Package 'rbedrock'

September 12, 2021

Title Analysis and Manipulation of Data from Minecraft Bedrock Edition

Version 0.1.1

Description Implements an interface to Minecraft (Bedrock Edition) worlds. Supports the analysis and management of these worlds and game saves.

License MIT + file LICENSE

Encoding UTF-8

Language en-US

RoxygenNote 7.1.1

SystemRequirements C++11, cmake, zlib, GNU make, Solaris: g++ is a requirement

NeedsCompilation yes

Depends R (>= 3.6.0)

Imports tibble, R6, stringr, bit64, rappdirs, rlang, dplyr, purrr, magrittr, readr, utils, digest, vctrs, tidyr, fs

Suggests zip, testthat, jsonlite,

URL https://github.com/reedacartwright/rbedrock

BugReports https://github.com/reedacartwright/rbedrock/issues

Author Reed Cartwright [aut, cre] (<https://orcid.org/0000-0002-0837-9380>),
Rich FitzJohn [ctb],
Christian Stigen Larsen [ctb],
The LevelDB Authors [cph]

Maintainer Reed Cartwright <racartwright@gmail.com>

Repository CRAN

Date/Publication 2021-09-11 23:40:02 UTC

R topics documented:

`bedrockdb` *Open a Bedrock Edition world for reading and writing.*

Description

`bedrockdb` opens a handle to a leveldb database that contains save-game data for a Bedrock Edition world. On success, it returns an R6 class of type 'bedrockdb' that can be used directly for low-level reading and writing access to the db or can be passed to higher-level functions. The handle to the database can be closed by passing it to `close`.

Usage

```
bedrockdb(
  path,
  create_if_missing = FALSE,
  error_if_exists = NULL,
  paranoid_checks = NULL,
  write_buffer_size = 4194304L,
  max_open_files = NULL,
  block_size = NULL,
  cache_capacity = 41943040L,
```

```
  bloom_filter_bits_per_key = 10L,
  compression_level = -1L
)

## S3 method for class 'bedrockdb'
close(con, compact = TRUE, ...)
```

Arguments

path	The path to a world folder. If the path does not exist, it is assumed to be the base name of a world folder in the local minecraftWorlds directory.
create_if_missing	Create world database if it doesn't exist.
error_if_exists	Raise an error if the world database already exists.
paranoid_checks	Internal leveldb option
write_buffer_size	Internal leveldb option
max_open_files	Internal leveldb option
block_size	Internal leveldb option
cache_capacity	Internal leveldb option
bloom_filter_bits_per_key	Internal leveldb option
compression_level	Internal leveldb option
con	An database object created by bedrockdb.
compact	Compact database before closing.
...	arguments passed to or from other methods.

Value

On success, bedrockdb returns an R6 class of type 'bedrockdb'.

Examples

```
# open an example works and get all keys
dbpath <- rbedrock_example_world("example1.mcworld")
db <- bedrockdb(dbpath)
keys <- get_keys(db)
close(db)

## Not run:

# open a world in the minecraftWorlds folder using a world id.
db <- bedrockdb("lrkkYFpUABA=")
# do something with db ...
```

```
close(db)

# open a world using absolute path
db <- bedrockdb("C:\\\\minecraftWorlds\\\\my_world")
# do something with db ...
close(db)

## End(Not run)
```

`bedrock_random` *Random Number Generation for Minecraft*

Description

Bedrock Edition's central random number algorithm is MT19937. However, R's MT19937 code is not compatible with Bedrock's. These routines provide an API that is compatible with Bedrock's.

`bedrock_random_seed()` seeds the random number generator.

`bedrock_random_state()` returns the current state of the random number generator as a raw vector.

`bedrock_random_get_uint()` returns a 32-bit random integer. Default range is [0, 2^32-1].

`bedrock_random_get_int()` returns a 31-bit random integer. Default range is [0, 2^31-1].

`bedrock_random_get_float()` returns a random real number. Default range is [0.0, 1.0).

`bedrock_random_get_double()` returns a random real number Default range is [0.0, 1.0).

Usage

```
bedrock_random_seed(value)

bedrock_random_state(new_state = NULL)

bedrock_random_get_uint(n, max)

bedrock_random_get_int(n, min, max)

bedrock_random_get_float(n, min, max)

bedrock_random_get_double(n)
```

Arguments

`value`	a scalar integer
`new_state`	a raw vector
`n`	number of observations.
`min, max`	lower and upper limits of the distribution. Must be finite. If only one is specified, it is taken as `max`. If neither is specified, the default range is used.

