Omnis SQL

This document describes how to use Omnis SQL to access an Omnis database[*], and assumes a working knowledge of SQL. A formal definition of the Omnis SQL language is included.

The Omnis Data Access Module (DAM), which is in the EXTERNAL folder, lets you connect to an Omnis database using Omnis SQL. No additional software is required to use Omnis SQL.

Connecting to the Database

For an application to logon to a database in runtime requires a set of commands in a method. For Omnis SQL all that is required is the command *Set hostname* to identify the database that contains the tables you want to access.

```
Set hostname { c:\omnis\df\my-dfile}
```

Sending SQL to the Database

Before a client application can get any data from a server, it must set up a corresponding place in Omnis to hold the data. This involves mapping the structure of the data, including column names and data types. Typically, you do this using Omnis schema classes. You can define a schema to include all columns of the server table or any subset of the columns.

The schema serves as a framework for defining a table class which, in turn, is the reference for defining lists and rows. Omnis uses list and row variables for handling client/server data. Creating schema and table classes is described earlier in this manual, and using lists and rows is described in the *List Programming* chapter.

To send SQL to the database, you can either write your own methods, or use the table instance methods that handle both single row and bulk SQL transactions. This section covers custom methods; Omnis table classes were described earlier.

Omnis provides two different ways of building a SQL statement and sending it to the database: the *Perform SQL* command and SQL scripts.

---

[*] This document covers Omnis SQL programming using Omnis Studio version 1 and 2. This functionality has been replaced in Omnis Studio version 3/4 by session and statement object methods and properties (see the *SQL Programming* chapter in the Studio 4.x Omnis Programming manual), however this document is included in order to assist your understanding of existing applications built using the older API.

Perform SQL

The *Perform SQL* command sends a single-line SQL statement to the current SQL session:

```
Perform SQL { SELECT name FROM Agent }
```

You can substitute text into the SQL statement using square bracket notation:

```
Perform SQL { SELECT name from [TABLENAME] }
```

where TABLENAME is an Omnis variable.

SQL Scripts and the SQL Buffer

For longer statements that you may want to enter on more than one line, Omnis provides the SQL script. Omnis has a SQL buffer, an area of memory that contains a single SQL statement that you build up with a series of commands. *Begin SQL script* clears the buffer; *SQL:* enters a line of SQL text, *End SQL script* closes the buffer and *Execute SQL script* sends the contents of the buffer to the database. For example,

```
Begin SQL script
SQL: INSERT INTO Agent
SQL: (name,number)
SQL: VALUES ('FRED',123)
End SQL script
Execute SQL script
```

is equivalent to the *Perform SQL* command

```
Perform SQL {INSERT INTO Agent (name, number) VALUES ('FRED', 123)}
```

You can also use *Get SQL script {field name}* to copy the contents of the SQL buffer into a variable and *Set SQL script {field name}* or to copy the contents of a variable into the current SQL buffer.

Error Handling

Both *Perform SQL* and *Execute SQL script* clear the flag when the operation is not successful and the functions *sys(131)* and *sys(132)* report the error code and error text respectively.

```
Perform SQL { SELECT name FROM Agent }
If flag false
   OK message {SQL error [sys(131)] [sys(132)] }
End If
```

The Name Functions

There are several functions you can use to create part of a SQL statement. Each of these functions takes a file class name or field name list as a parameter and evaluates to a string of text that Omnis inserts automatically into the SQL buffer. With these functions, you can write general-purpose methods that will work with any server without typing long SQL

statements. For more information, see the descriptions in the Omnis Help. The functions are:

– createnames()

– insertnames()

– selectnames()

– updatenames()

– wherenames()

Data Mapping

There are several ways to map Omnis data into SQL statements.

Square Bracket Notation

SQL statements can contain square bracket notation, which Omnis evaluates. If you use it you must supply quoted literals. For example, to update an Agent table by setting the name to a string from a variable called FIELD, you must quote the square bracket notation expression:

```
Perform SQL { UPDATE Agent SET name ='[FIELD]' }
```

You cannot use the string '[][]' (two sets of empty square braces) in your SQL statements because the DAMs use this string to mark the variables passed as @[]

Bind Variables

A *bind variable* is an Omnis variable to which you want to refer in a SQL statement. Instead of expanding the expression, Omnis binds, or associates the variable value with a SQL variable. If you place an @ before the opening square bracket, Omnis evaluates the expression and passes the value to the server directly rather than substituting it into the SQL statement as text. You can also use this syntax to bind large fields such as pictures into a SQL statement:

```
Perform SQL { INSERT INTO Agent (agentPortrait) values (@[P_FIELD])
```

 Never quote bind variables, and use them only to represent complete literals or values; otherwise you will get an error from the server.

