# Package: pjrt (via r-universe)

June 3, 2026

**Title** R Interface to PJRT

**Version** 0.4.0

**Description** Provides an R interface to PJRT (Pluggable Jit RunTime), which allows you to run XLA or stableHLO programs on a variety of hardware backends including CPU, GPU, and TPU.

**License** MIT + file LICENSE

**URL** https://r-xla.github.io/pjrt/, https://github.com/r-xla/pjrt

**BugReports** https://github.com/r-xla/pjrt/issues

**Imports** bit64, checkmate, cli, fs, Rcpp, rlang, safetensors(>= 0.2.0), withr, tengen (>= 0.2.0)

**Suggests** testthat (>= 3.0.0)

**LinkingTo** Rcpp

**Additional_repositories** https://r-xla.r-universe.dev

**Config/build/compilation-database** true

**Config/testthat/edition** 3

**Encoding** UTF-8

**Roxygen** list(markdown = TRUE)

**RoxygenNote** 7.3.3

**SystemRequirements** C++20, libprotobuf, protobuf-compiler

**Collate** 'RcppExports.R' 'async.R' 'client.R' 'buffer.R' 'compare.R' 'custom_call.R' 'device.R' 'execution_options.R' 'format.R' 'loaded_executable.R' 'package.R' 'pjrt-package.R' 'plugin.R' 'program.R' 'reexports.R' 'safetensors.R' 'utils.R' 'zzz.R'

**Config/pak/sysreqs** cmake make libprotobuf-dev libuv1-dev protobuf-compiler libprotoc-dev

**Repository** https://r-xla.r-universe.dev

**Date/Publication** 2026-06-03 11:19:05 UTC

**RemoteUrl** https://github.com/r-xla/pjrt

**RemoteRef** v0.4.0

**RemoteSha** ca5285ce17b690558ba091ace607a6d67f8e6d92

# Contents



---

`pjrt-package` *pjrt: R Interface to PJRT*

---

## Description

Provides an R interface to PJRT (Pluggable Jit RunTime), which allows you to run XLA or stableHLO programs on a variety of hardware backends including CPU, GPU, and TPU.

## Environment Variables

### Configuration options provided by XLA

XLA provides various configuration options, but their documentation is scattered across various websites. The options include:

- `TF_CPP_MIN_LOG_LEVEL`: Logging level for PJRT C++ API:

  - 0: shows info, warnings and errors
  - 1: shows warnings and errors
  - 2: shows errors
  - 3: shows nothing
- `XLA_FLAGS`: See the openxla website for more information.

**Configuration options provided by this package**

- `PJRT_PLATFORM`: Default platform to use, falls back to `"cpu"`.
- `PJRT_PLUGIN_PATH_<PLATFORM>`: Path to custom plugin library file for a specific platform (e.g., `PJRT_PLUGIN_PATH_CPU`, `PJRT_PLUGIN_PATH_CUDA`, `PJRT_PLUGIN_PATH_METAL`). If set, the package will use this path instead of downloading the plugin.
- `PJRT_PLUGIN_URL_<PLATFORM>`: URL to download plugin from for a specific platform (e.g., `PJRT_PLUGIN_URL_CPU`, `PJRT_PLUGIN_URL_CUDA`, `PJRT_PLUGIN_URL_METAL`). If set, overrides the default plugin download URL.
- `PJRT_ZML_ARTIFACT_VERSION`: Version of ZML artifacts to download. Only used when downloading plugins from zml/pjrt-artifacts.
- `PJRT_CPU_DEVICE_COUNT`: The number of CPU devices to use. Defaults to 1. This is primarily intended for testing purposes.
- `PJRT_CUDA_R_PACKAGE`: Name of the R package providing CUDA libraries. Defaults to the value of cuda12.8. Set this to use a different CUDA toolkit package, but note that other versions may not work with the XLA plugin.
- `PJRT_DEBUG`: If set (to any non-empty value), enables verbose debug output via `cli::cli_inform()`.

