WHITE PAPER

PowerBuilder® and EAServer Uniting the .NET and J2EE Communities

JIM O'NEIL

SYBASE®

TABLE OF CONTENTS

2 Connection Object
2 JaguarORB
5 Secure Client Connections
5 Win32 Client Applications
6 .NET Targets
9 Summary

In PowerBuilder 11.2, .NET meets J2EE head-on with the capability to deploy .NET Windows Forms and Web Forms applications (as well as assemblies and Web Services) that access Enterprise Java Beans (EJBs) in Sybase's own EAServer. As you'll see over the course of this article, integrating these "competing" technologies is quite straightforward and leverages mechanisms that have been available since PowerBuilder 7.

The enabling technology for the .NET to J2EE functionality in PowerBuilder is the new client support introduced in EAServer 6.1. The EAServer .NET runtime includes three fully-managed code assemblies:

com.sybase.iiop.net.dll	supporting the Internet Inter-ORB Protocol (IIOP), data marshalling, and secure sockets layer (SSL)
com.sybase.ejb.net.dll	supporting invocation of EJBs as well as PowerBuilder components (NVOs)
com.sybase.jms.net.dll	supporting interaction with the Java Message Service (JMS)

The first two of these assemblies are the .NET analog of the C++ client ORB (implemented by *libjcc.dll*) and so provide the underlying implementation for the following PowerBuilder classes:

- Connection
- JaguarORB
- CORBAObject
- CORBACurrent

These assemblies are automatically installed by PowerBuilder 11.2 in the Global Assembly Cache (GAC) since they need to be available for deploying .NET targets as well as at runtime. To aid in distributing end-user applications that access EAServer, the PowerBuilder Runtime Packager automatically installs these two assemblies in the GAC as well. If you forego use of the Runtime Packager, you can use the .NET *gacutil* utility to explicitly add the assemblies. You would also use *gacutil* to update the runtime environment with any patches issued for EAServer.

As for the last assembly, *com.sybase.jms.net.dll*, PowerBuilder does not offer direct access to the Java Message Service, so the file is not automatically installed; however, methods in that assembly can be accessed in conditional compilation blocks in your PowerScript code, just like other third-party and .NET Framework classes. Alternatively, you can access the message service functionality in your applications via EAServer's *CtsComponents::MessageService* API provided for CORBA clients such as PowerBuilder.

The techniques to access components in EAServer 6 from classic (Win32) PowerBuilder applications do not differ from previous versions of the server; however, the .NET client capability in EAServer 6.1, which is used by PowerBuilder .NET targets, does necessitate changes in application implementation and deployment. For the most part, though, you'll find that your job as a developer is actually made easier! Let's take a further look now at the changes to be aware of when using the Connection object, the JaguarORB object, and secure connections in .NET applications developed with PowerBuilder. A complete application demonstrating the various PowerBuilder-to-EAServer connectivity options is available in the "PB & EAServer" section at **https://powerbuilder.codexchange.sybase.com/files/documents/67/2837/pb11eas61.zip** and may help clarify the concepts discussed here.

CONNECTION OBJECT

Applications built for Win32 deployment that use the Connection object to access a non-secure IIOP port do not need to be modified when deployed as .NET targets. Since the Connection object abstracts the underlying implementations of the EAServer ORB, the interface exposed to the PowerBuilder developer is independent of the deployment target – Win32 or .NET. In fact, since the new .NET client functionality was not influenced by the long defunct Distributed PowerBuilder feature, the list of codes returned by the main methods of the Connection object – *ConnectToServer* and *CreateInstance* – is actually shorter!

Return Code	Explanation
0	Success
50	Distributed service error
57	Not connected
92	Required property missing or invalid
100	Unknown error

Listing 1 below provides sample code (for any PowerBuilder target type) that instantiates a Connection object and invokes a few component methods. In this sample and the ones that follow, *mathComp* is a proxy reference to a component named *SimpleMath* located in the EAServer package *PBDJ*. Also assume that *argX* and *argY* are integral values passed as parameters or perhaps provided as input by the end user.

JAGUARORB

The JaguarORB object in PowerBuilder is a lower-level wrapper of the client ORB functionality and provides finer control over instantiating and invoking EAServer components. When using JaguarORB, client applications are responsible for explicitly creating the EAServer session as well as instantiating the client proxy. A primary use of JaguarORB is to override the load balancing implicit in the Connection object, thus forcing component instantiation to occur on a specific EAServer instance.