Generally, using bind variables performs better than square bracket notation and is more flexible with respect to data representation. You should use square bracket notation only when the notation expression evaluates to a part of a SQL statement broader than just a value reference (such as an entire WHERE clause, for example) or where you know that simple value substitution is all you need. This works best for numeric data; strings tend to cause problems because of the issues with quoting. You must include quotes when using square bracket notation, but you don't need to when using bind variables. Also, if you are inserting NULL data into the database, you should use bind variables, since square bracket notation tends to insert empty strings into the SQL statement, not SQL nulls. This also applies to pictures, binary data, and very long text.

Select Tables and Cursors

A *select table* is a table of results that belongs to a session. When you send a SQL SELECT statement to the server and there is no error, the results of the SELECT become available to Omnis as the select table for the current session. The select table can be empty; in this case, the flag is true after the execution of the select and is only set to false when you attempt to fetch the first row after the end of the select table.

You can map the data in the select table into Omnis data in three ways:

– *Declare cursor* and *Fetch next row*

– *Build list from select table*

– *Retrieve rows to file*

When you fetch data from the server, Omnis converts data types between the native SQL server and the Omnis data type if possible, including numeric precision. If there is a total mismatch between Omnis field types and SQL column types, you can lose information or get a SQL error.

Declare cursor and Fetch next row

The *Declare cursor* and *Fetch next row* commands let you map each row in the select table into the CRB on a row-by-row basis.

Declare cursor defines a *SQL cursor*, a named pointer to a row in the select table, and associates a SQL select statement with the cursor. *Open cursor* opens the cursor, parses the SQL statement, binds input data, and executes the SQL statement. *Set current cursor* switches Omnis to use the named cursor.

When you execute a SQL SELECT statement, the *current cursor* points to the first row in the resulting select table. When you *Fetch next row*, you fetch the row pointed to by the current cursor and move the cursor to the next row. You can have more than one cursor active at a time, letting you select rows based on values retrieved from a completely separate select table. You use the *Set current cursor* command to use a particular cursor as the current cursor with the *Fetch* commands.

Unless you are using multiple cursors, you don't need to explicitly open a cursor; Omnis automatically opens one for you.

The *Fetch next row* command loads the column values for a single row of the select table into the Omnis CRB fields.

If the list of fields does not match the columns in the select table, Omnis tries to map the data as best it can. If there are more columns than fields, then Omnis doesn't copy the extra column values into Omnis variables. On the other hand, if there are more fields than columns, then Omnis leaves the extra field values unchanged.

The usual retrieval process is to fetch the rows in a loop, one at a time, until there are no more rows in the select table. To do this, you use *Fetch next row* which fetches the row

pointed to by the current cursor, then moves the cursor to the next row. After successfully fetching a row, the flag is true. After you fetch the last row, the next fetch returns a false flag and does nothing to the mapped fields. You then can use the *Close cursor* command to close a cursor explicitly, freeing the memory it uses.

Using the *Repeat* and *Until* commands with the flag lets you fetch until the flag turns false, though you must save the value of the flag in a separate variable for the test, since other commands may reset the flag before reaching the end of the loop. You can also use a *While* command, fetching the first row before entering the loop.

There are several variations on the *Fetch* command.

– *Fetch first row* fetches the first row in the select table and points the cursor at the second row

– *Fetch current row* fetches the current row and leaves the cursor pointing to that row

– *Fetch next row* fetches the current row and points the cursor to the following row

Build list from select table

The *Build list from select table* command fetches all the rows of the select table into a list that has been defined with the appropriate fields. The command appends the values rather than overwriting any values in the list so you can use this feature to put multiple select tables into a list, but there is a *Clear list* option to clear the list first. If you have defined the list with other fields or variables, the *Add CRB fields* option inserts these values to the list as well.

