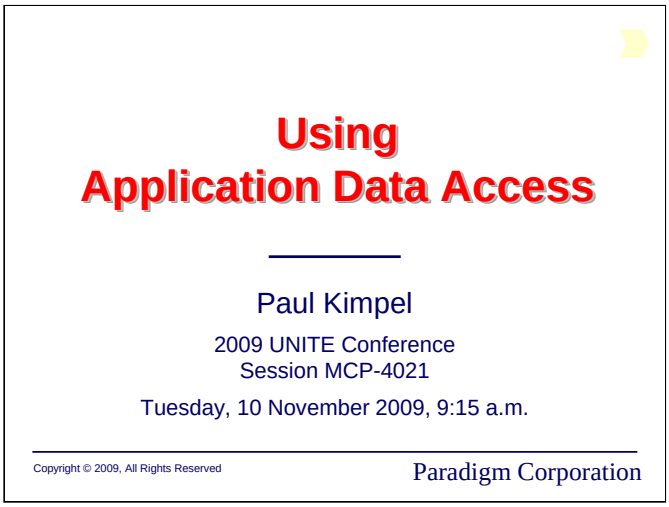


Using Application Data Access

2009 UNITE Conference
Minneapolis, Minnesota

Session MCP-4021

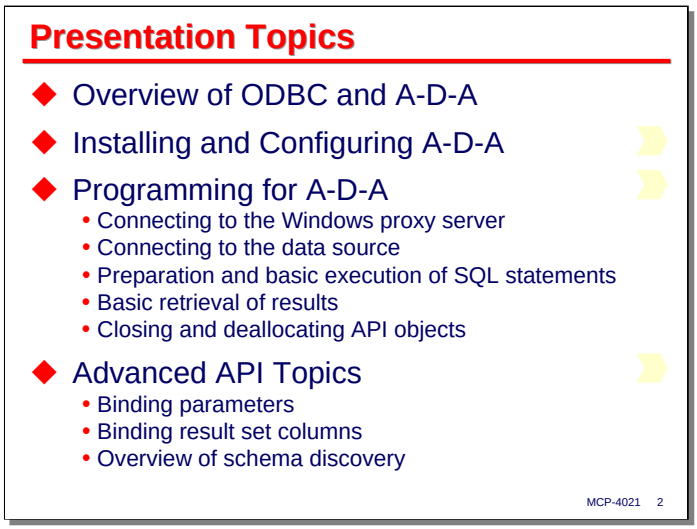
Tuesday, 10 November 2009, 9:15 a.m.

Paul Kimpel

Paradigm Corporation
San Diego, California

http://www.digm.com

e-mail: paul.kimpel@digm.com



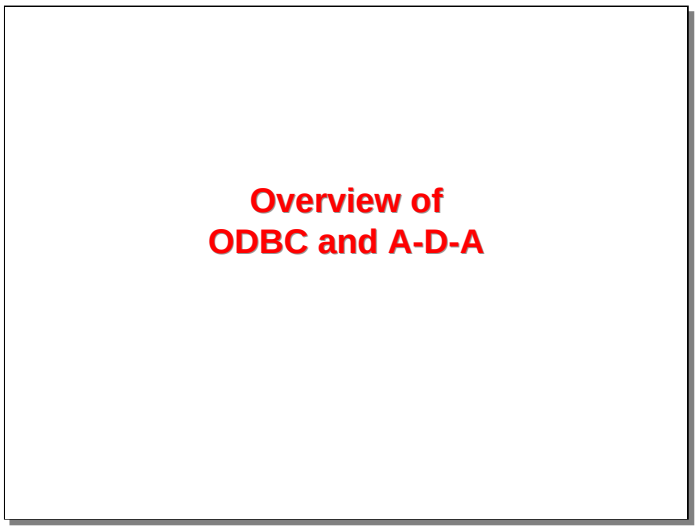
In this presentation I am going to discuss the Application Data Access (A-D-A) product for Unisys MCP systems. This software product permits MCP-based applications to access external databases and other data sources by mean of ODBC.

I will begin with a brief overview of ODBC and the interface that Application Data Access provides to it.

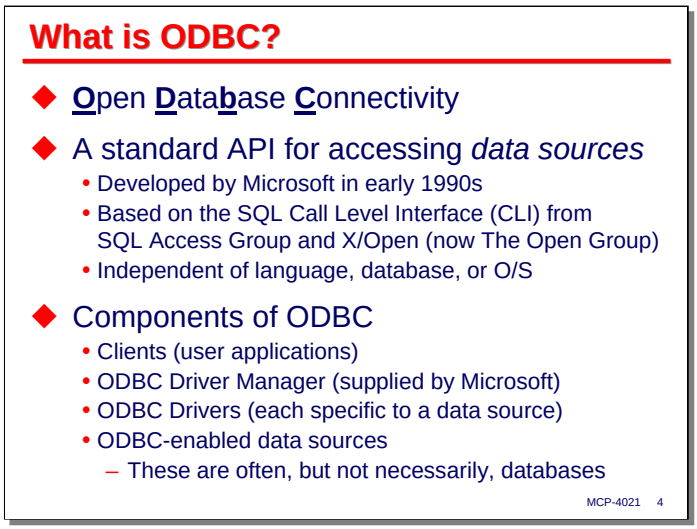
Next, I will discuss how you install and configure A-D-A for your environment.

The bulk of the presentation will be taken up with a discussion of programming for A-D-A, primarily as it would be done for COBOL programs. This is a complex API, with a number of alternatives available to do a particular task. I will try to step through the process of writing to the A-D-A API, with each topic building on the next.

This programming discussion is divided into two parts. The first part covers the basics – connecting to the Windows proxy and the data source, basic preparation and execution of SQL statements, processing result sets from queries, and closing and deallocating the API objects. The second part covers more advanced features – preparation, execution, and result sets – primarily the binding of parameters to SQL statements and column binding for result sets. This section also presents a brief overview of schema discovery.



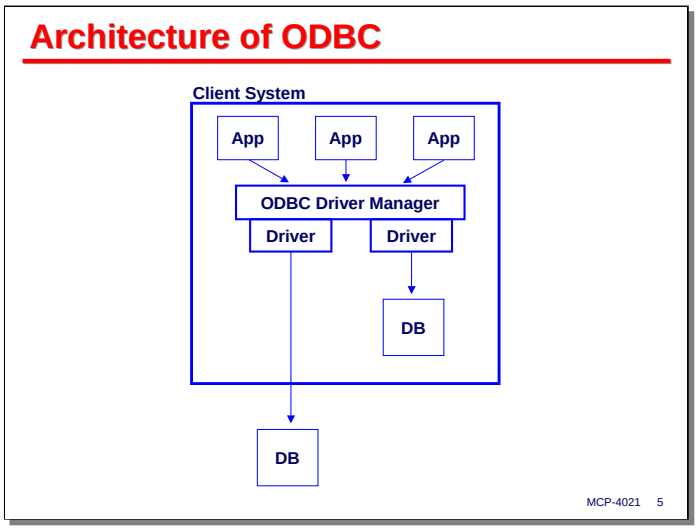
Let us begin with a brief overview of ODBC and how A-D-A relates to it.



ODBC stands for Open Database Connectivity. It is a standard that was developed by Microsoft in the early 1990s. It is based on an API called the SQL Call Level Interface (CLI), which in turn was originally developed by the SQL Access Group and X/Open (which is now known as The Open Group). The design goal of both SQL CLI and ODBC is to provide an interface to data sources that is completely independent of programming language, database type, or operating system environment.

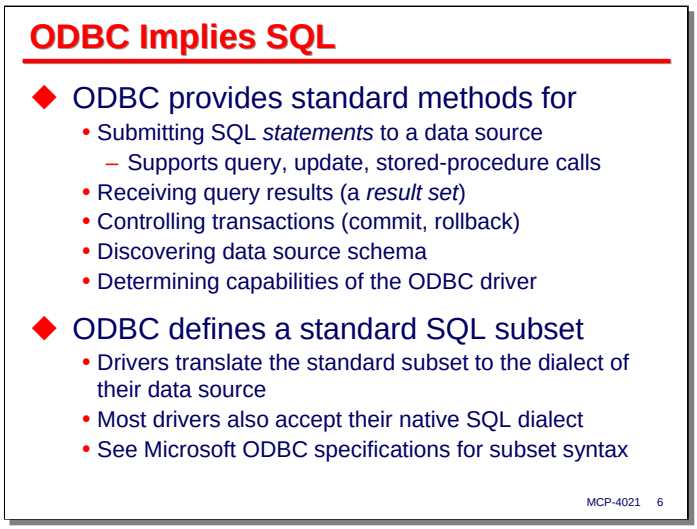
ODBC is really a framework that ties together a number of independently-developed parts and specifies how they will work together:

- At one end are the *clients* or end-user applications. These programs use the ODBC API to access databases and other types of data sources.
- The ODBC API used by clients does not talk directly to the data sources. Instead, it interfaces with a piece of Microsoft software called the ODBC Driver Manager. This software is responsible for interfacing driver modules associated with specific data sources and routing API calls to the correct ones.
- ODBC drivers are the interfaces that access the data sources on behalf of the ODBC Driver Manager. Each driver is specific to a particular data source, and is often developed and supplied by the vendor or creator for the data source. For example, Microsoft has an ODBC driver for SQL Server, Oracle has one for their database, and Unisys has one for its Data Access ODBC interface to DMSII.
- Finally, there are the data sources themselves. These are often, but not necessarily, databases. Anything that has an ODBC-compliant driver can be used by an ODBC-enabled application, e.g., data loggers, telemetry equipment, a GPS device – whatever.



The diagram on this slide shows the relationships among the various parts of the ODBC environment. Applications call the ODBC Driver Manager's API. The Driver Manager loads and connects to driver modules, which must be resident on the same system as the client application and Driver Manager. The drivers in turn translate the ODBC API calls into the specific forms required by their data sources.

Note that the data sources may be resident on the same system as the rest of the ODBC assemblage (e.g., a Microsoft Access database), or the data source could be on a separate system, possibly accessed over a network (as is commonly the case with database servers). The nature and location of the data source is hidden from the application client by the ODBC infrastructure.



ODBC uses SQL to access data sources, and the data source (or its driver) must accept SQL statements from the client applications. These statements include queries (**`SELECT`**), updates (**`INSERT`**, **`UPDATE`**, **`DELETE`**), and if the data source supports them, stored procedure calls.

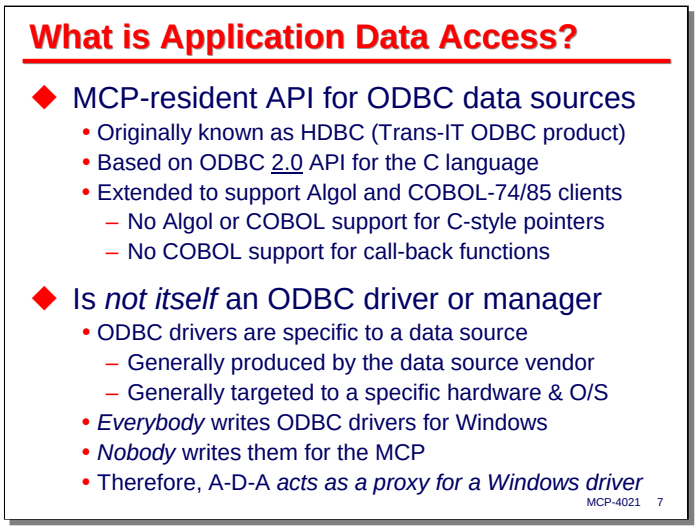
In addition to providing a means to submit SQL statements, the ODBC API also provides the following:

- Mechanisms to receive and process the results of queries. A SQL query returns a table-like structure, called a *result set*, that consists of rows and columns. The API contains routines that access the rows of the result set and retrieve the values of the columns.
- A mechanism to commit and roll back transactions.
- A mechanism to retrieve schema metadata from the data source. The schema describes the resources the data source makes available through ODBC – data tables, columns (fields), indexes, stored procedures, etc.
- A mechanism to determine the capabilities of the ODBC driver itself. The full ODBC standard has a lot of features and capabilities, of which probably no data source implements them all. This mechanism allows an application client to dynamically determine which features a particular driver and data source support.

While SQL has been the subject of a standardization effort, most data sources implement only a subset of that standard. Most data sources also implement non-standard extensions, so that SQL statements written for one data source may not work (or work the same way) with another data source.

ODBC defines a standard subset of the SQL language, and all compliant drivers are required to support that subset. Drivers generally convert that ODBC dialect of SQL into the native one used by their data source. Most drives will also accept the native dialect for the data source.

The dialect of SQL that ODBC uses is defined in the Microsoft specifications, a link to which can be found in the References at the end of this presentation.



With that background on ODBC, what is Application Data Access and how does it relate to ODBC?

A-D-A is an MCP-resident API for ODBC. It was originally known as HDBC (Host Database Connectivity), and first appeared as part of the old Trans-IT suite of MCP database products.

A-D-A is based on the version 2.0 ODBC API for the C language. At this writing, the current Microsoft specification is version 3.81. Unisys has extended that C-oriented API to support application clients written in MCP Algol and COBOL-74/85. There are two primary reasons why these extensions are necessary:

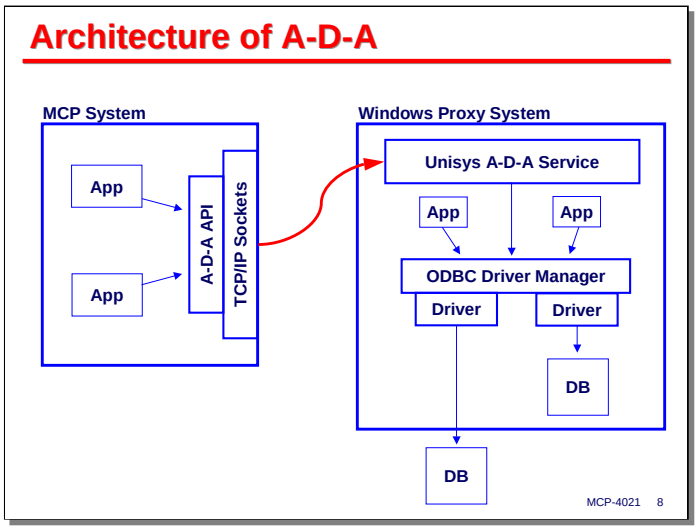
- The C API relies heavily on the use of C-style pointer variables. Neither Algol nor COBOL support C-style pointers.
- The C API also relies on the use of call-back functions as arguments to some of its routines. COBOL does not support call-back functions.

Note that A-D-A is not itself an ODBC driver or manager. As mentioned earlier, ODBC drivers are specific to a data source. They are generally produced by the vendor or creator of the data source. They are also generally written for a specific hardware platform and operating system environment.

There is a problem for MCP applications. *Everybody* writes ODBC drivers for Windows. *Nobody* writes ODBC drivers that run in the MCP environment, not even Unisys. It would be nice if there was a way to employ all of the ODBC drivers available for Windows so that those existing drivers could be used by MCP applications.

That is what A-D-A does – it leverages the Windows ODBC drivers that exist for almost every type of data source by acting as a proxy for the Windows drivers. It provides a connection between MCP applications and a Windows system that can run any Windows-based ODBC driver we need.

The next slide shows how this is done.



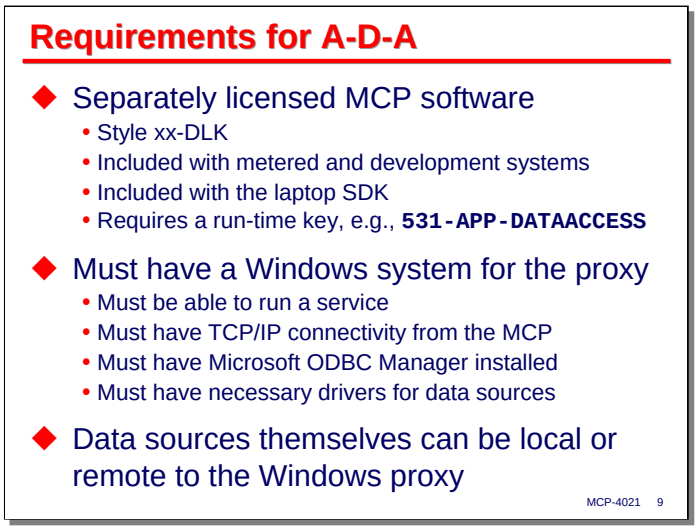
The diagram on this slide shows how Application Data Access leverages standard ODBC drivers on a separate Windows system. On the right is a diagram of the ODBC infrastructure on a Windows system, much as we saw it a few slides before. On the left is an MCP system running A-D-A.

A-D-A consists of two parts:

- An API (implemented as a library) running within the MCP environment.
- A Windows service running on the system which has the ODBC drivers we need. The MCP-resident API communicates with the Windows A-D-A service over a TCP/IP connection, typically on port 9876.

MCP applications make calls on the A-D-A API (which for the most part is equivalent to the ODBC API). Those calls are passed to the A-D-A service on the Windows system, which in turn converts them to standard ODBC API calls on the Windows ODBC Driver Manager. To the Driver Manager, the A-D-A service is the client application, but in fact, the service is simply acting as a proxy for the MCP-resident applications.

This arrangements allows MCP applications to make use of any data source which has a Windows ODBC driver. In practical terms, that is virtually all ODBC-enabled data sources.

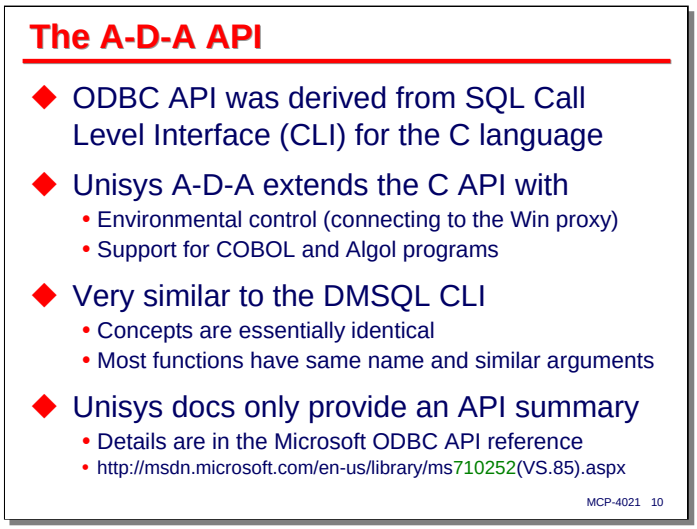


There are two main requirements for using Application Data Access:

- The first is that you have to get it from Unisys. A-D-A is not part of the standard ClearPath IOE. It is separately licensed (style xx-DLK). It is included, however, with system operating under metered and development licenses. It is also included with the MCP laptop SDK.[1] In addition to the **DLK** key, A-D-A requires a run-time key, *nnn*-**APP-DATAACCESS**, where *nnn* is the system release level (e.g., 531).
- The second is that you must have a Windows system on which to run the proxy service. You must have TCP/IP connectivity from the MCP system to that Windows system. The Windows system must have the Microsoft ODBC Driver Manager installed – it comes standard on all versions of Windows. Finally, you must have the necessary ODBC drivers for the data sources you want to access installed and configured on the Windows system.

Once again, the data sources themselves can be local to the Windows system or located elsewhere. As long as the Windows system can access the data sources through ODBC, MCP applications should be able to access them through A-D-A.

[1] For at least MCP 10 and MCP 12, the **511-APP-DATAACCESS** and **531-APP-DATAACCESS** run-time keys have not been included in the release. You will need to submit a UCF to Unisys to obtain them. Without the run-time key, you can install the A-D-A software, but you won't be able to use it. The **ODBCINIT** call (discussed later) will fail.