Examples

```
# seed the global random number generator
bedrock_random_seed(5490L)

# save and restore rng state
saved_state <- bedrock_random_state()
bedrock_random_get_uint(10)
bedrock_random_state(saved_state)
bedrock_random_get_uint(10)
```

`bedrock_random_create_seed`

Random Number Seeds for Minecraft

Description

`bedrock_random_create_seed()` constructs a seed using the formulas type 1: `x*a ^ z*b ^ salt`, type 2: `x*a + z*b + salt`, and type 3: `x*a + z*b ^ salt`.

Usage

```
bedrock_random_create_seed(x, z, a, b, salt, type)
```

Arguments

`x, z`	chunk coordinates
`a, b`	seed parameters
`salt`	seed parameter
`type`	which seed type to use

Details

Minecraft uses several different kind of seeds during world generation and gameplay.

Examples

```
# identify slime chunks
g <- tidyr::expand_grid(x=1:10, z=1:10)
is_slime_chunk <- purrr::pmap_lgl(g, function(x,z) {
  seed <- bedrock_random_create_seed(x,z,0x1f1f1f1f,1,0,type=1)
  bedrock_random_seed(seed)
  bedrock_random_get_uint(1,10) == 0
})
```

Biomes *Read and write biome data.*

Description

Biome data is stored as the second map in the 2DMaps data (tag 45). Each chunk stores its biome data as 256 uint8s.

`get_biomes_data()` loads biomes data from a `bedrockdb`. It will silently drop and keys not representing 2DMaps data.

`get_biomes_value()` loads biome data from a `bedrockdb`. It only supports loading a single value.

`put_biomes_data()` `put_biomes_values()`, and `put_biomes_value()` update the biome information of chunks. They preserve any existing height data.

Usage

```
get_biomes_data(db, x, z, dimension, return_names = TRUE)

get_biomes_value(db, x, z, dimension, return_names = TRUE)

put_biomes_data(db, data, missing_height = 0L)

put_biomes_values(db, x, z, dimension, values, missing_height = 0L)

put_biomes_value(db, x, z, dimension, value, missing_height = 0L)
```

Arguments

`db`	A bedrockdb object.
`x, z, dimension`	Chunk coordinates to extract data from. `x` can also be a character vector of db keys.
`return_names`	return biome names instead of biome ids.
`data`	A list of character or integer vectors. Each element of the list must contain 256 values or an error will be raised.
`missing_height`	if there is no existing height data, use this value for the chunk.
`values`	a list of arrays containing biome names or ids.
`value`	an array containing biome names or ids.

Value

`get_biomes_data()` returns a list of the of the values returned by `get_biome_value()`.

An array containing biome information, with dimension "x" and "z". The indexes of the array are a horizontal position relative to the chunk origin.

BlockEntities *Load and store BlockEntities NBT data*

Description

BlockEntities data (tag 49) holds a list of NBT values for entity data associated with specific blocks.

`get_block_entities_data()` loads BlockEntities data from a `bedrockdb`. It will silently drop and keys not representing BlockEntities data.

`put_block_entities_data()` stores BlockEntities data into a `bedrockdb`.

Usage

```
get_block_entities_data(db, x = get_keys(db), z, dimension)

put_block_entities_data(db, data)
```

Arguments

`db`	A bedrockdb object.
`x, z, dimension`	Chunk coordinates to extract data from. `x` can also be a character vector of db keys.
`data`	A named-list of key-value pairs for BlockEntities data.

Checksums *Load and store Checksums data*

Description

Checksums data (tag 59) holds checksums for several chunk records. These records are 2DMaps (tag 45), SubchunkBlocks (tag 47), BlockEntities (tag 49), and Entities (tag 50).

`get_checksums_data()` loads Checksums data from a `bedrockdb`. It will silently drop and keys not representing Checksums data.

`get_checksums_value()` loads Checksums data from a `bedrockdb`. It only supports loading a single value.

`update_checksums_data()` recalculates Checksums data. It calculates checksums for the specified chunks' SubchunkBlocks, 2DMaps, BlockEntities, and Entities records in `db` and updates the Checksums record to match.

