Linear Memory Inspector

Attention: Externally visible, non-confidential
Author: kimanh@chromium.org
Status: Inception | Draft | Accepted | Done
Created: 2020-10-07 / **Last Updated**: 2020-11-19

One-page overview

Summary

This design doc outlines the design proposal for the implementation of the Linear Memory Inspector for inspecting `ArrayBuffers` (in particular `WebAssembly.Memorys`). First and foremost, this is for developers to inspect the Wasm memory of a Wasm instance. The memory inspector allows the developer to view, navigate and inspect the values in the memory.

Platforms

Desktop, Android.

Team

kimanh@chromium.org
bmeurer@chromium.org
petermueller@google.com for UX design

Tracking issue

crbug.com/1110202

Value proposition

A Linear Memory Inspector can help the developer to better debug and understand the underlying Wasm application. Up to date, the Wasm memory is only shown as a UInt8Array in the scope view, without any way to properly inspect and understand the raw values. The inspector offers a way to navigate these values, select values and interpret them as different types, such that the developer can make sense of what is contained in the memory.

Code affected

DevTools Front-End; Sources View

Signed off by

Name	Write (not) LGTM in this row
bmeurer@chromium.org	LGTM
jacktfranklin@chromium.org	LGTM
alexrudenko@chromium.org	LGTM
leese@chromium.org	LGTM
hwi@chromium.org	LGTM

Core user stories

As a developer I want to be able to view the Wasm memory and inspect its values. This includes being able to navigate the memory and to be able to view different interpretations of selected memory values.

Design

UX Design

For the complete UX design, see UX deck (Google internal doc, not externally shared)

Implementation Design

The design of the memory inspector is described in the following, split up into two parts: first, the web components for the UI, and second, the change in the devtools front_end that are required to display and interact with the inspector within DevTools.

1. The Linear Memory Inspector Web Components and their data synchronization on updates

The Linear Memory Inspector will make use of three web components, as outlined below:

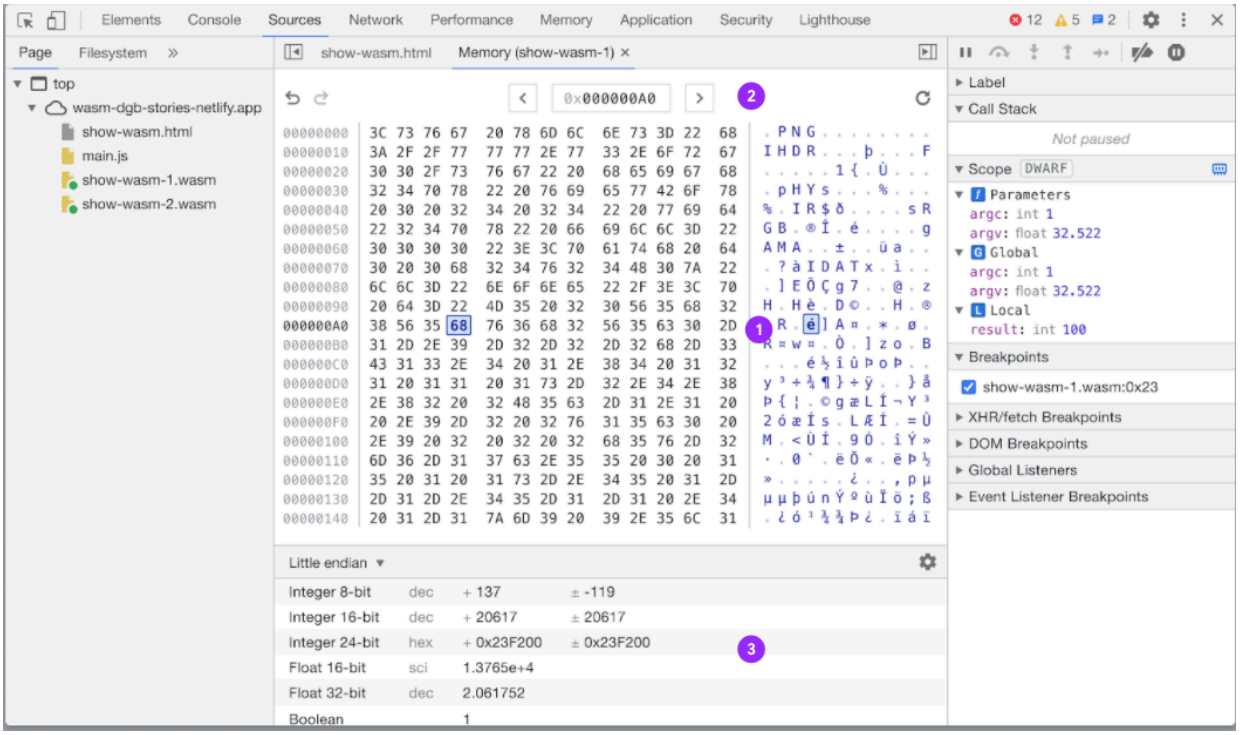


Figure 1: The Linear Memory Inspector view

1. The main byte viewer (`LinearMemoryViewer`),
2. The navigation (`LinearMemoryNavigator`), and
3. The value interpreter (`LinearMemoryValueInterpreter`).