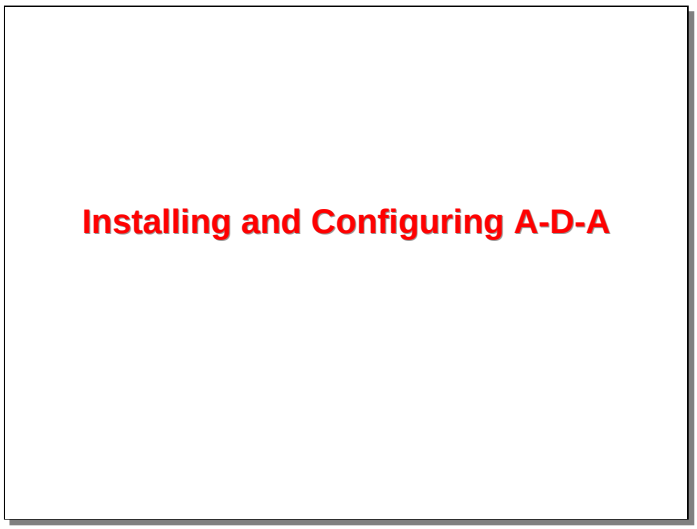
As previously mentioned, the ODBC API was derived from the SQL Call Level Interface (CLI) for the C language. Unisys has extended that API in two respects:

- Additional functions for environmental control, primarily to connect to the Windows proxy system.
- Changes and additions to the API to support COBOL and Algol programs where the C orientation to the original API makes that necessary.

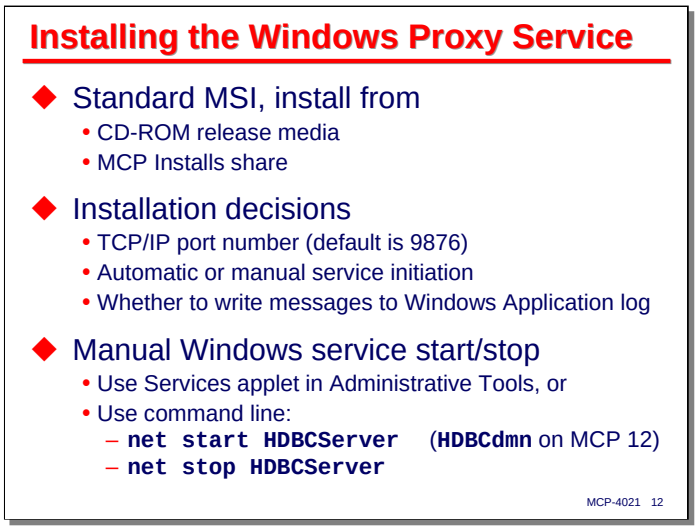
Because the ODBC API was derived from the SQL API, it turns out that it is very similar to the DMSQL CLI for DMSII. Most of the concepts between the two APIs are essentially identical, especially with respect to the extensions that support COBOL and Algol programs. Most of the API entry points have the same name, and the same or very similar argument signatures. If you have used one of these APIs in the MCP environment, it is fairly easy to learn the other one.

Because A-D-A is a thin layer on top of the standard ODBC API, the Unisys documentation for A-D-A contains very little information about ODBC per se. The documentation concentrates on the differences and additions between standard ODBC and A-D-A. You will need a good reference to the ODBC standard in order to understand completely what each API entry point does. The current Microsoft standard has changed in a number of ways from the version 2.0 standard on which A-D-A is based, but the current Microsoft standard maintains backward compatibility with the 2.0 API and describes the differences in enough detail that you can use it for A-D-A programming.

Note that the URL for specific document pages on the Microsoft MSDN web site tends to change over time. The URL shown on the slide was accurate as this is being written. If you find it no longer accesses the ODBC API documentation, do a search within MSDN for "ODBC API".



With that background on ODBC and A-D-A, the next topic is installing and configuring it for your environment.



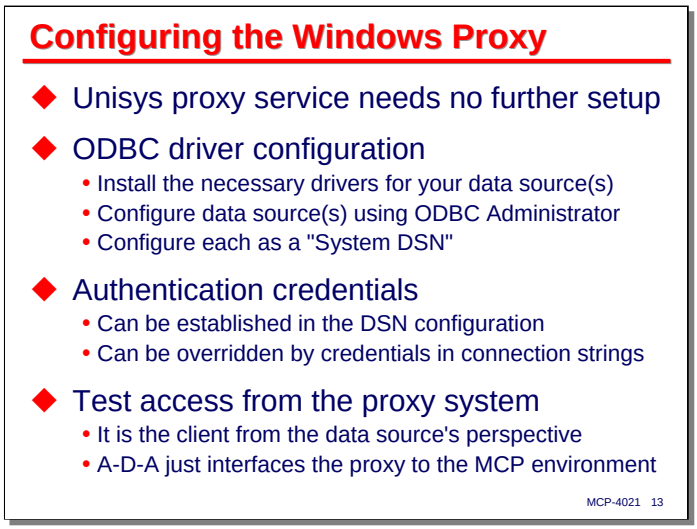
The first step is to install the A-D-A Windows service on the Windows system that will serve as your ODBC proxy. The A-D-A service is a standard MSI, which can be installed from CD-ROM media you received with your software release, or from the MCP Installs share.

When you run the MSI, you will need to make three decisions:

- The TCP/IP port number that will be used for connections between the MCP and the Windows service. The default is 9876. Unless you have a specific reason to use some other port, use that default.
- Whether you want the service to be initiated automatically or manually. With automatic initiation, the service is started each time Windows boots. With manual initiation, you will need to do that yourself, as described below.
- Whether to allow the Windows service to write messages to the Windows Application log. The service logs each connection from the MCP, along with any errors it encounters.

If you choose manual service initiation, there are two ways to start and stop the service.

- The first is using the Services applet in Administrative tools.
- The second is using a command window and net start/stop commands. The name of the A-D-A service on prior releases was **`HDBCServer`**. On MCP 12, the service name is **`HDBCdmn`**.

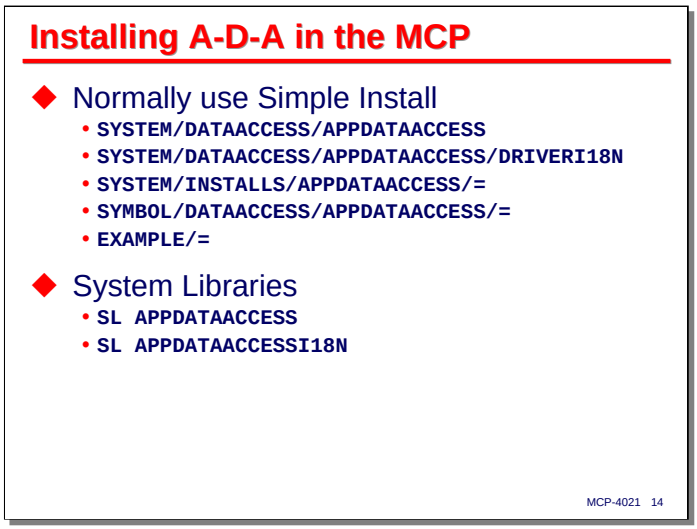


Once installed, the A-D-A proxy service needs no further configuration.

You will, however, need to install and configure the ODBC drivers for the data sources you want to access, if that has not been done already. You establish data source drivers through the Windows ODBC Administrator tool, usually found under Administrative Tools. You must configure data sources as a "System DSN." DSN is an acronym for Data Source Name.

Most data sources require some form of authentication in order to access them. You can include the credentials in the DSN configuration (which means anyone, including any MCP application, can then access the data source), or you can include the credentials in the connection string that the MCP application passes when opening a connection to a driver (a subject we will get to shortly). Credentials in the connection string will override those established in the DSN.

Finally, it is a good idea to test access to the data source from the Windows system. As far as the data source and its driver are concerned, the Windows proxy is the client. If you can't access the data source from Windows, you certainly won't be able to from the MCP through the A-D-A proxy service.



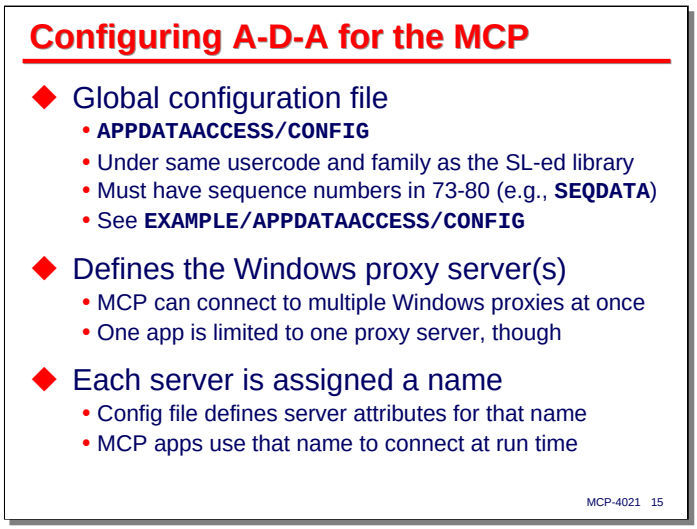
After installing and configuring A-D-A on the Windows system, the next step is to install and configure it in the MCP environment.

Normally, A-D-A should be installed using Simple Install. The package consists of:

- The main library, ***SYSTEM/DATAACCESS/APPDATAACCESS**
- The internationalization library, ***SYSTEM/DATAACCESS/APPDATAACCESSI18N**
- The Windows server installation software, under ***SYSTEM/INSTALLS/APPDATAACCESS/=**
- A set of include files for applications, under ***SYMBOL/DATAACCESS/APPDATAACCESS/=**
- A set of example program and configuration files, under ***EXAMPLES/=**

The installation process configures two system libraries, with the following function names:

- **APPDATAACCESS**
- **APPDATAACCESSI18N**



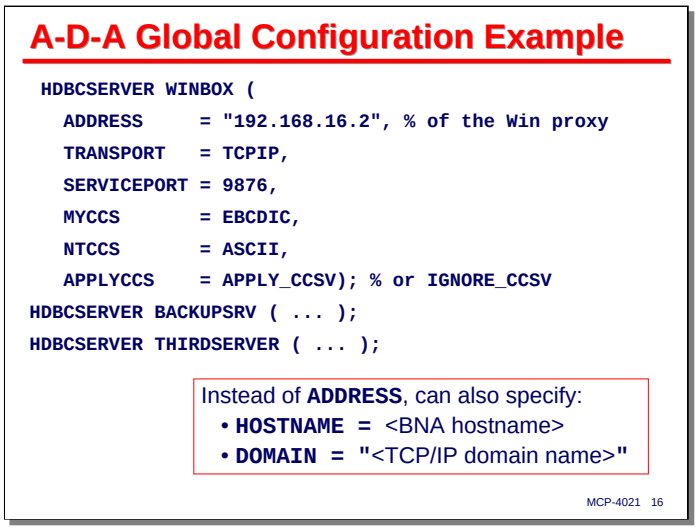
Once the A-D-A system software is installed, you need to create a configuration file for it. This configuration file must be named **APPDATAACCESS/CONFIG**, and be stored under the same usercode and family as the SL-ed library codefile **SYSTEM/DATAACCESS/APPDATAACCESS**. For most installations, this configuration file will be titled ***APPDATAACCESS/CONFIG ON DISK**.

The configuration file must be of a filekind that has sequence numbers in columns 73-80 (e.g., **SEQDATA**, **TEXTDATA**, **ALGOLSYMBOL**, etc.). There is a sample configuration file that is loaded with the system software, **EXAMPLE/APPDATAACCESS/CONFIG**, from which you can clone a configuration file for your site.

The purpose of this configuration file is to define the attributes of one or more Windows servers that run the A-D-A proxy service. A single MCP system can connect to multiple proxy servers at the same time (and can connect to multiple data sources through the same proxy server), but a given MCP application program is limited to using one proxy at a time.

Each server is assigned a simple name in the configuration file. MCP applications reference that name when they initialize their interface to the A-D-A software. In the configuration file, the name identifies attributes for the proxy server, primarily its network address. With this arrangement, MCP applications do not know (and do not need to know) the location of the proxy server – they reference a server only by its name defined in the configuration file.

The next slide shows what this configuration file looks like.

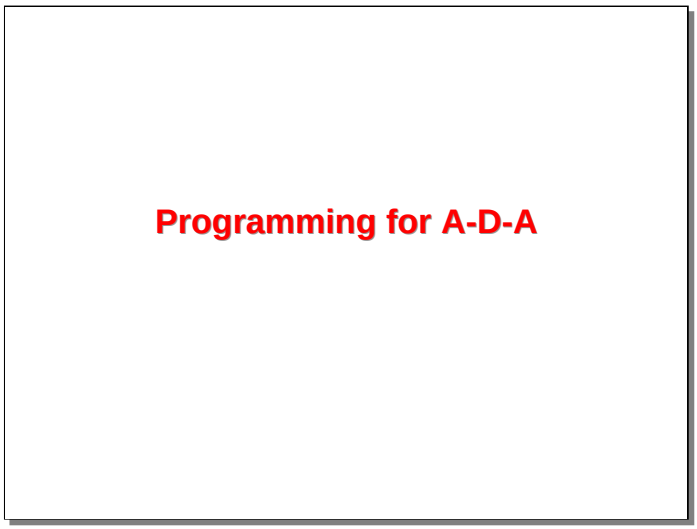


The structure of the A-D-A global configuration file is quite simple. It consists of one or more server definitions. The syntax is free form. A server definition can be continued across multiple lines.

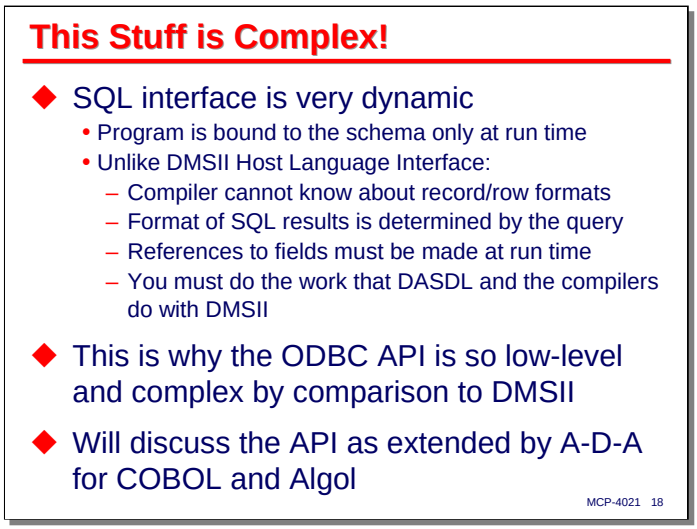
Each definition is introduced by the keyword **HDBCSERVER** followed by a simple identifier that defines the name of that server as it will be referenced by MCP applications. The name can be 1-17 characters long. Following the name is a parenthesized list of attributes for that server, viz,

- **ADDRESS** specifies the IP address (as a quoted string) for the Windows proxy server.
- **DOMAIN** can be used as an alternate to **ADDRESS**, and specifies the Internet domain name (.e.g, **"myhost.mycorp.com"**) of the Windows proxy server, again as a quoted string.
- **HOSTNAME** can also be used as an alternate for **ADDRESS**, and specifies the BNA host name for the proxy server. The BNA host name must *not* be enclosed in quotes.
- **TRANSPORT** specifies the type of connection that will be made to the proxy server. The valid mnemonic values for this attribute are **TCPIP** and **LANWISE** (an early third-party TCP/IP interface for MCP systems – this is probably now obsolete).
- **SERVICEPORT** specifies the TCP/IP port over which the connection to the proxy server will be made. The default is 9876. If you change this, you need to change the port definition on the proxy server as well.
- **MYCCS** specifies the Coded Character Set used by the MCP system. Typically this is specified as **EBCDIC**.
- **NTCCS** specifies the Coded Character Set that data should be translated to and from when exchanging it with the Windows proxy server. Typically this is specified as **ASCII**.
- **APPLYCCS** indicates whether translation should be performed. Normally this should be specified as **APPLY_CCSV**. The other choice is **IGNORE_CCSV** to suppress translation.

This configuration file is opened and parsed by the A-D-A library each time an MCP application attempts to open a connection, so it would be beneficial to keep it short and avoid having unnecessary entries in it.



That is all that is necessary to install and configure Application Data Access for the MCP and Windows environments where it will operate. The remainder of this presentation will be devoted to the large subject of how you write MCP application programs to use A-D-A.



Before launching into the subject of programming, I need to warn you that this API is extremely complex. If you are accustomed to using the DMSII Host Language Interface (**`DATA-BASE SECTION`**, **`FIND`** verbs, etc.), your initial impression may be that this interface is just too complex to bother with. The problem is, you don't have a choice – if you want to access external databases from the MCP environment, A-D-A is currently the only option available.

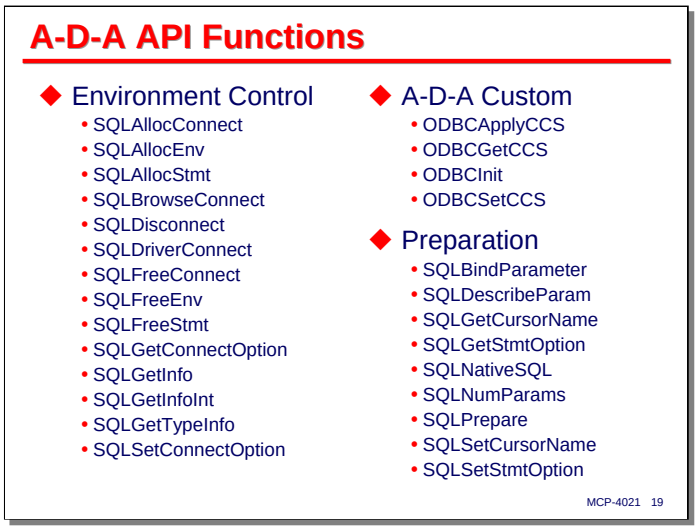
There are two reasons why the API is so complex. The first is that it is essentially the standard ODBC API for C – this is the nature of ODBC, and this is how the rest of the world uses ODBC (at least until OLE DB, ADO, and .Net came along).

The second reason is that a programmatic interface to SQL is by its nature very dynamic. The program can be bound to the schema of the database only at run time. This is very different from DMSII, which has a static schema (the one compiled by DASDL). MCP compilers know at compile time the structure of the database – the names of tables (data sets), the names of data items, and the data types and offsets of those data items in table records.

With SQL, you can't know any of that at compile time. With the DMSII Host Language Interface, you retrieve single records at a time. With SQL, you retrieve a result set. The data items, data types, and location of items in the result set rows are all determined by the SQL query (and by the underlying data source), so the compiler cannot give you any help in accessing that data as if it were fixed fields in some record area. The fields in result sets must be identified and accessed at run time. In other words, you need to do a lot of the work that DASDL and the compilers do for you with the standard DMSII interface.

It is this dynamic and run-time bound nature of SQL and ODBC that makes the API so low-level and complex to use by comparison with DMSII. There are a number of significant advantages to having a completely dynamic interface to a database, but ease of use is not one of them.

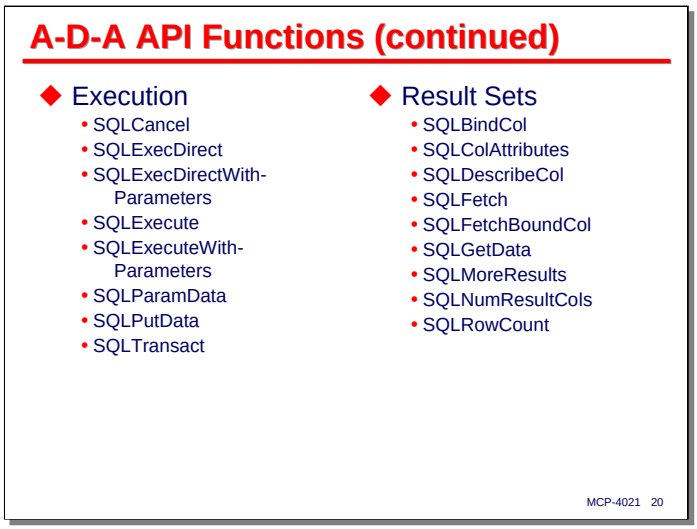
The MCP implementation of A-D-A does support the standard API for C, and if you program in C on the MCP, you can use that interface essentially as you would in other environments. In this presentation, however, I am going to concentrate on the API as it has been extended to support native MCP languages, particularly COBOL.