`read_checksums_value()` parses a binary Checksums record into a list of checksums.

`write_checksums_value()` converts Checksums from a named list into binary format.

Usage

```
get_checksums_data(db, x = get_keys(db), z, dimension)

get_checksums_value(db, x, z, dimension)

update_checksums_data(db, x, z, dimension)

read_checksums_value(rawdata)

write_checksums_value(object)
```

Arguments

db	A bedrockdb object.
x, z, dimension	Chunk coordinates to extract data from. x can also be a character vector of db keys.
rawdata	a raw vector holding binary Checksums data
object	a named character vector in the same format as returned by read_checksums_value().

Value

get_checksums_data() returns a named-list of the values returned by get_checksums_value().

get_checksums_value() and read_checksums_value() return a character vector. The names of the character vector indicate which chunk record (tag and subtag) the checksum is for.

write_checksums_value() returns a raw vector.

ChunkVersion *Read and write chunk version data*

Description

ChunkVersion data (tag 44) and ChunkVersionLegacy data (tag 118) store the version number of a chunk. In Minecraft version 1.16.100, chunk version data was moved from tag 118 to tag 44.

get_chunk_version_data() retrieves chunk versions from a bedrockdb. It will silently drop and keys not representing ChunkVersion data.

put_chunk_version_data(), put_chunk_version_values(), and put_chunk_version_value() store Finalization data into a bedrockdb. put_chunk_version_data() supports writing both ChunkVersion and ChunkVersionLegacy tags. put_chunk_version_values() and put_chunk_version_value() only support ChunkVersion tags.

read_chunk_version_value() decodes ChunkVersion data.

write_chunk_version_value() encodes ChunkVersion data.

Usage

```
get_chunk_version_data(db, x, z, dimension, include_legacy = TRUE)

get_chunk_version_value(db, x, z, dimension, include_legacy = TRUE)

put_chunk_version_data(db, data)

put_chunk_version_values(db, x, z, dimension, values)

put_chunk_version_value(db, x, z, dimension, value)

read_chunk_version_value(rawdata)

write_chunk_version_value(num)
```

Arguments

db	A bedrockdb object.
x, z, dimension	Chunk coordinates to extract version data from. x can also be a character vector of db keys.
include_legacy	If true, ChunkVersionLegacy tags will be included.
data	A named-vector of key-value pairs for Finalization data.
values	An integer vector
value	A scalar integer vector
rawdata	A scalar raw.
num	A scalar integer.

---

chunk_keys *Read and manipulate chunk keys*

Description

Chunk keys are keys to chunk data. A chunk key has a format which indicates the chunk it holds data for and the type of data it holds. This format is either @x:z:d:t or @x:z:d:t-s, where x and z indicates the coordinates of the chunk in chunk space, d indicates the dimension of the chunk, and t and s indicate the tag and subtag of the chunk.

parse_chunk_keys() splits chunk keys into their individual elements and returns a table with the results. Keys that do not contain chunk data are silently dropped.

create_chunk_keys() returns a vector of chunk keys formed from its arguments.

chunk_positions() returns a matrix containing the chunk coordinates of keys.

chunk_origins() returns a matrix containing the block coordinate of the NW corner of keys.

chunk_tag_str() and chunk_tag_int() convert between integer and character representations of chunk tags.

Usage

```
parse_chunk_keys(keys)

create_chunk_keys(x, z, dimension, tag, subtag)

chunk_positions(keys)

chunk_origins(keys)

chunk_tag_str(tags)

chunk_tag_int(tags)
```

Arguments

keys	A character vector of database keys.
x	Chunk x-coordinate
z	Chunk z-coordinate
dimension	dimension
tag	The type of chunk data.
subtag	The subchunk the key refers to (Only used for tag 47).
tags	a vector

Examples

```
parse_chunk_keys("@0:0:0:47-1")
create_chunk_keys(0, 0, 0, 47, 1)
```

---

delete_values *Remove values from a bedrockdb.*

Description

Remove values from a bedrockdb.

Usage

```
delete_values(
  db,
  keys,
  report = FALSE,
  readoptions = NULL,
  writeoptions = NULL
)
```

Arguments

db	A bedrockdb object
keys	A character vector of keys.
report	A logical indicating whether to generate a report on deleted keys
readoptions	A bedrock_leveldb_readoptions object
writeoptions	A bedrock_leveldb_writeoptions object

Value

If report == TRUE, a logical vector indicating which keys were deleted.