Listing 2 below demonstrates using JaguarORB to provide the same functionality as shown in the previous code example. Again, this sample code will work equally well regardless of whether it's deployed as a p-code or machine code application or as a .NET target.

Unlike the example using the Connection object in Listing 1, this code can actually terminate execution of the client application if exceptions are raised in the intermediate components *SessionManager/Manager* and *SessionManager/Session*. To address this, TRY/CATCH blocks should be incorporated into the client application code to provide adequate error detection and recovery. PowerBuilder's object hierarchy defines a number of CORBA exceptions that extend the RuntimeError class to trap for various error conditions. A sample TRY/CATCH block follows that could envelop Listing 2 and provide error handling for invalid credentials as well as for other remote and user-defined exceptions.

```
TRY
        // ... include listing 2 here
CATCH (CORBANoPermission exc1)
        // handle invalid user id/password combination
CATCH (CORBAUnknown exc2)
        // handle invalid component name and other errors
CATCH (CORBASystemException exc3)
        // handle other EAServer exceptions
CATCH (CORBAUserException exc4)
        // handle user-defined exceptions thrown in component method calls
END TRY
```

This error handling code will serve you well when deploying traditional p-code or machine code applications; however, due to how exceptions are handled in PowerBuilder .NET targets, these CATCH clauses will not be triggered. In fact, even the all-encompassing clause

```
CATCH (Throwable t)
```

will not capture the exceptions raised in an EAServer .NET client application. Figure 1, extracted from the PowerBuilder documentation, shows why.

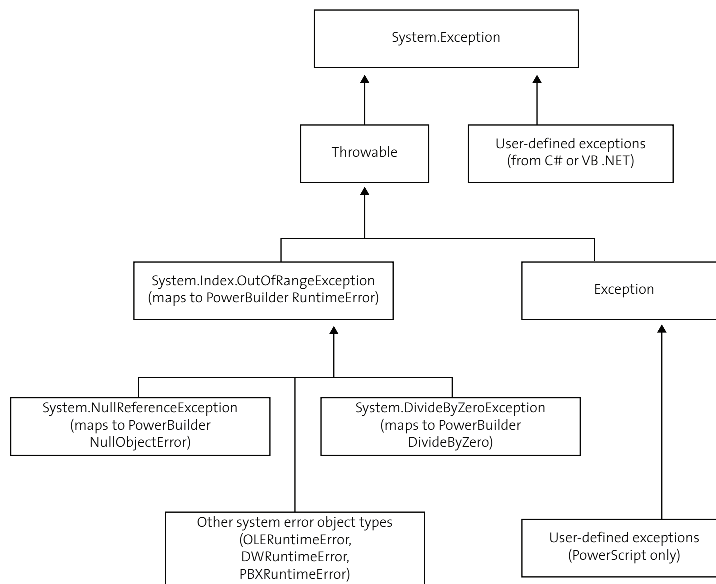

Figure 1

Note that most exceptions emanating from managed code implementations (C# and VB.NET) extend the base .NET *System.Exception* class so they cannot be caught from the built-in PowerBuilder exception types that descend from Throwable. There are a few deviations from this rule, such as *System.NullReferenceException* and *System.DivideByZeroException*, which do map to predefined PowerBuilder runtime errors. Although many CORBASystemException types are also defined in the PowerObject hierarchy, they are not currently mapped to the analogous .NET exceptions raised in the EAServer client assemblies. User-defined exceptions raised in component methods do, however, map appropriately to the CORBAUserException type (which extends Exception in the PowerObject hierarchy).

So, to address these exception-handling nuances in your PowerBuilder .NET targets, you'll need to introduce a CATCH clause for the specific .NET exceptions you want to capture. For example, you could add the following CATCH clause to handle the exceptions raised from an EAServer .NET client application:

```
#IF DEFINED PBDOTNET THEN
        CATCH (System.Exception exc4)
        // handle .NET exceptions
#END IF
```

Note the use of the conditional compilation block to allow referencing a .NET class, here *System.Exception*. For this code to deploy successfully, the .NET assembly containing *System.Exception*, namely *mscorlib.dll*, must be referenced in your PowerBuilder target's properties as shown in Figure 2.

