AromaSDK

Lightweight development toolkit for cross-platform applications on Linux, Android, and embedded systems.

Version 0.0.1 (alpha)

Generated May 03, 2026

Contents

SDK Installation

1. Clone the Repo

To get started with AromaUI, you need to clone the repository from GitHub. Open your terminal or command prompt and run the following command:

```
git clone https://github.com/BinaryInkTN/AromaUI.git --recursive
```

2. Add aroma to PATH

Once the download is complete, locate the `aroma` file (usually in the bin folder) and add its location to your system's PATH environment variable. This will allow you to run the `aroma` command from any location in your terminal or command prompt.

3. Verify Installation

To verify that you have the correct dependencies installed and that AromaUI is properly set up, you can run the following command:

```
yassine@DESKTOP-4SJS544:~/AromaUI$ aroma doctor
==> Running Aroma Doctor...
OS: Linux 6.6.87.2-microsoft-standard-WSL2
✓ CMake installed
✓ Ninja: 1.11.1
✓ GCC installed
✓ Java installed
✓ keytool installed
✓ Android SDK: /home/yassine/Android/Sdk
✓ Android NDK: /home/yassine/Android/Sdk/ndk/25.1.8937393
==> Doctor summary complete.
```

> If you don't have the SDK or NDK installed run the following command to install them:

```
aroma install-sdk
```

Project Setup

1. Project Creation via Command Line Tool

To create a new AromaUI project, you can use the `aroma` command-line tool. Open your terminal and run the following command:

```
aroma create
```

This command will prompt you to enter a project name and select the min, target and compile android API versions.

After providing the necessary information, the tool will generate a new project with the appropriate structure and configuration files.

2. Project Structure Overview

A typical AromaUI project contains:

```
project/
 ├── src/
 │    └── main.c
 ├── android/
 ├── CMakeLists.txt
 └── aroma.json
```

Your application logic is written in C inside `src/`.

Infotainment Example

revision	author	date	description
1	AHMED ALI Yassine	2026-03-19	Initial creation of the Infotainment example documentation.

The Infotainment example demonstrates how to create a simple infotainment system interface using AromaUI. It includes features such as media controls, navigation, voice commands, and vehicle information display.

> This example is still a work in progress and will be updated with more features and improvements in the future.

Voice Commands

The infotainment system supports voice commands for hands-free operation. Users can activate voice control by pressing the voice command button on the dashboard and speaking commands such as "Play music," "Navigate to home," or "Call John."

> The voice command feature is designed to enhance safety and convenience while driving, allowing users to interact with the infotainment system without taking their hands off the wheel or their eyes off the road.

Voice Commands

Command	Description
Hey Aroma	Wake-up command
Play Music	Start playing music from the media library
Stop Music	Stop playing music from the media library
Turn volume up/down	Adjust the volume level
Open Phone / Call / Dial	Open the phone application
Open Music	Open the music application
Open Settings	Open the settings application
Switch to light/dark theme	Switch the application theme
Colder / Hotter / Turn the ac (air conditioning) up/down	Control the air conditioning
What is the range?	Display the vehicle's range information
What is the charge?	Display the vehicle's battery information

Main Dashboard

The main dashboard provides an overview of the vehicle's status, including speed, fuel level, climate control. Infotainment Example

Settings Page

The settings page allows users to customize their infotainment experience, including theme selection and display preferences. Settings Page

Navigation Page

This features a map view with zoom controls, allowing users to navigate and explore their surroundings. Navigation Page

Running the Example

To run the Infotainment example, follow these steps:

1. Clone the AromaUI repository and build the project.

```
# Clone the repository
git clone https://github.com/BinaryInkTN/AromaUI.git

# Navigate to the project directory
cd AromaUI

# Build the project
cmake -S . -B build

# Navigate to the build directory
cd build

# Compile the project using all available CPU cores
make -j $(nproc)
```

1. Navigate to the examples directory and run the infotainment example.

```
# Navigate to the examples directory
cd examples/05_car_infotainment

# Build the example
cmake -S . -B build
cd build
make -j $(nproc)

# Run the example
./infotainment
```

Hello World Tutorial

This document explains a minimal AromaUI application written in C. It demonstrates initialization, layout creation, event handling, rendering, and cleanup, along with Android support.

Overview

The application creates a fullscreen window containing:

- A centered label displaying a greeting message
- A button that updates the message when clicked

Application Structure

State Management

The application uses a structure to store UI elements and runtime data:

```
typedef struct
{
    AromaNode *greeting_label;
    AromaNode *root_container;
    AromaNode *btn_click;
    char greeting_text[64];
    int click_count;
    AromaFont *title_font;
    AromaFont *button_font;
    AromaWindow *window;
} AppState;
```

Description

- `greeting_label`: Label displaying the current message
- `root_container`: Main layout container
- `btn_click`: Button widget
- `greeting_text`: Buffer holding the label text
- `click_count`: Tracks how many times the button was pressed
- `title_font`: Font used for the label
- `button_font`: Font used for the button
- `window`: Main application window

Initialization

```
if (!aroma_ui_init())
{
    printf("Failed to initialise AromaUI\n");
    return 1;
}
```

Initializes the AromaUI system. This must be called before any other UI operations.

Theme Setup

```
AromaTheme theme = aroma_theme_create_material_blue_dark();
aroma_ui_set_theme(&theme);
```

Creates and applies a Material Design dark theme.

Window Creation

```
state.window = aroma_ui_create_window(
    "AromaUI Hello World",
    aroma_android_dp_to_px(400),
    aroma_android_dp_to_px(600)
);
```

Creates the main window.

On Android:

- The title is ignored
- The window automatically uses the device screen size

Enable fullscreen:

```
aroma_window_set_fullscreen((AromaNode *)state.window, true);
```

Font Loading

```
state.title_font = aroma_font_create_from_memory(
    aroma_ubuntu_ttf, aroma_ubuntu_ttf_len,
    aroma_android_sp_to_px(36)
);

state.button_font = aroma_font_create_from_memory(
    aroma_ubuntu_ttf, aroma_ubuntu_ttf_len,
    aroma_android_sp_to_px(20)
);
```

Loads fonts from memory using embedded font data.

Layout System

```
state.root_container = aroma_ui_container(
    (AromaNode *)state.window,
    0, 0, w, h,
    AROMA_LAYOUT_MODE_FLEX,
    AROMA_FLEX_COLUMN,
    AROMA_JUSTIFY_CENTER,
    AROMA_ALIGN_CENTER
);
```

Creates a flex container that:

- Arranges children vertically
- Centers them horizontally and vertically

Spacing between elements:

```
aroma_node_set_gap(state.root_container, aroma_android_dp_to_px(32));
```

Label Creation

```
state.greeting_label = aroma_ui_label(
    state.root_container,
    state.greeting_text,
    0, 0,
    LABEL_STYLE_LABEL_LARGE,
    state.title_font
);
```

Displays the greeting text.

Button Creation

```
state.btn_click = aroma_ui_button(
    state.root_container,
    "Click me!",
    0, 0,
    btn_width,
    btn_height,
    on_click,
    &state,
    state.button_font
);
```

Creates a button and binds it to a click handler.

Event Handling

```
static bool on_click(AromaNode *btn, void *data)
{
    (void)btn;
    AppState *state = (AppState *)data;

    state->click_count++;
    snprintf(state->greeting_text, sizeof(state->greeting_text),
             "Hello, World! (%d)", state->click_count);

    aroma_label_set_text(state->greeting_label, state->greeting_text);
    return true;
}
```

Each click:

- Increments the counter
- Updates the label text

Main Loop

```
while (aroma_ui_is_running())
{
    aroma_ui_process_events();
    aroma_ui_render(state.window);
}
```

Handles:

- Input events (mouse, keyboard, touch)
- Rendering of the UI

Cleanup

```
if (state.btn_click)
    aroma_button_destroy(state.btn_click);

if (state.greeting_label)
    aroma_label_destroy(state.greeting_label);

if (state.root_container)
    aroma_container_destroy(state.root_container);

if (state.title_font)
    aroma_font_destroy(state.title_font);

if (state.button_font)
    aroma_font_destroy(state.button_font);

aroma_ui_destroy_window(state.window);
aroma_ui_shutdown();
```

Ensures all allocated resources are properly released.

