Quasar Documentation

Release v3.0.0

r52

Aug 05, 2023

MAIN

Quasar is a cross-platform desktop application that displays web-based widgets on your desktop. Quasar leverages the Chromium engine to serve desktop widgets in a platform agnostic manner. Widgets can be as simple as a single webpage with a couple of lines of HTML, a complex fully dynamic web app, or even a WebGL app.

Quasar provides a WebSocket-based Data Server that is extensible by custom extensions. The Data Server is capable of processing and sending data to client widgets that would otherwise not be available in a purely web-based context, for example, your PC's resource information such as CPU and memory usage, or your Spotify client's now-playing information.

See *Creating an Extension Quickstart* and *Extension API Reference* for more information on how to create a Data Server extension, and *Creating a Widget* for building widgets.

Quasar is licensed under GPL-3.0. All sample widgets are licensed under the MIT license.

CHAPTER

ONE

SYSTEM REQUIREMENTS

An OS and computer capable of running Chrome, preferably with Hardware Acceleration capabilities. Only 64-bit OSes are supported. On Windows, only Windows 10 and above are supported. On Linux, only X11 desktop environments with system tray support are supported.

While Quasar is reasonably fast and lightweight, do not expect Quasar to be power efficient, especially when running heavier widgets like the WebGL visualizer.

1.1 Note Regarding Wayland

Quasar does not work properly on Wayland compositors due to Wayland not supporting functions like global cursor position or explicitly setting window positions for top-level windows [1] [2], which completely defeats Quasar's core functionality. Until a workaround or solution is added to Qt itself, Quasar cannot support Wayland.

1.2 Qt on Linux

Quasar has been built and tested on the latest pre-built binaries supplied by Qt, which at the time of writing is Qt 6.5.1 built against OpenSSL 1.1.1. The Qt version offered by Ubuntu 22.04's package repository is Qt 6.2.4 built against OpenSSL 3.0. As of Ubuntu 22.04, OpenSSL 3 is the default and version 1.1.1 is no longer offered in the package repository. Building Quasar on versions of Qt older than 6.4 may work but is not supported. Ensure that the version of the OpenSSL libraries installed (i.e `libcrypto.so` and `libssl.so`) matches the version your Qt installation is built against or SSL functionality will fail.

CHAPTER

TWO

GETTING STARTED

Download the latest portable release here, for Windows x64.

Simply extract Quasar and run the application.

The Quasar icon will then show up in your desktop's notification bar. Right-click the icon, load your desired widgets, and enjoy! See *Basic Usage* for more details.

CHAPTER

THREE

CREATING WIDGETS/EXTENSIONS

See *Creating an Extension Quickstart* and *Extension API Reference* for more information on how to create a Data Server extension, and *Creating a Widget* for building widgets.

CHAPTER

FOUR

BUILDING REQUIREMENTS

Source code is available on GitHub.

- CMake 3.23 or later
- Qt 6.4 or later, with at least the following additional libraries:
  - WebEngine (qtwebengine)
  - Positioning (qtpositioning)
  - WebChannel (qtwebchannel)
  - Network Authorization (qtnetworkauth)
  - Serial Port (qtserialport)
- The `Qt6_DIR` environment variable defined for your Qt installation
  - Windows example: `C:\Qt\6.5.1\msvc2019_64`
  - Linux example: `$HOME/Qt/6.5.1/gcc_64/`
  - On Linux, additional dependencies may be needed for Qt such as the packages `libgl1-mesa-dev libglvnd-dev`
- Visual Studio 2022 or later is required
- Clang (or the Clang MSVC toolkit in Visual Studio) is required if you wish to build the `win_audio_viz` sample extension
- gcc/g++ 11 or later, or Clang 16 or later
  - Tested on Ubuntu 22.04 using both g++ 11 and 12
  - Clang 15 and earlier fails to compile gcc's implementation of the C++20 ranges library, which is used in Quasar
- Clang is required if you wish to build the `pulse_viz` sample extension
- vcpkg dependencies, for example including but not limited to the following Debian-based packages:
  - `build-essential tar curl zip unzip pkg-config`

Quasar is written in cross-platform C++ and should build on Mac with minimal changes. However, it is currently untested and unsupported.

```
git clone --recurse-submodules https://github.com/r52/quasar.git
cd quasar
```

4.1 Windows

The following example configures the project to build using the `clang-cl` toolkit with Visual Studio Community 2022 (plus the Clang MSVC toolkit) installed:

```
cmake --no-warn-unused-cli -DCMAKE_EXPORT_COMPILE_COMMANDS:BOOL=TRUE "-DCMAKE_C_
↪COMPILER:FILEPATH=C:\Program Files\Microsoft Visual Studio\2022\Community\VC\Tools\
↪Llvm\x64\bin\clang-cl.exe" "-DCMAKE_CXX_COMPILER:FILEPATH=C:\Program Files\Microsoft␣
↪Visual Studio\2022\Community\VC\Tools\Llvm\x64\bin\clang-cl.exe" -S./ -B./build -G
↪"Visual Studio 17 2022" -T ClangCL,host=x64 -A x64
```

4.2 Linux

The following example configures the project to build using `g++-12`, assuming g++ 12 is installed, and configures Quasar to be installed to `$HOME/.local/quasar/`:

```
export CC=gcc-12
export CXX=g++-12
cmake --no-warn-unused-cli -DCMAKE_EXPORT_COMPILE_COMMANDS:BOOL=TRUE --install-prefix
↪$HOME/.local/ -S./ -B./build -G "Unix Makefiles"
```

`-G Ninja` can also be used provided that Ninja is installed.

4.2.1 Building the Project

```
cmake --build ./build --config Release --
```

4.2.2 Installing from Build (optional)

```
cmake --install ./build
```

CHAPTER

FIVE

RESOURCES

5.1 Basic Usage

5.1.1 Loading a Widget

Right-click the Quasar icon in your notification bar, and click **Load**. Browse to the folder containing the widget, and load the `.json` Widget Definition file. If the widget folder does not contain a Widget Definition `.json` file, then it is not compatible with Quasar.

The loaded widget will then appear on your desktop.

5.1.2 Moving a Widget

Widgets can be moved simply by dragging them around.

5.1.3 Closing a Widget

Right-click the widget on your desktop, and click **Close**. If the widget is no longer visible for whatever reason, the widget menu can be accessed by right-clicking the Quasar icon in your notification bar, under the **Widgets** menu.

5.1.4 Widget Menu

The widget menu can be accessed by either right-clicking the widget itself, or by right-clicking the Quasar icon in your notification bar, and under the **Widgets** menu.

Widget Name
: Opens the folder of the widget's location

Reload
: Refreshes the widget

Set Position
: Sets the position of the widget to a specific X and Y position

Reset Position
: Resets the position of the widget to a visible position on your monitor

Custom Size
: Resizes the widget to a custom size. **Warning: Setting a custom size may break the widget's styling!**

Always on Top
: Forces the widget to be on top of all other windows

Fixed Position
Fixes the widget at its current position (and disables dragging)

Clickable
By default, widgets cannot be interacted with (besides dragging). If this option is enabled, then widget elements can be clicked on. This option is used for widgets with clickable elements such as App Launcher widgets.

Close
Closes the widget.

5.2 Installing Extensions

Extensions for the Data Server, which comes in the form of a `.dll` or `.so` file, should typically be installed to Quasar's data folder. On Windows, this is at `%AppData%\quasar\extensions\`. You can access this folder by clicking on **Open Data Folder** in Quasar's tray icon menu.

Extensions can also be installed by placing the file in the `extensions` directory where the Quasar executable is installed or extracted. For installations on Windows, this is typically `C:\Users\<USERNAME>\AppData\Local\Quasar\Release\extensions\`.

5.3 Settings

Right-click the Quasar icon in your notification bar, and click **Settings** to access the settings menu. Here, users can change advanced options for Quasar, as well as any custom options provided by extensions.

5.3.1 General Settings

WebSocket Server port
The port the WebSocket Data Server runs on. *(default: 13337)*

Allow only Quasar widgets to connect to the WebSocket server?
Enable to only allow Quasar loaded widgets access to the WebSocket server *(default: off)*

Log to file?
Sets whether log messages are written to a file.

Log Level
Severity of log messages that are logged. *(default: Warn)*

cookies.txt
Path to a Netscape formatted cookies.txt. These cookies will be loaded by all widgets.

