Architecture

Large parts of the Mbed OS API are generic between targets. This includes the GPIO layers, IP4/IPV6 networking stacks, and communication libraries like CoAP or LoRaWAN. Target-specific code, such as which registers to write to when toggling a GPIO pin, are implemented using the Mbed C HAL. The simulator uses the same approach, implementing a new target (`TARGET_SIMULATOR`) that implements the Mbed C HAL which passes events through to a JavaScript HAL. Then the UI subscribes to these events, and updates the simulator accordingly.

If you take a `DigitalOut` element, the flow goes like this:

```
User App -> DigitalOut.cpp (C++ API) -> gpio_api.c (C HAL) -> gpio.js (JS HAL) -> led.js (JS UI)
```

The C/C++ parts are in `mbed-simulator-hal`. This uses a fork of Mbed OS 5.10.2 (living here: [#mbed-os-5.10-simulator](https://github.com/janjongboom/mbed-os/tree/mbed-os-5.10-simulator)), where a new target was added (`TARGET_SIMULATOR`). In the long run this should be main-lined into core Mbed OS.

The JS HAL lives in `viewer/js-hal`, and dispatches events around between JS UI components and C++ HAL. It implements an event bus to let the UI subscribe to events from C++. For instance, see `js-hal/gpio.js` for GPIO and IRQ handling.

UI lives in `viewer/js-ui`, and handles UI events, and only communicates with JS HAL.

Device features need to be enabled in `targets/TARGET_SIMULATOR/device.h`.

Network proxy

The simulator implements two features which cannot be simulated by the browser alone:

* The `NetworkInterface` API, the generic IP networking interface for Mbed OS. This allows opening UDP and TCP sockets, which is not allowed in the browser.
* The LoRaWAN API. Messages from the LoRaWAN stack are sent to a network server using the Semtech UDP protocol, which is not allowed from the browser.

For these usecases a network proxy is implemented in `server/launch-server.js`. This network proxy allows opening raw UDP and TCP sockets, and relaying LoRaWAN messages. The proxy can be accessed over HTTP and through Web Sockets. See `viewer/js-hal/network.js` and `viewer/js-hal/lora.js` for the JS HAL, and `mbed-simulator-hal/easy-connect` `mbed-simulator-hal/lora-radio-drv` for the C++ HAL.

Debugging

Simulator applications can be debugged using your browser's debugger, because they contain source maps. To debug an application:

Chrome

1. Open the Developer Tools via **View > Developer > Developer Tools**.
1. Click **Sources**.
1. Under 'Network', select the name of the application (see the browser hash).
1. Now locate `main.cpp`.
   * On a pre-built demo, go to the `out` folder, select the name of the demo (e.g. `blinky`) and select `main.cpp`.
   * On a user-compiled app, go to the *orange* folder, go to the `out` folder, and select `main.cpp`.
1. Click in the gutter to add a breakpoint.
1. Click the **↻** icon in the simulator to restart the debug session.

Firefox

1. Open the Developer Tools via **Tools > Web Developer > Toggle Tools**.
1. Click **Debugger**.
1. Now locate `main.cpp`.
   * On a pre-built demo, go to the `out` folder, select the name of the demo (e.g. `demos/blinky`) and select `main.cpp`.
   * On a user-compiled app, go to the folder that starts with `/home/ubuntu`, go to the `out` folder, and select `user_XXX.cpp`.
1. Click in the gutter to add a breakpoint.
1. Click the **↻** icon in the simulator to restart the debug session.

File systems and block devices

Mbed OS allows you to load external flash through the [block device API](https://www.google.com/search?q=block+device+mbed&ie=utf-8&oe=utf-8&client=firefox-b-ab), then mount a file system to the block device. The simulator supports a simulated block device, but does not support mounting a file system to this device. There is a file system mounted automatically, but you need to manually persist the file system, see below for details.

Block device API

The simulator's block device persists its data to the browser's [local storage](https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage). Thus, data is persisted between page loads. The first argument to the constructor specifies the name of the cache. By changing the name you thus creata fresh block device without erasing the previous content. Because local storage only supports strings, the block device is serialized, making operations relatively inefficient (and taking up twice as much space).