## Third-Party Licenses

The `pjrt` package itself is MIT-licensed. The CUDA backend dynamically loads NVIDIA software which is not bundled with `pjrt`, but downloaded from NVIDIA's official redistributable channels by the CUDA toolkit R package (e.g. `cuda12.8`) at install time. Its use is governed by the NVIDIA CUDA Toolkit EULA, with the exception of cuDNN, which is covered by the NVIDIA cuDNN SLA, and NCCL, which is covered by its own license. By installing or using the CUDA backend you accept those terms.

## Author(s)

**Maintainer**: Sebastian Fischer <seb.fischer@tutamail.com> (ORCID)

Authors:

- Daniel Falbel <daniel@posit.co> (ORCID)

## See Also

Useful links:

- https://r-xla.github.io/pjrt/
- https://github.com/r-xla/pjrt
- Report bugs at https://github.com/r-xla/pjrt/issues

## as.character.PJRTElementType *Convert* PJRTElementType *to string*

### Description

Get a (lowercase) string representation of a PJRT element type

### Usage

```
## S3 method for class 'PJRTElementType'
as.character(x, ...)
```

### Arguments

| | |
|---|---|
| x | A PJRT element type object. |
| ... | Additional arguments (unused). |

### Value

A string representation of the element type.

## as_array.PJRTBuffer *Convert a PJRTBuffer to an R Array*

### Description

Transfer buffer data from device to host and return an R array.

### Usage

```
## S3 method for class 'PJRTBuffer'
as_array(x, check = FALSE, ...)
```

### Arguments

| | |
|---|---|
| x | (PJRTBuffer)<br>Buffer to convert. |
| check | (logical(1))<br>If TRUE, sanity-check the materialized R vector against losing information across the device-to-host boundary, and abort if any problematic value is detected:<br>• i32 / i64: any NA in the result. R's NA_integer_ shares the bit pattern INT_MIN; bit64's NA_integer64_ shares INT64_MIN. A legitimate device value at those bit patterns is indistinguishable from NA once materialized in R. |

- ui64: any negative value in the result. ui64 is stored as bit64::integer64 (signed 64-bit), which wraps values >= 2^63 to negative — exactly 2^63 becomes NA_integer64_, anything above becomes a non-NA negative integer64.

No-op for float, boolean, and small/unsigned-32 integer dtypes — ui32 is now stored as integer64 and has full headroom, so it cannot produce a wrapped or NA value.

... Additional arguments (unused).

## Value

An R array (or vector for shape integer()).

---

as_array_async *Convert buffer to R array asynchronously*

---

## Description

Start an asynchronous transfer of buffer data from device to host. Returns immediately with a PJRTArrayPromise object.

Use value() to get the R array (blocks if not ready). Use is_ready() to check if transfer has completed (non-blocking).

## Usage

```
as_array_async(x, ...)
```

## Arguments

| | |
|---|---|
| x | A PJRTBuffer object. |
| ... | Additional arguments (unused). |

## Value

A PJRTArrayPromise object. Call value() to get the R array.

## See Also

as_array(), value(), is_ready(), pjrt_execute(), await()

## Examples

```
buf <- pjrt_buffer(c(1.0, 2.0, 3.0, 4.0), shape = c(2, 2), dtype = "f32")
result <- as_array_async(buf)
is_ready(result)
value(result)
```

## as_pjrt_client *Convert to PJRT Client*

### Description

Convert a platform name to a PJRT client or verify that an object is already a client.

### Usage

```
as_pjrt_client(x)
```

### Arguments

| | |
|---|---|
| x | (PJRTClient \| character(1))<br>Either a PJRT client object or a platform name (e.g., "cpu", "cuda", "metal"). |

### Value

PJRTClient

### Examples

```
# Convert from platform name
client <- as_pjrt_client("cpu")
client
```

## as_pjrt_device *Convert to PJRT Device*

### Description

Convert a platform name or device to a PJRT device object.

