The tool of thought for expert programming

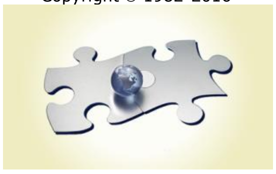
Dyalog APL SQAPL Interface Guide

Version 6.2

Dyalog Limited

Minchens Court
Minchens Lane
Bramley
Hampshire
RG26 5BH
United Kingdom

Tel: +44 (0)1256 830030
Fax: +44 (0)1256 830031
email: support@dyalog.com
http://www.dyalog.com



Contents

CHAPTER 1

Introduction

Note: With the exception of this chapter, this document is essentially unchanged since the release of Dyalog version 13.0 / SQAPL version 6.1, as there have been no significant changes in functionality.

ODBC

Since 1992, Open Database Connectivity (ODBC) has been a standard interface for accessing database management systems. In general, ODBC drivers will use Structured Query Language (SQL) to express queries and make updates to data. As a result, the use of an ODBC interface requires some understanding of SQL, unless you can make do with the functionality provided by the LoadSQL function described in this chapter.

ODBC drivers now exist for a very wide variety of databases, from simple drivers which give limited access to "flat" DOS files, through more sophisticated local database managers such as Microsoft Access, to multi-user DBMS systems such as Microsoft SQL Server, MySQL, Oracle, or DB2 running on a variety of server operating systems. ODBC drivers are even available for data sources which are not databases at all. For example, Dyalog provides an ODBC driver which allows APL applications to present themselves as “relational” data sources via ODBC (at the time of going to press, this product is still only available on a “project” basis).

ODBC has been the most widely used standard for database access under Microsoft Windows since the mid-1990’s – and the ODBC interface (known as *SQAPL*) has been bundled with Dyalog APL at no extra cost under Windows since that time. Under Linux and other UNIX platforms, ODBC adoption has accelerated since about 2005, and it now makes sense to think of ODBC as a standard on that platform. Support for unixODBC under Linux has been bundled with Dyalog since version 14.0.

However, as will be explained below, outside Microsoft Windows, ODBC is provided in a number of slightly different ways, and as a result SQAPL is still sold separately from the Dyalog interpreter - and only available for selected ODBC infrastructure components – on some platforms. As a result SQAPL runs on AIX, but requires chargeable proprietary drivers, and is not currently supported on OS X, although Dyalog is currently investigating the possibility of adding support.

The emergence of ODBC as a standard under Linux means that ODBC is likely to continue to be the most widely used standard for the foreseeable future, and is a particularly good choice for the development of portable applications.

SQAPL

SQAPL provides an interface between APL and database drivers which conform to the Microsoft ODBC specification. SQAPL consists of an APL workspace containing a set of interface functions which make calls to a DLL (under Windows) or Shared Library (under UNIX), which is written in C and provides a high-performance gateway to ODBC.

To be precise, SQAPL communicates with a component known as the *Driver Manager*, which is responsible for loading and managing database drivers. Under Microsoft Windows, Microsoft provides a standard *Driver Manager*, and a version of SQAPL which is compatible with this is bundled with Dyalog APL at no additional cost. On other platforms, a number of competing driver managers are available, and SQAPL needs to be compiled or at least linked with each of these different managers. For this reason, SQAPL is sold separately from the APL interpreter on some platforms. Platforms with "bundled" ODBC support included in version 14.1 are:

- Microsoft Windows
- Linux with unixODBC drivers

On other platforms, SQAPL is sold separately, and a porting fee may also apply in less common environments. Dyalog has tested SQAPL extensively with the driver manager and associated ODBC drivers from Progress Software, and is able to sell and support SQAPL bundled with Progress drivers on most platforms. Contact sales@dyalog.com for more information regarding SQAPL for Linux, UNIX and OS X.

With the exception of the `SQA.BrowseConnect` function, which only works under Microsoft Windows, all functionality described in this manual should be identical on all platforms where SQAPL is available.

Overview of ODBC Functionality

Using SQAPL, you can:

- Retrieve a list of available data sources, and connect to one or more of them.
- Query the database catalogue, to determine which tables, view and columns exist in a data source.
- Prepare and then repeatedly execute SQL statements, or execute SQL statements immediately without first preparing them. Multiple statements may be active simultaneously.
- Retrieve data from a result set. Retrieve a description of the contents of a result set.
- Execute SQL statements multiple times using a matrix containing a row of data for each execution (known as *Bulk Input*).
- Commit or roll back transactions.
- If connected with sufficient privileges, execute any SQL statement supported by the database management system, including the creation of tables or views (Data Definition Language – or DDL), indexes, stored procedures, or GRANT statements (etc).
- Retrieve a list of data types supported by a data source.

Most ODBC drivers are able to provide all of the functionality mentioned above. However, the ability to commit or roll back transactions and other more sophisticated functionality is often missing from drivers which access essentially non-SQL data sources such as CSV files.

The ODBC standard also defines SQL Grammar Conformance Levels. Some simple drivers lack all but the most basic SQL grammar, and support a very limited set of data types. Drivers for some systems often depart from the standard when it comes to data type names - the same data type may go under the name VARCHAR, LONGCHAR, MEMO, TEXT or NOTE depending on the driver being used.

While it is usually straightforward to write portable applications to execute queries, this becomes harder if you want to update data, and can be quite tricky if you want to create your own databases, as the DDL dialects can differ substantially.

If you use SQAPL functionality which is not supported by a driver, the result will typically be a "Driver not capable" or "SQL syntax error" message from the driver.

The LOADDATA Workspace

If all you need to do is load some data from an ODBC data source – or populate a table with data, the functions `LoadSQL` and `SaveSQL` from the utility workspace `LOADDATA` may be all that you need. The following example loads data from Microsofts sample Access Database called Northwind. The arguments to `LoadSQL` are [1] a DSN optionally followed by a password and user id if necessary, [2] a table name, [3] optionally a list of column names to be retrieved:

```
      )LOAD LOADDATA
C:\...\ws\loaddata saved Fri Mar 11 09:13:46 2011

      LoadSQL 'NorthWind' 'products'('ProductName' 'UnitPrice')
 Chai                                  18
 Chang                                 19
 Aniseed Syrup                         10
 Chef Anton's Cajun Seasoning          22
 Chef Anton's Gumbo Mix                21.35
…etc…
```

Below, we build an APL matrix and then use the `SaveSQL` function to create and populate a table to contain it:

```
      data←⍪'Jill' 'Jack' 'Betty' 'Bobby'
      data,←'1995-10-15' '1996-03-31' '1992-01-17' '1994-10-31'
      data,←167 175 172 180
      data
 Jill   1995-10-15  167
 Jack   1996-03-31  175
 Betty  1992-01-17  172
 Bobby  1994-10-31  180

      stmt←'create table sqatest'
      stmt,←'(name char(10),dob date,height integer)'
      DSN←'MySQL' 'secret' 'mkrom' ⍝ DSN, password, userid
      SaveSQL data DSN 'sqatest' stmt
0 4 4

      LoadSQL DSN 'sqatest' ⍝ Check that it worked
 Jill   1995-10-15  167
 Jack   1996-03-31  175
 Betty  1992-01-17  172
 Bobby  1994-10-31  180
```

For more information, see the comments in the two functions. The two utility functions demonstrated above are built on top of SQAPL and provide an easy way to move data between an ODBC data source and an APL workspace. The rest of this document describes how to use the SQAPL functions that the utilities above were built on top of. The `sqatest` table created above will be used in many of the examples.

Integrating SQAPL with Your Application

SQAPL is a workspace containing APL code, which you need to `)COPY` into your application (and may need to re-copy each time there is a new version of SQAPL). Having its origins in the 2nd millennium, SQAPL was originally designed as a “flat” workspace, which functions with names like `SQAInit` and `SQAConnect`. Version 5.0, released in 2005, placed the interface functions in a namespace called `SQA`. Dyalog recommends that you only `)COPY` this namespace into your application, and use the functions with names like `SQA.Init` and `SQA.Connect`, that can be found within it. A set of functions with the old names are still provided in order to support old application code, but they are simply cover-functions which call `SQA.Init`, `SQA.Connect`, and so forth. This documentation will use direct calls to the functions contained in the `SQA` namespace.

CHAPTER 2

Getting Started

Initialisation

Before you can use SQAPL, you must call the function `SQA.Init`. This function loads and initialises the appropriate SQAPL library.

```
      SQA.Init ''
0  SQAPL loaded from: C:\Program Files\Dyalog\Dyalog APL
13.0 Unicode\bin\cndya61Uni
```

Note that the first element of the result of `SQA.Init` is a return code – as is the case for all SQAPL functions. If the operation was completely successful the return code is zero, and in the event of a non-fatal error (warning), the return code is negative. The second element provides information about the version that was loaded.

Connecting to a Service

To establish a conversation with a driver, you must first create a connection object. The function `SQA.Connect` takes a right argument which must contain an object name and a data source name (DSN), and optionally a password and a user id if the data source requires them. It is possible to have several connections open at the same time, and the object name is used to identify the connection in subsequent calls to SQAPL.

The following example creates a *connection object* named `C1` that is connected to the data source name `MySQL` using the password `secret` and the user id `mkrom` (the password is provided first in case you connect to a data source which has a default user ID). The DSN (`MySQL`), is the name of an ODBC data source which has been defined using *Windows Control Panel => Administrative Tools => Data Sources (ODBC)*[1].

```
      SQA.Connect 'C1' 'MySQL' 'secret' 'mkrom'
0
```

The return code of `0` indicates that the connection was successful. You can create several connection objects, and access more than one data source at the same time. For example:

```
      ro←'ReadOnly' 1
      SQA.Connect 'C2' 'PROD DB2' 'secret' 'bhc' ro
0
```

When experimenting with live or production data, you can prevent yourself from accidentally damaging the data by using the `ReadOnly` option. This is also useful in applications where you give users the ability to enter SQL statements in a report generator, as it allows you to ensure that users will not be able to accidentally (or

[1] It is possible to connect to databases for which there is no pre-defined data source – see the description of the `SQA.Connect` function, in particular the `DriverOptions` parameter. Also see the warning on the next page regarding 32- vs 64-bit ODBC drivers under Windows.

intentionally) modify the database. `SQA.Connect` takes a large number of optional parameters which are described in the *Function Reference* section.

```
      SQA.BrowseConnect 'C3'
0
```

The last example above uses the function `SQA.BrowseConnect`, which instructs the driver manager to produce a dialog box which allows the user to select the data source and provide connection parameters interactively (this functionality is only available under Microsoft Windows).

You can also use the `SQA.DSN` function to get a list of available data source names:

```
      SQA.DSN ''
0   CRMIS                SQL Server
    dBASE Files          Microsoft Access dBASE Driver
    Excel Files          Microsoft Excel Driver
    MS Access Database   Microsoft Access Driver
    Northwind            Microsoft Access Driver (*.mdb)
    APLSRV               DataDirect OpenAccess SDK 6.0
    MySQL                MySQL ODBC 5.1 Driver
```

Warning: If you have a mixture of 32- and 64- bit data sources under 64-bit Windows, note that *Windows Control Panel => Administrative Tools => Data Sources (ODBC)* only administers 64-bit drivers, despite being a short cut to a file called `%windir%\system32\odbcad32.exe`. The 32-bit administrator is available on a 64-bit system, but it is in the folder `%windir%\sysWOW64`. `SQA.DSN` cannot be relied upon to return the complete list of 32- and 64-bit drivers in a mixed environment, although you *should* be able to connect to them all.

Return Codes

Errors: In the event that a function should fail, the first element of the result will be a positive number identifying the source of the error message. This number will either be 1 for errors originating in SQAPL, 4 for errors issued by an ODBC component[2] or 6 for errors from unknown sources. The second element will be one or more condition codes. The third element will contain a textual error message, and the fourth element provides a row index which is only relevant when multiple rows of input were involved (see `SQA.X`). See the *Troubleshooting* section and the appendix *Errors and Warnings* for further help on interpreting error return codes.

Warnings: If an SQAPL function receives a warning during execution, the result will have the normal form, but the return code will be minus one (`¯1`). You can retrieve the warning message using the function `SQA.GetWarning`. For example, connecting to Microsoft SQL Server nearly always produces one or two warning messages:

```
      SQA.Connect 'C1' 'CRMIS'
¯1
      t←SQA.GetWarning 'C1'
      1⊃t ⍝ Return code for GetWarning itself
0
      ρ2⊃t ⍝ How many warnings?
2
      2 1⊃t ⍝ First warning
4   01000  5701  [Microsoft][ODBC SQL Server Driver][SQL
Server]Changed database context to 'CRM'.  0
```

[2] Return codes 2, 3 and 5 were returned by a product called *SequeLink* which predates ODBC.

```
      (⍳4),⍪⍕¨2 2⊃t ⍝ Second warning (formatted)
1  4
2   01000  5703
3  [Microsoft][ODBC SQL Server Driver][SQL Server]
   Changed language setting to us_english.
4  0
```

Taking a closer look at the second element of the result of `SQA.GetWarning`, we can see that it is a 2-element vector (one for each warning) of 4-element vectors (return code structures). Looking at the 2nd warning, we see that [1] the source of the error message is 4, meaning that an ODBC component issued the error, [2] the return code was 01000 5703 (try searching the internet for "SQL Server 01000 5703" for more information), [3] the textual error message which also clearly identifies the source, and finally [4] a row index of 0.

SQAPL "Objects"

Note that SQAPL predates the introduction of real objects in Dyalog APL by about a decade; the "objects" described in this document are all external to the APL workspace, and referred to by names held in character vectors.

SQAPL is built around four classes of *objects*. The function `SQA.Init` creates a *root* object, which is named `'.'` or `'#'` (you can use either name). `SQA.Connect` creates *connection* objects as children of the root. Later, we will see how the function `SQA.Prepare` creates *cursor* objects as children of a connection. A cursor object contains an SQL statement and status information for a query.

Each segment of the name of an SQAPL object must begin with a letter of the English alphabet. The rest of the segment name must be alphanumeric. The names *are* case sensitive.

Disconnecting from a Service

The function `SQA.Close` can be used to close any SQAPL object. When applied to a connection object, all children of the connection (its cursors) are closed, and you are disconnected from the driver. All files, network conversations and host logons associated with the connection are also closed.

```
      SQA.Close 'C1'
0
      SQA.Close '.'
0
```

If you close the root object using an argument of `'.'` or `'#'`, you will disconnect all existing services, and unload the database drivers.

Data Dictionary

The functions `SQA.Tables` and `SQA.Columns` can be used to query the database catalogue. The first element of the result is the usual return code; the second contains information about tables or columns. The following examples all assume that we have a database which contains only the `sqatest` table that was created at the end of the section on the `LOADDATA` workspace in Chapter 1.

```
      2⊃SQA.Tables 'C1'
 TABLE_CAT  TABLE_SCHEM  TABLE_NAME  TABLE_TYPE  REMARKS
 mkrom                   sqatest     TABLE
```

```
      (⍳18),⍉2⊃SQA.Columns 'C1'
 1   TABLE_CAT
 2   TABLE_SCHEM
 3   TABLE_NAME           sqatest  sqatest  sqatest
 4   COLUMN_NAME             name      dob   height
 5   DATA_TYPE                ¯8       91        4
 6   TYPE_NAME               char     date  integer
 7   COLUMN_SIZE               10       10       10
 8   BUFFER_LENGTH             30        6        4
 9   DECIMAL_DIGITS             0        0        0
10   NUM_PREC_RADIX             0        0       10
11   NULLABLE                   1        1        1
12   REMARKS
13   COLUMN_DEF
14   SQL_DATA_TYPE             ¯8        9        4
15   SQL_DATETIME_SUB           0       91        0
16   CHAR_OCTET_LENGTH         30        0        0
17   ORDINAL_POSITION           1        2        3
18   IS_NULLABLE              YES      YES      YES
```

Note that the result of `SQA.Columns` is transposed, and has column numbers appended to the front. For a precise explanation of each of the output columns, see the appendix titled *Schema Information*.

You can follow the connection name with optional parameters which are used to filter the table name and (in the case of `SQA.Columns`) column names. For example, we can filter the result on tables with names beginning with "sqa" and columns with names beginning with "n":

```
      (2⊃SQA.Columns 'C1' 'sqa%' 'n%')[;3 4 5 6]
 TABLE_NAME  COLUMN_NAME  DATA_TYPE  TYPE_NAME
 sqatest     name                ¯8  char
```

Executing SQL Statements

The simplest way to execute SQL statements against a data source is to use the function `SQA.Do`, which simply takes a connection and an SQL statement as its arguments. For example:

```
      stmt←'select * from sqatest where name like ''J%'''
      SQA.Do 'C1' stmt
0  C1.s1    Jill  1995-10-15  167       6
            Jack  1996-03-31  175
```

The `SQA.Do` function creates a temporary cursor object under the connection named in the first element of the right argument. The name of the cursor object is returned as the second element of the result. The result of the SQL expression (if any) is returned as the third element. For reasons explained in the detailed description of the `SQA.Fetch` function which `SQA.Do` makes use of, the third element is a one-element vector containing the result matrix. The fourth element contains a status flag which will typically have one of the values 5 (more data to fetch) or 6 (all data has been fetched).