Entities *Load and store Entities NBT data*

Description

Entities data (tag 50) holds a list of NBT values for mobs and other entities in the game.

get_entities_data() loads Entities data from a bedrockdb. It will silently drop and keys not representing Entities data.

put_entities_data() stores Entities data into a bedrockdb.

Usage

```
get_entities_data(db, x = get_keys(db), z, dimension)

put_entities_data(db, data)
```

Arguments

db	A bedrockdb object.
x, z, dimension	Chunk coordinates to extract data from. x can also be a character vector of db keys.
data	A named-list of key-value pairs for Entities data.

`Finalization` *Load and store Finalization data*

Description

Finalization data (tag 54) holds a number which indicates a chunk's state of generation.

`get_finalization_data()` loads Finalization data from a `bedrockdb`. It will silently drop and keys not representing Finalization data.

`get_finalization_value()` loads Finalization data from a `bedrockdb`. It only supports loading a single value.

`put_finalization_data()`, `put_finalization_values()`, and `put_finalization_value()` store Finalization data into a `bedrockdb`.

`read_finalization_value()` parses a binary Finalization record.

`write_finalization_value()` converts a Finalization value to a raw vector.

Usage

```
get_finalization_data(db, x, z, dimension)

get_finalization_value(db, x, z, dimension)

put_finalization_data(db, data)

put_finalization_values(db, x, z, dimension, values)

put_finalization_value(db, x, z, dimension, value)

read_finalization_value(rawdata)

write_finalization_value(value)
```

Arguments

`db`	A bedrockdb object.
`x, z, dimension`	Chunk coordinates to extract data from. `x` can also be a character vector of db keys.
`data`	A named-vector of key-value pairs for Finalization data.
`values`	An integer vector
`value`	a scalar integer
`rawdata`	a raw vector

Details

Finalization data contains the following information.

Value	Name	Description
0	NeedsInstaticking	Chunk needs to be ticked
1	NeedsPopulation	Chunk needs to be populated with mobs
2	Done	Chunk generation is fully complete

Value

`get_finalization_data()` returns a named integer vector of the values returned by `get_finalization_value()`.

`get_finalization_value()` and `read_finalization_value()` return an integer.

`get_chunk_blocks_data` *Load block data from one or more chunks*

Description

These functions return block data as strings containing the block name and block states. The strings' format is `blockname@state1=value1@state2=value2` etc. Blocks may have 0 or more states.

`get_chunk_blocks_value()` is an alias for `get_chunk_blocks_data()`

`get_chunk_blocks_value()` loads block data from a `bedrockdb`. It only supports loading a single value.

`put_chunk_blocks_data()`, `put_chunk_blocks_values()`, and `put_chunk_blocks_value()` stores block data into a `bedrockdb`.

Usage

```
get_chunk_blocks_data(
  db,
  x,
  z,
  dimension,
  names_only = FALSE,
  extra_block = FALSE
)

get_chunk_blocks_values(
  db,
  x,
  z,
  dimension,
  names_only = FALSE,
  extra_block = FALSE
)

get_chunk_blocks_value(
  db,
```

```
  x,
  z,
  dimension,
  names_only = FALSE,
  extra_block = FALSE
)

put_chunk_blocks_data(db, data)

put_chunk_blocks_values(db, x, z, dimension, values)

put_chunk_blocks_value(db, x, z, dimension, value)
```

Arguments

db	A bedrockdb object.
x, z, dimension	Chunk coordinates to extract data from. x can also be a character vector of db keys.
names_only	A logical scalar. Return only the names of the blocks, ignoring block states.
extra_block	A logical scalar. Append the extra block layer to the output (separated by ";"). This is mostly useful if you have waterlogged blocks. If the extra block is air, it will not be appended.
data	A named list of 16xNx16 character() arrays
values	A list of 16xNx16 character() arrays
value	A 16xNx16 character array

Value

`get_chunk_blocks_data()` returns a list of the of the values returned by `read_chunk_blocks_value()`.

`get_chunk_blocks_value()` return a 16xNx16 character array. The axes represent the x, y, and z dimensions in that order. The size of the y-axis is based on the highest subchunk in the coordinate. Missing subchunks are considered air.

---

get_keys *Get a list of keys stored in a bedrockdb.*

Description

Get a list of keys stored in a bedrockdb.