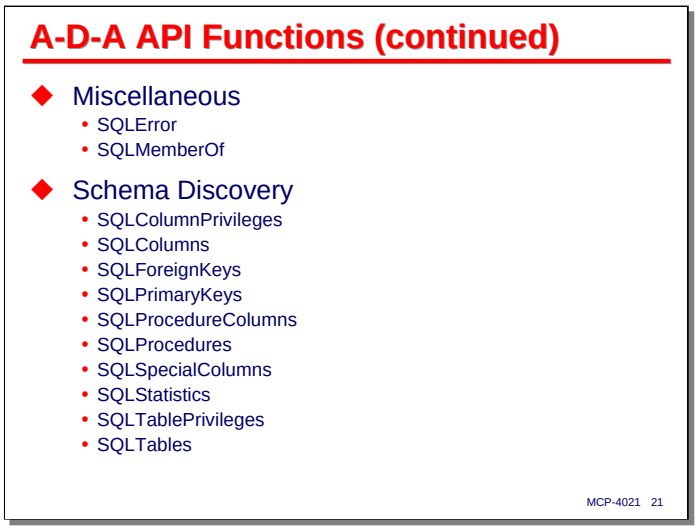
The ODBC API has a large number of functions, or entry points, you can call. A-D-A adds a few more to these. The next three slides list all of the functions and organize them into functional categories. We will discuss most of these functions, at least in passing, throughout the presentation.

The functions can be classified in seven parts:

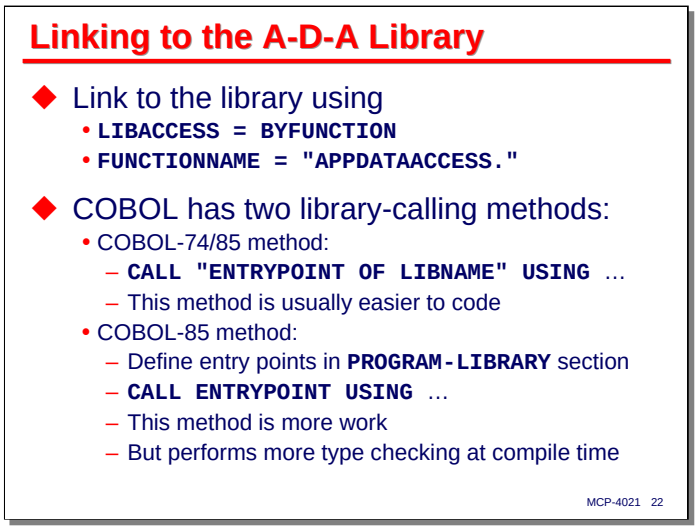
- **Environmental Control** – allocating API objects, connecting and disconnecting from data sources, and obtaining feature information about drivers and data sources.
- **A-D-A Custom** – functions specific to Application Data Access that deal with the Windows proxy server.
- **Preparation** – preparation of SQL statements and parameter bindings.
- **Execution** – execution of SQL statements and transaction control
- **Result Sets** – retrieving results from SQL statements, binding columns to a buffer, and retrieving metadata about result sets.
- **Miscellaneous** utility functions, including an error formatting routine.
- **Schema discovery** – retrieving schema metadata for a data source's table, columns, indexes, etc.



Continuing the listing and classification of API functions…



Completing the listing and classification of API functions…



The first step in using A-D-A is to specify the linkage to its library. Since it is configured as a system library, all you need is its library function name. In COBOL, you can specify this as follows:

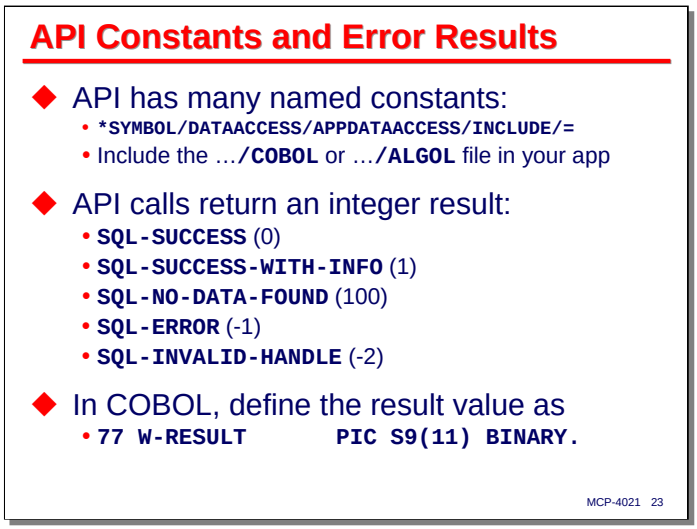
```
CHANGE ATTRIBUTE LIBACCESS OF "ADA" TO BYFUNCTION
CHANGE ATTRIBUTE FUNCTIONNAME OF "ADA" TO APPDATAACCESS."
```

Where "ADA" can be any string value. That value will be used in other CALL statements to access the library's entry points, e.g.,

```
CALL "SQLALLOCENV OF ADA" USING W-ENVIR GIVING W-RESULT
```

Actually, there are two methods for calling libraries in COBOL, an older one that works with both COBOL-74 and COBOL-85, and one that is implemented only in COBOL-85. The older style is usually easier to code, but the newer one available in COBOL-85 provides good type checking at run time.

You can use either method. The examples in this presentation use the older method that works with both dialects of COBOL.



The ODBC API uses many numeric values to specify options and attributes. These values are implemented as named constants in the include files that are loaded with the system software. You will want to include the appropriate file when you compile your program and reference these named constants. The include files are:

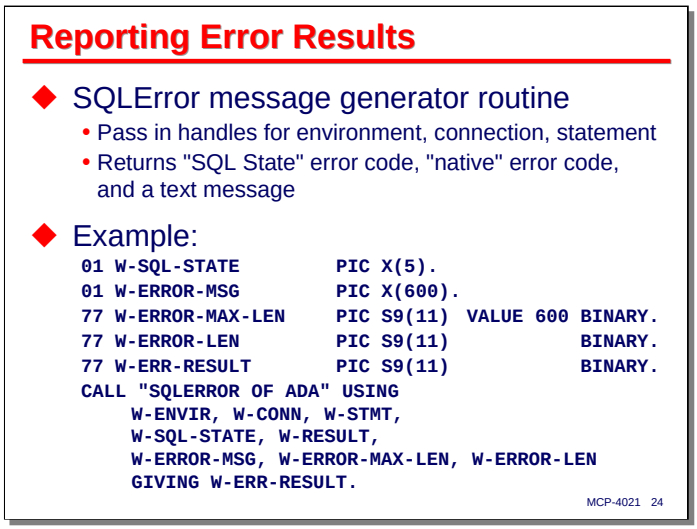
- For C: **`*SYMBOL/DATAACCESS/APPDATAACCESS/SQL/H`**
- For Algol: **`*SYMBOL/DATAACCESS/APPDATAACCESS/INCLUDE/ALGOL`**
- For COBOL: **`*SYMBOL/DATAACCESS/APPDATAACCESS/INCLUDE/COBOL`**

All calls to the API entry points return an integer value. The set of values is standardized across all functions:

- **`SQL-SUCCESS`** (0)
  This value indicates the call was successful.
- **`SQL-SUCCESS-WITH-INFO`** (1)
  This value indicates the call was successful, but that a warning condition was raised.
- **`SQL-NO-DATA-FOUND`** (100)
  This value, returned from calls that process result sets, indicates that no rows are available, or that the end of the result set has been reached.
- **`SQL-ERROR`** (-1)
  This value indicates some error occurred.
- **`SQL-INVALID-HANDLE`** (-2)
  This value indicates the program passed an invalid "handle" argument. We will discuss handles and how they are used shortly.

In COBOL, you must define the result variable that receives the return value as **`PIC S9(11) BINARY`**. In Algol, you must define it as **`INTEGER`**.

For results that return an error or warning indication, the API provides a message generating routine, discussed on the next slide.

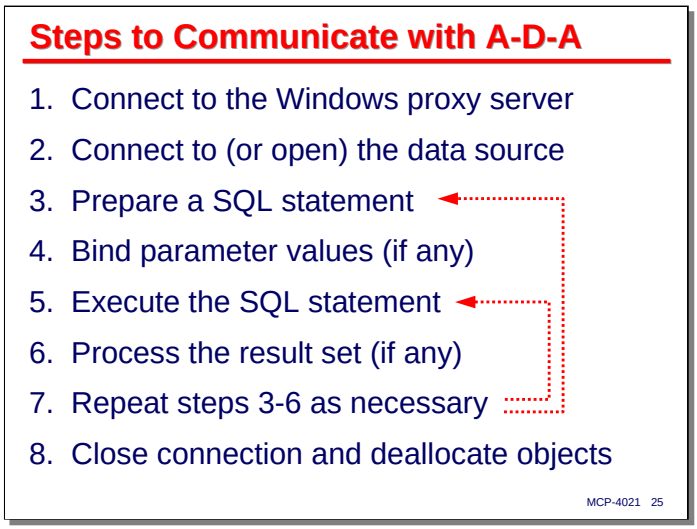


The ODBC standard defines a large number of error conditions that API calls can generate. To help your application to deal with this, the API includes a function that will generate a textual error message and a standard error code termed the "SQL State". You can call the error routine after any other API call.

You call the **SQLERROR** function passing the following arguments:

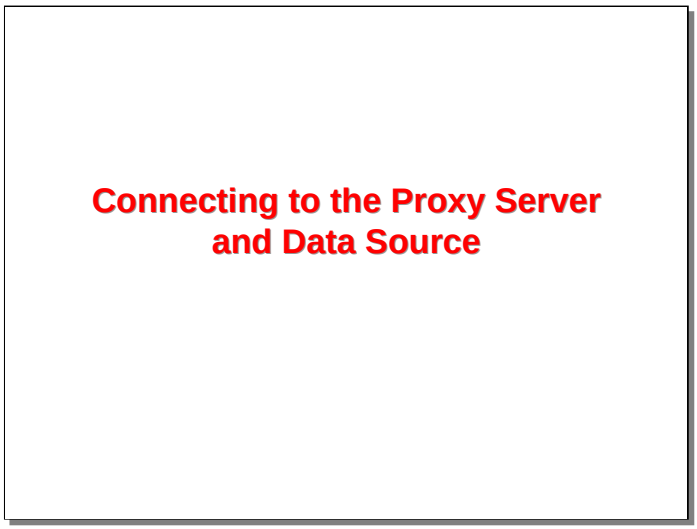
- Handles for the environment, connection, and statement objects. We will discuss "handles" in the next few slides. Not all of these objects may have been allocated before an error occurs (e.g., after allocating the environment and connection handles but before a statement handle has been allocated). The routine seems to understand, based on the error, which handles are appropriate to the error.
- A variable to receive the five-character SQL State result. This is the standard error identification code defined by the ODBC specification. It is alphanumeric. The slide shows how to declare it in COBOL. In Algol, you would pass an **EBCDIC ARRAY** for this argument.
- A variable to receive a binary "native" error code.
- A variable to receive a textual error message. The slide shows how this should be declared for COBOL; in Algol it is another **EBCDIC ARRAY**.
- A binary integer holding the maximum size of the error message argument. The **SQLERROR** routine will not place more than this number of characters in the error message area.
- A binary integer to receive the actual length of the message placed in the message area by the routine.

As with all API routines, **SQLERROR** returns a result value indicating success or error.

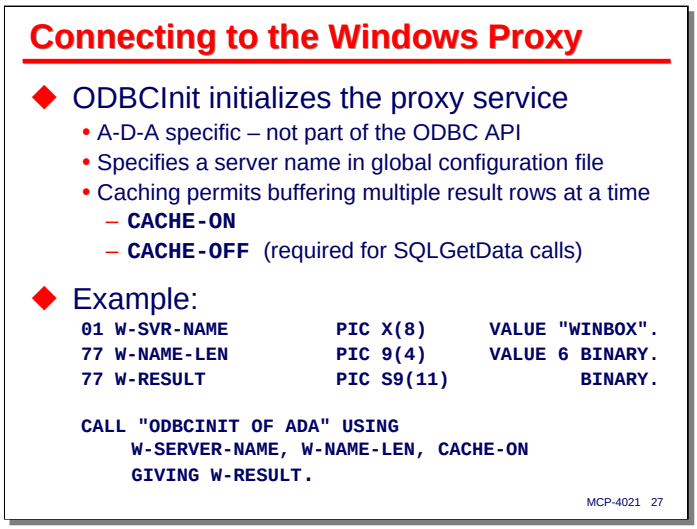


This slide shows the overall process used with Application Data Access to connect to the Windows proxy and execute SQL statements from an MCP application:

- Connect to the Windows proxy server. This is an initialization call, and is normally done only once during a program's execution.
- Connect to (or open) the data source. This establishes a connection through the Windows proxy to a specific ODBC driver and data source. You allocate environment and connection objects and call the driver connect function. You pass a connection string argument as part of this operation that identifies the data source and supplies any necessary options (such as authentication credentials).
- Now the real work begins. You allocate one or more statement objects and associate them with the text of a SQL statement. You can "prepare" (pre-compile) the statement in advance and execute it later, or you can have the driver prepare and execute the statement in one step.
- As we will discuss later, you can define your SQL statements to have parameters. If you do that, you will need to "bind" actual values to those parameters before executing the statement.
- The next step is execution of the SQL statement. For update statements and some stored procedure calls, the result of this step is a simple count of the number of rows in the database that were affected. For queries, the result is a "result set" – a table containing rows and columns of data produced by the query.
- If the SQL statement generated a result set, the next step is to process that result set. This is a little bit like reading a file, but since the format of the result set is determined by the SQL statement (which is specified dynamically), there is some extra work involved in accessing the columns of each row and extracting the values into local variables in your program. You process a result set one row at a time, but there are two ways to access the column values:
  - Column-by-column. For each row, you call a routine as many times as necessary to fetch each column in turn.
  - Using "bound columns." Prior to processing the result set, you call a series of functions to "bind" the columns of the result set to offsets you specify in a buffer (or record area). A single row-fetch call then formats all column values into the buffer at once. You can specify that data conversion is to be performed on a column-by-column basis when you establish the bindings. This approach is more work to set up, but makes the query results easier to access. It is also usually more efficient than the column-by-column method.
- After completing the processing of a SQL statement and any result set, you can repeat this sequence of steps as many times a necessary. If you prepared a statement in advance of executing it, the statement can be re-executed later, perhaps after supplying different parameter values for it. You can also modify the text of SQL statement associated with a statement object, re-prepare it, and execute that new statement.
- When you are completely done, you must close the connection to the data source and deallocate all statement, connection, and environment objects that were created by your program. This severs the connection with the driver and allows A-D-A to reclaim its memory resources.



The first step in getting a program to work with A-D-A involves connecting to the Windows proxy server and then to the data source.

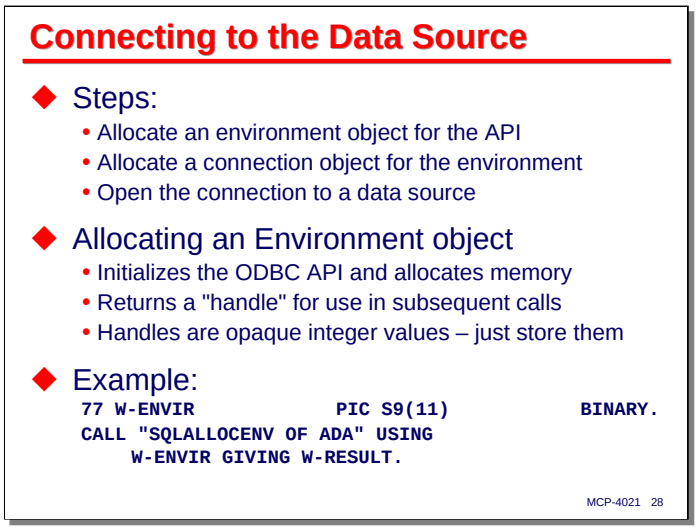


A-D-A provides a few functions that are not part of the ODBC API. The one you use most often is **ODBCINIT**. This function initializes the **APPDATAACCESS** library and the connection to the proxy server. You must call this function in your program before you call any other routines in the A-D-A API.

The routine requires three arguments:

- A string (COBOL record area) containing the name of the Windows proxy server. This must match one of the names previously configured in **APPDATAACCESS/CONFIG**.
- An integer containing the effective length of the server name.
- An integer that indicates how result sets should be cached. You should pass one of the named constants from the include file for your language. In COBOL, the choices are:
  - **CACHE-ON** – this enables the A-D-A library and Windows proxy service to request and cache multiple result rows from the data source. This is generally a good idea, as it reduces the number of round trips for network messages between the application and the data source. You cannot use caching, however, if you will be fetching result set values one column at a time using the **SQLGETDATA** function.
  - **CACHE-OFF** – this disables caching, and requires that each API call be passed to the driver when initiated by the MCP application. This is less efficient, but required if you are going to do column-by-column data retrieval using **SQLGETDATA**.

If the routine returns a success result, the library has been initiated, the Windows proxy has been connected, and your program is ready to begin issuing ODBC requests.



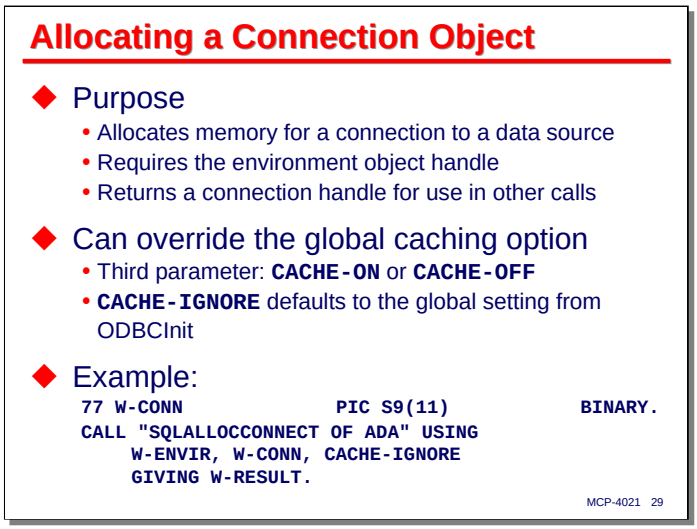
The next thing your program must do is connect to the ODBC data source. There are three steps to this:

- Allocate an environment object for the API
- Allocate a connection object for the environment
- Open the connection to the data source using the connection object.

The environment object is the core entity the API uses to keep track of your program's interface to it. Allocating this object also initializes the ODBC API for your program (whereas the **ODBCINIT** function initialized the A-D-A interface to ODBC).

All of the object allocation routines in the API return a binary integer value called a "handle." This value is used by the API to identify the object in its internal data structures. Handles are opaque objects –you are not concerned with their values – you just store them in your program and pass them as required to the various API routines. Handles are somewhat like COMS direct-window designators in this respect.

