SYBASE®
An SAP® Company

Programmers Guide

Adaptive Server® Enterprise Database Driver for Perl 15.7 SP100

DOCUMENT ID: DC01694-01-1570100-01
LAST REVISED: May 2013

Sybase, Inc., One Sybase Drive, Dublin, CA 94568.

Contents

Adaptive Server Enterprise Database Driver for Perl

The Adaptive Server® Enterprise database driver for the Perl scripting language allows Perl developers to connect to an Adaptive Server database and query or change information using a Perl script.

Perl Driver Module

DBD::SybaseASE is the Adaptive Server database driver for the Perl scripting language.

The DBD::SybaseASE database driver for the Perl scripting language is called through the generic Perl DBI interface and translates Perl DBI API calls into a form that is understood by Adaptive Server through the Open Client SDK using CT-Library.

Using DBI and DBD::SybaseASE, your Perl scripts can directly access Adaptive Server Enterprise database servers.

The generic Perl DBI API specification defines a set of methods that provide a database interface that is independent of the actual database being used.

The Perl DBI programmable API calls are documented at *http://search.cpan.org/~timb/DBI-1.616/DBI.pm* .

Note: The DBD::SybaseASE driver cannot function without the DBI. The DBI contains all user-visible APIs.

Required Components

Access to an Adaptive Server database using the Perl programming language requires the following components:

- Perl installation – generic core database API that is database-vendor-agnostic.
- DBD::SybaseASE – database driver for the Perl scripting language.
- CT-Library – (CT-Lib API) is part of the Open Client suite. CT-Library sends commands to Adaptive Server and processes results.
- Adaptive Server Enterprise
- Perl

Version Requirements

For information about platform support, see the *Software Developers Kit and Open Server Installation Guide* for your platform.

- Adaptive Server Enterprise – version 15.7 or later.

- Open Client and Open Server – version 15.7 or later.
- Perl – version 5.14.0 or 5.14.1.
- DBD::SybaseASE driver – no specific version requirements.
- CT-Library – (CT-Lib API) version 15.7.
- Perl DBI – version 1.616.
  The Sybase® installer does not check for a Perl installation or if the driver dependencies are installed on the target system.

Note: The build mode of the Perl driver released for your platform also dictates the build mode of your Perl installation and the DBI. As an example, for Linux the driver is released in 64-bit mode with threading enabled. This means Perl must be configured in full 64-bit mode with threading enabled. The build mode requirement also applies to the DBI interface.

Installing and Configuring the Driver for Perl

The database driver for Perl is a component you can install through the Sybase Installer.

The database driver for Perl is an optional installation component when you choose **Custom** as the installation type. The driver is installed by default if the installation type you choose is **Typical** or **Full**. For installation and configuration instructions, see the *Software Developers Kit and Open Server Installation Guide* for your platform.

Developing Perl Applications

Use the Perl DBI API to develop Perl applications.

Support for DSN Style Connection Properties

The driver uses a DSN mechanism that allows certain attributes to be set at connection time.

The DSN attribute syntax is the same as the Open Source DBD::Sybase driver. Therefore, you need not change Perl scripts or maintain different versions for DBD::Sybase versus DBD::SybaseASE. However, DBD::SybaseASE does not support some attributes that are considered obsolete. See *Currently unsupported DSN syntax*.

SybaseASE Driver Connect Syntax

The dbi:SybaseASE: section obtains the package name of the driver so it can be loaded in the following syntax.

```
DBI->connect("dbi:SybaseASE:attr=value;attr=value", $user_id,
$password, %attrib);
```

When the DSN is passed into the driver, the system removes this part and the remaining string holds the key and value pairs to be dissected.

Note: The *$user_id* and *$password* credentials are separate API arguments; they are not part of the DSN string.

The **%attrib** argument is an optional, comma-separated chain of key-value pairs that set options at connection time. They are passed into the driver and handled during a **connect()** call. For example:

```
DBI->connect("dbi:SybaseASE:server=mumbles; user, password,
PrintError => 1, AutoCommit = 0);
```

Attributes and Methods

The following attributes are currently supported when connecting to a server.

Attributes	Description
server	Specifies the server to which you are connecting. The driver currently assumes this option is set. If server is not specified, use the ENV{"DSQUERY"} mechanism to obtain a server name.
database	Specifies which database within the server is the target database at connect time. If no database is specified, the master database is used.
hostname	Specifies, in the value section, the host name that is stored in the `sysprocesses` table for this process. If no hostname is specified, the host on which the Perl application executes is used.
language	Specifies the locale to be used on this connection. If no language is specified, the internal default locale named CS_LC_ALL is used.
charset	Specifies the charset to be used on this connection. If no charset is specified, the internal default that is, **utf8**, is used.
host; port	Specifies the combination of host and port to use instead of relying on the interfaces file entries. **Note:** In the Perl DSN syntax, host and port are separate options. An alternative DSN form similar to the following is not currently supported: `host:port=mumbles:1234` When the host and port DSN options are provided with the intent of not using the interface file, the host and port must suffice to connect. If the DSN attribute “server=” is also provided with the host and port combination, the connection fails. Therefore, the usage of either host and port must be used to establish a connection or server alone must be used. The two DSN attributes (server versus host/port) are mutually exclusive.

Attributes	Description
timeout	Specifies the connection timeout value. Set to 0 or a negative value for no timeout.
loginTimeout	Specifies the login timeout value, in seconds. The default value is 60 seconds. Set **loginTimeout=*value in seconds*** to enable this attribute.
tds_keepalive	Specifies the KEEP_ALIVE attribute on the connection. Set **tds_keepalive=*1*** to enable this attribute.
packetSize	Specifies the TDS packet size for the connection. By default, the lower bound, which is set in the driver, is 2048. The maximum value is determined by the server, and is not set in the driver.
maxConnect	Increases or decreases the number of connections allowed. The range of values is 1 – 128; the default is 25.
encryptPassword	Specifies whether to use password encryption. Set **encryptPassword=*1*** to enable this attribute.
sslCAFile	Specifies an alternate location for the `trusted.txt` file. Specify an absolute path of up to 256 characters.
scriptName	Specifies the chosen name of the top-level Perl script that drives the application. This name appears in the `sysprocesses` table as the application name. Absence of this value gives a default application name that is obtained from the Perl internal environment. This value can be as many as 256 characters. **Note:** The application name fed into the SybaseASE Driver is either set through the DSN **scriptName** option or is derived from the Perl internal environment.
interfaces	Specifies an alternate location to the Sybase interfaces file. Same constraints apply to the **sslCAFile** and **scriptName** options.

