Mojo C System API

This document is a subset of the Mojo documentation.

Contents

Overview

The Mojo C System API is a lightweight API (with an stable, forward-compatible ABI) upon which all higher-level public Mojo APIs are built.

This API exposes the fundamental capabilities to: create, read from, and write to **message pipes**; create, read from, and write to **data pipes**; create **shared buffers** and generate sharable handles to them; wrap platform-specific handle objects (such as **file descriptors**, **Windows handles**, and **Mach ports**) for seamless transit over message pipes; and efficiently watch handles for various types of state transitions. Finally, there are also APIs to bootstrap Mojo IPC between two processes.

This document provides a brief guide to API usage with example code snippets. For a detailed API references please consult the headers in //mojo/public/c/system.

A Note About Multithreading

The Mojo C System API is entirely thread-agnostic. This means that all functions may be called from any thread in a process, and there are no restrictions on how many threads can use the same object at the same time.

Of course this does not mean you can completely ignore potential concurrency issues -- such as a handle being closed on one thread while another thread is trying to perform an operation on the same handle -- but there is nothing fundamentally incorrect about using any given API or handle from multiple threads.

A Note About Synchronization

Every Mojo API call is non-blocking and synchronously yields some kind of status result code, but the call's side effects -- such as affecting the state of one or more handles in the system -- may or may not occur asynchronously.

Mojo objects can be observed for interesting state changes in a way that is thread-agnostic and in some ways similar to POSIX signal handlers: *i.e.* user-provided notification handlers may be invoked at any time on arbitrary threads in the process. It is entirely up to the API user to take appropriate measures to synchronize operations against other application state.

The higher level system and bindings APIs provide helpers to simplify Mojo usage in this regard, at the expense of some flexibility.

Result Codes

Most API functions return a value of type `MojoResult`. This is an integral result code used to convey some meaningful level of detail about the result of a requested operation.

See //mojo/public/c/system/types.h for different possible values. See documentation for individual API calls for more specific contextual meaning of various result codes.

Handles

Every Mojo IPC primitive is identified by a generic, opaque integer handle of type `MojoHandle`. Handles can be acquired by creating new objects using various API calls, or by reading messages which contain attached handles.

A `MojoHandle` can represent a message pipe endpoint, a data pipe consumer, a data pipe producer, a shared buffer reference, a wrapped native platform handle such as a POSIX file descriptor or a Windows system handle, a trap object (see Signals & Traps below), or a process invitation (see Invitations below).

Message pipes, data pipes, shared buffers, and platform handles can all be attached to messages and sent over message pipes. Traps are an inherently process-local concept, and invitations are transmitted using special dedicated APIs.

Any `MojoHandle` may be closed by calling `MojoClose`:

```
MojoHandle x = DoSomethingToGetAValidHandle();
MojoResult result = MojoClose(x);
```

If the handle passed to `MojoClose` was a valid handle, it will be closed and `MojoClose` returns `MOJO_RESULT_OK`. Otherwise it returns `MOJO_RESULT_INVALID_ARGUMENT`.

Similar to native system handles on various popular platforms, `MojoHandle` values may be reused over time. Thus it is important to avoid logical errors which lead to misplaced handle ownership, double-closes, *etc.*

Message Pipes

A message pipe is a bidirectional messaging channel which can carry arbitrary unstructured binary messages with zero or more `MojoHandle` attachments to be transferred from one end of a pipe to the other. Message pipes work seamlessly across process boundaries or within a single process.

Invitations provide the means to bootstrap one or more primordial cross-process message pipes between two processes. Once such a pipe is established, additional handles -- including other message pipe handles -- may be sent to a remote process using that pipe (or in turn, over other pipes sent over that pipe, or pipes sent over *that* pipe, and so on...)

The public C System API exposes the ability to read and write messages on pipes and to create new message pipes.

See //mojo/public/c/system/message_pipe.h for detailed message pipe API documentation.