Android Support

To run on Android, include the native entry point:

```
#ifdef __ANDROID__
#include <android_native_app_glue.h>

void android_main(struct android_app *state)
{
    aroma_android_set_app(state);
    main(0, NULL);
}
#endif
```

This connects AromaUI to the Android native activity lifecycle.

Notes

- Window title and size are ignored on Android
- Density-independent units (`dp`, `sp`) ensure consistent UI scaling
- The layout system is based on flexbox principles
- Fonts are loaded from memory, allowing embedded assets

Summary

This example demonstrates the core workflow of an AromaUI application:

1. Initialize the UI system
2. Set a theme
3. Create a window
4. Build a layout
5. Add widgets
6. Handle user interaction
7. Run the main loop
8. Clean up resources

This structure serves as a foundation for building more complex cross-platform GUI applications using AromaUI.

Architecture Overview

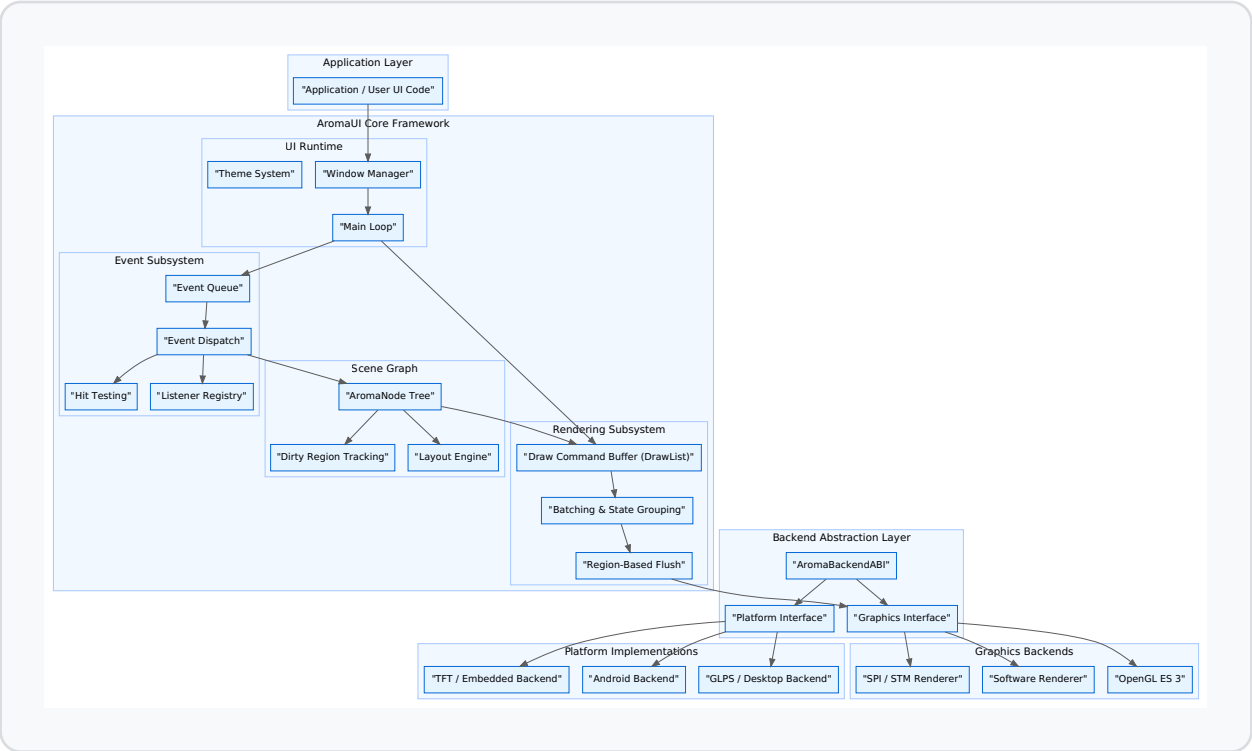
Overview

AromaUI is a modular, cross-platform retained-mode UI framework designed for embedded systems, Android environments, desktop platforms, and software-rendered targets.

The architecture follows a strict layered design to ensure:

- Clear separation of concerns
- Backend independence
- Deterministic memory behavior (embedded-friendly)
- Scalable rendering performance
- Maintainable subsystem boundaries

High-Level Architecture Diagram



Layer Breakdown

1. Application Layer

The application layer contains user-defined UI components and business logic. Applications interact with AromaUI through the Window Manager and Scene Graph APIs.

Responsibilities:

- Create windows and UI trees
- Register event listeners
- Trigger state updates
- Drive application-specific logic

2. AromaUI Core Framework

The core framework is divided into four primary subsystems.

UI Runtime

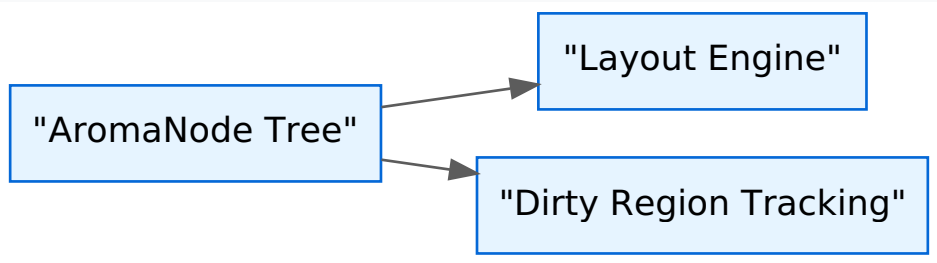
Manages lifecycle and execution.

- Window Manager - controls window instances and root nodes
- Main Loop - drives update and render cycles
- Theme System - centralized styling and visual configuration

Scene Graph

A retained tree of UI nodes.

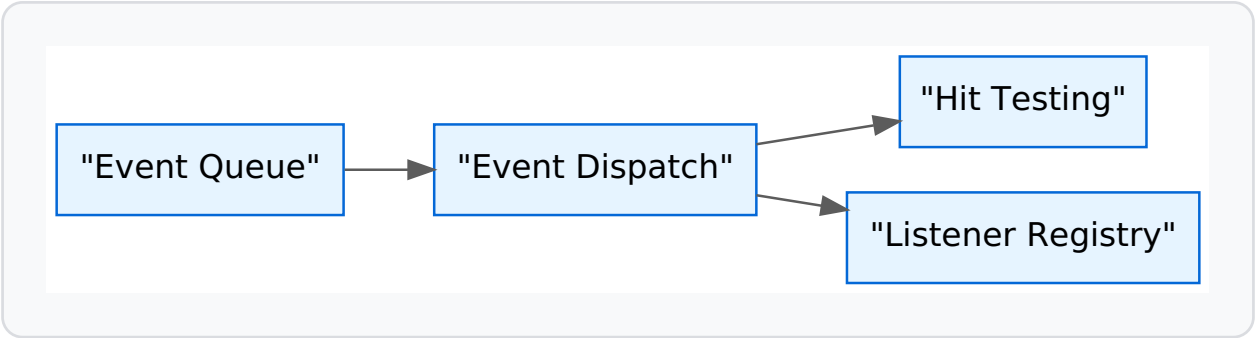
- AromaNode Tree - hierarchical UI structure
- Layout Engine - computes size and position
- Dirty Region Tracking - tracks areas requiring redraw



Event Subsystem

Handles input and dispatch.

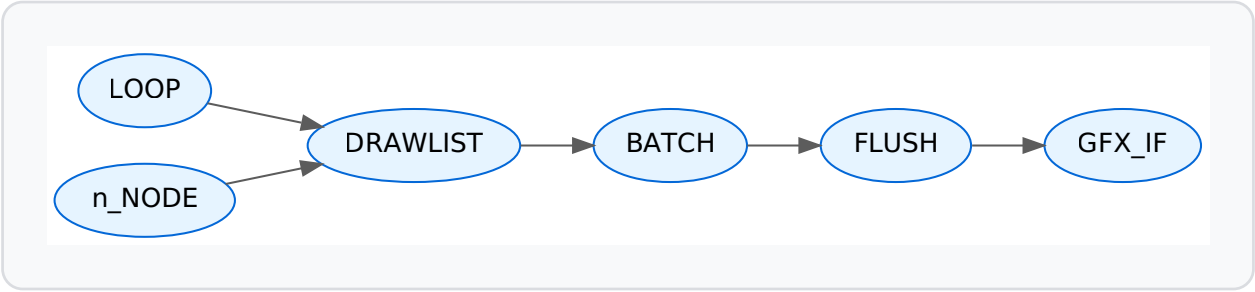
- Event Queue - buffered input events
- Event Dispatch - hierarchical propagation
- Hit Testing - resolves target node
- Listener Registry - efficient listener lookup



Rendering Subsystem

Command-buffer-based rendering pipeline.