Note that quotes need to be doubled up in the SQL expression, in the same way as they would need to be if you were creating a vector containing an APL expression to be executed.

When there is no left argument, or the left argument is `1`, `SQA.Do` will loop until all data has been fetched. If you need to deal with arbitrarily large amounts of data, you may need to *block* the output. The block size is determined by the *MaxRows* parameter, which you can set when you connect. If you want to set different values for the block size for each query, you must use the low-level functions described later in this chapter.

To fetch one block at a time, give `SQA.Do` a left argument of 0 (as in the above example). Fetch subsequent blocks by calling `SQA.Do` again with no left argument, and a right argument containing the name of the cursor:

```
      SQA.Connect 'C1' 'MySQL' 'secret' 'mkrom' ('MaxRows' 3)
0

       0 SQA.Do 'C1' 'select * from sqatest'
0  C1.s1     Jill   1995-10-15  167        5
             Jack   1996-03-31  175
             Betty  1992-01-17  172
       SQA.Do 'C1.s1'
0  C1.s1     Bobby  1994-10-31  180        6
       SQA.Do 'C1.s1'
1  10043   APL No data available  0
```

Important: The database cursor remains open until you have fetched all the records in the result set. If you are connected to a multi-user server, there may be important resources allocated to or locked by the cursor, which will not be released until you have fetched all the data. Be particularly careful when experimenting interactively, as you may lock other users out of one or more tables.

Not all databases are equally temperamental about this - some will allow you to have SELECT cursors open without locking others out completely, but as a general rule you should close cursors as soon as you no longer need them. If you decide that you do not want to fetch the rest of the data, or if you have been using `SQA.Do` with a left argument of `0`, you should ***always*** call `SQA.Close` to close the cursor:

```
       SQA.Close 'C1.s1'
0
```

Bind Variables

So far, all values used in our SQL have been provided as constants embedded in the SQL expression, which character strings surrounded by doubled quotes. In application code, it is often convenient to use SQL statements containing parameters, which are provided separately from the statement itself. There are at least two reasons for this: In the next chapter, we will look at using functions which allow us to *prepare* a statement and then *execute* it more than once – which gives more control and better performance. Even if you are using `SQA.Do` and are only able to execute a statement once, the use of parameters allows you to provide APL values directly, rather than formatting them for use as constants in the SQL statement, doubling quotes, etc.

The ODBC terminology for parameters which are “bound” at execution time is *bind variables*.

Bind Variable Notation

When using `SQA.Do` bind variable declarations are made “in-line” in the SQL statements, between colons. For example:

```
       SQA.Do 'C1' 'select * from sqatest where
            name like :n<C10: or height > :h<I:' 'J%' 175
0  C1.s1     Jill   1995-10-15  167        6
             Jack   1996-03-31  175
             Bobby  1994-10-31  180
```

This statement declares a statement with two bind variables. The first is a character string of up to 10 characters (`:n<C10:`), and the second is an integer (`:h<I:`). The two values for the variables are provided as trailing elements of the argument to `SQA.Do`. When you use the syntax with "inline" declarations, SQAPL will replace each bind variable declaration with a "`?`" before passing the statement to the ODBC driver. Apart from this, statements are always passed unchanged to the driver. Bind variable values must be provided in the same order as variables appear in the statement.

SQAPL allows a name to precede the `<` symbol, for example `:name<C10:` or `:height<I:`. For bind variables, this name is only currently used when doing *partial binding* of values which are too large to be provided in a single call (see the chapter titled *Advanced Topics*). The names also appear in the output of `SQA.Describe`[3], which can make it easier to understand the output of this function.

Note that numeric types generally do not require a length to be provided, but almost all other types do.

Output Variables

While it is your responsibility to declare all bind variables, ODBC will always provide data type information for the columns in the output, or *result set* of an SQL query. For every one of the 22 ODBC data types in the following table, SQAPL automatically selects a default APL type which is capable of representing the data.

The table shows the ODBC type name, the type number, the code letter that must be used if you want to explicitly select an ODBC type conversion for a bind variable, and the APL type that is used if you do not explicitly select an APL type on output. The final column shows whether a declaration of precision and scale is relevant in a type declaration:

ODBC Type Name	**Type Number**	**Code Letter**	**Default APL Type**
WLongVarChar	-10	Q	Char (C)
WVarChar	-9	W	Char (C)
WChar	-8	U	Char (C)
Bit	-7	B	Char (C)
TinyInt	-6	K	Integer (I)
BigInt	-5	G	Integer(I)
LongVarBinary	-4	Z	Binary (X)
VarBinary	-3	Y	Binary (X)
Binary	-2	X	Binary (X)
LongVarChar	-1	L	Char (C)
Char	1	C	Char (C)
Numeric	2	N	Float (F)
Decimal	3	M	Float (F)
Integer	4	I	Integer (I)
SmallInt	5	H	Integer (I)
Float	6	F	Float (F)
Real	7	R	Float (F)
Double	8	E	Float (F)
VarChar	12	V	Char (C)
Date	91	D	Char (C10)

[3] A utility function which displays information about prepared statements – see Chapter 4.

Time	92	T	Char (C8)
Timestamp	93	S	Char (C18-26)

Select (or Output) variables are declared in the same way as bind variables, except that the inline form begins with a `>`, and column `[;1]` contains a `0` in the matrix form. For example, we can ask to have the dates converted to day numbers since 1900-01-01 using the APL type `J` (see the table on the next page) as follows:

```
      stmt←'select :name>C10:,:dob>J:,:height>I: from sqatest'
      SQA.Do 'C1' stmt
0  C1.s1    Jill   34985 167       6
            Jack   35153 175
            Betty  33618 172
            Bobby  34636 180
```

Note that it is not *necessary* to declare output types; SQAPL will always select a type which can receive the data, but in many cases it will use character vectors as the default, as it would for dates and times. Careful selection of a good output type can simplify your code (and generally make it run faster, too).

Also note that, while the position of a bind variable must correspond exactly to its use in the SQL statement, several output variables can declared in a single declaration, separated by commas. We could have written either of:

```
      stmt←'select :*>C10,J,I: from sqatest'
      stmt←'select * from sqatest :>C10,J,I:'
```

Any text to between the leading `:` and the `>` will be passed through to the database. The latter form will often be used in this document. Using names in select declarations makes the output of `SQA.Describe` easier to read. It also makes it possible to declare the type of an individual columns, for example:

```
      stmt←'select name,:dob>J:,height from sqatest'
```

There is no reason to declare the type of `name` and `height`, as the defaults are fine. If you have both input and output declarations for the same column in a statement, you need to use the real name of the column for output declarations, and should use no name or a modified name for the bind variables, where the name generally has no significance.

APL Types

SQAPL version 6.1 recognizes 13 different APL variable “types”. First, there are the four types that APL and ODBC have in common, that are sufficient to represent all ODBC types in APL, and are mentioned in the ODBC type table on the previous page: Integers (I), Floating-point numbers (F), Character vectors (C – translated to *VarChar* in a “Classic” interpreter or *WVarChar* when using Unicode) – and a “Binary” type (X) to map single-byte character data (Classic or Unicode) to a byte stream without translation.

Three types have been added to allow the representation of APL types that ODBC does not support: Complex numbers (O), 128-bit decimal numbers (G) and the general Array type (Z), which can be used to serialize any APL array to a character representation that can be stored in a Char or Binary columns.

Finally, no less than six types have been added to provide different mappings from Dates and Timestamps to different APL representations (D, T, S, Y, H and J).

In the same way that SQAPL has a default APL type that it maps ODBC types to on output, there is a default ODBC type that is used for each APL type on input – unless a different type is explicitly selected. The following table lists all the APL types, the letters to use when declaring types in-line and the numbers to use in the matrix form, and the default ODBC type that will be used to store the value unless you specify a type to use.

APL Type Name	**#**		**Default ODBC Type**	**Description**
Char	1	C	VarChar (V) or WVarChar (W)	WVarChar for Unicode, Classic translates to ANSI VarChar
Integer	2	I	Integer (I)	Integer value
Float	3	F	Double (E)	Floating point value
Binary	4	X	Binary (X)	Stream of bytes (untranslated characters)
Complex	5	O	VarChar (V)	Complex number
Array	6	Z	LongVarChar (L)	Serialized APL Array (SCAR)
Date	7	D	Date (D)	3-vector: `yyyy mm dd`
Time	8	T	Time (T)	3-vector: `hh mm ss`
Stamp	9	S	Timestamp (S)	7-vector in `⎕TS` format: `yyyy mm dd hh mm ss fff` (fff is 0-999 msec)
DateInt	10	Y	Date (D)	8-digit integer: `yyyymmdd`
TimeInt	11	H	Time (T)	6-digit integer: `hhmmss`
StampFloat	12	J	TimeStamp (S)	Floating point value: Days since 1900 1 1, with time in the fractional part
Decf	13	G	VarChar (V)	128-bit decimal value

Explicit Type Conversions

In the early days of ODBC, drivers were often very bad at doing type conversions. If you had an input column of type Bit and provided an Integer value, early drivers would complain. For this reason, SQAPL has the ability to perform explicit type casts for many types. A type declaration of `:<I(B):` would be an instruction to accept an APL Integer (`I`) and cast it to ODBC Bit (`B`). Nowadays, drivers are pretty flexible, as demonstrated by the following example using MySQL (return codes are not displayed, they were all simply (`0 C1.s1`):

```
      SQA.Do 'C1' 'create table testbit (abit bit(4))'
      SQA.Do 'C1' 'insert into testbit values(b''0101'')'
      SQA.Do 'C1' 'insert into testbit values(:<I:)' 6
      SQA.Do 'C1' 'insert into testbit values(:<F:)' 7
      SQA.Do 'C1' 'insert into testbit values(:<C1:)' (⎕UCS 8)

      ,⎕UCS¨ 3 1⊃SQA.Do 'C1' 'select * from testbit'
 5  6  7  8

      SQA.Do 'C1' 'drop table testbit'
```

However, it is still possible that you will encounter a situation where explicit conversions are necessary. Apart from lazy drivers, they are necessary for the insertion of large character vectors – see the section titled *Very Wide Char Columns*.

Data Types Supported by a DSN

The function `SQA.TypeInfo` returns a table which lists the data types supported by a particular driver - assuming the driver supports the *SQLTypeInfo* ODBC API function. These days, most drivers can provide the information, and most drivers support a fairly complete set of types. However, drivers for file systems which are not real relational databases often support a restricted set. For example, if you connect to a driver for Microsoft Excel files, you might only have a handful of types. Note that the result has been transposed below – the 2nd element of the result of `SQA.TypeInfo` is a matrix with 19 columns:

```
      ⍉2⊃SQA.TypeInfo'C2'
 TYPE_NAME           LOGICAL  CURRENCY  NUMBER  VARCHAR  DATETIME
 DATA_TYPE                ¯7         2       8       12        93
 COLUMN_SIZE               1        19      53      255        19
 LITERAL_PREFIX                                       '         #
 LITERAL_SUFFIX                                       '         #
 CREATE_PARAMS
 NULLABLE                  0         1       1        1         1
 CASE_SENSITIVE            0         0       0        1         0
 SEARCHABLE                2         2       2        3         2
 UNSIGNED_ATTRIBUTE        0         0       0        0         0
 FIXED_PREC_SCALE          0         1       0        0         0
 AUTO_UNIQUE_VALUE         0         0       0        0         0
 LOCAL_TYPE_NAME
 MINIMUM_SCALE             0         4       0        0         0
 MAXIMUM_SCALE             0         4       0        0         0
 SQL_DATA_TYPE            ¯7         2       8       12         9
 SQL_DATETIME_SUB          0         0       0        0         3
 NUM_PREC_RADIX            0        10       2        0         0
 INTERVAL_PRECISION        0         0       0        0         0
```

From the above, we can see the names that the driver uses for (`TYPE_NAME`) the five data types that it supports, the corresponding ODBC data type number (DATA_TYPE) for each of the types, the number of bytes consumed by an element of the type, and a number of other items of information about the type. For a detailed explanation, see the appendix titled *Schema Information*.

CHAPTER 3

Using Data Types

This chapter provides discussions and examples of the ODBC and APL types that are supported by SQAPL.

Simple Numeric Types

ODBC supports 3 flavours of integer and 3 flavours of floating-point, which can be mapped to APL integers and floats without problems (the range of the APL types is greater than the ODBC types):

ODBC Type Name	**Type Number**	**Code Letter**	**Default APL Type**
TinyInt	-6	K	Integer (I)
Integer	4	I	Integer (I)
SmallInt	5	H	Integer (I)
Float	6	F	Float (F)
Real	7	R	Float (F)
Double	8	E	Float (F)

TinyInt has a range of 0-255, *SmallInt* is equivalent to the APL type 163 (from ¯32,768 to 32,767) and *Integer* is equivalent to APL type 323. Obviously, you won't be able to store numbers bigger than 255 in a TinyInt (etc), but apart from that the APL type `I` will work without problems.

Float and *Double* are synonyms, which both map exactly APL type 645. *Real* is a single-precision float, with ~7 digits of precision and an exponent range of +/-38. You will lose precision saving APL floats to a Real column, but otherwise the APL type `F` should work fine.

High Precision Numbers

ODBC has three types which have the potential to overflow the precision of the APL *Integer* and *Float* types:

ODBC Type Name	**Type Number**	**Code Letter**	**Default APL Type**
BigInt	-5	G	Integer (C)
Numeric	2	N	Float (F)
Decimal	3	M	Float (F)

BigInt is a 64-bit integer type, which is formatted and mapped to Char. *Numeric* and *Decimal* are synonyms for a fixed precision type with 38 decimal digits and a range of -1E38+1 to +1E38-1 are mapped to APL Floats – with the potential to lose significant precision. From version 13.0, it is probably better to map all types to the new 128-bit decimal floating-point type (`G`), but the default mappings have been retained in order to maintain backwards compatibility. The following examples show how to use `G` with these types:

```
      SQA.Do 'C1' 'create table testbignums
                   (col1 bigint, col2 numeric(19,2))'
      SQA.Do 'C1' 'insert into testbignums values
               (123456789012345678,12345678901234567.89)'
      ⎕pp←10 ⋄ ⎕fr←645
      expr←'select * from testbignums'
      3 1⊃SQA.Do 'C1' expr
 123456789012345678  1.23456789E16
      ⎕dr¨3 1⊃SQA.Do 'C1' expr
80 645
      ⎕pp ⎕fr←30 1287
      3 1⊃SQA.Do 'C1' (expr,':>G,G:')
123456789012345678 12345678901234567.89
      0.001+3 1⊃SQA.Do 'C1' (expr,':>G,G:')
123456789012345678.001 12345678901234567.891
```

Note that the maximum precision of these types is 38, and 128-bit decimal numbers have 34 digits of precision, so there is still a risk that precision will be lost.

Complex Numbers

From version 13.0, APL also supports complex numbers, and an APL type O supports this type. However, since there is no corresponding ODBC type, data must be stored in a *Char* or *Binary* column. The following example shows the use of both types in a single table:

```
      c←*0J1 ⍝ A complex number
      SQA.Do 'C1' 'create table testcomplex
             (c1 char(47), c2 Binary(16))'
```

47 characters are required to format a complex number. Only 16 bytes are required to store the untranslated internal form of a complex number. In the following statement, the first column is inserted using the default *Char* form, the second is explicitly cast to *Binary* (X):

```
      SQA.Do 'C1' 'insert into testcomplex
             values (:<O:,:<O(X):)' c c
      expr←'select * from testcomplex'
      3 1⊃SQA.Do 'C1' expr
 5.4030230586813977e-001j8.4147098480789650e-001■ µ(Já?î  ■Tíê?
      c=3 1⊃SQA.Do 'C1' (expr,':>O,O:')
1 1
```

If we select the data without specifying an APL output type, the raw character and binary data is returned to APL. The final statement forces conversion back to complex numbers in APL, and as we can see they have survived the round trip.

Character Data

ODBC has three basic (single-byte) character types: *Char* which is typically used for narrow, fixed-width fields containing text, *VarChar* which was added to allow the storage of longer strings (generally up to 255 characters), and *LongVarChar* for very long strings (with an upper limit of 64k, and sometimes more). Not all databases support all three types or have the same length limits, and the optimization strategies vary from one database to another, and are beyond the scope of this document.