Creating Message Pipes

`MojoCreateMessagePipe` can be used to create a new message pipe:

```
MojoHandle a, b;
MojoResult result = MojoCreateMessagePipe(NULL, &a, &b);
```

After this snippet, `result` should be `MOJO_RESULT_OK` (it's really hard for this to fail!), and `a` and `b` will contain valid Mojo handles, one for each end of the new message pipe.

Any messages written to `a` are eventually readable from `b`, and any messages written to `b` are eventually readable from `a`. If `a` is closed at any point, `b` will eventually become aware of this fact; likewise if `b` is closed, `a` will become aware of that.

The state of these conditions can be queried and watched asynchronously as described in the Signals & Traps section below.

Creating Messages

Message pipes carry message objects which may or may not be serialized. You can create a new message object as follows:

```
MojoMessageHandle message;
MojoResult result = MojoCreateMessage(nullptr, &message);
```

Note that we have a special `MojoMessageHandle` type for message objects.

Messages may be serialized with attached data or unserialized with an opaque context value. Unserialized messages support lazy serialization, allowing custom serialization logic to be invoked only if and when serialization is required, e.g. when the message needs to cross a process or language boundary.

To make a serialized message, you might write something like:

```
void* buffer;
uint32_t buffer_size;
MojoResult result = MojoAppendMessageData(message, nullptr, 6, nullptr, 0,
                                          &buffer, &buffer_size);
memcpy(buffer, "hello", 6);
```

This attaches a data buffer to `message` with at least `6` bytes of storage capacity. The outputs returned in `buffer` and `buffer_size` can be used by the caller to fill in the message contents.

Multiple calls to `MojoAppendMessageData` may be made on a single message object, and each call appends to any payload and handles accumulated so far. Before you can transmit a message carrying data you must commit to never calling `MojoAppendMessageData` again. You do this by passing the `MOJO_APPEND_MESSAGE_DATA_FLAG_COMMIT_SIZE` flag:

```
MojoAppendMessageDataOptions options;
options.struct_size = sizeof(options);
options.flags = MOJO_APPEND_MESSAGE_DATA_FLAG_COMMIT_SIZE;
MojoResult result = MojoAppendMessageData(message, &options, 0, nullptr, 0,
                                          &buffer, &buffer_size);
```

Creating lazily-serialized messages is also straightforward:

```
struct MyMessage {
  // some interesting data...
};

void SerializeMessage(MojoMessageHandle message, uintptr_t context) {
  struct MyMessage* my_message = (struct MyMessage*)context;
```

```
  MojoResult result = MojoAppendMessageData(message, ...);
  // Serialize however you like.
}

void DestroyMessage(uintptr_t context) {
  free((void*)context);
}

MyMessage* data = malloc(sizeof(MyMessage));
// initialize *data...

MojoResult result = MojoSetMessageContext(
    message, (uintptr_t)data, &SerializeMessage, &DestroyMessage, nullptr);
```

If we change our mind and decide not to send the message, we can destroy it:

```
MojoResult result = MojoDestroyMessage(message);
```

Note that attempting to write a message will transfer ownership of the message object (and any attached handles) into the message pipe, and there is therefore no need to subsequently call `MojoDestroyMessage` on that message.

Writing Messages

```
result = MojoWriteMessage(a, message, nullptr);
```

`MojoWriteMessage` is a *non-blocking* call: it always returns immediately. If its return code is `MOJO_RESULT_OK` the message will eventually find its way to the other end of the pipe -- assuming that end isn't closed first, of course. If the return code is anything else, the message is deleted and not transferred.

In this case since we know `b` is still open, we also know the message will eventually arrive at `b`. `b` can be queried or watched to become aware of when the message arrives, but we'll ignore that complexity for now. See Signals & Traps below for more information.

NOTE: Although this is an implementation detail and not strictly guaranteed by the System API, it is true in the current implementation that the message will arrive at `b` before the above `MojoWriteMessage` call even returns, because `b` is in the same process as `a` and has never been transferred over another pipe.

