4 Data Types

This chapter provides a reference to Oracle external data types used by OCI applications.

It also discusses Oracle data types and the conversions between internal and external representations that occur when you transfer data between your program and an Oracle database.

This chapter contains these topics:

See Also:

Oracle Database SQL Language Reference for detailed information about Oracle internal data types

4.1 Oracle Data Types

One of the main functions of an OCI program is to communicate with an Oracle database.

The OCI application may retrieve data from database tables through SQL `SELECT` queries, or it may modify existing data in tables through `INSERT`, `UPDATE`, or `DELET` statements.

Inside a database, values are stored in columns in tables. Internally, Oracle represents data in particular formats known as *internal data types*. Examples of internal data types include `NUMBER`, `CHAR`, and `DATE` (see Table 4-1).

In general, OCI applications do not work with internal data type representations of data, but with host language data types that are predefined by the language in which they are written. When data is transferred between an OCI client application and a database table, the OCI libraries convert the data between internal data types and *external data types*.

External data types are host language types that have been defined in the OCI header files. When an OCI application binds input variables, one of the bind parameters is an indication of the external data type code (or *SQLT code*) of the variable. Similarly, when output variables are specified in a define call, the external representation of the retrieved data must be specified.

In some cases, external data types are similar to internal types. External types provide a convenience for the developer by making it possible to work with host language types instead of proprietary data formats.

Note: Even though some external types are similar to internal types, an OCI application never binds to internal data types. They are discussed here because it can be useful to understand how internal types can map to external types.

OCI can perform a wide range of data type conversions when transferring data between an Oracle database and an OCI application. There are more OCI external data types than Oracle internal data types. In some cases, a single external type maps to an internal type; in other cases, multiple external types map to a single internal type.

The many-to-one mappings for some data types provide flexibility for the OCI programmer. For example, suppose that you are processing the following SQL statement:

```
SELECT sal FROM emp WHERE empno = :employee_number
```

You want the salary to be returned as character data, instead of a binary floating-point format. Therefore, you specify an Oracle database external string data type, such as `VARCHAR2` (code = 1) or `CHAR` (code = 96) for the `dty` parameter in the `OCIDefineByPos()` or `OCIDefineByPos2()` call for the `sal` column. You also must declare a string variable in your program and specify its address in the `valuep` parameter. See Table 4-2 for more information.

If you want the salary information to be returned as a binary floating-point value, however, specify the `FLOAT` (code = 4) external data type. You also must define a variable of the appropriate type for the `valuep` parameter.

Oracle Database performs most data conversions transparently. The ability to specify almost any external data type provides a lot of power for performing specialized tasks. For example, you can input and output `DATE` values in pure binary format, with no character conversion involved, by using the `DATE` external data type.

To control data conversion, you must use the appropriate external data type codes in the bind and define routines. You must tell Oracle Database where the input or output variables are in your OCI program and their data types and lengths.

OCI also supports an additional set of OCI typecodes that are used by the Oracle Database type management system to represent data types of object type attributes. Y use a set of predefined constants to represent these typecodes. The constants each contain the prefix `OCI_TYPECODE`.

In summary, the OCI developer must be aware of the following different data types or data representations:

- Internal Oracle data types, which are used by table columns in an Oracle database. These also include data types used by PL/SQL that are not used by Oracle Database columns (for example, indexed table, record).
- External OCI data types, which are used to specify host language representations of Oracle data.
- `OCI_TYPECODE` values, which are used by Oracle Database to represent type information for object type attributes.

Information about a column's internal data type is conveyed to your application in the form of an internal data type code. With this information about what type of data is to be returned, your application can determine how to convert and format the output data. The Oracle internal data type codes are listed in the section Internal Data Types.

See Also:

- DATE for a description of the external data type
- Internal Data Types
- External Data Types and About Using External Data Type Codes
- Typecodes, and Relationship Between SQLT and OCI_TYPECODE Values
- *Oracle Database SQL Language Reference* for detailed information about Oracle internal data types
- About Describing Select-List Items for information about describing select-list items in a query

4.1.1 About Using External Data Type Codes

An external data type code indicates to Oracle Database how a host variable represents data in your program.

This determines how the data is converted when it is returned to output variables in your program, or how it is converted from input (bind) variables to Oracle Database column values. For example, to convert a `NUMBER` in an Oracle database column to a variable-length character array, you specify the `VARCHAR2` external data type code in the `OCIDefineByPos()` call that defines the output variable.

To convert a bind variable to a value in an Oracle Database column, specify the external data type code that corresponds to the type of the bind variable. For example, to input a character string such as 02-FEB-65 to a `DATE` column, specify the data type as a character string and set the length parameter to 9.

It is always the programmer's responsibility to ensure that values are convertible. If you try to insert the string `"MY BIRTHDAY"` into a `DATE` column, you get an error when you execute the statement.

See Also:

Table 4-2 for a complete list of the external data types and data type codes

4.2 Internal Data Types

Lists and describes the internal data types.

Table 4-1 lists the internal Oracle Database data types (also known as *built-in*), along with each type's maximum internal length and data type code. PL/SQL types listed in Table 4-11 and Table 4-12 are also considered to be internal data types.

Table 4-1 Internal Oracle Database Data Types

Internal Oracle Database Data Type	Maximum Internal Length	Data Type Code
`VARCHAR2, NVARCHAR2`	4000 bytes (standard) 32767 bytes (extended)	1
`NUMBER`	21 bytes	2
`LONG`	2^31-1 bytes (2 gigabytes)	8
`DATE`	7 bytes	12
`RAW`	2000 bytes (standard) 32767 bytes (extended)	23
`LONG RAW`	2^31-1 bytes	24
`ROWID`	10 bytes	69
`CHAR, NCHAR`	2000 bytes	96
`BINARY_FLOAT`	4 bytes	100
`BINARY_DOUBLE`	8 bytes	101

Internal Oracle Database Data Type	Maximum Internal Length	Data Type Code
User-defined type (object type, VARRAY, `nested table`)	Not Applicable	108
`REF`	Not Applicable	111
`CLOB, NCLOB`	128 terabytes	112
`BLOB`	128 terabytes	113
`BFILE`	Maximum operating system file size or UB8MAXVAL	114
`JSON`	32 MB	119
`TIMESTAMP`	11 bytes	180
`TIMESTAMP WITH TIME ZONE`	13 bytes	181
`INTERVAL YEAR TO MONTH`	5 bytes	182
`INTERVAL DAY TO SECOND`	11 bytes	183
`UROWID`	3950 bytes	208
`TIMESTAMP WITH LOCAL TIME ZONE`	11 bytes	231

This section includes the following topics:

- LONG, RAW, LONG RAW, VARCHAR2
- Character Strings and Byte Arrays
- UROWID
- BINARY_FLOAT and BINARY_DOUBLE
- JSON

See Also:

Oracle Database SQL Language Reference for more information about these built-in data types

4.2.1 LONG, RAW, LONG RAW, VARCHAR2

Use piecewise capabilities provided by specific OCI APIs to perform inserts, updates or fetches of these data types.

You can use the piecewise capabilities provided by `OCIBindByName()` or `OCIBindByName2()`, `OCIBindByPos()` or `OCIBindByPos2()`, `OCIDefineByPos()` or `OCIDefineByPos2()`, `OCIStmtGetPieceInfo()`, and `OCIStmtSetPieceInfo()` to perform inserts, updates or fetches involving column data of the `LONG`, `RAW`, `LONG RAW`, and `VARCHAR2` data types.

See Also:

- OCIBindByName() or OCIBindByName2()
- OCIBindByPos() or OCIBindByPos2()
- OCIDefineByPos() or OCIDefineByPos2()
- OCIStmtGetPieceInfo()
- OCIStmtSetPieceInfo()

4.2.2 Character Strings and Byte Arrays

Use Oracle internal data types to specify columns that contain characters or arrays of bytes.

You can use following Oracle internal data types to specify columns that contain characters or arrays of bytes: `CHAR`, `VARCHAR2`, `RAW`, `LONG`, and `LONG RAW`.

Note: LOBs can contain characters and `BFILE`s can contain binary data. They are handled differently than other types, so they are not included in this discussion.

`CHAR`, `VARCHAR2`, and `LONG` columns normally hold character data. `RAW` and `LONG RAW` hold bytes that are not interpreted as characters (for example, pixel values in a bit-mapped graphic image). Character data can be transformed when it is passed through a gateway between networks. Character data passed between machines using different languages, where single characters may be represented by differing numbers of bytes, can be significantly changed in length. Raw data is never converted in this way.

It is the responsibility of the database designer to choose the appropriate Oracle internal data type for each column in the table. The OCI programmer must be aware of the many possible ways that character and byte-array data can be represented and converted between variables in the OCI program and Oracle Database tables.

When an array holds characters, the length parameter for the array in an OCI call is always passed in and returned in bytes, not characters.

See Also:

LOB and BFILE Operations for more information about CHAR, VARCHAR2, RAW, LONG, and LONG RAW data types

4.2.3 UROWID

The Universal ROWID (UROWID) is a data type that can store both logical and physical rowids of Oracle Database tables.

Logical rowids are primary key-based logical identifiers for the rows of index-organized tables (IOTs).

To use columns of the UROWID data type, the value of the COMPATIBLE initialization parameter must be set to 8.1 or later.