With the advent of Unicode, each of the basic types has a “wide” equivalent, with the addition of *WChar*, *WVarChar* and *WLongVarChar*, bringing the total to 6 character types:

ODBC Type Name	Type Number	Code Letter	Default APL Type
Char	1	C	Char (C)
VarChar	12	V	Char (C)
LongVarChar	-1	L	Char (C)
WChar	-8	U	Char (C)
WVarChar	-9	W	Char (C)
WLongVarChar	-10	Q	Char (C)

In all cases, the default APL type is Char, denoted by the letter C. In a Classic interpreter, SQAPL will translate character data between `⎕AV` and the ANSI character set, which means that the "wide" types are not really usable, and you will need to use a Binary column to store anything other ANSI data (or better: upgrade to Unicode!).

When using the Unicode edition of Dyalog APL, all character types should "just work". For example (return codes not displayed if 0):

```
      SQA.Connect 'C2' 'MSSQLSRV'
      SQA.Do 'C2' 'create table testuni
                   (name char(10),body ntext)'
      i←'insert into testuni values (:n<C10:,:b<C50:)'
      SQA.Do'C2' i 'avg' '{+/⍵÷⍴⍵}'
      SQA.Do'C2' i 'tally' '{⊃⍴⍵}'
      3 1⊃SQA.Do 'C2' 'select * from crm.testuni'
 avg         {+/⍵÷⍴⍵}
 tally       {⊃⍴⍵}
```

The example shows that Unicode strings (containing APL symbols) can be stored and faithfully retrieved from a WLongVarChar column in MS SQL Server, which calls that column type `ntext`.

Very Wide Char Columns

By default, the APL character type maps to the ODBC type WVarChar (`W`), which has a maximum length of 4,000 characters. For really long strings, you will need to explicitly select an ODBC type of `Q` for WLongVarChar (or if the data is all single-byte, `L` for LongVarChar). For example:

```
     SQA.Do 'C1' 'insert into testtab (a_wide_col)
                  values (:<C10000(L):)' (10000⍴⎕A)
```

UTF-8 Data

Many databases do not have "native" Unicode support. The previous example using a *WLongVarChar* type had to be performed with Microsoft SQL Server rather than MySQL, because MySQL has no "W" data types. In databases without actual Unicode data types, it is common to store Unicode data using an encoding known as UTF-8. UTF-8 is a variable-length encoding which is identical to 7-bit ASCII for code points 0-127, and then uses multi-byte sequences to represent characters outside that range. UTF-8 encoded text files are now so widespread that it is common for people to simply refer to UTF-8 as "Unicode", can cause much confusion.

The following utility functions can be used to convert to and from UTF-8:

```
      toUTF8←{⎕UCS 'UTF-8' ⎕UCS ⍵}
      fromUTF8←{'UTF-8' ⎕UCS ⎕UCS ⍵}
      ⎕←fromUTF8 ⎕←⎕UCS ⎕←toUTF8 '{+/⍵÷⍴⍵}'
{+/â■µÃ·â■´â■µ}
123 43 47 226 141 181 195 183 226 141 180 226 141 181 125
{+/⍵÷⍴⍵}
```

Using these utility functions, we can repeat the Unicode experiment using "ordinary" character columns and MySQL:

```
      SQA.Do 'C1' 'create table testuni
                   (name char(10),body varchar(100))'
0  C1.s1
      i←'insert into testuni values (:n<C10:,:b<C256:)'
      SQA.Do'C1' i 'avg' (toUTF8 '{+/⍵÷⍴⍵}')
0  C1.s1
      SQA.Do'C1' I 'tally' (toUTF8 '{⊃⍴⍵}')
0  C1.s1
      SQA.Do 'C1' 'select * from testuni'
0  C1.s1    avg    {+/â•µÃ·â•´â•µ}        6
            tally  {âŠƒâ•´â•µ}
      data←3 1⊃SQA.Do 'C1' 'select * from testuni'
      fromUTF8¨data
 avg    {+/⍵÷⍴⍵}
 tally  {⊃⍴⍵}
```

With a little extra work, UTF-8 allows us to store Unicode text in SQL databases without native support for Unicode data.

Binary Data

In the past, Character data was typically restricted to ASCII values in the range 0-127, while Binary columns could store "byte" values with a range of 0-255. Under the Classic edition of APL, character data was translated when written to Character buffers, but not when written to Binary.

Over time, Character columns have been extended to store values up to 255, accelerated by the advent of Unicode and the need to store UTF-8 values in Character columns. At the same time, Unicode APL means that character data no longer needs to be translated, and there is no difference between the APL types `C` and `X`. Note that you should always specify the size of the buffer that you need, for example `:<C10:` or `:>X256:`.

As a result, there is now practically no difference between Character and Binary columns. However, the ODBC and APL types still exist:

ODBC Type Name	**Type Number**	**Code Letter**	**Default APL Type**
Binary	-2	X	Binary (X)
VarBinary	-3	Y	Binary (X)
LongVarBinary	-4	Z	Binary (X)
Bit	-7	B	Binary (X)

Use of the `X` APL type is important to avoid translating data when using the Classic edition, otherwise it should make no difference whether you use `C` or `X`. The X type does accept a modifier which allows you to specify that you do want translation of Binary data (for example, `:<X256#1:`). This only applies to the Classic edition.

Dates and Timestamps

ODBC provides three types for storing dates and times:

ODBC Type Name	Type Number	Code Letter	Default APL Type
Date	91	D	Char (C10)
Time	92	T	Char (C8)
Timestamp	93	S	Char (C19)

By default, SQAPL will map all three types to character strings on selection. However, no less than six APL types are provided in order to provide flexible mapping for dates and times to APL arrays:

APL Type Name	#		Default ODBC Type	Description
Date	7	D	Date (D)	3-vector: `yyyy mm dd`
Time	8	T	Time (T)	3-vector: `hh mm ss`
Stamp	9	S	Timestamp (S)	7-vector in `⎕TS` format: `yyyy mm dd hh mm ss fff` (fff is 0-999 msec)
DateInt	10	Y	Date (D)	8-digit integer: `yyyymmdd`
TimeInt	11	H	Time (T)	6-digit integer: `hhmmss`
StampFloat	12	J	TimeStamp (S)	Floating point value: Days since 1900 1 1, with time in the fractional part

The following example creates a table and populates it with one date, one time and one timestamp value (as usual, `0` return codes are not displayed). First, we inspect the “typeinfo” to see what MySQL calls these types:

```
      t←2⊃SQA.TypeInfo 'C1'
      3↑[2](t[;2]∊91 92 93 'DATA_TYPE')⌿t
 TYPE_NAME  DATA_TYPE  COLUMN_SIZE
 date              91           10
 time              92            8
 datetime          93           21
 timestamp         93           14
```

The above reveals that there are two timestamp formats to choose between, with varying precision. 14 digits is only enough to store whole seconds (yyyymmddhhmmss), 21 gives 6 additional digits. We’ll go for the type called “datetime”, as we do want to store milliseconds:

```
      SQA.Do 'C1' 'create table testts
                  (d date, t time, s datetime)'
      ⎕←now←⎕ts
2011 3 26 22 45 40 282
      sql←'insert into testts values(:<D:,:<T:,:<S:)'
      SQA.Do 'C1' sql (3↑now) (3↑3↓now) now
      3 1⊃SQA.Do 'C1' 'select * from testts'
 2011-03-26  22:45:40  2011-03-26 22:45:40
```

As can be seen above, the default on output is to map all three data types to character vectors (C10, C8 and C19, respectively). If we want the values returned in the same format as we provided as input, we need explicit output declarations:

```
      3 1⊃SQA.Do 'C1' 'select * from testts :>D,T,S:'
 2011 3 26  22 45 40  2011 3 26 22 45 40 0
```

Well… almost! The observant reader will notice that we have lost our 282 milliseconds. There are a couple of reasons for this - for one thing, the default “scale” for the S type is 0, meaning no decimals. This default has been chosen because some databases will give an error if fractional timestamps are provided. To input milliseconds, you need to specify a *scale* of 3, using a declaration like `:<S.3:`, or `:<S23.3:` if you prefer.

Despite the reported precision of 21, a search on the internet finds the page http://dev.mysql.com/doc/refman/5.1/en/datetime.html, which says that “… microseconds cannot be stored into a column of any temporal data type. Any microseconds part is discarded.”

The ability to store fractions of seconds is one of the areas where the ODBC standard turns out not to be fully supported by all databases. Despite reporting that it supports microseconds when asked, MySQL does not. Other (commercial) databases suffer from similar problems in this area which vary from version to version, you will need to experiment with your own database to find out what is possible. Recent versions of Microsoft SQL Server will accept millisecond input but the internal precision is not quite high enough, so you may not get the same number back:

```
      SQA.Prepare 'C1.I1' 'insert into sqatest
                          values (:<C10:,:<S23.3:,:<I:)'
0
      SQA.Exec 'C1.I1' 'Mary' (1962 10 31 1 2 3 456) 170
0 1
      3 1⊃SQA.Do 'C1' 'select * from sqatest'
 Mary   1962-10-31 01:02:03.457  170
```

Types `Y`, `H` and `J` allow the manipulation of time, date and timestamp values as scalar numeric values:

```
      3 1⊃SQA.Do 'C1' 'select * from testts :>Y,H,J:'
20110326 224540 40626.948379629626
```

`Y` allows dates to be represented as a single 8-digit integer in (`yyyymmdd`) format. `H` is the equivalent for time columns (`hhmmss`). Finally, `J` represents timestamps as “the number of days since 1900-01-01”. The fraction represents the time (22:45:20) as a fraction of a 24-hour day.

As all APL programmers will agree (ahem), “the number of days ***since*** 1900-01-01” means that the first of January 1900 will have the number zero. However, in order to make it easier for you to adapt to the various strange (but popular) date numbering systems actually adopted by users of other programming languages, the `J` data takes a modifier which follows a hash (`#`) sign. In `J#0` (the default), 1900-01-01 is day number 0. In `J#1`, it is day number 1. In `J#2`, 1900-01-01 is also day number 1, but from March 1st 1900 and onwards, the offset is 2. J#2 represents the numbering system used by Excel, also known as the “OLE Date Time”, in which February 29th 1900 is day number 60, even though 1900 was not a leap year.

```
      SQA.Do 'C1' 'create table testdt
                   (ch1 char(10), dt1 datetime)'
      ⎕←data←3 2⍴'Jan 1' '1900-01-01'
             'Feb28' '1900-02-28' 'Mar 1' '1900-03-01'
 Jan 1  1900-01-01
 Feb28  1900-02-28
 Mar 1  1900-03-01
```

```
      stmt←'insert into testdt values(:<C5:,:<C10:)'
      SQA.ExecDirect 'C1.I1' stmt data ('Bulk' 3)
0 3 3
      3 1⊃SQA.Do 'C1' 'select ch1,dt1,dt1,dt1,dt1,dt1
                  from testdt :>C10,S,J,J#0,J#1,J#2:'
 Jan 1  1900 1 1 0 0 0 0    0  0  1  1
 Feb28  1900 2 28 0 0 0 0  58 58 59 59
 Mar 1  1900 3 1 0 0 0 0   59 59 60 61
```

APL Arrays

The `Z` type allows the storage of entire APL arrays in any sufficiently wide character or binary column, by “serializing” the array. In the following example we shall use a *LongVarChar* column, although a *LongVarBinary* column would allow a more efficient and compact representation. It is important to specify a length which is large enough to accommodate the largest array that you are planning to serialize.

```
      SQA.Do 'C1' 'create table testarrays (array text)'
      stmt←'insert into testarrays values(:<Z500(L):)'
      SQA.Do 'C1' stmt (2 3ρι6)
0  C1.s3
      3 1⊃SQA.Do 'C1' 'select * from testarrays'
 ALASCII &     ("   "     P    $" P0%!@  =F[]"6V< ( !
      SQA.Do 'C1' 'select * from testarrays :>Z:'
0  C1.s3    1 2 3        6
            4 5 6
```

If we select the data without specifying the `Z` type, we see the character encoding that was used to store the array. With a *Binary* column, the `(L)` is not necessary. However, if you omit the `(L)` when using a *Char* column, any nulls in the binary representation will cause premature termination when reading data from the database, resulting in a ill-formed array representation, which can cause a crash of the SQAPL library.

The format used is known as Self-Contained Array (SCAR), has been published and placed in the public domain by Insight Systems, and is supported by APL systems from Dyalog, IBM and APL2000: Data is stored without loss or conversion in the internal format of the writing system, and is only converted if necessary by the recipient. This allows SQL databases to be used as an efficient storage mechanism for arrays, and for the exchange of APL arrays between different APL implementations.

Utility functions to convert arrays to and from SCAR are described in the *Advanced Topics* chapter.

Conversion Errors

If you use a bind variable type which the driver is unable to convert to the required type, the result will either be an error or a warning. Nowadays, such errors are fortunately quite rare, as drivers have become very good at type conversions. However, they can still occur in edge cases, and it is useful to know that an explicit type conversion can usually solve the problem. The following example was captured in 1990’s:

```
      SQA.Do 'C' 'insert into tbl values(:<C10:)' 'Text'
 4  07006 0  [Q+E Software][ODBC dBase driver]
             Unable to convert column 1 to SQL_CHAR.
             Error in parameter 1.
```

It turns out that this old dBase driver was expecting a *LongVarChar* and was unable to convert the provided *VarChar* to that. The solution was to declare it as `:<C10(L):`.

CHAPTER 4

Prepare, Exec, Fetch

Low Level Interface

So far, all our examples have used the "high level" interface to ODBC, `SQA.Do` – or the even higher level tools found in the `loaddata` workspace. The `SQA.Do` function performs a number of steps that are required by the ODBC interface: It will *prepare* and then *execute* the statement, *fetch* the results (possibly several times), and finally *close* the database cursor. It is very convenient for ad-hoc queries or prototyping, but gives you little control over the use of database or workspace resources. `SQA.Do` only returns the result of the last call to `SQA.Fetch`; if warnings were issued by earlier calls, they will not be reported. If the last fetch did return a warning, the cursor will often have been closed, so you will not be able to retrieve the warnings using `SQA.GetWarning`.

If you are writing an application which performs updates, needs to detect warnings, or needs control over the use of database cursors, it is probably worth learning to use the some of the following *Low Level* functions:

`SQA.Prepare`	Creates a new cursor object, and prepares an SQL statement for execution
`SQA.Describe`	Describes the input and output for a prepared statement
`SQA.Exec`	Binds values (if any) to a statement and executes it
`SQA.X`	Executes a statement multiple times using a matrix of input data
`SQA.ExecDirect`	Prepares and executes a statement in a single call
`SQA.Fetch`	Fetches one or more blocks of data
`SQA.Close`	Closes a cursor object
`SQA.Transact`	Commits or rolls back the current transaction
`SQA.Cancel`	Discards the current cursor context without destroying the cursor

The most common sequence of low level functions is the set that `SQA.Do` is using under the covers: `Prepare`, `Exec`, `Fetch`, `Close`. For example:

```
      SQA.Prepare 'C1.S1' 'select * from sqatest
                           where name like ''J%'''
0
      SQA.Exec 'C1.S1'
0 2
      0 SQA.Fetch'C1.S1'
0    Jill  1995-10-15  167        6
     Jack  1996-03-31  175
      SQA.Close 'C1.S1'
0
```

We will start by examining each of these functions in a bit more detail.

SQA.Prepare

The right argument of `SQA.Prepare` must contain a cursor object name followed by an SQL statement. The first segment of the cursor name must be the name of an existing connection. The right argument may contain additional elements in order to set options such as the maximum block size for each subsequent call to `SQA.Fetch`. For example:

```
      SQA.Prepare 'C1.S1' 'select * from sqatest where
        name like :<C1: and height > :>I:' ('MaxRows' 5)
0
```

The example above prepares a statement named `S1` attached to the connection `C1` and reserves buffer space for five records per fetch. The return code 0 indicates that the statement was successfully prepared.