Reading Messages

We can read a new message object from a pipe:

```
MojoMessageHandle message;
MojoResult result = MojoReadMessage(b, nullptr, &message);
```

and extract its data:

```
void* buffer = NULL;
uint32_t num_bytes;
MojoResult result = MojoGetMessageData(message, nullptr, &buffer, &num_bytes,
                                       nullptr, nullptr);
printf("Pipe says: %s", (const char*)buffer);
```

`result` should be `MOJO_RESULT_OK` and this snippet should write `"hello"` to `stdout`.

If we try were to try reading again now that there are no messages on `b`:

```
MojoMessageHandle message;
MojoResult result = MojoReadMessage(b, nullptr, &message);
```

We'll get a `result` of `MOJO_RESULT_SHOULD_WAIT`, indicating that the pipe is not yet readable.

Note that message also may not have been serialized if it came from within the same process, in which case it may have no attached data and `MojoGetMessageData` will return `MOJO_RESULT_FAILED_PRECONDITION`. The message's unserialized context can instead be retrieved using `MojoGetMessageContext`.

Messages read from a message pipe are owned by the caller and must be subsequently destroyed using `MojoDestroyMessage` (or, in theory, written to another pipe using `MojoWriteMessage`.)

Messages With Handles

Probably the most useful feature of Mojo IPC is that message pipes can carry arbitrary Mojo handles, including other message pipes. This is also straightforward.

Here's an example which creates two pipes, using the first pipe to transfer one end of the second pipe. If you have a good imagination you can pretend the first pipe spans a process boundary, which makes the example more practically interesting:

```
MojoHandle a, b;
MojoHandle c, d;
MojoCreateMessagePipe(NULL, &a, &b);
MojoCreateMessagePipe(NULL, &c, &d);

// Allocate a message with an empty payload and handle |c| attached. Note that
// this takes ownership of |c|, effectively invalidating its handle value.
MojoMessageHandle message;
void* buffer;
uint32_t buffer_size;
MojoCreateMessage(nullptr, &message);

MojoAppendMessageDataOptions options;
options.struct_size = sizeof(options);
```

```
options.flags = MOJO_APPEND_MESSAGE_DATA_FLAG_COMMIT_SIZE;
MojoAppendMessageData(message, &options, 2, &c, 1, &buffer, &buffer_size);
memcpy(buffer, "hi", 2);
MojoWriteMessage(a, message, nullptr);

// Some time later...
MojoHandle e;
uint32_t num_handles = 1;
MojoReadMessage(b, nullptr, &message);
MojoGetMessageData(message, nullptr, &buffer, &buffer_size, &e, &num_handles);
```

At this point the handle in `e` is now referencing the same message pipe endpoint which was originally referenced by `c`.

Note that `num_handles` above is initialized to 1 before we pass its address to `MojoGetMessageData`. This is to indicate how much `MojoHandle` storage is available at the output buffer we gave it (`&e` above).

If we didn't know how many handles to expect in an incoming message -- which is often the case -- we can use `MojoGetMessageData` to query for this information first:

```
MojoMessageHandle message;
void* buffer;
uint32_t num_bytes = 0;
uint32_t num_handles = 0;
MojoResult result = MojoGetMessageData(message, nullptr, &buffer, &num_bytes,
                                       nullptr, &num_handles);
```

If `message` has some non-zero number of handles, `result` will be `MOJO_RESULT_RESOURCE_EXHAUSTED`, and both `num_bytes` and `num_handles` will be updated to reflect the payload size and number of attached handles in the message.

Data Pipes

Data pipes provide an efficient unidirectional channel for moving large amounts of unframed data between two endpoints. Every data pipe has a fixed **element size** and **capacity**. Reads and writes must be done in sizes that are a multiple of the element size, and writes to the pipe can only be queued up to the pipe's capacity before reads must be done to make more space available.

Every data pipe has a single **producer** handle used to write data into the pipe and a single **consumer** handle used to read data out of the pipe.

Finally, data pipes support both immediate I/O -- reading into and writing out from user-supplied buffers -- as well as two-phase I/O, allowing callers to temporarily lock some portion of the data pipe in order to read or write its contents directly.

See //mojo/public/c/system/data_pipe.h for detailed data pipe API documentation.