### Usage

```
as_pjrt_device(x)
```

### Arguments

| | |
|---|---|
| x | (PJRTDevice \| character(1) \| NULL)<br>Either a PJRT device object, a platform name (e.g., "cpu", "cuda", "metal"), a device specification with index (e.g., "cpu:0", "cuda:1" for 0-based indexing), or NULL (defaults to first CPU device). |

### Value

PJRTDevice

## as_pjrt_plugin *Convert to PJRT Plugin*

### Description

Convert a platform name to a PJRT plugin or verify that an object is already a plugin.

### Usage

```
as_pjrt_plugin(x)
```

### Arguments

| | |
|---|---|
| x | (any)<br>Object to convert to a PJRT plugin. Currently supports `PJRTPlugin` and `character(1)`. |

### Value

`PJRTPlugin`

### Examples

```
# Convert from platform name
plugin <- as_pjrt_plugin("cpu")
plugin
```

## await *Await an async operation*

### Description

Block until the async operation is complete and return the object.

### Usage

```
await(x, ...)
```

### Arguments

| | |
|---|---|
| x | An async value object. |
| ... | Additional arguments (unused). |

### Value

The awaited object (invisibly).

## copy_buffer *Copy Buffer to Device*

### Description

Copy a PJRTBuffer to a different device. Returns a new buffer on the target device; the original buffer is unchanged.

If the buffer already lives in the requested device, no copy is performed.

When the target device belongs to a different client (e.g. copying from CPU to CUDA), the transfer is performed via a host roundtrip.

### Usage

```
copy_buffer(buffer, device)
```

### Arguments

| | |
|---|---|
| buffer | (PJRTBuffer)<br>A PJRT buffer object. |
| device | (PJRTDevice \| character(1))<br>The target device. A PJRTDevice object or a device specification (e.g., "cpu:0", "cpu:1", "cuda:0"). |

### Value

A new PJRTBuffer on the target device.

### Examples

```
buf <- pjrt_buffer(c(1, 2, 3), device = "cpu")
buf2 <- copy_buffer(buf, "cuda")
device(buf2)
```

## devices *Devices*

### Description

Get the addressable devices.

### Usage

```
devices(x = NULL, ...)
```

### Arguments

| | |
|---|---|
| x | An object to get devices from: a PJRTClient, a PJRTLoadedExecutable, or NULL (default client). |
| ... | Additional arguments (currently unused). |

### Value

list of PJRTDevice

### Examples

```
# Create client (defaults to CPU)
client <- pjrt_client()
devices(client)
```

---

## elt_type *Element Type*

---

### Description

Get the element type of a buffer.

### Usage

```
elt_type(x)
```

### Arguments

| | |
|---|---|
| x | (PJRTBuffer)<br>Buffer. |

### Examples

```
buf <- pjrt_buffer(c(1.0, 2.0, 3.0))
elt_type(buf)
```

## format_buffer *Format Buffer Data*

### Description

Formats buffer data into a character vector of string representations of individual elements suitable for stableHLO.

### Usage

```
format_buffer(buffer)
```

### Arguments

| | |
|---|---|
| buffer | (PJRTBuffer)<br>A PJRT buffer object. |

### Value

`character()` A character vector containing the formatted elements.

### Examples

```
buf <- pjrt_buffer(c(1.5, 2.5, 3.5))
format_buffer(buf)
```

## is_ready *Check if an async operation is ready*

### Description

Non-blocking check to see if an async operation has completed.