A utility function `SQA.CursorName` can be used to generate a free cursor name, so that an application can avoid hard-coding these names:

```
      SQA.CursorName 'C1'
0  C1.s1
```

Variable Declarations, Revisited

If you use `SQA.Prepare` or `SQA.ExecDirect` (as we shall see in a moment), you can elect to use standard ODBC SQL syntax, in which each bind variable is denoted by a question mark, and provide type declarations in a separate 8-column numeric matrix:

To represent the two input variables of type `C10` and `I` (as in the first example in the previous section), we need:

```
      ⎕←BIND←2 8⍴(8↑1 1 1 0 10),8↑1 2 2
1 1 1 0 10 0 0 0
1 2 2 0  0 0 0 0
| | | |  | | | |
| | | |  | | | [;8]  Type Variant (see Handling Dates)
| | | |  | | [;7]  Partial Flag (see Partial Binding)
| | | |  | [;6]  Scale (number of digits following decimal point)
| | | |  [;5]  Precision (number of chars or digits)
| | | [;4]  ODBC Type Number to use (0 means use default)
| | [;3]  APL Type Number (1=C and 2=I, see table on next page)
| [;2]  Index into list of Bind or Select variables
[;1]  Direction (1=Bind or Input, 0=Select or Output)

      SQA.Prepare 'C1.S1' 'select * from sqatest
                  where name like ? or height > ?' BIND
0
      SQA.Exec 'C1.S1' 'B%' 170
0 3
      SQA.Fetch 'C1.S1'
0    Jack   1996-03-31  175       6
     Betty  1992-01-17  172
     Bobby  1994-10-31  180
      SQA.Close 'C1.S1'
0
```

A slight variant of the example used to demonstrate the use of variants of the J type for representing dates, which also uses a bind variable of type `J`:

```
      3 1⊃SQA.Do 'C1' 'select ch1,dt1,dt1,dt1,dt1,dt1
                  from testdt :>C10,S,J,J#0,J#1,J#2:
                  where dt1= :<J:' 59
```

… could be implemented as follows:

```
      info←⍪7↑1                    ⍝ 1 Bind, 6 Select
      info,←1,⍳6                   ⍝ Indices
      info,←12 1 9 12 12 12 12 ⍝ J C S J J J J
      info[6 7;8]←1 2              ⍝ #1 and #2
      info
1 1 12 0 0 0 0 0
0 1  1 0 0 0 0 0
0 2  9 0 0 0 0 0
0 3 12 0 0 0 0 0
0 4 12 0 0 0 0 0
0 5 12 0 0 0 0 1
0 6 12 0 0 0 0 2

      SQA.Prepare 'C1.S1' 'select ch1,dt1,dt1,dt1,dt1,dt1
                  from testdt where dt1=?' info
      SQA.Exec 'C1.S1' 59
      SQA.Fetch 'C1.S1'
0   Mar 1      1900 3 1 0 0 0 0   59 59 60 61
```

In the above example, it may seem a little cumbersome to use the matrix form, compared to the inline declarations. A utility function `SQA.Parse` is provided to help convert from the inline form to the matrix form:

```
      SQA.Parse 'select blah where xxx=:<C10: :>S,J#2:'
 select blah where xxx=?   1 1  1 0 10 0 0 0
                           0 1  9 0  0 0 0 0
                           0 2 12 0  0 0 0 2
```

If you are generating SQL statements using some kind of data dictionary, the matrix form can be easier to create under program control. Also, when using the inline form, if you want to declare an output type for the 3rd output column, you must be able to provide a name which exactly matches the name that the database gives to the column. This can be tricky with calculated or joined columns. The matrix form avoids these issues.

```
      SEL←1 8⍴8↑0 2 12 ⍝ Output, 2nd col, Type J
      SQA.Prepare 'C1.S1' 'select * from sqatest' SEL
      SQA.Exec 'C1.S1'
      2⊃SQA.Fetch 'C1.S1'
  Jill   34985 167
  Jack   35153 175
  Betty  33618 172
  Bobby  34636 180
```

SQA.Exec and SQA.X

Once a statement has been prepared, the next step is to execute the statement using `SQA.Exec` or `SQA.X`. We have already seen an example of the use of both in the previous section on bind variables. While `SQA.Exec` allows a single execution of a statement, `SQA.X` accepts a matrix of bind values and will cause the statement to be executed once for each input row. If the ODBC driver supports bulk input, the data will be transferred to the server in a single operation and the looping will be done in the database server, providing very high performance on large data volumes. If the driver does not support this, SQAPL will do the looping and there will be a smaller performance gain.

Use of SQA.X requires the use of the Bulk when preparing the statement. This option specifies the largest number of rows or sets of bind variables you intend to provide, so that SQAPL knows how much buffer space to set aside. In the following example, we will prepare a statement once and then use it with both SQA.X and SQA.Exec:. In both cases, the argument is the name of the prepared statement, and bind variable values. For SQA.Exec, the values follow on after the name as district arguments, in the case of SQA.X a single matrix provides data for all:

First, we extract all records from our existing table, delete them from the table, and immediately re-insert them using a single call to SQA.X:

```
      ⎕←data←3 1⊃SQA.Do 'C1' 'select * from sqatest
                                 :>C10,J,I:'
 Jill   34985 167
 Jack   35153 175
 Betty  33618 172
 Bobby  34636 180
      SQA.Do 'C1' 'delete from sqatest'
      SQA.Prepare 'C1.I1' 'insert into sqatest
              values (:n<C10:,:d<J:,:h<I:)' ('Bulk' 10)
      SQA.X 'C1.I1' data
0 4 4
```

Unfortunately, the result of SQA.X will vary from one driver to the next, but in theory if should mean that 4 database rows were affected and 4 input rows were processed. Regrettably, you cannot rely on these numbers to determine whether the execution was completely successful, and you must inspect the return code.

We can proceed to add one more row using SQA.Exec:

```
      SQA.Exec 'C1.I1' 'Fred' 35000 185
0 1
```

In this case, the second element of the result is the number of rows modified by the execution of the statement – and this is fairly reliable.

Nulls

SQL databases allow a value to be set to *null*, to indicate that it is unknown. If we have another person to add to the database, but her height is unknown, we can enter the height as null rather than simply setting it to zero. We can provide null flags by following the bind variable values with an equal number of boolean elements, where 1's indicate nulls. You still have to provide a placeholder value (in this case 0) for the columns which will be nulled:

```
      SQA.Exec 'C1.I1' 'Fiona' 35100 0 0 0 1
0 1
```

And we can add

```
      ⎕←data←2 3⍴'Harry' 35234 0 ,'Hermione' 34987 168
 Harry     35234   0
 Hermione  34987 168
      SQA.X 'C1.I1' data (2 3⍴0 0 1, 0 0 0)
0 2 2
```

In SQL databases, nulls are considered to be different from all other values, including other nulls – which makes them hard to find:

```
      stmt←'select * from sqatest where height=:<I:'
      ⍴3 1⊃SQA.Do 'C1' stmt 0   ⍝ Height=0
0 3
      ⍴3 1⊃SQA.Do 'C1' stmt 0 1 ⍝ Height=[null]
0 3
```

In fact, a special SQL keyword is used to find null values:

```
      3 1⊃SQA.Do'C1' 'select * from sqatest
                              where height is null'
 Fiona  1996-02-07  0
 Harry  1996-06-20  0
```

Note that null indicators are not retrieved by default, so the output of a query can be deceptive: Null indicators can be retrieved using `SQA.Fetch`. Strictly speaking, the value returned for elements which are null is undefined, but in practice it is usually `0` for numeric columns and `''` for characters.

Unfortunately, although the SQL standard is quite explicit about the treatment of nulls, few databases follow the standard exactly. A few databases ***do*** allow you to find nulls by binding a null value to a search parameter, although this violates the standard. Other interesting questions include: How many groups should you get if you do a GROUP BY on a column with multiple null values? Can you JOIN tables on nulls in your database? Writing portable applications which use nulls requires much testing.

Errors During Statement Execution

If the execution of a statement fails when using `SQA.Exec`, the error will either be returned as the result of the function, but it may also be returned as a warning:

```
      SQA.Do 'C1' 'create unique index uname
                                   on sqatest(name)'
      SQA.Exec 'C1.I1' 'Fred' 35000 185
¯1 0
      SQA.GetWarning 'C1.I1'
0   4   23000  1062  [MySQL][ODBC 5.1 Driver][mysqld-
5.1.41-3ubuntu12.10]Duplicate entry 'Fred' for key
'uname'  0
```

If we use `SQA.X`, more than one error may be produced, and the 2nd element of the result of `SQA.GetWarning` may contain more than one error:

```
      data
 Harry     35234   0
 Hermione  34987 168
      SQA.X 'C1.I1' data
¯1 0 2
      ρt←2⊃SQA.GetWarning 'C1.I1'
1
      1 2 3 4,⍪⍕¨1⊃t
1  4
2   23000  1062
3  [MySQL][ODBC 5.1 Driver][mysqld-5.1.41-
3ubuntu12.10]Duplicate entry 'Hermione' for key 'uname'
4  0
```

Unfortunately, drivers handle errors in different ways when using `SQA.X`. The result of `SQA.X` indicated that 2 records were processed, but only one message was returned. The 4th element, which is supposed to be the index of the input row that caused the error, is `0`. Other database drivers would have returned two errors.

It is possible to get more reliable behaviour by instructing SQAPL not to use the ability of the driver to loop (`'Loop' 0`), and stop on the first error and report it as an error rather than a warning using (`'StopOnError' 1`), but this may significantly reduce performance:

```
      SQA.Prepare 'C1.I1'
        'insert into sqatest values (:<C10:,:<J:,:<I:)'
                 ('Bulk' 10)('Loop' 0)('StopOnError' 1)
0
      SQA.X 'C1.I1' (('Polly' 35555 175)⍪data)
4   23000  1062  [MySQL][ODBC 5.1 Driver][mysqld-5.1.41-
3ubuntu12.10]Duplicate entry 'Harry' for key 'uname'  1
```

In this case, the row number will be the 0-origin index into the input matrix.

SQA.Describe

The function `SQA.Describe` allows you to retrieve a description of any SQAPL object. At this point, we will only show how to use it in conjunction with prepared statements, but the function reference explains how to use it on other objects.

```
      SQA.Prepare 'C5.S2' 'select name,height,:dob>J:
             from sqatest where name like :wname<C10:
             or height>:wheight<I:'
0
      SQA.Exec 'C5.S2' 'J%' 170
      SQA.Fetch 'C5.S2'
…data…
      SQA.Describe 'C5.S2'
0   Cursor C5.S2: Status=Fetch Completed, MaxRows=50

    select name,height,:dob>J: from sqatest where name
like :wname<C10: or height>:wheight<I:

    Bind Vars: 2
     Name     APL   DB   Prec.   Scale  Length    Buf
     wname    C      0     10       0      22 W   U
     wheight  I      0      0       0       4 I   I

    Select vars: 3
     Name     APL   DB   Prec.   Scale  Length    Buf
     name     C      1     10       0      11 C   C
     height   I      4     10       0       4 I   I
     dob      J     93     23       3      16 S   S
```

With some database drivers, information is available after Prepare. In other cases, some or all of the information does not become available until the statement has been executed, and in some cases only after the first fetch.

When used without a left argument, or with a left argument of 1, `SQA.Describe` provides a formatted report for the SQAPL object named in its right argument. The most common use is to describe a cursor, as in the example above. The first two columns of output give you the *Name* and the *APL* type of each variable. The *DB* type column gives the ODBC type. *Precision* is the number of significant digits in a numeric field, or the length of a character field. *Scale* is the number of digits to the right of the decimal point, where applicable. The *Length* column shows the number of bytes consumed for this column in each row of output. The *Int* and *Buf* columns give internal *logical* and C *buffer* types.

With a left argument of 0, the second element of the result of `SQA.Describe` is an APL array containing unformatted information for the named object and all of its children all the way down the object hierarchy. Each item is a two-element vector, where the first element contains information about the object itself, and the second element is a vector containing the result of `SQA.Describe` applied to each child (without return codes). For details, see `SQA.Describe` in the Function Reference section.

SQA.ExecDirect

You can prepare and execute a statement in a single call using this “new” function (introduced ca. 2005):

```
      SQA.Do 'C1' 'create table tenkrows
                        (col1 char(10),col2 integer)'
      ⍴data←(↓'ZI10' ⎕FMT ⍪⍳10000),⍪⍳10000
10000 2
      stmt←'insert into tenkrows values(:c1<C10:,:c2<I:)'
      SQA.ExecDirect'C1.I2' stmt data ('Bulk' 10000)
0 10000 10000
```

… or using a matrix to declare the bind variables:

```
      stmt←'insert into tenkrows values (?,?)'
      bind←2 8⍴(8↑1 1 1 0 10),8↑1 2 2
      SQA.ExecDirect'C1.I2' stmt data bind ('Bulk' 10000)
0 10000 10000
```

Note that, since the function essentially combines `SQA.Prepare` and `SQA.Exec`, all options for these two functions are valid as options of `SQA.ExecDirect` (with the exception that there is no support for Nulls. Also note that following the call, the cursor is open - even if an error was returned. You are expected to call `SQA.Fetch` to retrieve results (if any), and finally `SQA.Close`.

SQA.Fetch

After a statement has been executed using either `SQA.Exec`, `SQA.X` or `SQA.ExecDirect`, `SQA.Fetch` is used to retrieve the output.

```
      stmt←'select * from sqatest :>C10,J,I:'
      SQA.Prepare 'C1.S1' stmt ('MaxRows' 3)
      SQA.Exec 'C1.S1'
      0 SQA.Fetch 'C1.S1'
0    Jill   34985 167       5
     Jack   35153 175
     Betty  33618 172
```

When called with no left argument, or a left argument of 1, `SQAFetch` loops until all data is returned and the cursor is closed. Above, we have used a left argument of zero to request that `SQAFetch` only fetch one block of data, which contains the 3 rows requested through the use of the `MaxRows` parameter when preparing the statement.

Fetching Nulls

The `Nulls` option allows us to request that the 3rd element of the result (which is normally empty) be populated with null indicators in the form of a boolean array with the same shape as the data:

```
      0 SQA.Fetch 'C1.S1' ('Nulls' 1)
0    Bobby  34636 180    0 0 0    5
     Fred   35000 185    0 0 0
     Fiona  35100   0    0 0 1
```

Optimising Fetches

By default, SQA.Fetch returns all columns of output in a single nested matrix. This is a relatively inefficient format, both in terms of the memory required to store the result, and the processing time required to move the individual elements of this array.

The `ColumnWise` option can be used to request that each column be returned as a separate simple array:

```
         ]disp 2⊃0 SQA.Fetch 'C1.S1' ('ColumnWise' 1)
┌→────────┬─────────────────┬─────────┐
│Harry    │                 │         │
│Hermione │35234 34987 35555│0 168 175│
│Polly    ↓                 │         │
└────────→┴~───────────────→┴~───────→┘
```

Each element of the result is now a simple array, including character columns. Depending on your data, you may observe significant speedups.

If the application needs character data to be returned as a vector of character vectors, the number of bytes used to store the each element can be combined with null indications, using `('Nulls' 3)`:

```
     ]disp ⍪0 SQA.Fetch 'C1.S1'('ColumnWise' 1)('Nulls' 3)
┌→───────────────────────────────────────┐
↓                   0                    │
├~───────────────────────────────────────┤
│┌→────────┬─────────────────┬─────────┐ │
││Bobby    │                 │         │ │
││Fred     │34636 35000 35100│180 185 0│ │
││Fiona    ↓                 │         │ │
│└────────→┴~───────────────→┴~───────→┘ │
├───────────────────────────────────────→┤
│        ┌→──────┬────────┬──────┐       │
│        │10 8 10│16 16 16│4 4 ¯1│       │
│        └~─────→┴~──────→┴~────→┘       │
├───────────────────────────────────────→┤
│                   5                    │
└~───────────────────────────────────────┘
```

In this case, the null flags are `¯1` for null values, and otherwise the number of bytes used.

Note: `SQA.Connect` also supports the `Columnwise` option, allowing you to set a default for the connection. For example:

```
      SQA.Connect 'C1' 'MySQL' 'secret' 'userid'
                  ('MaxRows' 1000)('Columnwise' 1)
```

If the option is set on the connection, it will also affect the shape of the data returned by `SQA.Do`.

SQA.Cancel

The function `SQA.Cancel` is used to release all resources used by a cursor *without* closing it. For example, you could use this function on a cursor to release it before you have fetched all records.

```
     SQA.Cancel 'C1.S1'
0
```

Unfortunately, this function does not have the same effect with all drivers. Some drivers will not allow you to re-execute a cancelled statement. SQAPL will solve some of these problems, for example by re-preparing the statement if the driver returns ODBC Sequence Error on execution of a cancelled statement. The only completely portable strategy is to close the cursor, and re-prepare the statement yourself if you need it again.

SQA.Close

It is a good idea to close objects that you no longer need, so that they do not keep files open, tables locked, or reserve other server resources:

```
SQA.Close 'C1.S1' ⍝ Close a cursor
SQA.Close 'C1'    ⍝ Connection and all related cursors
SQA.Close '.'     ⍝ All connections and reloads Library
```

CHAPTER 5

Advanced Features

Large Objects and Partial Binding

SQAPL itself imposes no limitations on record or data element size. The only limits result from workspace size and other memory limitations, plus limitations in the drivers or databases themselves. These limitations are probably fading as everyone moves towards 64-bit operating systems, but there are still situations where it is impractical or undesirable to pass all data in a single function call. ODBC supports very large data elements in inputs (bind variables) and results using a protocol known as "Partial Binding".

To support partial binding, SQAPL provides two functions called `SQA.GetData` and `SQA.PutData`, and a special modifier in bind variable declarations.