Creating Data Pipes

Use `MojoCreateDataPipe` to create a new data pipe. The `MojoCreateDataPipeOptions` structure is used to configure the new pipe, but this can be omitted to assume the default options of a single-byte element size and an implementation-defined default capacity (64 kB at the time of this writing.)

```
MojoHandle producer, consumer;
MojoResult result = MojoCreateDataPipe(NULL, &producer, &consumer);
```

Immediate I/O

Data can be written into or read out of a data pipe using buffers provided by the caller. This is generally more convenient than two-phase I/O but is also less efficient due to extra copying.

```
uint32_t num_bytes = 12;
MojoResult result = MojoWriteData(producer, "datadatadata", &num_bytes,
                                  nullptr);
```

The above snippet will attempt to write 12 bytes into the data pipe, which should succeed and return `MOJO_RESULT_OK`. If the available capacity on the pipe was less than the amount requested (the input value of `*num_bytes`) this will copy what it can into the pipe and return the number of bytes written in `*num_bytes`. If no data could be copied this will instead return `MOJO_RESULT_SHOULD_WAIT`.

Reading from the consumer is a similar operation.

```
char buffer[64];
uint32_t num_bytes = 64;
MojoResult result = MojoReadData(consumer, nullptr, buffer, &num_bytes);
```

This will attempt to read up to 64 bytes, returning the actual number of bytes read in `*num_bytes`.

`MojoReadData` supports a number of interesting flags to change the behavior: you can peek at the data (copy bytes out without removing them from the pipe), query the number of bytes available without doing any actual reading of the contents, or discard data from the pipe without bothering to copy it anywhere.

This also supports a `MOJO_READ_DATA_FLAG_ALL_OR_NONE` which ensures that the call succeeds **only** if the exact number of bytes requested could be read. Otherwise such a request will fail with `MOJO_READ_DATA_OUT_OF_RANGE`.

Two-Phase I/O

Data pipes also support two-phase I/O operations, allowing a caller to temporarily lock a portion of the data pipe's storage for direct memory access.

```
void* buffer;
uint32_t num_bytes = 1024;
MojoResult result = MojoBeginWriteData(producer, nullptr, &buffer, &num_bytes);
```

This requests write access to a region of up to 1024 bytes of the data pipe's next available capacity. Upon success, `buffer` will point to the writable storage and `num_bytes` will indicate the size of the buffer there.

The caller should then write some data into the memory region and release it ASAP, indicating the number of bytes actually written:

```
memcpy(buffer, "hello", 6);
MojoResult result = MojoEndWriteData(producer, 6, nullptr);
```

Two-phase reads look similar:

```
void* buffer;
uint32_t num_bytes = 1024;
MojoResult result = MojoBeginReadData(consumer, nullptr, &buffer, &num_bytes);
// result should be MOJO_RESULT_OK, since there is some data available.

printf("Pipe says: %s", (const char*)buffer);  // Should say "hello".

// Say we only consumed one byte.
result = MojoEndReadData(consumer, 1, nullptr);

num_bytes = 1024;
result = MojoBeginReadData(consumer, nullptr, &buffer, &num_bytes);
printf("Pipe says: %s", (const char*)buffer);  // Should say "ello".
result = MojoEndReadData(consumer, 5, nullptr);
```

Shared Buffers

Shared buffers are chunks of memory which can be mapped simultaneously by multiple processes. Mojo provides a simple API to make these available to applications.

See //mojo/public/c/system/buffer.h for detailed shared buffer API documentation.

Creating Buffer Handles

Usage is straightforward. You can create a new buffer:

```
// Allocate a shared buffer of 4 kB.
MojoHandle buffer;
MojoResult result = MojoCreateSharedBuffer(4096, NULL, &buffer);
```

You can also duplicate an existing shared buffer handle:

```
MojoHandle another_name_for_buffer;
MojoResult result = MojoDuplicateBufferHandle(buffer, NULL,
                                              &another_name_for_buffer);
```

This is useful if you want to retain a handle to the buffer while also sharing handles with one or more other clients. The allocated buffer remains valid as long as at least one shared buffer handle exists to reference it.

