**Unit-4**

**Chapter-31:**

**Database Management with Microsoft ODBC**

**Q: What advantages does the DBMS did in VC++?**

**Ans:**

You could program your own random access disk file, thus inventing your own DBMS, but you probably have enough work to do already. Besides, using a real DBMS gives you many advantages, including the following:

* **Use of standard file formats**—Many people think of dBASE/Xbase DBF files when they think of database formats. This is only one database file format, but it's a popular one. A lot of data is distributed in DBF files, and many programs can read and write in this format. Lately, the Microsoft Access MDB format has become popular, too. With the MDB format, all of a database's tables and indexes can be contained in a single disk file.
* **Indexed file access**—If you need quick access to records by key (a customer name, for example), you need indexed file access. You could always write your own B-tree file access routines, but that's a tedious job that's been done already. All DBMS's contain efficient indexed access routines.
* **Data integrity safeguards**—Many professional DBMS products have procedures for protecting their data. One example is transaction processing. A transaction encompasses a series of related changes. If the entire transaction can't be processed, it is rolled back so that the database reverts to its original state before the transaction.
* **Multiuser access control**—If your application doesn't need multiuser access now, it might in the future. Most DBMS's provide record locking to prevent interference among simultaneous users. Some multiuser DBMS's use the client-server model, which means that most processing is handled on a single database server computer; the workstations handle the user interface. Other multiuser DBMSs handle database processing on the workstations, and they control each workstation's access to shared files.

**The ODBC Standard**

1. The Microsoft Open Database Connectivity (ODBC) standard defines not only the rules of SQL grammar but also the C-language programming interface to any SQL database. The ODBC Software Development Kit (SDK), included with Visual C++, contains 32-bit drivers for DBF files, Microsoft Access MDB databases, Microsoft Excel XLS files, Microsoft FoxPro files, ASCII text files, and Microsoft SQL Server databases.
2. Not only can C++ programs use ODBC but other DBMS programming environments can also take advantage of this new standard.
3. ODBC thus separate user interface from actual database management.

**Q: With the diagram explain 32-bit ODBC Architecture?**

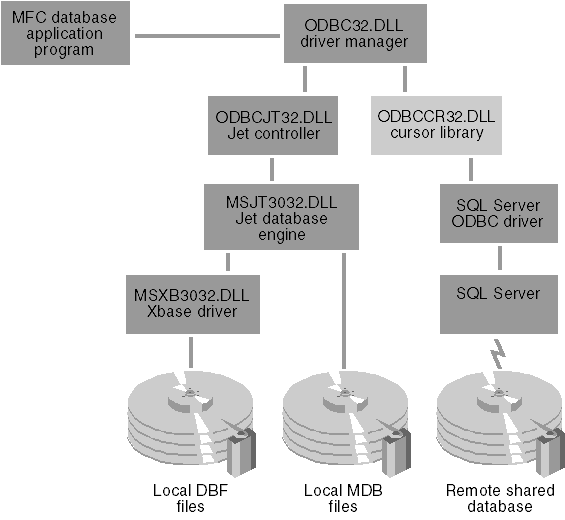
**Ans:**

**ODBC Architecture:**

1. ODBC's unique DLL-based architecture makes the system fully modular.
2. A small top-level DLL, ODBC32.DLL, defines the API.
3. ODBC32.DLL loads database-specific DLLs, known as drivers, during program execution.
4. From the figure that many standard database formats can be accessed through the Microsoft Access Jet database engine, a redistributable module packaged with Visual C++.

## ODBC SDK Programming:

1. First you need an environment that establishes the link between your program and the ODBC system.
2. Next you need one or more connections. The connection references a specific driver and data source combination.
3. Once you have connection you need a SQL statement to execute. ODBC lets you define the query result as a block of data, called a rowset, which is associated with an SQL statement.
4. Visual C++ includes the ODBC cursor library module ODBCCR32.DLL, which supports static rowsets (called snapshots) for Level 1 drivers.
5. An ODBC Level 2 driver, the rowset would probably be dynamic and ODBC could update the rowset whenever the database changed. A dynamic rowset is called a dynaset.



ODBC32.DLL has a built-in Windows dialog box that lists the connections that are defined in the Registry (under HKEY\_LOCAL\_MACHINE-\SOFTWARE\ODBC). Once you have a connection, you need a SQL statement to execute. The statement might be a query, such as this:

SELECT FNAME, LNAME, CITY FROM AUTHORS  
WHERE STATE = 'UT' ORDER BY LNAME

Or the statement could be an update statement, such as this:

UPDATE AUTHORS SET PHONE = '801 232-5780'  
WHERE ID = '357-86-4343'

Because query statements need a program loop to process the returned rows, your program might need several statements active at the same time. Many ODBC drivers allow multiple active statement handles per connection.

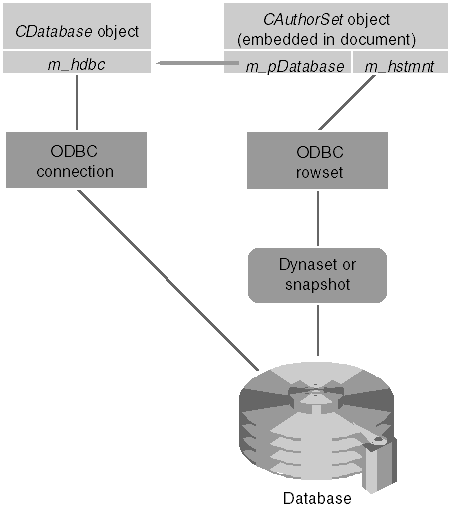
ODBC lets you define the query result as a block of data, called a rowset, which is associated with an SQL statement. Through the ODBC SDK function *SQLExtendedFetch*, your program can move forward and backward through the 10 selected records by means of an ODBC cursor. This cursor is a programmable pointer into the rowset.

Visual C++ includes the ODBC cursor library module ODBCCR32.DLL, which supports static rowsets (called snapshots) for Level 1 drivers. With a snapshot, a *SELECT* statement causes ODBC to make what amounts to a local copy of the 10 author records and build an in-memory list of pointers to those records.

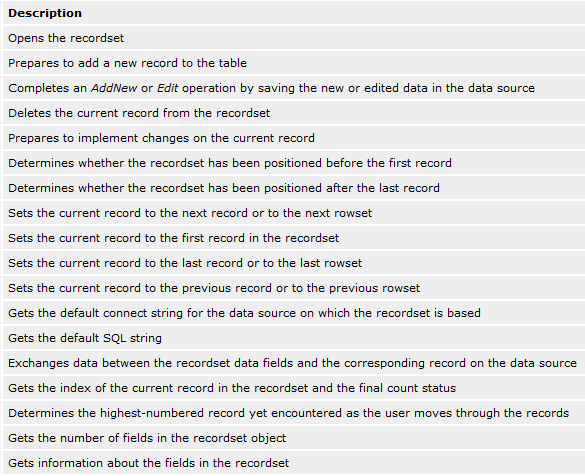
**Q: With the help of diagram explain MFC ODBC class database relationship?**

**Ans:**

* 1. With the MFC ODBC classes, you use objects instead of connection handles and statement handles.
  2. The environment handle is stored in a global variable and is not represented by a C++ object.
  3. The two principal ODBC classes are *CDatabase* and *CRecordset*.
  4. Objects of class *CDatabase* represent ODBC connecions to data sources, and objects of class *CRecordset* represent \scrollable rowsets.
  5. The Visual C++ documentation uses the term "recordset" instead of "rowset" to be consistent with Microsoft Visual Basic and Microsoft Access.



* 1. The important *CRecordset* member functions discussed in this chapter are summarized in the table below.



## Counting the Rows in a Recordset

ODBC doesn't provide an accurate count of the rows in a recordset until you've read past the end. Until that time, the count returned from the *CRecordset::GetRecordCount* member function is a "high-water mark" that returns only the last row accessed by *CRecordset::MoveNext*. The *CRecordset::GetStatus* function returns a *CRecordsetStatus* object, which has a member *m\_bRecordCountFinal* that indicates whether the count is final.

The *CRecordset::MoveLast* function does not register the record count for you, even for dynasets. If you want to know how many records are included in a recordset, loop through the whole table with *MoveNext* calls.

## Processing ODBC Exceptions

Many MFC ODBC calls don't return an error code but instead throw a *CDBException* object, which contains a string describing the error. Suppose you are trying to delete a record from a table in an Access database. Access might be enforcing referential integrity rules, which means that you're not allowed to delete that row because a row in another table depends on it. If you call *CRecordset::Delete*, you'll see an ODBC error message box that came from the MFC base classes.