Usage

```
get_keys(db, starts_with = NULL, readoptions = NULL)
```

Arguments

db	A bedrockdb object
starts_with	A string specifying chunk prefix or string prefix.
readoptions	A bedrock_leveldb_readoptions object

Value

A vector containing all the keys found in the bedrockdb.

If starts_with is specified, this vector will be filtered for based on the specified prefix.

---

get_nbt_data *Read and Write NBT Data*

Description

get_nbt_data() and get_nbt_value() load nbt-formatted data from db and parses it. get_nbt_values() is a synonym for get_nbt_data().

put_nbt_values, put_nbt_value, and put_nbt_data stores nbt data into db in binary form.

read_nbt reads NBT data from a raw vector.

write_nbt encodes NBT data into a raw vector.

read_nbt_data calls read_nbt on each element of a list.

write_nbt_data calls write_nbt on each element of a list.

Usage

```
get_nbt_data(
  db,
  keys,
  readoptions = NULL,
  max_elements = NULL,
  simplify = TRUE
)

get_nbt_value(
  db,
  key,
  readoptions = NULL,
  max_elements = NULL,
  simplify = TRUE
)

get_nbt_values(
  db,
  keys,
```

```
  readoptions = NULL,
  max_elements = NULL,
  simplify = TRUE
)

put_nbt_values(db, keys, values, writeoptions = NULL)

put_nbt_value(db, key, value, writeoptions = NULL)

put_nbt_data(db, data, writeoptions = NULL)

read_nbt(rawdata, max_elements = NULL, simplify = TRUE)

write_nbt(object)

read_nbt_data(data, max_elements = NULL, simplify = TRUE)

write_nbt_data(data)
```

Arguments

db	A bedrockdb object
keys	A character vector of keys.
readoptions	A bedrock_leveldb_readoptions object
max_elements	Maximum number of elements to parse.
simplify	If TRUE, simplifies a list containing a single unnamed nbtnode.
key	A single key.
values	A list of nbt objects
writeoptions	A bedrock_leveldb_writeoptions object
value	An nbt object.
data	A named-list specifying key-value pairs.
rawdata	A raw vector
object	An nbt object or a list of nbt objects

Details

The Named Binary Tag (NBT) format is used by Minecraft for various data types.

get_values *Read values stored in a bedrockdb.*

Description

`get_values()` and `get_data()` are synonyms.

Usage

```
get_values(db, keys, readoptions = NULL)

get_data(db, keys, readoptions = NULL)

get_value(db, key, readoptions = NULL)

has_values(db, keys, readoptions = NULL)
```

Arguments

`db`	A `bedrockdb` object
`keys`	A character vector of keys.
`readoptions`	A `bedrock_leveldb_readoptions` object
`key`	A single key.

Value

`get_values()` returns a named-list of raw vectors.

`get_value()` returns a raw vector.

`has_values()` returns a logical vector.

HSA *Read and write HardcodedSpawnArea (HSA) data*

Description

HardcodedSpawnArea (HSA) data (tag 57) stores information about any structure spawning locations in a chunk. An HSA is defined by a bounding box that specifies the location of an HSA in a chunk and a tag that specifies the type: 1 = NetherFortress, 2 = SwampHut, 3 = OceanMonument, and 5 = PillagerOutpost.

`get_hsa_data()` loads HardcodedSpawnArea data from a `bedrockdb`. It will silently drop and keys not representing HSA data.

`get_hsa_value()` loads HSA data from a `bedrockdb`. It only supports loading a single value.

`read_hsa_value()` decodes HSA data.

`put_hsa_data()` puts HSA data into a `bedrockdb`. HSA bounding boxes will be split across chunks and

`put_hsa_values()` and `put_hsa_value()` store HSA data into a `bedrockdb`.

`write_hsa_value()` encodes HSA data.

Usage

```
get_hsa_data(db, x = get_keys(db), z, dimension)

get_hsa_value(db, x, z, dimension)

read_hsa_value(rawdata)

put_hsa_data(db, data, merge = TRUE)

put_hsa_values(db, x, z, dimension, values)

put_hsa_value(db, x, z, dimension, value)

write_hsa_value(value)
```

Arguments

`db`	A bedrockdb object.
`x, z, dimension`	Chunk coordinates to extract data from. `x` can also be a character vector of db keys.
`rawdata`	A scalar raw.
`data`	A table containing HSA coordinates.
`merge`	Merge the new HSAs with existing HSAs.
`values`	A list of tables containing HSA coordinates and tags.
`value`	A table containing HSA coordinates

Value

`get_hsa_data()` returns a table in the same format as `get_hsa_value()`.