- DrawList - records draw commands
- Batching & State Grouping - minimizes backend calls
- Region-Based Flush - partial screen updates



3. Backend Abstraction Layer

> Want to learn more about the backend abstraction layer? Check out the Backend Abstraction Layer documentation for details on how to use it in your code.

Provides strict decoupling between the core engine and platform-specific implementations.

- AromaBackendABI - entry point binding runtime to platform
- Platform Interface - OS/window/input abstraction
- Graphics Interface - rendering abstraction

This design allows the core to remain platform-agnostic.

4. Platform Implementations

Platform-specific bindings provide:

- Window creation
- Input integration
- System lifecycle hooks

Examples:

- Desktop (GLPS)
- Android
- Embedded (TFT/SPI devices)

5. Graphics Backends

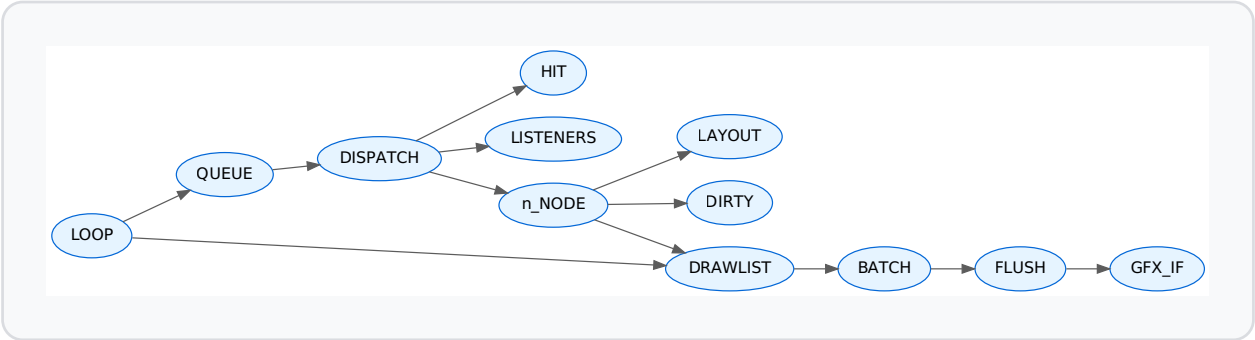
Rendering backends implement the Graphics Interface.

Supported targets:

- OpenGL ES 3
- Vulkan
- TFT/SPI renderer for embedded systems

Each backend receives pre-batched draw commands from the DrawList.

Data Flow Summary



Architectural Principles

- Layered separation of concerns
- Retained-mode scene graph
- Command-buffer rendering
- Backend abstraction
- Embedded-first determinism
- Region-based redraw optimization

Intended Evolution

The architecture supports future expansion including:

- Multi-window compositing
- Threaded rendering pipelines
- GPU resource management layers
- Advanced animation systems
- Custom layout strategies

Core JNI API

Author: AHMED ALI Mohamed Yassine

This section exposes low-level access to the Android runtime. These APIs are intended for advanced use cases such as custom JNI bridging, third-party SDK integration, or manual thread attachment.

All functions are available only when compiling with `__ANDROID__` defined.

`JNIEnv* aroma_android_get_env();`

Returns the `JNIEnv` for the current thread.

Returns

- Pointer to `JNIEnv`
- `NULL` if the thread is not attached to the JVM

Example

```
JNIEnv* env = aroma_android_get_env();
if (!env) {
    // Thread not attached
    return;
}
```

Notes

- Each native thread must be attached to the JVM before using JNI.

- Do not store `JNIEnv*` globally across threads.

`jobject aroma_android_get_activity();`

Returns a global reference to the active Android `Activity`.

Example

```
jobject activity = aroma_android_get_activity();
if (activity) {
    // Use activity in JNI calls
}
```

Notes

- Managed internally by the platform layer.
- Do not delete this reference.

`JavaVM* aroma_android_get_jvm();`

Returns the Java VM instance.

Example

```
JavaVM* jvm = aroma_android_get_jvm();
if (jvm) {
    // Attach worker threads if needed
}
```

Notes

- Useful for background threads requiring JNI access.
- Not required for standard AromaUI usage.

AromaABI Interface

> It is **highly discouraged** to directly interact with the AromaABI in most cases, as it is primarily intended for internal use within the Aroma UI framework. However, understanding the ABI can be beneficial for advanced users who wish to customize or extend the functionality of the framework. The ABI provides a standardized interface for interacting with the graphics and platform backends, allowing for greater flexibility and compatibility across different platforms and graphics APIs.

Overview

The AromaABI is an interface that defines the functions and types necessary for interacting with the graphics and platform backends of the Aroma UI framework. It allows developers to set and get the types of graphics and platform backends being used, as well as retrieve the corresponding interfaces for each backend. This ABI is essential for ensuring compatibility and proper functioning of the Aroma UI across different platforms and graphics APIs.

Functions

- `void (*set_graphics_backend_type)(AromaGraphicsBackendType type)`: Sets the type of graphics backend to be used by the Aroma UI framework.
- `void (*set_platform_backend_type)(AromaPlatformBackendType type)`: Sets the type of platform backend to be used by the Aroma UI framework.
- `AromaGraphicsBackendType (*get_graphics_backend_type)(void)`: Retrieves the current type of graphics backend being used by the Aroma UI framework.
- `AromaPlatformBackendType (*get_platform_backend_type)(void)`: Retrieves the current type of platform backend being used by the Aroma UI framework.

- `AromaGraphicsInterface* (*get_graphics_interface)(void)`: Retrieves a pointer to the graphics interface corresponding to the current graphics backend.
- `AromaPlatformInterface* (*get_platform_interface)(void)`: Retrieves a pointer to the platform interface corresponding to the current platform backend.

Available Backends

- Graphics Backends:
  - `GRAPHICS_BACKEND_GLES3`: OpenGL ES 3.0 graphics backend.
  - `GRAPHICS_BACKEND_VULKAN`: Vulkan graphics backend.
  - `GRAPHICS_BACKEND_TFT_ESPI`: TFT_eSPI graphics backend for embedded systems.
- Platform Backends:
  - `PLATFORM_BACKEND_GLPS`: GLPS platform backend for desktop applications.
  - `PLATFORM_BACKEND_ANDROID`: Android platform backend for mobile applications.
  - `PLATFORM_BACKEND_TFT_ESPI`: TFT_eSPI platform backend for embedded systems.

Usage

To use the AromaABI, developers can include the `aroma_abi.h` header file in their application code. They can then set the desired graphics and platform backends using the provided functions, and retrieve the corresponding interfaces to interact with the backends. For example:

```
#include "aroma_abi.h"

int main() {
    // Set the graphics and platform backends
    aroma_abi.set_graphics_backend_type(GRAPHICS_BACKEND_VULKAN);
    aroma_abi.set_platform_backend_type(PLATFORM_BACKEND_GLPS);

    // Retrieve the graphics and platform interfaces
    AromaGraphicsInterface* graphics_interface = aroma_abi.get_graphics_interface();
    AromaPlatformInterface* platform_interface = aroma_abi.get_platform_interface();

    // Use the interfaces to perform graphics and platform operations
    // ...

    return 0;
}
```

Backend Compatibility Table

> ⚠ Note: The actual implementation of the functions and the behavior of the backends may vary based on the specific platform and graphics API being used. Developers should refer to the documentation for each backend for more details on how to properly utilize them in their applications. It is also discouraged to mix and match graphics and platform backends that are not designed to work together, as this may lead to compatibility issues or unexpected behavior. Always ensure that the chosen graphics and platform backends are compatible with each other and with the target platform for optimal performance and functionality.