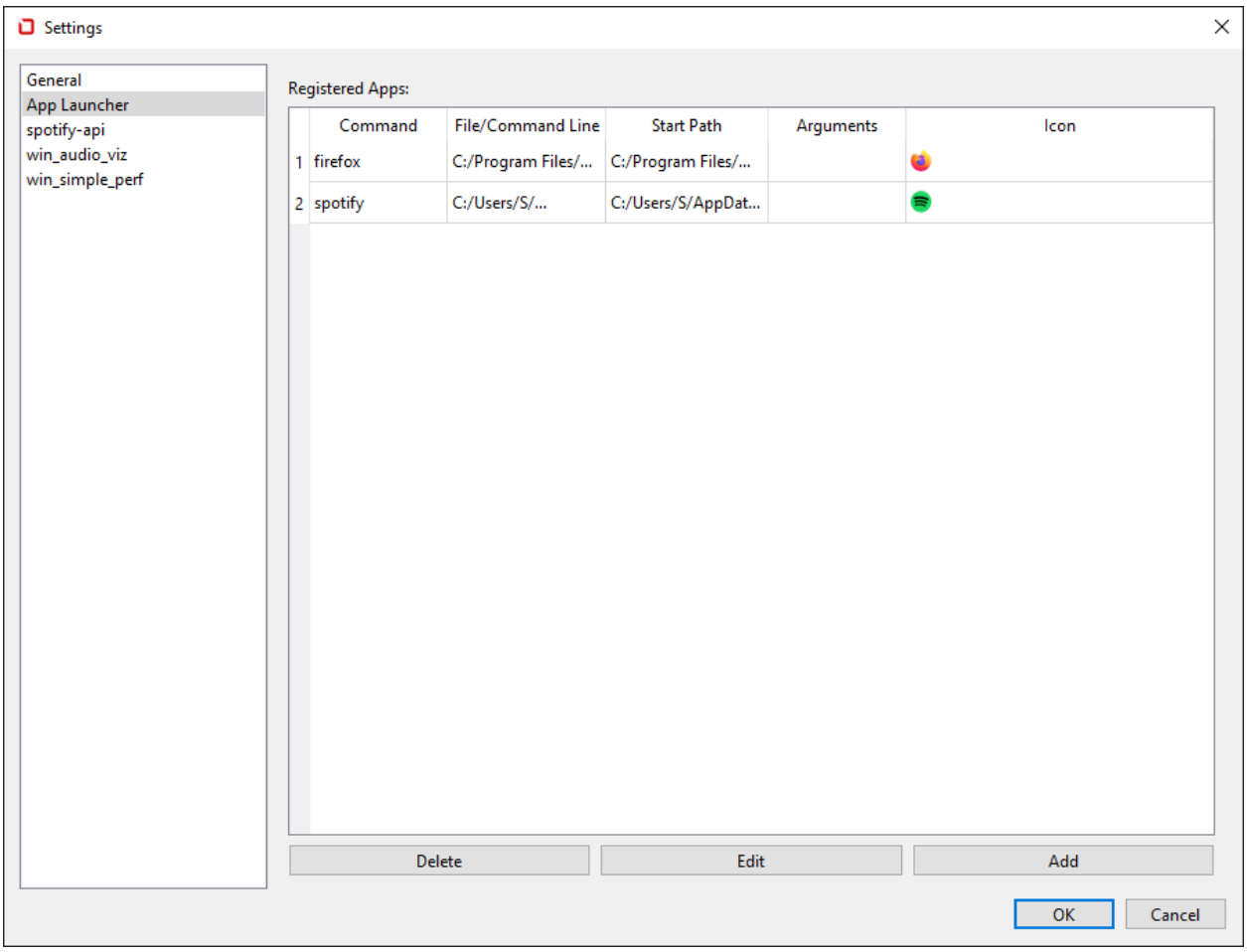
5.3.2 App Launcher Settings

Commands available to App Launcher widgets can be configured under the **Launcher** page. See *Setting up the App Launcher* for more information.

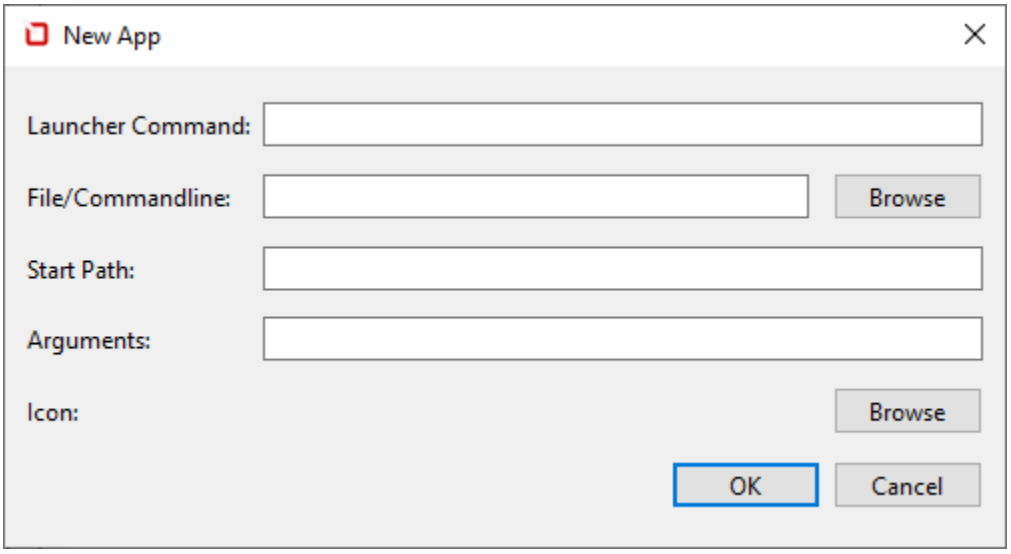
5.4 Setting up the App Launcher

App Launcher widgets require additional setup in order to function properly. Specific App Launcher commands needs to be created and configured so that apps can be launched from Quasar. App Launcher commands can be configured in the **Settings** menu under the **App Launcher** page.

Example:



In the image above, Quasar is configured to launch Firefox and Spotify using the App Launcher commands `firefox`, and `spotify` respectively. A properly configured App Launcher widget will receive the list of configured commands and will able to send these commands to Quasar in order to launch the configured applications.



Furthermore, you can also specify your own custom icon for each of these commands. In the previous example, icons have been added for all three commands. The sample widget `sample_app_launcher` that comes installed with Quasar is preconfigured to use custom icons. Of course, what the commands themselves execute can be completely arbitrary and is up to user discretion, so be sure to set them up properly!

5.5 Creating a Widget

5.5.1 The Basics

Note: See *Widget Definition Reference* for the complete Widget Definition file reference.

Widgets can be anything that can be loaded in Chromium. This includes, but is not limited to, webpages and webapps, files, URLs, media files such as videos or music etc. At its most basic form, a widget is a simple HTML webpage. To create a Quasar Widget, we must first start with a Widget Definition file.

A Widget Definition file is a JSON file that contains at least the following parameters:

name
: Name of the widget.

width
: Width of the widget.

height
: Height of the widget.

startFile
: Entry point for the widget. This can be a local file or a URL.

transparentBg
: Whether the widget background is transparent.

The following example is the Widget Definition file of the sample widget datetime:

```
{
    "name": "DateTime",
```

(continues on next page)

(continued from previous page)

```
    "width": 470,
    "height": 220,
    "startFile": "index.html",
    "transparentBg": true
}
```

The `height` and `width` parameters determine the size of the viewport of the widget window, while the `startFile` parameter provides the entry point for the widget. In this case it refers to the file `index.html`. The contents of index.html in the case of the sample widget `datetime` is a simple webpage that queries the current date and time and displays it (refer to its source code for more information).

As mentioned above, the contents of a widget can be as simple as a text page, or as complex as a WebGL app. Quasar is only limited by the contents that the Chromium engine is capable of displaying. Unfortunately, a tutorial for frontend development is beyond the scope of this guide, but the web is full of resources that will help design your widget, no matter how complex it may be.

5.5.2 Communicating with the Data Server

Note: See *Widget Client Protocol* for the complete Widget Client Protocol reference.

Note: The following code samples are adapted from the sample widget `simple_perf`. Refer to its source code for more information.

In order for widgets to be able to communicate with the Quasar Data Server and fetch data from installed Data Server extensions, its definition file must have the parameter `dataserver` defined with its value set to `true`. This tells Quasar to load the connection and authentication scripts into the widget (see *Widget Definition Reference*).

Once this is done, we first need to open up a WebSocket connection with Quasar.

Quasar widgets can simply execute the following JavaScript:

```
let websocket = quasar_create_websocket();
```

`quasar_create_websocket()` is a globally defined function only available to widgets loaded in Quasar when the `dataserver` parameter is set to `true`. This function creates a WebSocket object connecting to Quasar's Data Server.

However, **external clients** must manually establish the connection to Quasar:

```
let websocket = new WebSocket("ws://127.0.0.1:<port>");
```

Where <port> is the port that the Data Server is running on, as set in *Settings*.

Once the connection is established, we then need to authenticate with the Data Server to establish our widget's identity.