`get_hsa_value()` and `read_hsa_value()` return a table with columns indicating the coordinates of the HSA bounding box and the location of the HSS at the center of the bounding box. `get_hsa_value()` also records the dimension of the bounding box.

Examples

```
dbpath <- rbedrock_example_world("example1.mcworld")
db <- bedrockdb(dbpath)
# view all HSA in a world
hsa <- get_hsa_data(db)
hsa
# add an HSA to a world
```

```
dat <- data.frame(x1 = 0, x2 = 15, z1 = 0, z2 = 15,
                  y1 = 40, y2 = 60, tag = "SwampHut")
put_hsa_data(db, dat, merge = TRUE)
close(db)
```

`list_biomes` *List Minecraft Bedrock Edition biomes.*

Description

List Minecraft Bedrock Edition biomes.

Usage

```
list_biomes()

biome_id(x)
```

Arguments

`x`	A character vector containing biome name.

`Maps2D` *Read and write 2DMaps data*

Description

2DMaps data (tag 45) stores information about surface heights and biomes in a chunk. 2DMaps data is 768 bytes long and consists of a 256 int16s (heights) followed by 256 uint8s (biomes).

`get_2dmaps_data()` loads 2DMaps data from a `bedrockdb`. It will silently drop and keys not representing 2DMaps data.

`get_2dmaps_value()` loads 2DMaps data from a `bedrockdb`. It only supports loading a single value.

`read_2dmaps_value` decodes binary 2DMaps data.

`put_2dmaps_data()`, `put_2dmaps_values()`, and `put_2dmaps_value()` store 2DMaps data into a `bedrockdb`.

`write_2dmaps_value` encodes 2DMaps data into a raw vector.

Usage

```
get_2dmaps_data(db, x, z, dimension)

get_2dmaps_value(db, x, z, dimension)

read_2dmaps_value(rawdata)

put_2dmaps_data(db, data)

put_2dmaps_values(db, x, z, dimension, height_maps, biome_maps)

put_2dmaps_value(db, x, z, dimension, height_map, biome_map)

write_2dmaps_value(height_map, biome_map)
```

Arguments

db	A bedrockdb object.
x, z, dimension	Chunk coordinates to extract data from. x can also be a character vector of db keys.
rawdata	A raw vector.
data	A named-vector of key-value pairs for 2DMaps data.
height_maps, biome_maps	Lists of height and biome data. Values will be recycled if necessary to match the number of keys to be written to. If biome_maps is missing, height_maps should be in the same format as returned by get_2dmaps_data().
height_map, biome_map	16x16 arrays containing height and biome data. Values will be recycled if necessary. If biome_map is missing, height-map should be a list a list() with both "height_map" and "biome_map" elements.

Value

get_2dmaps_data() returns a list of the of the values returned by get_2dmaps_value().

get_2dmaps_values() returns a list with components "height_map" and "biome_map".

Examples

```
heights <- matrix(63,16,16)
biomes <- matrix(1,16,16)
# Pass heights and biomes as separate parameters
dat <- write_2dmaps_value(heights, biomes)
# Pass them as a list.
obj <- list(height_map = heights, biome_map = biomes)
dat <- write_2dmaps_value(obj)
# Pass them as scalars
dat <- write_2dmaps_value(63, 1)
```

`minecraft_worlds` *Utilities for working with Minecraft world folders.*

Description

`world_dir_path()` returns the path to the `minecraftWorlds` directory. Use `options(rbedrock.worlds_dir_path = "custom/path")` to customize the path as needed.

`list_worlds()` returns a `data.frame()` containing information about Minecraft saved games.

`create_world()` creates a new Minecraft world.

`export_world()` exports a world to an archive file.