You can repeat attribute values as long as they are recognized by the driver. Illegal attributes cause the **DBI->connect()** call to fail.

Note: The attribute names follow the Open Source Sybase Perl driver.

DSN-specific example:

```
$dbh = DBI->connect("dbi:SybaseASE:server=mumbles", $user, $passwd);
```

Alternatively, use the DSQUERY environment variable:

```
my $srv = $ENV{"DSQUERY"};
$dbh = DBI->connect("dbi:SybaseASE:server=$srv",  $user, $passwd);
$dbh = DBI-
>connect("dbi:SybaseASE:host=tzedek.sybase.com;port=8100", $user,
$passwd);
```

```
$dbh = DBI->connect("dbi:SybaseASE:maxConnect=100", $user, $passwd);
$dbh = DBI->connect("dbi:SybaseASE:database=sybsystemprocs", $user,
$passwd);
$dbh = DBI->connect("dbi:SybaseASE:charset=iso_1", $user, $passwd);
$dbh = DBI->connect("dbi:SybaseASE:language=us_english", $user,
$passwd);
$dbh = DBI->connect("dbi:SybaseASE:packetSize=8192", $user,
$passwd);
$dbh = DBI->connect("dbi:SybaseASE:interfaces=/opt/sybase/
interfaces", $user, $passwd);
$dbh = DBI->connect("dbi:SybaseASE:loginTimeout=240", $user,
$passwd);
$dbh = DBI->connect("dbi:SybaseASE:timeout=240", $user, $passwd);
$dbh = DBI->connect("dbi:Sybase:scriptName=myScript", $user,
$password);
$dbh = DBI->connect("dbi:SybaseASE:hostname=pedigree", $user,
$password);
$dbh = DBI->connect("dbi:SybaseASE:encryptPassword=1", $user,
$password);
$dbh = DBI>connect("dbi:SybaseASE:sslCAFile=/usr/local/sybase/
trusted.txt", $user, $password,
AutoCommit => 1);
```

DSN-specific example combination:

```
$dbh = DBI->connect("dbi:SybaseASE:server=mumbles,
database=tempdb;packetSize=8192;
language=us_english;charset=iso_1;encryptPassword=1", $user, $pwd,
AutoCommit=>1, PrintError => 0);
```

Currently Unsupported DSN Syntax

The following DSN syntax are not supported currently:

- **tdsLevel**
- **kerberos**; for example:

  ```
  $dbh = DBI->connect("dbi:SybaseASE:kerberos=$serverprincipal",
  '', '');
  ```

- **bulkLogin**; for example:

  ```
  $dbh = DBI->connect("dbi:SybaseASE:bulkLogin=1", $user,
  $password);
  ```

- **serverType**

Currently Supported Database Handle Attributes

The table lists currently supported database handle attributes.

Attribute	Description	Default
dbh->{AutoCommit} = (0|1);	Disables or enables AutoCommit.	0 (off)
dbh->{LongTruncOK} = (0|1);	Disables or enables truncation of `text` and `image` types.	0

Attribute	Description	Default
dbh->{LongReadLen}=(int);	Sets the default read chunk size for `text` and `image` data. For example: **dbh->{LongReadLen} = 64000**.	32767
dbh->{syb_show_sql} =(0|1);	If set, the current statement is included in the error string returned by the **$dbh->errstr** mechanism.	0
dbh->{syb_show_eed} = (0|1);	If set, the extended error information is included in the error string returned by **$dbh->errstr**.	0
dbh->{syb_chained_txn} = (0|1);	If set, CHAINED transactions are used when AutoCommit is off. Use this attribute only during the **connect()** call: `$dbh = DBI->connect("dbi:SybaseASE:", $user, $pwd, {syb_chained_txn => 1});` Using **syb_chained_txn** at any time with AutoCommit turned off forces a commit on the current handle. When set to 0, an explicit **BEGIN TRAN** is issued as needed.	0
dbh->{syb_use_bin_0x} = (0|1);	If set, `BINARY` and `VARBINARY` values are prefixed with '0x' in the result string.	0
dbh->{syb_binary_images} = (0|1);	If set, `image` data is returned in raw binary format. Otherwise, `image` data is converted into a hexadecimal string.	0
dbh->{syb_quoted_identifier} =(0|1);	Allows identifiers that conflict with Sybase reserved words if they are quoted using "identifier."	0
dbh->{syb_rowcount}=(int);	If set to a nonzero value, the number of rows returned by a **SELECT**, or affected by an **UPDATE** or **DELETE** statement are limited to the *rowcount* value. Setting it back to 0 clears the limit.	0
dbh->{syb_flush_finish} = (0|1);	If set, the driver drains any results remaining for the current command by actually fetching them. This can be used instead of a **ct_cancel()** command issued by the driver.	0

Attribute	Description	Default
dbh->{syb_date_fmt} = datefmt string	This private method sets the default date conversion and display formats. See *Default Date Conversion and Display Format*.	
dbh->{syb_err_handler}	Perl subroutine that can be created to execute an error handler or report before the regular error handling takes place. Useful for certain classes of warnings. See *Error Handling*.	0 (not present)
dbh->{syb_failed_db_fatal} = (0|1)	If the DSN has a **database=*mumbles*** attribute/value pair and this database does not exist at connection time, the **DBI->connect()** call fails.	0
dbh->{syb_no_child_con} =(0|1);	If set, the driver disallows multiple active statement handles on the **dbh**. In this case, a statement can be prepared but must be executed to completion before another statement prepare is attempted.	0
dbh->{syb_cancel_request_on_error}=(0|1);	If set, when a multistatement set is executed and one statement fails, **sth->execute()** fails.	1 (on)
dbh->{syb_bind_empty_string_as_null}= (0|1);	If set, a NULLABLE column attribute returns an empty string (one space) to represent the NULL character.	0
dbh->{syb_disconnect_in_child} = (0|1);	Handles closed connections across a fork. The DBI causes connections to be closed if a child dies.	0
dbh->{syb_enable_utf8} = (0|1);	If set, `UNICHAR`, `UNIVARCHAR`, and `UNITEXT` are converted to **utf8**.	0
sth->syb_more_results} = (0|1);	*See Multiple Result Sets*.	
sth->{syb_result_type} = (0|1);	If set, returns the numeric result number instead of the symbolic CS_ version.	0
sth->{syb_no_bind_blob} = (0|1);	If set, `image` or `text` columns are not returned upon **sth->{fetch}** or other variations. See *Text and Image Data Handling*.	0

Attribute	Description	Default
sth->{syb_do_proc_status} = (0|1);	Forces **$sth->execute()** to fetch the return status of a stored procedure executed in the SQL stream. If the return status is nonzero, **$sth->execute()** returns `undef` (that is, it fails). Setting this attribute does not affect existing statement handles. However, it affects those statement handles that are created after setting it. To revert behavior of an existing **$sth** handle, execute: **$sth->{syb_do_proc_status} = 0**;	0

See also

- *Error Handling* on page 18
- *Text and Image Data Handling* on page 17
- *Default Date Conversion and Display Format* on page 16
- *Multiple Statements Usage* on page 9

Unsupported Database Handle Options

The following database handle options are not supported.