**Chapter-32:**

**Database Management with Microsoft Data Access Objects**

**Q: What are DAO and MFC classes? What are the database work with DAO?**

**Ans:**

# DAO and MFC

The MFC library has the following five DAO database classes.

| **Class** | **Use** |
| --- | --- |
| *CDaoWorkspace* | An interface for managing a single user's database session |
| *CDaoDatabase* | An interface for working with a database |
| *CDaoRecordset* | An interface for working with a set of records (such as table-type recordsets, dynaset-type recordsets, or snapshot-type recordsets) |
| *CDaoTableDef* | An interface for manipulating a definition of a base table or an attached table |
| *CDaoQueryDef* | An interface for querying a database |

These classes more or less wrap the COM interfaces with corresponding names. (*CDaoRecordset* wraps *DAORecordset*, for example.) The *CDaoWorkspace* class actually wraps two interfaces, *DAOWorkspace* and *DAODBEngine*. The MFC wrapping is fairly complete, so you need to make direct COM DAO calls only when you need access to certain database security features. If you use the MFC library, all reference counting is taken care of; if you call DAO directly, you must be sure to call *Release* on your interfaces.

Both AppWizard and ClassWizard fully support DAO. You can use ClassWizard to generate a table-specific class that is derived from *CDaoRecordset*.

**What Databases Can You Open with DAO?**

The following four database options are supported by DAO:

* **Opening an Access database (MDB file)**—An MDB file is a self-contained database that includes query definitions, security information, indexes, relationships, and of course the actual data tables. You simply specify the MDB file's pathname.
* **Opening an ODBC data source directly**—There's a significant limitation here. You can't open an ODBC data source that uses the Jet engine as a driver; you can use only data sources that have their own ODBC driver DLLs.
* **Opening an ISAM-type (indexed sequential access method) data source (a group of dBASE, FoxPro, Paradox, Btrieve, Excel, or text files) through the Jet engine**—Even if you've set up an ODBC data source that uses the Jet engine to access one of these file types, you must open the file as an ISAM-type data source, not as an ODBC data source.
* **Attaching external tables to an Access database**—This is actually the preferred way of using DAO to access ODBC data. First you use Access to attach the ODBC tables to an MDB file, and then you use DAO to open the MDB file as in the first option. You can also use Access to attach ISAM files to an MDB file.

**Chapter**-**33:**

**The OLE DB Templates**

**Q: Why use OLE DB? Explain basic OLE DB Architecture?**

**Ans:**

OLE DB exists to provide a uniform way to access all sorts of disparate data sources. For example, imagine all the types of data sources you might find in a typical organization. These might include sources as varied as production systems, file systems, spreadsheets, personal databases (such as Xbase and Btrieve), and e-mail. The problem is that each of these sources requires its own protocol: if you want to access data from a specific source, you need to learn the protocol for managing the data source. (ugh!) OLE DB is the middle layer that makes accessing data from different sources uniform. With OLE DB, client-side developers need to concentrate on only a few details to get access to data (instead of needing to know tons of different database access protocols).

The most important thing to realize about OLE DB is that it is built upon COM. In other words, OLE DB is a set of ActiveX interfaces for accessing data through COM. The OLE DB interfaces are general enough to provide a uniform means of accessing data, regardless of the method that is used to store the data. For example, developers use the same OLE DB interfaces to get to data without being concerned as to whether data is stored in a DBMS or a non-DBMS information source. At the same time, OLE DB lets developers continue to take advantage of the benefits of the underlying database technology (like speed and flexibility) without having to move data around just to access those benefits.

At the highest level, the OLE DB architecture consists of consumers and providers. A consumer is any bit of system or application code that uses an OLE DB interface. This includes OLE DB components themselves. A provider is any software component that exposes an OLE DB interface.

There are two types of OLE DB providers: data providers and service providers. The names are pretty self-explanatory. Data providers own data and expose that data in a tabular form as a rowset. Some good examples of data providers include relational Database Management Systems (DBMS's), storage managers, spreadsheets, and Indexed Sequential Access Method (ISAM) databases.

A service provider is any OLE DB component that does not own data but encapsulates some service by massaging data through OLE DB interfaces. In one sense, a service component is both a consumer and a provider.

OLE DB defines an architecture that "component-izes" data access. As a component database-management system, OLE DB offers greater efficiency than traditional database-management systems by separating database functionality into the roles of consumers and producers. Because data consumers generally require only a portion of the database-management functionality, OLE DB separates that functionality, thereby reducing client-side resource overhead.

By the same token, OLE DB reduces the burden on the provider side, since providers need to worry only about providing data (and don't have to concern themselves with any client-side junk). For example, OLE DB allows a simple tabular data provider to implement functionality native to its data store yet provide a singular access protocol to get to the data.

In addition, SQL DBMS's can expose their functionality in a more layered manner by using the OLE DB interfaces.

**Basic OLE DB Architecture**

In addition to defining a basic relationship between consumers and providers, OLE DB defines the following components that make up the OLE DB architecture (each component is a COM object):

* **Enumerators** Enumerators search for available data sources. Consumers that are not hardwired for a particular data source employ enumerators to search for a data source to use.
* **Data source objects** Data source objects contain the machinery to connect to a data source, such as a file or a DBMS. A data source object generates sessions.
* **Sessions** Sessions represent connections to a database. For example, sessions provide a context for database transactions. A single data source object can create multiple sessions. Sessions generate transactions, commands, and rowsets.
* **Transaction objects** Transaction objects are used for managing database transactions in order to maintain database security.
* **Commands** Commands execute text commands, such as a SQL statement. If the text command specifies a rowset, such as a SQL SELECT statement, the command generates rowsets. A single session can create multiple commands.
* **Rowsets** Rowsets expose data in a tabular format. A special case of a rowset is an index. Rowsets can be created from the session or the command.
* **Errors** Errors can be created by any interface on any OLE DB object. They contain additional information about an error, including an optional custom error object.

.

The upside of using the OLE DB architecture is that you get a terrific, homogenous way to access heterogeneous data sources. The downside is that you have to implement a bunch of COM interfaces to make that happen. That's why the OLE DB templates exist.

**Q: Explain OLE DB template architecture?**

**Ans:**

# Basic OLE DB Template Architecture

OLE DB involves implementing a bunch of interfaces. Of course, just as with ActiveX Controls, you can choose to implement them by, or you can find someone else to do most of the dirty work. While OLE DB is a rich and powerful data access technology, getting it up and running by hand is a somewhat tedious task.

Just as Visual C++ provides a template library (ATL) for implementing ActiveX Controls, Visual C++ also provides a template library that helps you manage OLE DB. The OLE DB template support provides classes that implement many of the commonly used OLE DB interfaces. In addition, Visual C++ provides great wizard support for generating code to apply to common scenarios.

From a high level, you can divide the classes in this template library into the two groups defined by OLE DB itself: the consumer classes and the provider classes. The consumer classes help you implement database client (consumer) applications, while the provider classes help you implement database server (provider) applications.

1. **OLE DB Consumer Template Architecture**

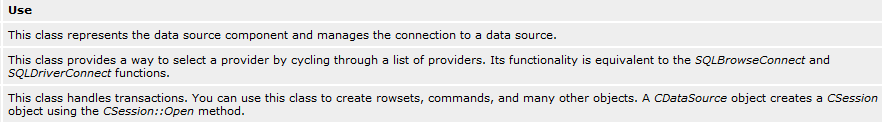
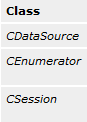
Microsoft has kept the top layer classes in the OLE DB Consumer Templates as close to the OLE DB specification as possible. That is, OLE DB templates don't define another object model. Their purpose is simply to wrap the existing OLE DB object model.

The OLE DB Templates are small and flexible. They are implemented using C++ templates and multiple inheritance. Because OLE DB templates are close to the metal (they wrap only the existing OLE DB architecture), each class mirrors an existing OLE DB component.

The OLE DB Consumer Template architecture can be divided into three parts: the general data source support classes, classes for supporting data access and rowset operations, and classes for handling tables and commands.

### General Data Source Support

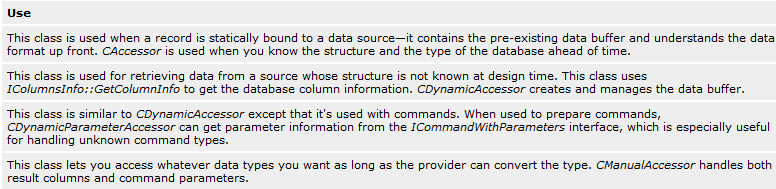
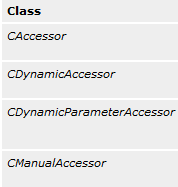
A data source is the most fundamental concept to remember when talking about data access using OLE DB. That is, where is the data coming from? Of course, the OLE DB templates have support for data sources. General data source support comprises three classes as shown in this table.



### Data Access and Rowset Support

The OLE DB templates provide binding and rowset support through several classes. The accessor classes talk to the data source while the rowset manages the data in tabular form. The data access and rowset components are implemented through the *CAccessorRowset* class. *CAccessorRowset* is a template class that's specialized on an accessor and a rowset. This class can handle multiple accessors of different types.

The OLE DB Template library defines the accessors in this table.



Along with the accessors, the OLE DB templates define three types of rowsets: single fetching, bulk, and array. These are fairly self-explanatory descriptions.