Mapping Buffers

You can map (and later unmap) a specified range of the buffer to get direct memory access to its contents:

```
void* data;
MojoResult result = MojoMapBuffer(buffer, 0, 64, nullptr, &data);

*(int*)data = 42;
result = MojoUnmapBuffer(data);
```

A buffer may have any number of active mappings at a time, in any number of processes.

Read-Only Handles

An option can also be specified on `MojoDuplicateBufferHandle` to ensure that the newly duplicated handle can only be mapped to read-only memory:

```
MojoHandle read_only_buffer;
MojoDuplicateBufferHandleOptions options;
options.struct_size = sizeof(options);
options.flags = MOJO_DUPLICATE_BUFFER_HANDLE_FLAG_READ_ONLY;
MojoResult result = MojoDuplicateBufferHandle(buffer, &options,
                                              &read_only_buffer);

// Attempt to map and write to the buffer using the read-only handle:
void* data;
result = MojoMapBuffer(read_only_buffer, 0, 64, nullptr, &data);
*(int*)data = 42;  // CRASH
```

NOTE: One important limitation of the current implementation is that read-only handles can only be produced from a handle that was originally created by `MojoCreateSharedBuffer` (*i.e.*, you cannot create a read-only duplicate from a non-read-only duplicate), and the handle cannot have been transferred over a message pipe first.

Native Platform Handles (File Descriptors, Windows Handles, *etc.*)

Native platform handles to system objects can be wrapped as Mojo handles for seamless transit over message pipes. Mojo currently supports wrapping POSIX file descriptors, Windows handles, Mach ports, and Fuchsia zx_handles.

See //mojo/public/c/system/platform_handle.h for detailed platform handle API documentation.

Wrapping Basic Handle Types

Wrapping a POSIX file descriptor is simple:

```
MojoPlatformHandle platform_handle;
platform_handle.struct_size = sizeof(platform_handle);
platform_handle.type = MOJO_PLATFORM_HANDLE_TYPE_FILE_DESCRIPTOR;
platform_handle.value = (uint64_t)fd;
MojoHandle handle;
MojoResult result = MojoWrapPlatformHandle(&platform_handle, nullptr, &handle);
```

Note that at this point `handle` effectively owns the file descriptor and if you were to call `MojoClose(handle)`, the file descriptor would be closed too; but we're not going to close it here! We're going to pretend we've sent it over a message pipe, and now we want to unwrap it on the other side:

```
MojoPlatformHandle platform_handle;
platform_handle.struct_size = sizeof(platform_handle);
MojoResult result = MojoUnwrapPlatformHandle(handle, nullptr, &platform_handle);
int fd = (int)platform_handle.value;
```

The situation looks nearly identical for wrapping and unwrapping Windows handles and Mach ports.

Wrapping Shared Buffer Handles

Unlike other handle types, shared buffers have special meaning in Mojo, and it may be desirable to wrap a native platform handle -- along with some extra metadata -- such that be treated like a real Mojo shared buffer handle. Conversely it can also be useful to unpack a Mojo shared buffer handle into a native platform handle which references the buffer object. Both of these things can be done using the `MojoWrapPlatformSharedBuffer` and `MojoUnwrapPlatformSharedBuffer` APIs.

On Windows, the wrapped platform handle must always be a Windows handle to a file mapping object.

On OS X, the wrapped platform handle must be a memory-object send right.

On all other POSIX systems, the wrapped platform handle must be a file descriptor for a shared memory object.

Signals & Traps

Message pipe and data pipe (producer and consumer) handles can change state in ways that may be interesting to a Mojo API user. For example, you may wish to know when a message pipe handle has messages available to be read or when its peer has been closed. Such states are reflected by a fixed set of boolean signals on each pipe handle.