The LinearMemoryValueInterpreter renders the settings toolbar (for changing endianness and values to show, as well as one of two subcomponents (depending on whether the settings are currently changed or not):

The first subcomponent will be the `ValueInterpreterDisplay` subcomponent, which displays the selected values with their configuration:

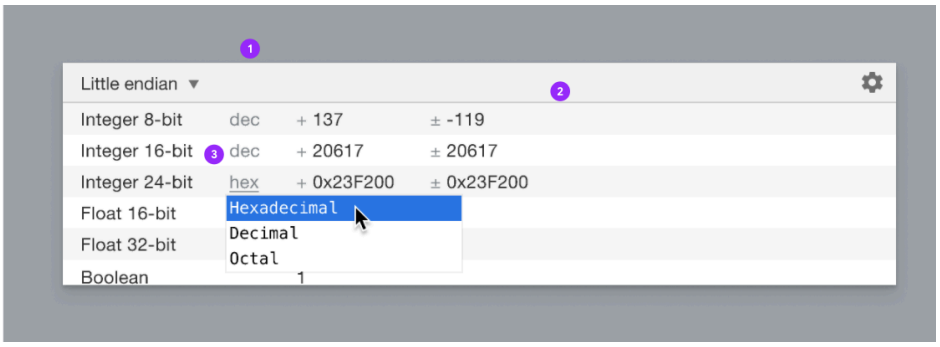


Figure 2: The `ValueInterpreterDisplay` subcomponent

The second subcomponent will be the `ValueInterpreterSettings` subcomponent, which allows the user to select the data types to show:

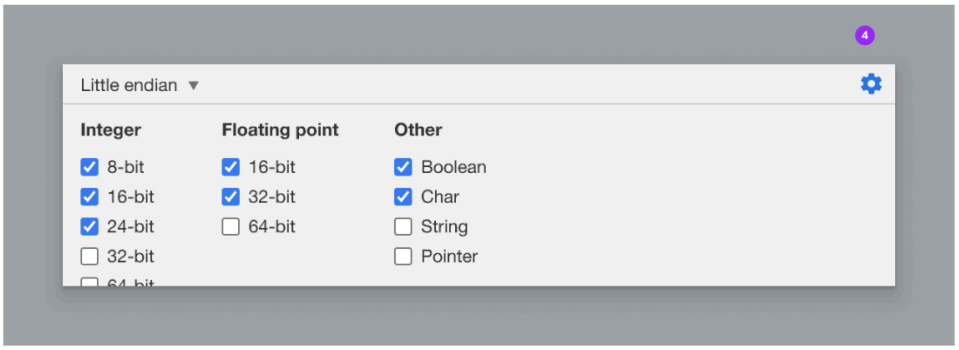


Figure 3: The `ValueInterpreterSettings` subcomponent

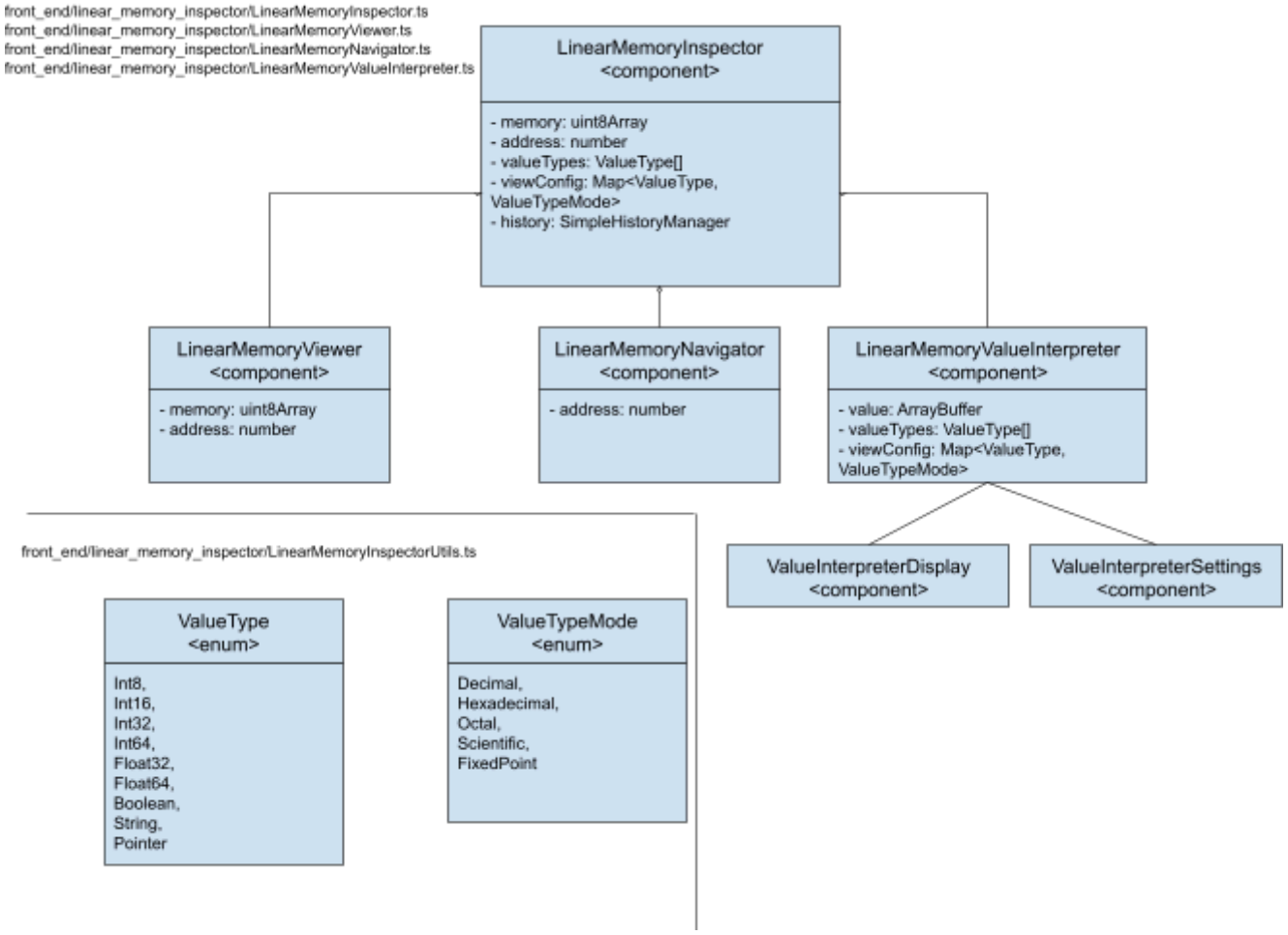


Figure 3: Web components

In the following is a brief explanation on what the components do. Note that the event types mentioned below are not explicitly included as classes in Figure 3.