### Table and Command Support

The final layer in the OLE DB Template consumer architecture consists of two more classes: table and command classes (*CTable* and *CCommand*). These classes are used to open the rowset, execute commands, and initiate bindings. Both classes derive from *CAccessorRowset*

The *CTable* class is a minimal class implementation that opens a table on a data source (which you can specify programmatically). Use this class when you need bare-bones access to a source, since *CTable* is designed for simple providers that do not support commands.

Other data sources also support commands. For those sources, you'll want to use the OLE DB Templates' *CCommand* class. As its name implies, *CCommand* is used mostly for executing commands. This class has a function named *Open* that executes singular commands. This class also has a function named *Prepare* for setting up a command to execute multiple times.

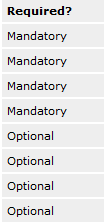
When using the *CCommand* class, you'll specialize it with three template arguments: an accessor, a rowset, and a third template argument (which defaults to *CNoMultipleResults*). If you specify *CMultipleResults* for this third argu- ment, the *CCommand* class will support the *IMultipleResults* interface for a command that returns multiple rowsets.

1. **OLE DB Provider Template Architecture**

Remember that OLE DB is really just a set of interfaces that specify a protocol for managing data. OLE DB defines several interfaces (some mandatory and others optional) for the following types of objects: data source, session, rowset, and command. Here's a description of each followed by a code snippet that shows how the templates bring in the correct functionality for each component.

1. **Data source object**

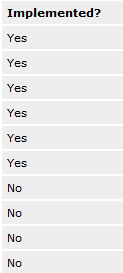
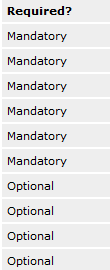
A data source object wraps most aspects of data access. For example, a data source consists of actual data and its associated database management system (DBMS), the platform on which the DBMS exists, and the network used to access that platform. A data source is just a COM object that implements a bunch of interfaces, as shown in Table 33-1.



1. **Command object**

Providers that support building and executing queries expose a command object. Command objects specify, prepare, and execute a Database Manipulation Language (DML) query or Data Definition Language (DDL) definition and its associated properties.

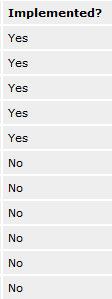
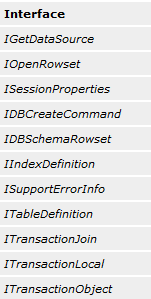
The command class is the heart of the data provider. Most of the action happens within this class.



1. **Session object**

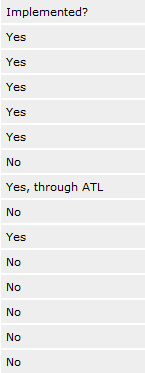
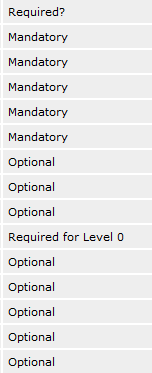
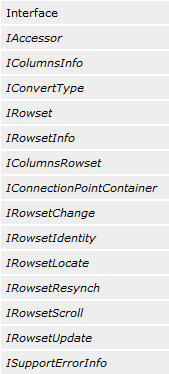
Session objects define the scope of a transaction and generate rowsets from the data source. Session objects also generate command objects. The command object executes commands on the rowset.

Table 33-3 shows the interfaces found on a session object.



1. **Rowset object**

A rowset object represents tabular data. At the raw OLE DB level, rowsets are generated by calling *IOpenRowset::OpenRowset* on the session. For providers that support commands, rowsets are used to represent the results of row-returning queries. In addition to *IOpenRowset::OpenRowset*, there are a number of other methods in OLE DB that return rowsets.



The wizard-generated rowset object implements the *IAccessor, IRowset*, and *IRowsetInfo* interfaces, among others. *IAccessorImpl* binds both output columns. The *IRowset* interface fetches rows and data. The *IRowsetInfo* interface handles the rowset properties.

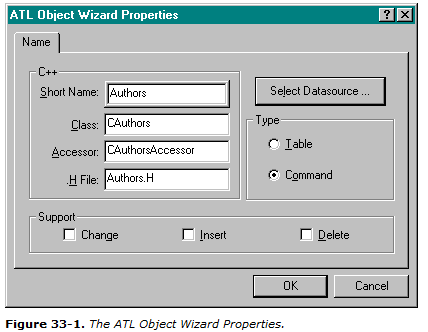
**Q: Write steps to create OLE DB consumer?**

**Ans:**

# Creating an OLE DB Consumer

Creating an OLE DB consumer is pretty straightforward—most of the support comes through the ATL Object Wizard. You can see an example of a consumer in the x33a folder on the companion CD. Here are the steps for creating a consumer using the ATL Object Wizard.

1. Create an application or a control to drive the data consumption. For example, you might want to create an ActiveX control.
2. While inside the IDE, use the ATL Object Wizard to insert a data consumer. Do this by either selecting New ATL Object from the Insert menu or by right-clicking on the project icon in ClassView and selecting New ATL Object from the context menu to start the ATL Object Wizard.
3. From the ATL Object Wizard, select the Data Access category of objects. Then select Consumer and click Next. This will cause the ATL Object Wizard Properties dialog, shown in Figure 33-1, to appear. There will be only one page in it, for naming the class and selecting the data source.



1. Click Select Datasource to configure the data consumer. Once you've picked out a data source, choose OK. The ATL Object Wizard will create an OLE DB Consumer template ready for you to use.

**Q: Why write an OLE DB provider? Or**

**Write steps to create OLE DB provider?**

**Ans:**

# Creating an OLE DB Provider

You just ask a wizard to create a wrapper for you, and you get a fairly easy way to access the data in a database. However, it might be a bit less obvious why you'd want to create an OLE DB provider.

## Why Write an OLE DB Provider?

Writing an OLE DB allows you to insert a layer between a client of some data and the actual data itself. Here are just a few reasons you might want to write a provider.

* Writing an OLE DB provider means clients don't necessarily touch the data directly. Therefore, you can add additional capabilities to your data, such as query processing.
* In some cases, writing an OLE DB provider gives you the opportunity to increase data access performance by controlling how the data is manipulated.
* Adding an OLE DB provider layer increases the potential audience of your data. For example, if you have a proprietary data format that can be accessed by only one programming language, you have a single point of failure. OLE DB providers give you a way to open that proprietary format to a wider variety of programmers, regardless of the programming language they use.

## Writing an OLE DB Provider

Working with the OLE DB Providers is similar to working with the Consumers. The wizards do a lot of the work for you. You just need to know how to work with the generated classes. The steps for creating an OLE DB Provider are listed here.

1. The first step is to decide what you want the provider to do. Remember the philosophy behind OLE DB: it's all about providing a singular way to access multiple data sources. For example, you might want to write a provider that recursively enumerates the contents of a structured storage file. Or you might want a provider that sifts through e-mail folders and allows clients database-style access to your e-mail system. The possibilities are nearly endless.
2. Just as you did when writing a data consumer, use the ATL Object Wizard to create a provider. Just start the ATL Object Wizard from ClassView or from the Insert menu. Select the Data Access objects category, and choose Provider. The ATL Object Wizard will ask you to provide a name for your object and will allow you to modify the default names for the files it will create.
3. After you click OK, the ATL Object Wizard creates the code for a provider, including a data source, a rowset, and a session. In addition to these objects, a provider supports one or more properties, which are defined in property maps within the files created by the OLE DB Provider Template Wizard. When the Wizard creates the files, it inserts maps for the properties belonging to the OLE DB property group defined for the object or objects included in those files. For example, the header file containing the data source object also contains the property map for the DataSource properties. The session header file contains the property map for the Session properties. Finally, the rowset and command objects reside in a single header file, which includes properties for the command object.

**Chapter-34:**

**TCP/IP, Winsock, and WinInet**

**Q: Explain IP and IP/UDP?**

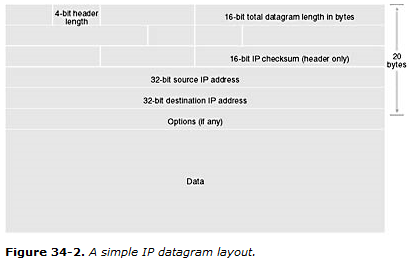
**Ans:**

## The Internet Protocol

The Internet Protocol (IP) layer is the best place to start in your quest to understand TCP/IP. The IP protocol defines packets called datagrams that are fundamental units of Internet communication. These packets, typically less than 1000 bytes in length, go bouncing all over the world when you open a Web page, download a file, or send e-mail. Figure 34-2 shows a simplified layout of an IP datagram.

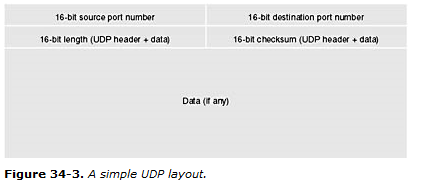
Notice that the IP datagram contains 32-bit addresses for both the source and destination computers. These IPaddresses uniquely identify computers on the Internet and are used by routers (specialized computers that act like telephone switches) to direct the individual datagrams to their destinations. The routers don't care about what's inside the datagrams—they're only interested in that datagram's destination address and total length. Their job is to resend the datagram as quickly as possible.