### Usage

```
is_ready(x, ...)
```

### Arguments

| | |
|---|---|
| x | An async value object. |
| ... | Additional arguments (unused). |

### Value

`TRUE` if the operation has completed, `FALSE` otherwise.

pjrt_buffer *Create a PJRT Buffer*

## Description

Create a PJRT Buffer from an R object. Any numeric PJRT buffer is an array and 0-dimensional arrays are used as scalars. `pjrt_buffer` will create a array with dimensions `(1)` for a vector of length 1, while `pjrt_scalar` will create a 0-dimensional array for an R vector of length 1.

To create an empty buffer (at least one dimension must be 0), use `pjrt_empty`.

**Important**: No checks are performed when creating the buffer, so you need to ensure that the data fits the selected element type (e.g., to prevent buffer overflow) and that no NA values are present.

## Usage

```
pjrt_buffer(
  data,
  dtype = NULL,
  device = NULL,
  shape = NULL,
  check = FALSE,
  ...
)

pjrt_scalar(data, dtype = NULL, device = NULL, check = FALSE, ...)

pjrt_empty(dtype, shape, device = NULL)
```

## Arguments

| | |
|---|---|
| data | (any)<br>Data to convert to a `PJRTBuffer`. |
| dtype | (`NULL` \| `character(1)` \| `DataType`)<br>The type of the buffer. Currently supported types are:<br>• `"pred"`: predicate (i.e. a boolean)<br>• `"{s,u}{8,16,32,64}"`: Signed and unsigned integer (for `integer` data).<br>• `"f{32,64}"`: Floating point (for `double` or `integer` data). The default (`NULL`) depends on the method:<br>• `logical` -> `"pred"`<br>• `integer` -> `"i32"`<br>• `double` -> `"f32"`<br>• `raw` -> must be supplied |
| device | (`NULL` \| `PJRTDevice` \| `character(1)`)<br>A `PJRTDevice` object or the name of the platform to use ("cpu", "cuda", ...), in which case the first device for that platform is used. The default is to use the CPU platform, but this can be configured via the `PJRT_PLATFORM` environment variable. |

shape (NULL | integer())
The dimensions of the buffer. The default (NULL) is to infer them from the data if possible. The default (NULL) depends on the method.

check (logical(1))
If TRUE, scan data for NA values before transferring to the device and raise an error if any are present. R's NA markers have no representation at the XLA level (e.g. NA_integer_ is just the bit pattern -2147483648, and NA of logical type is silently coerced to TRUE), so missing values are silently lost on transfer. Defaults to FALSE for performance; set to TRUE to fail loudly instead of silently corrupting data. Not applicable to raw input.

... (any)
Additional arguments. For raw types, this includes:

- row_major: Whether to read the data in row-major format or column-major format. R uses column-major format.

## Value

PJRTBuffer

## Extractors

- platform() -> character(1): for the platform name of the buffer ("cpu", "cuda", ...).
- device() -> PJRTDevice: for the device of the buffer (also includes device number)
- elt_type() -> PJRTElementType: for the element type of the buffer.
- shape() -> integer(): for the shape of the buffer.

## Converters

- as_array() -> array | vector: for converting back to R (vector is only used for shape integer()).
- as_raw() -> raw for a raw vector.

## Reading and Writing

- safetensors::safe_save_file for writing to a safetensors file.
- safetensors::safe_load_file for reading from a safetensors file.

## Scalars

When calling this function on a vector of length 1, the resulting shape is 1L. To create a 0-dimensional buffer, use pjrt_scalar where the resulting shape is integer().

## Examples

```
# Create a buffer from a numeric vector
buf <- pjrt_buffer(c(1, 2, 3, 4))
buf

# Create a buffer from a matrix
mat <- matrix(1:6, nrow = 2)
buf <- pjrt_buffer(mat)
buf

# Create an integer buffer from an array
arr <- array(1:8, dim = c(2, 2, 2))
buf <- pjrt_buffer(arr)


# Create a scalar (0-dimensional array)
scalar <- pjrt_scalar(42, dtype = "f32")
scalar


# Create an empty buffer
empty <- pjrt_empty(dtype = "f32", shape = c(0, 3))
empty
```

---

# pjrt_client *Create a Client*

---

## Description

Create a PJRT client for a specific device.