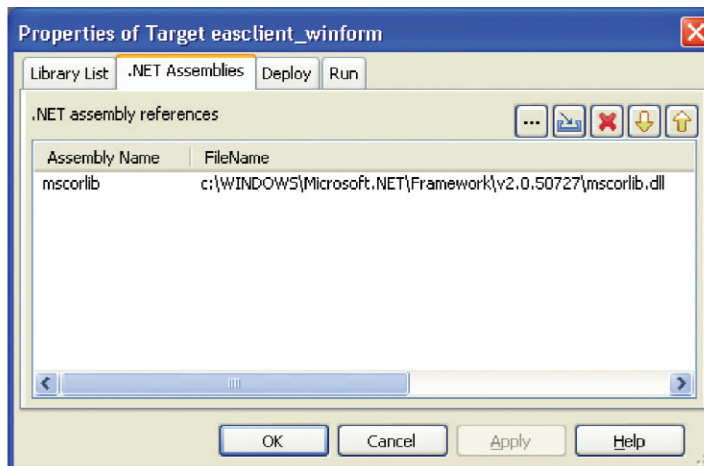

Figure 2

While you might be inclined to provide finer-grained exception handling, such as

```
CATCH (com.sybase.iiop.net.CorbaMarshalException cme)
        // exception handling code
CATCH (com.sybase.iiop.net.CorbaNetworkException cne)
        // exception handling code
```

you cannot currently refine your processing in this way due to a compilation error that occurs when adding the *com.sybase.iiop.net* assembly to your target. This deficiency should be addressed in a PowerBuilder patch shortly after the release of PowerBuilder 11.2. In the meantime, you can discern the actual exception type by using the following code:

```
CATCH (System.Exception ex)
        CHOOSE CASE ex.GetType().ToString()
CASE ("com.sybase.iiop.net.RemoteSystemException")
        // handle remote exception, e.g., incorrect password
CASE ("System.Net.Sockets.SocketException")
        // handle socket error, e.g., server not running
...
        END CHOOSE
```

SECURE CLIENT CONNECTIONS

Our last topic involves secure connections to EAServer via the SSL and IIOP protocols. SSL involves the use of Public Key Encryption and digital certificates to support two types of authentication in EAServer:

- server authentication to ensure that the server being accessed is the one it purports to be, and
- mutual (client and server) authentication that requires that the client also present trusted and valid credentials (an SSL certificate) to the application server.

EAServer implements its security infrastructure using the Java Secure Socket Extension (JSSE) API and exposes its configuration via security profiles. Each EAServer listener can be configured to support a security profile that defines the characteristics of a client-to-server connection in EAServer such as encryption requirements, protocol, authentication type, and hashing methods. The default instance of EAServer includes two security profiles: *default* and *default_mutual*. The default profile is associated with the IIOPS1 listener on port 2001 and supports server authentication using the Sample1 Test ID certificate. The *default_mutual* profile (associated with the IIOPS2 listener on port 2002) supports mutual authentication. It uses the Sample2 Test ID certificate for server authentication and the Sample1 Test ID certificate for client authentication.

WIN32 CLIENT APPLICATIONS

For PowerBuilder Win32 (as well as C/C++) applications, using SSL on a client machine requires the installation of the Client Runtime from the EAServer installation image. This option installs the required runtime libraries and certificate stores on the client machine and configures the JAGUAR_CLIENT_ROOT environment variable appropriately.

When coding a secure Win32 client application, you must specify the minimum quality of protection (QOP) required for the connection and the PIN for the PKCS #11 token prior to the connection request. When the client application negotiates with the server over the specified secure IIOP port, the server accepts or denies the connection based on whether the QOP can be satisfied. In a mutual authentication scenario, the client certificate is also verified before the connection is allowed.

Both the Connection and the JaguarORB objects have a property named *options* through which the QOP, PIN, and client certificate label (for mutual authentication only) can be specified. The specific options and their purposes are summarized below:

Option	Description
ORBqop	QOP (e.g., sybpks_domestic)
ORBpin	PKCS #11 token (default is 'sybase')
ORBCertificateLabel	Client certificate label (e.g., Sample1 Test ID)

To configure these values, set the *options* property to a concatenated string of the desired items, for example:

```
c.options = "ORBqop='sybpks_domestic',ORBpin='sybase'"
```

Alternatively, you can use the SSLCallback object in PowerBuilder to supply this information programmatically, perhaps by prompting the end user. SSLCallback exposes a number of methods like *GetPin* and *TrustVerify* that are triggered when the SSL negotiation requires a response from the client to continue. By creating a new user object extending SSLCallback, you can provide your own logic and user interface for obtaining this information from the client on-demand.