The following host variables can be bound to Universal ROWIDs:

- SQLT_CHR (VARCHAR2)
- SQLT_VCS (VARCHAR)
- SQLT_STR (NULL-terminated string)
- SQLT_LVC (LONG VARCHAR)
- SQLT_AFC (CHAR)
- SQLT_AVC (CHARZ)
- SQLT_VST (OCI String)
- SQLT_RDD (ROWID descriptor)

4.2.4 BINARY_FLOAT and BINARY_DOUBLE

The BINARY_FLOAT and BINARY_DOUBLE data types represent single-precision and double-precision floating point values that mostly conform to the IEEE754 Standard for Floating-Point Arithmetic.

Prior to the addition of these data types with release 10.1, all numeric values in an Oracle Database were stored in the Oracle NUMBER format. These new binary floating point types do not replace Oracle NUMBER. Rather, they are alternatives to Oracle NUMBER that provide the advantage of using less disk storage.

These internal types are represented by the following codes:

- SQLT_IBFLOAT for BINARY_FLOAT
- SQLT_IBDOUBLE for BINARY_DOUBLE

All the following host variables can be bound to `BINARY_FLOAT` and `BINARY_DOUBLE` data types:

- `SQLT_BFLOAT` (native float)
- `SQLT_BDOUBLE` (native double)
- `SQLT_INT` (integer)
- `SQLT_FLT` (float)
- `SQLT_NUM` (Oracle `NUMBER`)
- `SQLT_UIN` (unsigned)
- `SQLT_VNU` (`VARNUM`)
- `SQLT_CHR` (`VARCHAR2`)
- `SQLT_VCS` (`VARCHAR`)
- `SQLT_STR` (`NULL`-terminated String)
- `SQLT_LVC` (`LONG VARCHAR`)
- `SQLT_AFC` (`CHAR`)
- `SQLT_AVC` (`CHARZ`)
- `SQLT_VST` (OCIString)

For best performance, use external types `SQLT_BFLOAT` and `SQLT_BDOUBLE` in conjunction with the `BINARY_FLOAT` and `BINARY_DOUBLE` data types.

4.2.5 JSON

Release 21c introduces a dedicated `JSON` data type.

JSON is a new SQL and PL/SQL data type for JSON data. The data is stored in the database in a binary form for faster access to nested JSON values.

You can use JSON data type and its instances in most places where a SQL data type is allowed, such as in following cases:

- As the column type for table
- View DDL
- As a parameter type for a PL/SQL subprogram
- In expressions where a SQL/JSON function or conditions are allowed

See Also:

- Overview of JSON in Oracle Database

4.3 External Data Types

Lists and describes the data type codes for external data types.

Table 4-2 lists data type codes for external data types. For each data type, the table lists the program variable types for C from or to which Oracle Database internal data is normally converted.

Table 4-2 External Data Types and Codes

External Data Type	Code	Program Variable(1)	OCI-Defined Constant
`boolean`	252	bool	`SQLT_BOL`
`VARCHAR2`	1	char[n]	`SQLT_CHR`
`NUMBER`	2	unsigned char[21]	`SQLT_NUM`
8-bit signed `INTEGER`	3	signed char	`SQLT_INT`
16-bit signed `INTEGER`	3	signed short, signed int	`SQLT_INT`
32-bit signed `INTEGER`	3	signed int, signed long	`SQLT_INT`
64-bit signed `INTEGER`	3	signed long, signed long long	`SQLT_INT`
`FLOAT`	4	float, double	`SQLT_FLT`
`NULL`-terminated `STRING`	5	char[n+1]	`SQLT_STR`
`VARNUM`	6	char[22]	`SQLT_VNU`
`LONG`	8	char[n]	`SQLT_LNG`
`VARCHAR`	9	char[n+sizeof(short integer)]	`SQLT_VCS`

External Data Type	Code	Program Variable(1)	OCI-Defined Constant
DATE	12	char[7]	SQLT_DAT
VARRAW	15	unsigned char[n+sizeof(short integer)]	SQLT_VBI
native float	21	float	SQLT_BFLOAT
native double	22	double	SQLT_BDOUBLE
RAW	23	unsigned char[n]	SQLT_BIN
LONG RAW	24	unsigned char[n]	SQLT_LBI
UNSIGNED INT	68	unsigned	SQLT_UIN
LONG VARCHAR	94	char[n+sizeof(integer)]	SQLT_LVC
LONG VARRAW	95	unsigned char[n+sizeof(integer)]	SQLT_LVB
CHAR	96	char[n]	SQLT_AFC
CHARZ	97	char[n+1]	SQLT_AVC
ROWID descriptor	104	OCIRowid *	SQLT_RDD
NAMED DATATYPE	108	struct	SQLT_NTY
REF	110	OCIRef	SQLT_REF
Character LOB descriptor	112	OCILobLocator[Foot 2]	SQLT_CLOB
Binary LOB descriptor	113	OCILobLocator[Foot 2]	SQLT_BLOB
Binary FILE descriptor	114	OCILobLocator	SQLT_FILE
JSON descriptor	119	OCIJson	SQLT_JSON
OCI STRING type	155	OCIString	SQLT_VST[Foot 3]
OCI DATE type	156	OCIDate *	SQLT_ODT[Foot 3]

External Data Type	Code	Program Variable(1)	OCI-Defined Constant
ANSI DATE descriptor	184	OCIDateTime *	SQLT_DATE
TIMESTAMP descriptor	187	OCIDateTime *	SQLT_TIMESTAMP
TIMESTAMP WITH TIME ZONE descriptor	188	OCIDateTime *	SQLT_TIMESTAMP_TZ
INTERVAL YEAR TO MONTH descriptor	189	OCIInterval *	SQLT_INTERVAL_YM
INTERVAL DAY TO SECOND descriptor	190	OCIInterval *	SQLT_INTERVAL_DS
TIMESTAMP WITH LOCAL TIME ZONE descriptor	232	OCIDateTime *	SQLT_TIMESTAMP_LTZ

Footnote 1 Where the length is shown as n, it is a variable, and depends on the requirements of the program (or of the operating system for ROWID).

Footnote 2

In applications using data type mappings generated by OTT, CLOBs may be mapped as OCIClobLocator, and BLOBs may be mapped as OCIBlobLocator. For more information, see Chapter 15.

Footnote 3

For more information about the use of these data types, see Chapter 12.

This section includes the following topics describing these external data types:

- BOOLEAN
- VARCHAR2
- NUMBER
- 64-Bit Integer Host Data Type
- INTEGER
- FLOAT
- STRING
- VARNUM
- LONG
- VARCHAR

4.3.1 BOOLEAN

The BOOLEAN data type stores boolean (TRUE or FALSE) values as a single byte.

The valid values for BOOLEAN datatype are TRUE (1), FALSE (0), or NULL. You can select or update boolean values as integral types, such as char, short, int, long, unsigned short, unsigned int, unsigned long, unsigned char.

To check if NULL is returned, include an indicator parameter in the `OCIDefineByPos()` call when using the BOOLEAN data types.

Oracle Database sets the indicator parameter to `-1` when a NULL value is fetched.

4.3.2 VARCHAR2

The `VARCHAR2` data type is a variable-length string of characters with a maximum length of 4000 bytes.

If the `init.ora` parameter `max_string_size = standard` (default value), the maximum length of a `VARCHAR2` can be 4000 bytes. If the `init.ora` parameter `max_string_size = extended`, the maximum length of a `VARCHAR2` can be 32767 bytes.

Note: If you are using Oracle Database objects, you can work with a special `OCIString` external data type using a set of predefined OCI functions.

This section includes the following topics:

- Input
- Output

See Also:

- `init.ora` parameter MAX_STRING_SIZE in *Oracle Database Reference* for more information about extended data types
- Object-Relational Data Types in OCI for more information about the `OCIString` external data type

4.3.2.1 Input

The `value_sz` parameter determines the length in the `OCIBindByName()` or `OCIBindByPos()` call. If you are using extended `VARCHAR2` lengths, then the `value_sz` parameter determines the length in the `OCIBindByName2()` and `OCIBindByPos2()` calls.

If the `value_sz` parameter is greater than zero, Oracle Database obtains the bind variable value by reading exactly that many bytes, starting at the buffer address in your program. Trailing blanks are stripped, and the resulting value is used in the SQL statement or PL/SQL block. If, with an `INSERT` statement, the resulting value is longer than the defined length of the database column, the `INSERT` fails, and an error is returned.

Note: A trailing `NULL` is not stripped. Variables should be blank-padded but not `NULL`-terminated.

If the `value_sz` parameter is zero, Oracle Database treats the bind variable as a `NULL`, regardless of its actual content. Of course, a `NULL` must be allowed for the bind variable value in the SQL statement. If you try to insert a `NULL` into a column that has a `NOT NULL` integrity constraint, Oracle Database issues an error, and the row is not inserted.

When the Oracle internal (column) data type is `NUMBER`, input from a character string that contains the character representation of a number is legal. Input character strings are converted to internal numeric format. If the `VARCHAR2` string contains an illegal conversion character, Oracle Database returns an error and the value is not inserted into the database.

See Also:

- OCIBindByName()
- OCIBindByPos()
- OCIBindByName2()
- OCIBindByPos2()

4.3.2.2 Output

You must specify the desired length for the return value in `value_sz` for bind and define functions.

Specify the desired length for the return value in the `value_sz` parameter of the `OCIDefineByPos()` call, or the `value_sz` parameter of `OCIBindByName()` or `OCIBindByPos()` for PL/SQL blocks. If zero is specified for the length, no data is returned. If you are using extended `VARCHAR2` lengths, then the `value_sz` parameter determines the desired length for the return value in the `OCIDefineByPos2()` call, or in the `OCIBindByName2()` and `OCIBindByPos2()` calls for PL/SQL blocks.