Similar to the above, **Quasar widgets** can achieve this simply by calling the (similarity defined) global function `quasar_authenticate()` in the WebSocket's `onopen` handler, supplying our `websocket` connection object as an argument:

```
websocket.onopen = function(evt) {
    quasar_authenticate(websocket);
};
```

If the `Allow only Quasar widgets to connect to the WebSocket server?` setting is enabled, **external clients** will be unable to connect.

Once our widget is authenticated, we can start fetching data from a Data Source by placing a call to a data request function in the handler. For example:

```
websocket.onopen = function(evt) {
    quasar_authenticate(websocket);
    setInterval(poll, 5000);
};
```

Where the function `poll()` can be something like:

```
function poll() {
    let msg = {
        method: "query",
        params: {
            topics: ["win_simple_perf/sysinfo_polled"]
        }
    }

    websocket.send(JSON.stringify(msg));
}
```

The above example polls the Data Source `sysinfo_polled` provided by the sample extension win_simple_perf every 5000ms.

How that we have configured the Data Sources we want to receive data from, we must now setup our data processing for the data we will receive. We start by implementing another handler on the WebSocket connection. For example:

```
websocket.onmessage = function(evt) {
    parseMsg(evt.data);
};
```

We can then implement a function `parseMsg()` to process the incoming data. Refer to the *Widget Client Protocol* for the full message format:

```
function parseMsg(msg) {
    const data = JSON.parse(msg);

    if ("win_simple_perf/sysinfo_polled" in data) {
        const vals = data["win_simple_perf/sysinfo_polled"]
        setData(document.getElementById("cpu"), vals["cpu"]);
        setData(
            document.getElementById("ram"),
            Math.round((vals["ram"]["used"] / vals["ram"]["total"]) * 100),
        );
    }
}
```

We start by parsing the JSON message, then examining the object's fields to ensure that we have received what we wanted, namely the `data["win_simple_perf/sysinfo_polled"]` field, which is what we requested in the previous code examples. If everything matches, we finally process the payload. The `cpu` field in the data outputs a single integer containing the current CPU load percentage on your desktop, while the `ram` field contains the `total` and `used` RAM in bytes. We convert these numbers to a percentage if they are not already one, and output them to the HTML elements defined in the widget's `index.html` with the IDs `cpu` and `ram` respectively.

Putting everything together, your widget's script may end up looking something like this:

```
let websocket = null;

function poll() {
    let msg = {
        method: "query",
        params: {
            topics: ["win_simple_perf/sysinfo_polled"]
        }
    }

    websocket.send(JSON.stringify(msg));
}

function setData(elm, value) {
    if (elm != null) {
        elm.setAttribute("aria-valuenow", value);
        elm.textContent = value + "%";
        elm.style.width = value + "%";
        elm.classList.remove("bg-success", "bg-info", "bg-warning", "bg-danger");

        if (value >= 80) {
            elm.classList.add("bg-danger");
        } else if (value >= 60) {
            elm.classList.add("bg-warning");
        } else {
            elm.classList.add("bg-success");
        }
    }
}

function parseMsg(msg) {
    const data = JSON.parse(msg);

    if ("win_simple_perf/sysinfo_polled" in data) {
        const vals = data["win_simple_perf/sysinfo_polled"]
        setData(document.getElementById("cpu"), vals["cpu"]);
        setData(
            document.getElementById("ram"),
            Math.round((vals["ram"]["used"] / vals["ram"]["total"]) * 100),
        );
    }
}

function ready(fn) {
    if (document.readyState !== "loading") {
        fn();
    } else {
        document.addEventListener("DOMContentLoaded", fn);
    }
}
```

(continues on next page)

(continued from previous page)

```
ready(function() {
    try {
        if (websocket && websocket.readyState == 1)
            websocket.close();
        websocket = quasar_create_websocket();
        websocket.onopen = function(evt) {
            quasar_authenticate(websocket);
            setInterval(poll, 5000);
        };
        websocket.onmessage = function(evt) {
            parseMsg(evt.data);
        };
        websocket.onerror = function(evt) {
            console.log('ERROR: ' + evt.data);
        };
    } catch (exception) {
        console.log('Exception: ' + exception);
    }
});
```

5.6 Creating an Extension Quickstart

Note: See *Extension API Reference*, *Widget Client Protocol*, and the sample extension win_simple_perf for more details.

5.6.1 Getting Started

Quasar implements a WebSocket-based Data Server that facilitates the communication of various data that isn't available in a web-only context to widgets. The Data Server can be extended by **extensions** that provide additional Data Sources that can be used by widgets. The *Extension API* is provided as a pure-C interface, so extensions can be written in languages that support implementing C interfaces and produces a native library. This guide assumes that the language of choice is C++.

To begin, your extensions must include *extension_api.h* (which includes *extension_types.h*) as well as *extension_support.h*.

At minimum, an extension needs to implement 3 functions in additional to *quasar_ext_load()* and *quasar_ext_destroy()*.

These are:

- `bool init(quasar_ext_handle handle)`
- `bool shutdown(quasar_ext_handle handle)`
- `bool get_data(size_t uid, quasar_data_handle dataHandle, char* args)`

Within the *quasar_ext_info_t* structure.

For each Data Dource provided by the extension, a *quasar_data_source_t* entry needs to be created that contains the Data Source's identifier and refresh rate in microseconds or polling style. *quasar_data_source_t::validtime* specifies the amount of time in milliseconds that the data is cached and remains valid for, for sources using the Client Polling style. *quasar_data_source_t::uid* should be initialized to `0`.

The entries should be propagated in *quasar_ext_info_t::numDataSources* and *quasar_ext_info_t::dataSources*.

The rest of the static data fields in *quasar_ext_info_t::fields* such as *quasar_ext_info_fields_t::name*, *quasar_ext_info_fields_t::fullname*, and *quasar_ext_info_fields_t::version* should be filled in with the extension's basic information.

Example

Adapted from the sample extension win_simple_perf:

```
quasar_data_source_t sources[] = {
    {       "sysinfo",               5000000,    0, 0},
    {"sysinfo_polled", QUASAR_POLLING_CLIENT, 1000, 0}
};

quasar_ext_info_fields_t fields =
    {
        "win_simple_perf",                                  // char name[16]
        "Simple Performance Query",                         // char fullname[64]
        "3.0",                                              // char version[64]
        "r52",                                              // char author[64]
        "Provides basic PC performance metrics",            // char description[256]
        "https://github.com/r52/quasar"                     // char url[256]
    };

quasar_ext_info_t info =
    {
        QUASAR_API_VERSION,                                 // int api_version,␣
```

(continues on next page)

(continued from previous page)

```
↪should always be QUASAR_API_VERSION
        &fields,                                        // quasar_ext_info_
↪fields_t* fields. Must be initialized

        std::size(sources),                             // size_t numDataSources
        sources,                                        // quasar_data_source_t*␣
↪dataSources

        simple_perf_init,                               // bool init(quasar_ext_
↪handle handle)
        simple_perf_shutdown,                           // bool shutdown(quasar_
↪ext_handle handle)
        simple_perf_get_data,                           // bool get_data(size_t␣
↪uid, quasar_data_handle dataHandle, char* args)
        nullptr,                                        // quasar_settings_t*␣
↪create_settings(quasar_ext_handle handle)
        nullptr                                         // void update(quasar_
↪settings_t* settings)
    };
```

In this example, 2 Data Sources are defined, `sysinfo` and `sysinfo_polled`, where `sysinfo` uses the subscription model, while `sysinfo_polled` uses the Client Polling model. The functions `simple_perf_init()`, `simple_perf_shutdown()`, and `simple_perf_get_data()` are the implementations of `init()`, `shutdown()`, and `get_data()` respectively. Note that `create_settings()` and `update()` are not implemented by this extension. These functions are optional, and only needs to be implemented if the extension provides custom settings. See *Custom Settings* for more information.

quasar_ext_load()

This function should return a pointer to a populated *quasar_ext_info_t* structure.

Following previous example:

```
quasar_ext_info_t* quasar_ext_load(void)
{
    return &info;
}
```

Since the `quasar_ext_info_t info` structure is defined statically in the previous example, it is suffice for `quasar_ext_load()` to simply return the pointer to it.

quasar_ext_destroy()

This function should deallocate anything that was allocated for the *quasar_ext_info_t* structure.

Following previous examples:

```
void quasar_ext_destroy(quasar_ext_info_t* info)
{
    // does nothing; info is on stack
    return;
}
```

Since both the `quasar_data_source_t sources` as well as the `quasar_ext_info_t info` structure and all of its contents are defined statically in the previous examples, we do not need to deallocate anything for the destruction of the *quasar_ext_info_t* structure. Therefore, the function does nothing.

init()

If the extension was loaded successfully, each Data Source entry's *quasar_data_source_t::uid* is filled with a unique identifier. These are used in the `get_data()` function call to identify the Data Source being requested. It is up to the extension to remember these during `init()` as they will be referred to by future `get_data()` calls from Quasar.