## Usage

```
pjrt_client(platform = NULL, ...)
```

## Arguments

| | |
|---|---|
| platform | (character(1) \| NULL)<br>Platform name (e.g., "cpu", "cuda", "metal"). If NULL, use PJRT_PLATFORM environment variable or default to "cpu". |
| ... | Additional options passed to the PJRT client creation. For CPU clients, you can pass cpu_device_count to specify the number of CPU devices. You can also configure this via PJRT_CPU_DEVICE_COUNT environment variable. |

## Value

PJRTClient

### Extractors

- platform() for a character(1) representation of the platform.
- devices() for a list of PJRTDevice objects.

### Examples

```
# Create a client (defaults to CPU)
client <- pjrt_client()
client
```

---

## pjrt_compile *Compile a Program*

---

### Description

Compile a PJRTProgram program into a PJRTExecutable.

### Usage

```
pjrt_compile(program, compile_options = new_compile_options(), device = NULL)
```

### Arguments

| | |
|---|---|
| program | (character(1))<br>A program to compile. |
| compile_options | (PJRTCompileOptions)<br>Compile options. |
| device | (NULL \| PJRTDevice \| character(1))<br>A PJRTDevice object or the name of the platform to use ("cpu", "cuda", ...), in which case the first device for that platform is used. The default is to use the CPU platform, but this can be configured via the PJRT_PLATFORM environment variable. |

### Value

PJRTExecutable

### Examples

```
# Create a simple program
src <- r"(
func.func @main(\%arg0: tensor<2xf32>) -> tensor<2xf32> {
  return \%arg0 : tensor<2xf32>
}
)"
```

```
prog <- pjrt_program(src = src)
exec <- pjrt_compile(prog)
```

---

`pjrt_device` *Create a PJRT Device*

---

## Description

Create a PJRT Device from an R object.

## Usage

```
pjrt_device(device)
```

## Arguments

| | |
|---|---|
| `device` | (any)<br>The device. |

## Value

`PJRTDevice`

## Extractors

- `platform()` for a `character(1)` representation of the platform.

## Examples

```
# Show available devices for CPU client
devices(pjrt_client("cpu"))
# Create device 0 for CPU client
dev <- pjrt_device("cpu:0")
dev
```

`pjrt_execute` *Execute a PJRT program*

## Description

Execute a PJRT program with the given inputs and execution options. Returns immediately with `PJRTBuffer` object(s) that may not be ready yet.

**Important:** Arguments are passed by position and names are ignored.

Inputs can be `PJRTBuffer` objects, including buffers that are not yet ready. PJRT handles the dependencies internally.

Use `await()` to block until the result is ready. Use `is_ready()` to check if execution has completed (non-blocking). Use `as_array_async()` to chain async buffer-to-host transfer.

## Usage

```
pjrt_execute(executable, ..., execution_options = NULL, simplify = TRUE)
```

## Arguments

| | |
|---|---|
| `executable` | (`PJRTLoadedExecutable`)<br>A PJRT program |
| `...` | (`PJRTBuffer`)<br>Inputs to the program. Named are ignored and arguments are passed in order. |
| `execution_options` | (`PJRTExecuteOptions`)<br>Optional execution options for configuring buffer donation and other settings. |
| `simplify` | (`logical(1)`)<br>If `TRUE` (default), a single output is returned as a `PJRTBuffer`. If `FALSE`, a single output is returned as a `list` of length 1 containing a `PJRTBuffer`. |