If you omit the `rlenp` parameter of `OCIDefineByPos()`, returned values are blank-padded to the buffer length, and `NULL`s are returned as a string of blank characters. If `rlenp` is included, returned values are not blank-padded. Instead, their actual lengths are returned in the `rlenp` parameter.

To check if a `NULL` is returned or if character truncation has occurred, include an indicator parameter in the `OCIDefineByPos()` call. Oracle Database sets the indicator parameter to -1 when a `NULL` is fetched and to the original column length when the returned value is truncated. Otherwise, it is set to zero. If you do not specify an indicator parameter and a `NULL` is selected, the fetch call returns the error code `OCI_SUCCESS_WITH_INFO`. Retrieving diagnostic information for the error returns `ORA-1405`.

See Also:

- Indicator Variables
- OCIDefineByPos() or OCIDefineByPos2()
- OCIBindByName() or OCIBindByName2()
- OCIBindByPos() or OCIBindByPos2()

4.3.3 NUMBER

You should not need to use `NUMBER` as an external data type.

If you do use it as an external data type, Oracle Database returns numeric values in its internal 21-byte binary format and expects this format on input. The following discussion is included for completeness only.

Note: If you are using objects in an Oracle database, you can work with a special `OCINumber` data type using a set of predefined OCI functions.

Oracle Database stores values of the `NUMBER` data type in a variable-length format. The first byte is the exponent and is followed by 1 to 20 mantissa bytes. The high-order bit of the exponent byte is the sign bit; it is set for positive numbers, and it is cleared for negative numbers. The lower 7 bits represent the exponent, which is a base-100 digit with an offset of 65.

To calculate the decimal exponent, add 65 to the base-100 exponent and add another 128 if the number is positive. If the number is negative, you do the same, but subsequently the bits are inverted. For example, -5 has a base-100 exponent = 62 (0x3e). The decimal exponent is thus (~0x3e) -128 - 65 = 0xc1 -128 -65 = 193 -128 -65 = 0.

Each mantissa byte is a base-100 digit, in the range 1..100. For positive numbers, the digit has 1 added to it. So, the mantissa digit for the value 5 is 6. For negative numbers, instead of adding 1, the digit is subtracted from 101. So, the mantissa digit for the number -5 is 96 (101 - 5). Negative numbers have a byte containing 102 appended to the data bytes. However, negative numbers that have 20 mantissa bytes do not have the trailing 102 byte. Because the mantissa digits are stored in base 100, each byte can represent 2 decimal digits. The mantissa is normalized; leading zeros are not stored.

Up to 20 data bytes can represent the mantissa. However, only 19 are guaranteed to be accurate. The 19 data bytes, each representing a base-100 digit, yield a maximum precision of 38 digits for an Oracle `NUMBER`.

If you specify the data type code 2 in the `dty` parameter of an `OCIDefineByPos()` or `OCIDefineByPos2()` call, your program receives numeric data in this Oracle internal format. The output variable should be a 21-byte array to accommodate the largest possible number. Note that only the bytes that represent the number are returned. There is no blank padding or `NULL` termination. If you must know the number of bytes returned, use the `VARNUM` external data type instead of `NUMBER`.

See Also:

- OCINumber Examples
- VARNUM for a description of the internal `NUMBER` format
- Number (OCINumber) more information about the `OCINumber` data type
- OCIDefineByPos() or OCIDefineByPos2()

4.3.4 64-Bit Integer Host Data Type

You can bind and define integer values greater than 32-bit size (more than nine digits of precision) from and into a `NUMBER` column using a 64-bit native host variable and `SQLT_INT` or `SQLT_UIN` as the external data type in an OCI application.

Starting with release 11.2, OCI supports the ability to bind and define integer values greater than 32-bit size (more than nine digits of precision) from and into a `NUMBE` column using a 64-bit native host variable and `SQLT_INT` or `SQLT_UIN` as the external data type in an OCI application.

This feature enables an application to bind and define 8-byte native host variables using `SQLT_INT` or `SQLT_UIN` external data types in the OCI bind and define function calls on all platforms. The `OCIDefineByPos()` or `OCIDefineByPos2()`, `OCIBindByName()` or `OCIBindByName2()`, and `OCIBindByPos()` or `OCIBindByPos2()` function calls can specify an 8-byte integer data type pointer as the `valuep` parameter. This feature enables you to insert and fetch large integer values (up to 18 decimal digits of precision) directly into and from native host variables and to perform free arithmetic on them.

This section includes the following topics:

- OCI Bind and Define for 64-Bit Integers
- Support for OUT Bind DML Returning Statements
- OCIDefineByPos() or OCIDefineByPos2()
- OCIBindByName() or OCIBindByName2()
- OCIBindByPos() or OCIBindByPos2()

4.3.4.1 OCI Bind and Define for 64-Bit Integers

Shows a code fragment for an OCI bind and define for 64-bit integers.

Example 4-1 shows a code fragment that works without errors.

Example 4-1 OCI Bind and Define Support for 64-Bit Integers

Copy

```
...
/* Variable declarations */
orasb8    sbigval1, sbigval2, sbigval3; // Signed 8-byte variables.
oraub8    ubigval1, ubigval2, ubigval3; // Unsigned 8-byte variables.
...
/* Bind Statements */
OCIBindByPos(..., (void *) &sbigval1, sizeof(sbigval1), ..., SQLT_INT, ...);
OCIBindByPos(..., (void *) &ubigval1, sizeof(ubigval1), ..., SQLT_UIN, ...);
OCIBindByName(...,(void *) &sbigval2, sizeof(sbigval2), ..., SQLT_INT, ...);
OCIBindByName(...,(void *) &ubigval2, sizeof(ubigval2), ..., SQLT_UIN, ...);
...
/* Define Statements */
OCIDefineByPos(..., (void *) &sbigval3, sizeof(sbigval3), ..., SQLT_INT, ...);
OCIDefineByPos(..., (void *) &ubigval3, sizeof(ubigval3), ..., SQLT_UIN, ...);
...
```

4.3.4.2 Support for OUT Bind DML Returning Statements

Shows a code fragment that illustrates binding 8-byte integer data types for OUT binds of a DML returning statement.

Example 4-2 shows a code fragment that illustrates binding 8-byte integer data types for OUT binds of a DML returning statement.

Example 4-2 Binding 8-Byte Integer Data Types for OUT Binds of a DML Returning Statement

```
...
/* Define SQL statements to be used in program. */
static text *dml_stmt = (text *) " UPDATE emp SET sal = sal + :1
                                   WHERE empno = :2
                                   RETURNING sal INTO :out1";
...

/* Declare all handles to be used in program. */
OCIStmt    *stmthp;
OCIError   *errhp;
OCIBind    *bnd1p   = (OCIBind *) 0;
OCIBind    *bnd2p   = (OCIBind *) 0;
OCIBind    *bnd3p   = (OCIBind *) 0;
...

/* Bind variable declarations */
orasb8    sbigval;   // OUT bind variable (8-byte size).
sword     eno, hike; // IN bind variables.
...

/* get values for IN bind variables */
...

/* Bind Statements */
OCIBindByPos(stmthp, &bnd1p, errhp, 1, (dvoid *) &hike,
            (sb4) sizeof(hike), SQLT_INT, (dvoid *) 0,
            (ub2 *) 0, (ub2 *) 0, (ub4) 0, (ub4 *) 0, OCI_DEFAULT);
OCIBindByPos(stmthp, &bnd2p, errhp, 2, (dvoid *) &eno,
            (sb4) sizeof(eno), SQLT_INT, (dvoid *) 0,
            (ub2 *) 0, (ub2 *) 0, (ub4) 0, (ub4 *) 0, OCI_DEFAULT);
OCIBindByName(stmthp, &bnd3p, errhp, (text *) ":out1", -1,
             (dvoid *) &sbigval, sizeof(sbigval), SQLT_INT, (dvoid *) 0,
             (ub2 *) 0, (ub2 *) 0, (ub4) 0, (ub4 *) 0, OCI_DEFAULT);
...

/* Use the returned OUT bind variable value */
...
```

4.3.5 INTEGER

The `INTEGER` data type converts numbers.

An external integer is a signed binary number; the size in bytes is system-dependent. The host system architecture determines the order of the bytes in the variable. A length specification is required for input and output. If the number being returned from Oracle Database is not an integer, the fractional part is discarded, and no error or other indication is returned. If the number to be returned exceeds the capacity of a signed integer for the system, Oracle Database returns an "overflow on conversion" error.

4.3.6 FLOAT

The `FLOAT` data type processes numbers that have fractional parts or that exceed the capacity of an integer.

The number is represented in the host system's floating-point format. Normally the length is either 4 or 8 bytes. The length specification is required for both input and output.

The internal format of an Oracle number is decimal, and most floating-point implementations are binary; therefore, Oracle Database can represent numbers with greater precision than floating-point representations.

Note: You may receive a round-off error when converting between `FLOAT` and `NUMBER`. Using a `FLOAT` as a bind variable in a query may return an `ORA-1403` error. You can avoid this situation by converting the `FLOAT` into a `STRING` and then using `VARCHAR2` or a `NULL`-terminated string for the operation.

4.3.7 STRING

The `NULL`-terminated `STRING` format behaves like the `VARCHAR2` format, except that the string must contain a `NULL` terminator character.

The `STRING` data type is most useful for C language programs.

This section includes the following topics:

- Input
- Output

4.3.7.1 Input

The string length supplied in the `OCIBindByName()` or `OCIBindByPos()` call limits the scan for the `NULL` terminator.