Signals

Every message pipe and data pipe handle maintains a notion of **signaling state** which may be queried at any time. For example:

```
MojoHandle a, b;
MojoCreateMessagePipe(NULL, &a, &b);

MojoHandleSignalsState state;
MojoResult result = MojoQueryHandleSignalsState(a, &state);
```

The `MojoHandleSignalsState` structure exposes two fields: `satisfied_signals` and `satisfiable_signals`. Both of these are bitmasks of the type `MojoHandleSignals` (see //mojo/public/c/system/types.h for more details.)

The `satisfied_signals` bitmask indicates signals which were satisfied on the handle at the time of the call, while the `satisfiable_signals` bitmask indicates signals which were still possible to satisfy at the time of the call. It is thus by definition always true that:

```
(satisfied_signals | satisfiable_signals) == satisfiable_signals
```

In other words a signal obviously cannot be satisfied if it is no longer satisfiable. Furthermore once a signal is unsatisfiable, *i.e.* is no longer set in `sastisfiable_signals`, it can **never** become satisfiable again.

To illustrate this more clearly, consider the message pipe created above. Both ends of the pipe are still open and neither has been written to yet. Thus both handles start out with the same signaling state:

Field	State
`satisfied_signals`	`MOJO_HANDLE_SIGNAL_WRITABLE`
`satisfiable_signals`	`MOJO_HANDLE_SIGNAL_READABLE + MOJO_HANDLE_SIGNAL_WRITABLE + MOJO_HANDLE_SIGNAL_PEER_CLOSED`

Writing a message to handle `b` will eventually alter the signaling state of `a` such that `MOJO_HANDLE_SIGNAL_READABLE` also becomes satisfied. If we were to then close `b`, the signaling state of `a` would look like:

Field	State
`satisfied_signals`	`MOJO_HANDLE_SIGNAL_READABLE + MOJO_HANDLE_SIGNAL_PEER_CLOSED`
`satisfiable_signals`	`MOJO_HANDLE_SIGNAL_READABLE + MOJO_HANDLE_SIGNAL_PEER_CLOSED`

Note that even though `a`'s peer is known to be closed (hence making `a` permanently unwritable) it remains readable because there's still an unread received message waiting to be read from `a`.

Finally if we read the last message from `a` its signaling state becomes:

Field	State
`satisfied_signals`	`MOJO_HANDLE_SIGNAL_PEER_CLOSED`

Field	State
`satisfiable_signals`	`MOJO_HANDLE_SIGNAL_PEER_CLOSED`

and we know definitively that `a` can never be read from again.

Trapping Signals

The ability to query a handle's signaling state can be useful, but it's not sufficient to support robust and efficient pipe usage. Mojo traps empower users with the ability to **trap** changes in a handle's signaling state and automatically invoke a notification handler in response.

When a trap is created it must be bound to a function pointer matching the following signature, defined in //mojo/public/c/system/trap.h:

```
typedef void (*MojoTrapEventHandler)(const struct MojoTrapEvent* event);
```

The `event` parameter conveys details about why the event handler is being invoked. The handler may be called **at any time** and **from any thread**, so it is critical that handler implementations account for this.

It's also helpful to understand a bit about the mechanism by which the handler can be invoked. Essentially, any Mojo C System API call may elicit a handle state change of some kind. If such a change is relevant to conditions watched by a trap, and that trap is in a state which allows it raise a corresponding notification, its notification handler will be invoked synchronously some time before the stack unwinds beyond the outermost System API call on the current thread.

Handle state changes can also occur as a result of incoming IPC from an external process. If a pipe in the current process is connected to an endpoint in another process and the internal Mojo system receives an incoming message bound for the local endpoint, the arrival of that message may trigger a state change on the receiving handle and may therefore invoke one or more traps' notification handlers as a result.

The `MOJO_TRAP_EVENT_FLAG_WITHIN_API_CALL` flag on the `flags` field of `event` is used to indicate whether the handler was invoked due to such an internal system IPC event (if the flag is unset), or if it was invoked synchronously due to some local API call (if the flag is set.) This distinction can be useful to make in certain cases to *e.g.* avoid accidental reentrancy in user code.