Graphics Backend	Platform Backend	Compatibility
Vulkan	GLPS	Yes
Vulkan	Android	Yes
Vulkan	TFT_eSPI	No
OpenGL ES 3.0	GLPS	Yes
OpenGL ES 3.0	Android	Yes
OpenGL ES 3.0	TFT_eSPI	No
TFT_eSPI	GLPS	No
TFT_eSPI	Android	No
TFT_eSPI	TFT_eSPI	Yes

System Services

Author: AHMED ALI Mohamed Yassine

Provides access to Android system services and application storage paths.

`jobject aroma_android_get_system_service(const char* service_name);`

Returns a system service object.

Parameters

- `service_name` - e.g. "vibrator", "wifi", "bluetooth"

Example

```
jobject service = aroma_android_get_system_service("vibrator");
```

Notes

- Returned object is a JNI reference.
- Service availability depends on device and Android version.

`const char* aroma_android_get_internal_path();`

Returns the app's internal storage directory.

Example

```
const char* path = aroma_android_get_internal_path();
printf("Internal: %s\n", path);
```

Notes

- Private to the application.
- No special permissions required.

`const char* aroma_android_get_external_path();`

Returns the app's external storage directory.

Example

```
const char* path = aroma_android_get_external_path();
```

Notes

- May require storage permission depending on API level.
- Availability depends on device storage state.

Device Information

Author: AHMED ALI Mohamed Yassine

Provides runtime access to device-level system information.

`int aroma_android_get_battery_level();`

Returns the current battery percentage.

Returns

- 0-100
- `-1` if unavailable

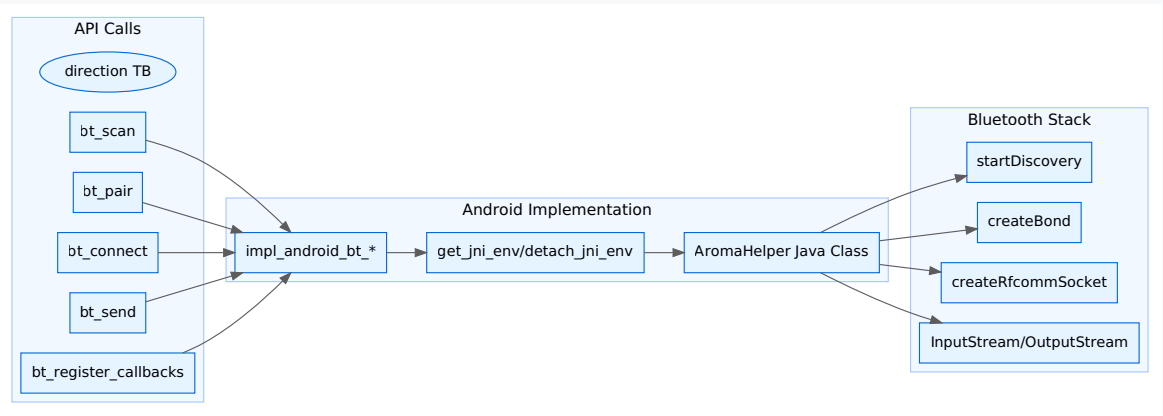
Example

```
int level = aroma_android_get_battery_level();
if (level >= 0) {
    printf("Battery: %d%%\n", level);
}
```

Notes

- May not update in real-time unless refreshed.
- Suitable for dashboards and monitoring apps.

Bluetooth Classic API



This guide provides a structured, comprehensive explanation of the AromaUI Bluetooth Classic API for Android.

The API supports:

Feature	Description
Device scanning	Discover nearby Bluetooth Classic devices
Pairing management	Create and remove device bonds
RFCOMM data communication	Send and receive raw data
Connection state management	Monitor and control active connections
Device type detection	Identify connected device category

Limitations:

Limitation	Status
Bluetooth Classic	Supported
Bluetooth Low Energy	Not supported
GATT	Not supported

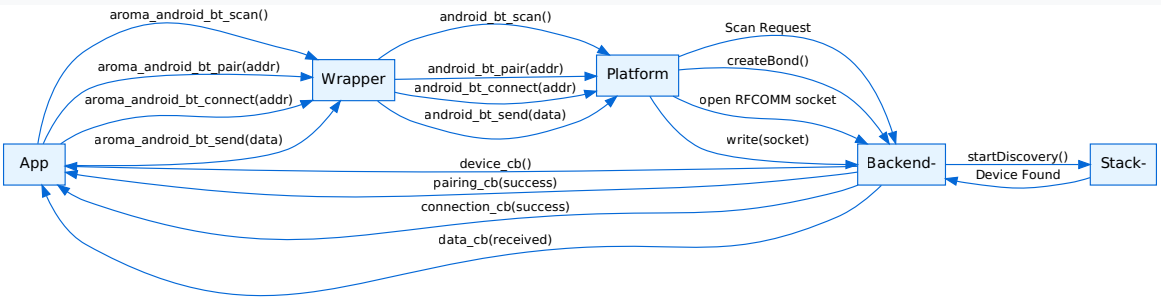
All APIs are valid only when compiling with **ANDROID** defined.

1. Architecture Overview

The Bluetooth layer is exposed through inline wrappers that delegate to AromaPlatformInterface.

The following sequence diagram illustrates a typical Bluetooth workflow:

- Scan for devices
- Pair with device
- Connect using RFCOMM
- Send and receive data



Many operations are asynchronous internally. Design your application around callbacks rather than blocking logic.

2. Android Permissions

Modern Android versions require runtime permissions.

For Android 12 and above:

- android.permission.BLUETOOTH_SCAN
- android.permission.BLUETOOTH_CONNECT
- android.permission.ACCESS_FINE_LOCATION

Check permission:

```
bool granted = aroma_android_check_permission(
    "android.permission.BLUETOOTH_SCAN"
);
```

Request permissions:

```
const char* perms[] = {
    "android.permission.BLUETOOTH_SCAN",
    "android.permission.BLUETOOTH_CONNECT"
};

aroma_android_request_permission(perms, 2);
```

Always verify Bluetooth state before scanning:

```
if (!aroma_android_is_bluetooth_enabled()) {
    // Prompt user to enable Bluetooth
}
```

3. Registering Callbacks

Register all callbacks before scanning or connecting.

```
aroma_android_bt_register_callbacks(
    device_cb,
    scan_finished_cb,
    pairing_cb,
    connection_cb,
    data_cb
);
```

Callback Signatures

Device discovered:

```
void device_cb(const char* addr,
               const char* name,
               int type,
               int rssi);
```

Scan finished:

```
void scan_finished_cb(void);
```

Pairing result:

```
void pairing_cb(bool success,
                const char* addr,
                const char* name);
```

Connection result:

```
void connection_cb(bool success,
                   const char* addr,
                   int mode,
                   int device_type);
```

Data received:

```
void data_cb(const char* data,
             int len);
```

4. Device Scanning

Scan Modes

Constant	Description
AROMA_BT_SCAN_MODE_PAIRED	Scan paired devices only
AROMA_BT_SCAN_MODE_NEW	Scan new unpaired devices only
AROMA_BT_SCAN_MODE_ALL	Scan all devices

Start scanning:

```
aroma_android_bt_scan(
    AROMA_BT_SCAN_MODE_ALL,
    device_cb
);
```

Stop scanning:

```
aroma_android_bt_stop_scan();
```

Device callback example:

```
void device_cb(const char* addr,
               const char* name,
               int type,
               int rssi) {

    printf("Device: %s [%s] RSSI: %d Type: %d\n",
           name, addr, rssi, type);
}
```

5. Getting Paired Devices

```
char addrs[8][18];
char names[8][248];

int count = aroma_android_bt_get_paired(
    addrs,
    names,
    8
);
```

Address format:

XX:XX:XX:XX:XX:XX

Buffers must be preallocated.