If the `NULL` terminator is not found within the length specified, Oracle Database issues the following error:

ORA-01480: trailing `NULL` missing from `STR` bind value

If the length is not specified in the bind call, OCI uses an implied maximum string length of 4000.

The minimum string length is 2 bytes. If the first character is a `NULL` terminator and the length is specified as 2, a `NULL` is inserted into the column, if permitted. Unlike

VARCHAR2 and CHAR, a string containing all blanks is not treated as a NULL on input; it is inserted as is.

Note: You cannot pass -1 for the string length parameter of a NULL-terminated string

4.3.7.2 Output

A NULL terminator is placed after the last character returned.

If the string exceeds the field length specified, it is truncated and the last character position of the output variable contains the NULL terminator.

A NULL select-list item returns a NULL terminator character in the first character position. An ORA-01405 error is also possible.

4.3.8 VARNUM

The VARNUM data type is like the external NUMBER data type, except that the first byte contains the length of the number representation.

This length does not include the length byte itself. Reserve 22 bytes to receive the longest possible VARNUM. Set the length byte when you send a VARNUM value to Oracle Database.

Table 4-3 shows several examples of the VARNUM values returned for numbers in a table.

Table 4-3 VARNUM Examples

Decimal Value	Length Byte	Exponent Byte	Mantissa Bytes	Terminator Byte
0	1	128	Not applicable	Not applicable
5	2	193	6	Not applicable
-5	3	62	96	102
2767	3	194	28, 68	Not applicable
-2767	4	61	74, 34	102
100000	2	195	11	Not applicable
1234567	5	196	2, 24, 46, 68	Not applicable

4.3.9 LONG

The `LONG` data type stores character strings longer than 4000 bytes.

You can store up to 2 gigabytes (`2^31-1` bytes) in a `LONG` column. Columns of this type are used only for storage and retrieval of long strings. They cannot be used in functions, expressions, or `WHERE` clauses. `LONG` column values are generally converted to and from character strings.

Do not create tables with `LONG` columns. Use `LOB` columns (`CLOB`, `NCLOB`, or `BLOB`) instead. `LONG` columns are supported only for backward compatibility.

Oracle also recommends that you convert existing `LONG` columns to `LOB` columns. `LOB` columns are subject to far fewer restrictions than `LONG` columns. Furthermore, `LOB` functionality is enhanced in every release, but `LONG` functionality has been static for several releases.

4.3.10 VARCHAR

The `VARCHAR` data type stores character strings of varying length.

The first 2 bytes contain the length of the character string, and the remaining bytes contain the string. The specified length of the string in a bind or a define call must include the two length bytes, so the largest `VARCHAR` string that can be received or sent is 65533 bytes long, not 65535.

4.3.11 DATE

The `DATE` data type can update, insert, or retrieve a date value using the Oracle internal date binary format.

A date in binary format contains 7 bytes, as shown in Table 4-4.

Table 4-4 Format of the DATE Data Type

Byte	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Meaning	Century	Year	Month	Day	Hour	Minute	Second
Example (for 30-NOV-1992, 3:17 PM)	119	192	11	30	16	18	1

The century and year bytes (bytes 1 and 2) are in excess-100 notation. The first byte stores the value of the year, which is 1992, as an integer, divided by 100, giving 119 in excess-100 notation. The second byte stores year modulo 100, giving 192. Dates Before Common Era (BCE) are less than 100. The era begins on 01-JAN-4712 BCE, which is Julian day 1. For this date, the century byte is 53, and the year byte is 88. The hour, minute, and second bytes are in excess-1 notation. The hour byte ranges from 1 to 24, the minute and second bytes from 1 to 60. If no time was specified when the date was created, the time defaults to midnight (1, 1, 1).

When you enter a date in binary format using the `DATE` external data type, the database does not do consistency or range checking. All data in this format must be carefully validated before input.

Note: There is little need to use the Oracle external `DATE` data type in ordinary database operations. It is much more convenient to convert `DATE` into character format, because the program usually deals with data in a character format, such as DD-MON-YY.

When a `DATE` column is converted to a character string in your program, it is returned using the default format mask for your session, or as specified in the `INIT.ORA` file.

If you are using objects in an Oracle database, you can work with a special `OCIDate` data type using a set of predefined OCI functions.

See Also:

- Date (OCIDate) for more information about the `OCIDate` data type
- Datetime and Interval Data Type Descriptors for information about `DATETIME` and `INTERVAL` data types

4.3.12 RAW

The `RAW` data type is used for binary data or byte strings that are not to be interpreted by Oracle Database, for example, to store graphics character sequences.

The maximum length of a `RAW` column is 2000 bytes. If the `init.ora` parameter `max_string_size = standard` (default value), the maximum length of a `RAW` can be 2000 bytes. If the `init.ora` parameter `max_string_size = extended`, the maximum length of a `RAW` can be 32767 bytes.

When `RAW` data in an Oracle Database table is converted to a character string in a program, the data is represented in hexadecimal character code. Each byte of the `RAW` data is returned as two characters that indicate the value of the byte, from '00' to 'FF'. To input a character string in your program to a `RAW` column in an Oracle Database table, you must code the data in the character string using this hexadecimal code.

You can use the piecewise capabilities provided by `OCIDefineByPos()`, `OCIBindByName()`, `OCIBindByPos()`, `OCIStmtGetPieceInfo()`, and `OCIStmtSetPieceInfo()` to perform inserts, updates, or fetches involving `RAW` (or `LONG RAW`) columns.

If you are using objects in an Oracle database, you can work with a special `OCIRaw` data type using a set of predefined OCI functions.

See Also:

- *Oracle Database SQL Language Reference* for more information about `MAX_STRING_SIZE`
- `init.ora` parameter MAX_STRING_SIZE in *Oracle Database Reference* for more information about extended data types
- Raw (OCIRaw)for more information about this data type

4.3.13 VARRAW

The `VARRAW` data type is similar to the `RAW` data type.

However, the first 2 bytes contain the length of the data. The specified length of the string in a bind or a define call must include the two length bytes, so the largest `VARRAW` string that can be received or sent is 65533 bytes, not 65535. For converting longer strings, use the `LONG VARRAW` external data type.

4.3.14 LONG RAW

The `LONG RAW` data type supports a 2 gigabyte length.

The `LONG RAW` data type is similar to the `RAW` data type, except that it stores raw data with a length up to 2 gigabytes (`2^31-1` bytes).

4.3.15 UNSIGNED

The `UNSIGNED` data type is used for unsigned binary integers.

The size in bytes is system-dependent. The host system architecture determines the order of the bytes in a word. A length specification is required for input and output. If the number being output from Oracle Database is not an integer, the fractional part is discarded, and no error or other indication is returned. If the number to be returned exceeds the capacity of an unsigned integer for the system, Oracle Database returns an "overflow on conversion" error.

4.3.16 LONG VARCHAR

The `LONG VARCHAR` data type stores data from and into an Oracle Database `LONG` column.

The first 4 bytes of a `LONG VARCHAR` contain the length of the item. So, the maximum length of a stored item is `2^31-5` bytes.

4.3.17 LONG VARRAW

The `LONG VARRAW` data type is used to store data from and into an Oracle Database `LONG RAW` column.

The length is contained in the first four bytes. The maximum length is `2^31-5` bytes.

4.3.18 CHAR

The `CHAR` data type is a string of characters, with a maximum length of 2000.

`CHAR` strings are compared using blank-padded comparison semantics.

This section includes the following topics:

- Input
- Output

See Also:

Oracle Database SQL Language Reference

4.3.18.1 Input

The length is determined by the `value_sz` parameter in the `OCIBindByName()` or `OCIBindByName2()` or `OCIBindByPos()` or `OCIBindByPos2()`call.

> **Note:** The entire contents of the buffer (`value_sz` chars) is passed to the database, including any trailing blanks or `NULL`s.

If the `value_sz` parameter is zero, Oracle Database treats the bind variable as a `NULL`, regardless of its actual content. Of course, a `NULL` must be allowed for the bind variable value in the SQL statement. If you try to insert a `NULL` into a column that has a `NOT NULL` integrity constraint, Oracle Database issues an error and does not insert the row.

Negative values for the `value_sz` parameter are not allowed for `CHAR`s.

When the Oracle internal (column) data type is `NUMBER`, input from a character string that contains the character representation of a number is legal. Input character strings are converted to internal numeric format. If the `CHAR` string contains an illegal conversion character, Oracle Database returns an error and does not insert the value. Number conversion follows the conventions established by globalization support settings for your system. For example, your system might be configured to recognize a comma (,) rather than a period (.) as the decimal point.

See Also:

- OCIBindByName() or OCIBindByName2()
- OCIBindByPos() or OCIBindByPos2()

4.3.18.2 Output

Specify the desired length for the return value in the `value_sz` parameter of the `OCIDefineByPos()` or `OCIDefineByPos2()` call.

If zero is specified for the length, no data is returned.

If you omit the `rlenp` parameter of `OCIDefineByPos()` or `OCIDefineByPos2()`, returned values are blank padded to the buffer length, and `NULL`s are returned as a of blank characters. If `rlenp` is included, returned values are not blank-padded. Instead, their actual lengths are returned in the `rlenp` parameter.

To check whether a `NULL` is returned or character truncation occurs, include an indicator parameter or array of indicator parameters in the `OCIDefineByPos()` or `OCIDefineByPos2()` call. An indicator parameter is set to `-1` when a `NULL` is fetched and to the original column length when the returned value is truncated. Otherwise, it is set to zero. If you do not specify an indicator parameter and a `NULL` is selected, the fetch call returns an `ORA-01405` error.