Whether the security attributes are provided explicitly or via the SSLCallback mechanism, the remaining code to connect to the server, instantiate the proxies, and execute component methods remains the same as when accessing a non-secure connection—with the exception, of course, that an IIOPS versus IIOP port must be specified in the server URL.

.NET TARGETS

For .NET clients, installation of the EAServer Client Runtime is not required. The only runtime components needed are the two .NET assemblies discussed at the beginning of the article. The Windows operating system manages the certificate files, configured via the Microsoft Management Console (MMC). Generally, the files will be made available by the EAServer administrator or obtained from the certificate authority that issued the credentials.

For purposes of this article and the associated sample application on CodeXchange, we'll use the sample certificates already installed in EAServer; however, since the standalone files are not distributed with the product, the certificates need to be extracted from the EAServer certificate store. The JSSE keystore is located in the *Repository/Security/keystore.jks* file of the EAServer installation. You can use the Java *keytool* utility, which is part of the Java Runtime Environment (JRE), to examine and extract certificates from that repository. However, *keytool* does not provide for exporting private keys from the certificate store, so you cannot use it to obtain the client certificate file needed for mutual authentication. There are alternative tools available such as the open source Portecle application (**http://portecle.sourceforge.net/**) to do so.

Using Portecle, you can browse to the *keystore.jks* file, provide a password of 'changeit' when prompted, and view the certificates in that store. As shown in Figure 3, you can use the Export option on the context menu to save a specific certificate as a standalone file.

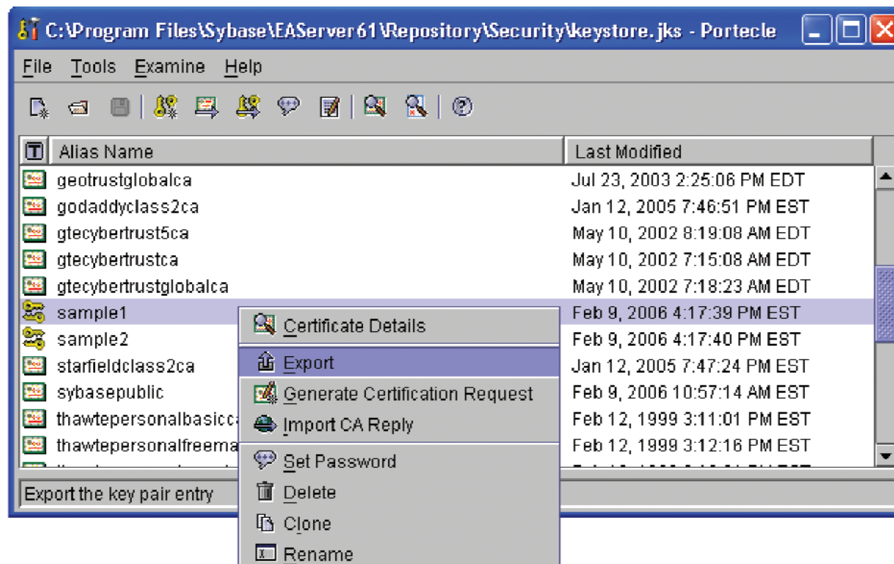

Figure 3

For the *sample1* certificate:

- Select the option to save Private Key and Certificates (see Figure 4). The private key is needed since we'll use sample1 as the client certificate for mutual authentication.
- When prompted for a Key Pair Entry Password, enter 'changeit' (the keystore password).
- Optionally enter a password to associate with the exported PCKS #12 file (see Figure 5). The code examples later in this article presume that you've set a password of "pbdj."

For the *sample2* certificate, simply select Certificate Chain as the Export Type; there are no further prompts for passwords. Once you have saved the two certificate files, transfer them to the client machine.