The IP layer doesn't tell the sending program whether a datagram has successfully reached its destination. That's a job for the next layer up the stack. The receiving program can look only at the checksum to determine whether the IP datagram header was corrupted.

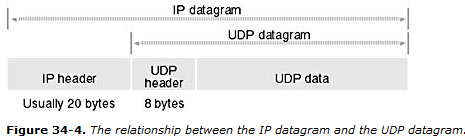


## The User Datagram Protocol

The TCP/IP protocol should really be called TCP/UDP/IP because it includes the User Datagram Protocol (UDP), which is a peer of TCP. All IP-based transport protocols store their own headers and data inside the IP data block. First let's look at the UDP layout in Figure 34-3.



A complete UDP/IP datagram is shown in Figure 34-4.



UDP is only a small step up from IP, but applications never use IP directly. Like IP, UDP doesn't tell the sender when the datagram has arrived. That's up to the application. The sender could, for example, require that the receiver send a response, and the sender could retransmit the datagram if the response didn't arrive within, say, 20 seconds. UDP is good for simple one-shot messages and is used by the Internet Domain Name System (DNS), which is explained later in this chapter. (UDP is used for transmitting live audio and video, for which some lost or out-of-sequence data is not a big problem.)

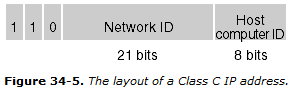
Figure 34-3 shows that the UDP header does convey some additional information—namely the source and destination portnumbers. The application programs on each end use these 16-bit numbers. For example, a client program might send a datagram addressed to port 1700 on the server. The server program is listening for any datagram that includes 1700 in its destination port number, and when the server finds one, it can respond by sending another datagram back to the client, which is listening for a datagram that includes 1701 in its destination port number.

**Q: Explain IP Address? Explain relationship between IP and TCP?**

**Ans:**

## IP Address Format—Network Byte Order

You know that IP addresses are 32-bits long. You might think that 232 (more than 4 billion) uniquely addressed computers could exist on the Internet, but that's not true. Part of the address identifies the LAN on which the hostcomputer is located, and part of it identifies the host computer within the network. Most IP addresses are ClassCaddresses, which are formatted as shown in Figure 34-5.

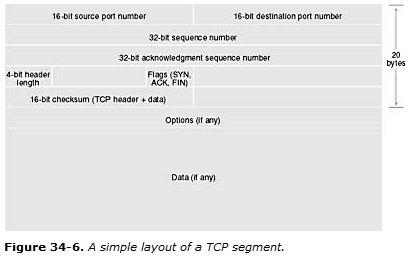


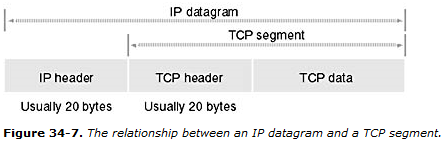
This means that slightly more than 2 million networks can exist, and each of those networks can have 28 (256) addressable host computers. The Class A and Class B IP addresses, which allow more host computers on a network, are all used up.

By convention, IP addresses are written in dotted-decimal format. The four parts of the address refer to the individual byte values.

## The Transmission Control Protocol

What you really need is a protocol that supports error-free transmission of large blocks of data. Obviously, you want the receiving program to be able to reassemble the bytes in the exact sequence in which they are transmitted, even though the individual datagrams might arrive in the wrong sequence. TCP is that protocol, and it's the principal transport protocol for all Internet applications, including HTTP and File Transfer Protocol (FTP). Figure 34-6 shows the layout of a TCP segment. (It's not called a datagram.) The TCP segment fits inside an IP datagram, as shown in Figure 34-7.





The TCP protocol establishes a full-duplex, point-to-point connection between two computers, and a program at each end of this connection uses its own port. The combination of an IP address and a port number is called a socket. The connection is first established with a three-wayhandshake. The initiating program sends a segment with the *SYN* flag set, the responding program sends a segment with both the *SYN* and *ACK* flags set, and then the initiating program sends a segment with the *ACK* flag set.

After the connection is established, each program can send a stream of bytes to the other program. TCP uses the sequence number fields together with *ACK* flags to control this flow of bytes. The sending program doesn't wait for each segment to be acknowledged but instead sends a number of segments together and then waits for the first acknowledgment. If the receiving program has data to send back to the sending program, it can piggyback its acknowledgment and outbound data together in the same segments.

The sending program's sequence numbers are not segment indexes but rather indexes into the byte stream. The receiving program sends back the sequence numbersto the sending program, thereby ensuring that all bytes are received and assembled in sequence. The sending program resends unacknowledged segments.

Each program closes its end of the TCP connection by sending a segment with the *FIN* flag set, which must be acknowledged by the program on the other end. A program can no longer receive bytes on a connection that has been closed by the program on the other end.

**Q: Write short note on DNS?**

**Ans:**

## The Domain Name System

When you surf the Web, you don't use IP addresses. Instead, you use human-friendly names like *microsoft.com* or *www.cnn.com*. A significant portion of Internet resources is consumed when hostnames (such as *microsoft.com*) are translated into IP addresses that TCP/IP can use. A distributed network of nameserver (domain server) computers performs this translation by processing DNSqueries. The entire Internet namespace is organized into domains, starting with an unnamed rootdomain. Under the root is a series of top-leveldomains such as *com*, *edu*, *gov*, *and org*.

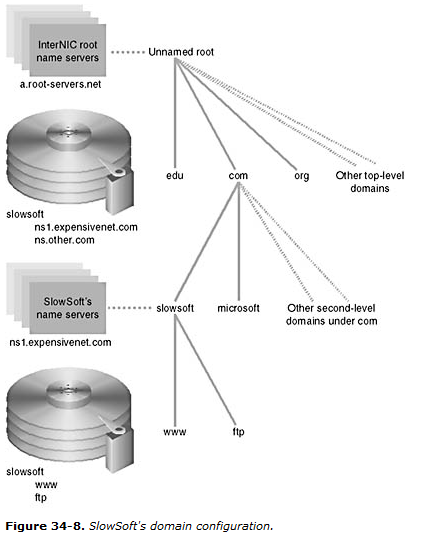
### Servers and Domain Names

Suppose a company named SlowSoft has two host computers connected to the Internet, one for World Wide Web (WWW) service and the other for FTP service. By convention, these host computers are named *www.slowsoft.com* and *ftp.slowsoft.com*, respectively, and both are members of the second-leveldomain slowsoft, which SlowSoft has registered with an organization called InterNIC.

Now SlowSoft must designate two (or more) host computers as its name servers. The name servers for the *com* domain each have a database entry for the *slowsoft* domain, and that entry contains the names and IP addresses of SlowSoft's two name servers. Each of the two *slowsoft* name servers has database entries for both of SlowSoft's host computers. These servers might also have database entries for hosts in other domains, and they might have entries for name servers in third-level domains. Thus, if a name server can't provide a host's IP address directly, it can redirect the query to a lower-level name server. Figure 34-8 illustrates SlowSoft's domain configuration.

### Clients and Domain Names

Now for the client side. A user types *http:*//*www.slowsoft.com* in the browser. (The *http://* prefix tells the browser to use the HTTP protocol when it eventually finds the host computer.) The browser must then resolve *www.slowsoft.com* into an IP address, so it uses TCP/IP to send a DNS query to the defaultgateway IP address for which TCP/IP is configured. This default gateway address identifies a local name server, which might have the needed host IP address in its cache. If not, the local name server relays the DNS query up to one of the root name servers. The root server looks up *slowsoft* in its database and sends the query back down to one of SlowSoft's designated name servers. In the process, the IP address for *www.slowsoft.com* will be cached for later use if it was not cached already. If you want to go the other way, name servers are also capable of converting an IP address to a name.



**Q: What is Winsock? Explain Synchronus and Asynchronous winsock?**

**Ans:**

# Winsock

Winsock is the lowest level Windows API for TCP/IP programming. Part of the code is located in wsock32.dll (the exported functions that your program calls), and part is inside the Windows kernel. You can write both internet server programs and internet client programs using the Winsock API. This API is based on the original Berkely Sockets API for UNIX. A new and much more complex version, Winsock 2, is included for the first time with Windows NT 4.0, but we'll stick with the old version because it's the current standard for both Windows NT, Windows 95, and Windows 98.

## Synchronous vs. Asynchronous Winsock Programming

Winsock was introduced first for Win16, which did not support multithreading. Consequently, most developers used Winsock in the asynchronous mode. In that mode, all sorts of hidden windows and *PeekMessage* calls enabled single-threaded programs to make Winsock send and receive calls without blocking, thus keeping the user interface (UI) alive. Asynchronous Winsock programs were complex, often implementing "state machines" that processed callback functions, trying to figure out what to do next based on what had just happened. Well, we're not in 16-bit land anymore, so we can do modern multithreaded programming.

We will make the most of our Winsock calls from worker threads so that the program's main thread is able to carry on with the UI. The worker threads contain nice, sequential logic consisting of blocking Winsock calls.