- **dbh->{syb_dynamic_supported}**
- **dbh->{syb_ocs_version}**
- **dbh->{syb_server_version}**
- **dbh->{syb_server_version_string}**
- **dbh->{syb_has_blk}**

Note: Perl scripts attempting to use these options generate an error.

Perl Supported Datatypes

The Perl driver currently supports string, numeric, and date and time datatypes.

String types	Numeric types	Date and time data-types
char	integer	datetime
varchar	smallint	date
binary	tinyint	time
varbinary	money	bigtime
text	smallmoney	bigdatetime
image	float	
unichar	real	
univarchar	double	
	numeric	
	decimal	
	bit	
	bigint	

Note: Perl returns numeric and decimal types as strings. Other datatypes are returned in their respective formats.

The default time/date format used by the Sybase ASE driver is the short format, for example, Aug 7 2011 03:05PM.

This format is based on the C (default) locale. See *Default Date Conversion and Display Format* for other date and time formats supported.

See also

- *Default Date Conversion and Display Format* on page 16

Multiple Statements Usage

Adaptive Server can handle multistatement SQL in a single batch.

For example:

```
my $sth = $dbh->prepare("
    insert into publishers (col1, col2, col3) values (10, 12, 14)
    insert into publishers (col1, col2, col3) values (1, 2, 4)
    insert into publishers (col1, col2, col3) values (11, 13, 15)
```

```
    ");
my $rc = $sth->execute();
```

If any of these statements fail, **sth->execute()** returns `undef`. If **AutoCommit** is on, statements that complete successfully may have inserted data in the table, which may not be the result you expect or want.

Multiple Result Sets

The Perl driver allows you to prepare multiple statements with one call and execute them with another single call. For example, executing a stored procedure that contains multiple selects returns multiple result sets.

Results of multiple statements prepared with one call are returned to the client as a single stream of data. Each distinct set of results is treated as a normal single result set, which means that the statement handle's **fetch()** method returns `undef` at the end of each set.

The CT-Lib API **ct_fetch()** returns `CS_END_RESULTS` that the driver converts to `undef` after the last rows have been retrieved.

The driver allows the application to obtain the result type by checking **sth->{syb_result_type}**. You can then use the **sth->{syb_more_results}** statement handle attribute to determine if there are additional result sets still to be returned. The (numerical) value returned by **sth->{syb_results_type}** is one of:

- `CS_MSG_RESULT`
- `CS_PARAM_RESULT`
- `CS_STATUS_RESULT`
- `CS_COMPUTE_RESULT`
- `CS_ROW_RESULT`

Example for multiple result sets:

```
do {
    while($a = $sth->fetch) {
    ..for example, display data..
    }
} while($sth->{syb_more_results});
```

Sybase recommends that you use this if you expect multiple result sets.

Note: The Perl driver currently does not support cursors using the **ct_cursor()** API. Therefore, the driver does not report CS_CURSOR_RESULT.

Multiple Active Statements on a DatabaseHandle (dbh)

There can be multiple active statements on a single database handle by opening a new connection in the **$dbh->prepare()** method if there is already an active statement handle on this **$dbh**.

The **dbh->{syb_no_child_con}** attribute controls whether this feature is on or off. By default, DatabaseHandle is off, which indicates that multiple statement handles are supported. If it is on, multiple statements on the same database handle are disabled.

Note: If AutoCommit is off, multiple statement handles on a single **$dbh** are unsupported. This avoids deadlock problems that may arise. Also, using multiple statement handles simultaneously provides no transactional integrity, as different physical connections are used.

Supported Character Lengths

Supported character lengths for different types of identifiers.

The names of Sybase identifiers, such as tables and columns, can exceed 255 characters in length.

Logins, application names, and password lengths that are subject to TDS protocol limits cannot exceed 30 characters.

Configuring Locale and Charsets

You can configure the Perl driver of CT-Library locale and charset using the DSN attributes **charset** and **language**.

The driver's default character set is *UTF8* and the default locale is *CS_LC_ALL*.

Dynamic SQL Support, Placeholders, and Bind Parameters

The Perl driver supports dynamic SQL, including parameter usage.

For example:

```
$sth = $dbh->prepare("select * from employee where empno = ?");

# Retrieve rows from employee where empno = 1024:
$sth->execute(1024);
while($data = $sth->fetch) {
    print "@$data\n";
}
# Now get rows where empno = 2000:
$sth->execute(2000);
while($data = $sth->fetch) {
    print "@$data\n";
}
```

Note: The Perl driver supports the '?' style parameter, but not ':1' placeholder types. You cannot use placeholders to bind a `text` or `image` datatype.

DBD::SybaseASE uses the Open Client **ct_dynamic()** family of APIs for the **prepare()** method. See the *Sybase Open Client C Programmers guide* for information about "?" style placeholder constraints and general dynamic SQL usage.

This is another example showing dynamic SQL support:

```
my $rc;
my $dbh;
my $sth;


# call do() method to execute a SQL statement.
#
$rc = $dbh->do("create table tt(string1 varchar(20), date datetime,
    val1 float, val2 numeric(7,2))");


$sth = $dbh->prepare("insert tt values(?, ?, ?, ?)");
$rc = $sth->execute("test12", "Jan 3 2012", 123.4, 222.33);


# alternate way, call bind_param() then execute without values in the
# execute statement.
$rc = $sth->bind_param(1, "another test");
$rc = $sth->bind_param(2, "Jan 25 2012");
$rc = $sth->bind_param(3, 444512.4);
$rc = $sth->bind_param(4, 2);
$rc = $sth->execute();

# and another execute, with args.....
$rc = $sth->execute("test", "Feb 30 2012", 123.4, 222.3334);
```

Note: The last statement throws an extended error information (EED) as the date is invalid. In the Perl script, set **dbh->{syb_show_eed} = *1*** before execution to write the Adaptive Server error message in the **dbh->errstr**.

Another example that illustrates the "?" style placeholder:

```
$sth = $dbh->prepare("select * from tt where date > ? and val1 > ?");
$rc = $sth->execute('Jan 1 2012', 120);

# go home....
$dbh->disconnect;
exit(0);
```

Stored Procedure Support for Placeholders

The Adaptive Server Enterprise database driver for Perl supports stored procedures that include both input and output parameters.