You can also request output to a character string from an internal `NUMBER` data type. Number conversion follows the conventions established by the globalization support settings for your system. For example, your system might use a comma (,) rather than a period (.) as the decimal point.

See Also:

- Indicator Variables
- OCIDefineByPos() or OCIDefineByPos2()

4.3.19 CHARZ

The `CHARZ` external data type is similar to the `CHAR` data type, except that the string must be `NULL`-terminated on input, and Oracle Database places a `NULL`-terminator character at the end of the string on output.

The `NULL` terminator serves only to delimit the string on input or output; it is not part of the data in the table.

On input, the length parameter must indicate the exact length, including the `NULL` terminator. For example, if an array in C is declared as follows, then the length parameter when you bind `my_num` must be seven. Any other value would return an error for this example.

```
char my_num[] = "123.45";
```

Copy

The following new external data types were introduced with or after release 8.0. These data types are not supported when you connect to an Oracle release 7 server.

Note: Both internal and external data types have Oracle-defined constant values, such as `SQLT_NTY`, `SQLT_REF`, corresponding to their data type codes. Although the constants are not listed for all of the types in this chapter, they are used in this section when discussing new Oracle data types. The data type constants are also used in other chapters of this guide when referring to these new types.

4.3.20 Named Data Types: Object, VARRAY, Nested Table

Named data types are user-defined types that are specified with the `CREATE TYPE` command in SQL.

Examples include object types, varrays, and nested tables. In OCI, *named data type* refers to a host language representation of the type. The `SQLT_NTY` data type code is used when binding or defining named data types.

In a C application, named data types are represented as C structs. These structs can be generated from types stored in the database by using the Object Type Translator. These types correspond to `OCI_TYPECODE_OBJECT`.

See Also:

- Object Type Information Storage and Access for more information about working with named data types in OCI
- Using the Object Type Translator with OCI for information about how named data types are represented as C structs

4.3.21 REF

This is a reference to a named data type.

The C language representation of a REF is a variable declared to be of type `OCIRef *`. The `SQLT_REF` data type code is used when binding or defining REFs.

Access to REFs is only possible when an OCI application has been initialized in object mode. When REFs are retrieved from the server, they are stored in the client-side object cache.

To allocate a REF for use in your application, you should declare a variable to be a pointer to a REF, and then call `OCIObjectNew()`, passing `OCI_TYPECODE_REF` as the `typecode` parameter.

See Also:

- OCIObjectNew()
- Object Advanced Topics in OCI for more information about working with REFs in the OCI

4.3.22 ROWID Descriptor

The `ROWID` data type identifies a particular row in a database table.

`ROWID` can be a select-list item in a query, such as:

```
SELECT ROWID, ename, empno FROM emp
```

In this case, you can use the returned `ROWID` in further `DELETE` statements.

If you are performing a `SELECT` for `UPDATE`, the `ROWID` is implicitly returned. This `ROWID` can be read into a user-allocated `ROWID` descriptor by using `OCIAttrGet()` on the statement handle and used in a subsequent `UPDATE` statement. The prefetch operation fetches all `ROWID`s on a `SELECT` for `UPDATE`; use prefetching and then a single row fetch.

You access rowids using a `ROWID` descriptor, which you can use as a bind or define variable.

See Also:

- OCIAttrGet()
- OCI Descriptors and Positioned Updates and Deletes for more information about the use of the `ROWID` descriptor

4.3.23 LOB Descriptor

A LOB (large object) stores binary or character data up to 128 terabytes (TB) in length.

Binary data is stored in a `BLOB` (binary LOB), and character data is stored in a `CLOB` (character LOB) or `NCLOB` (national character LOB).

LOB values may or may not be stored inline with other row data in the database. In either case, LOBs have the full transactional support of the Oracle database. A database table stores a *LOB locator* that points to the LOB value, which may be in a different storage space.

When an OCI application issues a SQL query that includes a LOB column or attribute in its select list, fetching the results of the query returns the locator, rather than the actual LOB value. In OCI, the LOB locator maps to a variable of type `OCILobLocator`.

> **Note:** Depending on your application, you may or may not want to use LOB locators. You can use the data interface for LOBs, which does not require LOB locators. In this interface, you can bind or define character data for `CLOB` columns or `RAW` data for `BLOB` columns.

The OCI functions for LOBs take a LOB locator as one of their arguments. The OCI functions assume that the locator has already been created, whether or not the LOB to which it points contains data.

Bind and define operations are performed on the LOB locator, which is allocated with the `OCIDescriptorAlloc()` function.

The locator is always fetched first using SQL or `OCIObjectPin()`, and then operations are performed using the locator. The OCI functions never take the actual LOB value as a parameter.

The data type codes available for binding or defining LOBs are:

- `SQLT_BLOB` - A binary LOB data type
- `SQLT_CLOB` - A character LOB data type

The `NCLOB` is a special type of `CLOB` with the following requirements:

- To write into or read from an `NCLOB`, the user must set the character set form (`csfrm`) parameter to be `SQLCS_NCHAR`.
- The amount (`amtp`) parameter in calls involving `CLOB`s and `NCLOB`s is always interpreted in terms of characters, rather than bytes, for fixed-width character sets.

This section includes the following topics:

- BFILE
- BLOB
- CLOB
- NCLOB

See Also:

- OCI Descriptors for more information about descriptors, including the LOB locator
- *Oracle Database SQL Language Reference* and *Oracle Database SecureFiles and Large Objects Developer's Guide* for more information about LOBs
- About Binding and Defining LOB Data
- About Defining LOB Data
- LOB and BFILE Functions in OCI
- OCIDescriptorAlloc()
- OCIObjectPin()
- LOB and BFILE Operations for more information about OCI LOB functions

4.3.23.1 BFILE

Oracle Database supports access to binary files (BFILEs).

The BFILE data type provides access to LOBs that are stored in file systems outside an Oracle database.

A BFILE column or attribute stores a file LOB locator, which serves as a pointer to a binary file on the server's file system. The locator maintains the directory object and the file name. The maximum size of a BFILE is the smaller of the operating system maximum file size or UB8MAXVAL.

Binary file LOBs do not participate in transactions. Rather, the underlying operating system provides file integrity and durability.

The database administrator must ensure that the file exists and that Oracle Database processes have operating system read permissions on the file.

The BFILE data type allows read-only support of large binary files; you cannot modify a file through Oracle Database. Oracle Database provides APIs to access file data.

The data type code available for binding or defining BFILEs is SQLT_BFILE (a binary FILE LOB data type)

See Also:

Oracle Database SecureFiles and Large Objects Developer's Guide for more information about directory aliases

4.3.23.2 BLOB

The BLOB data type stores unstructured binary large objects.

BLOBs can be thought of as bit streams with no character set semantics. BLOBs can store up to 128 terabytes of binary data.

BLOBs have full transactional support; changes made through OCI participate fully in the transaction. The BLOB value manipulations can be committed or rolled back. You cannot save a BLOB locator in a variable in one transaction and then use it in another transaction or session.

4.3.23.3 CLOB

The CLOB data type stores fixed-width or variable-width character data.

CLOBs can store up to 128 terabytes of character data.

CLOBs have full transactional support; changes made through OCI participate fully in the transaction. The CLOB value manipulations can be committed or rolled back. You cannot save a CLOB locator in a variable in one transaction and then use it in another transaction or session.

4.3.23.4 NCLOB

An NCLOB is a national character version of a CLOB.

It stores fixed-width, single-byte or multibyte national character set (NCHAR) data, or variable-width character set data. NCLOBs can store up to 128 terabytes of character text data.

NCLOBs have full transactional support; changes made through OCI participate fully in the transaction. NCLOB value manipulations can be committed or rolled back. You cannot save an NCLOB locator in a variable in one transaction and then use it in another transaction or session.

4.3.24 JSON Descriptor

JSON data type is used to store JSON data in a native binary format.

When an OCI application executes a SQL statement that includes a JSON column, the results fetched from the query populates the descriptor. In OCI, JSON maps to a variable of type OCIJson.

Bind and define operations are performed on the JSON descriptor, which is allocated with the OCIDescriptorAlloc() function. The data type code for binding and defining JSON is SQLT_JSON. This is referred to as JSON descriptor interface.

Apart from JSON descriptor interface, depending on your application, you can use either LOB locator or data interface to fetch the JSON value. In such cases, the conversion to textual JSON happens on the server.

The OCI functions for JSON has JSON descriptor as one of the arguments. You can read and write data from or to a JSON descriptor.

4.3.25 Datetime and Interval Data Type Descriptors

Lists and describes the datetime and interval data type descriptors.

The datetime and interval data type descriptors are briefly summarized here.

This section includes the following topics:

- ANSI DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE
- TIMESTAMP WITH LOCAL TIME ZONE
- INTERVAL YEAR TO MONTH
- INTERVAL DAY TO SECOND
- About Avoiding Unexpected Results Using Datetime

See Also:

Oracle Database SQL Language Reference

4.3.25.1 ANSI DATE

ANSI DATE is based on DATE, but contains no time portion. It also has no time zone.

ANSI DATE follows the ANSI specification for the DATE data type. When assigning an ANSI DATE to a DATE or a time stamp data type, the time portion of the Oracle DATE and the time stamp are set to zero. When assigning a DATE or a time stamp to an ANSI DATE, the time portion is ignored.

Instead of using the ANSI DATE data type, Oracle recommends that you use the TIMESTAMP data type, which contains both date and time.

4.3.25.2 TIMESTAMP

The TIMESTAMP data type is an extension of the DATE data type. It stores the year, month, and day of the DATE data type, plus the hour, minute, and second values.