## The MFC Winsock Classes

We try to use MFC classes where it makes sense to use them, but the MFC developers informed us that the *CAsyncSocket* and *CSocket* classes were not appropriate for 32-bit synchronous programming. The Visual C++ online help says you can use *CSocket* for synchronous programming, but if you look at the source code you'll see some ugly message-based code left over from Win16.

**Q: Write a code for starting server and the server thread?**

**Ans:**

### Starting the Server

The server starts in response to some user action, such as a menu choice. Here's the command handler:

CBlockingSocket g\_sListen; // one-and-only global socket for listening

void CSocketView::OnInternetStartServer()

{

try {

CSockAddr saServer(INADDR\_ANY, 80);

g\_sListen.Create();

g\_sListen.Bind(saServer);

g\_sListen.Listen();

AfxBeginThread(ServerThreadProc, GetSafeHwnd());

}

catch(CBlockingSocketException\* e) {

g\_sListen.Cleanup();

// Do something about the exception

e->Delete();

}

}

Pretty simple, really. The handler creates a socket, starts listening on it, and then starts a worker thread that waits for some client to connect to port 80. If something goes wrong, an exception is thrown. The global *g\_sListen* object lasts for the life of the program and is capable of accepting multiple simultaneous connections, each managed by a separate thread.

### The Server Thread

Now let's look at the *ServerThreadProc* function:

UINT ServerThreadProc(LPVOID pParam)

{

CSockAddr saClient;

CHttpBlockingSocket sConnect;

char request[100];

char headers[] = "HTTP/1.0 200 OK\r\n"

"Server: Inside Visual C++ SOCK01\r\n"

"Date: Thu, 05 Sep 1996 17:33:12 GMT\r\n"

"Content-Type: text/html\r\n"

"Accept-Ranges: bytes\r\n"

"Content-Length: 187\r\n"

"\r\n"; // the important blank line

char html[] =

"<html><head><title>Inside Visual C++ Server</title></head>\r\n"

"<body><body background=\"/samples/images/usa1.jpg\">\r\n"

"<h1><center>This is a custom home page</center></h1><p>\r\n"

"</body></html>\r\n\r\n";

try {

if(!g\_sListen.Accept(sConnect, saClient)) {

// Handler in view class closed the listening socket

return 0;

}

AfxBeginThread(ServerThreadProc, pParam);

// read request from client

sConnect.ReadHttpHeaderLine(request, 100, 10);

TRACE("SERVER: %s", request); // Print the first header

if(strnicmp(request, "GET", 3) == 0) {

do { // Process the remaining request headers

sConnect.ReadHttpHeaderLine(request, 100, 10);

TRACE("SERVER: %s", request); // Print the other headers

} while(strcmp(request, "\r\n"));

sConnect.Write(headers, strlen(headers), 10); // response hdrs

sConnect.Write(html, strlen(html), 10); // HTML code

}

else {

TRACE("SERVER: not a GET\n");

// don't know what to do

}

sConnect.Close(); // Destructor doesn't close it

}

catch(CBlockingSocketException\* e) {

// Do something about the exception

e->Delete();

}

return 0;

}

The most important function call is the *Accept* call. The thread blocks until a client connects to the server's port 80, and then *Accept* returns with a new socket, *sConnect*. The current thread immediately starts another thread.

In the meantime, the current thread must process the client's request that just came in on *sConnect*. It first reads all the request headers by calling *ReadHttpHeaderLine* until it detects a blank line. Then it calls *Write* to send the response headers and the HTML statements. Finally, the current thread calls *Close* to close the connection socket. End of story for this connection. The next thread is sitting, blocked at the *Accept* call, waiting for the next connection.

### Cleaning Up

To avoid a memory leak on exit, the program must ensure that all worker threads have been terminated. The simplest way to do this is to close the listening socket. This forces any thread's pending *Accept* to return *FALSE*, causing the thread to exit.

try {

g\_sListen.Close();

Sleep(340); // Wait for thread to exit

WSACleanup(); // Terminate Winsock

}

catch(CUserException\* e) {

e->Delete();

}

A problem might arise if a thread were in the process of fulfilling a client request. In that case, the main thread should positively ensure that all threads have terminated before exiting.

**Q: Write a short note on WinInet? Explain Advantages of WinInt over Winsock?**

**Ans:**

# WinInet

WinInet is a higher-level API than Winsock, but it works only for HTTP, FTP, and gopher client programs in both asynchronous and synchronous modes. You can't use it to build servers. The WININET DLL is independent of the WINSOCK32 DLL. Microsoft Internet Explorer 3.0 (IE3) uses WinInet, and so do ActiveX controls.

## WinInet's Advantages over Winsock

WinInet far surpasses Winsock in the support it gives to a professional-level client program. Following are just some of the WinInet benefits:

* **Caching**—Just like IE3, your WinInet client program caches HTML files and other Internet files. You don't have to do a thing. The second time your client requests a particular file, it's loaded from a local disk instead of from the Internet.
* **Security**—WinInet supports basic authentication, Windows NT challenge/response authentication, and the Secure Sockets Layer (SSL). Authentication is described in [Chapter 35](about:blank).
* **Web proxy access**—You enter proxy server information through the Control Panel (click on the Internet icon), and it's stored in the Registry. WinInet reads the Registry and uses the proxy server when required.
* **Buffered I/O**—WinInet's read function doesn't return until it can deliver the number of bytes you asked for. (It returns immediately, of course, if the server closes the socket.) Also, you can read individual text lines if you need to.
* **Easy API**—Status callback functions are available for UI update and cancellation. One function, *CInternetSession::OpenURL*, finds the server's IP address, opens a connection, and makes the file ready for reading, all in one call. Some functions even copy Internet files directly to and from disk.
* **User friendly**—WinInet parses and formats headers for you. If a server has moved a file to a new location, it sends back the new URL in an HTTP Location header. WinInet seamlessly accesses the new server for you. In addition, WinInet puts a file's modified date in the request header for you.

## The MFC WinInet Classes

WinInet is a modern API available only for Win32. The MFC wrapping is quite good, which means we didn't have to write our own WinInet class library. Yes, MFC WinInet supports blocking calls in multithreaded programs, and by now you know that makes us happy.

The MFC classes closely mirror the underlying WinInet architecture, and they add exception processing. These classes are summarized in the sections on the following pages.

### *CInternetSession*

You need only one *CInternetSession* object for each thread that accesses the Internet. After you have your *CInternetSession* object, you can establish HTTP, FTP, or gopher connections or you can open remote files directly by calling the *OpenURL* member function. You can use the *CInternetSession* class directly, or you can derive a class from it in order to support statuscallback functions.

The *CInternetSession* constructor calls the WinInet *InternetOpen* function, which returns an *HINTERNET*sessionhandle that is stored inside the *CInternetSession* object. This function initializes your application's use of the Win- Inet library, and the session handle is used internally as a parameter for other WinInet calls.

### *CHttpConnection*

An object of class *CHttpConnection* represents a "permanent" HTTP connection to a particular host. You know already that HTTP doesn't support permanent connections and that FTP doesn't either. (The connections last only for the duration of a file transfer.) WinInet gives the appearance of a permanent connection because it remembers the host name.

After you have your *CInternetSession* object, you call the *GetHttpConnection* member function, which returns a pointer to a *CHttpConnection* object. (Don't forget to delete this object when you are finished with it.)

The *GetHttpConnection* member function calls the WinInet *InternetConnect* function, which returns an *HINTERNET*connectionhandle that is stored inside the *CHttpConnection* object and used for subsequent WinInet calls.

### *CFtpConnection*, *CGopherConnection*

These classes are similar to *CHttpConnection*, but they use the FTP and gopher protocols. The *CFtpConnection* member functions *GetFile* and *PutFile* allow you to transfer files directly to and from your disk.

### *CInternetFile*

With HTTP, FTP, or gopher, your client program reads and writes byte streams. The MFC WinInet classes make these byte streams look like ordinary files. If you look at the class hierarchy, you'll see that *CInternetFile* is derived from *CStdioFile*, which is derived from *CFile*. Therefore, *CInternetFile* and its derived classes override familiar *CFile* functions such as *Read* and *Write*. For FTP files, you use *CInternetFile* objects directly, but for HTTP and gopher files, you use objects of the derived classes *CHttpFile* and *CGopherFile*. You don't construct a *CInternetFile* object directly, but you call *CFtpConnection::OpenFile* to get a *CInternetFile* pointer.

If you have an ordinary *CFile* object, it has a 32-bit *HANDLE* data member that represents the underlying disk file. A *CInternetFile* object uses the same *m\_hFile* data member, but that data member holds a 32-bit Internetfilehandle of type *HINTERNET*, which is not interchangeable with a *HANDLE*. The *CInternetFile* overridden member functions use this handle to call WinInet functions such as *InternetReadFile* and *InternetWriteFile*.

### *CHttpFile*

This Internet file class has member functions that are unique to HTTP files, such as *AddRequestHeaders*, *SendRequest*, and *GetFileURL*. You don't construct a *CHttpFile* object directly, but you call the *CHttpConnection::OpenRequest* function, which calls the WinInet function *HttpOpenRequest* and returns a *CHttpFile* pointer. You can specify a GET or POST request for this call.