The LinearMemoryViewer

The linear memory viewer shows an extract of the current memory (`$memory`). Its address (`$address`) determines which part of the memory is currently shown. The bytes in the view are going to be wrapped at 4 byte boundaries.

This component triggers a :

- `AddressChangedEvent` if the user selects a byte in the current view.

The LinearMemoryNavigator

The linear memory navigator allows the navigation through the current memory. This includes navigating to the next/previous page, returning to previously selected addresses and input-ing the next address to view. It stores the current address in `$address`.

This component triggers a

- `PageNavigationEvent` (on navigating the page),
- `AddressChangedEvent` (on typing in an address),
- `RefreshEvent` (on clicking refresh), or a
- `HistoryNavigationEvent` (on clicking on backward/forward).

The LinearMemoryValueInterpreter

The linear memory value interpreter shows the current value of the current address as different types. These types can be selected in the settings view (see Figure 2).
It keeps an `ArrayBuffer` (`$value`) of the current address in order to be able to render different types. The size of that `ArrayBuffer` is the maximum number of bytes that we want to interpret as one value. The types that should be rendered are stored in `$valueTypes`, and their viewing mode (hex, oct, dec, sci) is defined in a configuration map (`$viewConfig`).

This component triggers a

- ValueTypeToggledEvent (on toggling the types to be included or not), and a
- ViewingModeChangedEvent (on changing the viewing mode).

The LinearMemoryValueInspector

The `LinearMemoryInspector` component renders the three subcomponents mentioned above. It keeps the state of the linear memory inspector and delegates the rendering to its subcomponents. Therefore, all events from its subcomponents are handled within this parent component. For example, updating the address in the `LinearMemoryNavigator` will cause an `AddressChangedEvent,` that is handled by the `LinearMemoryInspector`. Consequently, it will update the address field and trigger a re-render.

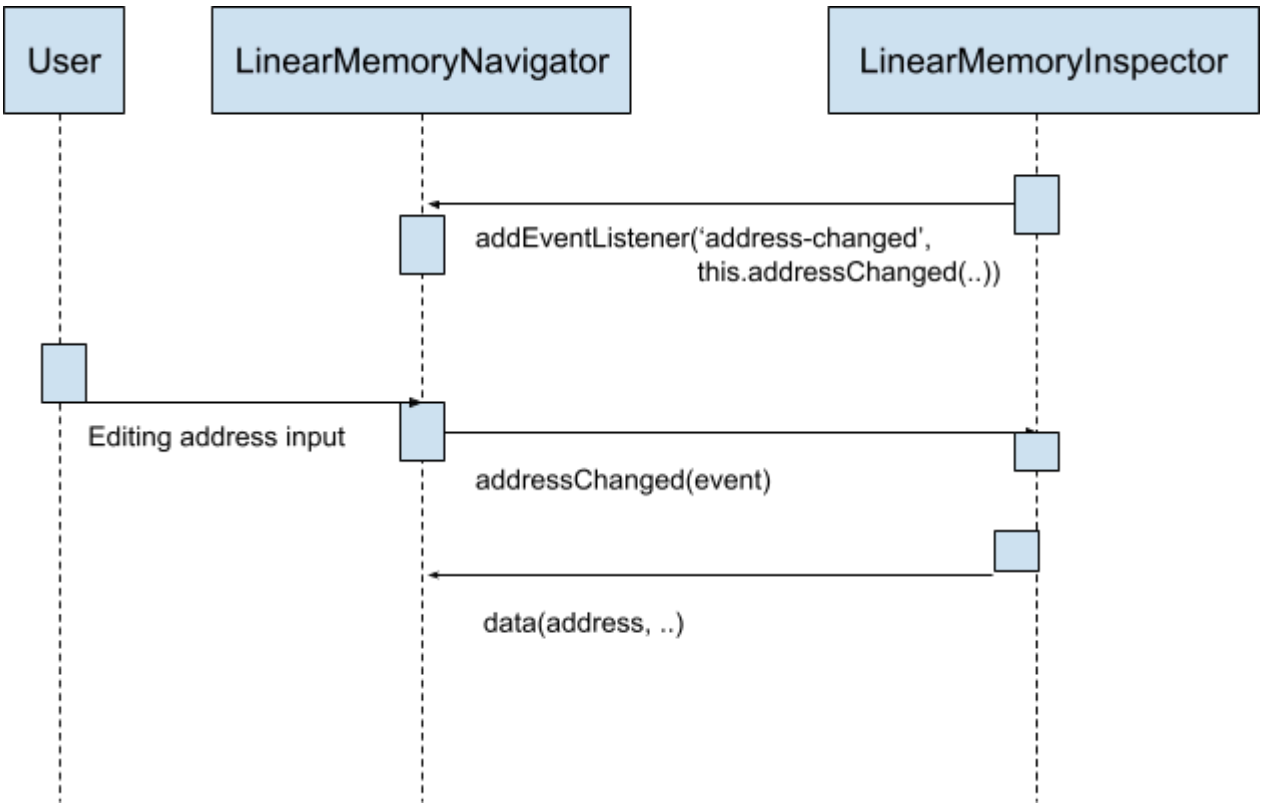


Figure 4: Sequence diagram for editing the address

LinearMemoryInspectorUtils

Contains the `ValueType`, and the `ValueTypeMode`.
For all integer types, the `ValueType` that is allowed is:

- Decimal
- Octal
- Hexadecimal

For all float types, the `ValueType` that is allowed is:

- Scientific
- FixedPoint

Strings and Pointers do not have any value type assigned.