Stored procedures are handled in the same way as any other Transact-SQL statement. However, Sybase stored procedures return an extra result set that includes the return status that corresponds to the return statement in the stored procedure code. This extra result set, named CS_STATUS_RESULT with numeric value 4043, is a single row and is always returned last.

The driver can process the stored procedure using a special attribute, **$sth->{syb_do_proc_status}**. If this attribute is set, the driver processes the extra result set, and places the return status value in **$sth->{syb_proc_status}**. An error is generated if the result set is a value other than 0.

Examples

```
$sth = $dbh->prepare("exec my_proc \@p1 = ?, \@p2 = ?");
    $sth->execute('one', 'two');
```

This example illustrates the use of positional parameters:

```
$sth = $dbh->prepare("exec my_proc ?, ?");
  $sth->execute('one', 'two');
```

You cannot mix positional and named parameters in the same prepare statement; for example, this statement fails on the first parameter:

```
$sth = $dbh->prepare("exec my_proc \@p1 = 1, \@p2 = ?");
```

If the stored procedure returns data using output parameters, you must declare them first:

```
$sth = $dbh->prepare(qq[declare @name varchar(50) exec getname abcd,
@name output]);
```

You cannot call stored procedures with bound parameters, as in:

```
$sth = $dbh->prepare("exec my_proc ?");
  $sth->execute('foo');
```

This works as follows:

```
$sth = $dbh->prepare("exec my_proc 'foo'");
  $sth->execute('foo');
```

Because stored procedures almost always return more than one result set, use a loop until syb_more_results is 0:

```
do {
    while($data = $sth->fetch) {
        do something useful...
    }
 } while($sth->{syb_more_results});
```

Parameter examples

```
declare @id_value int, @id_name char(10)
    exec my_proc @name = 'a_string', @number = 1234,
        @id = @id_value OUTPUT, @out_name = @id_name OUTPUT
```

If your stored procedure returns only OUTPUT parameters, you can use:

```
$sth = $dbh->prepare('select * .......');
    $sth->execute();
@results = $sth->syb_output_params();  #  this method is available in
SybaseASE.pm
```

This returns an array for all the OUTPUT parameters in the procedure call and ignores any other results. The array is undefined if there are no OUTPUT parameters or if the stored procedure fails.

Generic examples

```
$sth = $dbh->prepare("declare \@id_value int, \@id_name
        OUTPUT, @out_name = @id_name OUTPUT");
```

```
    $sth->execute();
    {
        while($d = $sth->fetch) {
            # 4042 is CS_PARAMS_RESULT
        if ($sth->{syb_result_type} == 4042) {
            $id_value  = $d->[0];
            $id_name  = $d->[1];
        }
    }
    redo if $sth->{syb_more_results};
}
```

The OUTPUT parameters are returned as a single row in a special result set.

Parameter Types

The driver does not attempt to determine the correct parameter type for each parameter. The default for all parameters defaults to the ODBC style SQL_CHAR value, unless you use **bind_param()** with a type value set to a supported bind type.

The driver supports these ODBC style bind types:

- SQL_CHAR
- SQL_VARCHAR
- SQL_VARBINARY
- SQL_LONGVARCHAR
- SQL_LONGVARBINARY
- SQL_BINARY
- SQL_DATETIME
- SQL_DATE
- SQL_TIME
- SQL_TIMESTAMP
- SQL_BIT
- SQL_TINYINT
- SQL_SMALLINT
- SQL_INTEGER
- SQL_REAL
- SQL_FLOAT
- SQL_DECIMAL
- SQL_NUMERIC
- SQL_BIGINT
- SQL_WCHAR
- SQL_WLONGVARCHAR

The ODBC types are mapped in the driver to equivalent Adaptive Server datatypes. See the *Adaptive Server Enterprise ODBC Driver by Sybase User Guide 15.7*.

Execute the stored procedure, **sp_datatype_info** to get a full list of supported types for the particular Adaptive Server. For example:

```
$sth = $dbh->prepare("exec my_proc \@p1 = ?, \@p2 = ?");
    $sth->bind_param(1, 'one', SQL_CHAR);
    $sth->bind_param(2, 2.34, SQL_FLOAT);
    $sth->execute;
    ....
    $sth->execute('two', 3.456);
    etc...
```

Note: Once you have set a column type for a parameter, you cannot change it unless you deallocate and retry the statement handle. When binding SQL_NUMERIC or SQL_DECIMAL data, you may get fatal conversion errors if the scale or the precision exceeds the size of the target parameter definition.

For example, consider this stored procedure definition:

```
declare proc my_proc @p1 numeric(5,2) as...
    $sth = $dbh->prepare("exec my_proc \@p1 = ?");
    $sth->bind_param(1, 3.456, SQL_NUMERIC);
```

which generates this error:

```
DBD::SybaseASE::st execute failed: Server message number=241
severity=16 state=2 line=0 procedure=my_proc text=Scale error
during implicit conversion of NUMERIC value '3.456' to a
NUMERIC field.
```

Set the **arithabort** option as follows to ignore these errors:

$dbh->do("set arithabort off");

See the Adaptive Server reference documentation.

Supported Private Driver Methods

dbh->syb_isdead() returns a true or false representation of the state of the connection. A false return value may indicate a specific class or errors on the connection, or that the connection has failed.

$sth->syb_describe() returns an array that includes the description of each output column of the current result set. Each element of the array is a reference to a hash that describes the column.

You can set the description fields such as NAME, TYPE, SYBTYPE, SYBMAXLENGTH, MAXLENGTH, SCALE, PRECISION, and STATUS, as shown in this example:

```
$sth = $dbh->prepare("select name, uid from sysusers");
    $sth->execute;
    my @description = $sth->syb_describe;
    print "$description[0]->{NAME}\n";                # prints name
    print "$description[0]->{MAXLENGTH}\n";    # prints 30
    etc, etc.
```

```
    ....
    while(my $row = $sth->fetch) {
    ....
}
```

Note: The STATUS field is a string which can be tested for the following values: CS_CANBENULL, CS_HIDDEN, CS_IDENTITY, CS_KEY, CS_VERSION_KEY, CS_TIMESTAMP and CS_UPDATABLE, CS_UPDATECOL and CS_RETURN.

See the Open Client documentation.

Default Date Conversion and Display Format

You can set your own default date conversion and display format using the **syb_data_fmt()** private method.

Sybase date format depends on the locale settings for the client. The default date format is based on the 'C' locale, for example, Feb 16 2012 12:07PM.