Once you have your *CHttpFile* pointer, you call the *CHttpFile::SendRequest* member function, which actually sends the request to the server. Then you call *Read*.

### *CFtpFileFind*, *CGopherFileFind*

These classes let your client program explore FTP and gopher directories.

### *CInternetException*

The MFC WinInet classes throw *CInternetException* objects that your program can process with try/catch logic.

**Chapter-35:**

**Programming the Microsoft Internet Information Server**

IIS is actually three separate servers—one for HTTP (for the World Wide Web), one for FTP, and one for gopher. There are two kinds of extensions: an ISAPI server extension and an ISAPI filter, both of which are DLLs. An ISAPIserverextension can perform Internet business transactions such as order entry. An ISAPIfilter intercepts data traveling to and from the server and thus can perform specialized logging and other tasks.

**IIS Alternatives**

You have Windows NT Server 4.0 and IIS. If you are running Windows NT Workstation, you can use Peer Web Services, which supports fewer connections and doesn't allow virtual servers. If you are running Microsoft Windows 95 or Windows 98, you can use Personal Web Server, which is packaged with Microsoft FrontPage. Internet Information Server, Peer Web Services, and Personal Web Server can all use ISAPI extension DLLs.

**Q: Write a short note on Microsoft IIS?**

**Ans:**

# Microsoft IIS

Microsoft IIS is a high-performance Internet/intranet server that takes advantage of Windows NT features such as I/O completion ports, the Win32 function *TransmitFile*, file-handle caching, and CPU scaling for threads.

## Installing and Controlling IIS

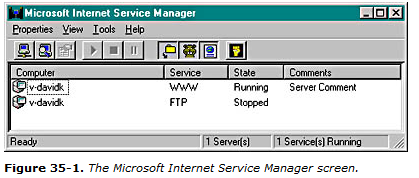
When you install Windows NT Server 4.0, you are given the option of installing IIS. If you selected IIS at setup, the server will be running whenever Windows NT is running. IIS is a special kind of Win32 program called a service (actually three services—WWW, HTTP, and gopher—in one program called inetinfo.exe), which won't appear in the taskbar. You can control IIS from the Services icon in the Control Panel.

## Running Internet Service Manager

You can run Internet Service Manager from the Microsoft Internet Server menu that's accessible on the Start menu.

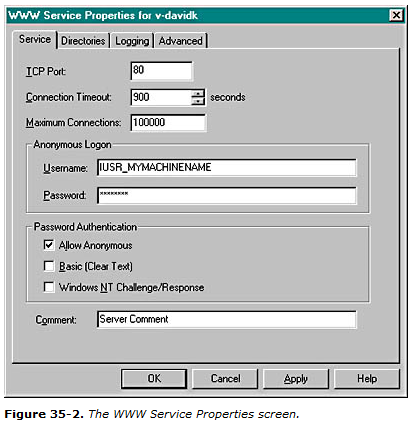
Figure 35-1 shows the Microsoft Internet Service Manager screen with the World Wide Web (WWW) running and FTP services stopped.

You can select a service by clicking on its icon at the left. The triangle and square buttons on the toolbar of the screen allow you to turn the selected service on or off.



### IIS Security

After you double-click on the WWW service icon of the Microsoft Internet Service Manager screen, you'll see a property sheet. The Service page lets you configure IIS security. When a client browser requests a file, the server impersonates a local user for the duration of the request and that user name determines which files the client can access. Which local user does the server impersonate? Most often, it's the one you see in the Username field, shown in Figure 35-2.

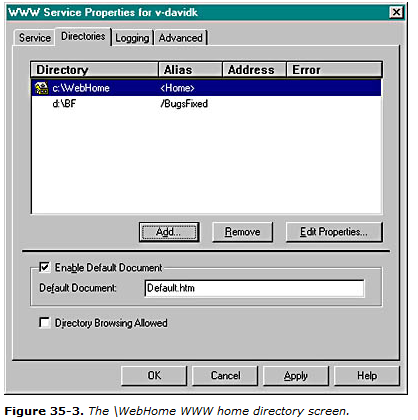


Most Web page visitors don't supply a user name and password, so they are considered anonymoususers. Those users have the same rights they would have if they had logged on to your server locally as IUSR\_MYMACHINENAME. That means that IUSR\_MYMACHINENAME must appear in the list of users that is displayed when you run User Manager or User Manager For Domains (from the Administrative Tools menu), and the passwords must match. The IIS Setup program normally defines this anonymous user for you. You can define your own WWW anonymous user name, but you must be sure that the entry on the Service page matches the entry in the computer's (or Windows NT domain's) user list.

Note also the Password Authentication options. For the time being, stick to the Allow Anonymous option only, which means that all Web users are logged on as IUSR\_MYMACHINENAME. Later in this chapter, we'll explain Windows NT Challenge/Response.

### IIS Directories

If you requested the URL http://slowsoft.com/newproducts.html, the newproducts.html file would be displayed from the slowsoft.com homedirectory. Each server needs a home directory, even if that directory contains only subdirectories. The home directory does not need to be the server computer's root directory, however. As shown in Figure 35-3, the WWW home directory is really \WebHome, so clients read the disk file \WebHome\newproducts.html.



Your server could get by with a home directory only, but the IIS virtual directory feature might be useful. Suppose SlowSoft wanted to allow Web access to the directory \BF on the D drive. The screen above shows a virtual /BugsFixed directory that maps to D:\BF. Clients would access files with a URL similar to this: http://slowsoft.com/BugsFixed/file1.html.

### IIS Logging

IIS is capable of making log entries for all connections. You control logging from the Internet Service Manager's Logging property page. You can specify text log files, or you can specify logging to an SQL/ODBC database. Log entries consist of date, time, client IP address, file requested, query string, and so forth.

## Testing IIS

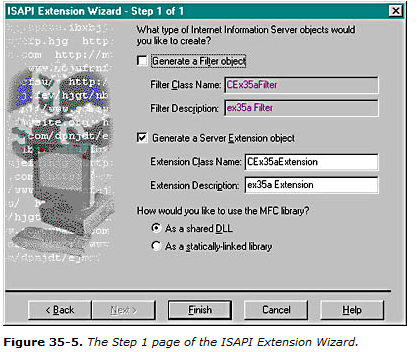
It's easy to test IIS with a browser or with any of the EX35A clients. Just make sure that IIS is running and that the EX35A server is not running. The default IIS home directory is \Winnt\System32\inetsrv\wwwroot, and some HTML files are installed there.

**Q: Write steps to create ISAPI sever extension DLL?**

**Ans:**

## Writing an ISAPI Server Extension DLL

Visual C++ gives you a quick start for writing ISAPI server extensions. Just select ISAPI Extension Wizard from the Projects list. After you click the OK button, your first screen looks like Figure 35-5.



Check the Generate A Server Extension Object box, and you've got a do-nothing DLL project with a class derived from the MFC *CHttpServer* class and a *Default* member function. Now it's time for a little programming.

You must write your ISAPI functions as members of the derived *CHttpServer* class, and you must write parsemap macros to link them to IIS. There's no "parse map wizard," so you have to do some coding. It's okay to use the *Default* function, but you need a *GetMap* function too. First insert these lines into the wizard-generated parse map:

ON\_PARSE\_COMMAND(GetMap, CWeatherExtension, ITS\_PSTR)

ON\_PARSE\_COMMAND\_PARAMS("State")

Then write the *GetMap* function:

void CWeatherExtension::GetMap(CHttpServerContext\* pCtxt, LPCTSTR pstrState)

{

StartContent(pCtxt);

WriteTitle(pCtxt);

\*pCtxt << "Visualize a weather map for the state of ";

\*pCtxt << pstrState;

EndContent(pCtxt);

}

This function doesn't actually generate the weather map (what did you expect?), but it does display the selected state name in a custom HTML file. The *CHttpServer::StartContent* and *CHttpServer::EndContent* functions write the HTML boilerplate, and *CHttpServer::WriteTitle* calls the virtual *CHttpServer::GetTitle* function, which you need to override:

LPCTSTR CWeatherExtension::GetTitle() const

{

return "Your custom weather map"; // for browser's title window

}

The MFC *CHttpServerContext* class has an overloaded << operator, which you use to put text in the HTML file you're building. Behind the scenes, an attached object of the MFC class *CHtmlStream* represents the output to the server's socket.

## The MFC ISAPI Server Extension Classes

Now is a good time to review the three MFC classes that are used to create an MFC ISAPI server extension. Remember that these classes are for ISAPI server extensions only. Don't even think of using them in ordinary Winsock or WinInet applications.

### *CHttpServer*

With the help of the ISAPI Extension Wizard, you derive a class from *CHttpServer* for each ISAPI server extension DLL that you create. You need one member function for each extension function (including the default function), and you need an overridden *GetTitle* function. The framework calls your extension functions in response to client requests, using the connections established in the parse map. The ISAPI Extension Wizard provides an overridden *GetExtensionVersion* function, which you seldom edit unless you need initialization code to be executed when the DLL is loaded.