The TIMESTAMP data type has no time zone. The TIMESTAMP data type has the following form:

```
TIMESTAMP(fractional_seconds_precision)
```

Copy

In this form, the optional fractional_seconds_precision specifies the number of digits in the fractional part of the SECOND datetime field and can be a number in the range 0 to 9. The default is 6.

4.3.25.3 TIMESTAMP WITH TIME ZONE

TIMESTAMP WITH TIME ZONE (TSTZ) is a variant of TIMESTAMP that includes an explicit time zone displacement in its value.

The time zone displacement is the difference in hours and minutes between local time and UTC (coordinated universal time—formerly Greenwich mean time). The TIMESTAMP WITH TIME ZONE data type has the following form:

```
TIMESTAMP(fractional_seconds_precision) WITH TIME ZONE
```

Copy

In this form, fractional_seconds_precision optionally specifies the number of digits in the fractional part of the SECOND datetime field, and can be a number in t range 0 to 9. The default is 6.

Two TIMESTAMP WITH TIME ZONE values are considered identical if they represent the same instant in UTC, regardless of the TIME ZONE offsets stored in the data.

4.3.25.4 TIMESTAMP WITH LOCAL TIME ZONE

TIMESTAMP WITH LOCAL TIME ZONE (TSLTZ) is another variant of TIMESTAMP that includes a time zone displacement in its value.

Storage is in the same format as for TIMESTAMP. This type differs from TIMESTAMP WITH TIME ZONE in that data stored in the database is normalized to the database time zone, and the time zone displacement is not stored as part of the column data. When retrieving the data, Oracle Database returns it in your local session time zone.

The time zone displacement is the difference (in hours and minutes) between local time and UTC (coordinated universal time—formerly Greenwich mean time). The TIMESTAMP WITH LOCAL TIME ZONE data type has the following form:

```
TIMESTAMP(fractional_seconds_precision) WITH LOCAL TIME ZONE
```

In this form, fractional_seconds_precision optionally specifies the number of digits in the fractional part of the SECOND datetime field and can be a number in the range 0 to 9. The default is 6.

4.3.25.5 INTERVAL YEAR TO MONTH

INTERVAL YEAR TO MONTH stores a period of time using the YEAR and MONTH datetime fields.

The INTERVAL YEAR TO MONTH data type has the following form:

```
INTERVAL YEAR(year_precision) TO MONTH
```

In this form, the optional year_precision is the number of digits in the YEAR datetime field. The default value of year_precision is 2.

4.3.25.6 INTERVAL DAY TO SECOND

INTERVAL DAY TO SECOND stores a period of time in terms of days, hours, minutes, and seconds.

The INTERVAL DAY TO SECOND data type has the following form:

```
INTERVAL DAY (day_precision) TO SECOND(fractional_seconds_precision)
```

In this form:

- `day_precision` is the number of digits in the `DAY` datetime field. It is optional. Accepted values are 0 to 9. The default is 2.
- `fractional_seconds_precision` is the number of digits in the fractional part of the `SECOND` datetime field. Accepted values are 0 to 9. The value should be provided as nanoseconds. The Default Day to Second precision is 6 unless the precision is specified to a different value at the time of creating the table. In this case, the least significant three digits will be truncated.

4.3.25.7 About Avoiding Unexpected Results Using Datetime

How to avoid unexpected results using datetime.

Note: To avoid unexpected results in your data manipulation language (DML) operations on datetime data, you can verify the database and session time zones by querying the built-in SQL functions `DBTIMEZONE` and `SESSIONTIMEZONE`. If the time zones have not been set manually, Oracle Database uses the operating system time zone by default. If the operating system time zone is not a valid Oracle Database time zone, Oracle Database uses UTC as the default value.

4.3.26 Native Float and Native Double

The native float (`SQLT_BFLOAT`) and native double (`SQLT_BDOUBLE`) data types represent the single-precision and double-precision floating-point values.

They are represented natively, that is, in the host system's floating-point format.

These external types were added in release 10.1 to externally represent the `BINARY_FLOAT` and `BINARY_DOUBLE` internal data types. Thus, performance for the internal types is best when used in conjunction with external types native float and native double respectively. This draws a clear distinction between the existing representation of floating-point values (`SQLT_FLT`) and these types.

4.3.27 C Object-Relational Data Type Mappings

OCI supports Oracle-defined C data types for mapping user-defined data types to C representations (for example, `OCINumber`, `OCIArray`).

OCI provides a set of calls to operate on these data types, and to use these data types in bind and define operations, in conjunction with OCI external data types.

See Also:

Object-Relational Data Types in OCI for information about using these Oracle-defined C data types

4.4 Data Conversions

Shows the supported conversions from internal data types to external data types and from external data types into internal column representations.

Table 4-5 shows the supported conversions from internal data types to external data types, and from external data types into internal column representations, for all data types available through release 7.3. Information about data conversions for data types newer than release 7.3 is listed here:

- REFs stored in the database are converted to `SQLT_REF` on output.
- `SQLT_REF` is converted to the internal representation of REFs on input.
- Named data types stored in the database can be converted to `SQLT_NTY` (and represented by a C struct in the application) on output.
- `SQLT_NTY` (represented by a C struct in an application) is converted to the internal representation of the corresponding type on input.

LOBs are shown in Table 4-6, because of the width limitation.

See Also:

Object-Relational Data Types in OCI for information about `OCIString`, `OCINumber`, and other new data types

Table 4-5 Data Conversions

NA(4)	INTERNAL DATA TYPES->	NA	NA	NA	NA	NA	NA	NA	NA
EXTERNALDATA TYPES	**VARCHAR2**	**NUMBER**	**LONG**	**ROWID**	**UROWID**	**DATE**	**RAW**	**LONG RAW**	**CHAR**
`VARCHAR2`	I/O[Foot 5]	I/O	I/O	I/O[Foot 6]	I/O[Foot 6]	I/O[Foot 7]	I/O[Foot 8]	I/O[Foot 8]	NA
`NUMBER`	I/O[Foot 9]	I/O	I[Foot 10]	NA	NA	NA	NA	NA	I/O[Foot 9]
`INTEGER`	I/O[Foot 9]	I/O	I	NA	NA	NA	NA	NA	I/O[Foot 9]
`FLOAT`	I/O[Foot 9]	I/O	I	NA	NA	NA	NA	NA	I/O[Foot 9]

NA(4)	**INTERNAL DATA TYPES->**	**NA**	**NA**	**NA**	**NA**	**NA**	**NA**	**NA**	**NA**
EXTERNALDATA TYPES	**VARCHAR2**	**NUMBER**	**LONG**	**ROWID**	**UROWID**	**DATE**	**RAW**	**LONG RAW**	**CHAR**
`STRING`	I/O	I/O	I/O	I/O[Foot 6]	I/O[Foot 6]	I/O[Foot 7]	I/O[Foot 8]	I/O[Foot 8, Foot 11]	I/O
`VARNUM`	I/O[Foot 9]	I/O	I	NA	NA	NA	NA	NA	I/O[Foot 9]
`DECIMAL`	I/O[Foot 9]	I/O	I	NA	NA	NA	NA	NA	I/O[Foot 9]
`LONG`	I/O	I/O	I/O	I/O[Foot 6]	I/O[Foot 6]	I/O[Foot 7]	I/O[Foot 8]	I/O[Foot 8, Foot 11]	I/O
`VARCHAR`	I/O	I/O	I/O	I/O[Foot 6]	I/O[Foot 6]	I/O[Foot 7]	I/O[Foot 8]	I/O[Foot 8, Foot 11]	I/O
`DATE`	I/O	NA	I	NA	NA	I/O	NA	NA	I/O
`VARRAW`	I/O[Foot 12]	NA	I[Foot 11, Foot 12]	NA	NA	NA	I/O	I/O	I/O[Foot 12]
`RAW`	I/O[Foot 12]	NA	I[Foot 11, Foot 12]	NA	NA	NA	I/O	I/O	I/O[Foot 12]
`LONG RAW`	O[Foot 13, Foot 12]	NA	I[Foot 11, Foot 12]	NA	NA	NA	I/O	I/O	O[Foot 12]
`UNSIGNED`	I/O[Foot 9]	I/O	I	NA	NA	NA	NA	NA	I/O[Foot 9]
`LONG VARCHAR`	I/O	I/O	I/O	I/O[Foot 6]	I/O[Foot 6]	I/O[Foot 7]	I/O[Foot 8]	I/O[Foot 8, Foot 11]	I/O
`LONG VARRAW`	I/O[Foot 12]	NA	I[Foot 11, Foot 12]	NA	NA	NA	I/O	I/O	I/O[Foot 12]
`CHAR`	I/O	I/O	I/O	I/O[Foot 6]	I/O[Foot 6]	I/O[Foot 7]	I/O[Foot 8]	I[Foot 8]	I/O
`CHARZ`	I/O	I/O	I/O	I/O[Foot 6]	I/O[Foot 6]	I/O[Foot 7]	I/O[Foot 8]	I[Foot 8]	I/O
`ROWID` descriptor	I[Foot 6]	NA	NA	I/O	I/O	NA	NA	NA	I[Foot 6]

Footnote 4 NA means not applicable.

Footnote 5

I/O = Conversion is valid for input or output.

Footnote 6

For input, host string must be in Oracle ROWID/UROWID format. On output, column value is returned in Oracle ROWID/UROWID format.

Footnote 7

For input, host string must be in the Oracle DATE character format. On output, column value is returned in Oracle DATE format.