## Value

`PJRTBuffer` | `list` of `PJRTBuffers`

## See Also

`await()`, `is_ready()`, `as_array_async()`

## Examples

```
# Create and compile a simple identity program
src <- r"(
func.func @main(
  \%x: tensor<2x2xf32>,
  \%y: tensor<2x2xf32>
) -> tensor<2x2xf32> {
```

```
  \%0 = "stablehlo.add"(\%x, \%y) : (tensor<2x2xf32>, tensor<2x2xf32>) -> tensor<2x2xf32>
  "func.return"(\%0): (tensor<2x2xf32>) -> ()
}
)"
prog <- pjrt_program(src = src)
exec <- pjrt_compile(prog)

# Execute with input
x <- pjrt_buffer(c(1.0, 2.0, 3.0, 4.0), shape = c(2, 2), dtype = "f32")
y <- pjrt_buffer(c(5, 6, 7, 8), shape = c(2, 2), dtype = "f32")
pjrt_execute(exec, x, y)
```

---

`pjrt_execution_options` *Create Execution Options*

---

## Description

Create execution options for configuring how a PJRT program is executed, including buffer donation settings. **Important**: It is not enough to only mark a buffer as donatable (not not donatable) during runtime. The program also needs to specify this during compile-time via input-output aliasing (stableHLO attribute `tf.aliasing_output`).

## Usage

```
pjrt_execution_options(non_donatable_input_indices = integer(), launch_id = 0L)
```

## Arguments

`non_donatable_input_indices` (`integer()`)
A vector of input buffer indices that should not be donated during execution (0-based). Buffer donation allows the runtime to reuse input buffers for outputs when possible, which can improve performance. However, if an input buffer is referenced multiple times or needs to be preserved, it should be marked as non-donatable.

`launch_id` (`integer(1)`)
An optional launch identifier for multi-device executions. This can be used to detect scheduling errors in multi-host programs.

## Value

`PJRTExecuteOptions`

## Examples

```
# Create default execution options
opts <- pjrt_execution_options()

# Mark buffer 0 as non-donatable
opts <- pjrt_execution_options(non_donatable_input_indices = 0L)
```

---

`pjrt_plugin` *Create PJRT Plugin*

---

## Description

Create a PJRT plugin for a specific platform.

## Usage

```
pjrt_plugin(platform)
```

## Arguments

| | |
|---|---|
| `platform` | (`character(1)`)<br>Platform name (e.g., "cpu", "cuda", "metal"). |

## Value

`PJRTPlugin`

## Extractors

- `plugin_attributes()` -> `list()`: for the attributes of the plugin.

## Examples

```
plugin <- pjrt_plugin("cpu")
plugin
```

## pjrt_program *Create a* PJRTProgram

### Description

Create a program from a string or file path.

### Usage

```
pjrt_program(src = NULL, path = NULL, format = c("mlir", "hlo"))
```

### Arguments

| | |
|---|---|
| src | (character(1)) Source code. |
| path | (character(1)) Path to the program file. |
| format | (character(1)) One of "mlir" or "hlo". |

### Value

PJRTProgram

### Examples

```
# Create a program from source
src <- "
func.func @main(%arg0: tensor<2xf32>) -> tensor<2xf32> {
  return %arg0 : tensor<2xf32>
}
"
prog <- pjrt_program(src = src)
prog
```

## pjrt_register_custom_call *Register a Custom Call Handler*

### Description

Register an XLA FFI handler for use with `stablehlo.custom_call`.

Handlers are C/C++ functions defined using the XLA FFI API (see `xla/ffi/api/ffi.h` shipped in pjrt's `inst/include/`). They are passed to this function as external pointers.

Registration is deferred: if the PJRT plugin for a given platform is not yet loaded, the handler is queued and registered automatically when `pjrt_plugin()` loads it.

### Usage

```
pjrt_register_custom_call(target_name, handler, .package = NULL)
```

### Arguments

| | |
|---|---|
| target_name | (character(1))<br>The target name used in stablehlo.custom_call @target_name(...). |
| handler | A named list of external pointers (externalptr) to XLA_FFI_Handlers, keyed by PJRT platform name (e.g., list(host = ptr) or list(host = cpu_ptr, cuda = cuda_ptr)). |
| .package | (character(1) or NULL)<br>The package registering this handler. When provided, handlers are automatically removed from the registry when the package unloads. |

### Value

NULL (invisibly).

