Table of Contents

Overview

This is the user documentation for directly accessing the YottaDB engine without the need to go through a shim implemented in its embedded scripting language, M. A process can both call the engine directly as well as call functions written in M and exported.

Caveat: This code does not exist yet. The user documentation is being written ahead of the code, and will change in the event the code needs to differ from this documentation.

Using libyottadb

1. Install YottaDB.
2. Include the `yottadb.h` file in your C program and compile it.
3. Perform any database configuration and initialization needed (configuring global directories, creating database files, starting a Source Server process, etc.).
4. Run your program, ensuring either that `libyottadb.so` is in the load path of your program, or that it is preloaded.

Concepts

Key-value

Local and global variables

Aliases

Data Types

Data types are defined by including `yottadb.h` and are one of:

- User Defined Types, which in turn are one of:
  - Integer
  - Floating Point
  - Other
- Enumerated Types

User Defined Types

Byte

`ydb_zchar_t` — An unsigned data value that is *exactly* 8-bits (one byte).

Integer

`ydb_int_t` and `ydb_uint_t` — Signed and unsigned integers, that are *at least* 16 bits.

`ydb_long_t` and `ydb_ulong_t` — Signed and unsigned integers, that are *at least* 32 bits.

`ydb_longlong_t` and `ydb_ulonglong_t` - Signed and unsigned integers that are *at least* 64 bits. See Numeric Considerations below.

Floating Point

`ydb_float_t` — A floating point number that is *at least* 32 bits in the representation of the underlying computing platform. See Numeric Considerations below.

`ydb_double_t` — A floating point number that is *at least* 64 bits in the representation of the underlying computing platform. See Numeric Considerations below.

Other Scalars

`ydb_numeric_t` — A numeric quantity in YottaDB's internal representation used to get values known to be numeric from YottaDB in order to pass them back to other functions without processing by the caller. Except when a caller needs to manipulate a numeric value returned by YottaDB, passing parameters as `ydb_numeric_t` types is the most efficient way to pass numeric quantities between YottaDB and C.

`ydb_status_t` — Return value (status) of a call to a libyottadb function.

`ydb_token_t` — The type of a token that represents a value stored within YottaDB. Functions such as `ydb_get()` or `ydb_subscript_*()` used to get values — either numeric or strings — to be passed to other functions without processing by the caller can be directed to return token values of type `ydb_token_t`. Depending on the circumstances, using tokens may save CPU cycles on type conversion. See Tokens below. *Consider whether to omit tokens on initial implementation.*

`ydb_tpfnptr_t` — A pointer to a function with a single `void *` parameter passed by value, and a single `ydb_status__t` parameter passed by reference. see Transaction Processing below.

Ennumerated Types

`ydb_type_t` — Defines the type of value in a `ydb_value_t` structure. Values of a `ydb_type_t` are:

- `YDB_CONSTSTRING_STAR` — pointer to a literal string constant
- `YDB_DOUBLE_STAR` — pointer to a `ydb_double_t` value
- `YDB_DOUBLE_VAL` — value of type `ydb_double_t`
- `YDB_EMPTY` — the `ydb_value_t` structure does not contain a value
- `YDB_FLOAT_STAR` — pointer to a `ydb_float_t` value
- `YDB_FLOAT_VAL` — value of type `ydb_float_t`
- `YDB_INT_STAR` — pointer to a `ydb_int_t` value
- `YDB_INT_VAL` — value of type `ydb_int_t`
- `YDB_LONG_STAR` — pointer to a `ydb_long_t` value
- `YDB_LONG_VAL` — value of type `ydb_long_t`
- `YDB_LONGLONG_STAR` — pointer to a `ydb_longlong_t` type
- `YDB_LONGLONG_VAL` — value of type `ydb_long_t`
- `YDB_NUMERIC_STAR` — pointer to a `ydb_numeric_t` type
- `YDB_NUMERIC_VAL` — value of type `ydb_numeric_t`
- `YDB_STRING_STAR` — pointer to a structure of type `ydb_string_t`
- `YDB_TOKEN_VAL` — value of type `ydb_token_t`
- `YDB_UINT_STAR` — pointer to a `ydb_uint_t` type
- `YDB_UINT_VAL` — value of type `ydb_uint_t`
- `YDB_ULONG_STAR` — pointer to a `ydb_ulong_t` value
- `YDB_ULONG_VAL` — value of type `ydb_ulong_t`

- YDB_ZCHAR_STAR - pointer to a ydb_zchar_t value
- YDB_ZCHAR_VAL — value of type ydb_zchar_t

Symbolic Constants

The yottadb.h file defines several symbolic constants, which are one of the following types:

- Function Return Codes, which in turn are one of:
    - Normal Return Codes
    - Error Return Codes
- Limits
- Other

Function Return Codes

Return codes from calls to libyottadb are of type ydb_status_t.