SQA.PutData

Assume that we have created a table as follows (this example was done using Microsoft Access):

```
SQA.Do 'C1' 'create table persons
   (id counter, ename text(50), fname text(50),
   birth datetime,comment longtext, extra longtext)'
```

The last two fields have the ODBC type *LongVarChar*, which can hold up to 64k of text in each element. If we do not feel that we can safely provide all this information at once, we can use partial binding to provide the values. This is done using a trailing P for Partial in the bind variable declaration (when using the matrix form, put a 1 in column `[;7]` to indicate a partly bound variable.

```
SQA.Prepare'C1.I1' 'insert into persons values
          (:<C50:,:<C50:,:<Y:,
           :comment<C65536(L)P:,:extra<C65536(L)P:)'
```

The two type declarations provide SQAPL with the following information:

`comment`	The name of the bind variable
`<C65536`	It is an input variable and we will provide up to 64k elements of character data
`(L)`	We want SQAPL to convert the data to *LongVarChar*
`P`	We will use partial binding

When we execute the statement, we supply values for the fully bound variables using a normal call to `SQA.Exec`. For each partially bound variable, we provide an integer which indicates the maximum size of a block of data that we might use when we provide the data:

```
      SQA.Exec 'C1.I3' 'Clinton' 'Bill' 19500616 500 500
0 ¯1  C1.I3.comment
```

SQA.Exec returns a zero return code to let us know that all is well, that ¯1 rows have been modified (indicating that the work has not yet been done). The third element of the result contains the name of the first partial buffer for which data is expected (using the name that we provided in the declaration).

We are now expected to make repeated calls to SQA.PutData for the first partial variable.

```
      SQA.PutData 'C1.I3.comment'(100⍴'USA ')0
0 0  C1.I3.comment
      SQA.PutData 'C1.I3.comment'(200⍴'USA ')0
0 0  C1.I3.comment
      SQA.PutData 'C1.I3.comment'(150⍴'USA ')1
0 ¯1  C1.I3.extra
```

The arguments to SQA.PutData are the name of the buffer, the next block of data, and a flag which allows us to declare whether we are done with this variable. When we declare that we are done, SQAPL tells us that there is more data to supply for the buffer called C1.I3.extra. We continue providing data:

```
      SQA.PutData'C1.I3.extra'(100⍴'USA ')0
0 0  C1.I3.extra
      SQA.PutData'C1.I3.extra'(200⍴'USA ')0
0 0  C1.I3.extra
      SQA.PutData'C1.I3.extra'(150⍴'USA ')1
S1000  ¯1302  [Microsoft][ODBC Microsoft Access 97
Driver] Table 'persons' is exclusively locked by user
'Admin' on machine 'GOLLUM'.
```

Apart from revealing that this demo is rather old, we can see that the author was careless and had the table in question open in Microsoft Access while creating the example. After shutting Access down and repeating the experiment, the final call completes successfully:

```
…all over again…
      SQA.PutData'C1.I3.extra'(150⍴'USA ')1
0 0
```

SQA.GetData

The procedure for reading data using partial binding is almost identical to providing input values, in reverse:

```
      SQA.Prepare 'C1.S1' 'select id, ename, fname,
birth, :comment>C15P:, :extra>C160P: from persons'
```

Again, the trailing P in the output variable declarations are the clue that lets us know that these variables will be fetched "partially". When partial fetching is in use, MaxRows is automatically set to 1 – you cannot combine partial fetching with blocking records on output.

```
      SQA.Exec 'C1.S1'
0 0
      0 SQA.Fetch 'C1.S1'
0   1  Lennon  John  1950-07-18 00:00:00       5
```

You must use `SQA.Fetch` with a left argument of zero, so that only one block (containing one record) is fetched. There is no indication that partial data is available; it is our responsibility to pick up partial data using `SQA.GetData` after fetching each record. If we continue fetching SQAPL and ODBC will allow us to proceed without protests.

```
      SQA.GetData 'C1.S1.comment'
¯1  This is one of   ¯11 5
      SQA.GetData 'C1.S1.comment'
0  the Beatles  0 5
      SQA.GetData 'C1.S1.extra'
0  He is playing the guitar  0 5
```

The first element of the result is 0 if all is well, `¯1` if there was a warning, and a positive number if there was an error (as for any SQAPL function). The second element contains the data, and the third element is the number of elements of data which have not yet been fetched. The fourth element is the state of the entire fetch operation, and 5 means there are more records available. Once we have fetched all the partial data we want for the current record, we can proceed to the next one – until we are done:

```
      0 SQAFetch 'C.S1'
0   2  Star  Ringo  1945-09-04 00:00:00        5
```

SCAR Conversion Functions

In order to make it possible to read and write APL objects using partial binding and fetching, there are two functions which convert APL objects to the Self-Contained Array (SCAR) format which is used to represent arrays passed using APL type `Z` - and back again. Of course, the functions can also be used to convert APL arrays to the SCAR format and write them to other types of files, or transfer them across sockets to other APL systems.

The functions are `SQA.APL2Scar` and `SQA.Scar2Apl`

```
      z←SQA.Apl2Scar (⍳3) 0
```

The first element of the argument is the array to be converted, the second element is 1 if the result must use printable ASCII characters only, 0 if full 8-bit representation may be used. You can only use the latter if the database column you are going to store the data in is Binary or Long Binary.

```
      8↑2⊃z
BLASCII
      ⎕UCS 8↓2⊃SQA.Apl2Scar (⍳3) 0
16 0 0 0 2 1 0 0 3 0 0 0 1 2 3 0
```

The first eight bytes of the SCAR representation identify the encoding used: The first byte is A for ASCII Printable (can be saved in a Char column) or B for Binary. The second byte is L for Little-Endian (Intel) byte order or B for Big-Endian. The remaining 6 bytes are the name of the translate table (entry in APLUNICD.INI) used, in order to provide translation of characters to the alphabet used by the decoding system. For Unicode systems, the translate table is called “ASCII”, which is unfortunately a little misleading. Following the 8-byte header is the data, where data values are stored in the internal format used by the interpreter that created the SCAR.

To convert the data back to an APL array, pass the SCAR to the function `SQA.Scar2Apl`:

```
      SQAScar2Apl 2⊃z
0  1 2 3
```

Transactions

Most multi-user SQL databases support the concept of a *transaction*. This allows an application to ensure that a group of related changes (the *transaction*) is performed as a unit, or not at all.

However, the default for most ODBC drivers is to operate in “autocommit” mode, where every individual modification to a database table is committed immediately. You may be able to turn this off using (`'OdbcAutoCommit' 0`) as a parameter to `SQA.Connect`, if this does not work you will need to read the documentation for your database and ODBC driver to find out how to use transactions.

Once transactions are enabled, the first transaction starts when a connection is created. It lasts until you call the function `SQA.Transact` and decide whether the changes performed should be *committed* (made permanent) or *rolled back* (discarded). The call to `SQA.Transact` also triggers the start of the next transaction.

The function `SQA.Transact` takes a connection name and a flag which is 0 in order to commit or 1 to do a roll back.

```
     SQA.Prepare 'C1.U2' 'update emp
         set   hourly_rate = :newrate<F:
         where name        = :name:'
0
     SQA.Exec 'C1.U2' 36 'Randall, David'
0 1
     SQA.Exec 'C1.U2' 37 'Moore, Holly'
0 1
     SQA.Transact 'C1' 0
0
```

Closing a connection causes a roll back, so changes will normally be discarded if the connection is lost or terminated during a transaction. However, closing a cursor does ***not*** affect the changes performed using the cursor; they are still part of the current unit of recovery.

Transaction Portability Issues

Some databases do not support transactions, and it is common for systems to configured to *auto-commit*. This is often done in order to improve performance. Some databases also support statements like “BEGIN TRANSACTION” and “COMMIT TRANSACTION”, which you can execute as an alternative to using `SQA.Transact`. You should not mix the use of `SQA.Transact` with BEGIN TRANSACTION statements!

Check with your database administrator to find out whether your database and driver support transactions (if you are the database administrator, read the documentation ☺)!

Two-phase Commit

SQAPL does not let you include changes made through two different *connections* in a single unit of recovery. The ability to support *distributed* units of recovery is sometimes referred to as *two-phase commit* in SQL DBMS terminology. If the database manager you are using has a distributed architecture, you may in fact be using more than one physical database through one connection. If the DBMS itself supports two-phase commit, you will be able to make use of this capability with SQAPL - but this will not be apparent to you (that's the whole point).

Releasing Cursor Resources

Relational database managers use a system of locks to ensure data integrity when accessed by several users simultaneously. Unfortunately, relational database systems behave quite differently when acquiring and releasing these locks. Some acquire locks when a statement is prepared, others wait until the statement is executed. Some systems lock entire tables while others are capable of locking single rows or items of data. Some automatically release locks when you have fetched all data, others require you to explicitly cancel or close the cursor in order for locks to be released. Some databases purge all prepared statements when a commit or rollback is performed. You or the system administrator may be able to set options which change the behaviour of the DBMS.

If you want to write portable applications, you must either learn the behaviour of each class of database and code alternatives depending on the type, or write "defensive" code which always assumes the worst and explicitly closes cursors if there is any doubt as to how a database will react. Compared to reusing objects when possible, this strategy may give a significant reduction in performance, especially when using drivers provide sophisticated "cursor caching" mechanisms.

Remote Procedure Calls

If you are going to develop applications which perform non-trivial transactions, you will probably need to make use of more sophisticated techniques than *commit* and *rollback*. Many modern SQL DBMS systems provide additional transaction processing facilities, which are unfortunately quite different from one vendor to the next. Common names for these facilities are *stored procedures* or *triggers*. These facilities make it possible to have the DBMS system execute multiple SQL statements under control of a *procedure*. The execution of such procedures can be initiated using a statement, or be *triggered* by the update of a particular table.

Almost all databases declare the output of normal SELECT statements, but some do not describe the output of a stored procedure. SQAPL allows you to declare the expected output of a statement and thus extract results from a driver even if the driver does not provide the information.

Buffers

Data for bind and select variables is stored in *buffers*. SQAPL allocates buffers when a statement is prepared, and releases them when the cursor is closed. The size of each buffer depends on the data type and the number of records which SQAPL is asked to fetch in each call to `SQA.Fetch`.

You can limit the number of records for which SQAPL allocates buffers using the `MaxRows` parameter. A high number should increase performance by reducing the number of times you need to execute `SQA.Fetch`, but it also increases the amount of buffer space which is required. In some environments, buffer space may be a critical resource, so you may need to be careful. A SELECT statement with a large number of output columns can easily consume many megabytes of buffer space if you get enough for thousands of records at a time. As a rule of thumb, the performance improvement is marginal once you exceed a few hundred records.

CHAPTER 6

Troubleshooting Guide

Common Error Codes

SQAPL uses numeric error codes which can be looked up in the appendix titled Errors and Warnings. This section discusses some of the most commonly occurring return codes.

No Error Messages

If you are not seeing any textual error messages, this is because SQAPL can't find the error message file. Check that you have a file named `sqapl.err` in the same folder as the SQAPL library (`cndya61uni.dll` for a version 13.0 Unicode system). If you don't have this file, you will only get numeric error messages, for example:

```
      SQA.Exec 'C1.NoSuchCursor'
1 10010     0
```

Instead of:

```
1 10010  OBJ Object not found  0
```

¯1 (Warning)

This does not generally indicate an error. For example, `SQA.Connect` almost always gives a return code of `¯1` when connecting to Microsoft SQL Server, but this is just in order to provide information. At other times, it does indicate a real problem: `SQA.X` will return `¯1` even when fatal errors have occurred, so that you can call `SQA.GetWarning` to retrieve the relevant messages from multiple statement executions and decide whether further action needs to be taken:

```
      SQA.Connect 'C2' 'CRMIS'
¯1
      SQA.GetWarning 'C2'
0   4   01000  5701  [Microsoft][ODBC SQL Server
Driver][SQL Server]Changed database context to 'CRM'.
0   4   01000  5703  [Microsoft][ODBC SQL Server
Driver][SQL Server]Changed language setting to
us_english.
```

10001 APL Length Error

Typically happens when the argument to an SQAPL function has the wrong number of arguments:

```
       SQA.Prepare 'C2.I1' 'insert into sqatest values
                            (:<C10:,:<J:,:<I:)'
0
      SQA.Exec 'C2.I1' 'Fred' 35123 ⍝ Only 2 bind vars
1 10001  APL Length Error  0
```

10002 APL Rank Error

An argument has the wrong rank. Below, SQA.X is expecting a matrix of bind values:

```
      SQA.X 'C1.I1' (⍳6)
1 10002  APL Rank Error  0
```

10007 OBJ Parent Not Found

This error often happens when you use the name of a connection which has been closed. For example:

```
      SQA.Prepare 'C1.S1' 'select * from sqatest'
1 10007  OBJ Parent not found  0
```

10009 OBJ Name already in use

This error often happens when you have forgotten that the name of a connection or cursor is still in use:

```
      SQA.Prepare 'C1.S1' 'select * from sqatest'
1 10009  OBJ Name already in use  0
      SQA.Close 'C1.S1'
0
      SQA.Prepare 'C1.S1' 'select * from sqatest'
0
```

10010 OBJ Object Not Found

You are trying to use an object that has been closed:

```
      SQA.Prepare 'C2.I1' 'insert into sqatest values
                           (:<C10:,:<J:,:<I:)'
0
      SQA.Exec 'C1.S1'
1 10010  OBJ Object not found  0
```

10081 APL Length error

This error is issued when a bind variable value has an unexpected length – usually because it is too long:

```
      SQA.Prepare 'C2.I1' 'insert into sqatest values
                           (:<C10:,:<J:,:<I:)'
0
      SQA.Exec 'C2.I1' 'MuchTooLongName'
1 10081  APL Length error  0
```

10084 APL Domain error

A bind variable has the wrong type:

```
      SQA.Prepare 'C2.I1' 'insert into sqatest values
                           (:<C10:,:<J:,:<I:)'
0
      SQA.Exec 'C2.I1' 'Fred' 35123 170.5
1 10084  APL Domain error  0
```

The last bound value is floating-point, but needs to be integer.

10086 APL Index error

For “implementation reasons”, this error message is occasionally given when “rank error” would be more natural.

```
      SQA.Prepare 'C2.I1' 'insert into sqatest values
                           (:<C10:,:<J:,:<I:)'
0
      SQA.Exec 'C1.I1' 'Fred' '1995-03-26' 170
1 10086  APL Index error  0
```

Exception Handling

SQAPL traps all exceptions occurring within the SQAPL C code, or the ODBC driver that it is calling. In the event of a failure in or below the SQAPL library, you will see something like the following:

```
[SQAPL] Exception c0000005 at 642574b Memory read at 10
ExecDirect[3] r←∆SQAPL'ExecDirect'y
             ^
```

The event number signalled is 11 (DOMAIN ERROR). You can trap this error in order to prevent your application from crashing. However, we do not recommend that you attempt to automate recovery by using error trapping - because the internal state of SQAPL and any ODBC drivers you have been using is unknown following an exception of this kind. If you must restart, you should at least unload and reload the SQAPL library using (`SQA.Close'.'`) followed by a new call to `SQA.Init`.

SQA.GetInfo

The function `SQA.GetInfo` makes it possible to ask a driver to provide information regarding the capabilities of the data source, and optional settings. The argument to `SQA.GetInfo` is a connection name followed by a vector of integer information codes. The list of valid arguments can be found in the ODBC C header file `sqlext.h`, which can be easily found by searching the internet. For example, a copy can be found at:

http://www.opensource.apple.com/source/iodbc/iodbc-36/iodbc/include/sqlext.h

The definitions can be found following comments containing the name SQLGetInfo in the file. Unfortunately, you will need to read the ODBC reference documentation to know what they all mean – but a few are easy to guess. The first useful snippet begins with:

```
/*
 *  SQLGetInfo - ODBC 2.x extensions to the X/Open
standard
 */
#define SQL_INFO_FIRST                          0
#define SQL_ACTIVE_CONNECTIONS                  0 /*
MAX_DRIVER_CONNECTIONS */
#define SQL_ACTIVE_STATEMENTS                   1 /*
MAX_CONCURRENT_ACTIVITIES */
#define SQL_DRIVER_HDBC                         3
#define SQL_DRIVER_HENV                         4
#define SQL_DRIVER_HSTMT                        5
#define SQL_DRIVER_NAME                         6
#define SQL_DRIVER_VER                          7
#define SQL_ODBC_API_CONFORMANCE                9
#define SQL_ODBC_VER                            10
#define SQL_ROW_UPDATES                         11
#...etc...
```

We can find out the name and version of the current ODBC driver, the ODBC version that is supported, and whether it supports “Row Updates” as follows:

```
      SQA.GetInfo 'C1' (6 7 10 11)
0   myodbc5.dll  05.01.0008  03.80.0000  N
```

SQA.NativeSQL

Under some circumstances, an ODBC driver will make slight changes to SQL statements before they are submitted to the database engine. This typically happens when the database is using a slightly non-standard dialect of SQL. If you need to know exactly what the “native” SQL statement is, you can retrieve it using this function. For example:

```
      SQA.Prepare 'C1.S1' 'select * from datedemo'
      2⊃SQA.NativeSQL 'C1.S1'
SELECT *

FROM datedemo;
```

The result is a character vector. The above example used an early version of Microsoft Access; the “Access SQL” statement contains embedded carriage returns. You are in dire straits if this is really useful to you as a diagnostic tool, hopefully you will only ever use it to satisfy your curiosity.