One of the *CHttpServer* memberfunctions that you're likely to call is *AddHeader*, which adds special response headers, such as Set-Cookie, before the response is sent to the server. (More on cookies later.)

### *CHttpServerContext*

There's one *CHttpServer* object per DLL, but there is one *CHttpServerContext* object for each server transaction request. Your extension functions each provide a pointer to one of these objects. You don't derive from *CHttpServerContext*, so you can't easily have variables for individual transactions. Because different IIS threads can manage transactions, you have to be careful to perform synchronization for any data members of your *CHttpServer* class or global variables.

You've already seen the use of the *StartContent*, *EndContent*, and *WriteTitle* functions of the *CHttpServer* class plus the overloaded >> operator. You might also need to call the *CHttpServerContext::GetServerVariable* function to read information sent by the client in the request headers.

### *CHtmlStream*

Most of the time, you don't use the *CHtmlStream* class directly. The *CHttpServerContext* class has a *CHtmlStream* data member, *m\_pStream*, that's hooked up to the >> operator and serves as the output for HTML data. You could access the *CHtmlStream* object and call its *Write* member function if you needed to send binary data to the client. Because objects of the *CHtmlStream* class accumulate bytes in memory before sending them to the client, you need an alternative approach if your DLL relays large files directly from disk.

**Q: Explain ISAPI filters?Or**

**Write a short note on ISAPI filters?**

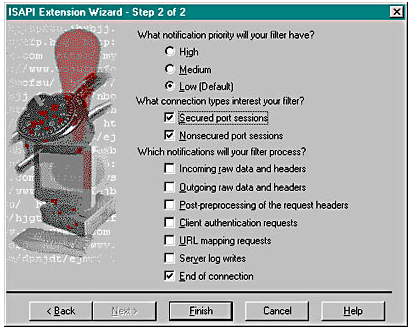
**Ans:**

# ISAPI Filters

An ISAPI server extension DLL is loaded the first time a client references it in a GET or POST request. An ISAPI filter DLL is loaded (based on a Registry entry) when the WWW service is started. The filter is then in the loop for all HTTP requests, so you can read and/or change any data that enters or leaves the server.

## Writing an ISAPI Filter DLL

The ISAPI Extension Wizard makes writing filters as easy as writing server extensions. Choose Generate A Filter Object, and Step 2 looks like this.



The list of options under Which Notifications Will Your Filter Process? refers to seven places where your filter can get control during the processing of an HTTP request. You check the boxes, and the wizard generates the code.

## The MFC ISAPI Filter Classes

There are two MFC classes for ISAPI filters, *CHttpFilter* and *CHttpFilterContext*.

### *CHttpFilter*

With the help of the ISAPI Extension Wizard, you derive a class from *CHttpFilter* for each ISAPI filter you create. There's just one object of this class. The class has virtual functions for each of seven notifications. The list of filters in the order in which IIS calls them is below.

virtual DWORD OnReadRawData(CHttpFilterContext\* pCtxt,

PHTTP\_FILTER\_RAW\_DATA pRawData);

virtual DWORD OnPreprocHeaders(CHttpFilterContext\* pCtxt,

PHTTP\_FILTER\_PREPROC\_HEADERS pHeaderInfo);

virtual DWORD OnUrlMap(CHttpFilterContext\* pCtxt,

PHTTP\_FILTER\_URL\_MAP pMapInfo);

virtual DWORD OnAuthentication(CHttpFilterContext\* pCtxt,

PHTTP\_FILTER\_AUTHENT pAuthent);

virtual DWORD OnSendRawData(CHttpFilterContext\* pCtxt,

PHTTP\_FILTER\_RAW\_DATA pRawData);

virtual DWORD OnLog(CHttpFilterContext\* pfc, PHTTP\_FILTER\_LOG pLog);

virtual DWORD OnEndOfNetSession(CHttpFilterContext\* pCtxt);

If you override a function, you get control. It would be inefficient, however, if IIS made virtual function calls for every notification for each transaction. Another virtual function, *GetFilterVersion*, is called once when the filter is loaded. The ISAPI Extension Wizard always overrides this function for you, and it sets flags in the function's *pVer* parameter, depending on which notifications you want. Here's a simplified sample with all the flags set:

BOOL CMyFilter::GetFilterVersion(PHTTP\_FILTER\_VERSION pVer)

{

CHttpFilter::GetFilterVersion(pVer);

pVer->dwFlags |= SF\_NOTIFY\_ORDER\_LOW | SF\_NOTIFY\_NONSECURE\_PORT |

SF\_NOTIFY\_LOG | SF\_NOTIFY\_AUTHENTICATION |

SF\_NOTIFY\_PREPROC\_HEADERS | SF\_NOTIFY\_READ\_RAW\_DATA |

SF\_NOTIFY\_SEND\_RAW\_DATA | SF\_NOTIFY\_URL\_MAP |

SF\_NOTIFY\_END\_OF\_NET\_SESSION;

return TRUE;

}

If you had specified URL mapping requests only, the wizard would have set only the *SF\_NOTIFY\_URL\_MAP* flag and it would have overridden only *OnUrlMap*. IIS would not call the other virtual functions, even if they were overridden in your derived class.

### *CHttpFilterContext*

An object of this second MFC class exists for each server transaction, and each of the notification functions gives you a pointer to that object. The *CHttpFilterContext* member functions you might call are *GetServerVariable*, *AddResponseHeaders*, and *WriteClient*.

**Chapter-36:**

**ActiveX Document Servers and the Internet**

An ActiveXdocument is a special file that you can download from a Web server. When the browser sees an ActiveX document file, it automatically loads the corresponding ActiveXdocumentserver program from your hard disk, and that program takes over the whole browser window to display the contents of the document. The Microsoft Internet Explorer browser is not the only ActiveXdocumentcontainer program.

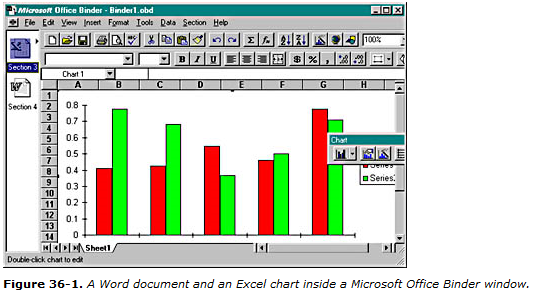
**Q: Explain ActiveX Document server Vs OLE embedded server?**

**Ans:**

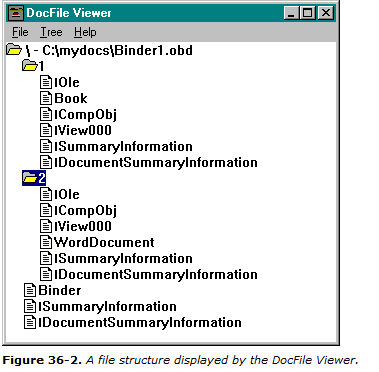
## ActiveX Document Servers vs. OLE Embedded Servers

An OLE embedded server program runs in a child window of an OLE container application and occupies a rectangular area in a page of the container's document (see Figure 28-1). Unless an embedded server program is classified as a mini-server, it can run stand-alone also. In embedded mode, the server program's data is held in a storage inside the container application's file. The embedded server program takes over the container program's menu and toolbar when the user activates it by double-clicking on its rectangle.

In contrast to an embedded server, an ActiveX document server takes over a whole frame window in its container application, and the document is always active. An ActiveX server application, running inside a container's frame window, runs pretty much the same way it would in stand-alone mode. You can see this for yourself if you have Microsoft Office 97. Office includes an ActiveX container program called Binder, and the Office applications have ActiveX server capability. Figure 36-1 shows a Word document and an Excel chart inside the same binder.



Like an embedded server, the ActiveX document server saves its data in a storage inside the ActiveX container's file. When the Office user saves the Binder program from the File menu, Binder writes a single OBD file to disk; the file contains one storage for the Word document and another for the Excel spreadsheet. You can see this file structure yourself with the DFVIEW utility, as shown in Figure 36-2.

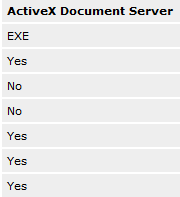
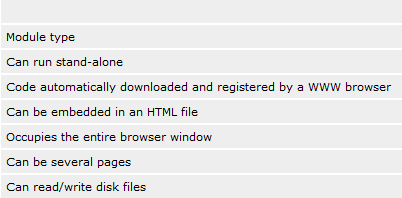


**Q: Explain ActiveX Document server Vs ActiveX control?**

**Ans:**

## ActiveX Document Servers vs. ActiveX Controls

Both ActiveX document servers and ActiveX controls can run with and without the Internet. Both are compiled programs that can run inside a browser. The following table lists some of the differences between the two.



**Q: Explain the OLE Interface for ActiveX Document Server and Containers?**

**Ans:**

## OLE Interfaces for ActiveX Document Servers and Containers

ActiveX document servers implement the same interfaces as OLE embedded servers, including *IOleObject*, *IOleInPlaceObject*, and *IOleInPlaceActiveObject*. ActiveX document containers implement *IOleClientSite*, *IOleInPlaceFrame*, and *IOleInPlaceSite*. The menu negotiation works the same as it does for Visual Editing.