2. DevTools Front-end changes required to show the Linear Memory Inspector

As for the MVP, we first want to show the Linear Memory Inspector upon a right-click on the Wasm memory:

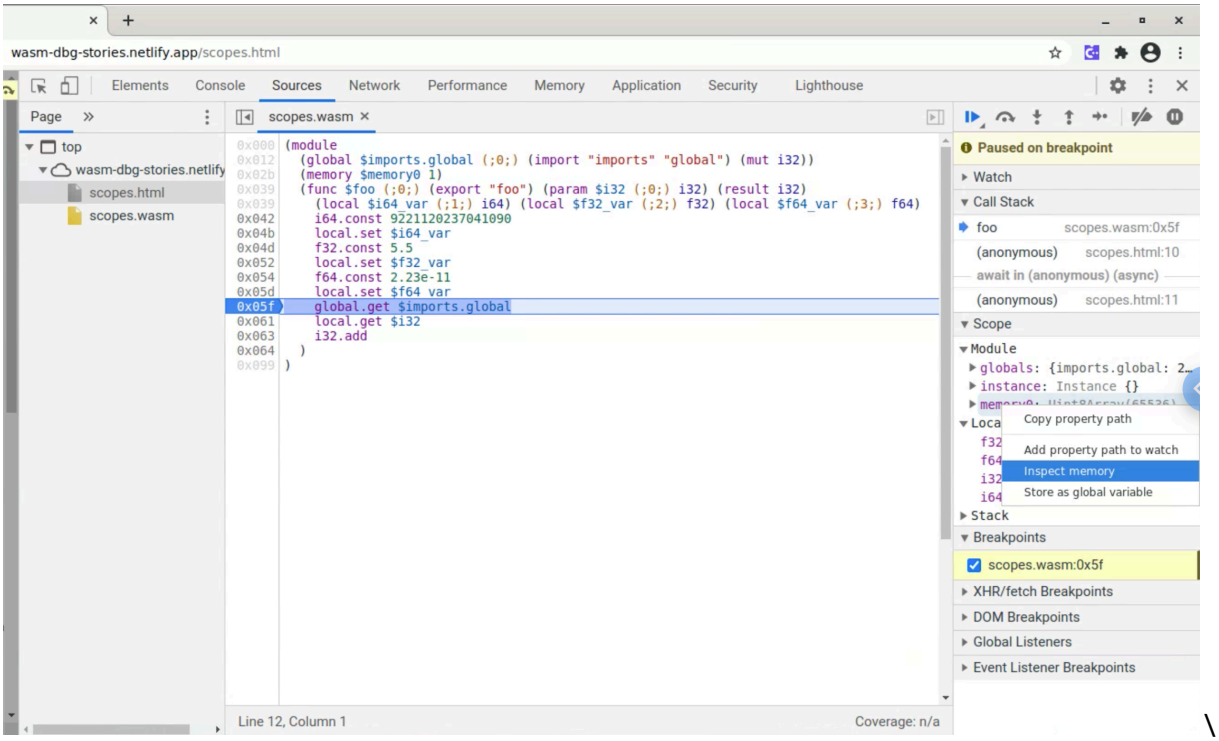


Figure 5: Entering the Linear Memory Inspector via Scope View

The Linear Memory Inspector is then shown as a drawer at the bottom of DevTools:

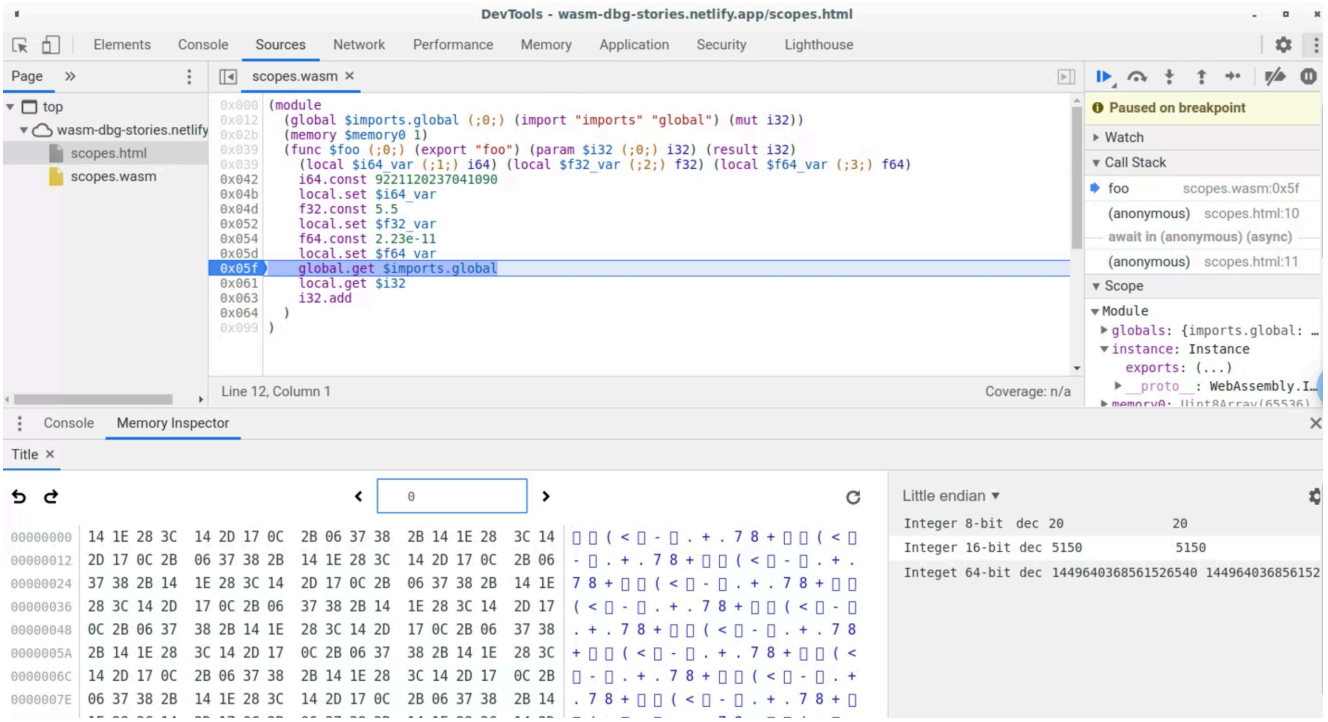


Figure 6: Linear Memory Inspector included as a drawer at the bottom of DevTools

Figure 7 shows a class diagram designed to show the Linear Memory Inspector as suggested above:

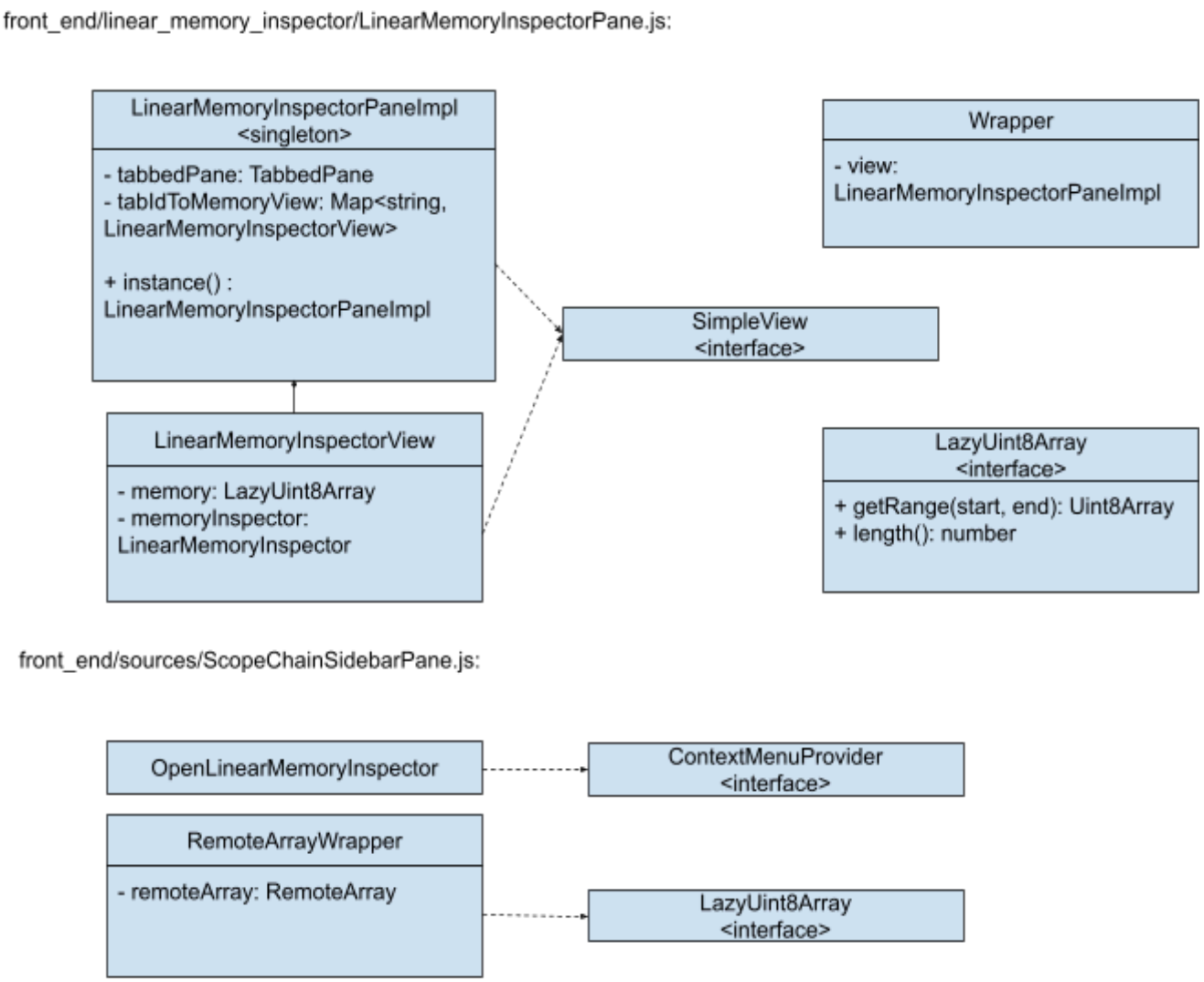


Figure 7: Diagram with classes required to show the Linear Memory Inspector on a right click of the Wasm memory in the scope view

LinearMemoryInspectorPaneImpl

The `LinearMemoryInspectorPaneImpl` shows the tabbed pane at the bottom of DevTools. It is opened upon a call to `UI.ViewManager.ViewManager.instance().showView('linear-memory-inspector')`. The ViewManager accesses the `LinearMemoryInspectorPaneImpl` through the Wrapper class. The wrapper is necessary in order to ensure that the `LinearMemoryInspectorPaneImpl` is a Singleton, which is not instantiated through the constructor, but rather through an `instance()` method.

Bookkeeping of tabs and history

The `LinearMemoryInspectorPaneImpl` needs to keep track of all the opened LinearMemoryInspectorViews. For this, it keeps a `$tabIdToMemoryView` map for events, such as closing the tab. The `$tabId` will be the `$scriptId`.

The `LinearMemoryInspector` records the history of the addresses visited in order to be able to record the history for navigated addresses, and to support navigation through history.

OpenLinearMemoryInspector NEW

Opening the LinearMemoryInspectorPaneImpl through the Scope View Context Menu

The `OpenLinearMemoryInspector` class within the `ScopeChainSidebarPane.js` (see Figure 7) implements the UI.ContextMenu.Provider class that is required to populate the context menu on a right click within the Scope View.

On selecting 'Inspect memory' from the context menu, the `OpenLinearMemoryInspector` gets the `LinearMemoryInspectorPaneImpl` instance and forwards the information on the RemoteObject that wraps the Uint8Array, along with other information on the current call stack.