Function Reference

Naming Conventions

All functions in the SQAPL workspace which have names beginning with an uppercase letter are public functions that should be usable from applications. Every attempt will be made to ensure "upwards compatibility" of these functions from one release to the next.

Any functions with lowercase names, or names beginning with the symbol ⍙ should not be used. These functions are for internal use by SQAPL, and may change without notice.

Function Descriptions

The following pages describe each documented SQAPL application function; its syntax, arguments and results.

Unless otherwise noted, the results described in the following are the results returned by **successful** calls to the functions in question . In the event of errors, **all** functions return the same form of result, described in the *Errors and Warnings* section.

SQAPL Quick Reference Card

`rc scar`	`←`		`Apl2Scar`	`array ascii`
`rc`	`←`		`BrowseConnect`	`con`
`rc`	`←`		`Cancel`	`cur`
`rc`	`←`		`Close`	`obj`
`rc data`	`←`		`Columns`	`con [tbl col qua own]`
`rc`	`←`		`Connect`	`con dsn [pwd usr]`
`rc cur`	`←`		`CursorName`	`con`
`rc data`	`←`	`[fmt]`	`Describe`	`obj`
`rc cur data nul`	`←`	`[all]`	`Do`	`con stmt [bind]`
`rc rows`	`←`		`Exec`	`cur [val[,nul]]`
`rc rows`	`←`		`ExecDirect`	`cur stmt [val vars]`
`rc data nul`	`←`	`[all]`	`Fetch`	`cur`
`rc data`	`←`		`GetInfo`	`con infodefs`
`rc msgs`	`←`		`GetWarning`	`obj`
`rc`	`←`		`Init`	`inifile`
`rc stmt`	`←`		`NativeSQL`	`cur`
`stmt vars`	`←`		`Parse`	`stmt`
`rc`	`←`		`Prepare`	`name sql [vars]`
`rc array`	`←`		`Scar2Apl`	`scar`
`rc data`	`←`		`Tables`	`con [tbl typ qua own]`
`rc`	`←`		`Transact`	`con state`
`rc tree`	`←`		`Tree`	`obj`
`rc`	`←`		`TypeInfo`	`con`
`rc rows rows`	`←`		`X`	`cur [bind] [nul]`

`array`	Any APL array	`own`	Owner name (or prefix%)
`ascii`	1 for Ascii, 0 for Binary	`pwd`	Logon password
`all`	1 to fetch all data	`qua`	Qualifier (or prefix%)
`con`	Connection name	`rc`	Return code
`col`	Column name (or prefix%)	`rows`	Row count
`cur`	Cursor name	`scar`	Self-contained array
`data`	Output values	`state`	0=commit, 1=rollback
`dsn`	ODBC Data Source Name	`tbl`	Table name (or prefix%)
`fmt`	1 to format, 0 for raw data	`tree`	Nested object tree
`infodefs`	Numeric ids from sqlext.h	`typ`	Type (TABLE or VIEW)
`inifile`	Name of sqapl.ini file or `''`	`vars`	Bind and Select definitions
`msgs`	vector of 4-element vecs	`val`	Bind variable values
`nul`	Null matrix or empty vector	`usr`	Logon user id
`obj`	`'.'`, connection or cursor		

In addition, many functions will accept options in the form of (name value) pairs.

SQA.Apl2Scar

Purpose:	“Serialize” any APL array to a character vector, for storage in a character or binary column.
Syntax:	`rc scar ← SQA.Apl2Scar array ascii`

`rc`	0
`scar`	A character vector
`array`	Any APL array
`ascii`	1 to create a SCAR using printable ASCII chars only (suitable for a Char field), or 1 to use 8-bit binary.

See also Scar2Apl.

Example:

```
      SQA.Apl2Scar (⍳3) 1
0  ALASCII $     (!   #     0(#             !
```

SQA.BrowseConnect

Purpose:	Browse available data sources and connect interactively. This functionality is only available under Microsoft Windows.
Syntax:	`rc ← SQA.BrowseConnect con`

`rc`	0
`con`	Name of a connection (which is not already in use).

Example:

```
      SQA.BrowseConnect 'C1'
0
```

SQA.Cancel

Purpose:	Deactivate a cursor (rolls changes back if the driver is capable of this, and releases any resources that it might be holding).
Syntax:	`rc ← SQA.Cancel cur`

`rc`	0
`cur`	A cursor name

Example:

```
      SQA.Cancel 'C1.S1'
0
```

SQA.Close

Purpose:	Close an SQAPL object.	
Syntax:	`rc ← SQA.Close cur|con|root`	
	`rc`	0
	`cur|con|root`	Name of a cursor, a connection, or the root object.

Example:

```
      SQA.Close 'C1.S1'
0
```

SQA.Columns

Purpose:	List names and types of columns contained by a table or view	
Syntax:	`rc data ← SQA.Columns con [tbl col qua own]`	
	`rc`	0
	`data`	Output matrix – see Appendix B for details.
	`con`	Name of the connection.
	`tbl`	Table name
	`col`	Column name
	`qua`	Qualifier
	`own`	Owner

The last four arguments are optional, and used to filter the result. You can use SQL wildcards, for example values like `'T%'` to find all values starting with T.

Example:

```
         (⍳18),⍉2⊃SQA.Columns 'C1' 'sqatest'
 1   TABLE_CAT
 2   TABLE_SCHEM
 3   TABLE_NAME          sqatest  sqatest  sqatest
 4   COLUMN_NAME            name      dob   height
 5   DATA_TYPE                ¯8       91        4
 6   TYPE_NAME              char     date  integer
 7   COLUMN_SIZE              10       10       10
 8   BUFFER_LENGTH            30        6        4
 9   DECIMAL_DIGITS            0        0        0
10   NUM_PREC_RADIX            0        0       10
11   NULLABLE                  1        1        1
12   REMARKS
13   COLUMN_DEF
14   SQL_DATA_TYPE            ¯8        9        4
15   SQL_DATETIME_SUB          0       91        0
16   CHAR_OCTET_LENGTH        30        0        0
17   ORDINAL_POSITION          1        2        3
18   IS_NULLABLE             YES      YES      YES
```

SQA.Connect

Purpose: Connect to a data source.

Syntax: `rc ← SQA.Connect con dsn [pwd] [user] [opt]`

`rc`	0
`con`	The name of the connection object to be created.
`dsn`	A character vector containing the name of a service.
`pwd`	Optional password.
`user`	Optional user id.
`opt`	Optional parameters

Optional Parameters for SQA.Connect

`SQA.Connect` accepts a number of optional parameters which may be specified after `srv`. The first four parameters may be specified in the order in which they are defined, the rest should be added using keyword/value pairs.

Name	Description
Password	Password
UserID	User Id
MaxCursors	The maximum number of cursors which can be opened on the connection
MaxRows	The default number of rows reserved to hold blocks of data when fetching
DriverOptions	Driver specific options (passed to Driver for processing)
WindowHandle	The handle of an existing Window (form). BrowseConnect uses this to to guide the user through logon interactively.
BindType	The method of giving the input variables (default ?)
BindChar	Character used to delimit bind variable declarations (default :)
DefaultType	Default type for undeclared variables (default `<C80:`)
Cache	Yes/No depending on whether closed statements should be stored in the cache
AplServer	Indicates that the data source is an SQAPL Server, which is capable of returning a single APL object as the result of a query
OdbcAutoCommit	Whether to set the AutoCommit option (for drivers which support it).
MaxColSize	The maximum column size to reserve space for when fetching or binding variables of unspecified width.
ReadOnly	If 1, the data source is opened in a mode where it can only be used to select data. Any attempt to update data will fail.
Columnwise	Sets the default to be used for `SQA.Prepare`.

Examples:

```
      SQA.Connect 'C2' 'NorthWind' ('MaxRows' 10000)
0
      SQA.Connect 'C1' 'Ingres' 'changeit'
0
      SQA.Connect 'NW' 'Microsoft Access Database'
            ('DriverOptions' 'DBQ=Northwind.mdb')
0
```

If you connect to a data source using `SQA.Connect` (or `SQA.BrowseConnect`) having specified the `DriverOptions` or `WindowHandle` parameter, SQAPL uses an ODBC call named *SQLDriverConnect*, rather than *SQLConnect* which is used in simple cases. In these cases, `SQA.Describe` can be used to extract a complete list of driver options used to connect, including values set by the driver (as opposed to being provided from APL). For example:

```
      SQA.BrowseConnect 'X'
0
      2 1 7⊃0 SQA.Describe 'X'
DSN=Costs;DBQ=c:\Windows\Desktop\odbcii.mdb;DriverId=25;F
IL=MS Access;MaxBufferSize=512;PageTimeout=5;UID=admin;
```

This string can be stored and provided as *DriverOptions*, to exactly recreate the connection. Note that the result may contain both userid ***AND PASSWORD*** if these were specified, so your application may want to remove these before saving the parameters in permanent storage.

SQA.CursorName

Purpose: Generate an unused cursor name.

Syntax: `rc cur ← SQA.CursorName con`

`rc`	0
`con`	Name of an existing connection object.
`cur`	Good name for a new cursor.

Example:

```
      SQA.CursorName 'C1'
0  C1.s1
```

SQA.Describe

Purpose:	Provide information about an SQAPL object.
Syntax:	`rc data← [fmt] SQA.Describe obj`

`rc`	0
`data`	If `fmt` is 1 or elided, a formatted report for the object. If `fmt` is 0, unformatted data for the object and all its children (see below for details).
`fmt`	Optional flag; 0 to avoid formatting the output.
`obj`	An object name

Example:

```
      SQA.Describe 'C5.S2'
0   Cursor C5.S2: Status=Fetch Completed, MaxRows=50

    select name,height,:dob>J: from sqatest where name
like :wname<C10: or height>:wheight<I:

    Bind Vars: 2
     Name     APL  DB  Prec.  Scale  Length    Buf
     wname    C     0     10      0      22 W  U
     wheight  I     0      0      0       4 I  I

    Select vars: 3
     Name     APL  DB  Prec.  Scale  Length    Buf
     name     C     1     10      0      11 C  C
     height   I     4     10      0       4 I  I
     dob      J    93     23      3      16 S  S
```

With a left argument of 0, the second element of the result of `SQA.Describe` is an APL array containing unformatted information for the object named in the right argument, and all of its children all the way down the object hierarchy (as for all SQAPL functions, the first element is a zero return code indicating success).

The first element of this array contains information about the named object. The second element is a vector containing two-element vectors resulting from the application of `SQA.Describe` to each child (discarding the return codes). The data returned for each SQAPL object class is as follows:

Root

Element	Contents
2	Class (1 for Root)
3	SQAPL Version
4	Full pathname of INI file
5	ODBC Environment Handle

Connection

Element	Contents
2	Class (2 for Connection)
3	ODBC Connection Handle
4	Service (ODBC Data Source) Name
5	MaxCursors
6	Block Size (MaxRows)
7	DriverOptions for connection
8	Boolean vector of ODBC Functions Supported (see ODBC documentation for *SQLGetFunctions*)

Cursor

Element	Contents
2	**Class** (3 for Cursor)
3	**State:** 1=New, 2=Executed, 3=Fetching, 4=Fetch Completed, 5=Free (cached)
4	Number of Bind Variables
5	Number of Select Variables
6	SQL Statement
7	Block Size (MaxRows)
8	Statement Handle

Buffer

Element	Contents
2	**Class** (4 for Buffer)
3	**Type** (0=Select, 1=Bind)
4	**APL Type: (formatted as CIFXOZDTSYHJG)** 1=CHAR, 2=INTEGER, 3=FLOAT, 4=BINARY 5=COMPLEX, 6=ARRAY, 7=DATE, 8=TIME, 9=TIMESTAMP, 10=YYYYMMDD, 11=HHMMSS, 12=DAYNO, 13=DECF
5	**ODBC Type: CNMIHFEUDTSV** 1=CHAR, 2=NUMERIC, 3=DECIMAL, 4=INTEGER, 5=SMALLINT, 6=FLOAT, 7=REAL, 8=DOUBLE, 9=DATE, 10=TIME, 11=TIMESTAMP, 12=VARCHAR **And: LXYZGKBUWQ** -1=LONG VARCHAR, -2=BINARY, -3=VAR BINARY, -4=LONG VAR BINARY, -5=BIG INTEGER, -6=TINY INTEGER, -7=BIT, -8=WCHAR, -9=WVARCHAR -10=WLONGVARCHAR (see the ODBC Programmers Reference for details)
6	**Internal (Logical) Type: XCIFDTSU** 1=Binary, 2=Char, 3=Integer, 4=Float, 5=Date, 6=Time, 7=Stamp, 8=Unicode
7	**Database Type Number**
8	**Precision:** The number of significant digits, or the length of a character field.
9	**Scale:** The number of digits to the right of the decimal point, if any.
10	**Length:** The number of bytes of storage consumed by one item of data.

Note that some of the type information is not available (will be reported as zero) until after the first fetch.

SQA.Do

Purpose:	Create a cursor, prepare a statement for execution, execute it, and fetch results.
Syntax:	`rc cur data nul← [all] SQA.Do con stmt [bind]`
then:	`rc cur data nul← SQA.Do cur`

`rc`	0
`cur`	Name of cursor which was created by the first call to `SQA.Do`.
`data`	Output
`nul`	Although `SQA.Do` does not support the 'Nulls' option, it still returns the empty null result from `SQA.Fetch` as the fourth element of the result.
`all`	1 or elided: fetch all data. Use 0 to fetch one block only, then make repeated calls to `SQA.Do` with the cursor name as the right argument to fetch following blocks.
`con`	Connection name
`stmt`	The SQL statement to be executed.
`bind`	Data for bind variables, if any.

Example:

```
      SQA.Do 'C1' 'select * from emp'
0  C1.S1  Alcott, Scott      Sr Programmer   50 ...
          Bee, Charles       Sr Programmer   43 ...
          Applegate, Donald  Analyst         51 ...
...
```

Note that the fifth element of the result contains a status flag for the cursor. Under normal circumstances, only two values should be returned:

5=Fetching (more records to come)

6=Fetch completed

SQA.Exec

Purpose: Execute a prepared statement.

Syntax: `rc rows← SQA.Exec cur [val]`

`rc`	0
`rows`	The number of rows modified by execution of the statement.
`cur`	The name of a cursor.
`val`	Data for bind variables, if any. To input nulls, follow bind values by equal number of boolean elements where 1's indicate nulls.

Example:

```
      SQA.Exec 'C1.S1' 'Programmer' 30
0 0
```

SQA.ExecDirect

Purpose: Execute a prepared statement.

Syntax:
```
rc rows← SQA.ExecDirect cur stmt
            [val vars opt]
```

`rc`	0
`rows`	The number of rows modified by execution of the statement.
`cur`	The name of a cursor.
`stmt`	An SQL statement.
`val`	Optional data for bind variables, if any.
`vars`	Optional bind variable declarations.
`opt`	`SQA.ExecDirect` accepts all options supported by `SQA.Prepare` and `SQA.X`.

Example:

```
       data←(↓'ZI10' ⎕FMT ⍪⍳100),⍪⍳100
       stmt←'insert into numtable values (?,?)'
       bind←2 8⍴(8↑1 1 1 0 10),8↑1 2 2
       SQA.ExecDirect 'C1.I2' stmt data bind ('Bulk' 100)
0 0 100
```

SQA.Fetch

Purpose: Fetch results of an executed statement.

Syntax: `rc data nul← [all] SQA.Fetch cur [opt]`

`rc`	0
`data`	One-element vector containing data matrix.
`nul`	Empty unless Nulls parameter is 1 (see `opt`). If Nulls is 1, boolean data with same shape as `data` indicates a null was returned for the corresponding element of `data`.
`all`	0 if you only want to fetch one block of data. 1 or elided to fetch all data.
`cur`	The name of a cursor which has been executed.
`opt`	Options (see below)

Elements after the first must be two-element vectors containing (option name) (value) pairs. Valid options are:

`nulls`	If set to 1, the third element of the result contains null indicators. The default is 0. If set to 3…
`columnwise`	If set to 1, returns each column as a simple vector or matrix.

Example:

```
      SQA.Fetch 'C1.S1' ('Nulls' 1)
0  Belter, Kris    Programmer   0 0
   Beringer, Tom                0 1
   Holton, Connie  Programmer   0 0
...
```

In the example, the data in the second row of column two was a null.

Note that the fifth element of the result contains a status flag for the cursor. Under normal circumstances, only two values should be returned:

5=Fetching (more records to come)

6=Fetch completed

This allows you to avoid the final unnecessary call to `SQA.Fetch`, returning an empty block.

SQA.GetInfo

Purpose: Retrieve information regarding a connection.