Normal Return Codes

Symbolic constants for normal return codes are prefixed with YDB_.

YDB_STATUS_OK — Normal return following successful execution.

YDB_VALUE_EQU — A call to a ydb_*_compare() function reports that the arguments are equal.

YDB_VALUE_GT — A call to a ydb_*_compare() function reports that the first argument is greater than the second (for numeric comparisons) or lexically follows the second (for string comparisons).

YDB_VALUE_LT — A call to a ydb_*_compare() function reports that the first argument is less than the second (for numeric comparisons) or lexically precedes the second (for string comparisons).

Error Return Codes

Symbolic constants for error codes returned by calls to libyottadb are prefixed with YDB_ERR_.

YDB_ERR_GVUNDEF — No value exists at a requested global variable node.

YDB_ERR_INVMSGNNUM — A call to ydb_zmessage() specified an invalid message code.

YDB_ERR_INVSTRLEN — A buffer provided by the caller is not long enough for the string to be returned.

YDB_ERR_INVSUBS — The number of entries in a ydb_varsub_t structure provided by the caller is insufficient for the actual number of subscripts to be returned.

YDB_ERR_INVSVN — A call referenced a non-existent intrinsic special variable.

YDB_ERR_INVTOKEN — Either a call parameter specifies that the value is a token, but the token is invalid, or libyottadb expects a token, but the tag field is not YDB_INTERNAL.

YDB_ERR_LVUNDEF — No value exists at a requested local variable node.

Limits

Symbolic constants for limits are prefixed with YDB_MAX_. Unless otherwise noted, symbolic constants are unsigned integers guaranteed to fit within the range of a ydb_uint_t type.

YDB_MAX_IDENT —The maximum space in bytes required to store a complete variable name, including the preceding caret for a global variable.

YDB_MAX_MSG — The maximum length in bytes of any message string associated with a message code. A buffer of length YDB_MAX_MSG bytes ensures that a call to ydb_zmessage() will not return a YDB_ERR_INVSTRLEN return code.

YDB_MAX_STR — The maximum length of a string (or blob) in bytes. A caller to ydb_get() that provides a buffer of YDB_MAX_STR will never get a YDB_ERR_INVSTRLEN error. YDB_MAX_STR is guaranteed to fit in a ydb_ulong_t type.

YDB_MAX_SUB — The maximum number of subscripts (keys) for a local or global variable. An array of YDB_MAX_SUB elements always suffices to pass subscripts.

YDB_MAX_VAR — The maximum space in bytes required to store a complete subscripted variable [1] (including caret and subscripts, but not including any preceding global directory name for a global variable reference).

Other

YDB_UNTIMED is a negative integer of type ydb_long_t to be provided by a caller as the timeout parameter for the functions ydb_lock() and ydb_lock_incr().

Data Structures

ydb_string_t is a descriptor for a string [2] value, and consists of the following fields:

- alloc and used — fields of type ydb_strlen_t where alloc ≥ used
- address — pointer to a ydb_zchar_t, the starting address of a string

ydb_value_t — used to transfer data between libyottadb and callers. As libyottadb freely accepts both numbers and strings, automatically convering as needed (see Canonical Numbers below), whereas C is statically typed, the ydb_value_t is a structure that contains a tag describing the data, and a container for the data which is a union of the supported types. ydb_value_t consists of:

- tag — a field of type ydb_type_t
- a union of fields with the following names:
  - double_star — pointer to a ydb_double_t value
  - double_val — value of type ydb_double_t
  - float_star — pointer to a ydb_float_t value
  - float_val — value of type ydb_float_t
  - int_star — pointer to a ydb_int_t value
  - int_val — value of type ydb_int_t
  - long_star — pointer to a ydb_long_t value
  - long_val — value of type ydb_long_t
  - longlong_star — pointer to a ydb_longlong_t type