A call to UI.ViewManager.ViewManager.instance().showView('linear-memory-inspector') finally opens a LinearMemoryInspector (simplified) through the wrapper:

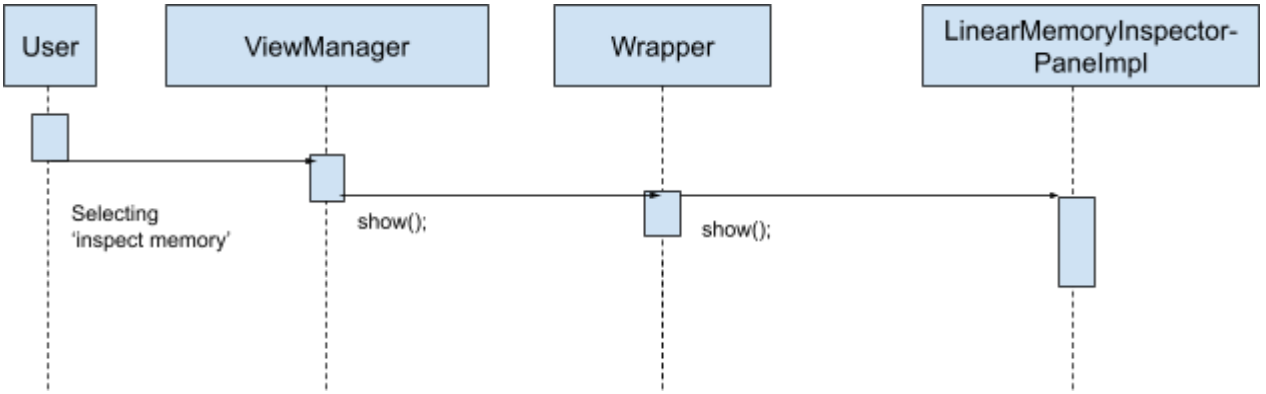


Figure 8: Simplified sequence diagram to open a LinearMemoryInspectorView from the Scope View

LazyUint8Array NEW

The LazyUint8Array is an interface that wraps (some form of) a Uint8Array. In this particular use case we actually do not have a Uint8Array in the scope view, but instead a RemoteObject that represents a Uint8Array. This interface is a wrapper that provides methods to selectively retrieve a Uint8Array for a certain range. The reason for using a LazyUint8Array is outlined in part 3.

Things left for later

There are several features that I haven't investigated yet, but fall into the responsibility of the `LinearMemoryInspectorPaneImpl`:

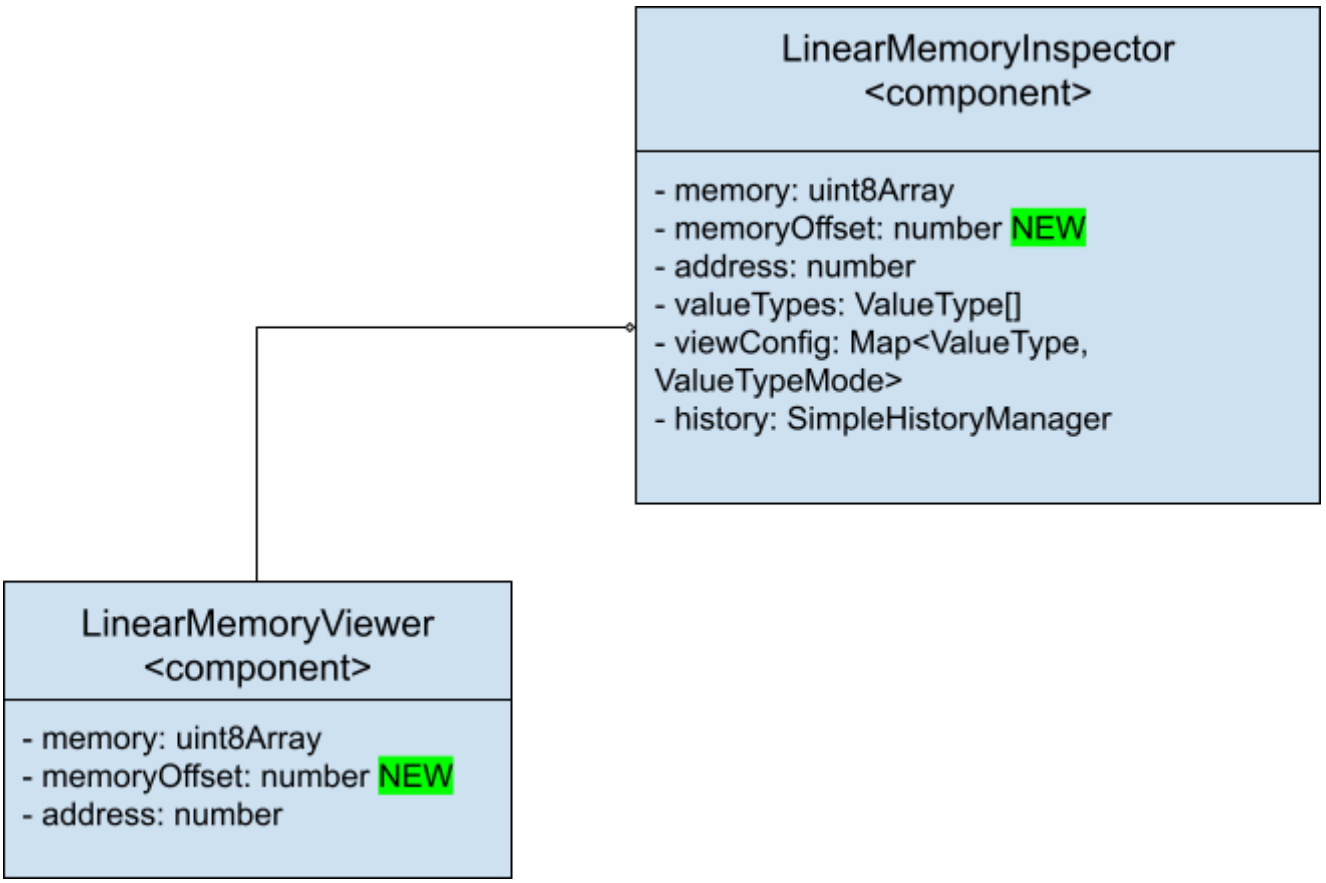
- Pulling the latest memory on refresh. We will store the `$scriptId` as `$tabId`, and should be able to pull the latest data from that.
- Closing the views; when do we want to close the view? Whenever the corresponding source code is closed?