This function should also allocate or initialize any other resources needed, as well as remember the extension handle if necessary.

```
bool simple_perf_init(quasar_ext_handle handle)
{
    extHandle = handle;

    // Process uid entries.
    if (sources[0].uid == 0)
    {
        // "sysinfo" Data Source didn't get a uid
        return false;
    }

    if (sources[1].uid == 0)
    {
        // "sysinfo_polled" Data Source didn't get a uid
        return false;
    }

    return true;
}
```

shutdown()

This function should deallocate and clean up any resources allocated in `init()`, including waiting on any threads spawned. Since we have no allocations in our sample `init()` function, our `shutdown()` can simply return.

```
bool simple_perf_shutdown(quasar_ext_handle handle)
{
    return true;
}
```

get_data()

This function is responsible for retrieving the data requested by the `uid` argument and populating it into the `quasar_data_handle` handle using functions from *extension_support.h*.

Note: This function needs to be both re-entrant and thread-safe!

```
bool simple_perf_get_data(size_t uid, quasar_data_handle hData, char* args)
{
    if (srcUid != sources[0].uid && srcUid != sources[1].uid)
    {
        warn("Unknown source {}", srcUid);
        return false;
    }

    // CPU data
    double cpu = GetCPULoad() * 100.0;

    // https://stackoverflow.com/questions/63166/how-to-determine-cpu-and-memory-
↪consumption-from-inside-a-process
    // RAM data
    MEMORYSTATUSEX memInfo;
    memInfo.dwLength = sizeof(MEMORYSTATUSEX);
    GlobalMemoryStatusEx(&memInfo);
    DWORDLONG totalPhysMem = memInfo.ullTotalPhys;
    DWORDLONG physMemUsed  = memInfo.ullTotalPhys - memInfo.ullAvailPhys;

    auto      res          = fmt::format("{{\"cpu\":{},\"ram\":{{\"total\":{},\"used\":{}
↪}}}}", (int) cpu, totalPhysMem, physMemUsed);

    quasar_set_data_json(hData, res.c_str());

    return true;
}
```

See *extension_support.h* and *extension_support.hpp* for all supported data types.

5.6.2 Data Models

Quasar supports three different types of data models for Data Sources.

By default, Data Sources in Quasar operate on a timer-based subscription model.

This can be changed by initializing *quasar_data_source_t::rate* of a Data Source entry to different values. A positive value means the default timer-based subscription. A value of `QUASAR_POLLING_CLIENT` means the client widget is responsible for polling the extension for new data. A value of `QUASAR_POLLING_SIGNALED` means the extension will signal when new data becomes available (i.e. from a thread) and automatically send the new data to all subscribed widgets.

See *Widget Client Protocol* for details on client message formats.

Timer-based Subscription

Enabled by initializing *quasar_data_source_t::rate* of a Data Source entry to a positive value.

Multiple client widgets may subscribe to a single data source, which is polled for new data every *quasar_data_source_t::rate* microseconds. This new data is then propagated to every subscribed widget.

Signal-based Subscription

Enabled by initializing *quasar_data_source_t::rate* to `QUASAR_POLLING_SIGNALED`.

This model supports Data Sources which require inconsistent timing, as well as Data Sources which require background processing, such as a producer-consumer thread.

To use this model, utilize the functions *quasar_signal_data_ready()* and *quasar_signal_wait_processed()* in *extension_support.h*.

For example:

```
quasar_data_source_t sources[2] =
    {
      { "some_thread_source", QUASAR_POLLING_SIGNALED, 0, 0 },
      { "some_timer_source", 5000000, 0, 0 }
    };

quasar_ext_handle extHandle = nullptr;
std::atomic_bool running = true;
std::thread workThd;

void workerThread()
{
    while (running)
    {
        // do the work
        ...

        // signal that data is ready
        quasar_signal_data_ready(extHandle, "some_thread_source");

        // call this function if the thread needs to wait for the data to be consumed
        // before processing new data
        quasar_signal_wait_processed(extHandle, "some_thread_source");
    }
}

bool init_func(quasar_ext_handle handle)
{
    extHandle = handle;

    // start the worker thread
    workThd = std::thread{workerThread};

    return true;
}
```

(continues on next page)

(continued from previous page)

```
bool shutdown_func(quasar_ext_handle handle)
{
    running = false;

    // join the worker thread
    workThd.join();

    return true;
}
```

Client Polling

Enabled by initializing *quasar_data_source_t::rate* to QUASAR_POLLING_CLIENT.

This data model transfers the responsibility of polling for new data to the client widget. The data source does not accept subscribers.

Example:

```
quasar_data_source_t sources[2] =
    {
      { "some_polled_source", QUASAR_POLLING_CLIENT, 1000, 0 },
      { "some_timer_source", 5000000, 0, 0 }
    };
```

From the client:

```
function poll() {
    const reg = {
        method: "query",
        params: {
            topics: ["some_extension/some_polled_source"]
        }
    };

    websocket.send(JSON.stringify(reg));
}
```

In this example, *quasar_data_source_t::validtime* is configured with a value of 1000ms. This is the time that the data returned by some_polled_source is cached for after retrieval. Any polls to some_polled_source within the time duration will return the cached data.

This model also allows the extension to signal data ready using *quasar_signal_data_ready()* for an asynchronous poll request/response timing.

The sample code in the above sections are based on this model.

5.6.3 Custom Settings

By default, users can enable or disable a Data Source as well as change its refresh rate from the *Settings* dialog.

However, a extension can provide further custom settings by utilizing the *extension_support.h* API and implementing the `create_settings()` and `update()` functions in *quasar_ext_info_t*. These custom settings will appear under the Settings dialog.

Sample code:

```
quasar_settings_t* create_custom_settings(quasar_ext_handle handle)
{
    quasar_settings_t* settings = quasar_create_settings(handle);
    quasar_add_bool_setting(handle, settings, "s_levelenabled", "Process Level", true);
    quasar_add_int_setting(handle, settings, "s_level", "Level", 1, 30, 1, 1);

    return settings;
}

void custom_settings_update(quasar_settings_t* settings)
{
    g_levelenabled = quasar_get_bool_setting(extHandle, settings, "s_levelenabled");
    g_level = quasar_get_int_setting(extHandle, settings, "s_level");
}
```

5.7 win_simple_perf

A sample extension for Quasar on Windows that provides basic performance metrics.

5.7.1 Data Sources

win_simple_perf provides two Data Sources: the subscription based `sysinfo`, and client polled `sysinfo_polled`. Both provide the same information.

`sysinfo` refreshes at a default rate of 5000ms.

cpu field

Contains a single integer that is the current total CPU load percentage.

ram field

Contains the total (`total`) and currently used (`used`) memory as a JSON object, in bytes.

Sample Output

```
{
    "win_simple_perf/sysinfo": {
        "cpu": 15,
        "ram": {
            "total": 34324512768,
            "used": 10252300288
        }
    }
}
```

5.8 win_audio_viz

A sample extension for Quasar on Windows that captures audio device output and provides various audio data.

This extension is adapted from the Rainmeter AudioLevel plugin for Quasar usage.

Rainmeter is licensed under GPLv2.

5.8.1 Usage

win_audio_viz provides all functionality available to the Rainmeter AudioLevel plugin, except not at a per channel/band level. Instead, win_audio_viz will typically provide the entire spectrum of values from a source as an array.

See the Rainmeter AudioLevel documentation for more details.

Data Sources

win_audio_viz provides all options available in Rainmeter AudioLevel's `Type` setting as a Data Source. These are:

- `rms` : The current RMS level (0.0 to 1.0) for all channels. Subscription, default 16.67ms refresh.
- `peak` : The current Peak level (0.0 to 1.0) for all channels. Subscription, default 16.67ms refresh.
- `fft` : The current FFT level (0.0 to 1.0) for all FFT bins. Subscription, default 16.67ms refresh.
- `fftfreq` : The frequency in Hz for each FFT bin. Client polled.
- `band` : The current FFT level (0.0 to 1.0) for all bands. Subscription, default 16.67ms refresh.
- `bandfreq` : The frequency in Hz for all bands. Client polled.
- `format` : A string describing the audio format of the device connected to. Client polled.
- `dev_status` : Status (bool - true/false) of the device connected to. Client polled.
- `dev_name` : A string with the name of the device connected to. Client polled.
- `dev_id` : A string with the Windows ID of the device connected to. Client polled.
- `dev_list` : A string with a list of all available device IDs. Client polled.

Sample Output


```
{
    "win_audio_viz/band": [
        0.123123,
        0.237892,
        0.83792,
        0.37855,
        0.382793,
        0.38927,
        0.72893,
        0.83792,
        0.8327492,
        0.3827938,
        0.84641651,
        0.62826286,
        0.6654456,
        0.4864866,
        0.1691962,
        0.8641233
    ]
}
```


5.8.2 Settings

win_audio_viz provides the same set of settings available to the Rainmeter AudioLevel plugin, with the exception of parameters which define specific data retrieval settings such as `Channel`, `FFTIdx`, and `BandIdx`. The parameter `Port` is not supported in win_audio_viz.