This same default locale supports several additional input formats:

- 2/16/2012 12:07PM
- 2012/02/16 12:07
- 2012-02-16 12:07
- 20120216 12:07

Use **dbh->{syb_date_fmt}** with a string as argument, to change the date input and output format.

Table 1. Supported date/time formats

Date format	Example
LONG	Nov 15 2011 11:30:11:496AM
SHORT	Nov 15 2011 11:30AM
DMY4_YYYY	Nov 15 2011
MDY1_YYYY	11/15/2011
DMY1_YYYY	15/11/2011
DMY2_YYYY	15.11.2011
DMY3_YYYY	15-11-2011
DMY4_YYYY	15 November 2011
HMS	11:30:11
LONGMS	Nov 15 2011 11:30:33.532315PM

The Adaptive Server Enterprise database driver for Perl supports all date and time values supported up to version 15.7.

Text and Image Data Handling

The Adaptive Server Enterprise database driver for Perl supports image and a `text` type for LONG/BLOB data. Each type can as much as 2GB of binary data.

The default size limit for text/image data is 32KB. Use the **LongReadLen** attribute to change this limit, which is set by a call to the **fetch()** API.

You cannot use bind parameters to insert text or image data.

When using regular SQL, image data is normally converted to a hex string, but you can use the **syb_binary_images** handle attribute to change this behavior. As an alternative, you can use a Perl function similar to **$binary = pack("H*", $hex_string);** to perform the conversion.

As the DBI has no API support for handling BLOB style (`text`/`image`) types, the `SybaseASE.pm` file includes a set of functions you can install, and use in application-level Perl code to call the Open Client **ct_get_data()** style calls. The **syb_ct_get_data()** and **syb_ct_send_data()** calls are wrappers to the Open Client functions that transfer `text` and `image` data to and from Adaptive Server.

Example

```
$sth->syb_ct_get_data($col, $dataref, $numbytes);
```

You can use the **syb_ct_get_data()** call to fetch the image/text data in raw format, either in one piece or in chunks. To enable this call, set the **dbh->{syb_no_bind_blob}** statement handle to *1*.

The **syb_ct_get_data()** call takes these arguments: the column number (starting at 1) of the query, a scalar reference, and a byte count. A byte count of 0 reads as many bytes as possible. The image/text column must be last in the select list for this call to work.

The call sequence is:

```
$sth = $dbh->prepare("select id, img from a_table where id = 1");
    $sth->{syb_no_bind_blob} = 1;
    $sth->execute;
    while($d = $sth->fetchrow_arrayref) {
        # The data is in the second column
        $len = $sth->syb_ct_get_data(2, \$img, 0);
}
```

syb_ct_get_data() returns the number of bytes that were fetched, if you are fetching chunks of data, you can use:

```
while(1) {
$len = $sth->syb_ct_get_data(2, $imgchunk, 1024);
... do something with the $imgchunk ...
 last if $len != 1024;
}
```

Other TEXT/IMAGE APIs

The **syb_ct_data_info()** API fetches or updates the CS_IODESC structure for the image/text data item you want to update.

For example:

```
$stat = syb_ct_data_info($action, $column, $attr)
```

- *$action* – CS_SET or CS_GET.
- *$column* – the column number of the active select statement (ignored for a CS_SET operation).
- *$attr* – a hash reference that sets the values in the structure.

You must fist call **syb_ct_data_info()** with CS_GET to fetch the CS_IODESC structure for the image/text data item you want to update. Then update the value of the **total_txtlen** structure element to the length (in bytes) of the image/text data you are going to insert. Set the **log_on_update** to true to enable full logging of the operation.

Calling **syb_ct_data_info()** with a CS_GET fails if the image/text data for which the CS_IODESC is being fetched is NULL. Use standard SQL to update the NULL value to non-NULL value (for example, an empty string) before you retrieve the CS_IODESC entry.

In this example, consider updating the data in the image column where the id column is 1:

1. Find the CS_IODESC data for the data:

   ```
   $sth = $dbh->prepare("select img from imgtable where id = 1");
           $sth->execute;
       while($sth->fetch) {    # don't care about the data!
           $sth->syb_ct_data_info('CS_GET', 1);
       }
   ```

2. Update with the CS_IODESC values:

   ```
   $sth->syb_ct_prepare_send();
   ```

3. Set the size of the new data item to be inserted and make the operation unlogged:

   ```
   $sth->syb_ct_data_info('CS_SET', 1, {total_txtlen
   => length($image), log_on_update => 0});
   ```

4. To transfer the data in a single chunk:

   ```
   $sth->syb_ct_send_data($image, length($image));
   ```

5. To commit the operation:

   ```
   $sth->syb_ct_finish_send();
   ```

Error Handling

All errors from the Adaptive Server database driver for Perl and CT-Lib are propagated into the DBI layer.

Exceptions include errors or warnings that must be reported during driver start-up, when there is no context available yet.

The DBI layer performs basic error reporting when the **PrintError** attribute is enabled. Use DBI trace method to enable tracing on DBI operations to track program- or system-level problems.

Examples of adding more detailed error messages (server messages) are as follows:

- Set **dbh->{syb_show_sql} = *1*** on the active **dbh** to include the current SQL statement in the string returned by **$dbh->errstr**.
- Set **dbh->{syb_show_eed} = *1*** on the active **dbh** to add extended error information (EED) such as duplicate insert failures and invalid date formats to the string returned by **$dbh->errstr**.
- Use the **syb_err_handler** attribute to set an ad hoc error handler callback (that is, a Perl subroutine) that gets called before the normal error handler performs its processing. If this subroutine returns 0, the error is ignored. This is useful for handling **PRINT** statements in Transact-SQL, and **showplan** output and **dbcc** output.
  The subroutine is called with parameters that include the Sybase error number, the severity, the state, the line number in the SQL batch, the server name (if available), the stored procedure name (if available), the message text, the SQL text and the strings "client" or "server" to denote type.

Configuring Security Services

Use the `ocs.cfg` and `libtcl.cfg` files to configure security options.

1. For a connection, use `ocs.cfg` to set directory and security properties.

   Note: In the `ocs.cfg` file, add an entry for the application name so you can set that driver-specific option.

2. Edit `libtcl.cfg` to load security and directory service drivers.
3. To encrypt passwords, use the **encryptPassword** DSN option. For example:

```
DBI-
>connect("dbi:SybaseASE:server=mumbles;encryptPassword
=1", $user, $pwd);
```

Examples

Use sample programs to view the basic usage of stored procedure and retrieve rows from the `pubs2 authors` table.