Example

```cpp
#include "SimulatorBlockDevice.h"

// arguments: name of the block device, total storage, page size
SimulatorBlockDevice bd("myblockdevice", 128 * 512, 512);

// read first page, note that all operations must be page aligned
char buffer[512];
bd.read(buffer, 0, 512);    // make sure to check the return value

// write something back
buffer[1] = 0xff;
bd.program(buffer, 0, 512);
```

File system

A file system is automatically mounted under `/` and you can use standard POSIX calls (`fopen`, `fread`, etc.) without any configuration. However, this file system is not persisted throughout page refreshes. A persistent file system is mounted under `/IDBFS` but you need to manually persist it.

To persist the file system, call:

```cpp
```

```
#include "emscripten.h"

EM_ASM({
    // alternatively: call this function from JavaScript
    window.MbedJSHal.syncIdbfs();
});
```

Clearing the persistent file system

To remove all files from the persistent file system, open your browsers console and run:

```js
window.MbedJSHal.clearIdbfs();
```

Prepopulating the file system

You can populate the file system at compile time with data from your computer. This can be done through the CLI or via the [simconfig](simconfig.md) file.

CLI

```
# the part after the @ is the mount location (in this case /fs)

$ mbed-simulator -i . --preload-file folder-to-load/@/fs
```

simconfig.md

```json
{
    "compiler-args": [
        "--preload-file", "sdcard@/fs"
    ]
}
```

Peripherals

Loading peripherals

The simulator comes with peripherals (such as LCD displays, LEDs, etc.). To load these for your project add a `peripherals` section to your [simconfig](simconfig.md) file. You can pass in the pin names via `"PinNames.p5"`, these will be automatically resolved on startup. Note that you can change the peripheral config at runtime by clicking **__Add component__**. This is then cached by the browser. To clean the cache run `sessionStorage.removeItem('model-dirty')` from your browsers console.

Adding new peripherals

For an example of how a peripheral looks like, see `mbed-simulator-hal/peripherals/Sht31`.

A peripheral consists of a C++ implementation, a JS HAL file and a JS UI file. Peripherals in the `mbed-simulator-hal/peripherals` folder are automatically picked up at compile time. In your own project you can add a section to [simconfig](simconfig.md):

```json
    "components": {
        "jshal": [
            "./ui/js-hal.js"
        ],
        "jsui": [
            "./ui/js-ui.js"
        ]
    }
```

The C++ component is automatically picked up. The JS files are loaded when you run the project. Note that all `jshal` and `jsui` files will be copied to the BUILD directory, so to change things on the fly, edit those files (or create a symlink).

You can communicate between C++ and JS through `ES_ASM` macros (C++ -> JS), or via `ccall` (JS -> C++). See the components that ship with the simulator for examples.

Configuration and compiler options

Simconfig

You can specify simulator options via a `simconfig.json` object, this is useful because you can check it in. In here you can specify compiler options and ignore paths. Just create the file in your project folder according to the following structure:

```json
{
    "compiler-args": [
        "-std=c++11",
        "--preload-file", "sdcard@/fs"
    ],
    "emterpretify": true,
    "ignore": [
        "./BSP_DISCO_F413ZH",
        "./F413ZH_SD_BlockDevice"
    ],
    "peripherals": [
        { "component": "ST7789H2", "args": {} }
    ],
    "components": {
        "jshal": [
            "./ui/js-hal.js"
        ],
        "jsui": [
            "./ui/js-ui.js"
        ]
    },
    "disableTlsNullEntropy": true
}
```

It will automatically be picked up by the simulator CLI.

Compiler options

__C++11__

To use C++11 (or a different version), pass in `-c "-std=c++11"`.

__Emterpretify__

If you see that compilation hangs this might be due to a bug in asyncify. To switch to Emterpretify for async operations, pass in `--emterpretify`. This is f.e. used for uTensor. Note thet emterpretify is not fully supported.

Null entropy

By default no entropy sources are defined. If you see an error by the build process about this (e.g. `#error "MBEDTLS_TEST_NULL_ENTROPY defined, but not all prerequisites"`), you can add `"disableTlsNullEntropy": true` to your simconfig.

mbed_app.json

Configuration options and macros are automatically picked up from your mbed_app.json file. To specify options that are only loaded for the simulator, use the `target_overrides` section:

```json
{
    "target_overrides": {
        "SIMULATOR": {
            "some-config-option": 12
        }
    }
}
```