See the Rainmeter AudioLevel documentation for more details.

5.9 pulse_viz

A sample extension for Quasar on Linux that captures audio output from a PulseAudio server and provides various audio data.

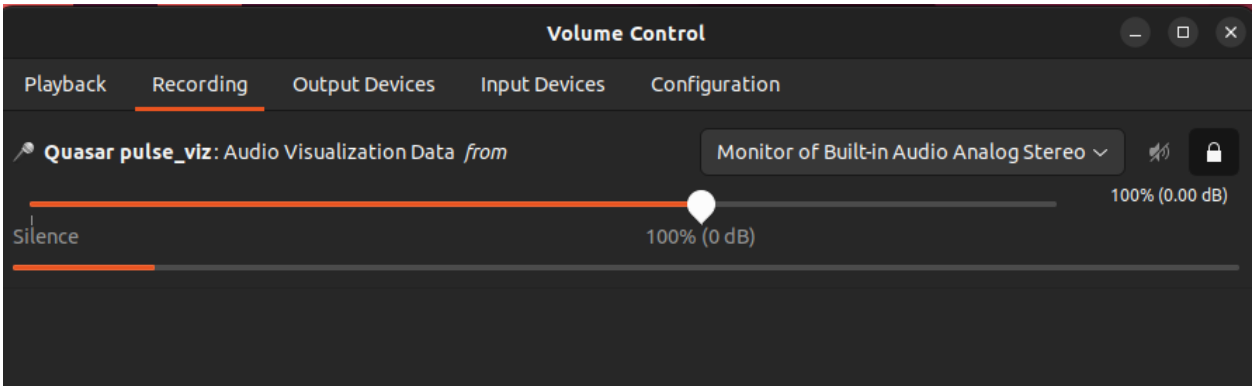
This extension is adapted from the `win_audio_viz` sample extension, which itself is adapted from the Rainmeter AudioLevel plugin for Quasar usage.

Rainmeter is licensed under GPLv2.

5.9.1 Usage

pulse_viz provides only a subset of the functionality available in `win_audio_viz` for simplicity's sake.

See the win_audio_viz extension for more details.



Use `pavucontrol` to set pulse_viz's monitoring device to that of your primary desktop audio device.

Data Sources

- `fft` : The current FFT level (0.0 to 1.0) for all FFT bins. Subscription, default 16.67ms refresh.
- `band` : The current FFT level (0.0 to 1.0) for all bands. Subscription, default 16.67ms refresh.

Sample Output

```
{
    "pulse_viz/band": [
        0.123123,
        0.237892,
        0.83792,
        0.37855,
        0.382793,
        0.38927,
        0.72893,
        0.83792,
        0.8327492,
        0.3827938,
        0.84641651,
        0.62826286,
        0.6654456,
        0.4864866,
        0.1691962,
        0.8641233
    ]
}
```

5.9.2 Settings

pulse_viz provides most of settings available to the Rainmeter AudioLevel plugin, with the exception of parameters which define specific data retrieval settings such as `Channel`, `FFTIdx`, and `BandIdx`. The parameter `Port` is not supported in pulse_viz.

See the Rainmeter AudioLevel documentation for more details.

5.10 Extension API Reference

5.10.1 extension_api.h

Base Extension functions.

This file defines base functions that must be implemented by a Quasar extension. These are called to load and destroy the extension.

Defines

EXPORT

Functions

quasar_ext_info_t ***quasar_ext_load**(void)

Loads this extension.

This function should only populate a *quasar_ext_info_t* struct with this extension's info and return it. Allocations for resources required by this extension should be performed in *quasar_ext_info_t::init* instead.

See also:

quasar_ext_info_t, *quasar_ext_info_t.init*

Returns

pointer to a populated *quasar_ext_info_t* struct if successful, nullptr otherwise

void **quasar_ext_destroy**(*quasar_ext_info_t* *info)

Destroys this extension.

This function should extension any resources allocated for the *quasar_ext_info_t* struct (as well as the struct instance itself if necessary)

See also:

quasar_ext_info_t

Parameters
info – **[in]** Extension info data

5.10.2 extension_types.h

Types used by the Extension API.

This file defines types used by the Extension API. Included by extension_api.h.

Defines

QUASAR_API_VERSION

Quasar extension API version.

Typedefs

typedef void ***quasar_settings_t**

Handle for creating and storing extension settings.

This handle is opaque to the front facing API.

See also:

extension_support.h

typedef void ***quasar_selection_options_t**

Handle for creating and storing selection options in a selection type setting.

This handle is opaque to the front facing API.

See also:

extension_support.h

typedef void ***quasar_ext_handle**

Type for the extension handle pointer.

typedef void ***quasar_data_handle**

Handle type for storing return data.

See also:

extension_support.h, *quasar_ext_info_t.get_data*

typedef bool (*`ext_info_call_t`)(*quasar_ext_handle*)

Function pointer type for the *quasar_ext_info_t::init* and *quasar_ext_info_t::shutdown* functions.

See also:

quasar_ext_info_t.init, *quasar_ext_info_t.shutdown*

typedef *quasar_settings_t* *(*`ext_create_settings_call_t`)(*quasar_ext_handle*)

Function pointer type for the *quasar_ext_info_t::create_settings* function.

See also:

quasar_ext_info_t.create_settings

typedef void (*`ext_settings_call_t`)(*quasar_settings_t**)

Function pointer type for settings related functions, like *quasar_ext_info_t::update*.

See also:

quasar_ext_info_t.update

typedef bool (*`ext_get_data_call_t`)(size_t, *quasar_data_handle*, char*)

Function pointer type for the *quasar_ext_info_t::get_data* function.

See also:

quasar_ext_info_t.get_data

Enums

enum `quasar_log_level_t`

Defines valid log levels for logging.

Values:

enumerator `QUASAR_LOG_DEBUG`

Debug level.

enumerator `QUASAR_LOG_INFO`

Info level.

enumerator `QUASAR_LOG_WARNING`

Warning level.

enumerator `QUASAR_LOG_ERROR`

Error level.

enumerator **QUASAR_LOG_CRITICAL**

Critical level.

enum **quasar_polling_type_t**

Defines valid polling type values.

Positive values determine extension timed data refresh rate

See also:

quasar_data_source_t.rate

Values:

enumerator **QUASAR_POLLING_SIGNALED**

Extension is responsible for signaling data send when data is ready.

enumerator **QUASAR_POLLING_CLIENT**

Data is polled on-demand by the client.

struct **quasar_data_source_t**

#include <extension_types.h> Struct for defining Data Sources.

Defines the Data Sources available to widgets provided by this extension.

See also:

quasar_ext_info_t.dataSources

Public Members

char **name**[32]

Identifier for this data source.

int64_t **rate**

Default rate of refresh for this Data Source (in microseconds). See *quasar_polling_type_t* for additional polling options.

See also:

quasar_signal_data_ready(), *quasar_signal_wait_processed()*, *quasar_polling_type_t*

uint64_t **validtime**

For client polled data (*QUASAR_POLLING_CLIENT*), this defines the duration in milliseconds that newly retrieved data is cached remains valid. Additional poll requests during this valid duration will return the cached data. A value of 0 means the data is never cached. Not used for other polling types.

See also:

quasar_polling_type_t

size_t **uid**

uid assigned to this Data Source by Quasar. An integer uid is assigned to each Data Source by Quasar to reduce discrepancies and avoid string comparisons. This uid is passed to *quasar_ext_info_t::get_data*.

struct **quasar_ext_info_fields_t**

#include <extension_types.h> Struct for defining information and description fields for the extension.

Defines the information and description fields for this extension.

See also:

quasar_ext_info_t.fields

Public Members

char **name**[32]

A unique short identifier for this extension. Used by widgets to identify and subscribe to this extension.

char **fullname**[64]

Full name of this extension.

char **version**[64]

Version string.

char **author**[64]

Author.

char **description**[256]

Extension description.

char **url**[256]

Extension website url, if any.

struct **quasar_ext_info_t**

#include <extension_types.h> Struct for defining a Quasar extension.

An extension should populate this struct with data upon initialization and return it to Quasar when *quasar_ext_load()* is called

See also:

quasar_ext_load()

Public Members

int **api_version**

API version. Should always be initialized to *QUASAR_API_VERSION*.

quasar_ext_info_fields_t ***fields**

Extension info/description fields. Must be initialized.

size_t **numDataSources**

Number of Data Sources provided by this extension.

quasar_data_source_t ***dataSources**

Array of Data Sources provided by this extension.

See also:

quasar_data_source_t

ext_info_call_t **init**

bool init(*quasar_ext_handle* handle)

> **Attention:** Extensions are **REQUIRED** to implement this function.