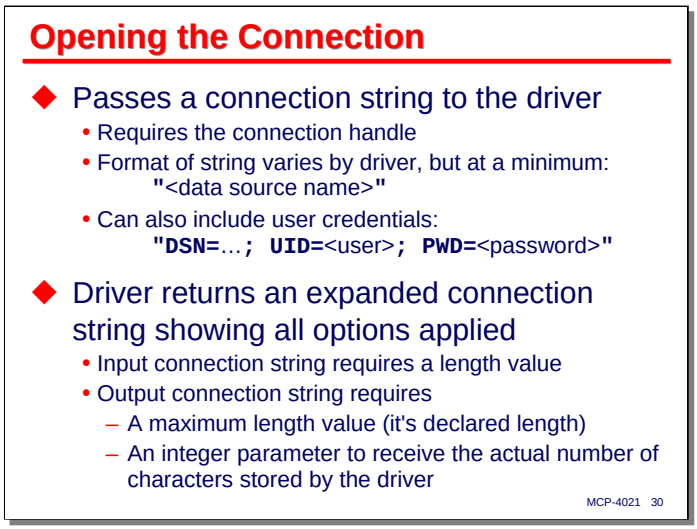
The **SQLALLOCENV** function simply allocates an environment object and returns its handle to your program. We will use that handle in the next step.



After you have an environment object, you allocate a connection object. The connection object is what allows your program to communicate with the driver for a specific data source.

Connection objects are allocated within an environment, so you must pass the environment handle you previously obtained to the **SQLALLOCCONNECT** routine. It will allocate the connection object and return a separate handle for it.

In addition to allocating the connection object, this routine can also override the caching option you specified on the **ODBCINIT** call. You can specify a value of **CACHE-ON** and **CACHE-OFF** to change the caching status, or a value of **CACHE-IGNORE**, which will apply the caching status from the **ODBCINIT** call to the connection.

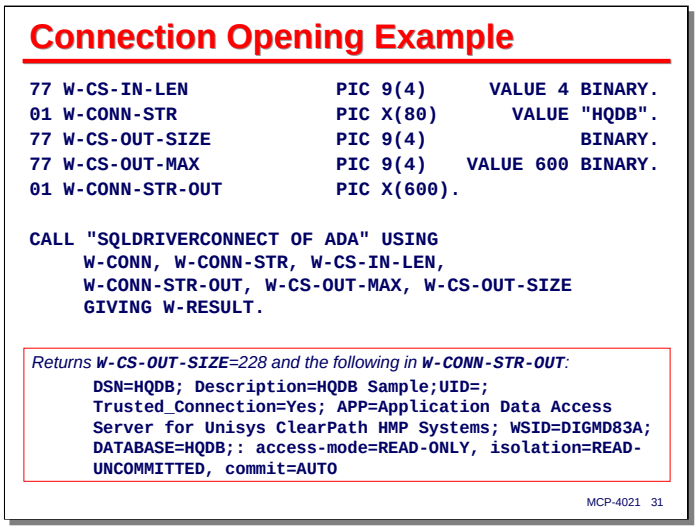


Once a connection object has been allocated, your program can proceed to open the connection to a data source by calling **`SQLDRIVERCONNECT`**. It specifies the data source and options for the connection through a text value called the "connection string."

The syntax of the connection string varies by data source and by driver, but at a minimum it must contain the data source name (DSN) defined when the data source was configured on the Windows proxy server by the ODBC Administrator utility. Options are specified as a series of semicolon-delimited name/value pairs.

A common option used in a connection string is the specification of user credentials for authentication. Most drivers accept a user identifier (**`UID`**) and password (**`PWD`**) option. At present, there is no way to secure the password when it is passed from the MCP application, so this is a potential security issue you must consider.

If the connection attempt is successful the data source driver will return an expanded connection string, showing all of the options and defaults that have been applied. You pass a separate string (record area) argument for this expanded connection string, along with an integer value indicating the area's maximum length, and another integer variable that will receive the actual length of the returned string. This is shown in more detail on the next slide.

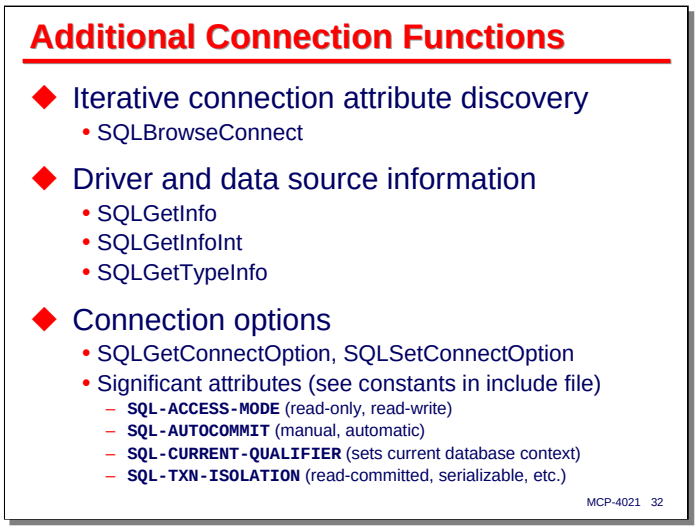


The **SQLDRIVERCONNECT** function attempts to open a connection to a data source based on the contents of the connection string you pass it. It has the following arguments:

- The handle for the connection object.
- The record area containing the text of the connection string.
- An integer variable containing the effective length of the connection string.
- A second record area to receive an expanded connection string that will be returned by the driver.
- An integer variable containing the maximum length of the expanded connection string. The driver will truncate any result that is longer than this.
- An integer variable that will receive the actual length of the expanded connection string that the driver formats in the second record area.

The slide shows an example of the variables for the input and expanded connection strings. In this case the connection was to a Microsoft SQL Server 2005 database.

After a connection is opened successfully, your program can proceed to submit SQL statements to the data source.



There are some additional functions for handling connections and data sources that I will not describe here in detail. You should refer to the Microsoft ODBC documentation for more information on these routines.

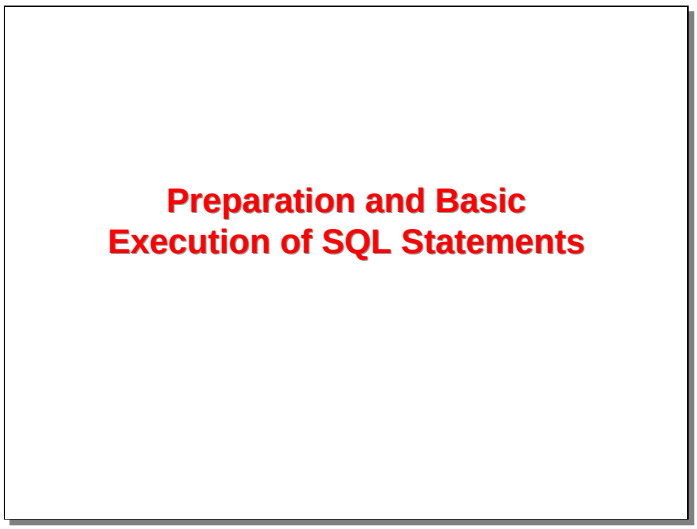
SQLBROWSECONNECT provides a mechanism to iteratively discover the capabilities of a data source and connect to it appropriately. This capability might be of use in a general-purpose tool that is not targeted to a specific type of data source.

SQLGETINFO, **SQLGETINFOINT**, and **SQLGETTYPEINFO** can interrogate the features and capabilities supported by a data source. Again this capability might be of interest in a general-purpose program.

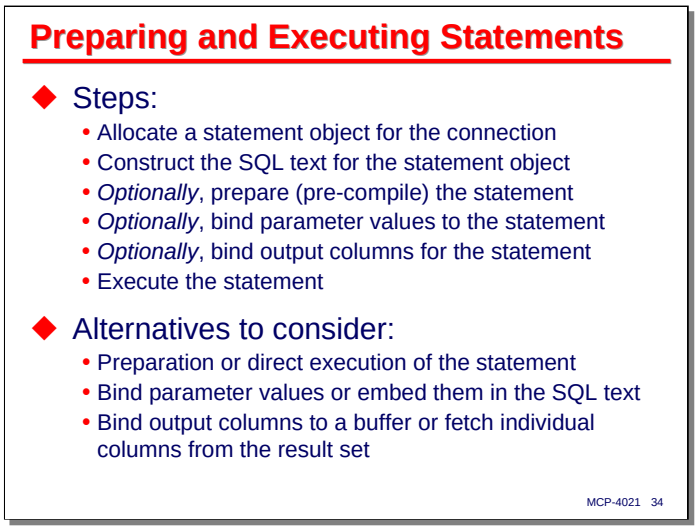
SQLGETCONNECTOPTION and **SQLSETCONNECTOPTION** allow you to interrogate and set certain options for the connection. In some cases, this can be done as an alternative to including options in the connection string. These options are specified using named constant values. You should refer to the A-D-A include file for your language for the names of the constants, but the following are of particular interest:

- **SQL-ACCESS-MODE** specifies whether the connection is to support read-only or read-write (update) operations.
- **SQL-AUTOCOMMIT** indicates whether or not update statements will be executed within their own transactions. When autocommit is set to automatic (the default for most data sources), each update operation is committed individually, as it occurs. When autocommit is set to manual, update operations become part of a larger transaction, which eventually must be committed by calling the **SQLTRANSACT** function described later.
- **SQL-CURRENT-QUALIFIER** can be used to set the database context. This is equivalent to executing a **USE** <database name> statement in many database dialects.
- **SQL-TXN-ISOLATION** specifies the degree of record locking that the database will do for queries. The possible choices (not necessarily supported by all drivers) are:
  - **SQL-TXN-READ-UNCOMMITTED** ("dirty reads")
  - **SQL-TXN-READ-COMMITTED**
  - **SQL-TXN-REPEATABLE-READ**
  - **SQL-TXN-SERIALIZABLE**
  - **SQL-TXN-VERSIONING**

There are named constants in the include file to support each of the connection options that have numeric options (e.g., **SQL-MODE-READ-ONLY**, **SQL-TXN-READ-COMMITTED**).



Having established the A-D-A environment and opened a connection to a data source, your program is now ready to start accessing data. This next section discusses the preparation and basic execution of SQL statements.



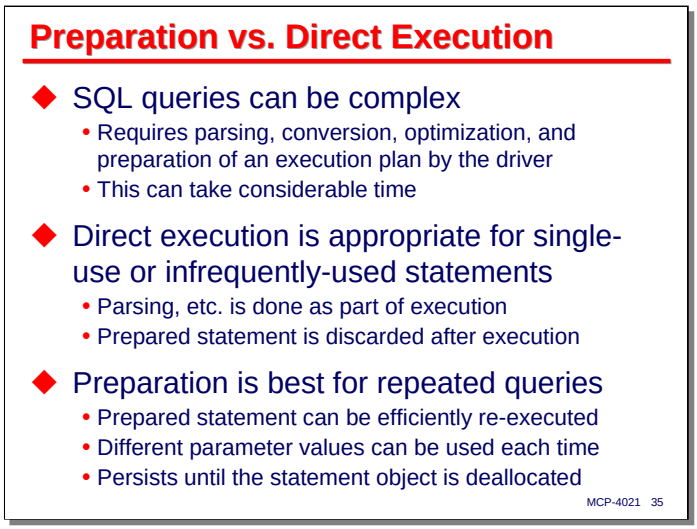
The basic activity that ODBC supports is the execution of SQL statements. That involves the following steps:

- First, you need to allocate a statement object and associate it with an existing connection object. If you need multiple queries or updates in your program, you can allocate multiple statement objects and use a separate one for each SQL query or update.
- Next, you construct the SQL text to be applied to the statement object. This can be a query, one of the update statements, or a call on a stored procedure (if the data source and driver support such).
- Now you have some choices:
  - Optionally, you can prepare the statement before executing it. This pre-compiles the SQL text and builds an execution plan. Doing this allows you to re-execute the statement multiple times without the on-going overhead involved in preparing the statement for execution.
  - Optionally you can define the SQL statement with parameter markers and bind values to those markers. The alternative is to embed the parameter values directly in the query text. Using parameter markers in conjunction with prepared statements allows you to re-execute the statement efficiently by only changing the values of the bound parameters.
  - Optionally, you can bind output columns for a query statement so that multiple columns can be retrieved at once. The alternative is to retrieve query results one column at a time using the **`SQLGETDATA`** function. Binding columns will be discussed in more detail later in the Advanced Topics section of the presentation. Note that if you choose to do column-by-column retrieval, caching must be disabled.
- After these preparatory steps, you then execute the statement. It is possible to skip the preparation step and do the prepare and execute together. That is an easy way to begin using the API, and is the approach we will discuss over the next few slides.

Once again, you have three alternatives to consider when planning the execution of a SQL statement through ODBC –

- Preparation (pre-compilation) or direct execution of the SQL text
- Binding parameters or embedding the parameter values directly in the SQL text
- Binding output columns or retrieving query results on a column-by-column basis.

Preparation and both types of binding require more knowledge of the API and are a little more programming effort, but are usually more efficient (and for frequently-executed queries involving lots of data, substantially more efficient). In this initial portion of the presentation, we are going to discuss the simpler ways of doing things first. Both types of binding will be discussed later in the Advanced Topics section.



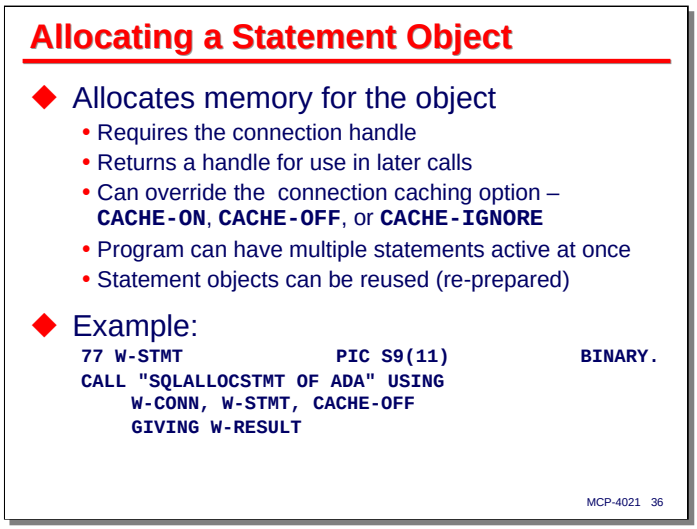
The easiest of the alternatives from the last slide to describe and understand is statement preparation vs. direct statement execution.

SQL queries can be complex. The driver and/or data source must parse the SQL text, convert it to an internal form, optimize the query, and prepare an execution plan. This can take considerable time and resources (although in an A-D-A environment, the resources will be consumed on the Windows proxy server and the server hosting the data source).

Therefore, direct query execution is usually best for statements that will be executed only once, or at least infrequently. All of the preparation mentioned above must still be done, but the results of preparation will be discarded after the statement execution has completed.

If you have SQL statements that will need to be executed repeatedly (especially queries), it is usually more efficient to prepare them in advance and execute the prepared statement. If you choose to bind parameter values rather than embed them in the SQL text, you can simply change the bound parameter values and re-execute the statement without preparing it again. Doing so can save considerable overhead.

Once you prepare a statement, the execution plan generally persists until you either deallocate the statement object or prepare against that object again using different SQL text. There is at present no way to save a prepared statement beyond the life of a program – each execution of the program must do its own preparation of statements.

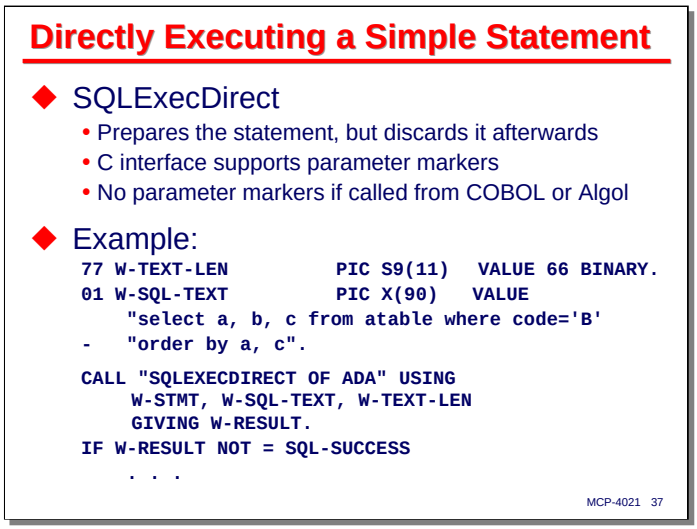


In order to execute a SQL statement, regardless whether you plan to do preparation or direct execution, you must allocate a statement object for it. Your program can allocate multiple statement objects and keep them active simultaneously. Statements are allocated in the context of a connection object. The allocation routine returns a statement handle, which you need to save and use in subsequent API calls involving that statement object.

When allocating a statement object, you can override the caching status of the connection object with which it is associated (which in turn could have overridden the caching status established by the **ODBCINIT** call). You can explicitly turn caching on or off, or use the **CACHE-IGNORE** constant to use the caching established for the connection object. This allows you to set a caching default for your entire program through the **ODBCINIT** call, a separate default for a connection, and still override it at the statement level.

If you prepare a SQL statement for a statement object, you can replace that prepared statement simply by doing another prepare against that object with different SQL text.

The example shows how the **SQLALLOCSTMT** function is used. You pass in the connection handle and one of the caching constants. The routine returns the statement handle.



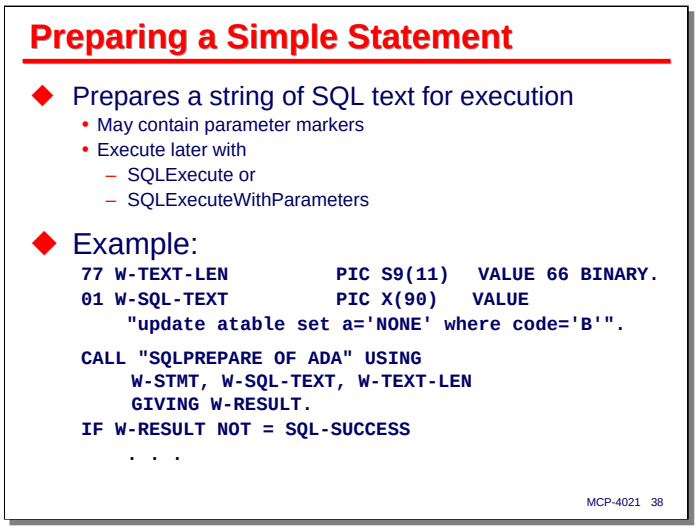
Once you have allocated a statement object and obtained its handle, you can execute a SQL statement. The easiest way to do this is with direct execution. As mentioned earlier, this type of call prepares the statement, executes it, and then discards the prepared statement.

Using the C language, you can bind parameter values when doing direct execution. COBOL and Algol handle bound parameters differently, due to their lack of support for C-type pointers. A-D-A provides a different function for direct execution in these languages, **SQLEXECDIRECTWITHPARAMETERS**, which will be discussed in the Advanced Topics section. Therefore, when using **SQLEXECDIRECT** with COBOL and Algol, parameter markers (which will be discussed in the section on bound parameters) are not permitted in the SQL text.

The example shows how the **SQLEXECDIRECT** function is used in COBOL. It requires three arguments:

- A statement handle
- An 01-record area containing the text of the SQL statement you wish to execute
- An integer variable holding the effective length of the SQL text.

The function returns only a success or failure status result. The data output from the statement must be retrieved separately, as will be discussed shortly.