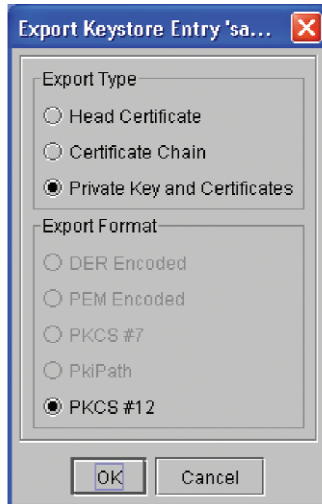

Figure 4

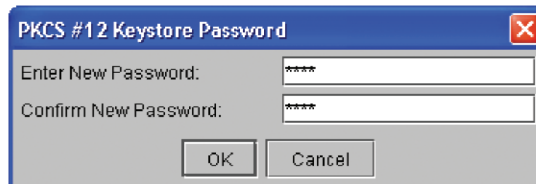

Figure 5

To install the certificates on the client machine, select Start>Run from the Windows task bar and enter "mmc" as the application to open. This will launch the default MMC console to which you'll need to add the Certificates snap-in via the File>Add/Remove Snap-in menu item. When adding the snap-in, you'll be asked for which account the snap-in should manage certificates; select the Computer account for the local computer to make the certificates available to all users of the client machine.

Once the snap-in has been configured, you should see something similar to Figure 6 (which shows the Sample1 Test ID has already been added). To add additional certificates, such as Sample2 Test ID:

- Select the appropriate certificate folder (the Personal folder in this case),
- Right-mouse click, and select All Tasks>Import from the context menu, and
- Complete the Certificate Import Wizard, which prompts for the certificate file name and a password (for PKCS #12 certificates only).

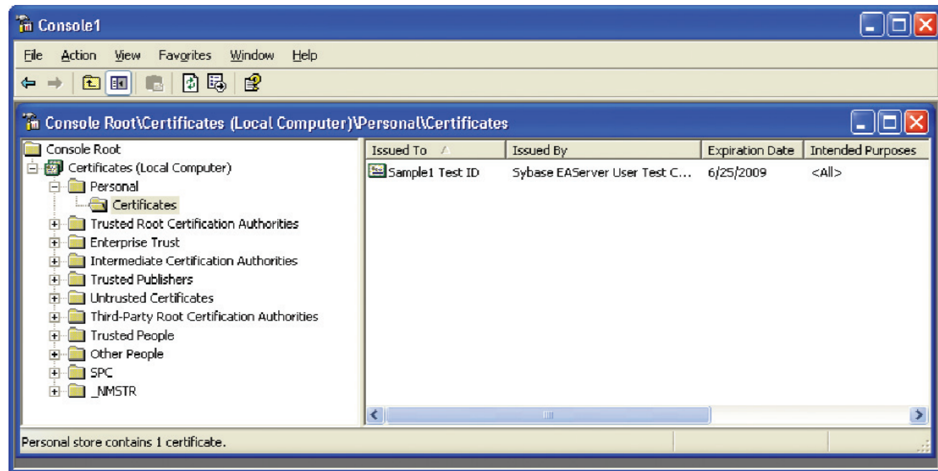

Figure 6

Assuming there were no errors, and the certificates successfully imported, the security configuration for your .NET applications is nearly complete. If you are accessing EAServer from within a Web Forms application or .NET Web Service and require mutual authentication, an additional configuration step is required to grant the ASP .NET process permission to access the client certificate's private key.

To perform this final configuration step, download and install the Windows HTTP Services Certificate Configuration Tool from the Microsoft Web site (**http://www.microsoft.com/downloads/details.aspx?familyid=C42E27AC-3409-40E9-8667-C748E422833F**). By default, this command line tool (*winhttpcertcfg.exe*) will be placed in the *\Program Files\Windows Resource Kits\Tools* directory. The utility has a number of command line options used to control access to client certificates by user account. The specific parameters relevant for PowerBuilder Web Forms and Web Services are summarized in the table below.

Option	Explanation
-g	indicate access should be granted; other options are available to revoke access (-r) or list accounts with access (-l)
-a <*account*>	account to be granted access. Set <*account*> to: IIS5 (Windows XP): "ASPNET" IIS6 (Windows 2003): "NetworkService" IIS7 (Windows Vista): "NetworkService"
-s <*subject name*>	certificate for which access is to be granted. Set <*subject name*> to the certificate's "Issued To" field shown in MMC.
-c <*store*>	certificate store. Set <*store*> to *LOCAL_MACHINE\My*, which refers to the Personal folder in MMC where the certificates were installed.