This function should save data source uids assigned by Quasar as well as initialize any resources needed by the extension. The extension's handle will be passed into this function. This function should save the handle.

Return
true if success, false otherwise

ext_info_call_t **shutdown**

bool shutdown(*quasar_ext_handle* handle)

> **Attention:** Extensions are **REQUIRED** to implement this function.

This function should cleanup any resources initialized by this extension.

Return
Should always return true

ext_get_data_call_t **get_data**

bool get_data(size_t uid, *quasar_data_handle* handle, char* args)

> **Attention:** Extensions are **REQUIRED** to implement this function.

Retrieves the data of a specific Data Source entry.

args contains a null terminate string that consists of any arguments passed to the Data Source entry, if arguments are accepted. args is null if no arguments are passed.

Use support functions in extension_support.h to populate data into handle.

Important: This function needs to be both re-entrant and thread-safe.

See also:

quasar_set_data_string(), *quasar_set_data_json()*, *quasar_set_data_int()*, *quasar_set_data_double()*, *quasar_set_data_bool()*, *quasar_set_data_string_array()*, *quasar_set_data_int_array()*, *quasar_set_data_float_array()*, *quasar_set_data_double_array()*

Return
true if success, false otherwise

ext_create_settings_call_t **create_settings**

quasar_settings_t* create_settings(*quasar_ext_handle* handle), **OPTIONAL**

Creates extension settings (and corresponding UI elements) if any.

See also:

quasar_create_settings(), *quasar_add_int_setting()*, *quasar_add_bool_setting()*, *quasar_add_double_setting()*

Return
quasar_settings_t pointer if successful, nullptr otherwise

ext_settings_call_t **update**

void update(quasar_settings_t* settings), **OPTIONAL**

This function should update local settings values.

See also:

quasar_get_int_setting(), *quasar_get_uint_setting()*, *quasar_get_bool_setting()*, *quasar_get_double_setting()*

5.10.3 extension_support.h

Extension API support functionality.

This file defines functions that augment extension functionality

Functions

char *`quasar_strcpy`(char *dest, size_t destSize, const char *src, size_t srcSize)

A custom string copy implementation to work around inherently unsafe C library functions.

If src contains a null terminator before srcSize is reached, this function will correctly terminate. If the [0, srcSize) portion of src does not contain a null terminator, it will copy up to src[srcSize-1] and terminate. If destSize is smaller than src at null terminator or srcSize, it will copy up to destSize-2 and terminate. Assuming that the copy occurred, dest will be null terminated.

Parameters

- `dest` – **[in]** Destination buffer
- `destSize` – **[in]** Size of destination buffer
- `src` – **[in]** Source string
- `srcSize` – **[in]** Maximum size of source string

Returns
dest

void `quasar_log`(*quasar_log_level_t* level, const char *msg)

Logs a message to Quasar console.

See also:

quasar_log_level_t

Parameters

- `level` – **[in]** Log level
- `msg` – **[in]** Log message

quasar_settings_t *`quasar_create_settings`(*quasar_ext_handle* handle)

Returns an instance of *quasar_settings_t*.

Use in *quasar_ext_info_t::create_settings* to create extension settings. Ensure that only a single instance is used per extension.

Parameters
`handle` – **[in]** Extension handle

Returns
quasar_settings_t instance if successful, nullptr otherwise

quasar_selection_options_t *`quasar_create_selection_setting`(void)

Creates a new *quasar_selection_options_t* setting.

Use with *quasar_add_selection_setting()* after populating options.

Returns
quasar_selection_options_t instance if successful, nullptr otherwise

void **quasar_free_selection_setting**(*quasar_selection_options_t* *handle)

Deletes a *quasar_selection_options_t* setting. Should only be used in error case exits.

Parameters
handle – **[in]** Handle to instance to be freed

quasar_data_handle **quasar_set_data_string**(*quasar_data_handle* hData, const char *data)

Sets the return data to be a null terminated string.

Parameters

- **hData** – **[in]** Data handle
- **data** – **[in]** Data to set

Returns
Data handle if successful, nullptr otherwise

quasar_data_handle **quasar_set_data_int**(*quasar_data_handle* hData, int data)

Sets the return data to be an integer.

Parameters

- **hData** – **[in]** Data handle
- **data** – **[in]** Data to set

Returns
Data handle if successful, nullptr otherwise

quasar_data_handle **quasar_set_data_double**(*quasar_data_handle* hData, double data)

Sets the return data to be a floating point double.

Parameters

- **hData** – **[in]** Data handle
- **data** – **[in]** Data to set

Returns
Data handle if successful, nullptr otherwise

quasar_data_handle **quasar_set_data_bool**(*quasar_data_handle* hData, bool data)

Sets the return data to be a bool.

Parameters

- **hData** – **[in]** Data handle
- **data** – **[in]** Data to set

Returns
Data handle if successful, nullptr otherwise

quasar_data_handle **quasar_set_data_json**(*quasar_data_handle* hData, const char *data)

Sets the return data to be a valid JSON object string.

Parameters

- **hData** – **[in]** Data handle
- **data** – **[in]** Data to set

Returns
Data handle if successful, nullptr otherwise

quasar_data_handle **quasar_set_data_string_array**(*quasar_data_handle* hData, char **arr, size_t len)

Sets the return data to be an array of null terminated strings.

Parameters

- **hData** – **[in]** Data handle
- **arr** – **[in]** Array of data to set
- **len** – **[in]** Length of array

Returns
Data handle if successful, nullptr otherwise

quasar_data_handle **quasar_set_data_int_array**(*quasar_data_handle* hData, int *arr, size_t len)

Sets the return data to be an array of integers.

Parameters

- **hData** – **[in]** Data handle
- **arr** – **[in]** Array of data to set
- **len** – **[in]** Length of array

Returns
Data handle if successful, nullptr otherwise

quasar_data_handle **quasar_set_data_float_array**(*quasar_data_handle* hData, float *arr, size_t len)

Sets the return data to be an array of floats.

Parameters

- **hData** – **[in]** Data handle
- **arr** – **[in]** Array of data to set
- **len** – **[in]** Length of array

Returns
Data handle if successful, nullptr otherwise

quasar_data_handle **quasar_set_data_double_array**(*quasar_data_handle* hData, double *arr, size_t len)

Sets the return data to be an array of doubles.

Parameters

- **hData** – **[in]** Data handle
- **arr** – **[in]** Array of data to set
- **len** – **[in]** Length of array

Returns
Data handle if successful, nullptr otherwise

quasar_data_handle **quasar_set_data_null**(*quasar_data_handle* hData)

Sets the return data to be null.

Parameters
hData – **[in]** Data handle

Returns
Data handle if successful, nullptr otherwise

quasar_data_handle **quasar_append_error**(*quasar_data_handle* hData, const char *err)

Adds an error to the return data to be sent back to the client.

Parameters

- **hData** – **[in]** Data handle
- **err** – **[in]** Error to add

Returns
Data handle if successful, nullptr otherwise

quasar_settings_t ***quasar_add_int_setting**(*quasar_ext_handle* handle, *quasar_settings_t* *settings, const char *name, const char *description, int min, int max, int step, int dflt)

Creates an integer setting in extension settings.

Parameters

- **handle** – **[in]** Extension handle
- **settings** – **[in]** The extension settings handle
- **name** – **[in]** Name of the setting
- **description** – **[in]** Description for the setting
- **min** – **[in]** Minimum value
- **max** – **[in]** Maximum value
- **step** – **[in]** Incremental step
- **dflt** – **[in]** Default value

Returns
The settings handle if successful, nullptr otherwise

quasar_settings_t ***quasar_add_bool_setting**(*quasar_ext_handle* handle, *quasar_settings_t* *settings, const char *name, const char *description, bool dflt)

Creates a bool setting in extension settings.

Parameters

- **handle** – **[in]** Extension handle
- **settings** – **[in]** The extension settings handle
- **name** – **[in]** Name of the setting
- **description** – **[in]** Description for the setting
- **dflt** – **[in]** Default value

Returns
The settings handle if successful, nullptr otherwise

quasar_settings_t ***quasar_add_double_setting**(*quasar_ext_handle* handle, *quasar_settings_t* *settings, const char *name, const char *description, double min, double max, double step, double dflt)

Creates a double setting in extension settings.

Parameters

- **handle** – **[in]** Extension handle
- **settings** – **[in]** The extension settings handle
- **name** – **[in]** Name of the setting
- **description** – **[in]** Description for the setting
- **min** – **[in]** Minimum value
- **max** – **[in]** Maximum value
- **step** – **[in]** Incremental step
- **dflt** – **[in]** Default value

Returns
The settings handle if successful, nullptr otherwise

quasar_settings_t ***quasar_add_string_setting**(*quasar_ext_handle* handle, *quasar_settings_t* *settings, const char *name, const char *description, const char *dflt, bool password)

Creates a string type setting in extension settings.