You can use #LM or $linemax to limit the size of the list regardless of the number of rows in the select table. There are also commands provided by most servers to limit number of rows returned; do not confuse these with the #LM value, which just affects the list.

The following method selects a table called **Contacts** with columns **name** and **number** directly into a list.

```
; Local variable lvContacts (list)
Set current list lvContacts
Define list { fContacts }
Perform SQL { SELECT name, number FROM Contacts }
; Creates the select table of all rows and columns
If flag true
   Build list from select table
End If
```

Retrieve Rows to File

This command copies the select table on a row by row basis into the current client import file, where the data is appended in tab-delimited format.

```
Set client import file name {my_file}
Open client import file
Perform SQL {Select * from my_table}
Retrieve rows to file
Close client import file
```

Omnis SQL Language Definition

The following sections show the grammar of Omnis SQL using BNF (Backus-Naur Form) diagrams, using the conventions from the ANSI standard.

Each statement includes a note specifying what parts, if any, of the statement depart from the ANSI 1989 standard for SQL.

SQL Statement

```
SQL_statement ::=
    create_table_statement
  | create_index_statement
  | delete_statement_searched
  | drop_index_statement
  | drop_table_statement
  | insert_statement
  | select_statement
  | update_statement_searched
  | update_statement_positioned
  | alter_table_statement
```

The SQL statement is the text that goes in the *Perform SQL* command or in a SQL script starting with *Begin SQL script*. The rest of the grammar depends on this main element.

ANSI SQL has the following statements that Omnis does not implement. Most statement involve cursors, and Omnis implements these as commands rather than as SQL statements.

– **close_statement**
closes a cursor (see the *Close cursor*, *Quit cursor*, and *Reset cursors* commands)

– **commit_statement**
commits a transaction (see the *Commit current session* command)

– **declare_cursor**
declares a cursor (see the *Declare cursor* command)

– **delete_statement_positioned**
deletes a row based on current cursor position

– **fetch_statement**
fetches a row using the current cursor (see the *Fetch* commands)

– **open_statement**
opens a cursor (see the *Open cursor* command)

- **create_schema_statement**
  creates a schema containing tables and views; Omnis SQL does not support schemas

- **create_view_statement**
  creates a view; Omnis SQL does not support views

- **grant_privilege**
  grants an access privilege on an object to a user; Omnis SQL does not implement any SQL security

CREATE TABLE

```
create_table_statement ::=
   CREATE TABLE table ( table_element_comma_list )
   CONNECTIONS ( table_comma_list )
```

The CONNECTIONS clause is an Omnis extension to the ANSI standard that lets you specify a list of file classes to which to connect a file class. Connections are parent-child relationships between file classes. See the *Omnis Data Files* chapter for information on connections

```
table_element ::= column_definition | UNIQUE ( column_comma_list )
```

You can define a file class using the SQL CREATE TABLE statement. The fields in the format come from the list of column definitions. You can also specify that the values for a group of columns are unique, taken together, with the UNIQUE constraint. You can have more than one UNIQUE constraint. All the columns in a UNIQUE constraint must be defined with the NOT NULL qualifier (see below).

The ANSI standard contains several other table constraints, namely PRIMARY KEY, FOREIGN KEY and CHECK that Omnis SQL does not implement.

```
column_definition ::= column_data [ [ NOT ] NULL ]
```

The NOT NULL constraint specifies that when you insert a row, the value for this column must not be NULL.

The ANSI standard specifies a default clause that lets you define a default value for the column. It also lets you specify that the column is UNIQUE, REFERENCES a primary key in another table, or satisfies a CHECK constraint. Omnis SQL does not implement any of these features.

```
column_data ::=
   column_name data_type
```

```
data_type ::=
   [ LONG ] VARBINARY
   |  BIT
   |  VARCHAR ( NUMBER )
   |  CHAR ( NUMBER )
   |  NATIONAL CHAR[ACTER] VARYING (NUMBER)
   |  NCHAR VARYING ( NUMBER )
   |  SEQUENCE_TYPE
   |  DATE [ ( { 1900..1999 | 1980..2079
   |  2000..2099 } ) ]
   |  TIME
   |  TIMESTAMP
   |  TINYINT
   |  SMALLINT
   |  INTEGER
   |  NUMERIC ( number, integer)
   |  DEC[IMAL] ( number, integer)
   |  FLOAT_TYPE [ ( integer ) ]
   |  REAL
   |  LIST
   |  PICTURE
```

ANSI data types include CHARACTER, NUMERIC, DECIMAL, INTEGER, INT, SMALLINT, FLOAT, REAL, and DOUBLE PRECISION. Omnis does not implement FLOAT and DOUBLE PRECISION directly, though FLOAT_TYPE is similar to FLOAT. The other data types are Omnis specific. The integer value in the NUMERIC, DECIMAL, and FLOAT_TYPE types corresponds to the Omnis subtypes for numbers; 0-8, 10, 12, and 14 are the possible values.