3. Ensuring that speed and memory consumption are independent of the actual buffer size NEW

The incoming buffer for the Wasm memory can be large. Currently, the buffer is transferred to the Chrome DevTools front-end through a RemoteObject. Unpacking each element requires to separately

request each element from the back-end, which incurs a significant overhead (and thus a long, visible delay for the user).

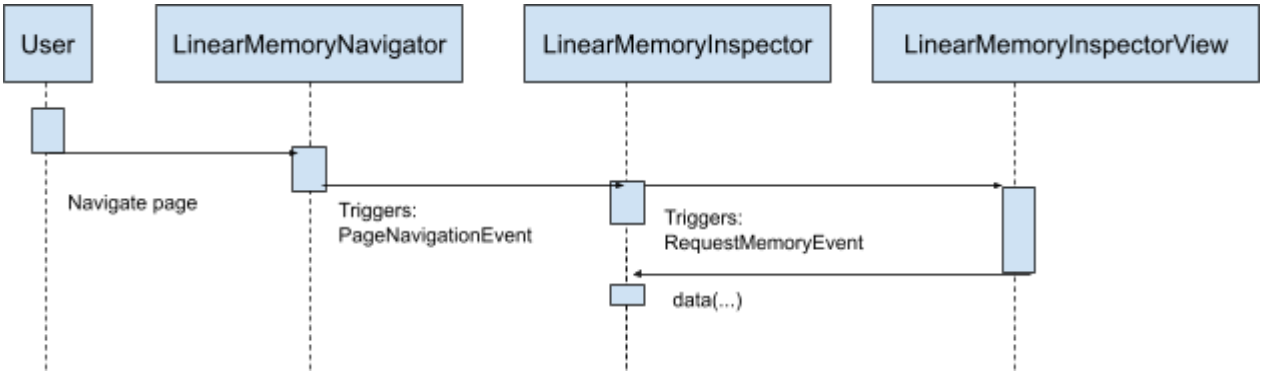
Instead of requesting all elements within the buffer, we only want to access those that are relevant, i.e. those that we show in the view. In order to make the `LinearMemoryViewer` deal with only showing a portion of the actual buffer, the `LinearMemoryViewer` now also takes in a `memoryOffset`:



The memoryOffset defines the **index of the first element in the memory** within the original Uint8Array. This way, the LinearMemoryViewer can still correctly render the address of the bytes in the view.

Since the LinearMemoryInspector and the LinearMemoryViewer only see a portion of the actual Uint8Array, changes need to be introduced in order to allow for resizing events (now showing more elements than currently available) or for navigation changes.

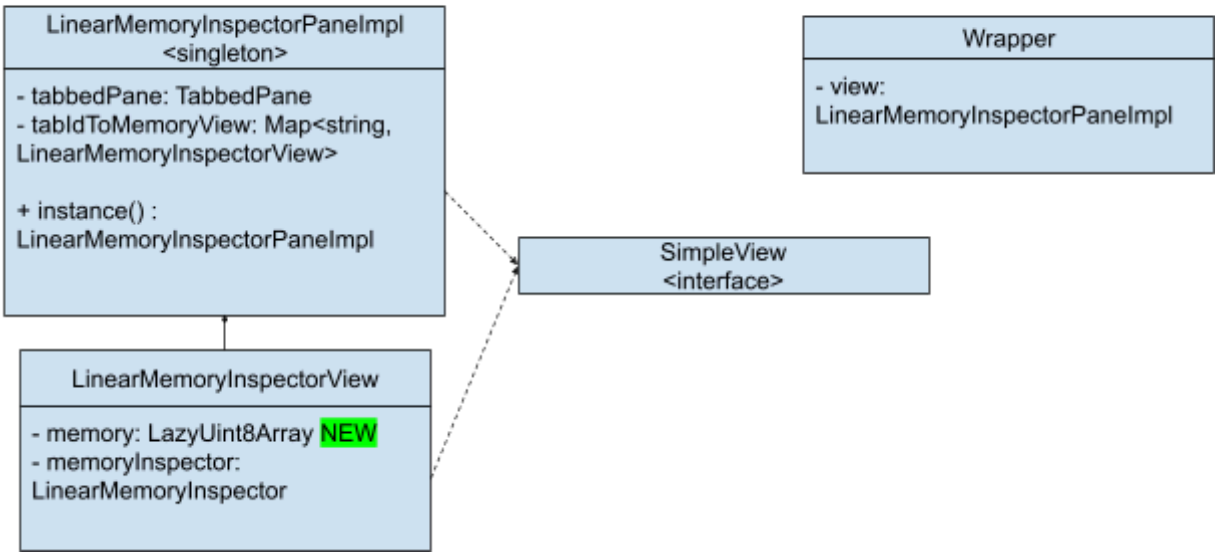
The `LinearMemoryInspector` is responsible to request more memory if required. On navigation changes and resize events, the `LinearMemoryInspector` requests a different Uint8Array range from the `LinearMemoryView` by sending a `MemoryRequestEvent`:



As a response, the LinearMemoryView will update the Uint8Array and set the data on the `LinearMemoryInspector`, which will trigger a re-render.