6. Pairing Management

Start pairing:

```
bool started = aroma_android_bt_pair(addr);
```

Unpair:

```
aroma_android_bt_unpair(addr);
```

Check bond state:

```
int state = aroma_android_bt_get_pair_state(addr);
```

Bond states:

Constant	Description
AROMA_BT_BOND_NONE	Not bonded
AROMA_BT_BOND_BONDING	Bonding in progress
AROMA_BT_BOND_BONDED	Bonded

7. Connecting to a Device

Default connection:

```
aroma_android_bt_connect(addr);
```

Connection with mode:

```
aroma_android_bt_connect_with_mode(
    addr,
    AROMA_BT_MODE_DATA
);
```

Connection modes:

Constant	Description
AROMA_BT_MODE_DATA	RFCOMM data mode
AROMA_BT_MODE_AUDIO	Audio profile mode
AROMA_BT_MODE_HID	Human Interface Device mode
AROMA_BT_MODE_AUTO	Automatically selected mode

Disconnect:

```
aroma_android_bt_disconnect();
```

Check connection:

```
bool connected = aroma_android_bt_is_connected();
```

8. Sending Data

```
const char* msg = "HELLO";

int sent = aroma_android_bt_send(msg, 5);
```

Returns:

- Number of bytes sent
- Returns negative one on error

Data reception happens through data_cb.

9. Connected Device Information

Device type:

```
int type = aroma_android_bt_get_device_type();
```

Device name:

```
const char* name = aroma_android_bt_get_device_name();
```

Current mode:

```
int mode = aroma_android_bt_get_current_mode();
```

Mode name:

```
const char* modeName = aroma_android_bt_get_mode_name();
```

Wi-Fi API

Author: AHMED ALI Mohamed Yassine

Provides programmatic access to Wi-Fi state and control.

`bool aroma_android_is_wifi_enabled();`

Returns whether Wi-Fi is enabled.

Example

```
if (!aroma_android_is_wifi_enabled()) {
    aroma_android_set_wifi_enabled(true);
}
```

Notes

- Some Android versions restrict direct Wi-Fi toggling.

`void aroma_android_set_wifi_enabled(bool enabled);`

Enables or disables Wi-Fi.

Example

```
aroma_android_set_wifi_enabled(false);
```

Notes

- Behavior varies across Android API levels.
- May require elevated permissions.

UI Utilities

Author: AHMED ALI Mohamed Yassine

Provides lightweight access to common Android UI feedback mechanisms.

`void aroma_android_toast(const char* msg, bool long_duration);`

Displays a Toast message.

Example

```
aroma_android_toast("Operation successful", false);
```

Notes

- Non-blocking UI feedback.
- Avoid excessive usage.

`void aroma_android_open_settings();`

Opens Android system settings.

Example

```
aroma_android_open_settings();
```

Notes

- Useful when users must manually enable permissions.

`void aroma_android_vibrate(int ms);`

Triggers device vibration.

Example

```
aroma_android_vibrate(150);
```

Notes

- Keep vibration short for good UX.

Animation Manager

AromaUI provides a built-in, lightweight animation engine to create smooth and fluid user experiences. Animations can be applied to any `AromaNode` to transition its properties (like position, size, or opacity) over a specified duration.

Animation Types

AromaUI supports several built-in animation types out of the box, defined in the `AromaAnimationType` enum:

- `AROMA_ANIM_SLIDE_X`: Animates the `x` coordinate of the node.
- `AROMA_ANIM_SLIDE_Y`: Animates the `y` coordinate of the node.
- `AROMA_ANIM_SCALE_X`: Animates the `width` of the node.
- `AROMA_ANIM_SCALE_Y`: Animates the `height` of the node.
- `AROMA_ANIM_FADE`: Animates the `opacity` of the node.
- `AROMA_ANIM_CUSTOM`: Allows you to animate an arbitrary value and apply it manually via a callback.
- `AROMA_ANIM_NONE`: No animation.

Easing Functions

Easings define the rate of change of a parameter over time. AromaUI provides the following easing functions (`AromaEasingType`):

- `AROMA_EASE_LINEAR`: Linear constant speed.
- `AROMA_EASE_IN_QUAD`: Starts slow, accelerates.
- `AROMA_EASE_OUT_QUAD`: Starts fast, decelerates.
- `AROMA_EASE_IN_OUT_QUAD`: Starts slow, accelerates, then decelerates.
- `AROMA_EASE_OUT_CUBIC`: Fluid deceleration (default for many UI elements).

- `AROMA_EASE_OUT_BACK`: Overshoots slightly and then settles back.
- `AROMA_EASE_OUT_ELASTIC`: Elastic bounce effect.

Starting an Animation

You can start a basic animation using `aroma_animation_start`.

```
// Example: Slide a card in from the left side of the screen
AromaAnimation* anim = aroma_animation_start(
    card_node,              // Target AromaNode
    AROMA_ANIM_SLIDE_X,     // Type of animation
    -350.0f,                // Start value
    20.0f,                  // End value
    300                     // Duration in milliseconds
);

// Optional: Change the easing from the default
aroma_animation_set_easing(anim, AROMA_EASE_OUT_BACK);
```

Stopping an Animation

If you need to interrupt an animation before it naturally finishes, you can call:

```
aroma_animation_stop(card_node);
```

Custom Animations

If you need to animate a property that isn't covered by the default types (e.g. rotating an element, blending colors, or animating custom widget data), you can use custom animations.

```
void my_custom_anim_cb(AromaNode* target, float current_val, void* user_data) {
    // Apply `current_val` to your data structure
    AromaRect* my_widget_data = (AromaRect*)target->node_widget_ptr;
    // e.g. update a custom offset, angle or size
}

// Start custom animation
AromaAnimation* custom_anim = aroma_animation_start_custom(
    target_node,      // Custom AromaNode
    0.0f,             // Start value
    100.0f,           // End value
    500,              // 500ms duration
    my_custom_anim_cb,// The callback that applies the value
    NULL              // User context data
);
```

Internal workings

The animation engine is driven by the main UI thread. During each frame layout and render pass, the animation manager updates all active animations, computes the eased values, writes them to the node struct or calls the custom callbacks, and automatically invalidates the dirty rects to ensure the screen redraws cleanly.

Widgets Library

Author: AHMED ALI Mohamed Yassine

All widget helper functions return:

```
AromaNode*
```

This represents the widget node inserted into the UI tree.

---

1. Button

Creates a standard clickable button.

```
AromaNode* aroma_ui_button(
    AromaNode* parent,
    const char* text,
    int x, int y, int width, int height,
    bool (*on_click)(AromaNode*, void*),
    void* user_data,
    AromaFont* font
);
```

Parameters: - parent: Parent node - text: Button label - x, y: Position relative to parent - width, height: Dimensions - on_click: Click callback - user_data: Passed to callback - font: Optional font

---

2. Button with Icon

```
AromaNode* aroma_ui_button_with_icon(
    AromaNode* parent,
    const char* text,
    int x, int y, int width, int height,
    bool (*on_click)(AromaNode*, void*),
    void* user_data,
    AromaFont* font,
    const char* icon_code,
    AromaFont* icon_font
);
```

Adds an icon to a standard button.

---

3. Label

```
AromaNode* aroma_ui_label(
    AromaNode* parent,
    const char* text,
    int x, int y,
    AromaLabelStyle style,
    AromaFont* font
);
```

Displays styled text.

---

4. Container

```
AromaNode* aroma_ui_container(
    AromaNode* parent,
    int x, int y, int width, int height,
    AromaLayoutMode layout_mode,
    AromaFlexDirection flex_dir,
    AromaJustifyContent justify,
    AromaAlignItems align
);
```

Generic layout container supporting flex configuration.

---

5. Image

From file:

```
AromaNode* aroma_ui_image(
    AromaNode* parent,
    const char* path,
    int x, int y, int width, int height
);
```

From memory:

```
AromaNode* aroma_ui_image_mem(
    AromaNode* parent,
    unsigned char* data,
    size_t len,
    int x, int y, int width, int height
);
```

---

6. Icon

```
AromaNode* aroma_ui_icon(
    AromaNode* parent,
    const char* icon_code,
    int x, int y, int size,
    uint32_t color,
    AromaFont* font
);
```

Renders an icon using an icon font.

---

7. Checkbox

```
AromaNode* aroma_ui_checkbox(
    AromaNode* parent,
    const char* label,
    int x, int y, int width, int height,
    void (*callback)(bool, void*),
    void* user_data,
    AromaFont* font
);
```

---

8. Radio Button

```
AromaNode* aroma_ui_radiobutton(
    AromaNode* parent,
    const char* label,
    int x, int y, int width, int height,
    int group_id,
    void (*callback)(void*),
    void* user_data,
    AromaFont* font
);
```

Grouped selection control.