ALTER TABLE

```
alter_table_statement ::=
   ALTER TABLE table ADD
   { column_data | ( column_data_comma_list ) }
```

The ALTER TABLE statement lets you add a column to an already existing table using the same syntax as in CREATE TABLE.

The ALTER TABLE statement does not exist in the 1989 ANSI standard.

DROP TABLE

```
drop table statement ::=
   DROP TABLE table_name
```

The DROP TABLE statement removes a file slot and any data for that slot from an Omnis datafile.

The DROP TABLE statement does not exist in the 1989 ANSI standard.

CREATE INDEX

```
create_index_statement ::=
   CREATE [CASE SENSITIVE] [UNIQUE] INDEX index
   ON table ( index_column_comma_list )

index_column ::=
   column_reference [ ASC ]
```

The CREATE INDEX statement lets you create an index on an Omnis database column. You can make the index UNIQUE, asserting that no two rows of the database have the same value for this combination of columns. You can also make the index CASE SENSITIVE, this will usually result in more efficient queries. The index column list contains columns from the table, and the table must already exist. You can also specify ASC on an individual column to sort it in ascending, as opposed to descending, order.

The CREATE INDEX statement does not exist in the 1989 ANSI standard.

DROP INDEX

```
drop_index_statement ::= DROP INDEX index
```

The DROP INDEX statement removes the named index, which must already exist.

The DROP INDEX statement does not exist in the 1989 ANSI standard.

SELECT

```
select_statement ::=
   SELECT [ ALL | DISTINCT ] { value_expression_comma_list | * }
   from_clause
   [ where_clause ]
   [ group_by_clause ]
   [order_by_clause ]
   [FOR UPDATE ]
```

The SELECT statement is the basic query statement in Omnis SQL. It largely matches the ANSI standard, one exception being the having clause, which in Omnis SQL is part of the group by clause instead of being a separate clause in the select statement. That is, in Omnis SQL you cannot have a HAVING clause separate from the GROUP BY clause.

The FOR UPDATE clause initiates special locking for the records in the query. When you fetch a row from a cursor containing a SELECT statement with a FOR UPDATE clause, Omnis locks the row for update. One of three things can then happen:

– You update the record with an UPDATE ... WHERE CURRENT OF cursor_name (see below), which on completion unlocks the row

– You fetch another row, which releases the lock on the previous row and locks the current one

– You terminate the transaction, which releases all locks

The order_by clause is separated out in ANSI SQL so that there is only one ordering for a query. Since Omnis SQL does not have any set operators, such as UNION, there is no need to separate out the ordering clause.

The ANSI 1989 standard has no for_update clause. This comes from embedded SQL, the syntax there is FOR UPDATE OF column_name_list.

Value Expression

```
value_expression ::=
     term
   | value_expression { + | - } term

term ::=
   factor
   | term { * | / } factor

factor ::=
   [ { + | - } ] primary

primary ::=
     literal
   | column_reference
   | function_reference
   | ( value_expression )
```

A value expression is a key element of SQL that lets you calculate a value using an arithmetic expression language. You build an expression out of literal numbers and strings, references to columns, or parenthesized, nested expressions. You can combine expressions with any of the four arithmetic operators. The grammar above expresses the precedence relationships between the operators: unary + and - take precedence over * and /, all of which take precedence over binary + and -.

Column and Table References

```
column_reference ::=
     [ table . ] column_name

   | [ alias . ] column_name
```

The column name corresponds to a field in a file class.

```
table ::=
   [ library_name . ] table_name
```

The table name corresponds to a file class or to a table alias in the same SELECT statement, and the library name corresponds to a library. The table must belong to the library.