Syntax: `rc data← SQA.GetInfo con keys`

`rc`	0
`keys`	Vector of numeric constants from the file sqlext.h.
`data`	Vector of results, one for each key.

Example:

Retrieve the name and version of the ODBC driver, the ODBC version that is supported, and whether it supports “Row Updates” as follows:

```
      SQA.GetInfo 'C1' (6 7 10 11)
0   myodbc5.dll  05.01.0008  03.80.0000  N
```

SQA.GetWarning

Purpose: Fetch warnings encountered during last SQAPL function call on an object.

Syntax: `rc msgs← SQA.GetWarning obj`

`rc` 0

`msgs` Vector of four-element vectors. Each element is in the standard error format described in *Errors and Warnings*:

(origin) (code) (text) (row index)

One element is returned for each warning returned during the execution of the most recent SQAPL function call on the object in question.

`obj` The name of any SQAPL object for which the most recent function call returned a return code of `¯1`.

Example:

```
      SQA.GetWarning 'C1.S1'
 0   4  01004 0  [Q+E Software][ODBC Btrieve driver]
                 Data truncated  0
```

SQA.Init

Purpose: Initialize SQAPL.

Syntax: `rc ← SQA.Init inifile`

`SQA.Init` initializes functions which form the interface to ODBC, by loading the SQAPL library.

The right argument is usually empty, but can be used to give the name of an sqapl.ini file to use for configuration.

SQA.NativeSQL

Purpose:	Retrieve the exact SQL statement that was sent to the database.
Syntax:	`rc sql ← SQA.NativeSQL cur`
	`rc` 0
	`cur` Name of a prepared cursor.
	`sql` The SQL that was sent to the database engine.

Example:

```
      SQA.Prepare 'C1.S1' 'select * from datedemo'
      2⊃SQA.NativeSQL 'C1.S1'
SELECT *

FROM datedemo;
```

SQA.Parse

Purpose:	Convert a statement containing embedded variable declarations to standard ODBC SQL and a numeric declaration matrix.
Syntax:	`sql vars ← SQA.Parse stmt`
	`stmt` A statement containing variable declarations
	`sql` A statement without variable declarations
	`vars` An 8-column matrix declaring variables

See also Apl2Scar.

Example:

```
      SQA.Parse 'select blah where xxx=:<C10: :>S,J#2:'
 select blah where xxx=?   1 1  1 0 10 0 0 0
                           0 1  9 0  0 0 0 0
                           0 2 12 0  0 0 0 2
```

The columns of the variable declaration matrix are:

`[;1]`	Direction (1=Bind or Input, 0=Select or Output)
`[;2]`	Index into list of Bind or Select variables
`[;3]`	APL Type Number (See APL Type table)
`[;4]`	ODBC Type Number to use (0 means use default)
`[;5]`	Precision (number of chars or digits)
`[;6]`	Scale (number of digits following decimal point)
`[;7]`	Partial Flag (see *Partial Binding)*
`[;8]`	Type Variant (see *Handling Dates*)

SQA.Prepare

Purpose:	Prepare a statement for execution.
Syntax:	`rc← SQA.Prepare cur sql [opt]`

`rc`	0
`cur`	Name of cursor object to be created. The name must have two segments separated by a dot, where the first segment is the name of an existing connection object.
`sql`	The statement to be executed (usually an SQL expression).
`opt`	Options (see below)

Elements after number 2 must be two-element vectors containing (option name) (value) pairs. Valid options are:

`MaxRows`	Maximum block size for subsequent Fetch calls.
`Bulk`	Preparing for a call to `SQA.X`: The maximum number of rows of input that will be provided.
`Loop`	Set `Loop` to 0 to request that `SQA.X` should not ask the driver to loop on multiple rows of input.
`StopOnError`	Request that `SQA.X` stop if an error occurs, rather than continuing with the rest of the input.

Example:

```
      SQAPrepare 'C1.S1' 'select name,sal* from emp'
                              ('MaxRows' 10)
0
```

SQA.Scar2Apl

Purpose:	Convert a serialized array created by SQA.Apl2Scar or inserted into a table using APL type Z back into an APL array.
Syntax:	`rc array ← SQA.Scar2Apl scar`

`rc`	0
`scar`	A character vector
`array`	Any APL array

See also Apl2Scar.

Example:

```
      z←2⊃SQA.Apl2Scar (⍳3) 1
      SQA.Scar2Apl z
0  1 2 3
```

SQA.Tables

Purpose: List the tables and views accessible through a particular connection.

Syntax: `rc data← SQA.Tables con [tbl typ qua own]`

`rc`	0
`data`	A matrix with the following columns: TABLE_QUALIFIER, TABLE_OWNER TABLE_NAME, TABLE_TYPE, REMARKS
`con`	Connection Name
`tbl`	Table name
`typ`	Type: 'TABLE', 'VIEW', or other data source-specific identifiers. Comma-separated lists also accepted.
`qua`	Qualifier
`own`	Owner

The last four arguments are optional, and used to select data from the result. For `tbl`, `qua` and `own`, you can use SQL wildcards, for example values like `'T%'`.

Example:

```
      2⊃SQA.Tables 'd' 'C%'
TABLE_QUALIFIER TABLE_OWNER TABLE_NAME TABLE_TYPE REMARKS
D:\SMPLDATA                 CUSTOMER   TABLE
```

SQA.Tree

Purpose:	Returns a nested array naming all the children of an object.
Syntax:	`rc tree ← SQA.Tree obj`

`rc`	0
`obj`	The name of any SQAPL object
`tree`	A nested vector – see below

Each node in the tree is a 2-element vector containing the name of an object, followed by a list of its children. For example, when called on a connection, the second element will contain a list of cursors, and each cursor node will contain a list of input or output buffers.

Example:

```
      ]disp 2⊃SQA.Tree 'C1'
┌→─┬─────────────────────────────┐
│  │┌→───────────────────┬──────┐│
│  ││┌→─┬───────────────┐│      ││
│  │││  │┌→─────┬──────┐││      ││
│  │││  ││┌→─┬─┐│┌→─┬─┐│││┌→─┬─┐││
│C1│││I2│││#1│0│││#2│0│││││S1│0│││
│  │││  ││└─→┴⊖┘│└─→┴⊖┘│││└─→┴⊖┘││
│  │││  │└─────→┴─────→┘││      ││
│  ││└─→┴──────────────→┘│      ││
│  │└───────────────────→┴─────→┘│
└─→┴────────────────────────────→┘
```

SQA.Transact

Purpose:	Commit or roll back the current "unit of recovery".
Syntax:	`rc ← SQA.Transact con 0|1`

`rc`	0
`con`	A connection name
`0|1`	0=Commit, 1=Roll Back

Example:

```
      SQA.Transact 'C1' 0
0
```

SQA.TypeInfo

Purpose: List the data types supported by a driver. This function may not be supported by all drivers.

Syntax: `rc data← SQA.TypeInfo con`

`rc`	0
`data`	A matrix with the following columns: TYPE_NAME, DATA_TYPE, PRECISION, LITERAL_PREFIX, LITERAL_SUFFIX, CREATE_PARAMS, NULLABLE, CASE_SENSITIVE, SEARCHABLE, UNSIGNED_ATTRIBUTE, MONEY, AUTO_INCREMENT, LOCAL_TYPE_NAME
`con`	Connection Name

Example:

```
      2⊃SQATypeInfo 'c'
TYPE_NAME DATA_TYPE PRECISION LITERAL_PREFIX
LITERAL_SUFFIX
LONGCHAR         ¯1     65500 '              '
CHAR              1       255 '              '
INTEGER           4        10
FLOAT             8        15
DATE              9        10 '              '

CREATE_PARAMS NULLABLE CASE_SENSITIVE SEARCHABLE
max length           1              1          3
length               1              1          3
                     1              0          2
                     1              0          2
                     1              0          3

UNSIGNED_ATTRIBUTE MONEY AUTO_INCREMENT LOCAL_TYPE_NAM
                 0     0              0 LONGCHAR
                 0     0              0 CHAR
                 0     0              0 INTEGER
                 0     0              0 FLOAT
                 0     0              0 DATE
```

SQA.X

Purpose: Execute a prepared statement, using a matrix of bind values.

Syntax: `rc ra rp←SQA.X cur [val] [nul]`

`rc`	0
`ra`	Rows affected (not always reliable).
`rp`	Rows processed (ditto).
`cur`	Name of a prepared cursor.
`val`	A matrix of bind values, with no more than MaxRows rows (declared at `SQA.Prepare` time).
`nul`	A matrix of null flags, with the same shape as `val`.

Example:

```
      ⎕←data←2 3⍴'Harry' 35234 0 ,'Hermione' 34987 168
 Harry     35234   0
 Hermione  34987 168
      SQA.X 'C1.I1' data (2 3⍴0 0 1, 0 0 0)
0 2 2
```

Appendix A: Errors and Warnings

In the event of an error, all SQAPL functions return a four-element enclosed vector. The same format is used for warning messages retrieved by the function `SQA.GetWarning`:

[1]	Origin	Identifies the origin of the error message.
[2]	Code	One or more return codes.
[3]	Text	Wherever possible, SQAPL will obtain a textual error message from the originator, or try to tell you where to look for more information.
[4]	Index	Row index into multi-row input data

There are five possible origins of which only 3 are relevant to current-day SQAPL:

1: SQAPL Client (APL Code or AP)

```
      SQAPrepare 'C1.S1' 'select * from emp'
1 1007 OBJ Parent not found
```

The operation failed because SQAPL could not find the parent object (`C1`).

4: ODBC

```
      SQAPrepare 'C1.S1' 'select * from emp'
4  S0002 0  [Microsoft][ODBC dBASE Driver][dBASE]
            Invalid file name;file EMP.DBF not found
```

When an error is returned by ODBC, the second element contains two elements. The first element is a character vector containing the ODBC state, and the second element is either zero or a numeric native return code from the database server.

ODBC error messages also have one of two fixed formats. For errors that do not occur in the data source, the error text will have the form:

"[vendor-identifier][ODBC-component-identifier]component-supplied-text"

For errors that do occur in the data source (such as the above example), the error text will have the form:

"[vendor-identifier][ODBC-component-identifier][data-source-identifier]data-source-supplied-text"

6: Unknown error

If an unknown error is received by SQAPL, SQAPL returns

6 <the error code> '' 0

These are most likely to be generated by the underlying operating system.

If you see such errors, please contact Dyalog support; include the error code and details of the SQAPL call that lead to the error.

SQAPL Error Codes

The following errors are returned by SQAPL itself (origin code=1). The first three letters of the error test identify the internal SQAPL module in which the error occurred:

Module	Type of Error
APL	An error was detected in the shape, type, rank (etc) of an argument
INI	There is an error in the SQAPL.INI file
INT	Internal error
OBJ	There is an error in one of the SQAPL object names in the argument
SQL	SQAPL could not prepare the statement

Most error messages are explained in more detail below. The complete list of errors can be found in the file `sqapl.err`:

10001 APL Length Error

The length of an argument is incorrect

10002 APL Rank Error

The rank of an argument is incorrect

10003 INT Function does not exist

Internal error

10004 APL Domain Error

The domain of an argument is incorrect

10005 APL Nonce Error

This error is reserved for future enhancements

10006 OBJ Please call SQAInit

After starting APL or closing the root, you must call SQAInit

10007 OBJ Parent not found

You attempted to create a new object under a parent object which does not exist

10008	**OBJ Wrong Class**
	The operation is not allowed on that class of object
10009	**OBJ Name already in use**
	An object by that name already exists
10010	**OBJ Object not found**
	The named object does not exist
10011	**OBJ Already initialised**
	You called SQAInit, but SQAPL was already initialized
10012	**INT Memory allocation error**
	Internal error
10013	**INT Unknown Class (ACT)**
	Internal error
10014	**INT Unknown Action (ACT)**
	Internal error
10015	**INT Function not supported (ACT)**
	Internal error
10016	**SQL Unbalanced bind characters**
	The statement contains an odd number of bind characters (:)
10017	**SQL Unbalanced single quotes**
	The statement contains an odd number of single quotes
10018	**SQL Unbalanced double quotes**
	The statement contains an odd number of double quotes (")
10019	**SQL Invalid variable definition**
	A bind variable name is invalid
10020	**SQL Bind variable syntax error**
	A bind variable definition has invalid syntax
10021	**SQL Invalid data type identifier**
	You have used an invalid data type letter
10022	**SQL Invalid data type modifier**
	The data type modifier (following the type letter) is invalid

10023 OBJ Invalid object name

The object name is not well-formed

10024 OBJ Invalid parent

You are attempting to create an SQAPL object under a parent of the wrong type

10025 INI Invalid service definition

There is an error in the section of SQAPL.INI which describes the named service.

10026 OBJ Nonce Error

You attempted an operation which is not currently supported

10027 SQL Row too large

The space required to represent one row of output as an APL variable exceeds 32,000 bytes

10028 SQL Unbalanced Parentheses

... in a bind variable declaration

10030 SQL Column not found

A Group specification contains an unknown column name.

10031 SQL Maximum number of cursors exceeded

The driver cannot open more cursors

10032 SQL Unsupported conversion requested

An output declaration has requested a data type conversion which is not supported.

10033 PED Professional Edition Required

Internal error – should no longer appear.

10034 SCA Bound value too large

When converted to SCAR form, the bound value was bigger than the allocated buffer.

10035 SCA SCAR contains unsupported type

You tried to read a SCAR which contains a data type which is not known to your version of SQAPL. It was probably written by a later version of SQAPL.

10036 SCA SCAR conversion error

The data does not seem to be a valid SCAR.

10037 APL Invalid Option

You have used an option name which is not applicable to the SQAPL function called.

No explanations are currently provided for the following errors, hopefully they are reasonably self-explanatory. Contact support@dyalog.com for assistance if required.

10043=APL No data available

10044=OBJ Object is not defined as partial

10045=OBJ To many rows in bind variable

10046=APL Operation cancelled by user

10047=DD Cannot identify your licence

10050=SCA SCAR memory allocation

10051=SCA SCAR Conversion error

10080=APL Limit error

10081=APL Length error

10082=APL Rank error

10083=APL Value error

10084=APL Domain error

10085=APL Nonce error

10086=APL Index error

10087=APL Allocation error

10088=APL Conversion error

10089=APL Memory allocation error

10090=UNI Memory allocation

10091=UNI Translation not found

Appendix B: Schema Information

SQA.Tables

Copied verbatim from ODBC documentation for SQLTables function:

[;1] TABLE_CAT

Catalog name; NULL if not applicable to the data source. If a driver supports catalogs for some tables but not for others, such as when the driver retrieves data from different DBMSs, it returns an empty string ("") for those tables that do not have catalogs.

[;2] TABLE_SCHEM

Schema name; NULL if not applicable to the data source. If a driver supports schemas for some tables but not for others, such as when the driver retrieves data from different DBMSs, it returns an empty string ("") for those tables that do not have schemas.

[;4] TABLE_NAME

Table name.

[;5] TABLE_TYPE

Table type name; one of the following: "TABLE", "VIEW", "SYSTEM TABLE", "GLOBAL TEMPORARY", "LOCAL TEMPORARY", "ALIAS", "SYNONYM", or a data source–specific type name.

The meanings of "ALIAS" and "SYNONYM" are driver-specific.

[;6] REMARKS

A description of the table.

SQA.Columns

[;1] TABLE_CAT (ODBC 1.0)

Catalog name; NULL if not applicable to the data source. If a driver supports catalogs for some tables but not for others, such as when the driver retrieves data from different DBMSs, it returns an empty string ("") for those tables that do not have catalogs.

[;2] TABLE_SCHEM (ODBC 1.0)

Schema name; NULL if not applicable to the data source. If a driver supports schemas for some tables but not for others, such as when the driver retrieves data from different DBMSs, it returns an empty string ("") for those tables that do not have schemas.

[;3] TABLE_NAME (ODBC 1.0)

Table name.

[;4] COLUMN_NAME (ODBC 1.0)

Column name. The driver returns an empty string for a column that does not have a name.

[;5] DATA_TYPE (ODBC 1.0)

SQL data type. This can be an ODBC SQL data type or a driver-specific SQL data type. For datetime and interval data types, this column returns the concise data type (such as SQL_TYPE_DATE or SQL_INTERVAL_YEAR_TO_MONTH, instead of the nonconcise data type such as SQL_DATETIME or SQL_INTERVAL). For a list of valid ODBC SQL data types, see SQL Data Types in Appendix D: Data Types. For information about driver-specific SQL data types, see the driver's documentation.

The data types returned for ODBC 3.*x* and ODBC 2.*x* applications may be different. For more information, see Backward Compatibility and Standards Compliance.

[;6] TYPE_NAME (ODBC 1.0)

Data source–dependent data type name; for example, "CHAR", "VARCHAR", "MONEY", "LONG VARBINAR", or "CHAR ( ) FOR BIT DATA".