---

## platform *Platform Name*

---

### Description

Get the platform name of a PJRT buffer.

### Usage

```
platform(x, ...)
```

### Arguments

| | |
|---|---|
| x | (PJRTBuffer)<br>The buffer. |
| ... | Additional arguments (unused). |

### Value

character(1)

### Examples

```
buf <- pjrt_buffer(c(1, 2, 3))
platform(buf)
```

## plugin_attributes *Get Plugin Attributes*

### Description

Get the attributes of a PJRT plugin. This commonly includes:

- `xla_version`
- `stablehlo_current_version`
- `stablehlo_minimum_version`

But the implementation depends on the plugin.

### Usage

```
plugin_attributes(plugin)
```

### Arguments

| | |
|---|---|
| `plugin` | (`PJRTPlugin` \| `character(1)`)<br>The plugin (or platform name) to get the attributes of. |

### Value

named `list()`

### Examples

```
plugin_attributes("cpu")
```

## plugin_client_create *Create PJRT Client*

### Description

Create a PJRT client for a specific plugin and platform.

### Usage

```
plugin_client_create(plugin, platform, options = list())
```

### Arguments

| | |
|---|---|
| plugin | (PJRTPlugin)<br>The plugin to create a client for. |
| platform | (character(1))<br>The platform to create a client for. |
| options | (list())<br>Additional options to pass to the client. |

### Value

PJRTClient

---

## plugins_downloaded *Check if Plugin is Downloaded*

---

### Description

Check if one more more plugin is already downloaded.

### Usage

```
plugins_downloaded(platforms = NULL)
```

### Arguments

| | |
|---|---|
| platforms | (character())<br>Platform names. |

### Value

logical(1)

### Examples

```
# Check if CPU plugin is downloaded
plugins_downloaded("cpu")
```

`print.PJRTBuffer` *Print a PJRT Buffer*

## Description

Print a `PJRTBuffer`.

## Usage

```
## S3 method for class 'PJRTBuffer'
print(
  x,
  max_rows = getOption("pjrt.print_max_rows", 30L),
  max_width = getOption("pjrt.print_max_width", 85L),
  max_rows_slice = getOption("pjrt.print_max_rows_slice", max_rows),
  header = TRUE,
  footer = NULL,
  ...
)
```

## Arguments

| | |
|---|---|
| `x` | (`PJRTBuffer`)<br>The buffer. |
| `max_rows` | (`integer(1)`)<br>The maximum number of rows to print, excluding header and footer. |
| `max_width` | (`integer(1)`)<br>The maximum width (in characters) of the printed buffer. Set to negative values for no limit. Note that for very small values, the actual printed width might be slightly smaller as at least one column will be printed. Also, this limit only affects the printed rows containing the actual data, other rows might exceed the width. |
| `max_rows_slice` | (`integer(1)`)<br>The maximum number of rows to print for each slice. |
| `header` | (`logical(1)`)<br>Whether to print the header. |
| `footer` | (`NULL` or `character(1)`)<br>The footer line to print. If `NULL` (default), prints the standard `[ <PLATFORM><TYPE>{<SHAPE>} ]` summary. Use `""` to suppress. |
| `...` | Additional arguments (unused). |

`value` *Get the value of an async operation*

## Description

Materialize and return the result of an async operation. Blocks until the operation is complete if it hasn't finished yet.

For `PJRTArrayPromise`, returns the materialized R array. For `PJRTBuffer`, use `await()` to block until ready.

## Usage

```
value(x, ...)
```

## Arguments

| | |
|---|---|
| `x` | An async value object. |
| `...` | Additional arguments (unused). |

## Value

The materialized value.

# Index