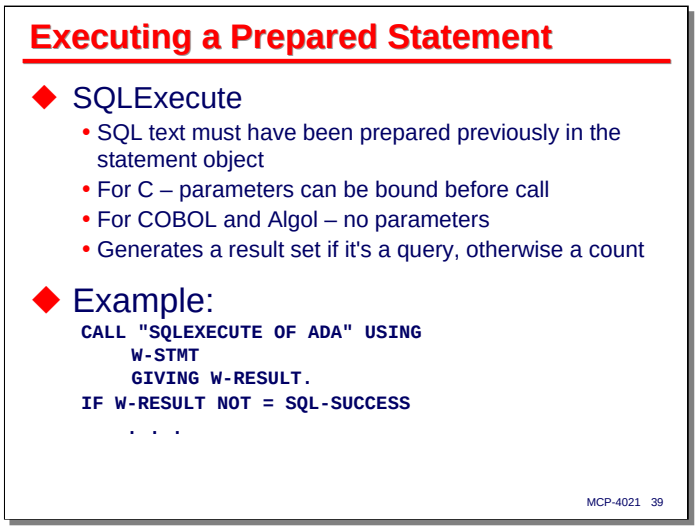
Instead of direct execution, you can prepare the SQL statement for later execution. This is identical to the call for direct execution, except that no execution takes place, and the execution plan for the statement is preserved in the statement object. In addition, the SQL text can contain parameter markers.

After preparing the statement, you execute it later using either **SQLEXECUTE** or **SQLEXECUTEWITHPARAMETERS**. The latter routine is required in COBOL or Algol if the SQL text contains parameter markers.

The calling sequence for **SQLPREPARE** is identical to that for **SQLEXECDIRECT**:

- A statement handle
- An 01-record area containing the text of the SQL statement you wish to prepare
- An integer variable holding the effective length of the SQL text.

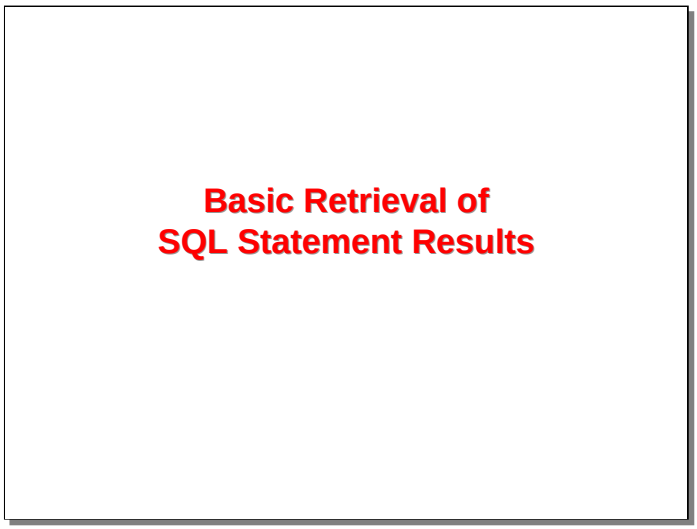
The function result indicates success or error. In particular, SQL syntax errors will return a result of **SQL-ERROR**.



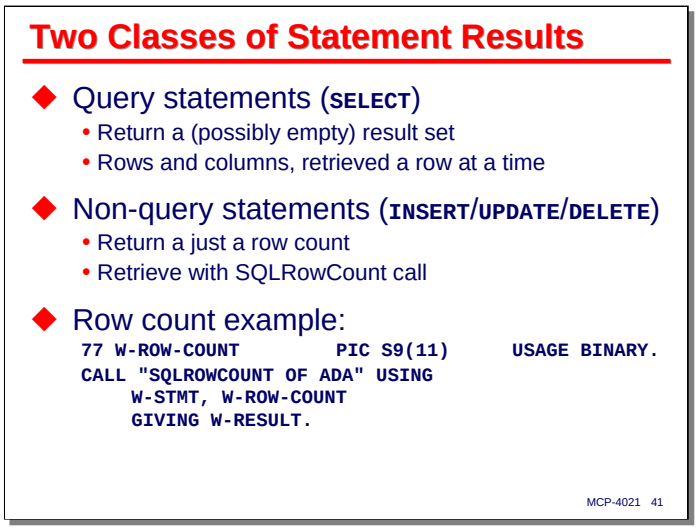
After preparing a statement, you can execute it multiple times. In the case of a statement without parameter markers, you use the **SQLEXECUTE** function. With C, you can bind parameters when using this function, but COBOL and Algol require a different function for statements with parameter markers.

Executing a statement generates a result set if the statement is a query, or a simple count of records affected if it is an update. Those results must be retrieved separately, as will be discussed next.

Calling **SQLEXECUTE** is very simple – you just pass the statement handle for a previously prepared statement.



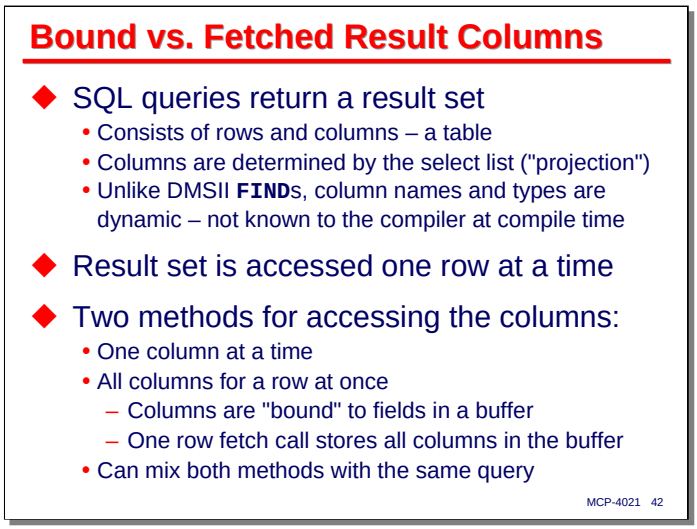
Thus far we have discussed preparing a statement and executing it (albeit only statements that do not have bound parameters at this point). The next step, assuming the execution of the statement was successful, is to retrieve the results.



The result from execution of a SQL statement falls into two classes:

- For query statements (i.e., the SQL **SELECT** statement or a stored procedure that executes one), the result is in the form of a *result set*, which is a table containing rows and columns of data. It is possible that this result set may be empty (the number of rows is zero). Data is retrieved from result sets one row at a time, similar to reading records from a sequential file.
- For non-query statements (i.e., SQL **INSERT**, **UPDATE**, and **DELETE** statements or a stored procedure that executes one of these), the result is simply a count of the number of records affected.

This latter case is the easiest to discuss. The **SQLROWCOUNT** function retrieves the count of affected rows. You simply pass it the statement handle and an integer variable to receive the count.



Result sets generated by SQL queries are more complex and require more effort to retrieve than do the simple count returned by update statements. As mentioned on the prior slide, result sets are tables, consisting of rows and columns. The rows are determined largely by the tables and filtering conditions (**`WHERE`**, **`GROUP BY`**, **`HAVING`** clauses) in the SQL text. The columns are determined by the list of column names and expressions in the **`SELECT`** clause, which is termed the *projection* of the query.

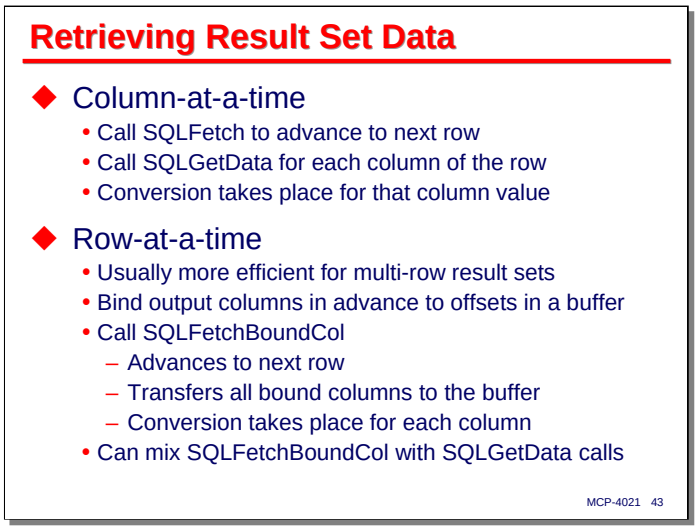
Unlike retrieval using the DMSII Host Language Interface, the columns and data types in a result set are dynamic and defined by the query text. Therefore the compiler cannot know about column names and types at compile time. Your program must deal with this dynamic data at run time.

Result sets are accessed one row at a time. You fetch the next row in sequence, then obtain the column values from that row.

There are two methods for accessing the columns of a row:

- One column at a time. This requires one function call to fetch the row and a separate function call for each column in the row to retrieve its value.
- All columns at once. This requires only one function call to fetch the row. The column values are automatically formatted into a buffer (record area). In order for this to work, your program must have previously "bound" the result set columns to positions and data types in the buffer. We will discuss this method in the Advanced Topics section.

Actually, the second method does not need to bind all columns to a buffer. You can mix the bound-columns and column-at-a-time approaches, but it is customary to use either one or the other for a particular result set.

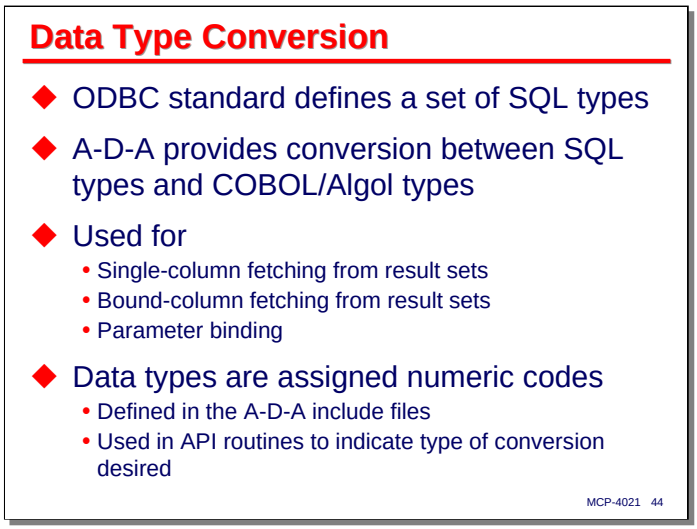


To retrieve columns one at a time after executing your query, you first call **SQLFETCH** to advance the result set cursor to the next row. Generally, the row fetch is written inside a loop. A result set is initially positioned before the first row, so you need to call this function for every row in the result set. After the last row has been fetched (or if the result set is empty), **SQLFETCH** will return the **SQL-NO-DATA-FOUND** (value=100) result.

Once you have fetched the row, you can then fetch the values for individual columns using **SQLGETDATA**. Note that in order to call this function successfully, caching for result sets must be disabled, which can be done through the **ODBCINIT**, **SQLALLOCCONNECT**, or **SQLALLOCSTMT** functions. As part of its duties, **SQLGETDATA** will convert the column value from its SQL data type to the data type specifications you provide as arguments to the function.

To retrieve columns a row at a time, you must first bind the output columns of the result set to offsets within a buffer (01-record area) in your program using the **SQLBINDCOL** function. You then fetch rows using the **SQLFETCHBOUNDCOL** function, which uses the previously-specified bindings to convert and store the result set column values into the buffer. Conversion takes place for each column as specified by arguments to the **SQLBINDCOL** function. We will discuss binding output columns in detail in the Advanced Topics section of this presentation.

As mentioned previously, it is not necessary to bind all columns of a result set to a buffer. You can bind only some of the columns, and use **SQLGETDATA** to obtain values for the rest. Just remember that whenever **SQLGETDATA** is used, caching must be disabled for the result set.



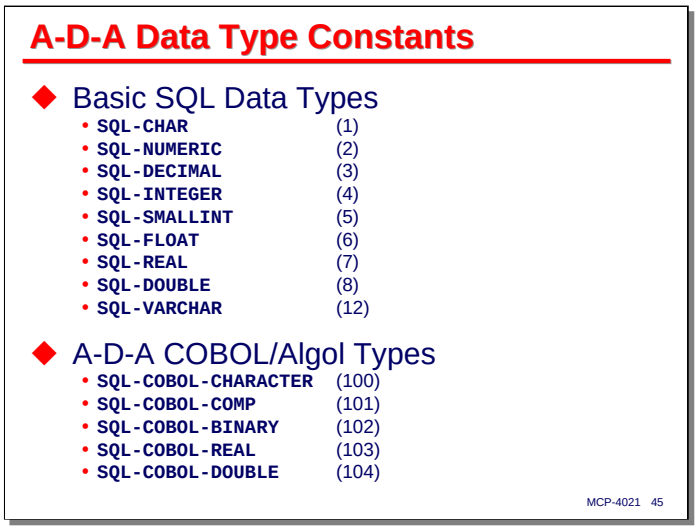
The subject of data conversion has come up a couple of times thus far, and before proceeding with the discussion on retrieving data from result sets, we need to cover conversion in more detail.

The ODBC standard defines a number of SQL data types, **`char`**, **`int`**, **`float`**, **`varchar`**, etc. A-D-A provides a facility to convert these SQL data types (which are what the ODBC driver and data source understand) to native MCP data types that can be used by COBOL and Algol programs.

This conversion facility is used in three places of the A-D-A API:

- When fetching single column results using **`SQLGETDATA`**.
- When specifying bound columns using **`SQLBINDCOL`**.
- When specifying parameter binding using **`SQLBINDPARAMETER`**.

The SQL and COBOL/Algol data types are each assigned numeric values, which in turn are represented by named constants in the language-specific include files. You use these named constants in calls to the three API functions mentioned above to specify the type of conversion you need. The next slide discusses these values.



The top portion of the slide shows the basic SQL data types supported by A-D-A. There are also values for extended types defined in the include file. The numbers in parentheses are the values defined by the data names.

The bottom portion of the slide shows the COBOL/Algol types supported by A-D-A.

- **SQL-COBOL-CHARACTER** signifies data that is COBOL **USAGE DISPLAY**, which typically implies EBCDIC character data. The values may be numeric or alphanumeric, but their representation is as characters. Values stored with this data type will be truncated or space-filled on the right as necessary for them to fit in the receiving data area that you specify.
- **SQL-COBOL-COMP** signifies COBOL **USAGE COMPUTATIONAL** (**COMP**), which is packed-decimal numeric data. All packed-decimal values for use with A-D-A *must* have signs. Unsigned numeric data is not supported.
- **SQL-COBOL-BINARY** signifies COBOL **USAGE BINARY** or **BINARY EXTENDED**. This is an integer value stored in a 48-bit binary field, although it need not be word-aligned in memory. It is also always signed.
- **SQL-COBOL-REAL** signifies COBOL **USAGE REAL**, which is the 48-bit single-precision floating-point representation used by MCP systems. As with **BINARY** data, it need not be word-aligned in memory.
- **SQL-COBOL-DOUBLE** signifies COBOL **USAGE DOUBLE**, which is the 96-bit double-precision floating-point representation used by MCP systems. It consists of 12 contiguous bytes in memory, and need not be word-aligned.

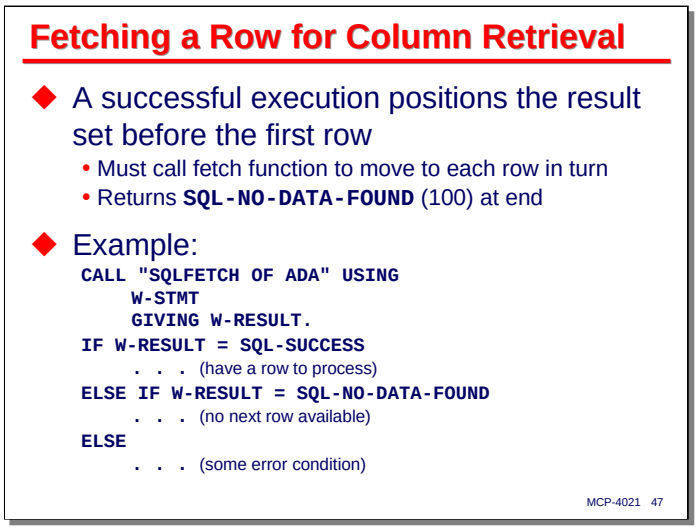
A-D-A Data Type Conversions

	COBOL Data Type (USAGE)				
SQL Type	**CHAR**	**COMP**	**BINARY**	**REAL**	**DOUBLE**
Char	X				
VarChar	X				
Decimal	X	X			
Numeric	X	X			
Smallint	X	X	X		
Integer	X	X	X		
Real				X	
Double				X	X

From A-D-A User's Guide, Appendix B

MCP-4021 46

The chart on this slide shows the data type conversions supported by A-D-A. SQL character data and numeric values can be converted to MCP character values, SQL numeric values can be converted to MCP packed-decimal and binary integer values, and the floating-point types can only be converted among themselves.

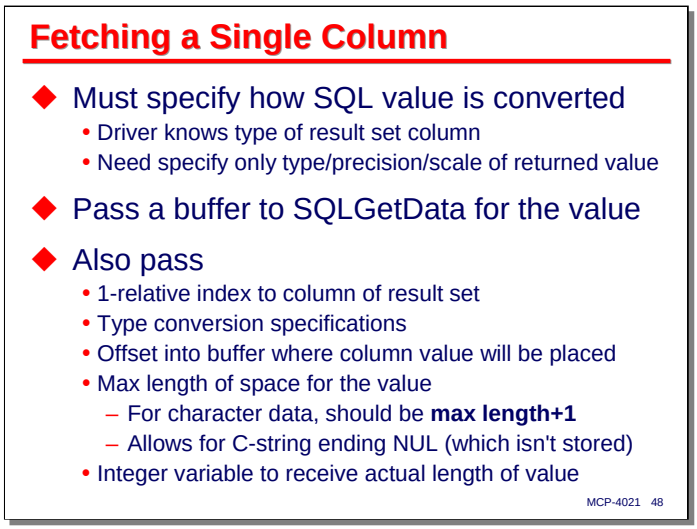


With that discussion of data conversion completed, we can return to the topic of column-by-column data retrieval from result sets.

When a SQL query statement executes successfully, it generates a result set and positions the cursor before the first row of the result set. You must perform "fetch" calls to position the cursor to each row in turn. After the last row has been fetched (or if the result set was empty, due to the query not returning any data), the fetch call returns a result value of **SQL-NO-DATA-FOUND** (which has a numeric value of 100). That is effectively an EOF signal for the result set.

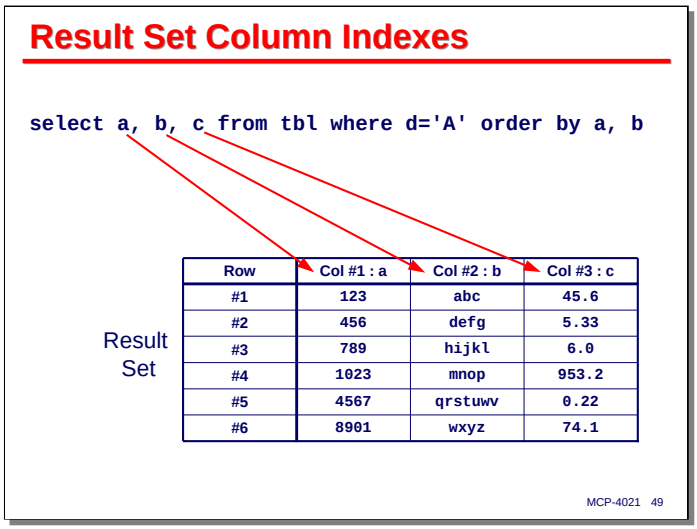
The slide shows an example of the skeletal code required to fetch rows for column-by-column retrieval. This code would normally be embedded in a loop, to sequentially retrieve all of the rows from the result set.

You call the **SQLFETCH** function, passing the statement handle you used to prepare and execute the SQL query. All that this call does is position the cursor to the next row in the result set and return a success or failure result to your program.