Parameters

- **handle** – **[in]** Extension handle
- **settings** – **[in]** The extension settings handle
- **name** – **[in]** Name of the setting
- **description** – **[in]** Description for the setting
- **dflt** – **[in]** Default value (if any)
- **password** – **[in]** Whether this field is a password/obscured field in the UI

Returns
The settings handle if successful, nullptr otherwise

quasar_settings_t ***quasar_add_selection_setting**(*quasar_ext_handle* handle, *quasar_settings_t* *settings, const char *name, const char *description, *quasar_selection_options_t* *select)

Creates a selection type setting in extension settings.

Parameters

- **handle** – **[in]** Extension handle
- **settings** – **[in]** The extension settings handle
- **name** – **[in]** Name of the setting
- **description** – **[in]** Description for the setting
- **select** – **[in]** Handle to the selection setting instance (takes ownership)

Returns
The settings handle if successful, nullptr otherwise

quasar_selection_options_t ***quasar_add_selection_option**(*quasar_selection_options_t* *select, const char *name, const char *value)

Creates a selection type setting in extension settings.

Parameters

- **select** – **[in]** The selection setting handle
- **name** – **[in]** Name of the option (shown in UI)
- **value** – **[in]** Actual value of the option

Returns
The setting handle if successful, nullptr otherwise

intmax_t **quasar_get_int_setting**(*quasar_ext_handle* handle, *quasar_settings_t* *settings, const char *name)

Retrieves an integer setting from Quasar.

Parameters

- **handle** – **[in]** Extension handle
- **settings** – **[in]** The extension settings handle
- **name** – **[in]** Name of the setting

Returns
Value of the setting if successful, default value otherwise

uintmax_t **quasar_get_uint_setting**(*quasar_ext_handle* handle, *quasar_settings_t* *settings, const char *name)

Retrieves an unsigned integer setting from Quasar.

Parameters

- **handle** – **[in]** Extension handle
- **settings** – **[in]** The extension settings handle
- **name** – **[in]** Name of the setting

Returns
Value of the setting if successful, default value otherwise

bool **quasar_get_bool_setting**(*quasar_ext_handle* handle, *quasar_settings_t* *settings, const char *name)

Retrieves a bool setting from Quasar.

Parameters

- **handle** – **[in]** Extension handle
- **settings** – **[in]** The extension settings handle
- **name** – **[in]** Name of the setting

Returns
Value of the setting if successful, default value otherwise

double **quasar_get_double_setting**(*quasar_ext_handle* handle, *quasar_settings_t* *settings, const char *name)

Retrieves a double setting from Quasar.

Parameters

- **handle** – **[in]** Extension handle
- **settings** – **[in]** The extension settings handle
- **name** – **[in]** Name of the setting

Returns
Value of the setting if successful, default value otherwise

bool **quasar_get_string_setting**(*quasar_ext_handle* handle, *quasar_settings_t* *settings, const char *name, char *buf, size_t size)

Retrieves a string setting from Quasar.

Parameters

- **handle** – **[in]** Extension handle
- **settings** – **[in]** The extension settings handle
- **name** – **[in]** Name of the setting
- **buf** – **[in]** Buffer to copy results to
- **size** – **[in]** Size of buffer

Returns
true if successful, false otherwise

bool **quasar_get_selection_setting**(*quasar_ext_handle* handle, *quasar_settings_t* *settings, const char *name, char *buf, size_t size)

Retrieves a selection setting from Quasar.

Parameters

- **handle** – **[in]** Extension handle
- **settings** – **[in]** The extension settings handle
- **name** – **[in]** Name of the setting
- **buf** – **[in]** Buffer to copy results to
- **size** – **[in]** Size of buffer

Returns
true if successful, false otherwise

void **quasar_signal_data_ready**(*quasar_ext_handle* handle, const char *source)

Signals to Quasar that data is ready to be sent to clients.

This function is for Data Sources with *quasar_data_source_t::rate* set to *QUASAR_POLLING_CLIENT* or *QUASAR_POLLING_SIGNALED*. This function signals to Quasar that the data for the specified source is ready to be sent.

See also:

quasar_data_source_t.rate

Parameters

- **handle** – **[in]** Extension handle
- **source** – **[in]** Data Source identifier

void **quasar_signal_wait_processed**(*quasar_ext_handle* handle, const char *source)

Waits for a set of data to be sent to clients before processing the next set.

This function is for Data Sources with *quasar_data_source_t::rate* set to *QUASAR_POLLING_SIGNALED*. This function can be used to allow a thread to wait until a set of data has been consumed before moving on to processing the next set.

See also:

quasar_data_source_t.rate

Parameters

- **handle** – **[in]** Extension handle
- **source** – **[in]** Data Source identifier

void **quasar_set_storage_string**(*quasar_ext_handle* handle, const char *name, const char *data)

Stores a string type data.

Parameters

- **handle** – **[in]** Extension handle
- **name** – **[in]** Data name
- **data** – **[in]** Data to set

void **quasar_set_storage_int**(*quasar_ext_handle* handle, const char *name, int data)

Stores a int type data.

Parameters

- **handle** – **[in]** Extension handle
- **name** – **[in]** Data name
- **data** – **[in]** Data to set

void **quasar_set_storage_double**(*quasar_ext_handle* handle, const char *name, double data)

Stores a double type data.

Parameters

- **handle** – **[in]** Extension handle
- **name** – **[in]** Data name
- **data** – **[in]** Data to set

void **quasar_set_storage_bool**(*quasar_ext_handle* handle, const char *name, bool data)

Stores a bool type data.

Parameters

- **handle** – **[in]** Extension handle
- **name** – **[in]** Data name
- **data** – **[in]** Data to set

bool **quasar_get_storage_string**(*quasar_ext_handle* handle, const char *name, char *buf, size_t size)

Gets a string type data from storage.

Parameters

- **handle** – **[in]** Extension handle
- **name** – **[in]** Data name
- **buf** – **[in]** Buffer to copy results to
- **size** – **[in]** Size of buffer

Returns
true if successful, false otherwise

bool **quasar_get_storage_int**(*quasar_ext_handle* handle, const char *name, int *buf)

Gets a int type data from storage.

Parameters

- **handle** – **[in]** Extension handle
- **name** – **[in]** Data name
- **buf** – **[in]** Buffer to copy results to

Returns
true if successful, false otherwise

bool **quasar_get_storage_double**(*quasar_ext_handle* handle, const char *name, double *buf)

Gets a double type data from storage.

Parameters

- **handle** – **[in]** Extension handle
- **name** – **[in]** Data name
- **buf** – **[in]** Buffer to copy results to

Returns
true if successful, false otherwise

bool **quasar_get_storage_bool**(*quasar_ext_handle* handle, const char *name, bool *buf)

Gets a bool type data from storage.

Parameters

- **handle** – **[in]** Extension handle
- **name** – **[in]** Data name
- **buf** – **[in]** Buffer to copy results to

Returns
true if successful, false otherwise

5.10.4 extension_support.hpp

C++ specific Extension API support functions.

Attention: Functions in this file passes STL types across library boundaries. ENSURE THAT CRT LINKAGE IS SET TO DYNAMIC ON WINDOWS WHEN USING THESE!!!

Functions

quasar_data_handle **quasar_set_data_string_hpp**(*quasar_data_handle* hData, std::string_view data)

Sets the return data to be a null terminated string.

Parameters

- **hData** – **[in]** Data handle
- **data** – **[in]** Data to set

Returns

Data handle if successful, nullptr otherwise

quasar_data_handle **quasar_set_data_json_hpp**(*quasar_data_handle* hData, std::string_view data)

Sets the return data to be a valid JSON object string.

Parameters

- **hData** – **[in]** Data handle
- **data** – **[in]** Data to set

Returns

Data handle if successful, nullptr otherwise

quasar_data_handle **quasar_set_data_string_vector**(*quasar_data_handle* hData, const std::vector<std::string> &vec)

Sets the return data to be an array of null terminated strings.

Parameters

- **hData** – **[in]** Data handle
- **vec** – **[in]** Vector of data to set

Returns

Data handle if successful, nullptr otherwise

quasar_data_handle **quasar_set_data_int_vector**(*quasar_data_handle* hData, const std::vector<int> &vec)

Sets the return data to be an array of integers.

Parameters

- **hData** – **[in]** Data handle
- **vec** – **[in]** Vector of data to set

Returns

Data handle if successful, nullptr otherwise

quasar_data_handle **quasar_set_data_float_vector**(*quasar_data_handle* hData, const std::vector<float> &vec)

Sets the return data to be an array of floats.

Parameters

- **hData** – **[in]** Data handle
- **vec** – **[in]** Vector of data to set

Returns

Data handle if successful, nullptr otherwise

quasar_data_handle **quasar_set_data_double_vector**(*quasar_data_handle* hData, const std::vector<double> &vec)

Sets the return data to be an array of doubles.