[;7] COLUMN_SIZE (ODBC 1.0)

If DATA_TYPE is SQL_CHAR or SQL_VARCHAR, this column contains the maximum length in characters of the column. For datetime data types, this is the total number of characters required to display the value when it is converted to characters. For numeric data types, this is either the total number of digits or the total number of bits allowed in the column, according to the NUM_PREC_RADIX column. For interval data types, this is the number of characters in the character representation of the interval literal (as defined by the interval leading precision, see Interval Data Type Length in Appendix D: Data Types). For more information, see Column Size, Decimal Digits, Transfer Octet Length, and Display Size in Appendix D: Data Types.

[;8] BUFFER_LENGTH (ODBC 1.0)

The length in bytes of data transferred on an SQLGetData, SQLFetch, or SQLFetchScroll operation if SQL_C_DEFAULT is specified. For numeric data, this size may differ from the size of the data stored on the data source. This value might differ from COLUMN_SIZE column for character data. For more information about length, see Column Size, Decimal Digits, Transfer Octet Length, and Display Size in Appendix D: Data Types.

[;9] DECIMAL_DIGITS (ODBC 1.0)

The total number of significant digits to the right of the decimal point. For SQL_TYPE_TIME and SQL_TYPE_TIMESTAMP, this column contains the number of digits in the fractional seconds component. For the other data types, this is the decimal digits of the column on the data source. For interval data types that contain a time component, this column contains the number of digits to the right of the decimal point (fractional seconds). For interval data types that do not contain a time component, this column is 0. For more information about decimal digits, see Column Size, Decimal Digits, Transfer Octet Length, and Display Size in Appendix D: Data Types. NULL is returned for data types where DECIMAL_DIGITS is not applicable.

[;10] NUM_PREC_RADIX (ODBC 1.0)

For numeric data types, either 10 or 2. If it is 10, the values in COLUMN_SIZE and DECIMAL_DIGITS give the number of decimal digits allowed for the column. For example, a DECIMAL(12,5) column would return a NUM_PREC_RADIX of 10, a COLUMN_SIZE of 12, and a DECIMAL_DIGITS of 5; a FLOAT column could return a NUM_PREC_RADIX of 10, a COLUMN_SIZE of 15, and a DECIMAL_DIGITS of NULL.

If it is 2, the values in COLUMN_SIZE and DECIMAL_DIGITS give the number of bits allowed in the column. For example, a FLOAT column could return a RADIX of 2, a COLUMN_SIZE of 53, and a DECIMAL_DIGITS of NULL.

NULL is returned for data types where NUM_PREC_RADIX is not applicable.

[;11] NULLABLE (ODBC 1.0)

SQL_NO_NULLS if the column could not include NULL values.

SQL_NULLABLE if the column accepts NULL values.

SQL_NULLABLE_UNKNOWN if it is not known whether the column accepts NULL values.

The value returned for this column differs from the value returned for the IS_NULLABLE column. The NULLABLE column indicates with certainty that a column can accept NULLs, but cannot indicate with certainty that a column does not accept NULLs. The IS_NULLABLE column indicates with certainty that a column cannot accept NULLs, but cannot indicate with certainty that a column accepts NULLs.

[;12] REMARKS (ODBC 1.0)

A description of the column.

[;13] COLUMN_DEF (ODBC 3.0)

The default value of the column. The value in this column should be interpreted as a string if it is enclosed in quotation marks.

If NULL was specified as the default value, this column is the word NULL, not enclosed in quotation marks. If the default value cannot be represented without truncation, this column contains TRUNCATED, without enclosing single quotation marks. If no default value was specified, this column is NULL.

The value of COLUMN_DEF can be used in generating a new column definition, except when it contains the value TRUNCATED.

[;14] SQL_DATA_TYPE (ODBC 3.0)

SQL data type, as it appears in the SQL_DESC_TYPE record field in the IRD. This can be an ODBC SQL data type or a driver-specific SQL data type. This column is the same as the DATA_TYPE column, except for datetime and interval data types. This column returns the nonconcise data type (such as SQL_DATETIME or SQL_INTERVAL), instead of the concise data type (such as SQL_TYPE_DATE or SQL_INTERVAL_YEAR_TO_MONTH) for datetime and interval data types. If this column returns SQL_DATETIME or SQL_INTERVAL, the specific data type can be determined from the SQL_DATETIME_SUB column. For a list of valid ODBC SQL data types, see SQL Data Types in Appendix D: Data Types. For information about driver-specific SQL data types, see the driver's documentation.

The data types returned for ODBC 3.*x* and ODBC 2.*x* applications may be different. For more information, see Backward Compatibility and Standards Compliance.

[;15] SQL_DATETIME_SUB (ODBC 3.0)

The subtype code for datetime and interval data types. For other data types, this column returns a NULL. For more information about datetime and interval subcodes, see "SQL_DESC_DATETIME_INTERVAL_CODE" in SQLSetDescField.

[;16] CHAR_OCTET_LENGTH (ODBC 3.0)

The maximum length in bytes of a character or binary data type column. For all other data types, this column returns a NULL.

[;17] ORDINAL_POSITION (ODBC 3.0)

The ordinal position of the column in the table. The first column in the table is number 1.

[;18] IS_NULLABLE (ODBC 3.0)

"NO" if the column does not include NULLs.
"YES" if the column could include NULLs.
This column returns a zero-length string if nullability is unknown.

ISO rules are followed to determine nullability. An ISO SQL–compliant DBMS cannot return an empty string. The value returned for this column differs from the value returned for the NULLABLE column. (See the description of the NULLABLE column.)

SQA.TypeInfo

Also copied unchanged from the SQLTypeInfo documentation.

[;1] TYPE_NAME (ODBC 2.0)

Data source–dependent data-type name; for example, "CHAR()", "VARCHAR()", "MONEY", "LONG VARBINARY", or "CHAR ( ) FOR BIT DATA". Applications must use this name in **CREATE TABLE** and **ALTER TABLE** statements.

[;2] DATA_TYPE (ODBC 2.0)

SQL data type. This can be an ODBC SQL data type or a driver-specific SQL data type. For datetime or interval data types, this column returns the concise data type (such as SQL_TYPE_TIME or SQL_INTERVAL_YEAR_TO_MONTH). For a list of valid ODBC SQL data types, see SQL Data Types in Appendix D: Data Types. For information about driver-specific SQL data types, see the driver's documentation.

[;3] COLUMN_SIZE (ODBC 2.0)

The maximum column size that the server supports for this data type. For numeric data, this is the maximum precision. For string data, this is the length in characters. For datetime data types, this is the length in characters of the string representation (assuming the maximum allowed precision of the fractional seconds component). NULL is returned for data types where column size is not applicable. For interval data types, this is the number of characters in the character representation of the interval literal (as defined by the interval leading precision; see Interval Data Type Length in Appendix D: Data Types).

For more information on column size, see Column Size, Decimal Digits, Transfer Octet Length, and Display Size in Appendix D: Data Types.

[;4] LITERAL_PREFIX (ODBC 2.0)

Character or characters used to prefix a literal; for example, a single quotation mark (') for character data types or 0x for binary data types; NULL is returned for data types where a literal prefix is not applicable.

[;5] LITERAL_SUFFIX (ODBC 2.0)

Character or characters used to terminate a literal; for example, a single quotation mark (') for character data types; NULL is returned for data types where a literal suffix is not applicable.

[;6] CREATE_PARAMS (ODBC 2.0)

A list of keywords, separated by commas, corresponding to each parameter that the application may specify in parentheses when using the name that is returned in the TYPE_NAME field. The keywords in the list can be any of the following: length, precision, or scale. They appear in the order that the syntax requires them to be used. For example, CREATE_PARAMS for DECIMAL would be "precision,scale"; CREATE_PARAMS for VARCHAR would equal "length." NULL is returned if there are no parameters for the data type definition; for example, INTEGER.

The driver supplies the CREATE_PARAMS text in the language of the country/region where it is used.

[;7] NULLABLE (ODBC 2.0)

Whether the data type accepts a NULL value:

SQL_NO_NULLS if the data type does not accept NULL values.

SQL_NULLABLE if the data type accepts NULL values.

SQL_NULLABLE_UNKNOWN if it is not known whether the column accepts NULL values.

[;8] CASE_SENSITIVE (ODBC 2.0)

Whether a character data type is case-sensitive in collations and comparisons:

SQL_TRUE if the data type is a character data type and is case-sensitive.

SQL_FALSE if the data type is not a character data type or is not case-sensitive.

[;9] SEARCHABLE (ODBC 2.0)

How the data type is used in a **WHERE** clause:

SQL_PRED_NONE if the column cannot be used in a **WHERE** clause. (This is the same as the SQL_UNSEARCHABLE value in ODBC 2.*x*.)

SQL_PRED_CHAR if the column can be used in a **WHERE** clause, but only with the **LIKE** predicate. (This is the same as the SQL_LIKE_ONLY value in ODBC 2.*x*.)

SQL_PRED_BASIC if the column can be used in a **WHERE** clause with all the comparison operators except **LIKE** (comparison, quantified comparison, **BETWEEN**, **DISTINCT**, **IN**, **MATCH**, and **UNIQUE**). (This is the same as the SQL_ALL_EXCEPT_LIKE value in ODBC 2.*x*.)

SQL_SEARCHABLE if the column can be used in a **WHERE** clause with any comparison operator.

[;10] UNSIGNED_ATTRIBUTE (ODBC 2.0)

Whether the data type is unsigned:

SQL_TRUE if the data type is unsigned.

SQL_FALSE if the data type is signed.

NULL is returned if the attribute is not applicable to the data type or the data type is not numeric.

[;11] FIXED_PREC_SCALE (ODBC 2.0)

Whether the data type has predefined fixed precision and scale (which are data source–specific), such as a money data type:

SQL_TRUE if it has predefined fixed precision and scale.

SQL_FALSE if it does not have predefined fixed precision and scale.

[;12] AUTO_UNIQUE_VALUE (ODBC 2.0)

Whether the data type is autoincrementing:

SQL_TRUE if the data type is autoincrementing.

SQL_FALSE if the data type is not autoincrementing.

NULL is returned if the attribute is not applicable to the data type or the data type is not numeric.

An application can insert values into a column having this attribute, but typically cannot update the values in the column.

When an insert is made into an auto-increment column, a unique value is inserted into the column at insert time. The increment is not defined, but is data source–specific. An application should not assume that an auto-increment column starts at any particular point or increments by any particular value.

[;13] LOCAL_TYPE_NAME (ODBC 2.0)

Localized version of the data source–dependent name of the data type. NULL is returned if a localized name is not supported by the data source. This name is intended for display only, such as in dialog boxes.

[;14] MINIMUM_SCALE (ODBC 2.0)

The minimum scale of the data type on the data source. If a data type has a fixed scale, the MINIMUM_SCALE and MAXIMUM_SCALE columns both contain this value. For example, an SQL_TYPE_TIMESTAMP column might have a fixed scale for fractional seconds. NULL is returned where scale is not applicable. For more information, see Column Size, Decimal Digits, Transfer Octet Length, and Display Size in Appendix D: Data Types.

[;15] MAXIMUM_SCALE (ODBC 2.0)

The maximum scale of the data type on the data source. NULL is returned where scale is not applicable. If the maximum scale is not defined separately on the data source, but is instead defined to be the same as the maximum precision, this column contains the same value as the COLUMN_SIZE column. For more information, see Column Size, Decimal Digits, Transfer Octet Length, and Display Size in Appendix D: Data Types.

[;16] SQL_DATA_TYPE (ODBC 3.0)

The value of the SQL data type as it appears in the SQL_DESC_TYPE field of the descriptor. This column is the same as the DATA_TYPE column, except for interval and datetime data types.

For interval and datetime data types, the SQL_DATA_TYPE field in the result set will return SQL_INTERVAL or SQL_DATETIME, and the SQL_DATETIME_SUB field will return the subcode for the specific interval or datetime data type.

[;17] SQL_DATETIME_SUB (ODBC 3.0)

When the value of SQL_DATA_TYPE is SQL_DATETIME or SQL_INTERVAL, this column contains the datetime/interval subcode. For data types other than datetime and interval, this field is NULL.

For interval or datetime data types, the SQL_DATA_TYPE field in the result set will return SQL_INTERVAL or SQL_DATETIME, and the SQL_DATETIME_SUB field will return the subcode for the specific interval or datetime data type.

[;18] NUM_PREC_RADIX (ODBC 3.0)

If the data type is an approximate numeric type, this column contains the value 2 to indicate that COLUMN_SIZE specifies a number of bits. For exact numeric types, this column contains the value 10 to indicate that COLUMN_SIZE specifies a number of decimal digits. Otherwise, this column is NULL.

[;19] INTERVAL_PRECISION (ODBC 3.0)

If the data type is an interval data type, then this column contains the value of the interval leading precision. (See Interval Data Type Precision in Appendix D: Data Types.) Otherwise, this column is NULL.

Appendix C: More about Translation

SQAPL performs two different types of translation of character data:

1. **All character data** must be translated between the APL Atomic Vector and the operating system environment (Classic edition only).

2. **APL objects** containing text may need to be translated from the APL Atomic Vector of the APL system which has saved a binary APL object in an ODBC table or created an array used as the argument to `SQA.Scar2Apl`, to the character set used by the receiving system.

The first form of translation is always required if you are using the Classic edition: Whenever an item of textual information is moved between the APL system and a database or back, character data must be translated. This requires knowledge of character set used by the APL system and that used by the host. In the Unicode edition, APL is using the same character set as the operating system, so no translation is required.

The second form of translation is only used when you read an APL object ***originating in a different APL system and containing character data***, from the SCAR (Self Contained ARay) format to an APL variable in the current workspace. This can happen when you use an output variable in "Z" format, or when you call the function `SQA.Scar2Apl`. This translation requires knowledge of the character sets used by the sending and receiving APL systems.

The file APLUNICD.INI contains definitions of all the different characters sets known to SQAPL: At least one for each APL platform to which SQAPL has been ported (with some national variations), and for completeness, one representing the operating system, named ASCII, defined as the first 256 UNICODE characters, also known as ANSI. Each character set is defined as a list of 256 UNICODE characters. Using these tables, SQAPL is able to translate text between any two character sets.

The standard distribution version of this file starts with the section

```
[Charsets]
APL2=IBMA
APLIII=MAN3
APLUNX=MAN3
APLWIN=MAN3
DYALOG=DYA_IN
SAXAPL=SAX_US
HOST=ASCII
```

The names on the left identify a particular implementation (or "port") of SQAPL, plus one entry for the host operating system. On the right is the name of the table defining the alphabet which is used by the named environment.

When a SCAR object is created, character data is stored without translation (it is stored as a sequence of bytes which are indices into the atomic vector). Part of the header of the SCAR object names the translate table which was in use by the system which created the SCAR. The system which reads the SCAR checks the header, and if translation is necessary SQAPL needs to be able to locate BOTH tables (that of the reader as well as the writer) in the APLUNICD.INI file, in order to create the translate table between the two systems and perform the translation.

Locating the Required Tables

SQAPL searches the registry to find the string `APL_UNICODE`, which is used to locate the file `aplunicd.ini` and `SQAPLPATH`, which identifies the path in which `sqapl.ini` is located. The right argument to `SQA.Init` identifies the registry key which should be searched, if no key is provided, the default is HKEY_CURRENT_USER\Software\Insight\SQAPL.

If the files are not found, then for a Classic system `SQA.Init` will establish a default translate table between APL and the DLL using `⎕NXLATE`.

If your application was using the default translate table and had no special national requirements, and you do not intend to transfer APL objects in SCAR format, you do not need to worry about this, as translation of textual data between APL and SQL databases will be done correctly.

If you intend to use SCAR as a transfer mechanism between APL systems, an error message will be issued when a SCAR needs to be translated, unless the aplunicd file has been found and contains the necessary definitions.

Appendix D: Release Change Notes

Version 6.2

SQAPL DLL/Shared Library Naming Convention

Beginning with version 6.2, the name of the DLL (under Windows) or shared library (under UNIX) file for SQAPL will use the following convention.

c<platform>**dya**<version><classic/Unicode><bits><driver manager>**.**<ext>

Where:

platform is ‘x’ for UNIX or ‘w’ for Windows
version is the major/minor version number of SQAPL
classic/Unicode is ‘c” for classic or ‘u’ for Unicode
bits is either 32 or 64
driver manager is ‘u’ for unixODBC, ‘d’ for DataDirect, or ‘w’ for Windows
ext is ‘so’ for UNIX and ‘dll’ for Windows

For example:

The library for the version 6.2, 64 bit, Unicode version would be:

`cxdya62u64u.so` under UNIX
`cwdya62u64w.dll` under Windows

Use of unixODBC Under Linux

Beginning with version 6.2, when running under Linux, SQAPL requires unixODBC to be installed. If your Linux implementation does not have unixODBC installed already, you may find information for how to download and install unixODBC at `http://www.unixodbc.org`.