---

9. Switch

```
AromaNode* aroma_ui_switch(
    AromaNode* parent,
    int x, int y, int width, int height,
    bool initial_state,
    bool (*on_change)(AromaNode*, void*),
    void* user_data
);
```

Boolean toggle control.

---

10. Slider

```
AromaNode* aroma_ui_slider(
    AromaNode* parent,
    int x, int y, int width, int height,
    int min, int max, int value,
    bool (*on_change)(AromaNode*, void*),
    void* user_data
);
```

Numeric range selector.

---

11. Textbox

```
AromaNode* aroma_ui_textbox(
    AromaNode* parent,
    int x, int y, int width, int height,
    const char* placeholder,
    bool (*on_text_changed)(AromaNode*, const char*, void*),
    void* user_data,
    AromaFont* font
);
```

Text input field.

---

12. Floating Action Button

```
AromaNode* aroma_ui_fab(
    AromaNode* parent,
    int x, int y,
    AromaFABSize size,
    const char* icon_text,
    void (*callback)(void*),
    void* user_data,
    AromaFont* font
);
```

Circular primary action button.

---

13. Icon Button

```
AromaNode* aroma_ui_iconbutton(
    AromaNode* parent,
    const char* icon,
    int x, int y,
    int size,
    AromaIconButtonVariant variant,
    void (*callback)(void*),
    void* user_data,
    AromaFont* font
);
```

Button that displays only an icon.

---

14. Chip

```
AromaNode* aroma_ui_chip(
    AromaNode* parent,
    const char* label,
    int x, int y,
    AromaChipType type,
    void (*callback)(void*),
    void* user_data,
    AromaFont* font
);
```

With icon:

```
AromaNode* aroma_ui_chip_with_icon(
    AromaNode* parent,
    const char* label,
    int x, int y,
    AromaChipType type,
    void (*callback)(void*),
    void* user_data,
    AromaFont* font,
    const char* icon_code,
    AromaFont* icon_font
);
```

Compact selectable element.

---

15. Card

```
AromaNode* aroma_ui_card(
    AromaNode* parent,
    int x, int y, int width, int height,
    AromaCardType type
);
```

Surface container with elevation or outline.

---

16. Progress Bar

```
AromaNode* aroma_ui_progressbar(
    AromaNode* parent,
    int x, int y, int width, int height,
    AromaProgressType type,
    float progress
);
```

Determinate or indeterminate progress indicator.

---

17. Divider

```
AromaNode* aroma_ui_divider(
    AromaNode* parent,
    int x, int y, int length,
    AromaDividerOrientation orientation
);
```

Horizontal or vertical separator.

---

18. List View

```
AromaNode* aroma_ui_listview(
    AromaNode* parent,
    int x, int y, int width, int height,
    void (*callback)(int, void*),
    void* user_data,
    AromaFont* font
);
```

Scrollable list container.

---

19. Menu

```
AromaNode* aroma_ui_menu(
    AromaNode* parent,
    int x, int y,
    AromaFont* font
);
```

Popup or contextual menu.

---

20. Dialog

```
AromaNode* aroma_ui_dialog(
    AromaNode* parent,
    const char* title,
    const char* message,
    int width, int height,
    AromaDialogType type,
    AromaFont* font
);
```

Modal dialog component.

---

21. Tabs

```
AromaNode* aroma_ui_tabs(
    AromaNode* parent,
    int x, int y, int width, int height,
    const char** labels, int count,
    void (*on_change)(AromaNode*, int, void*),
    void* user_data,
    AromaFont* font
);
```

With icons:

```
AromaNode* aroma_ui_tabs_with_icons(
    AromaNode* parent,
    int x, int y, int width, int height,
    const char** labels,
    const char** icons,
    int count,
    void (*on_change)(AromaNode*, int, void*),
    void* user_data,
    AromaFont* font,
    AromaFont* icon_font
);
```

---

22. Snackbar

```
AromaNode* aroma_ui_snackbar(
    AromaNode* parent,
    const char* message,
    int duration_ms,
    AromaFont* font
);
```

Temporary bottom notification.

---

23. Tooltip

```
AromaNode* aroma_ui_tooltip(
    AromaNode* parent,
    const char* text,
    int x, int y,
    AromaTooltipPosition pos,
    AromaFont* font
);
```

Hover hint element.

---

24. Sidebar

```
AromaNode* aroma_ui_sidebar(
    AromaNode* parent,
    int x, int y, int width, int height,
    const char** labels, int count,
    void (*on_select)(AromaNode*, int, void*),
    void* user_data,
    AromaFont* font
);
```

With icons:

```
AromaNode* aroma_ui_sidebar_with_icons(
    AromaNode* parent,
    int x, int y, int width, int height,
    const char** labels,
    const char** icons,
    int count,
    void (*on_select)(AromaNode*, int, void*),
    void* user_data,
    AromaFont* font,
    AromaFont* icon_font
);
```

---

25. Dropdown

```
AromaNode* aroma_ui_dropdown(
    AromaNode* parent,
    int x, int y, int width, int height,
    char** options,
    int option_count,
    void (*on_selection_changed)(int, const char*, void*),
    void* user_data,
    AromaFont* font
);
```

Selectable list popup.

---

26. Debug Overlay

```
AromaNode* aroma_ui_debug_overlay(
    AromaNode* parent,
    int x, int y, int width,
    AromaFont* font
);
```

Displays runtime diagnostics.

---

27. Window Node

```
AromaNode* aroma_ui_window(
    const char* title,
    int width, int height,
    bool fullscreen
);
```

Creates a top level window node.

19. Loading Spinner

```
AromaNode* aroma_ui_loading(
    AromaNode* parent,
    int x, int y, int radius, int thickness,
    uint32_t color
);
```

Creates an animated loading spinner to indicate progress or working states in the application.

Parameters: - `parent`: Parent node - `x`, `y`: Position relative to parent - `radius`: Spinner radius in pixels - `thickness`: Line thickness in pixels - `color`: Spinner color in `0xAARRGGBB` format

---

20. Map

```
AromaNode* aroma_ui_map(
    AromaNode* parent,
    int x, int y, int width, int height
);
```

An interactive map widget with support for zooming, panning, route drawing, and custom markers. Used for navigation and location features.

Parameters: - `parent`: Parent node - `x`, `y`: Position relative to parent - `width`, `height`: Dimensions

Related functions: - `aroma_map_set_center(node, lat, lon)` - Set map center coordinates - `aroma_map_set_zoom(node, zoom_level)` - Set zoom level (e.g. 10 to 18) - `aroma_map_zoom_in(node)`, `aroma_map_zoom_out(node)` - Adjust zoom level safely - `aroma_map_add_marker(node, lat, lon, color)` - Add plain colored marker - `aroma_map_add_popup_marker(node, lat, lon, color, text)` - Add marker with an interactive popup label - `aroma_map_set_route(node, start_lat, start_lon, end_lat, end_lon, color)` - Outline a route path on the map - `aroma_map_clear_markers(node)` - Clear all set markers - `aroma_map_clear_route(node)` - Clear defined route graphic

Data Table

The **Data Table** widget (`AromaTableInternal`) provides a high-level structure for displaying tabular data. It inherently supports headers, scrollable containers (when built using `aroma_ui_table`), resizable columns, selectable rows, and the embedding of child widgets inside specific cells.

Features

- **Scrolling Support**: Automatically places the table within an internally managed scroll container using `aroma_ui_table()`.
- **Zebra Striping**: Alternating row backgrounds enhance readability.
- **Row Selection**: Built-in visual highlighting and click callbacks for selected rows.
- **Cell Widgets**: Easily insert interactive elements (like buttons or toggles) into cells, with the table automatically managing their layout during scrolling and resizing.