Creating a Trap

Creating a trap is simple:

```
void OnNotification(const struct MojoTrapEvent* event) {
  // ...
}

MojoHandle t;
MojoResult result = MojoCreateTrap(&OnNotification, NULL, &t);
```

Like all other `MojoHandle` types, traps may be destroyed by closing them with `MojoClose`. Unlike most other `MojoHandle` types, trap handles are **not** transferrable across message pipes.

In order for a trap to be useful, it has have at least one **trigger** attached to it.

Adding a Trigger to a Trap

Any given trap can watch any given (message or data pipe) handle for some set of signaling conditions. A handle may be watched simultaneously by multiple traps, and a single trap can watch multiple different handles simultaneously.

```
MojoHandle a, b;
MojoCreateMessagePipe(NULL, &a, &b);

// Watch handle |a| for readability.
const uintptr_t context = 1234;
MojoResult result = MojoAddTrigger(t, a, MOJO_HANDLE_SIGNAL_READABLE,
                                   MOJO_TRIGGER_CONDITION_SIGNALS_SATISFIED,
                                   context, NULL);
```

We've successfully instructed trap `t` to begin watching pipe handle `a` for readability. However, our recently created trap is still in a **disarmed** state, meaning that it will never fire a notification pertaining to this trigger. It must be **armed** before that can happen.

Arming a Trap

In order for a trap to invoke its notification handler in response to a relevant signaling state change on a watched handle, it must first be armed. A trap may only be armed if none of its attached triggers would elicit a notification immediately once armed.

In this case `a` is clearly not yet readable, so arming should succeed:

```
MojoResult result = MojoArmTrap(t, NULL, NULL, NULL);
```

Now we can write to `b` to make `a` readable:

```
MojoMessageHandle m;
MojoCreateMessage(nullptr, &m);
MojoWriteMessage(b, m, nullptr);
```

Eventually -- and in practice possibly before `MojoWriteMessage` even returns -- this will cause `OnNotification` to be invoked on the calling thread with the `context` value (*i.e.* 1234) that was given when the trigger was added to the trap.

The `result` field of the event will be `MOJO_RESULT_OK` to indicate that the trigger's condition has been met. If the handle's state had instead changed in such a way that the trigger's condition could never be met

again (*e.g.* if `b` were instead closed), `result` would instead indicate `MOJO_RESULT_FAILED_PRECONDITION`.

NOTE: Immediately before a trigger decides to invoke its event handler, it automatically disarms itself to prevent another state change from eliciting another notification. Therefore a trap must be repeatedly rearmed in order to continue dispatching events.

As noted above, arming a watcher may fail if any of its triggers would be activated immediately. In that case, the caller may provide buffers to `MojoArmTrap` to receive information about a subset of the triggers which caused it to fail:

```
// Provide some storage for information about triggers that would have been
// activated immediately.
uint32_t num_blocking_events = 2;
MojoTrapEvent blocking_events[2] = {{sizeof(MojoTrapEvent)},
                                    {sizeof(MojoTrapEvent)}};
MojoResult result = MojoArmTrap(t, NULL, &num_blocking_events,
                                &blocking_events);
```

Because `a` is still readable this operation will now fail with `MOJO_RESULT_FAILED_PRECONDITION`. The input value of `num_blocking_events` informs `MojoArmTrap` that it may store information regarding up to 2 triggers which have prevented arming. In this case of course there is only one active trigger, so upon return we will see:

- `num_blocking_events` is `1`.
- `blocking_events[0].trigger_context` is `1234`.
- `blocking_events[0].result` is `MOJO_RESULT_OK`
- `blocking_events[0].signals_state` is the last known signaling state of handle `a`.

In other words the stored information mirrors what would have been the resulting event structure if the trap were allowed to arm and then notify immediately.

Removing a Trigger

There are three ways a trigger can be removed:

- The handle being watched by the trigger is closed
- The trap handle is closed, in which case all of its attached triggers are implicitly removed.
- `MojoRemoveTrigger` is called for a given `context`.