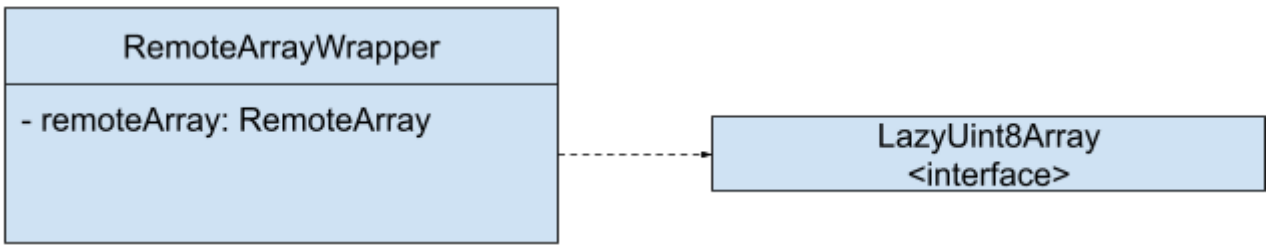
The `LinearMemoryInspectorView` keeps a `LazyUint8Array` instead of the Uint8Array to process these requests. After receiving the request for memory, it will use the LazyUint8Array instance in order to get the requested Uint8Array range and set the new data on the LinearMemoryInspector.

front_end/linear_memory_inspector/LinearMemoryInspectorPane.js:



In this particular use case the `LazyUint8Array` wraps a `RemoteObject`, see `RemoteArrayWrapper`:

front_end/sources/ScopeChainSidebarPane.js:



Rollout plan

1. Experiment first, gated behind the Wasm Debugging Experiment
2. Waterfall

Core principle considerations

Speed

The linear memory inspector is only showing one excerpt of the memory at a time. The memory consumption should be independent of the actual size of the memory to be inspected. This is explored above in this section.

Security

The Linear Memory Inspector will be gated behind the Wasm Dwarf experiment for now.

Simplicity

Originally the proposal was to show the Linear Memory Inspector within the SourcesView, i.e. as a tab next to the sources:

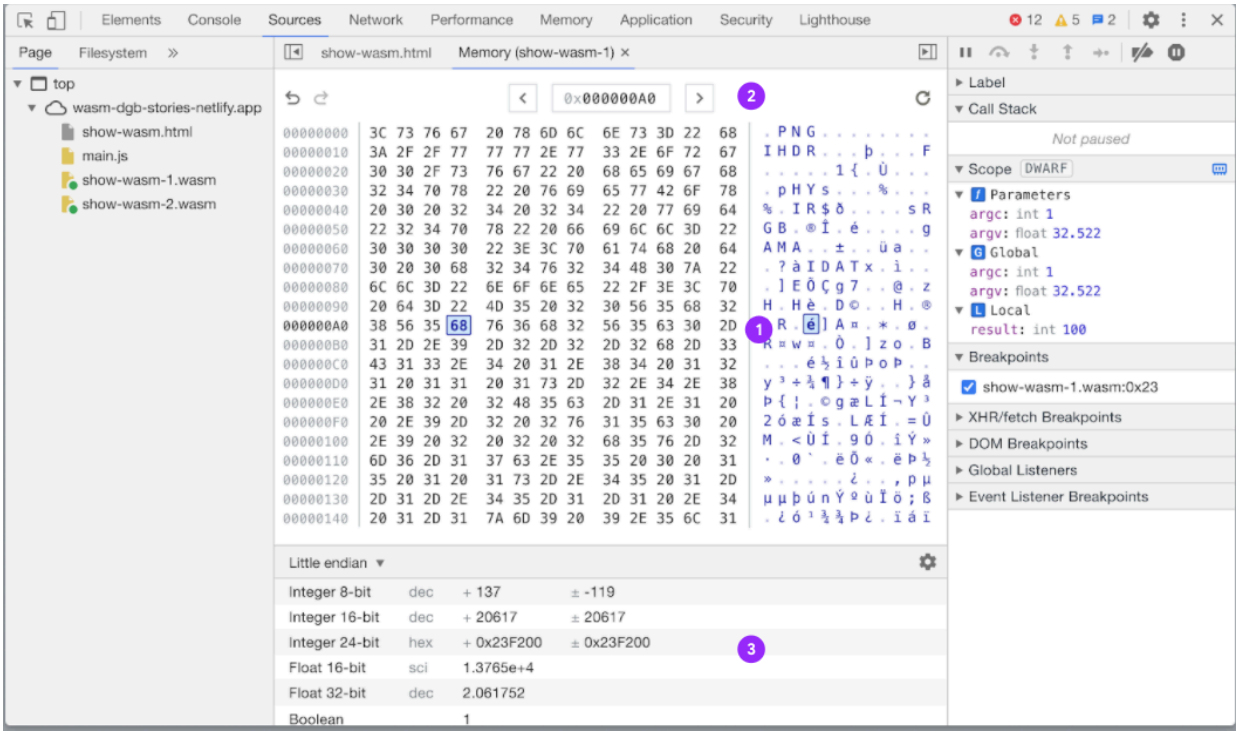


Figure 9: Showing the linear memory inspector within the Source View

Extending the SourcesView turns out to be difficult to accomplish from an architectural perspective. The SourcesView and the TabbedEditorContainer are written such that they expect to only deal with UISourceCode objects.

For now, we therefore chose to open the Linear Memory Inspector at the bottom as a drawer instead. This may also have the advantage of still being able to view the source code while inspecting the memory.

Accessibility

The linear memory inspector should support keyboard navigation and accessibility to screenreaders.

Testing plan

Unit tests and e2e tests

Follow up work

This is the design doc for the MVP, afterwards we still need to extend it to support more features.

Other use cases of the Linear Memory Inspector

Tracking bug: crbug.com/1144654

The linear memory inspector can also be reused in different parts of the Chrome DevTools:

- for inspecting binary responses in the Network tab (including the Web Socket view)

Extension of the Linear Memory Inspector

One thing to think about is to also support Blobs.