Example 1

Use the sample program to view the basic usage of stored procedures in Perl.

This program connects to a server, creates two stored procedures, calls prepare, binds, or executes the procedures, prints the results to STDOUT, disconnects, and exits the program.

```
use strict;

use DBI qw(:sql_types);
use DBD::SybaseASE;
```

```
require_version DBI 1.51;

my $uid = "sa";
my $pwd = "";
my $srv = $ENV{"DSQUERY"} || die 'DSQUERY appears not set';
my $dbase = "tempdb";

my $dbh;
my $sth;
my $rc;

my $col1;
my $col2;
my $col3;
my $col4;

# Connect to the target server.
#
$dbh = DBI->connect("dbi:SybaseASE:server=$srv;database=$dbase",
    $uid, $pwd, {PrintError => 1});

# One way to exit if things fail.
#
if(!$dbh) {
    warn "Connection failed, check if your credentials are set
correctly?\n";
    exit(0);
}

# Ignore errors on scale for numeric. There is one marked call below
# that will trigger a scale error in ASE. Current settings suppress
# this.
#
$dbh->do("set arithabort off")
        || die "ASE response not as expected";

# Drop the stored procedures in case they linger in ASE.
#
$dbh->do("if object_id('my_test_proc') != NULL drop proc
my_test_proc")
    || die "Error processing dropping of an object";

$dbh->do("if object_id('my_test_proc_2') != NULL drop proc
my_test_proc_2")
    || die "Error processing dropping of an object";

# Create a stored procedure on the fly for this example. This one
# takes input args and echo's them back.
#
$dbh->do(qq{
create proc my_test_proc \@col_one varchar(25), \@col_two int,
    \@col_three numeric(5,2), \@col_four date
as
    select \@col_one, \@col_two, \@col_three, \@col_four
}) || die "Could not create proc";
```

```
# Create another stored procedure on the fly for this example.
# This one takes dumps the pubs2..authors table. Note that the
# format used for printing is defined such that only four columns
#appear in the output list.
#
$dbh->do(qq{
create proc my_test_proc_2
as
    select * from pubs2..authors
}) || die "Could not create proc_2";

# Call a prepare stmt on the first proc.
#
$sth = $dbh->prepare("exec my_test_proc \@col_one = ?, \@col_two
= ?,
    \@col_three = ?, \@col_four = ?")
        || die "Prepare exec my_test_proc failed";

# Bind values to the columns. If SQL type is not given the default
# is SQL_CHAR. Param 3 gives scale errors if arithabort is disabled.
#
$sth->bind_param(1, "a_string");
$sth->bind_param(2, 2, SQL_INTEGER);
$sth->bind_param(3, 1.5411111, SQL_DECIMAL);
$sth->bind_param(4, "jan 12 2012", SQL_DATETIME);

# Execute the first proc.
#
$rc = $sth->execute || die "Could not execute my_test_proc";

# Print the bound args
#
dump_info($sth);

# Execute again, using different params.
#
$rc = $sth->execute("one_string", 25, 333.2, "jan 1 2012")
    || die "Could not execute my_test_proc";

dump_info($sth);

# Enable retrieving the proc status.
$sth->{syb_do_proc_status} = 1;

$rc = $sth->execute(undef, 0, 3.12345, "jan 2 2012")
    || die "Could not execute my_test_proc";
dump_info($sth);

$rc = $sth->execute("raisin", 1, 1.78, "jan 3 2012")
    || die "Could not execute my_test_proc";
dump_info($sth);

$rc = $sth->execute(undef, 0, 3.2233, "jan 4 2012")
    || die "Could not execute my_test_proc";
dump_info($sth);
```

```
$rc = $sth->execute(undef, 0, 3.2234, "jan 5 2012")
    || die "Could not execute my_test_proc";
dump_info($sth);

$rc = $sth->execute("raisin_2", 1, 3.2235, "jan 6 2012")
    || die "Could not execute my_test_proc";
dump_info($sth);

$rc = $sth->execute(undef, 0, 3.2236, "jan 7 2012")
    || die "Could not execute my_test_proc";
dump_info($sth);

# End of part one, generate blank line.
#
print "\n";

# Undef the handles (not really needed but...).
#
undef $sth;
undef $rc;

# Prepare the second stored proc.
#
$sth = $dbh->prepare("exec my_test_proc_2")
        || die "Prepare exec my_test_proc_2 failed";

# Execute and print
#
$rc = $sth->execute || die "Could not execute my_test_proc_2";
dump_info($sth);

#
# An example of a display/print function.
#
sub dump_info {
    my $sth = shift;
    my @display;

    do {
    while(@display = $sth->fetchrow) {
     foreach (@display) {
        $_ = '' unless defined $_;
    }
     $col1 = $display[0];
     $col2 = $display[1];
     $col3 = $display[2];
     $col4 = $display[3];

    # Proc status is suppressed, assume proc
    # execution was always successful. Enable
    # by changing the write statement.
    #
    #write;
    write unless $col1 eq 0;
    }
```

```
} while($sth->{syb_more_results});
}

#
# The FORMAT template for this example.
#
format STDOUT_TOP =

Column1             Column2             Column3             Column4
------              ------              ------              ------
.

# Treat all data as left-justified strings
#
format STDOUT =
@<<<<<<<<<<<<   @<<<<<<<<<<<< @<<<<<<<<<<<<
@<<<<<<<<<<<<
$col1, $col2, $col3, $col4
.

# The End.....
#
$dbh->do("drop proc my_test_proc");
$dbh->do("drop proc my_test_proc_2");
$dbh->disconnect;
```