On a Windows XP machine, for instance, you would issue the following command to grant the ASP .NET process access to the Sample1 Test ID client certificate for mutual authentication to EAServer:

```
winhttpcertcfg -g -c LOCAL_MACHINE\MY -s "Sample1 Test ID" -a "ASPNET"
```

In terms of the .NET client application logic, the code to access an EAServer instance configured for *server authentication* is no different from the code to access a non-secure connection (with the exception of the port designation of course). Contrast this to Win32 client applications, where a secure connection additionally requires configuration of the ORBqop and ORBpin options or implementation of an SSLCallback mechanism, as well as deployment of the EAServer Client Runtime.

However, when using *mutual* authentication with .NET client applications, there are two items that must be specified in either the Connection or JaguarORB object's *options* property:

Option	Description
ORBclientCertificateFile	the path and file name of the client PKCS #12 certificate on the local machine
ORBclientCertificatePassword	the (optional) password protecting the public/private key pair contained in the certificate

These values are analogous to the ORBCertificateLabel setting required for mutual authentication in Win32 client applications and would be set as shown in the following line of code (N.B., these option names are case-sensitive):

```
orb.options =
"ORBclientCertificateFile='c:\easclient\sample1.p12'," +
        "ORBclientCertificatePassword='pbdj'"
```

SUMMARY

So, with the PowerBuilder 11.2 and EAServer 6.1 releases, these products propel their reputation of synergy and openness into the .NET world. Existing PowerBuilder applications that access J2EE resources in EAServer can now be deployed as .NET targets generally with few or even no changes in the PowerScript code. In fact, the two most significant differences affecting EAServer client applications in a .NET environment are the streamlined deployment requirements and the procedures to configure certificate stores on the Windows operating system.

Listing 1

```
Connection c
SimpleMath mathComp
c = CREATE Connection
c.driver = "jaguar"
c.location = "iiop://localhost:2000"
c.userid = "admin@system"
c.password = "sybase1"
IF c.ConnectToServer()= 0 THEN
        IF c.CreateInstance(mathComp, "PBDJ/SimpleMath") = 0 THEN
        st_sum.text = "x + y = " + String(mathComp.add(argX, argY))
        st_difference.text = "x – y = " +
String(mathComp.subtract(argX, argY))
        st_product.text = "x * y = " +
String(mathComp.multiply(argX, argY))
        st_quotient.text = "x / y = " + String(mathComp.divide(argX, argY))
        END IF
END IF
```

Listing 2

```
JaguarORB orb
CORBAObject objRef
Manager mgrEAS
Session sessEAS
SimpleMath mathComp

// create ORB
orb = CREATE JaguarORB
orb.init("")

// get reference to server
IF orb.string_to_object("iiop://localhost:2000", objRef) = 0 THEN

	// narrow to Manager interface
	IF objRef._is_a("SessionManager/Manager") THEN
		IF objRef._narrow(mgrEAS, "SessionManager/Manager") = 0 THEN

			// create a session
			sessEAS = mgrEAS.createSession("admin@system", "sybase1")

			// instantiate the component within the session
			objRef = sessEAS.create("PBDJ/SimpleMath")
			IF objRef._narrow(mathComp, "PBDJ/SimpleMath") = 0 THEN

				st_sum.text = "x + y = " + String(mathComp.add(argX, argY))
				st_difference.text = "x – y = " + String(mathComp.subtract(argX, argY))
				st_product.text = "x * y = " + String(mathComp.multiply(argX, argY))
				st_quotient.text = "x / y = " + String(mathComp.divide(argX, argY))
			END IF
		END IF
	END IF
END IF
```


SYBASE, INC. WORLDWIDE HEADQUARTERS, ONE SYBASE DRIVE, DUBLIN, CA 94568 USA 1 800 8 SYBASE Copyright © 2008 Sybase, Inc.
All rights reserved. Unpublished rights reserved under U.S. copyright laws. Sybase, the Sybase logo and PowerBuilder are trademarks of Sybase, Inc. or its subsidiaries. All other trademarks are the property of their respective owners. ® indicates registration in the United States. Specifications are subject to change without notice. 04-08

SYBASE®