  - `longlong_val` — value of type `ydb_long_t`
  - `numeric_star` — pointer to a `ydb_numeric_t` type
  - `numeric_val` — value of type `ydb_numeric_t`
  - `string_star` — pointer to a structure of type `ydb_string_t`
  - `uint_star` — pointer to a `ydb_uint_t` type
  - `uint_val` — value of type `ydb_uint_t`
  - `ulong_star` — pointer to a `ydb_ulong_t` value
  - `ulong_val` — value of type `ydb_ulong_t`
  - `zchar_star` — pointer to a `ydb_zchar_t` value
  - `zchar_val` — value of type `ydb_zchar_t`

`ydb_var_t` — used to specify names (i.e., without subscripts). It consists of two fields:

- `name` — a pointer to a `ydb_string_t` structure whose `alloc` ≥ `YDB_MAX_IDENT`
- `accel` — a field that is opaque to the caller, but which libyottadb may use to optimize variable name processing. When a caller initializes a `ydb_var_t` structure, or changes the `varname` field to point to a different variable name, the caller **must** directly or indirectly invoke the `YDB_RESET_ACCEL()` macro. A caller **must not** modify or otherwise use the `accel` field except to reset it.

`ydb_varsub_t` — used to transfer complete variable names between caller and libyottadb, and consists of the four fields:

- `varname` — a `ydb_var_t` structure
- `varsub_alloc` and `varsub_used` —`ydb_uint_t` values with a range of 0 through `YDB_MAX_SUB` that specify the number of subscripts for which space has been allocated and used in the `varsubs` array
- `varsubs` — an array of `ydb_value_t` structures, each providing the value of a subscript

We recommend that applications use the `YDB_VARSUB_ALLOC(num_subs)` and `YDB_VARSUB_RELEASE()` macros to allocate `ydb_varsub_t` structures.

Macros

`YDB_RESET_ACCEL(x)` — Reset (initializes) the `accel` field of a `ydb_var_t` structure.

`YDB_SET_STRING(x, strlit)` — Allocate a `ydb_string_t` structure and initialize it to `strlit`. Note that while the `used` field is the size of `strlit`, the `alloc` field may be rounded up to a larger value.

`YDB_SET_VARNAME_LIT(x, strlit)` and `YDB_SET_VARNAME(x, varname)` — Where x is a pointer to a `ydb_var_t` structure initialize that structure, with a literal string in the first case, and where `varname` is a pointer to a `ydb_string_t` structure, to the string in that structure. They also reset the `accel` field, removing the need to call `YDB_RESET_ACCEL()`.

`YDB_VARSUB_ALLOC(num_subs)` — Allocate a `ydb_varsubs_t` structure with space for `num_subs` subscripts, and initialize the `varsub_alloc` field to `num_subs` and the `varsub_used` field to zero.

`YDB_VARSUB_FREE(x)` — Free (release back to unused memory) the `ydb_varsub_t` structure pointed to by x.

API

API functions are classified as one of:

- Query — Functions that do not update any intrinsic, local or global variables. [3]
- Update — Functions that update intrinsic, local or global variables.
- Transaction Processing — Functions that implement support for ACID transactions.

Query

`ydb_status_t ydb_alias_handle( ydb_string_t *value, ydb_varsub_t *lvn )`

In the location pointed to by `value->address` returns the handle of the local variable referenced by lvsub. It is not meaningful for a caller to perform any operations on handles except to compare two handles for equality.

`ydb_status_t ydb_exists( ydb_uint_t *value, ydb_varsub_t *glvn )`

In the location pointed to by `value`, returns the following information about the local or global variable node identified by glvn:

- 0 — There is neither a value nor a sub-tree, i.e., it is undefined.
- 1 — There is a value, but no sub-tree
- 10 — There is no value, but there is a sub-tree.
- 11 — There are both a value and a subtree.

The following values are only meaningful if glvn identifies a local variable node:

- 100 - The node is an alias, but there is neither a value nor a sub-tree.
- 101 - The node is an alias with a value but sub-tree.
- 110 - The node is an alias with no value, but with a sub-tree.
- 111 - The node is an alias with a value and a sub-tree.