Omnis SQL does not support the ANSI standard syntax alias.*, meaning all the columns from the table to which the alias refers. Also, if you use something other than a library name, or a name that Omnis cannot recognize as a library name, you will get a syntax error.

Function Reference

```
function_reference ::=
      scalar_function
    | aggregate_function
```

A function reference is either a scalar function or an aggregate function. *Scalar functions* operate on each row of data in the select; *aggregate functions* operate on groups of rows.

The ANSI SQL standard has no scalar functions.

```
scalar_function ::=
    scalar_function_name ( value_expression_comma_list )
```

There are a number of scalar functions, summarized below.

Function	Purpose	Parameters
ABS	absolute value of a number	number
ACOS	angle in radians, the cosine of which is a specified number	number
ASCII	ASCII character corresponding to an integer between 0 and 255, inclusive	integer
ASIN	angle in radians whose sine is the specified number	number
ATAN	the angle in radians whose tangent is the specified number	number
ATAN2	the angle in radians whose tangent is one number divided by another number	number 1, number 2
CHARINDEX	the starting character position of one string in a second string	index string, source string
CHR	ASCII character corresponding to an integer between 0 and 255, inclusive	integer
COS	cosine of a number	number
TODATE	converts a date string or number to a date value using a format string	date string/number, format string
DIM	increments a date string by some number of months	date string, months
DTCY	a string containing the year and century of a date string	date string
DTD	a string containing the day part of a date string or a number representing the day of the month, depending on context	date string
DTM	a string containing the month part of a date	date string

Function	Purpose	Parameters
	string or a number representing the month of the year, depending on context	
DTW	a string containing the day of the week part of a date string or a number representing the day of the week, depending on context	date string
DTY	a string containing the year part of a date string or a number representing the year, depending on context	date string
EXP	exponential value of a number	number
INITCAP	transforms string by capitalizing the initial letter of each word in the string and lower-casing every other letter	string
LENGTH	number of characters in a string	string
LOG	natural logarithm of a number	number
LOG10	base 10 logarithm of number	number
LOWER	transforms string by lower-casing all letters	string
MOD	modulus of a number given another number	number, modulo number
POWER	the value of a number raised to the power of another number	number, power
ROUND	rounds a number to an integer number of significant digits	number, significant digits
SIN	sine of a number	number
SQRT	square root of a number	number
STRING	concatenates some number of strings into a string	string[, string, ...]
SUBSTRING	extracts part of a string starting at a given index and moving a certain number of characters	string, start index, length
TAN	tangent of a number	number
UPPER	transforms a string by upper-casing all letters	string

```
aggregate function ::=
     COUNT(*)
   | aggregate function name ( DISTINCT column reference )
   | aggregate function name ( [ ALL ] value expression

aggregate_function_name ::=
   AVG | MAX | MIN | SUM | COUNT
```

There are some departures from the ANSI standard for DISTINCT aggregates: you can use only one such function in a given SQL statement, and you cannot use aggregate functions in expressions in a GROUP BY clause or WHERE clause.

FROM Clause

```
from_clause ::=
     FROM table_reference_comma_list

table_reference ::=
   table_name [ AS ] [ alias ]
```

The FROM clause lets you specify the table to input into the SQL statement. Multiple tables in the list indicate a join, and the WHERE clause specifies the join condition. Each table reference can have an optional alias that lets you refer to the table in other parts of the SQL statement by the alias. You can use this to abbreviate references to the table in the other clauses.

The ANSI standard does not have the optional AS keyword.

WHERE Clause

```
where_clause ::=
   WHERE search_condition

search_condition ::=
   boolean_term | search_condition OR boolean_term

boolean_term ::=
   boolean_factor | boolean_term AND boolean_factor

boolean_factor ::=
   [ NOT ] boolean_primary

boolean_primary ::=
   predicate | ( search_condition )
```

The WHERE clause lets you select a subset of the input rows using a logical predicate. The above grammar defines the precedence of the logical operators AND, OR, and NOT.

```
predicate ::=
     comparison_predicate
   | between_predicate
   | in_predicate
   | like_predicate
   | relation_predicate
   | null_predicate
```

The ANSI standard has, in addition to the above predicates, the quantified and exists predicates (nested selects), which Omnis does not support. The relation_predicate is an Omnis extension to the standard that lets you use Omnis connections; see below.

```
comparison_predicate ::=
   value_expression comparison_operator value_expression

comparison_operator ::=
   < | > | = | <> | != | >= | <= | *= | =*
```

The standard comparison predicate involves one of the relational operators (greater than, less than, and so on).