Some additional interfaces are implemented, however. ActiveX document servers implement *IOleDocument*, *IOleDocumentView*, *IOleCommandTarget*, and *IPrint.* ActiveX document containers implement *IOleDocumentSite*. The architecture allows for multiple views of the same document—sort of like the MFC document-view architecture—but most ActiveX document servers implement only one view per document.

The critical function in an OLE embedded server is *IOleObject::DoVerb*, which is called by the container when the user double-clicks on an embedded object or activates it through the menu. For an ActiveX document server, however, the critical function is *IOleDocumentView::UIActivate*. (Before calling this function, the container calls *IOleDocument::CreateView*, but generally the server just returns an interface pointer to the single document-view object.) *UIActivate* finds the container site and frame window, sets that window as the server's parent, sets the server's window to cover the container's frame window, and then activates the server's window.

**Chapter-37:**

**Introducing Dynamic HTML**

Dynamic HyperText Markup Language (DHTML) is a new and exciting technology—introduced as part of Microsoft Internet Explorer 4.0 (IE4)—that provides serious benefits to Webmasters and developers. DHTML could ultimately change the way we think about developing Windows applications.

It began with the IE4 "HTML display engine"—sometimes called Trident in Microsoft literature. As part of the design of Internet Explorer 4, Microsoft made Trident a COM component that exposes many of its internal objects that are used for displaying HTML pages in Internet Explorer 4. The real power of DHTML, however, is not this ability to access the HTML objects but the ability to actually change and manipulate the HTML page on the fly—thus the name Dynamic HTML.

Once you grasp the concept of DHTML, a million possible applications come to mind. For Webmasters, DHTML means that much of the logic that manipulates a Web page can live in scripts that are downloaded to the client. C++ developers can embed DHTML in their applications and use it as an embedded Web client or as a super-flexible, dynamic "form" that their application changes on the fly.

**Q: Describe in detail DHTML Object Model?**

**Ans:**

# The DHTML Object Model

If you've been heads down on a Visual C++ project and haven't yet had time to take a peek at HTML, the first thing you should know is that HTML is an ASCII markup language format. Here is the code for a very basic HTML page:

<html>

<head>

<title>

This is an example very basic HTML page!

</title>

</head>

<body>

<h1>This is some text with H1!

</h1>

<h3>

This is some text with H3!

</h3>

</body>

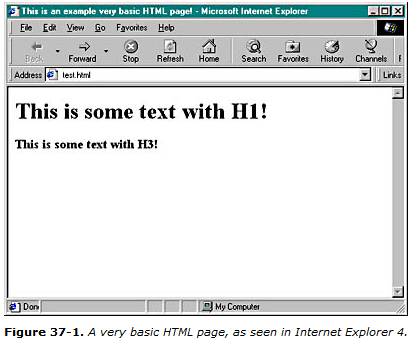
</html>

This basic HTML "document" is composed of the following elements:

* **A head (or header)** In this example, the header contains a title: "This is an example very basic HTML page!"
* **The body of the document** The body in this example contains two text elements. The first has the heading 1 (h1) style and reads, "This is some text with H1!" The second text element has the heading 3 (h3) style and reads, "This is some text with H3!"

The end result is an HTML page that—when displayed in Internet Explorer 4—looks like Figure 37-1.

When Internet Explorer 4 loads this sample HTML page, it creates an internal representation that you can traverse, read, and manipulate through the DHTML object model. Figure 37-2 shows the basic hierarchy of the DHTML object model.



At the root of the object model is the *window* object. This object can be used from a script to perform some action, such as popping up a dialog box. Here's an example of some JScript that accesses the window object:

<SCRIPT LANGUAGE="JScript">

function about()

{

window.showModalDialog("about.htm","",

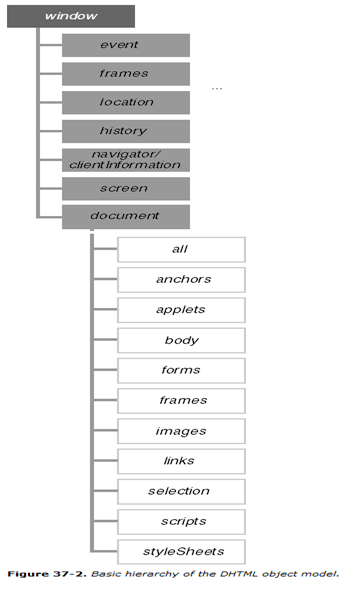
"dialogWidth:25em;dialogHeight13em")

}

</SCRIPT>

When the *about* script function is called, it will call the *showModalDialog* function in the *window* DHTML object to display a dialog. This example also illustrates how scripts access the object model—through globally accessible objects that map directly to the corresponding object in the DTHML object model.

The window object has several "subobjects" that allow you to further manipulate portions of Internet Explorer 4. The *document* object is what we will spend most of our time on in this chapter because it gives us programmatic access to the various elements of the currently loaded HTML document. Below, some JScript shows how to create basic dynamic content that changes the document object.



**Chapter-38:**

**Visual C++ for Windows CE**

Microsoft released a new version of the Windows family of operating systems named Windows CE. Original equipment managers (OEMs) are the target audience for Windows CE. OEMs create small, portable devices—such as hand-held computers—and embedded systems.

Microsoft hopes that Windows CE can do for the embedded and handheld markets what Windows did for the desktop PC industry. Based on the target audience, you can probably guess that Windows CE has different design goals than Windows 98 and Windows NT. One goal was modularity: if an OEM is using Windows CE in an embedded device for a refrigerator, the keyboard and graphics output modules are not required. The OEM does not pay a penalty for modules not used by the application of Windows CE.

There have been two major releases of Windows CE. The first release was primarily for Handheld PCs (H/PCs) and was limited to noncolor applications. Windows CE 1.0 lacked many advanced Win32 features such as COM and ActiveX, large chunks of GDI, and many Windows controls. Win- dows CE 2.0 was released in late 1997 and added support for a variety of new device types, color, COM and ActiveX technology, and also a Java virtual machine.

**Q: Explain the features of VC++ for windows CE?**

**Ans:**

# Visual C++ for Windows CE

Visual C++ for Windows CE is an add-on to Visual C++. When you install C++ for Windows CE, it extends the Visual C++ environment by adding several Windows CE-specific features:

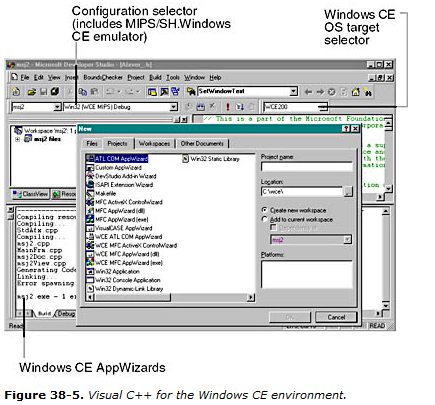
* An Intel-based Windows CE emulation environment
* New targets for each of the Windows CE supported processors (MIPS/SH and the emulation environment)
* New AppWizards for Windows CE applications
* A Windows CE compatible port of MFC
* A Windows CE compatible port of ATL
* Tools for remote execution and debugging of Windows CE applications on actual devices

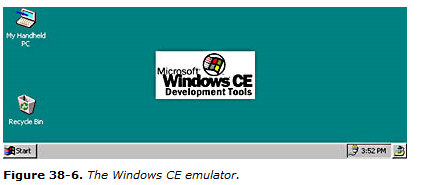
One interesting aspect of Visual C++ for Windows CE is the fact that it also supports the older 1.0 and 1.01 versions of Windows CE. Figure 38-5 shows the Windows CE operating system and processor configuration bars that have been added to Visual C++.

While the environment lets you remotely run and debug your applications on a connected Windows CE device, it also includes a very powerful Windows CE emulation environment. The Windows CE emulator (WCE) is an Intel-based software-only version of Windows CE that runs on your desktop and gives you the convenience of being able to run and test your applications on your development machine. Of course, to ensure that your applications work correctly, you still need to test on real devices, but the emulator takes much of the pain out of the early compile and debug stages of Windows CE development. Figure 38-6 shows the emulation environment in action.

There are four Windows-CE-specific AppWizards that ship with Visual C++ for WCE:

* **WCE ATL COM AppWizard**—An ATL-based COM object project
* **WCE MFC ActiveX ControlWizard**—An MFC ActiveX control project
* **WCE MFC AppWizard (dll)**—An MFC DLL project
* **WCE MFC AppWizard (exe)**—An MFC executable project





The WCE AppWizards are basically the same as their big brother Win32 counterparts, except that they have different features that take advantage of the Windows CE operating system, such as the Windows CE help environment. Figure 38-7 shows the first three steps of the Windows CE MFC executable AppWizard. Notice that there are only two project types: SDI and dialog-based. Notice also the variety of Windows-CE-specific options that you can choose from.