Footnote 8

For input, host string must be in hexadecimal format. On output, column value is returned in hexadecimal format.

Footnote 9

For output, column value must represent a valid number.

Footnote 10

I = Conversion is valid for input only.

Footnote 11

Length must be less than or equal to 2000.

Footnote 12

On input, column value is stored in hexadecimal format. On output, column value must be in hexadecimal format.

Footnote 13

O = Conversion is valid for output only.

This section includes the following topics:

- Data Conversions for LOB Data Type Descriptors
- Data Conversions for Datetime and Interval Data Types
- Datetime and Date Upgrading Rules
- Data Conversion for BINARY_FLOAT and BINARY_DOUBLE in OCI

4.4.1 Data Conversions for LOB Data Type Descriptors

Shows the data conversions for LOBs.

Table 4-6 shows the data conversions for LOBs. For example, the external character data types (VARCHAR, CHAR, LONG, and LONG VARCHAR) convert to the internal CLOB data type, whereas the external raw data types (RAW, VARRAW, LONG RAW, and LONG VARRAW) convert to the internal BLOB data type.

Table 4-6 Data Conversions for LOBs

EXTERNAL DATA TYPES	INTERNAL CLOB	INTERNAL BLOB
VARCHAR	I/O[Foot 14]	NA[Foot 15]
CHAR	I/O	NA
LONG	I/O	NA
LONG VARCHAR	I/O	NA
RAW	NA	I/O
VARRAW	NA	I/O
LONG RAW	NA	I/O
LONG VARRAW	NA	I/O

Footnote 14

I/O = Conversion is valid for input or output.

Footnote 15

NA means not applicable.

4.4.2 Data Conversions for JSON Data Type

Shows the data conversion for JSON data type.

Table 4-7 Data Conversions for JSON Data Type

External Types/Internal Types	JSON
VARCHAR2	I/O[Foot 16]
CLOB	I/O
BLOB	I/O

Footnote 16

I/O = Conversion is valid for input or output.

4.4.3 Data Conversions for Datetime and Interval Data Types

Shows the data conversion for datetime and interval data types.

You can also use one of the character data types for the host variable used in a fetch or insert operation from or to a datetime or interval column. Oracle Database does the conversion between the character data type and datetime or interval data type for you (see Table 4-8.

Table 4-8 Data Conversion for Datetime and Interval Types

External Types/Internal Types	VARCHAR,CHAR	DATE	TS	TSTZ	TSLTZ	INTERVAL YEAR TO MONTH	INTERVAL DAY TO SECOND
VARCHAR2, CHAR	I/O[Foot 17]	I/O	I/O	I/O	I/O	I/O	I/O
DATE	I/O	I/O	I/O	I/O	I/O	NA[Foot 18]	NA
OCI DATE	I/O	I/O	I/O	I/O	I/O	NA	NA
ANSI DATE	I/O	I/O	I/O	I/O	I/O	NA	NA
TIMESTAMP (TS)	I/O	I/O	I/O	I/O	I/O	NA	NA
TIMESTAMP WITH TIME ZONE (TSTZ)	I/O	I/O	I/O	I/O	I/O	NA	NA
TIMESTAMP WITH LOCAL TIME ZONE (TSLTZ)	I/O	I/O	I/O	I/O	I/O	NA	NA
INTERVAL YEAR TO MONTH	I/O	NA	NA	NA	NA	I/O	NA

External Types/Internal Types	VARCHAR,CHAR	DATE	TS	TSTZ	TSLTZ	INTERVAL YEAR TO MONTH	INTERVAL DAY TO SECOND
`INTERVAL DAY TO SECOND`	I/O	NA	NA	NA	NA	NA	I/O

Footnote 17

I/O = Conversion is valid for input or output.

Footnote 18

NA means not applicable.

This section includes the following topics:

- Assignment Notes
- Data Conversion Notes for Datetime and Interval Types

4.4.3.1 Assignment Notes

When you assign a source with a time zone to a target without a time zone, the time zone portion of the source is ignored.

When you assign a source without a time zone to a target with a time zone, the time zone of the target is set to the session's default time zone.

When you assign an Oracle Database `DATE` to a `TIMESTAMP`, the `TIME` portion of the `DATE` is copied over to the `TIMESTAMP`. When you assign a `TIMESTAMP` to Oracle Database `DATE`, the `TIME` portion of the result `DATE` is set to zero. This is done to encourage upgrading of Oracle Database `DATE` to `ANSI`-compliant `DATETIME` data types.

When you assign an `ANSI DATE` to an Oracle `DATE` or a `TIMESTAMP`, the `TIME` portion of the Oracle Database `DATE` and the `TIMESTAMP` are set to zero. When you assign an Oracle Database `DATE` or a `TIMESTAMP` to an ANSI `DATE`, the `TIME` portion is ignored.

When you assign a `DATETIME` to a character string, the `DATETIME` is converted using the session's default `DATETIME` format. When you assign a character string to a `DATETIME`, the string must contain a valid `DATETIME` value based on the session's default `DATETIME` format

When you assign a character string to an `INTERVAL`, the character string must be a valid `INTERVAL` character format.

4.4.3.2 Data Conversion Notes for Datetime and Interval Types

Describes some information for datetime and interval types.

When you convert from `TSLTZ` to `CHAR`, `DATE`, `TIMESTAMP`, and `TSTZ`, the value is adjusted to the session time zone.

When you convert from `CHAR`, `DATE`, and `TIMESTAMP` to `TSLTZ`, the session time zone is stored in memory.

When you assign TSLTZ to ANSI DATE, the time portion is zero.

When you convert from TSTZ, the time zone that the time stamp is in is stored in memory.

When you assign a character string to an interval, the character string must be a valid interval character format.

4.4.4 Datetime and Date Upgrading Rules

OCI has full forward and backward compatibility between a client application and the Oracle database for datetime and date columns.

This section includes the following topics:

- Pre-9.0 Client with 9.0 or Later Server
- Pre-9.0 Server with 9.0 or Later Client

4.4.4.1 Pre-9.0 Client with 9.0 or Later Server

The only datetime data type available to a pre-9.0 application is the DATE data type, SQLT_DAT.

When a pre-9.0 client that defined a buffer as SQLT_DAT tries to obtain data from a TSLTZ column, only the date portion of the value is returned to the client.

4.4.4.2 Pre-9.0 Server with 9.0 or Later Client

When a pre-9.0 server is used with a 9.0 or later client, the client can have a bind or define buffer of type SQLT_TIMESTAMP_LTZ.

The following compatibilities are maintained in this case.

If any client application tries to insert a SQLT_TIMESTAMP_LTZ (or any of the new datetime data types) into a DATE column, an error is issued because there is potential data loss in this situation.

When a client has an OUT bind or a define buffer that is of data type SQLT_TIMESTAMP_LTZ and the underlying server-side SQL buffer or column is of DATE type, then the session time zone is assigned.

4.4.5 Data Conversion for BINARY_FLOAT and BINARY_DOUBLE in OCI

Shows the supported conversions between internal numeric data types and all relevant external types.

Table 4-9 shows the supported conversions between internal numeric data types and all relevant external types. An (I) implies that the conversion is valid for input only (binds), and (O) implies that the conversion is valid for output only (defines). An (I/O) implies that the conversion is valid for input and output (binds and defines).

Table 4-9 Data Conversion for External Data Types to Internal Numeric Data Types

External Types/Internal Types	BINARY_FLOAT	BINARY_DOUBLE
VARCHAR	I/O[Foot 19]	I/O
VARCHAR2	I/O	I/O
NUMBER	I/O	I/O
INTEGER	I/O	I/O
FLOAT	I/O	I/O
STRING	I/O	I/O
VARNUM	I/O	I/O
LONG	I/O	I/O
UNSIGNED INT	I/O	I/O
LONG VARCHAR	I/O	I/O
CHAR	I/O	I/O
BINARY_FLOAT	I/O	I/O
BINARY_DOUBLE	I/O	I/O

Footnote 19

An (I/O) implies that the conversion is valid for input and output (binds and defines)

Table 4-10 shows the supported conversions between all relevant internal types and numeric external types. An (I) implies that the conversion is valid for input only (only for binds), and (O) implies that the conversion is valid for output only (only for defines). An (I/O) implies that the conversion is valid for input and output (binds and defines).

Table 4-10 Data Conversions for Internal to External Numeric Data Types

Internal Types/External Types	Native Float	Native Double
VARCHAR2	I/O[Foot 20]	I/O

Internal Types/External Types	Native Float	Native Double
NUMBER	I/O	I/O
LONG	I[Foot 21]	I
CHAR	I/O	I/O
BINARY_FLOAT	I/O	I/O
BINARY_DOUBLE	I/O	I/O

Footnote 20

An (I/O) implies that the conversion is valid for input and output (binds and defines)

Footnote 21

An (I) implies that the conversion is valid for input only (only for binds)

4.5 Typecodes

A unique typecode is associated with each Oracle Database type, whether scalar, collection, reference, or object type.

This typecode identifies the type, and is used by Oracle Database to manage information about object type attributes. This typecode system is designed to be generic and extensible. It is not tied to a direct one-to-one mapping to Oracle data types. Consider the following SQL statements:

```
CREATE TYPE my_type AS OBJECT
( attr1    NUMBER,
  attr2    INTEGER,
  attr3    SMALLINT);

CREATE TABLE my_table AS TABLE OF my_type;
```

These statements create an object type and an object table. When it is created, `my_table` has three columns, all of which are of Oracle `NUMBER` type, because `SMALLINT` and `INTEGER` map internally to `NUMBER`. The internal representation of the attributes of `my_type`, however, maintains the distinction between the data types of the three attributes: `attr1` is `OCI_TYPECODE_NUMBER`, `attr2` is `OCI_TYPECODE_INTEGER`, and `attr3` is `OCI_TYPECODE_SMALLINT`. If an application describes `my_type`, these typecodes are returned.