Creation

To create a Data Table, typically you will use the `aroma_ui_table` inline function which wraps it in a vertical scroll container:

```
// Create a table with 3 columns
AromaNode *table = aroma_ui_table(
    parent_node,
    x, y, width, height,
    3,
    on_row_selected_cb,
    user_data
);
```

Usage

Headers & Columns

Set up your columns explicitly by specifying their logical widths and text titles.

```
aroma_table_set_col_width(table, 0, 150); // width of col 0 is 150px
aroma_table_set_col_width(table, 1, 200);

aroma_table_set_header(table, 0, "Name");
aroma_table_set_header(table, 1, "Age");
```

Adding Rows

Instead of manually tracking row counts via a custom layout loop, you append a single row and modify its cell contents:

```
int new_row = aroma_table_add_row(table);
aroma_table_set_cell_text(table, new_row, 0, "Alice");
aroma_table_set_cell_text(table, new_row, 1, "28");
```

Cell Widgets

You can embed complex nodes inside cells. The data table handles their coordinate translations and layout updates internally. First, make the `table` the parent of your widget. Then, tell the table which cell should dock it:

```
// Create a button as a child of the table
AromaNode *call_btn = aroma_ui_button(table, "Call", 0, 0, 80, 30, my_callback, NULL, my_fo

// Dock it inside Row 0, Column 2
aroma_table_set_cell_widget(table, 0, 2, call_btn);
```

APIs

- `AromaNode* aroma_table_create(AromaNode* parent, int x, int y, int width, int height, int num_cols)`
- `void aroma_table_set_col_width(AromaNode* table_node, int col_idx, int width)`
- `void aroma_table_set_header(AromaNode* table_node, int col_idx, const char* text)`
- `int aroma_table_add_row(AromaNode* table_node)`
- `void aroma_table_set_cell_text(AromaNode* table_node, int row_idx, int col_idx, const char* text)`
- `void aroma_table_set_cell_widget(AromaNode* table_node, int row_idx, int col_idx, AromaNode* widget)`
- `int aroma_table_get_selected_row(AromaNode* table_node)`
- `void aroma_table_set_callback(AromaNode* table_node, void (*callback)(int row_idx, void* user_data), void* user_data)`
- `void aroma_table_set_font(AromaNode* table_node, AromaFont* font)`

Layout Management

1 Introduction

The layout system computes the screen-space position and dimensions of every widget in the UI tree. It is driven by two orthogonal properties attached to each node: a **self layout type**, which determines how the node positions itself within its parent, and a **container layout mode**, which determines how the node arranges its own children.

Layout is resolved by calling the layout update function on the root node of a window or sub-tree. The engine traverses the tree recursively, writing final pixel coordinates and dimensions into each widget's geometry record.

> **NOTE** All coordinates are in pixels, expressed as signed 32-bit integers. The origin `(0, 0)` is the top-left corner of the parent bounds.

---

2 Layout model

2.1 Self layout

The self layout type controls how a node positions and sizes itself relative to the bounds provided by its parent. It is set using one of the functions described in Section 3. The default type is absolute positioning the node's geometry is left unchanged by the engine.

2.2 Container layout

The container layout mode controls how a node arranges its direct children. It is set using `aroma_node_set_layout_mode()`. The three available modes are described in Section 4.

The two properties are independent. A node may, for example, use anchor self layout to pin itself to the bottom of its parent, while simultaneously acting as a flex container for its own children.

---

3 Self layout types

3.1 Absolute positioning

Default behavior no function call required.

The widget geometry (`x`, `y`, `width`, `height`) is not modified by the layout engine. The application is responsible for setting these values directly at creation time.

3.2 Fill parent

```
aroma_node_set_layout_fill(node);
```

The node is resized and repositioned to exactly match the bounds provided by its parent. All four geometry fields are overwritten on every layout pass.

Field	Resolved value
`x`	`parent_x`
`y`	`parent_y`
`width`	`parent_width`
`height`	`parent_height`

3.3 Center

```
aroma_node_set_layout_center(node);
```

The node is centered within its parent bounds. The application must set `width` and `height` on the widget before the layout pass. If either dimension is zero at layout time, it is set to the corresponding parent dimension before centering is applied.

Field	Resolved value
`x`	`parent_x + (parent_width − width) / 2`
`y`	`parent_y + (parent_height − height) / 2`
`width`	Unchanged (or parent dimension if zero)
`height`	Unchanged (or parent dimension if zero)

3.4 Anchor

```
aroma_node_set_layout_anchor(node, left, top, right, bottom);
```

Pins the node to one or more edges of its parent. Each parameter is a pixel offset from the corresponding edge. Pass `-1` to leave an edge unanchored.

Table 1. Anchor resolution rules

`left`	`right`	Horizontal resolution
≥ 0	−1	`x = parent_x + left`; width unchanged
−1	≥ 0	`x = parent_x + parent_width − right − width`; width unchanged
≥ 0	≥ 0	`x = parent_x + left`; `width = parent_width − left − right`

Table 2. Anchor resolution rules (vertical)

`top`	`bottom`	Vertical resolution
≥ 0	−1	`y = parent_y + top`; height unchanged
−1	≥ 0	`y = parent_y + parent_height − bottom − height`; height unchanged
≥ 0	≥ 0	`y = parent_y + top`; `height = parent_height − top − bottom`

> **NOTE** When both edges on an axis are anchored, the widget's `width` or `height` value is overwritten by the engine. Any value previously set by the application on that axis will be discarded.

Example pin to bottom edge, full width:

```
// left = 0, top = -1 (unanchored), right = 0, bottom = 0
aroma_node_set_layout_anchor(action_bar, 0, -1, 0, 0);
```

Example 8 px inset on all sides:

```
aroma_node_set_layout_anchor(panel, 8, 8, 8, 8);
```

---

4 Container layout modes

4.1 No layout (NONE)

```
aroma_node_set_layout_mode(container, AROMA_LAYOUT_MODE_NONE);
```

No automatic child positioning is performed. Each child uses its own self layout type to determine its position. This is the default mode.

4.2 Flexbox layout

```
aroma_node_set_layout_mode(container, AROMA_LAYOUT_MODE_FLEX);
```

Children are arranged sequentially along a single main axis. The axis direction, main-axis alignment, cross-axis alignment, and inter-child spacing are all configurable. Refer to Sections 5, 6, and 7 for the relevant properties.

Setting the flex direction:

```
aroma_node_set_flex_direction(container, AROMA_FLEX_ROW);     // left-to-right
aroma_node_set_flex_direction(container, AROMA_FLEX_COLUMN);  // top-to-bottom
```

Full flex container configuration:

```
aroma_node_set_layout_mode(container, AROMA_LAYOUT_MODE_FLEX);
aroma_node_set_flex_direction(container, AROMA_FLEX_ROW);
aroma_node_set_justify_content(container, AROMA_JUSTIFY_START);
aroma_node_set_align_items(container, AROMA_ALIGN_CENTER);
aroma_node_set_gap(container, 8);
```

The `aroma_ui_container()` helper combines all of the above into a single call:

```
AromaNode *cont = aroma_ui_container(
    parent,
    x, y, width, height,
    AROMA_LAYOUT_MODE_FLEX,
    AROMA_FLEX_ROW,
    AROMA_JUSTIFY_START,
    AROMA_ALIGN_CENTER
);
```

NOTE Only visible children with valid widget data participate in flex layout. Hidden nodes do not consume space on the main axis.
CAUTION If any child has `flex_grow > 0` and remaining space is available, `justify_content` distribution is **not** applied. The growing children absorb all remaining space. Do not combine `flex_grow` with `justify_content` spacing modes on the same container. Refer to Section 6.

4.3 Grid layout

```
aroma_node_set_layout_mode(container, AROMA_LAYOUT_MODE_GRID);
aroma_node_set_grid_cols(container, 3);
aroma_node_set_grid_rows(container, 2);
aroma_node_set_gap(container, 12);
```

Children are placed into equal-sized cells in row-major order (left-to-right, then top-to-bottom). Every child is resized to exactly fill its assigned cell.

Cell dimensions are computed as follows:

```
cell_width  = (container_width  − (cols − 1) × gap) / cols
cell_height = (container_height − (rows − 1) × gap) / rows
```

> **CAUTION** The grid does not scroll or wrap to additional rows. If the number of children exceeds `cols × rows`, excess children are not positioned. Size the grid to accommodate the maximum expected child count.

---

5 Alignment properties

Alignment properties apply to flex containers only. They have no effect in `AROMA_LAYOUT_MODE_GRID` or `AROMA_LAYOUT_MODE_NONE`.