Parameters

- **hData** – **[in]** Data handle
- **vec** – **[in]** Vector of data to set

Returns
Data handle if successful, nullptr otherwise

std::string_view **quasar_get_string_setting_hpp**(*quasar_ext_handle* handle, *quasar_settings_t* *settings, std::string_view name)

Retrieves a string setting from Quasar.

Parameters

- **handle** – **[in]** Extension handle
- **settings** – **[in]** The extension settings handle
- **name** – **[in]** Name of the setting

Returns
string setting if successful, empty string_view otherwise

std::string_view **quasar_get_selection_setting_hpp**(*quasar_ext_handle* handle, *quasar_settings_t* *settings, std::string_view name)

Retrieves a selection setting from Quasar.

Parameters

- **handle** – **[in]** Extension handle
- **settings** – **[in]** The extension settings handle
- **name** – **[in]** Name of the setting

Returns
string setting if successful, empty string_view otherwise

5.11 Widget Client Protocol

The Widget Client Protocol is the protocol that is used by Quasar and Quasar widgets to communicate with each other. The messages are JSON-encapsulated data that is sent over WebSocket. This document defines the message format and accepted fields, as well as sample usages in JavaScript.

5.11.1 Globals Functions

These global JavaScript functions are defined for all Quasar loaded widgets:

quasar_create_websocket()
: Creates a WebSocket object connecting to Quasar's Data Server.

quasar_authenticate(socket)
: Authenticates this widget with the Quasar Data Server.

Sample Usage

```
websocket = quasar_create_websocket();
websocket.onopen = function(evt) {
    quasar_authenticate(websocket);
};
```

5.11.2 Data Protocol

Client to Server

The following is the basic message format used by a client widget to send messages to the Quasar Data Server.

Basic Message Format

```
{
    method: <method>,
    params: {
        topics: [<targets>],
        args: <args>,
        params: [<list of target params>]
    }
}
```

Field Descriptions

method
: The method/function to be invoked by this message. For client widgets, supported values are: `subscribe`, and `query`. `subscribe` is used to subscribe to timer-based or extension signaled Data Sources, while `query` is used for client polled Data Sources as well as any other commands. For Quasar loaded widgets, `auth` is also supported for authenication purposes.

params
: The parameters sent to the method. This field should be a JSON object that is typically comprised of at least the field `topics`.

topics
: List of intended targets. Typically, this is an extension's identifier plus the Data Source identifier separated by a forward slash.

args
: Optional arguments sent to the target. Only supported by queried/client polled sources, if arguments are supported by the source.

target params
: List of parameters sent to all targets. Typically, this field is unused.

Sample Usages

```
function subscribe() {
    const msg = {
        method: "subscribe",
        params: {
            topics: ["win_audio_viz/band"]
        }
    }

    websocket.send(JSON.stringify(msg));
}

function poll() {
    const msg = {
        method: "query",
        params: {
            topics: ["win_simple_perf/sysinfo_polled"]
        }
    }

    websocket.send(JSON.stringify(msg));
}

function get_launcher_list() {
    const msg = {
        method: "query",
        params: {
            topics: ["applauncher/list"]
        }
    }
```

(continues on next page)

(continued from previous page)

```
    websocket.send(JSON.stringify(msg));
}

function launcher_cmd(cmd, arg) {
    let msg = {
        method: "query",
        params: {
            topics: [`applauncher/${cmd}`],
        },
    };

    if (arg) {
        msg.params["args"] = arg;
    }

    websocket.send(JSON.stringify(msg));
}

function authenticate() {
    const msg = {
        method: "auth",
        params: {
            code: "6EFBBE6542D52FDD294337343147B033"
        }
    }

    websocket.send(JSON.stringify(msg));
}
```

Refer to the source code of sample widgets for concrete examples of client to server communications, or the source code of sample extensions for examples of specific targets.

Server to Client

The following is the basic message format used by the Data Server to send data and messages to client widgets.

Basic Message Format

```
{
    <target>: {
        <target data>
    },
    ...<target>: {
        <target data>
    },
    errors: <errors>
}
```

Field Descriptions

The top level `target` fields holds all the data sent with the message.

`target` and `target data`
: Typically specifies the Data Source identifier and the data payload sent by the extension.

`errors`
: Any errors that occurred while retrieving the data.

Sample Messages

Sample messages sent by various sources, including sample extensions and extension settings, and App Launcher command list:

```
{
    "win_simple_perf/sysinfo": {
        "cpu": 15,
        "ram": {
            "total": 34324512768,
            "used": 10252300288
        }
    },
    "errors": ["Unknown topic win_simple_perf/band requested in extension win_simple_perf
↪"]
}

{
    "win_audio_viz/settings": [
        {
            "def": 256,
            "desc": "FFTSize",
            "max": 8192,
            "min": 0,
            "name": "FFTSize",
            "step": 2,
            "type": "int",
            "val": 1024
        },
        {
            "def": 16,
            "desc": "Number of Bands",
            "max": 1024,
            "min": 0,
            "name": "Bands",
            "step": 1,
            "type": "int",
            "val": 32
        }
    ]
}
```

Sample Usage

This following sample is taken from the *Creating a Widget* documentation, and defines functions which processes incoming data sent by the win_simple_perf sample extension.

```
function parseMsg(msg) {
    const data = JSON.parse(msg);

    if ("win_simple_perf/sysinfo_polled" in data) {
        const vals = data["win_simple_perf/sysinfo_polled"]
        setData(document.getElementById("cpu"), vals["cpu"]);
        setData(
            document.getElementById("ram"),
            Math.round((vals["ram"]["used"] / vals["ram"]["total"]) * 100),
        );
    }
}
```

5.11.3 App Launcher

The App Launcher follows the basic message formats as described above.

For example, sending the following message:

```
{
    "method": "query",
    "params": {
        "topics": ["applauncher/list"]
    }
}
```

Will see Quasar respond with the following sample reply:

```
{
    "applauncher/list": [{
        "command": "chrome",
        "icon": "..."
    }, {
        "command": "spotify",
        "icon": "data:image/png;base64..."
    }, {
        "command": "steam",
        "icon": "data:image/png;base64..."
    }]
}
```

Where `chrome`, `spotify`, and `steam` are commands preconfigured in the *App Launcher Settings*. Subsequently, an App Launcher widget may then send:

```
{
    "method": "query",
    "params": {
        "topics": ["applauncher/launch"],
```

(continues on next page)

(continued from previous page)

```
        "args": "chrome"
    }
}
```

At which point the command/application registered with the App Launcher command `chrome` will then execute.

See *Setting up the App Launcher* for details on setting up the App Launcher.

5.12 Widget Definition Reference

A Widget Definition file is a JSON file that contains at least the following parameters:

name
: Name of the widget.

width
: Width of the widget.

height
: Height of the widget.

startFile
: Entry point for the widget. This can be a local file or a URL.

transparentBg
: Whether the widget background is transparent.

5.12.1 Example

```
{
    "name": "simple_perf",
    "width": 400,
    "height": 120,
    "startFile": "index.html",
    "transparentBg": true,
    "required": ["win_simple_perf"]
}
```

5.12.2 Optional Parameters

dataserver: true/false (default false)
: Defines whether connection and authentication scripts are loaded into the widget. This parameter must be defined and set to `true` if the widget requires a connection to the Data Server. If this parameter is not defined, or if `false` is set, then the connection/authentication scripts are not loaded and the widget will not be able to connect to the Data Server.

required: String array
: A string array containing extension identifiers that the widget requires to function. The widget will fail to load if one or more of the extensions listed here are unavailable.

remoteAccess: true/false
: By default, the Chrome instances hosting locally defined widgets cannot access remote URLs (including

Javascript libraries and stylesheets hosted on the internet) due to Cross-Origin restrictions. This parameter can be defined by a widget to allow remote URL accesses. If the parameter is defined, a security warning will be shown when loading the widget to remind the user that the widget should be downloaded from a trusted source when allowing remote URL access.

`clickable`: true/false (default false)
Defines whether the widget's contents can be interacted with by default (e.g. links, App Launcher widgets). See also *Widget Menu*.

5.13 Logging

Quasar provides logging for debugging purposes, where all log messages sent to Quasar are outputted.

The log window can be accessed by right-clicking the Quasar icon in the notification bar, and clicking **Log**. Here, the log messages as well as its time, severity, and in some cases, the location of its source code can be inspected.

If the **Log to file?** setting is enabled in *Settings*, then all log messages will be written to a file as well (typically in `%AppData%\quasar\quasar.log` on Windows). Similarily, the **Log Level** setting controls the minimum severity of log messages that gets outputted.

Quasar log will output log messages from the main Quasar application, extensions, as well as widgets. For widgets, Quasar hooks into the JavaScript console `console.log()`. For extensions, the function *`quasar_log()`* is provided in the *Extension API*.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- search

INDEX