In the above example this means any of the following operations will cancel the watch on `a`:

```
// Close the watched handle...
MojoClose(a);

// OR close the trap handle...
MojoClose(t);
```

```
// OR explicitly remove it.
MojoResult result = MojoRemoveTrigger(t, 1234, NULL);
```

In every case the trap's event handler is invoked for the cancelled trigger(es) regardless of whether or not the trap was armed at the time. The event handler receives a `result` of `MOJO_RESULT_CANCELLED` for each of these invocations, and this is guaranteed to be the final event for any given trigger context.

Practical Trigger Context Usage

It is common and probably wise to treat a trigger's `context` value as an opaque pointer to some thread-safe state associated in some way with the handle being watched. Here's a small example which uses a single trap to watch both ends of a message pipe and accumulate a count of messages received at each end.

```
// NOTE: For the sake of simplicity this example code is not in fact
// thread-safe. As long as there's only one thread running in the process and
// no external process connections, this is fine.

struct WatchedHandleState {
  MojoHandle trap;
  MojoHandle handle;
  int message_count;
};

void OnNotification(const struct MojoTrapEvent* event) {
  struct WatchedHandleState* state =
      (struct WatchedHandleState*)(event->trigger_context);
  MojoResult rv;

  if (event->result == MOJO_RESULT_CANCELLED) {
    // Cancellation is always the last event and is guaranteed to happen for
    // every context, assuming no handles are leaked. We treat this as an
    // opportunity to free the WatchedHandleState.
    free(state);
    return;
  }

  if (result == MOJO_RESULT_FAILED_PRECONDITION) {
    // No longer readable, i.e. the other handle must have been closed. Better
    // cancel. Note that we could also just call MojoClose(state->trap) here
    // since we know there's only one attached trigger.
    MojoRemoveTrigger(state->trap, event->trigger_context, NULL);
    return;
  }

  // This is the only handle watched by the trap, so as long as we can't arm
  // the watcher we know something's up with this handle. Try to read messages
  // until we can successfully arm again or something goes terribly wrong.
```

```
  while (MojoArmTrap(state->trap, NULL NULL, NULL) ==
         MOJO_RESULT_FAILED_PRECONDITION) {
    rv = MojoReadMessageNew(state->handle, NULL, NULL, NULL,
                            MOJO_READ_MESSAGE_FLAG_MAY_DISCARD);
    if (rv == MOJO_RESULT_OK) {
      state->message_count++;
    } else if (rv == MOJO_RESULT_FAILED_PRECONDITION) {
      MojoRemoveTrigger(state->trap, event->trigger_context, NULL);
      return;
    }
  }
}

MojoHandle a, b;
MojoCreateMessagePipe(NULL, &a, &b);

MojoHandle a_trap, b_trap;
MojoCreateTrap(&OnNotification, NULL, &a_trap);
MojoCreateTrap(&OnNotification, NULL, &b_trap)

struct WatchedHandleState* a_state = malloc(sizeof(struct WatchedHandleState));
a_state->trap = a_trap;
a_state->handle = a;
a_state->message_count = 0;

struct WatchedHandleState* b_state = malloc(sizeof(struct WatchedHandleState));
b_state->trap = b_trap;
b_state->handle = b;
b_state->message_count = 0;

MojoAddTrigger(a_trap, a, MOJO_HANDLE_SIGNAL_READABLE,
               MOJO_TRIGGER_CONDITION_SIGNALS_SATISFIED, (uintptr_t)a_state,
               NULL);
MojoAddTrigger(b_trap, b, MOJO_HANDLE_SIGNAL_READABLE,
               MOJO_TRIGGER_CONDITION_SIGNALS_SATISFIED, (uintptr_t)b_state,
               NULL);

MojoArmTrap(a_trap, NULL, NULL, NULL);
MojoArmTrap(b_trap, NULL, NULL, NULL);
```

Now any writes to `a` will increment `message_count` in `b_state`, and any writes to `b` will increment `message_count` in `a_state`.

If either `a` or `b` is closed, both watches will be cancelled - one because watch cancellation is implicit in handle closure, and the other because its watcher will eventually detect that the handle is no longer readable.

Invitations

TODO.

For now see the C header and the documentation for the equivalent C++ API.