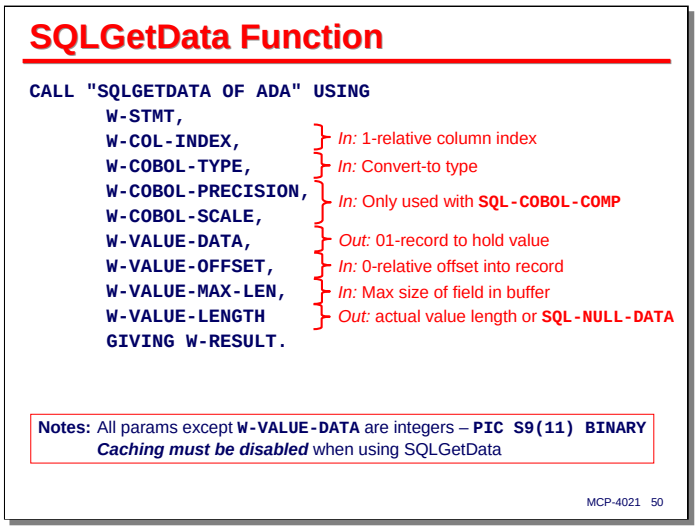


Once you have fetched a row from the result set, you can proceed to retrieve the data for its columns. As part of that retrieval, you must specify how you want the SQL data type for that column to be converted for storage in your program. The **SQLGETDATA** function is used to retrieve individual columns. You pass the following arguments to this function:

- The statement handle
- A buffer (COBOL 01-record area or Algol **EBCDIC ARRAY**) to this function to receive the converted value
- A one-relative index to the column of the result set you want to retrieve
- Some values that specify how you want the value to be converted. These will be detailed over the next couple of slides
- A zero-relative offset into the buffer where the column value will be stored
- For character data, the maximum number of characters that can be stored at this position in the buffer. For non-character data (**COMP**, **BINARY**, etc.) the value in this field is ignored. There is a quirk to specifying this value that comes from the C-oriented nature of the ODBC API. C terminates character strings with a **NUL** (hex 00) character. A-D-A never stores this character in your buffer, but it does account for it in determining the maximum length of the value's area in the buffer. As a result, you must specify a length one greater than the maximum length you actually expect to be stored in the buffer. For example, if you have a field in the database that is a maximum of 30 characters long, you need to specify a length for your buffer of 31. If you specify the maximum length as 30, and you attempt to retrieve a character value that is actually 30 characters long, it will be truncated after the 29th character. Whether this behavior is a feature or a bug is perhaps something over which reasonable people can disagree, but it's the way it works. Truncation of data will result in the **SQLGETDATA** function returning a value of **SQL-SUCCESS-WITH-INFO** (1), but the truncated value will be stored in your buffer.
- Finally, you must pass an integer variable that will receive the actual length of the data stored in your buffer. This argument is relevant only for character data, except that if the value retrieved is null, the value **SQL-NULL-DATA** (-1) will be stored in the variable.

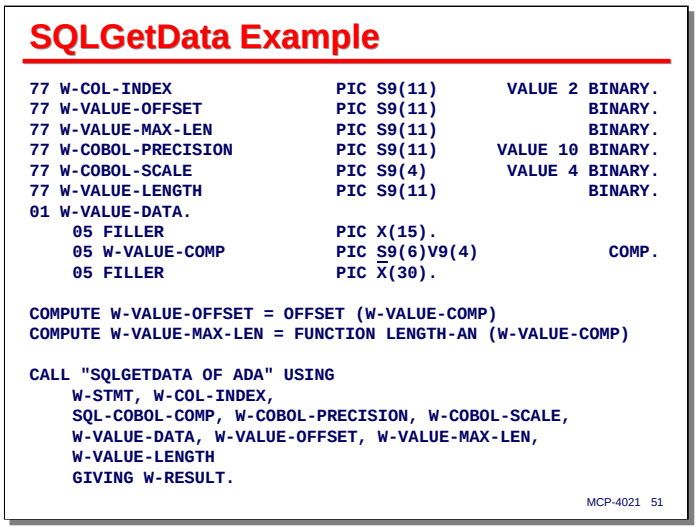


The result columns (the "projection") from a SQL query have names, but the ODBC API requires you to retrieve them by their one-relative column number instead of their name. This slide shows the relationship between the columns and expressions in a SQL **SELECT** statement and their corresponding column numbers. The result set columns are simply numbered from left to right, starting with one for the first column.



This slide discusses the **SQLGETDATA** function's calling sequence in more detail. The arguments are:

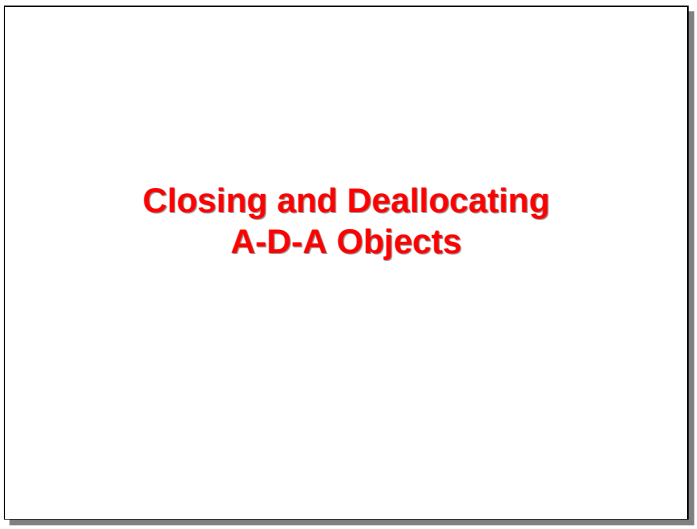
- The statement handle you used to execute the query and perform the row fetch
- The one-relative column index for the column you wish to retrieve
- The MCP data type you wish the value to be converted to, **SQL-COBOL-CHAR**, **SQL-COBOL-COMP**, etc.
- The next two arguments are only used with packed decimal data (**SQL-COBOL-COMP**). The values for these arguments are ignored for other data types.
  - Precision indicates the total number of digits in the converted value, not including the sign. All packed-decimal data used with A-D-A must have a sign.
  - Scale indicates the number of fractional digits in the converted value. A COBOL picture of **S9(6)V9(4)** equates to a precision of 10 and a scale of 4.
- The buffer to receive the converted value. This must be a 01-record area in COBOL and an **EBCDIC ARRAY** in Algol.
- The zero-relative offset into the buffer where the value will be stored. Note that COBOL is usually one-relative, but this must be a zero-relative offset. The next slide shows how this offset can be computed efficiently.
- An integer variable holding the maximum length of the converted data. As mentioned previously, this value is used only for character data, and must be at least one greater than the maximum length you expect to store. Character strings longer than this value will be truncated.
- An integer variable that will receive the actual length of the converted data. This value is also only relevant to character data, except that if a column of any type is null, **SQL-NULL-DATA** (-1) will be stored in the variable.



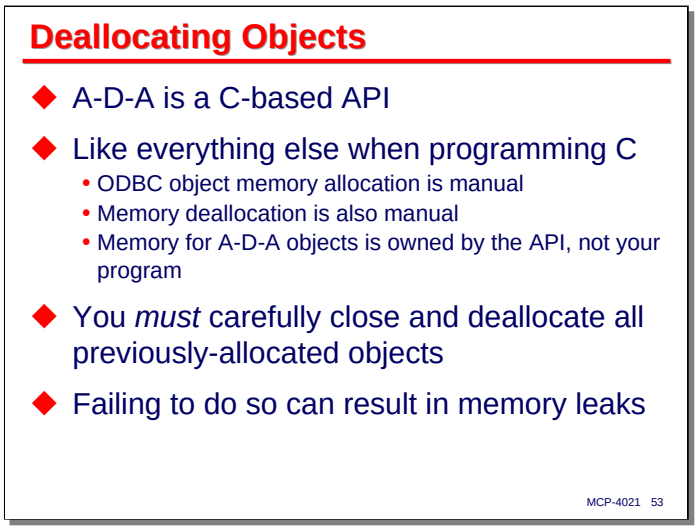
This slide shows an example of a call on **SQLGETDATA**. Note that in both COBOL-74 and COBOL-85, there is an **OFFSET** function (a Unisys extension to COBOL) that will return the zero-relative offset to the beginning of the storage for a data name within its 01-record. For scalar data items, this offset is evaluated at compile time, so the **OFFSET** function is very efficient. For indexed data items, the offset must be computed at run time, but it still relatively efficient.

To determine the length of a data value, in COBOL-85 you can use **FUNCTION LENGTH-AN**. This returns number of 8-bit bytes preceding the data field in its 01-record area. This function is also very efficient, and in most cases can be evaluated at compile time. There is a similar **FUNCTION LENGTH**, which returns the number of *characters* (not bytes) preceding the data field. For eight-bit character sets (i.e., all of those used by Western languages) **LENGTH** and **LENGTH-AN** return the same result. They return different values when multi-byte character sets are in effect.

COBOL-74 does not support **FUNCTION LENGTH** or **LENGTH-AN**, but the byte-length of a data field can be obtained by using **FUNCTION FORMATTED-SIZE (…)** instead.

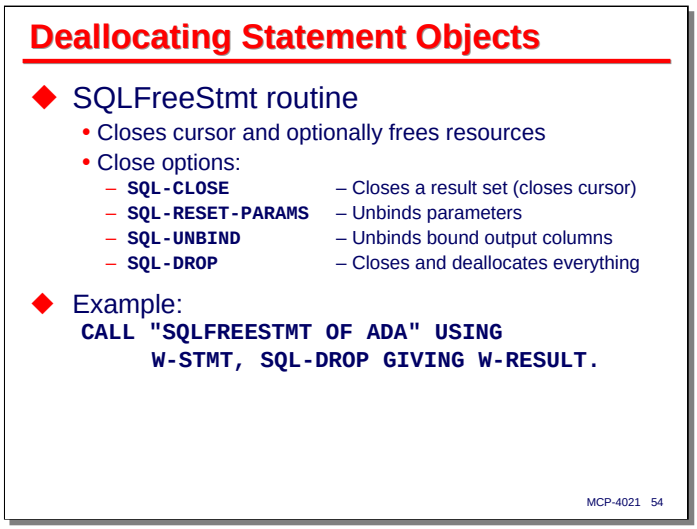


We have been talking about allocating and using objects for the ODBC API. The last subject in this section on basic A-D-A usage is how you close and deallocate those objects when you are finished with them.



A-D-A is based on the C-oriented ODBC API. Like everything else when programming with C, memory allocation is done manually. More importantly, memory deallocation is also done manually. The environment, connection, and statement objects you allocate in your program are owned by the A-D-A library, not by your program. All that your program owns is a copy of the handle value used to reference an object.

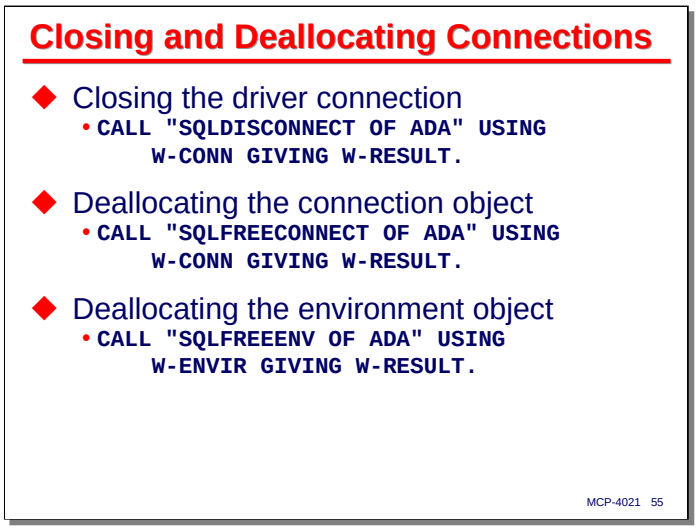
When you are finished using an object, you *must* close and deallocate them. Failing to do so can result in memory leaks.



We will discuss deallocation of objects in the reverse order we discussed allocating them. The first of these is the statement object.

Statement objects are handled somewhat differently from the other ones. One function call, **SQLFREESTMT**, performs four different functions. You call this routine passing a statement handle and one of four constant values from the include file for your language. These constant values indicate the following actions:

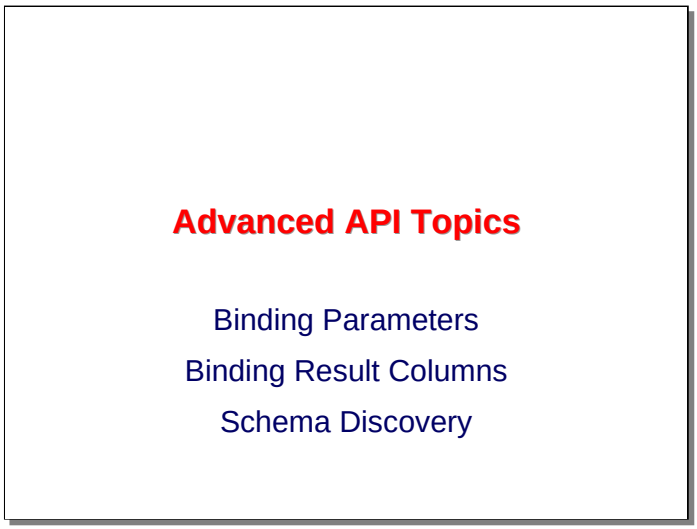
- **SQL-CLOSE** – This type of call simply closes any cursor (result set) associated with the statement. You can use this option if you are finished fetching rows from a statement's result set and want to free those resources.
- **SQL-RESET-PARAMS** – This type of call will unbind any parameter bindings associated with the statement object.
- **SQL-UNBIND** – This type of call will unbind any bound output columns associated with the statement object.
- **SQL-DROP** – This type of call closes and unbinds everything associated with the statement, then deallocates all memory for the object. You must make this type of call before exiting your program, and before freeing any connection and environment objects.



Connections to an ODBC data source are closed by calling the **SQLDISCONNECT** function. You simply pass the connection handle to the routine.

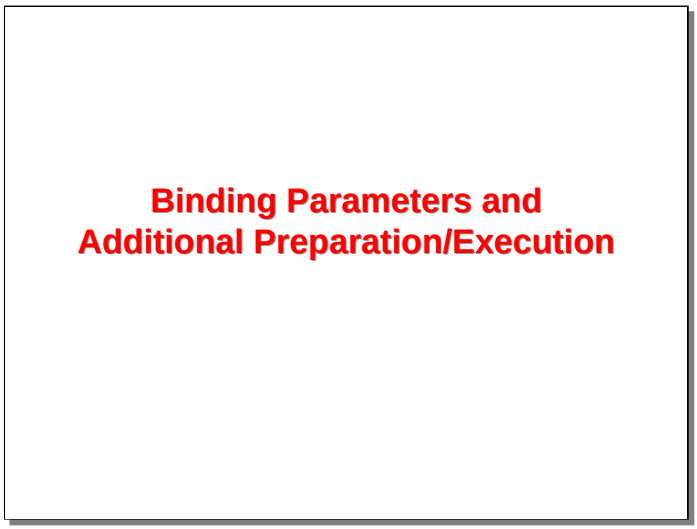
After closing a connection, you deallocate the memory for its object by calling the **SQLFREECONNECT** function. You also pass the connection handle to this function.

Finally, you deallocate the environment object by calling **SQLFREEENV**, passing the environment handle.

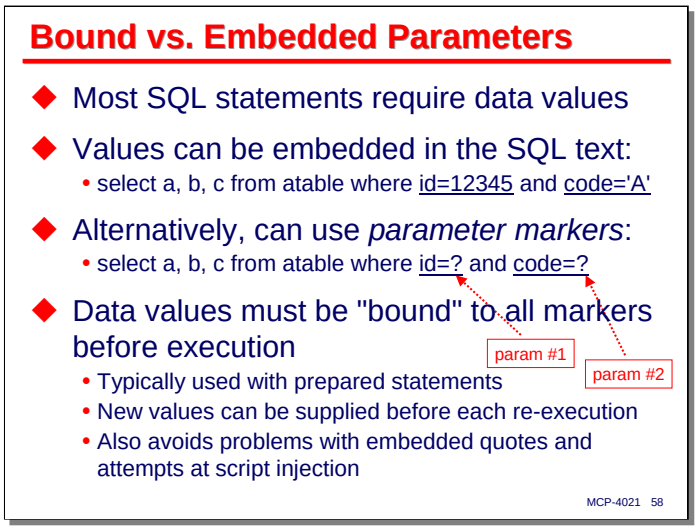


That concludes the discussion of the basic A-D-A API. In this next section, we will discuss a few more advanced topics, particularly binding parameters to a statement, binding output columns for result sets, and (briefly) the functions that will discover the schema associated with a data source.

Both of the binding subjects are very important, and are ones you will want to master for everyday use of Application Data Access.



In this section, we'll discuss binding parameters to statements, along with some additional functions related to statement preparation and execution.



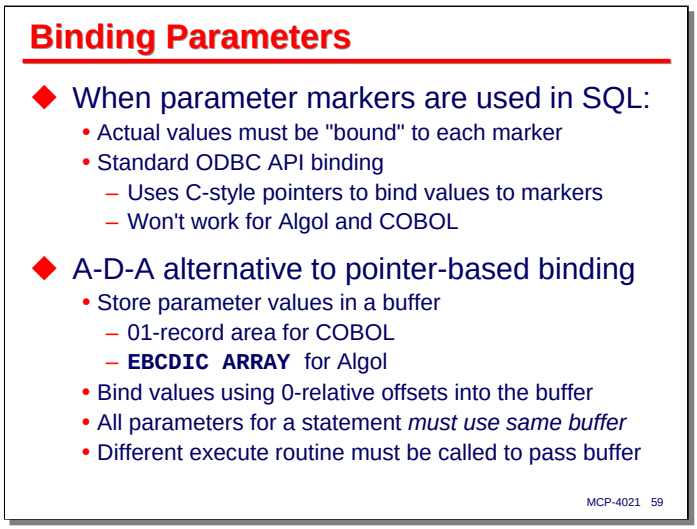
Most SQL statements require data values of some sort, whether as values used in query predicates or values used to set columns in **`INSERT`** and **`UPDATE`** statements. These data values can be specified in two ways:

- Embedded literally in the SQL text.
- Provided externally and represented in the SQL text as a "parameter marker." These markers are simply a question mark (**?**) written in the place where the parameter value will eventually be inserted. Parameter markers are numbered sequentially, starting at one, according to their linear position in the SQL text. The first question mark represents parameter #1, the second one parameter #2, etc. Note that the marker is a separate token, so that question marks in literal strings are interpreted literally, not as a parameter marker. Also note that you can mix the use of literal values and parameter markers in the same SQL statement.

Before you execute a SQL statement that contains parameter markers, you must "bind" an actual value to each marker. We will see how this is done over the next few slides.

Parameter markers are typically used with prepared statements. Their use allows the prepared statement to be executed multiple times without re-preparation. When literal values are embedded in SQL text, the statement must be re-prepared if those values are changed. Bound parameters allow you to decouple the value from the SQL text, supply it at execution time, and change it so that a prepared statement can be re-executed efficiently with different parameter values without the overhead of re-preparation.

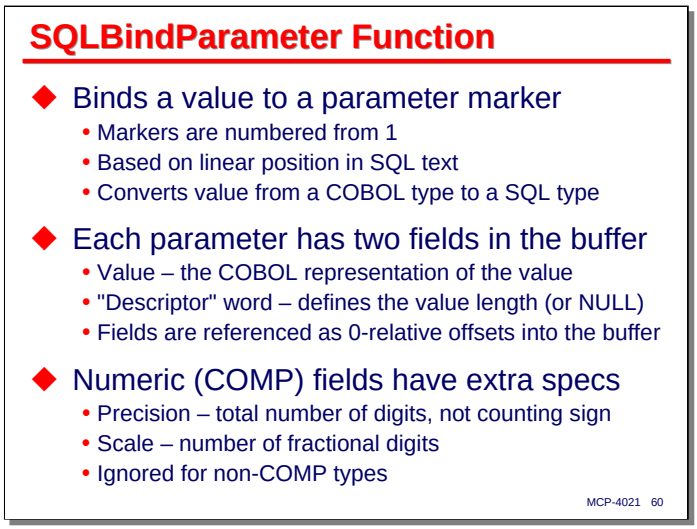
Another advantage of bound parameters is that they avoid potential parsing problems such as embedded quotes. In addition to requiring special handling when used with literal string values, improper handling of embedded quotes can enable script injection attacks when a literal value obtained externally (e.g., from a user input) is inserted into SQL text.