OCITypeCode is the C data type of the typecode. The typecode is used by some OCI functions, like OCIObjectNew(), where it helps determine what type of object is created. It is also returned as the value of some attributes when an object is described; for example, querying the OCI_ATTR_TYPECODE attribute of a type returns an OCITypeCode value.

Table 4-11 lists the possible values for an OCITypeCode. There is a value corresponding to each Oracle data type.

Table 4-11 OCITypeCode Values and Data Types

Value	Data Type
OCI_TYPECODE_REF	REF
OCI_TYPECODE_DATE	DATE
OCI_TYPECODE_TIMESTAMP	TIMESTAMP
OCI_TYPECODE_TIMESTAMP_TZ	TIMESTAMP WITH TIME ZONE
OCI_TYPECODE_TIMESTAMP_LTZ	TIMESTAMP WITH LOCAL TIME ZONE
OCI_TYPECODE_INTERVAL_YM	INTERVAL YEAR TO MONTH
OCI_TYPECODE_INTERVAL_DS	INTERVAL DAY TO SECOND
OCI_TYPECODE_REAL	Single-precision real
OCI_TYPECODE_DOUBLE	Double-precision real
OCI_TYPECODE_FLOAT	Floating-point
OCI_TYPECODE_NUMBER	Oracle NUMBER
OCI_TYPECODE_BFLOAT	BINARY_FLOAT
OCI_TYPECODE_BDOUBLE	BINARY_DOUBLE
OCI_TYPECODE_DECIMAL	Decimal
OCI_TYPECODE_OCTET	Octet

Value	Data Type
`OCI_TYPECODE_INTEGER`	Integer
`OCI_TYPECODE_SMALLINT`	Small int
`OCI_TYPECODE_RAW`	RAW
`OCI_TYPECODE_VARCHAR2`	Variable string ANSI SQL, that is, VARCHAR2
`OCI_TYPECODE_VARCHAR`	Variable string Oracle SQL, that is, VARCHAR
`OCI_TYPECODE_CHAR`	Fixed-length string inside SQL, that is SQL CHAR
`OCI_TYPECODE_VARRAY`	Variable-length array (varray)
`OCI_TYPECODE_TABLE`	Multiset
`OCI_TYPECODE_CLOB`	Character large object (CLOB)
`OCI_TYPECODE_BLOB`	Binary large object (BLOB)
`OCI_TYPECODE_BFILE`	Binary large object file (`BFILE`)
`OCI_TYPECODE_OBJECT`	Named object type, or SYS.XMLType
`OCI_TYPECODE_NAMEDCOLLECTION`	Collection
`OCI_TYPECODE_BOOLEAN`[Foot 22]	Boolean
`OCI_TYPECODE_RECORD`[Foot 22]	Record
`OCI_TYPECODE_ITABLE`[Foot 22]	Index-by BINARY_INTEGER
`OCI_TYPECODE_INTEGER`[Foot 22]	PLS_INTEGER or BINARY_INTEGER

Footnote 22

This type is a PL/SQL type only.

This section includes the following topic: Relationship Between SQLT and OCI_TYPECODE Values.

4.5.1 Relationship Between SQLT and OCI_TYPECODE Values

Oracle Database recognizes two different sets of data type code values.

One set is distinguished by the SQLT_ prefix, the other by the OCI_TYPECODE_ prefix.

The SQLT typecodes are used by OCI to specify a data type in a bind or define operation, enabling you to control data conversions between Oracle Database and OCI client applications. The OCI_TYPECODE types are used by Oracle's type system to reference or describe predefined types when manipulating or creating user-defined types.

In many cases, there are direct mappings between SQLT and OCI_TYPECODE values. In other cases, however, there is not a direct one-to-one mapping. For example, OCI_TYPECODE_SIGNED8, OCI_TYPECODE_SIGNED16, OCI_TYPECODE_SIGNED32, OCI_TYPECODE_INTEGER, OCI_TYPECODE_OCTET, and OCI_TYPECODE_SMALLINT are all mapped to the SQLT_INT type.

Table 4-12 illustrates the mappings between SQLT and OCI_TYPECODE types.

Table 4-12 OCI_TYPECODE to SQLT Mappings

Oracle Type System Typename	Oracle Type System Type	Equivalent SQLT Type
BFILE	OCI_TYPECODE_BFILE	SQLT_BFILE
BLOB	OCI_TYPECODE_BLOB	SQLT_BLOB
BOOLEAN[Foot 23]	OCI_TYPECODE_BOOLEAN	SQLT_BOL
CHAR	OCI_TYPECODE_CHAR (n)	SQLT_AFC(n)[Foot 24]
CLOB	OCI_TYPECODE_CLOB	SQLT_CLOB
COLLECTION	OCI_TYPECODE_NAMEDCOLLECTION	SQLT_NCO
DATE	OCI_TYPECODE_DATE	SQLT_DAT
TIMESTAMP	OCI_TYPECODE_TIMESTAMP	SQLT_TIMESTAMP
TIMESTAMP WITH TIME ZONE	OCI_TYPECODE_TIMESTAMP_TZ	SQLT_TIMESTAMP_TZ
TIMESTAMP WITH LOCAL TIME ZONE	OCI_TYPECODE_TIMESTAMP_LTZ	SQLT_TIMESTAMP_LTZ
INTERVAL YEAR TO MONTH	OCI_TYPECODE_INTERVAL_YM	SQLT_INTERVAL_YM

Oracle Type System Typename	Oracle Type System Type	Equivalent SQLT Type
INTERVAL DAY TO SECOND	OCI_TYPECODE_INTERVAL_DS	SQLT_INTERVAL_DS
FLOAT	OCI_TYPECODE_FLOAT (b)	SQLT_FLT (8)[Foot 25]
DECIMAL	OCI_TYPECODE_DECIMAL (p)	SQLT_NUM (p, 0)[Foot 26]
DOUBLE	OCI_TYPECODE_DOUBLE	SQLT_FLT (8)
BINARY_FLOAT	OCI_TYPECODE_BFLOAT	SQLT_BFLOAT
BINARY_DOUBLE	OCI_TYPECODE_BDOUBLE	SQLT_BDOUBLE
INDEX-BY BINARY_INTEGER[Foot 22]	OCI_TYPECODE_ITABLE	SQLT_NTY
INTEGER	OCI_TYPECODE_INTEGER	SQLT_INT (i)[Foot 27]
NUMBER	OCI_TYPECODE_NUMBER (p, s)	SQLT_NUM (p, s)[Foot 28]
OCTET	OCI_TYPECODE_OCTET	SQLT_INT (1)
PLS_INTEGER or BINARY_INTEGER[Foot 22]	OCI_TYPECODE_PLS_INTEGER	SQLT_INT
POINTER	OCI_TYPECODE_PTR	<NONE>
RAW	OCI_TYPECODE_RAW	SQLT_LVB
REAL	OCI_TYPECODE_REAL	SQLT_FLT (4)
REF	OCI_TYPECODE_REF	SQLT_REF
RECORD[Foot 22]	OCI_TYPECODE_RECORD	SQLT_NTY
OBJECT or SYS.XMLType	OCI_TYPECODE_OBJECT	SQLT_NTY
SIGNED(8)	OCI_TYPECODE_SIGNED8	SQLT_INT (1)
SIGNED(16)	OCI_TYPECODE_SIGNED16	SQLT_INT (2)

Oracle Type System Typename	Oracle Type System Type	Equivalent SQLT Type
SIGNED(32)	OCI_TYPECODE_SIGNED32	SQLT_INT (4)
SMALLINT	OCI_TYPECODE_SMALLINT	SQLT_INT (i)[Foot 27]
TABLE[Foot 29]	OCI_TYPECODE_TABLE	<NONE>
UNSIGNED(8)	OCI_TYPECODE_UNSIGNED8	SQLT_UIN (1)
UNSIGNED(16)	OCI_TYPECODE_UNSIGNED16	SQLT_UIN (2)
UNSIGNED(32)	OCI_TYPECODE_UNSIGNED32	SQLT_UIN (4)
VARRAY[Foot 29]	OCI_TYPECODE_VARRAY	<NONE>
VARCHAR	OCI_TYPECODE_VARCHAR (n)	SQLT_CHR (n)[Foot 24]
VARCHAR2	OCI_TYPECODE_VARCHAR2 (n)	SQLT_VCS (n)[Foot 24]

Footnote 23

This type is a PL/SQL type only.

Footnote 24

n is the size of the string in bytes.

Footnote 25

These are floating-point numbers, the precision is given in terms of binary digits. b is the precision of the number in binary digits.

Footnote 26

This is equivalent to a NUMBER with no decimal places.

Footnote 27

i is the size of the number in bytes, set as part of an OCI call.

Footnote 28

p is the precision of the number in decimal digits; s is the scale of the number in decimal digits.

Footnote 29

Can only be part of a named collection type.

4.6 Definitions in oratypes.h

Describes the contents of the `oratypes.h` header file.

Throughout this guide there are references to data types like `ub2` or `sb4`, or to constants like `UB4MAXVAL`. These types are defined in the `oratypes.h` header file, which is found in the `public` directory. The exact contents may vary according to the operating system that you are using.

Note: The use of the data types in `oratypes.h` is the only supported means of supplying parameters to OCI.