Example 2

Use the sample program to retrieve rows from the `pubs2 authors` table, insert them into `tempdb`, and append new rows for batch insert. The program then prints the updated `authors` table to STDOUT, disconnects, and exits.

```
use strict;

use DBI ();
use DBD::SybaseASE ();

require_version DBI 1.51;

# trace(n) where n ranges from 0 - 15.
# use 2 for sufficient detail.
#DBI->trace(2); # 0 - 15, use 2 for sufficient detail

# Login credentials, handles and other variables.
#
my $uid = "sa";
my $pwd = "";
my $srv = $ENV{"DSQUERY"} || die 'DSQUERY appears not set';
my $dbase = "tempdb";
my $temp_table = "$dbase..authors";

my $rows;
my $col1;
my $col2;
my $dbh;
```

```
my $sth;
my $rc;

# Connect to the target server:
#
$dbh = DBI->connect("dbi:SybaseASE:server=$srv;database=$dbase",
    $uid, $pwd, {PrintError => 0, AutoCommit => 0})
    || die "Connect failed, did you set correct credentials?";

# Switch to the pubs2 database.
#
$rc = $dbh->do("use pubs2") || die "Could not change to pubs2";

# Retrieve 2 columns from pubs2..authors table.
#
$sth = $dbh->prepare(
    "select au_lname, city from authors where state = 'CA'")
    || die "Prepare select on authors table failed";

$rc = $sth->execute
    || die "Execution of first select statement failed";

# We may have rows now, present them.
#
$rows = dump_info($sth);
print "\nTotal # rows: $rows\n\n";

# Switch back to tempdb, we take a copy of pubs2..authors
# and insert some rows and present these.
#
$rc = $dbh->do("use $dbase") || die "Could not change to $dbase";

# Drop the authors table in tempdb if present
#
$rc = $dbh->do("if object_id('$temp_table') != NULL drop table
$temp_table")
    || die "Could not drop $temp_table";

# No need to create a tempdb..authors table as the select into will
# do that.

$rc = $dbh->do("select * into $temp_table from pubs2..authors")
    || die "Could not select into table $temp_table";

# Example of a batch insert...
#
$sth = $dbh->prepare("
    insert into $temp_table
    (au_id, au_lname, au_fname, phone, address, city, state,
        country, postalcode) values
    ('172-39-1177', 'Simpson', 'John', '408 496-7223',
        '10936 Bigger Rd.', 'Menlo Park', 'CA', 'USA', '94025')

insert into $temp_table
(au_id, au_lname, au_fname, phone, address, city, state,
    country, postalcode) values
```

```
('212-49-4921', 'Greener', 'Morgen', '510 986-7020',
    '309 63rd St. #411', 'Oakland', 'CA', 'USA', '94618')

insert into $temp_table
(au_id, au_lname, au_fname, phone, address, city, state,
    country, postalcode ) values
('238-95-4766', 'Karson', 'Chernobyl', '510 548-7723',
    '589 Darwin Ln.', 'Berkeley', 'CA', 'USA', '94705')

insert into $temp_table
(au_id, au_lname, au_fname, phone, address, city, state,
    country, postalcode ) values
('267-41-4394', 'OLeary', 'Mich', '408 286-2428',
    '22 Cleveland Av. #14', 'San Jose', 'CA', 'USA', '95128')

insert into $temp_table
(au_id, au_lname, au_fname, phone, address, city, state,
    country, postalcode ) values
('274-80-4396', 'Straight', 'Shooter', '510 834-2919',
    '5420 College Av.', 'Oakland', 'CA', 'USA', '94609')

insert into $temp_table
(au_id, au_lname, au_fname, phone, address, city, state,
    country, postalcode ) values
('345-22-1785', 'Smiths', 'Neanderthaler', '913 843-0462',
    '15 Mississippi Dr.', 'Lawrence', 'KS', 'USA', '66044')

insert into $temp_table
(au_id, au_lname, au_fname, phone, address, city, state,
    country, postalcode ) values
('405-56-7012', 'Bennetson', 'Abra', '510 658-9932',
    '6223 Bateman St.', 'Berkeley', 'CA', 'USA', '94705')

insert into $temp_table
(au_id, au_lname, au_fname, phone, address, city, state,
    country, postalcode ) values
('427-17-2567', 'Dullest', 'Annie', '620 836-7128',
    '3410 Blonde St.', 'Palo Alto', 'CA', 'USA', '94301')

insert into $temp_table
(au_id, au_lname, au_fname, phone, address, city, state,
    country, postalcode ) values
('527-72-3246', 'Greene', 'Mstar', '615 297-2723',
    '22 Graybar House Rd.', 'Nashville', 'TN', 'USA', '37215')

insert into $temp_table
(au_id, au_lname, au_fname, phone, address, city, state,
    country, postalcode ) values
('672-91-3249', 'Yapan', 'Okiko', '925 935-4228',
    '3305 Silver Ct.', 'Walnut Creek', 'CA', 'USA', '94595')
");

$rc = $sth->execute || die "Could not insert row";

# Retrieve 2 columns from tempdb..authors table and present these
#
```

```
$sth = $dbh->prepare(
    "select au_lname, city from $temp_table where state = 'CA'")
    || die "Prepare select on $temp_table table failed";

$rc = $sth->execute
    || die "Execution of second select statement failed";

# Output
#
$rows = dump_info($sth);
print "\nTotal # rows: $rows";
print "\n";

sub dump_info {
    my $sth = shift;
    my @display;
    my $rows = 0;

while(@display = $sth->fetchrow) {
    $rows++;
    foreach (@display) {
    $_ = '' unless defined $_;
    }
    $col1 = $display[0];
    $col2 = $display[1];
    write;
    }
    $rows;
}

# The FORMAT template for this example.
#
format STDOUT_TOP =

Lastname            City
--------            -------
.

format STDOUT =

@<<<<<<<<<<<<    @<<<<<<<<<<<<
$col1, $col2
.

$dbh->disconnect;
```

Perl Error Messages

The Perl Driver generates error messages if internal driver APIs or CT-Lib's APIs fail. For example:

Example 1, message id 3

```
 OpenClient message: LAYER = (1) ORIGIN = (1) SEVERITY = (1) NUMBER =
(3)
```

```
Message String: s_command_alloc() : SybaseASE : Internal error:

        The ct_cmd_alloc() api failed.
```

Example 2, message id 14

```
OpenClient message: LAYER = (1) ORIGIN = (1) SEVERITY = (1) NUMBER =
(14)
Message String: s_db_connect() : SybaseASE : Internal error:
        cannot change to database tempdb.
```

The following table contains four fields:

- The message ID, which is printed in the NUMBER = (<num>) field
- The message text with parameters substituted at the specific code location in the driver
- The severity, which are either warning or fatal
- A possible fix or explanation

Message ID	Message Text	Severity	Fix/Explanation
1	`%1! unable to allocate memory.`	Fatal	Check system and memory resources. %1 can contain function names such as **cs_con_alloc()**, **ct_conn_alloc()**, and **malloc()**.
2	`%1! handle is null.`	Fatal	Internal Driver error. The connection or cmd handle is NULL
3	`The %1! api failed.`	Fatal	Internal Driver error. Can be caused by an invalid DSN connection string or internal CT-Lib. The error string contains the function name of the failing API.
4	`handle is null for statement id %1!.`	Reserved	Reserved
5	`failure for statement id %1!.`	Fatal	Internal Error. **ct_dynamic()** API trying to de-allocate a statement id that is failed.
6	`send failure for statement id %1!.`	Reserved	Reserved
7	`%1! must be > 0.`	Fatal	DSN string validation failed. Check DSN string for illegal characters.