ANSI SQL also allows you to use a nested select statement in place of the right-hand value_expression; Omnis SQL does not support that. Omnis adds the !=, *=, and =* operators (not equal, left outer join, and right outer join, respectively) to the ANSI standard operators.

An *outer join* is a join that includes all the rows in the tables regardless of the matching of the rows. The *= operator includes all rows from the table on the left that satisfy the rest of the WHERE clause. The =* operator includes all rows from the table on the right that satisfy the WHERE clause. Rows from the other table (right and left, respectively, contribute values if there is a match and NULLs if not. This syntax is similar to the SYBASE outer join syntax.

```
between_predicate ::=
   value_expression [ NOT ] BETWEEN value_expression AND
   value_expression

in_predicate ::=
   value_expression [ NOT ] IN ( literal_comma_list )
```

The ANSI standard lets you use a subquery (a nested select) as well as a literal list; Omnis does not.

```
like_predicate ::=
   column_reference [ NOT ] LIKE literal
```

The ANSI standard adds an ESCAPE clause to the like_predicate to let you specify an escape character so you can match a % or _; Omnis does not implement this.

```
null_predicate ::=
   column_reference IS [ NOT ] NULL

relation_predicate ::=
   { CHILD | PARENT } OF table
```

The relation_predicate lets you test the current row as being either a child or a parent of rows in the specified table. . See the *Omnis Data Files* chapter for information on parent-child connection relationships

GROUP BY Clause

```
group_by_clause ::=
   GROUP BY column_reference_comma_list [ HAVING search_condition ]
```

The group_by_clause lets you group the input rows into groups according to a set of columns. The HAVING clause lets you select the groups, as opposed to the WHERE clause, which selects the rows going into the groups.

ANSI SQL has no ordering dependency between GROUP BY and HAVING, and you can have a HAVING clause without an accompanying GROUP BY. Omnis does not allow this.

Omnis SQL does not support the use of functions in a GROUP BY clause.

ORDER BY Clause

```
order_by_clause ::=
   ORDER BY order_column_comma_list

order_column ::=
   column_reference [ ASC | DESC ]
```

The order_by_clause lets you sort the output rows of the SQL statement using columns from the input tables.

The ANSI standard lets you sort by value_expressions in the select list by specifying the number of the expression; Omnis does not.

INSERT

```
insert statement ::=
   INSERT INTO table [ ( column_reference_comma_list ) ]
   { VALUES ( insert_value_comma_list ) | select_statement }
```

The INSERT statement inserts rows into an Omnis table. The first list of columns names the columns you are creating; this exists to let you reorder the list to match your list of values or select statement.

There are two alternative ways to supply values to the INSERT statement. You can supply actual values through a VALUES clause that contains a list of values, or you can give a SELECT statement that creates a table of data matching the insert list. See the *SELECT* statement section above for details on SELECT.

```
insert_value ::=
   literal | NULL
```

An insert value is a literal value or the NULL value specified by the string "NULL".

UPDATE

```
update_statement_searched ::=
   UPDATE table SET assignment_comma_list [ where_clause ]

assignment ::=
   column_reference = { value_expression | NULL }
```

The searched update statement updates all rows that satisfy the predicate in the WHERE clause by assigning the indicated value or NULL to the column.

Omnis SQL will let you preface the column name in the assignment with the library and table names, which extends the ANSI standard. There is no need to specify the additional names, but you can do so for clarity if you wish. Specifying a table other than the table in the UPDATE table clause, generates an error.

```
update_statement_positioned ::=
   UPDATE table SET assignment_comma_list
   WHERE CURRENT OF cursor
```

The positioned update statement updates the current row, the row to which the current cursor points. See the description of the *Declare cursor* command in the Omnis Help. The

WHERE CURRENT OF cursor clause works with the SELECT ... FOR UPDATE statement to update rows locked for update.

DELETE

```
delete_statement_searched ::=
   DELETE FROM table [ where_clause ]
```

The DELETE statement deletes rows from the Omnis database based on the predicate in the WHERE clause. Omnis deletes all rows that satisfy the predicate.