Usage

```
worlds_dir_path(default = FALSE)

list_worlds(worlds_dir = worlds_dir_path())

create_world(id = NULL, ..., worlds_dir = worlds_dir_path())

export_world(id, file, worlds_dir = worlds_dir_path(), replace = FALSE)

import_world(file, id = NULL, ..., worlds_dir = worlds_dir_path())

get_world_path(id, worlds_dir = worlds_dir_path())
```

Arguments

`default`	If `TRUE`, return most likely world path on the system.
`worlds_dir`	The path of a `minecraftWorlds` directory.
`id`	The path to a world folder. If the path is not absolute or does not exist, it is assumed to be the base name of a world folder in `worlds_dir`. For `import_world()`, if `id` is `NULL` a unique world id will be generated. How it is generated is controlled by the `rbedrock.rand_world_id` global options. Possible values are "pretty" and "mcpe".
`...`	Arguments to customize `level.dat` settings. Supports dynamic dots via `rlang::list2()`.
`file`	The path to an mcworld file. If exporting, it will be created. If importing, it will be extracted.
`replace`	If `TRUE`, overwrite an existing file if necessary.

Examples

```
## Not run:

create_world(LevelName = "My World", RandomSeed = 10)

## End(Not run)
```

nbt *Create an NBT value*

Description

The Named Binary Tag (NBT) format is used by Minecraft for various data types. An NBT value holds a 'payload' of data and a 'tag' indicating the type of data held.

`nbt()` creates an nbt value. nbt_*() family of functions are wrappers around `nbt()` to create specific tags. `unnbt()` recursively strips NBT metadata from an NBT value.

`payload()` and payload<-() read and write an nbt value's payload.

Usage

```
nbt(x = list(), tag = 0L)

nbt_end()

nbt_byte(x = 0L)

nbt_short(x = 0L)

nbt_int(x = 0L)

nbt_long(x = 0L)

nbt_float(x = 0)

nbt_double(x = 0)

nbt_string(x = "")

nbt_byte_array(x = integer())

nbt_int_array(x = integer())

nbt_long_array(x = bit64::integer64())

nbt_compound(...)

nbt_list(...)

is_nbt(x)

unnbt(x)

payload(object)
```

```
payload(object) <- value
```

Arguments

x	An nbt payload.
tag	The tag of the data.
...	Arguments to collect into an NBT compound or NBT list value. Supports dynamic dots via `rlang::list2()`.
object	An nbt value
value	A new payload

PendingBlockTicks *Load and store PendingBlockTicks NBT data*

Description

PendingBlockTicks data (tag 51) holds a list of NBT values for pending ticks.

`get_pending_block_ticks_data()` loads PendingBlockTicks data from a `bedrockdb`. It will silently drop and keys not representing PendingBlockTicks data.

`put_pending_block_ticks_data()` stores PendingBlockTicks data into a `bedrockdb`.

Usage

```
get_pending_block_ticks_data(db, x = get_keys(db), z, dimension)

put_pending_block_ticks_data(db, data)
```

Arguments

db	A bedrockdb object.
x, z, dimension	Chunk coordinates to extract data from. `x` can also be a character vector of db keys.
data	A named-list of key-value pairs for PendingBlockTicks data.

put_values *Write values to a bedrockdb.*

Description

Write values to a bedrockdb.

Usage

```
put_values(db, keys, values, writeoptions = NULL)

put_value(db, key, value, writeoptions = NULL)

put_data(db, data, writeoptions = NULL)
```

Arguments

db	A bedrockdb object
keys	A character vector of keys.
values	A list of raw values.
writeoptions	A bedrock_leveldb_writeoptions object
key	A key that will be used to store data.
value	A raw vector that contains the information to be written.
data	A named-list of raw values, specifying key-value pairs.

Value

An invisible copy of db.

RandomBlockTicks *Load and store RandomBlockTicks NBT data*

Description

RandomBlockTicks data (tag 59) holds a list of NBT values for random ticks.

get_random_block_ticks_data() loads RandomBlockTicks data from a bedrockdb. It will silently drop and keys not representing RandomBlockTicks data.

put_random_block_ticks_data() stores RandomBlockTicks data into a bedrockdb.

Usage

```
get_random_block_ticks_data(db, x = get_keys(db), z, dimension)

put_random_block_ticks_data(db, data)
```

Arguments

db	A bedrockdb object.
x, z, dimension	Chunk coordinates to extract data from. x can also be a character vector of db keys.
data	A named-list of key-value pairs for RandomBlockTicks data.

---

rbedrock_example *Get path to rbedrock example*

Description

rbedrock comes bundled with a number of sample files in its inst/extdata directory. This function make them easy to access.