Update

Transaction Processing

Programming Notes

Numeric Considerations

The YottaDB engine internally automatically converts values between numbers and strings as needed. Thus it is legitimate to lexically compare the numbers 2 and 11, with the expected result that 11 precedes 2, and it is equally legitimate to numerically compare the strings “2” and “11”, with the expected result that 11 is greater than 2. The functions for numeric and lexical comparisons are different. A subscript (key) of a variable can include numbers as well as non-numeric strings, with all numeric subscripts preceding all non-numeric strings when stepping through the subscripts in order.

To ensure the accuracy of financial calculations, YottaDB internally stores nnumbers as, and performs arithmetic using, a scaled packed decimal representation with 18 signicant decimal digits, with optimizations for values within a certain subset of its full range. Consequently:

- Any number that is exactly represented in YottaDB can be exactly represented as a string, with reasonably efficient conversion back and forth.
- Any integer value of up to 18 significant digits can be exactly represented by an integer type such as `ydb_longlong-t`, and integers in the inclusive range ±999,999 are handled more efficiently than larger integers.
- In YottaDB there are numbers which can be exactly represented (such as 0.1), but whcih cannot be exactly represented in binary floating point.
- In 64 bit integers and binary floating point formats, there are numbers which can be exactly represented, but which cannot be exactly represented in YottaDB.

This means that for numeric keys which are not guaranteed to be integers:

- In theory, there are edge cases where a value (which would internally be in YottaDB format) returned by a function such as `ydb_subscript_next()` and converted to a `ydb_double_t` when passed back to C application code, and then converted back to YottaDB internal format in a call to `ydb_get()` can result in the node not being found because the double conversion produces a number not identical to the original. Furthermore, there is a cost to the conversion.
- Passing keys back and forth as strings avoids those edge cases, but of course still has a conversion cost.

To preserve accuracy of numeric values that are returned by libyottadb, and which an application code intends to simply pass back to libyottadb as a libyottadb provides a `ydb_numeric_t` type. A value obtained from libyottadb in `ydb_numeric_t` loses no precision when returned to libyottadb, and as a further benefit is very efficient. While the actual value of `ydb_numeric_t` is opaque to application cod, the `ydb_convert()` function is available.

Conversely, when passed a string that is a canonical number for use as a key, libyottadb automatically converts it to a number. This automatic internal conversion is irrelevant for the majority of typical application that:

- simply store and retrieve data associated with keys, potentially testing for the existence of nodes; or
- transfer keys which are numeric values between application code and libyottadb using numeric types and expect numeric ordering.

However, this automatic internal conversion does affect applications that:

- use numeric keys and expect the keys to be sorted in lexical order rather than numeric order; or
- transfer keys which are numeric values between application code and libyottadb as strings that may or may not be canonical numbers.

Applications that are affected by automatic internal conversion should prefix their keys with a character such as “x” which ensures that keys are not canonical numbers.

Canonical Numbers

Conceptually, a canonical number is a string that represents a decimal number in a standard, concise, form.

1. Any string of decimal digits, optionally preceded by a minus sign ("-"), the first of which is not "0" (except for the number zero itself), that represents an integer of no more than 18 significant digits.
    - The following are canonical numbers: "-1", "0", "3", "10", "999999999999999999", "9999999999999999990". Note that the last string has only 18 significant digits even though it is 19 characters long.
    - The following are not canonical numbers: "+1" (starts with "+"), "00" (has an extra leading zero), "9999999999999999999" (19 significant digits).
2. Any string of decimal digits, optionally preceded by a minus sign that includes one decimal point ("."), the first and last of which are not "0", that represents a number of no more than 18 significant digits.
    - The following are canonical numbers: "-.1", ".3", ".999999999999999999".
    - The following are not canonical numbers "+.1" (starts with "+"), "0.3" (first digit is "0"), ".9999999999999999990" (last digit is "0"), ".9999999999999999999" (more than 18 significant digits).
3. Any of the above two forms followed by "E" followed by a canonical number integer in the range -43 to +47 such that the magnitude of the resulting number is between 1E-43 through.1E47.

Tokens

Since numeric and non-numeric subscripts can be freely intermixed in YottaDB, it requires knowledge of the application schema to know whether an application mixes numeric and string subscripts at the same level for a variable.

Consider whether this can be deferred for an initial implementation.

---

1 In M source code, as might be appropriate for an indirect reference.

2 Strings in YottaDB are arbitrary sequences of bytes that are not null-terminated. Other languages may refer to them as binary data or blobs.

3 Any call to libyottadb is permitted to update the `accel` field of a `ydb_var_t` structure passed in to it.