Message ID	Message Text	Severity	Fix/Explanation
8	`%1! must be <= %2!, setting to maximum allowed value.`	Warning	An attempt was made to configure more connections than currently allowed (128).
9	`ct_config(CS_SET, %1!) failed. The supplied option %2! is illegal or missing.`	Fatal	**ct_config()** API failed because an invalid option was given. Check the DSN string and the string in the second parameter.
10	`cs_locale(CS_SET, %1! %2!) failed.`	Fatal	**cs_locale()** API failed. The error string indicates where the failure happened.
11	`cs_dt_info(CS_SET, CS_DT_CONVFMT) failed.`	Fatal	Error string will indicate where error occurred.
12	`ct_debug(CS_SET CS_DBG_PROTOCOL) failed.`	Reserved	Reserved
13	`ct_con_props(CS_SET, %1!) failed. The supplied option %2! is illegal or missing.`	Fatal	**ct_con_props()** API failed. The error string indicates where the failure happened.
14	`cannot change to database %1!.`	Fatal	Driver fails. Check DSN if the given database name exists.
15	`ct_command() failed for %1!.`	Fatal	**ct_command()** API failed. The error string indicates what type of CMD failed.

Message ID	Message Text	Severity	Fix/Explanation
16	`ct_send() failed for %1!.`	Fatal	The **ct_send()** API failed. The error string contained in %1 will give details.
17	`ct_describe() failed for column %1!.`	Fatal	**ct_describe()** API failed. Error string indicates where error occurred.
18	`ct_compute_info() failed on column %1! when describing column %2!.`	Fatal	**ct_compute_info()** API failed. Error string indicates column number and operation type involved.
19	`conversion failed %1!.`	Warning	**cs_convert()** had failures. The error string indicates where error occurred.
20	`ct_param() failed.`	Fatal	**ct_param()** API failed. Error string indicates failure location.
21	`ct_command() %1! failed for statement %2!.`	Fatal	**ct_command()** API failed. The error string indicates what type of CMD failed including the statement.
22	`ct_results() failed for %1!.`	Fatal	**ct_results()** API failed. The error string indicates the driver facility and failed statement.
23	`%1! command is ineffective with autocommit enabled.`	Warning	Error string indicates attempted commit or rollback that is invalid.
24	`ct_dynamic(CS_PREPARE) failed on statement %1!.`	Fatal	Dynamic Prepare failed. Statement name is supplied in the error string.
25	`ct_dynamic(CS_DESCRIBE INPUT) failed on statement %1!.`	Fatal	Dynamic Describe failed. Statement name is supplied in the error string.

Message ID	Message Text	Severity	Fix/Explanation
26	`ct_dynamic(CS_EXECUTE) failed on statement %1!.`	Fatal	Dynamic Execute failed. Statement name is supplied in the error string.
27	`%1! database handle is inactive, not connected to server.`	Fatal	An attempt was made to connect to a server with invalid database handle or inactive connection.
28	`sub connections are not allowed.`	Fatal	Sub connections not allowed if database handle active and in use
29	`cannot bind placeholder %1!.`	Fatal	Error attempting to bind placeholders. Error string indicates statement.
30	`unexpected cancel.`	Fatal	An unexpected cancel type was encountered while processing rows.
31	`unexpected return code from %1!.`	Fatal	An unexpected return code was encountered while processing rows.
32	`invalid format %1! provided to syb_date_fmt.`	Fatal	Invalid date format was given prior to date or time conversion.
33	`Fatal: multiple active statement handles on database handle without autocommit enabled.`	Fatal	User error where in the user Perl script more than one active handle is used without autocommit enabled.
34	`Fatal: invalid or unsupported DSN option provided.`	Fatal	If the Perl script has unsupported or obsolete options, DSN parsing fatally fails.

Additional Resources

Additional information about using the Perl driver.

- Building, testing and installation of the DBI driver:
  http://dbi.perl.org/
- The Perl DBI user programmable API calls:
  http://search.cpan.org/~timb/DBI-1.616/DBI.pm
- Open Client and Open Server documentation for configuration information:
  Open Client and Open Server Configuration Guide for UNIX > Configuration Files
- Initializing an application so that it executes using a specific language and related cultural conventions from a system configuration perspective:
  Open Client and Open Server Configuration Guide for UNIX > Localization
- Platform related issues for all the Open Client and Open Server products:
  Open Client and Open Server Programmers Supplement for UNIX
- Using the Open Client and Open Server runtime configuration file:
  Open Client Client-Library/C Reference Manual > Using the runtime configuration file > Open Client and Open Server runtime configuration file syntax
- Enabling an application to support multiple languages and cultural conventions:
  Open Client and Open Server International Developers Guide for UNIX > Understanding Internationalization and Localization
- Platform support:
  Software Developer Kit and Open Server Installation Guide for your platform.

Glossary

Glossary of term specific to scripting languages.

- **Client-Library –** part of Open Client, a collection of routines for use in writing client applications. Client-Library is designed to accommodate cursors and other advanced features in the Sybase product line.
- **CPAN –** Comprehensive Perl Archive Network. The Web site that holds a large collection of Perl software and documentation. See *http://www.cpan.org.*
- **CS-Library –** included with both the Open Client and Open Server products, a collection of utility routines that are useful to both Client-Library and Server-Library applications.
- **CT-Library –** (CT-Lib API) is part of the Open Client suite and is required to let an scripting application connect to Adaptive Server.
- **DBD –** database vendor-specific-driver that translates DBI database API calls into a form that is understood by the target database SDK.
- **DBI –** generic core database API that is database-vendor-agnostic and is the current standard for database access in a Perl application. See *http://dbi.perl.org*.
- **Driver –** the collection of Perl and C code that constitutes DBD::SybaseASE.
- **Extension or module –** the Perl language can be extended by modules that are written in Perl or a combination of Perl and C. In this document, extension and module denote the same.
- **Perl directory tree –** is one of:
  - The complete Perl installation that is installed as a binary module when the system is configured and has its operating system installed. A complete Perl installation is sometimes called the system (Perl) tree and owned by a system account (root, admin).
  - A private Perl (directory) tree, which has been built from source by a user other than the system account and is usually installed in a different location than the system Perl tree. This allows for new feature and bug-fix testing without compromising the system tree. A private directory tree is usually owned by the account that built the tree.
- **Perl script –** Perl is a scripting language that is widely used in system and database administration. See *http://www.perl.org*.
- **thread –** a path of execution through Open Server application and library code and the path's associated stack space, state information, and event handlers.
- **Transact-SQL –** an enhanced version of the database language SQL. Applications can use Transact-SQL to communicate with Adaptive Server Enterprise.

Index