The standard method for binding parameters in the ODBC API involves C-style pointers. This will not work for COBOL and Algol, since they do not support C-style addressing.

In order to support parameter binding for COBOL and Algol, A-D-A provides a different model for binding parameter values to parameter markers. Instead of addressing parameter values using pointers (which allows the values to be stored anywhere in the application program's address space), the API for COBOL and Algol require that all parameter values for a given statement be stored in a single buffer. For COBOL, this buffer is an 01-level record area. For Algol, the buffer is an **`EBCDIC ARRAY`**.

Instead of binding using pointer addresses, the buffer method binds columns using zero-relative offsets into the buffer's memory area. In addition, a different execute function must be called in order to pass this buffer. This is why all parameter values must use the same buffer – only one buffer area can be passed to the execute function.



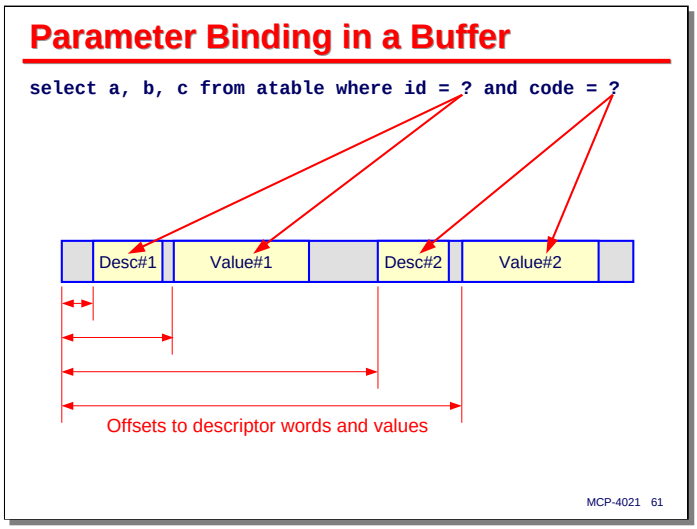
Parameter binding is established using the **SQLBINDPARAMETER** function. We speak of this function as binding a value to a statement, but what it actually does is bind the *position* of a parameter's value in its buffer to a particular parameter marker. As mentioned previously, the markers are numbered sequentially starting at one. A separate **SQLBINDPARAMETER** call is required for each parameter to be bound. In addition to establishing a relationship between a parameter marker and a position in the parameter value buffer, this function specifies how the actual value will be converted from its COBOL or Algol representation to the representation that will be used by the ODBC driver.

Each parameter marker is associated with two fields in the buffer. One is the field that will hold the actual parameter value. The size of this field is determined by the size and type of the parameter value. The second field is a "descriptor" word, consisting of six bytes of binary data, but not necessarily word-aligned in memory. This word is termed a "pcbValue" by the A-D-A documentation, with "pcb" originally meaning "pointer to control block." For character data, the value of the descriptor word is the actual length of the character string. For other data types the length is implied by the data type. For all data types, however, a special value, **SQL-NULL-DATA** (-1) can be stored in the descriptor word to indicate the parameter has a null value.

Numeric fields represented as COBOL **COMP** (packed decimal) values have two additional specifications that must be specified when binding them as parameters:

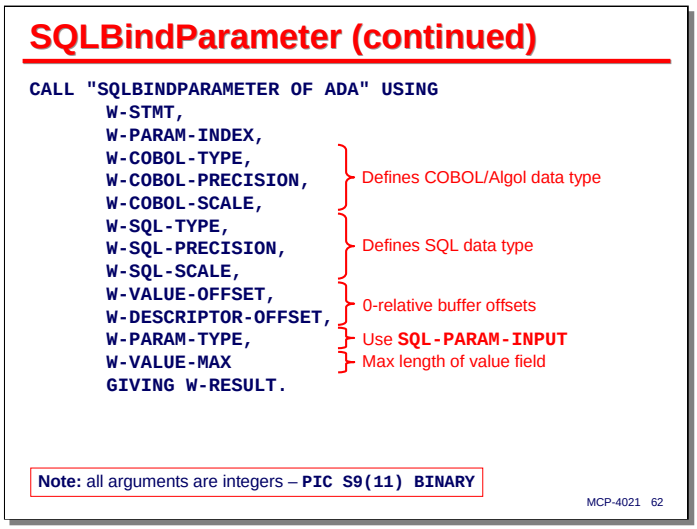
- Precision is the total number of digits in the number, not including the sign. All packed-decimal values used with A-D-A must be signed, however.
- Scale is the number of fractional digits. A COBOL picture of **S9(5)V9(2)** would have a precision of 7 and a scale of 2.

The precision and scale attributes are the same as those used with the **SQLGETDATA** function. Their values are ignored for non-**COMP** data types.



The diagram on this slide shows the relationship between parameter markers and their two fields in the buffer area. The first marker is associated with a descriptor word and value field for its value. The second marker is associated with its descriptor word and value field, and so on.

As we will see on the next slide, you must specify the zero-relative offset from the beginning of the parameter buffer to each of the fields for a given marker. This slide shows the descriptor word preceding the value field in the buffer, but the descriptors and values can be in any order and at any position in the buffer. A common alternative is to place all of the descriptor words together in either the front or back of the buffer, and all of the value fields together, one after the other, in the opposite end of the buffer.



This slide shows the calling sequence for the **SQLBINDPARAMETER** function. This looks daunting at first, but it breaks down into a meaningful sequence of values. You must pass the following arguments to this function:

- The statement handle against which you will execute the SQL text
- The value of the one-relative parameter number
- The next three arguments define the COBOL/Algol representation of the parameter value in the buffer area:
  - The COBOL/Algol data type (**SQL-COBOL-CHAR**, **SQL-COBOL-COMP**, etc.)
  - The precision of a **COMP** value. The value of this argument is ignored if the type is not **SQL-COBOL-COMP**
  - The scale of a **COMP** value. The value of this argument is also ignored if the type is not **SQL-COBOL-COMP**
- The next three arguments define the representation of the parameter value as it will be converted for use by the ODBC driver:
  - The SQL data type (**SQL-CHAR**, **SQL-NUMERIC**, etc.)
  - The precision of a numeric value. The value of this argument is ignored if the type is not **SQL-NUMERIC** or **SQL-DECIMAL**
  - The scale of a numeric value. The value of this argument is also ignored if the type is not **SQL-NUMERIC** or **SQL-DECIMAL**
- The zero-relative offset to the value field for the parameter's actual value
- The zero relative offset to the descriptor word for the parameter
- A value that indicates the mode of the parameter. ODBC provides for both input and output parameters, but A-D-A only supports input parameters, so you should always pass the value **SQL-PARAM-INPUT** for this argument.
- A value that specifies the maximum number of bytes the parameter value can occupy. This argument is relevant only for character data types. For other data types, the length is determined by the data type or from the precision attribute.

Note that all arguments to this function are binary integers, or in COBOL terms, **PICTURE S9(11) USAGE BINARY**.

SQLBindParameter Example

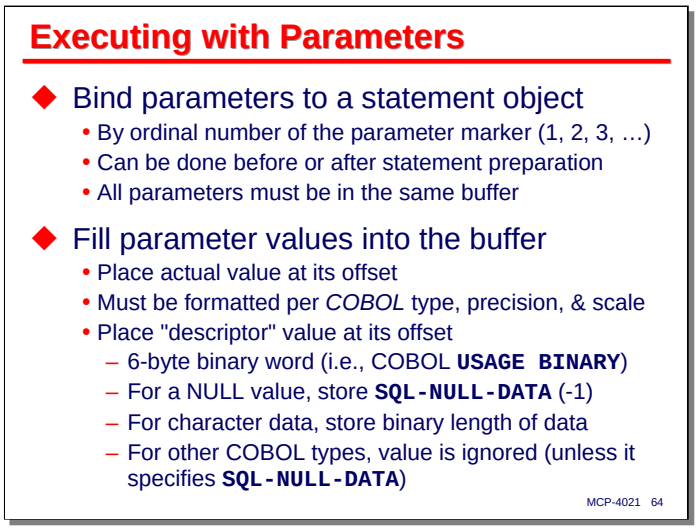
```
77 W-PARAM-INDEX              PIC S9(11)       VALUE 1 BINARY.
77 W-VALUE-OFFSET             PIC S9(11)               BINARY.
77 W-VALUE-MAX-LEN            PIC S9(11)               BINARY.
77 W-DESC-OFFSET              PIC S9(11)               BINARY.
01 W-PARAM-DATA.
    05 W-COMP-DESC            PIC S9(11)               BINARY.
    05 W-COMP-VALUE           PIC S9(6)V9(3)             COMP.
    05 W-CHAR-DESC            PIC S9(11)               BINARY.
    05 W-CHAR-VALUE           PIC X(30).

COMPUTE W-VALUE-OFFSET = OFFSET (W-CHAR-VALUE)
COMPUTE W-VALUE-MAX-LEN = FUNCTION LENGTH-AN (W-CHAR-VALUE)
COMPUTE W-DESC-OFFSET = OFFSET (W-CHAR-DESC)
MOVE W-VALUE-MAX-LEN TO W-CHAR-DESC  *> actual value length

CALL "SQLBINDPARAMETER OF ADA" USING
    W-STMT, W-PARAM-INDEX,
    SQL-COBOL-CHAR, SQL-NULL-DATA, SQL-NULL-DATA,
    SQL-CHAR, SQL-NULL-DATA, SQL-NULL-DATA,
    W-VALUE-OFFSET , W-DESC-OFFSET,
    SQL-PARAM-INPUT, W-VALUE-MAX-LEN GIVING W-RESULT.
```

MCP-4021 63

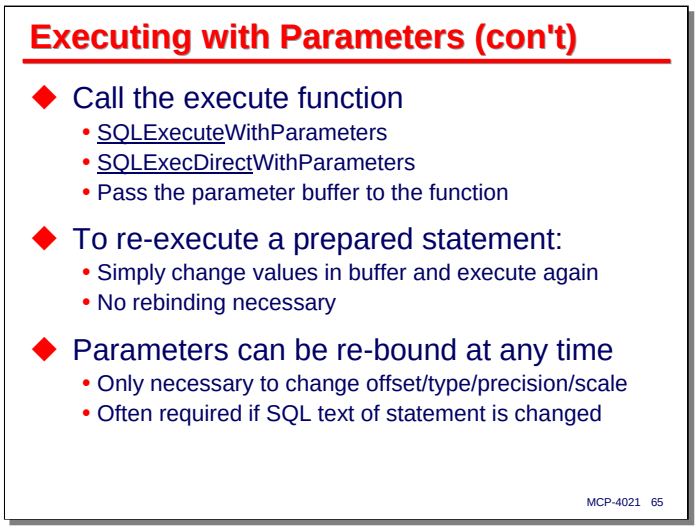
This slide shows an example of the data declarations and calling sequence for **SQLBINDPARAMETER**. As with the **SQLGETDATA** example shown earlier, you can use the COBOL-85 **OFFSET** and **FUNCTION LENGTH-AN** intrinsics to obtain the zero-relative offset and byte length of a data item. In COBOL-74, **OFFSET** is also supported, but you must use the **FUNCTION FORMATTED-SIZE** intrinsic instead of **FUNCTION LENGTH-AN**.



When you execute a SQL statement that contains parameter markers, you must first bind the markers to positions in the parameter buffer according to their one-relative sequence in the SQL text. This binding can be done before or after statement preparation, as the bindings and markers are matched up at execute time. As previously mentioned, all parameter bindings must be to positions in the same buffer area.

You must also store the actual values of the parameters in the buffer area before executing the statement. These values can be established before or after the bindings are established, but generally they are done just before the statement is executed.

- The value in the buffer must be formatted according to the COBOL type (and if of type **COMP**, the precision and scale attributes) specified in the **SQLBINDPARAMETER** call
- The value of the descriptor word must also be stored at its offset in the buffer area. For character data, the binary descriptor value indicates the actual length of the parameter value. For other data types, this value is ignored, unless it has the value **SQL-NULL-DATA** (-1).



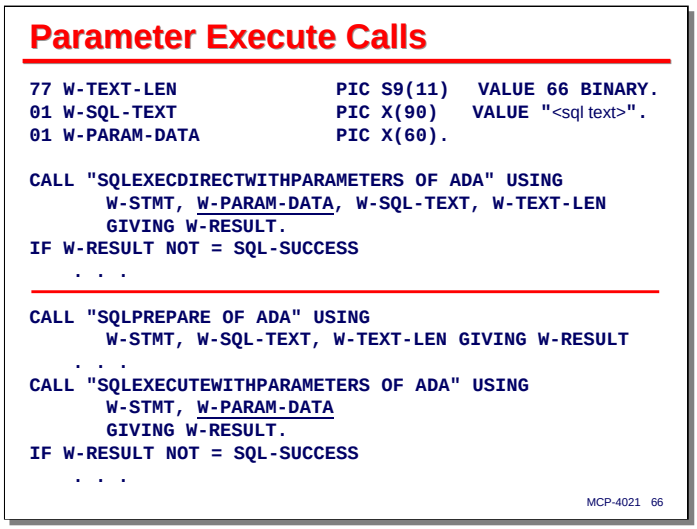
Once the parameter values are established in the buffer, you call one of the execute functions that are used with parameters:

- **SQLEXECUTEWITHPARAMETERS** is the equivalent of **SQLEXECUTE**, and is used with previously-prepared statement objects
- **SQLEXECDIRECTWITHPARAMETERS** is the equivalent of **SQLEXECDIRECT**, and is used when you want to prepare and execute a statement in one step.

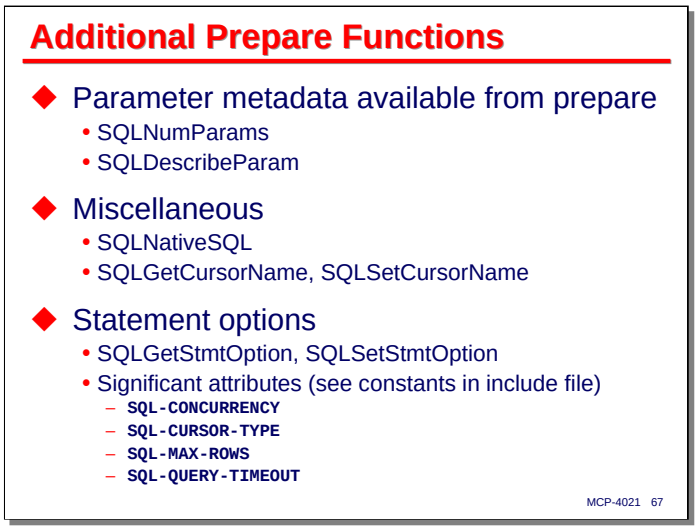
Both of these functions accept an additional argument for the parameter buffer area.

If you execute a prepared statement with parameters, you can re-execute that statement simply by moving new values into the parameter buffer area. It is not necessary to re-bind the parameters, because the offsets and data types have not changed. This is the most efficient way to execute a SQL statement many times with different parameter values.

You can also rebind the parameters at any time. This is normally necessary only if you change the SQL text of the statement, or for some reason need to change the offset and data type attributes of the parameter value. You can also unbind all parameters by calling **SQLFREESTMT** specifying an option value of **SQL-RESET-PARAMS**.

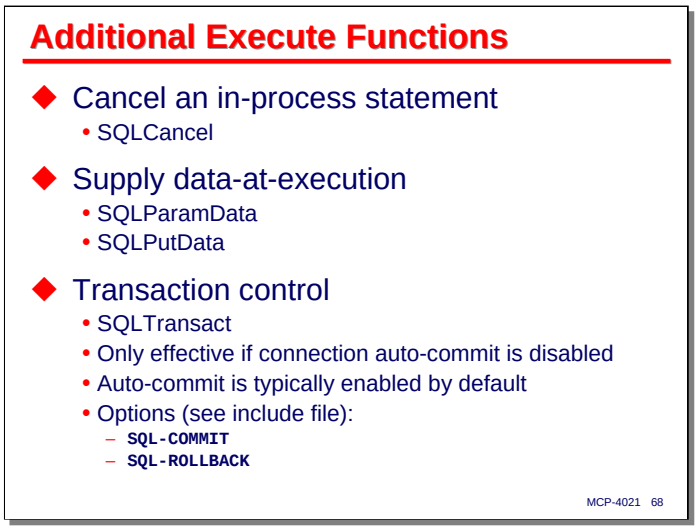


This slide shows examples of the two parameter-enabled forms of execute functions. Note that in both cases, the second argument to the function is a reference to the parameter buffer.



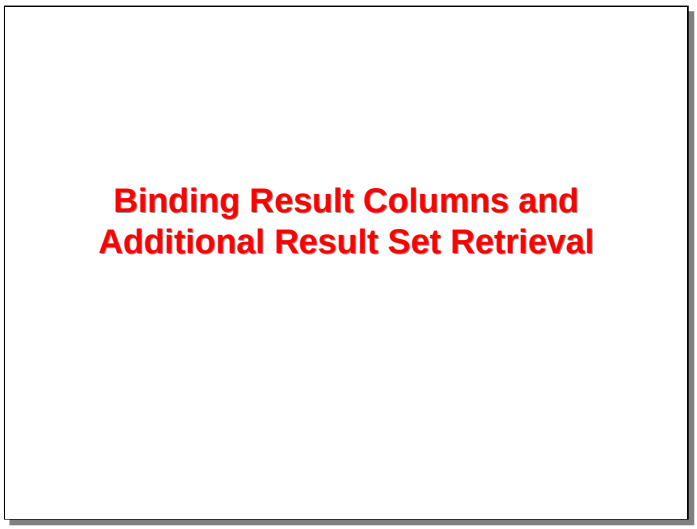
There are some additional API functions related to statement preparation that you may be interested in exploring.