Usage

```
rbedrock_example(path = NULL)

rbedrock_example_world(path)
```

Arguments

path	Name of file or directory. If NULL, the examples will be listed.

Examples

```
rbedrock_example()
rbedrock_example("example1.mcworld")
rbedrock_example_world("example1.mcworld")
```

---

read_leveldat *Read and write data from a world's level.dat file.*

Description

Read and write data from a world's level.dat file.

Usage

```
read_leveldat(path, old = FALSE)

write_leveldat(object, path, old = FALSE, version = 8L)
```

Arguments

path	The path to a world folder. If the path does not exist, it is assumed to be the base name of a world folder in the local minecraftWorlds directory.
old	Read/write to 'level.dat_old' instead.
object	NBT data to be written to level.dat.
version	The level.dat format version for the file header.

Value

`read_leveldat` returns an nbtnode object.

`write_leveldat` returns a copy of the data written.

SubchunkBlocks *Load and store SubchunkBlocks data*

Description

SubchunkBlocks data (tag 47) holds information about the blocks in a subchunks. Each chunk is divided into multiple 16x16x16 subchunks, and each subchunk is stored separately and indicated by the use of the subtag. Blocks are stored in a palette-based format. Subchunks can have two layers of blocks, and the extra layer is most-often used to store water for water-logged blocks.

These functions return block data as strings containing the block name and block states. The strings' format is `blockname@state1=value1@state2=value2` etc. Blocks may have 0 or more states.

`get_subchunk_blocks_data()` loads SubchunkBlocks data from a `bedrockdb`. It will silently drop and keys not representing SubchunkBlocks data.

`get_subchunk_blocks_value()` loads SubchunkBlocks data from a `bedrockdb`. It only supports loading a single value.

`put_subchunk_blocks_data()`, `put_subchunk_blocks_values()`, and `put_subchunk_blocks_value()` store SubchunkBlocks data into a `bedrockdb`.

`read_subchunk_blocks_value()` decodes binary SubchunkBlock data.

`subchunk_origins()` returns a matrix containing the block coordinate of the lower NW corner of subchunk keys

`subchunk_coords()` determines the block coordinates of blocks based on their array indexes and their subchunk origins.

Usage

```
get_subchunk_blocks_data(
  db,
  x,
  z,
  dimension,
  subchunk,
```

```
  names_only = FALSE,
  extra_block = FALSE
)

get_subchunk_blocks_value(
  db,
  x,
  z,
  dimension,
  subchunk,
  names_only = FALSE,
  extra_block = FALSE
)

put_subchunk_blocks_data(db, data)

put_subchunk_blocks_values(db, x, z, dimension, subchunk, values)

put_subchunk_blocks_value(db, x, z, dimension, subchunk, value)

read_subchunk_blocks_value(rawdata, names_only = FALSE, extra_block = FALSE)

write_subchunk_blocks_value(object)

subchunk_origins(keys)

subchunk_coords(ind, origins = subchunk_origins(names(ind)))
```

Arguments

db	A bedrockdb object.
x, z, dimension	Chunk coordinates to extract data from. x can also be a character vector of db keys.
subchunk	Subchunk indexes to extract data from.
names_only	A logical scalar. Return only the names of the blocks, ignoring block states.
extra_block	A logical scalar. Append the extra block layer to the output (separated by ";"). This is mostly useful if you have waterlogged blocks. If the extra block is air, it will not be appended.
data	A named list of 16x16x16 character() arrays
values	A list of 16x16x16 character() arrays
value	A 16x16x16 character array
rawdata	a raw vector holding binary SubchunkBlock data
object	A 16x16x16 character array.
keys	A character vector of database keys.
ind	Numeric vector or a named list of numeric vectors containing indexes for blocks in a subchunk.
origins	A matrix of subchunk origins.

Details

If a subchunk contains only air it will not be stored in the database, and missing subchunks are considered air.

Value

`get_subchunk_blocks_data()` returns a list of the of the values returned by `read_subchunk_blocks_value()`.

`get_subchunk_blocks_value()` and `read_subchunk_blocks_value()` return a 16x16x16 character array. The axes represent the `x`, `y`, and `z` dimensions in that order.

`read_subchunk_blocks_value()` returns a 16x16x16 character array. The axes represent the `x`, `y`, and `z` dimensions in that order.

`subchunk_coords()` returns a 3-column matrix of block coordinates.

Index