- After you prepare a statement, the **SQLNUMPARAMS** and **SQLDESCRIBEPARAM** functions will provide information about any parameter markers the SQL parser found in the text. These functions are useful in an environment, such as a general-purpose query program, where the SQL text may be coming from an external source (e.g., user input).
- **SQLNATIVESQL** will return the text of the SQL statement as it has been transformed by the ODBC driver from the ODBC standard dialect to the native dialect of the data source.
- The cursors associated with query result sets may be given names, which can be set and retrieved using the **SQLSETCURSORNAME** and **SQLGETCURSORNAME**. If you do not assign a name to a cursor, the ODBC driver will assign one for you.
- There are a number of options associated with statement objects, as detailed in the Microsoft ODBC documentation. **SQLGETSTMTOPTION** and **SQLSETSTMTOPTION** can be used to interrogate and change values for these options. The various options have constant names defined in the include files. Four statement options of particular interest are shown on the slide:
  - **SQL-CONCURRENCY**
  - **SQL-CURSOR-TYPE**
  - **SQL-MAX-ROWS**
  - **SQL-QUERY-TIMEOUT**

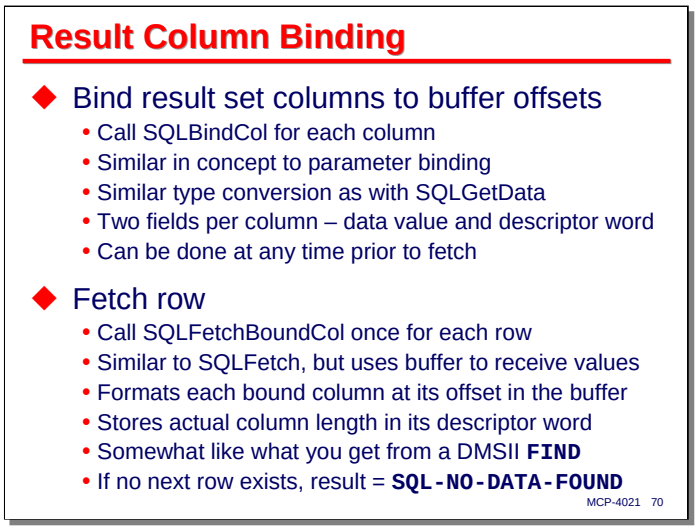


There are also some additional functions for statement execution that deserve mention:

- **SQLCANCEL** can cancel a statement that is in the process of execution.
- There are two functions, **SQLPARAMDATA** and **SQLPUTDATA**, that are used to supply data as a statement is being executed. These are used with large object (LOB) parameters and data columns.
- **SQLTRANSACT** is used to control transaction commit and rollback. This function is only applicable when the connection's auto-commit option is disabled (which for most data sources is enabled by default). With auto-commit enabled, each SQL statement executed in its own transaction, and a commit is performed implicitly when the statement finishes. When auto-commit is disabled, the first SQL statement to be executed starts a transaction. This transaction continues until **SQLTRANSACT** is called to either commit or roll back the transaction. The next SQL statement to be executed starts a new transaction, and so forth.



That concludes the discussion of parameter binding and additional prepare/execute functions. The next subject is result column binding and some additional functions for retrieving data from result sets.



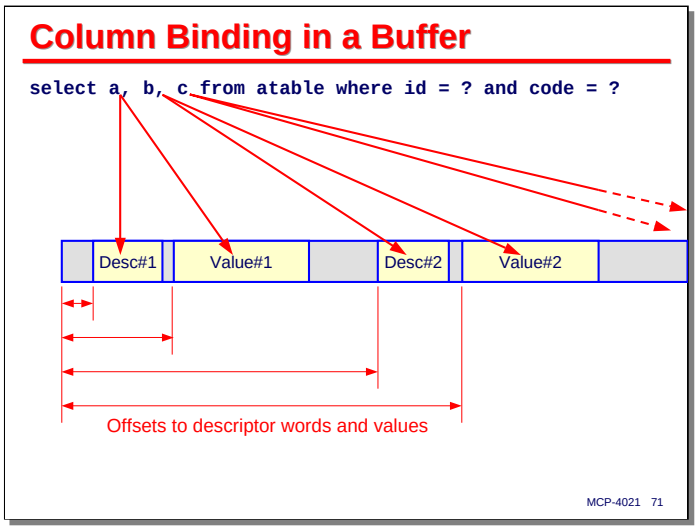
Earlier in this presentation we looked at one method for retrieving data from an SQL query's result set – doing **SQLFETCH** calls to move the result set cursor to the next row and then calling **SQLGETDATA** to retrieve values for individual columns. This is fine for retrieving small amounts of data, but there is another way – output column binding.

You bind output columns using the **SQLBINDCOL** function. The technique is similar to parameter binding, except that it works in reverse. Instead of your program supplying multiple parameter values at specified offsets in a buffer, the result set provides your program column values at specified offsets in a buffer. You also get data type conversion with output column binding, similar to that for **SQLGETDATA**. As with parameter binding, there are two fields per column in the output buffer, a data value and a descriptor word. Output column binding can be established at any time prior to the first row fetch – before statement preparation, between preparation and execution, or even after execution of the statement.

When using bound output columns, you call **SQLFETCHBOUNDCOL** instead of **SQLFETCH** to position the result set cursor to each row in turn. The big difference is that in addition to advancing the cursor, **SQLFETCHBOUNDCOL** also formats values for each of the bound output columns and moves them into a buffer area that your application passes to the function. The descriptor word, as in parameter binding, is used to report the actual length of character values or the null status of that column's data.

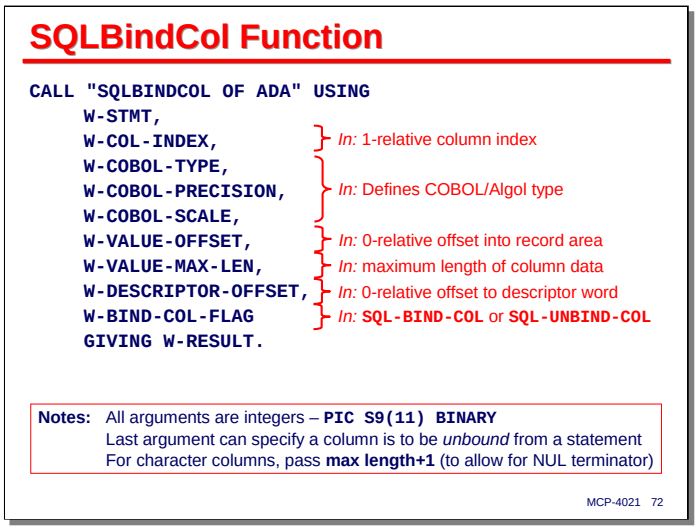
You can arrange output column binding to store converted column values into your application's buffer in a way that is very similar to the record area you get from a DMSII **FIND** statement. This can make supplying query results to the rest of our program much more convenient. Bound output columns are usually more efficient than individual **SQLGETDATA** calls, as well, especially since you can enable row caching on the statement object.

As with the **SQLFETCH** function, **SQLFETCHBOUNDCOL** returns the **SQL-NO-DATA-FOUND** result when there are no more rows in the result set.



The diagram on this slide shows the relationship between output columns in a SQL **SELECT** statement (the "projection" of the query) and how they are bound to offsets in the buffer area. As with parameter binding, each output column you bind is associated with two fields in the output buffer – a value field and a descriptor word. You specify the locations of these two fields using zero-relative offsets into the buffer area. Descriptors and value fields can be positioned and grouped in any way you like. This diagram shows the descriptor words next to their value fields, but you can just as well group all of the descriptor words together and all of the value fields together.

Note that you are not required to bind all output columns for a result set or to use output column binding exclusively. You can bind only some of the columns and use **SQLGETDATA** to retrieve values for other columns. Whenever you use **SQLGETDATA**, however, you must have caching disabled on the statement object, so typically bound output columns are used exclusively, especially when the result set is expected to contain many rows.



This slide shows the arguments to **SQLBINDCOL** required to establish one bound column. You pass the following values:

- The handle for the statement you are using to execute the query.
- The one-relative index for the column you wish to bind.
- The MCP data type you wish the value to be converted to, **SQL-COBOL-CHAR**, **SQL-COBOL-COMP**, etc.
- The next two arguments are only used with packed decimal data (**SQL-COBOL-COMP**). The values for these arguments are ignored for other data types.
  - Precision indicates the total number of digits in the converted value, not including the sign. All packed-decimal data used with A-D-A must have a sign.
  - Scale indicates the number of fractional digits in the converted value. A COBOL picture of **S9(6)V9(3)** equates to a precision of 9 and a scale of 3.
- The zero-relative offset into the output buffer where the value will be stored. The next slide shows how this offset can be computed efficiently.
- The maximum length of the converted data. As mentioned previously for parameter binding, this value is used only for character data, and must be at least one greater than the maximum length you expect to store. Character strings longer than this value will be truncated.
- The zero-relative offset to the descriptor word in the buffer area.
- A constant value indicating the action to be performed on this column. Normally you want the column to be bound to the statement, so you would specify the constant **SQL-BIND-COL**. You can also use this function to unbind a single column by specifying **SQL-UNBIND-COL**. You can cause all output columns to be unbound from the statement by calling **SQLFREESTMT** and passing **SQL-UNBIND** as an option.

Note that all arguments to this function are binary integers.

SQLBindCol Example

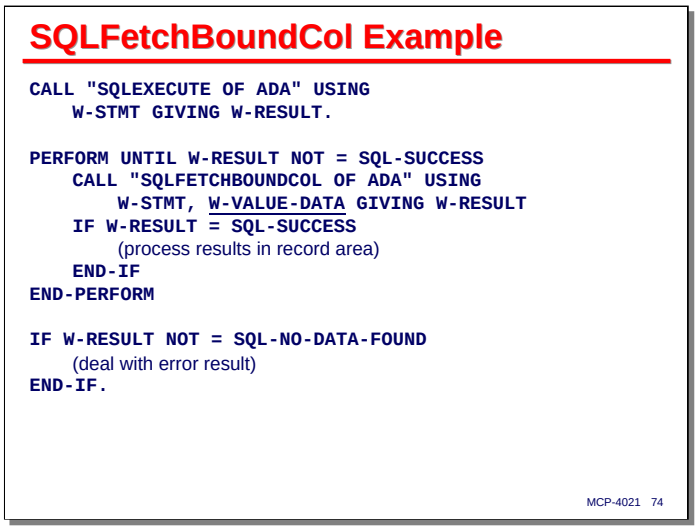
```
77 W-COL-INDEX                PIC S9(11)       VALUE 3 BINARY.
77 W-VALUE-OFFSET             PIC S9(11)               BINARY.
77 W-VALUE-MAX-LEN            PIC S9(11)               BINARY.
77 W-W-DESC-OFFSET            PIC S9(11)               BINARY.
01 W-VALUE-DATA.
    05 W-COMP-DESC            PIC S9(11)               BINARY.
    05 W-COMP-VALUE           PIC S9(6)V9(4)             COMP.
    05 FILLER                 PIC X(7).
    05 W-CHAR-DESC            PIC S9(11)               BINARY.
    05 W-CHAR-VALUE           PIC X(30).

COMPUTE W-VALUE-OFFSET = OFFSET (W-CHAR-VALUE)
COMPUTE W-VALUE-MAX-LEN = FUNCTION LENGTH-AN (W-CHAR-VALUE)
COMPUTE W-DESC-OFFSET = OFFSET (W-CHAR-DESC)

CALL "SQLBINDCOL OF ADA" USING
    W-STMT, W-COL-INDEX,
    SQL-COBOL-CHAR, SQL-NULL-DATA, SQL-NULL-DATA,
    W-VALUE-OFFSET, W-VALUE-MAX-LEN, W-DESC-OFFSET
    SQL-BIND-COL GIVING W-RESULT.
```

MCP-4021 73

This slide shows the data definitions and argument sequence involved in binding an output column. As with **SQLGETDATA** and **SQLBINDPARAMETER**, you can use the COBOL-85 **OFFSET** and **FUNCTION LENGTH-AN** intrinsics to determine the zero-relative byte offset of a field and its length in bytes. In COBOL-74, you must use **FUNCTION FORMATTED-SIZE** instead of **LENGTH-AN**.

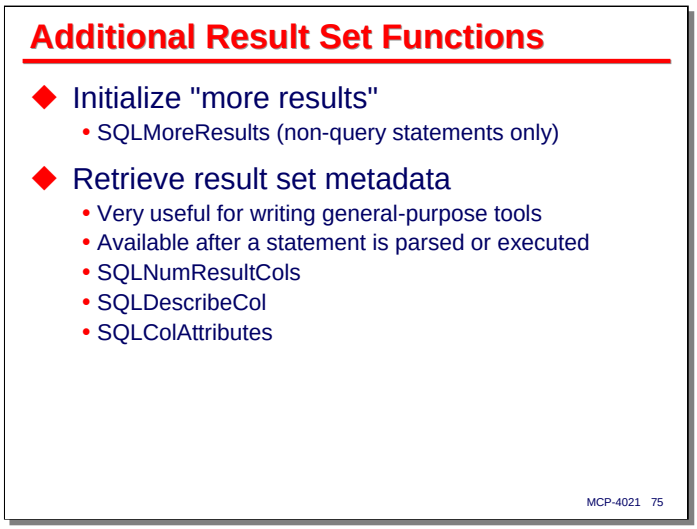


This slide shows a simple example that retrieves the results of a SQL query using a COBOL-85 in-line perform. This example assumes that no parameter binding is being done, and that all output column binding has already been established.

After executing the SQL statement, your program enters a loop that will terminate when the result is no longer **SQL-SUCCESS**. It calls **SQLFETCHBOUNDCOL** to advance to the next row of the result set. If that call is successful, the bound columns will have been moved to their offsets within the output buffer, **W-VALUE-DATA**, where they can be processed by your program.

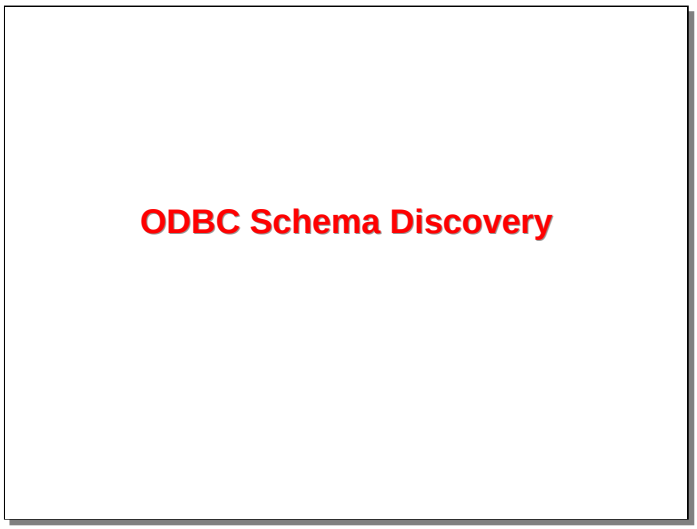
After control falls out of the loop, you can check for a result of **SQL-NO-DATA-FOUND**. If this was the last result returned, you know that you have processed all of the rows from the result set, and you are done. Any other value indicates a warning or error, which your program must then deal with.

Actually, this example is a little too simple to be practical in most cases, as it treats a warning result as an error. The most common type of non-success result you should expect from **SQLFETCHBOUNDCOL** is a warning (**SQL-SUCCESS-WITH-INFO**, value 1) indicating that one of the output column values was truncated. Depending on your situation, this may or may not be an error, but in most cases your program can probably ignore such warnings.

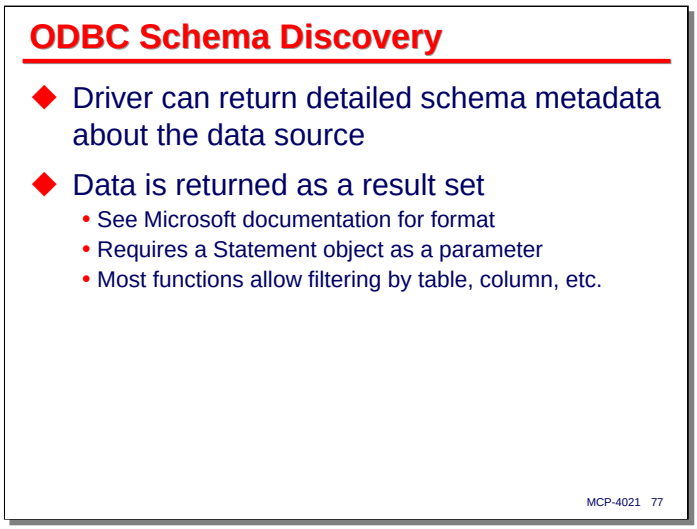


There are a few additional functions that are useful for processing result sets:

- Some data sources permit submitting multiple SQL statements in one execution. Each of these will be executed by the data source in sequence and the results stacked up for retrieval by your application. Your application will retrieve the row count or result set for the first statement. To obtain the results for the second statement, it must call **`SQLMORERESULTS`**. After processing those results, it calls the function again to step through the results for the remaining statements, one at a time.
- After preparing or directly executing a statement, the **`SQLNUMRESULTCOLS`**, **`SQLDESCRIBECOL`**, and **`SQLCOLATTRIBUTES`** functions can be used to determine the number of output columns and the characteristics of those columns. These functions are similar to the **`SQLNUMPARAMS`** and **`SQLDESCRIBEPARAM`** functions, and are very useful when writing a general-purpose query processor that may obtain its SQL text from an external source.

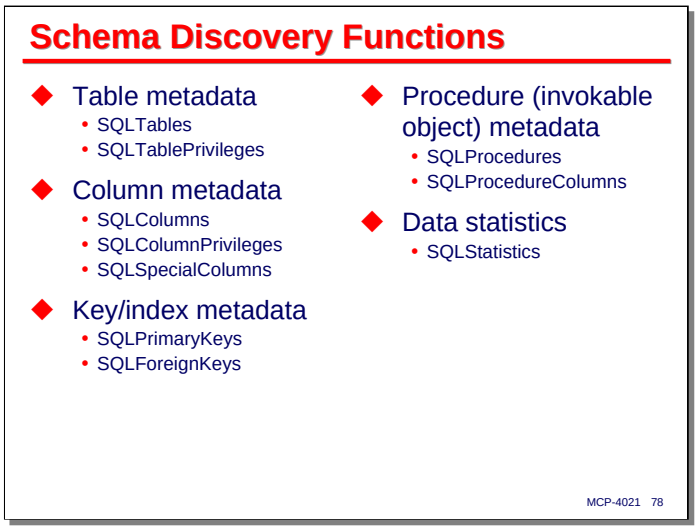


The final section of this presentation will briefly discuss ODBC schema discovery.



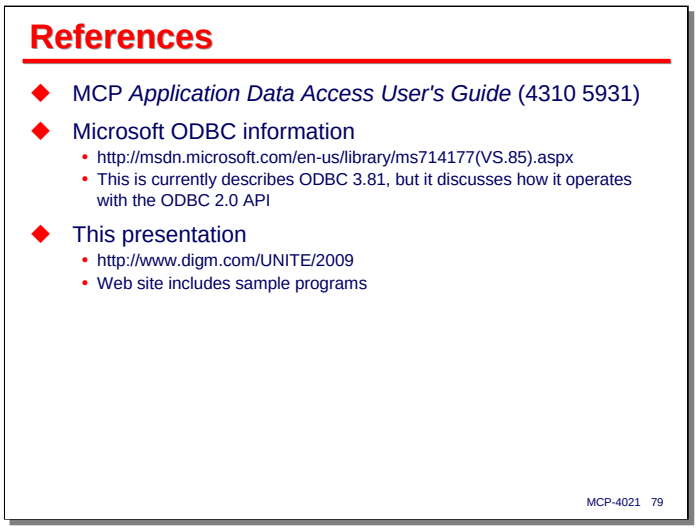
One of the features that most ODBC drivers support is exposing the schema of the underlying data source. The "schema" consists of the tables, columns, indexes, and other objects that comprise the definition of the data source's data. You can interrogate this schema information through A-D-A.

The details involved in schema discovery are beyond the scope of this presentation. You should access the Microsoft ODBC documentation for information on the arguments and results for the functions listed on the next slide. Each schema discovery function takes a statement handle as an argument, along with values that specify how the results are to be filtered. Each function returns a result set with the schema metadata. Most of these functions allow you to filter the metadata by table, column, etc.



There are separate functions for retrieving metadata for the following types of objects:

- Tables
- Columns of tables
- Keys and indexes to tables
- Stored procedures and other invokable objects
- Statistics on table populations and other metrics for the data.



The MCP implementation of Application Data Access is documented in the *Application Data Access User's Guide*, which is part of the standard documentation library.

As mentioned earlier in the presentation, the A-D-A documentation discusses the calling sequence for each of the functions and how the A-D-A implementation differs from standard ODBC, but almost nothing about how the functions work. For that you need to consult the Microsoft ODBC documentation, which is available on their MSDN web site. The link to this information seems to move around, so you may need to search the MSDN site for "ODBC API".

Finally, this presentation, along with some sample programs, is available on our web site at the URL shown on the slide.

End

Using Application Data Access

2009 UNITE Conference

Session MCP-4021