5.1 justify_content

Controls the distribution of children along the **main axis** when total child size is less than the container size and no child has `flex_grow > 0`.

```
aroma_node_set_justify_content(container, justify);
```

Table 3. justify_content values

Value	Description
`AROMA_JUSTIFY_START`	Children packed toward the start of the main axis. Default.
`AROMA_JUSTIFY_CENTER`	Children centered as a group along the main axis.
`AROMA_JUSTIFY_END`	Children packed toward the end of the main axis.
`AROMA_JUSTIFY_SPACE_BETWEEN`	Children distributed evenly; first and last children are flush with the container edges.
`AROMA_JUSTIFY_SPACE_AROUND`	Children distributed evenly; half the inter-child gap is placed at each end.
`AROMA_JUSTIFY_SPACE_EVENLY`	Children distributed evenly; equal space between all children and at each end.

5.2 align_items

Controls the alignment of children along the **cross axis** (perpendicular to the main axis).

```
aroma_node_set_align_items(container, align);
```

Table 4. align_items values

Value	Description
`AROMA_ALIGN_START`	Children aligned to the start of the cross axis.
`AROMA_ALIGN_CENTER`	Children centered on the cross axis.
`AROMA_ALIGN_END`	Children aligned to the end of the cross axis.
`AROMA_ALIGN_STRETCH`	Children stretched to fill the full cross-axis dimension of the container.

> **NOTE** `AROMA_ALIGN_STRETCH` overwrites the child's cross-axis dimension (`width` for column containers, `height` for row containers) on every layout pass.

---

6 Flex grow

A child node may declare a grow factor to consume remaining main-axis space after fixed-size children and gaps have been accounted for.

```
aroma_node_set_flex_grow(child, grow_factor);
```

When multiple children have `flex_grow > 0`, remaining space is distributed among them in proportion to their respective grow factors.

Condition	Result
All children have `flex_grow = 0`	`justify_content` governs remaining space distribution
One or more children have `flex_grow > 0`	Remaining space is distributed proportionally; `justify_content` has no effect

> **CAUTION** A `flex_grow` value of `0.0f` (the default) means the child is fixed-size and does not participate in space distribution.

Example fixed sidebar with a growing content area:

```
// Sidebar: fixed width, no growth
aroma_node_set_flex_grow(sidebar, 0.0f);

// Content: grows to fill all remaining width
aroma_node_set_flex_grow(content, 1.0f);
```

Example toolbar with a flexible spacer:

```
AromaNode *toolbar = aroma_ui_container(
    root, 0, 0, 800, 48,
    AROMA_LAYOUT_MODE_FLEX, AROMA_FLEX_ROW,
    AROMA_JUSTIFY_START, AROMA_ALIGN_CENTER
);
aroma_node_set_gap(toolbar, 8);

AromaNode *logo   = aroma_ui_label(toolbar, "MyApp", 0, 0, LABEL_STYLE_TITLE, font);
AromaNode *spacer = aroma_ui_container(toolbar, 0, 0, 0, 0,
                        AROMA_LAYOUT_MODE_NONE, 0, 0, 0);
aroma_node_set_flex_grow(spacer, 1.0f);

AromaNode *btn    = aroma_ui_button(toolbar, "Settings", 0, 0, 120, 36,
                        on_settings, NULL, font);
```

7 Gap

Sets the uniform spacing in pixels between consecutive children in a flex or grid container.

```
aroma_node_set_gap(container, gap_px);
```

Gap is applied **between** children only. No space is added before the first child or after the last child.

8 Scrollable containers

`aroma_ui_scrollable_container()` creates a container whose children may extend beyond the visible viewport. Scroll position, clipping, and scrollbar rendering are handled internally.

```
AromaNode *scroll = aroma_ui_scrollable_container(
    parent,
    x, y,
    viewport_width, viewport_height,
    AROMA_SCROLL_VERTICAL
);
```

The content area is measured automatically after each layout pass. Scrolling is activated only when the total child extent exceeds the viewport dimension in the configured scroll direction.

> **NOTE** The default child arrangement for a scrollable container is a vertical flex column (`AROMA_JUSTIFY_START`, `AROMA_ALIGN_CENTER`). Child layout

> properties may be overridden after creation if a different arrangement is required.

---

9 API summary

Table 5. Self layout functions

Function	Description
`aroma_node_set_layout_fill(node)`	Node fills parent bounds completely.
`aroma_node_set_layout_center(node)`	Node is centered within parent bounds.
`aroma_node_set_layout_anchor(node, left, top, right, bottom)`	Node is pinned to one or more parent edges.

Table 6. Container layout functions

Function	Description
`aroma_node_set_layout_mode(node, mode)`	Sets the child layout strategy (`NONE`, `FLEX`, `GRID`).
`aroma_node_set_flex_direction(node, dir)`	Sets flex axis direction (`ROW` or `COLUMN`).
`aroma_node_set_justify_content(node, justify)`	Sets main-axis alignment.
`aroma_node_set_align_items(node, align)`	Sets cross-axis alignment.
`aroma_node_set_flex_grow(node, grow)`	Sets grow factor for a flex child.
`aroma_node_set_gap(node, gap)`	Sets inter-child spacing in pixels.
`aroma_node_set_grid_cols(node, cols)`	Sets the number of grid columns.
`aroma_node_set_grid_rows(node, rows)`	Sets the number of grid rows.

Table 7. Convenience creation helpers

Function	Description
`aroma_ui_window(title, w, h, fullscreen)`	Creates a root window node.
`aroma_ui_container(parent, x, y, w, h, mode, dir, justify, align)`	Creates a container with layout configured in one call.
`aroma_ui_scrollable_container(parent, x, y, w, h, direction)`	Creates a scrollable container with auto content sizing.

10 Code examples

10.1 Centered card within a full-screen background

```
AromaNode *root = aroma_ui_window("Application", 800, 600, false);

AromaNode *bg = aroma_ui_container(
    root, 0, 0, 800, 600,
    AROMA_LAYOUT_MODE_FLEX, AROMA_FLEX_COLUMN,
    AROMA_JUSTIFY_CENTER, AROMA_ALIGN_CENTER
);
aroma_node_set_layout_fill(bg);

AromaNode *card = aroma_ui_card(bg, 0, 0, 320, 200, AROMA_CARD_ELEVATED);
```

10.2 Two-panel layout: fixed sidebar and growing content area

```
AromaNode *root_layout = aroma_ui_container(
    root, 0, 0, 800, 600,
    AROMA_LAYOUT_MODE_FLEX, AROMA_FLEX_ROW,
    AROMA_JUSTIFY_START, AROMA_ALIGN_STRETCH
);
aroma_node_set_layout_fill(root_layout);

AromaNode *sidebar = aroma_ui_container(
    root_layout, 0, 0, 200, 0,
    AROMA_LAYOUT_MODE_NONE, 0, 0, 0
);

AromaNode *content = aroma_ui_container(
    root_layout, 0, 0, 0, 0,
    AROMA_LAYOUT_MODE_FLEX, AROMA_FLEX_COLUMN,
    AROMA_JUSTIFY_START, AROMA_ALIGN_STRETCH
);
aroma_node_set_flex_grow(content, 1.0f);
```

10.3 Bottom-anchored action bar

```
AromaNode *bar = aroma_ui_container(
    root, 0, 0, 800, 56,
    AROMA_LAYOUT_MODE_FLEX, AROMA_FLEX_ROW,
    AROMA_JUSTIFY_END, AROMA_ALIGN_CENTER
);
aroma_node_set_layout_anchor(bar, 0, -1, 0, 0);
aroma_node_set_gap(bar, 8);

aroma_ui_button(bar, "Cancel",  0, 0, 100, 36, on_cancel,  NULL, font);
aroma_ui_button(bar, "Confirm", 0, 0, 100, 36, on_confirm, NULL, font);
```

10.4 3 × 2 uniform card grid

```
AromaNode *grid = aroma_ui_container(
    root, 16, 16, 768, 400,
    AROMA_LAYOUT_MODE_GRID, 0, 0, 0
);
aroma_node_set_grid_cols(grid, 3);
aroma_node_set_grid_rows(grid, 2);
aroma_node_set_gap(grid, 12);

for (int i = 0; i < 6; i++) {
    aroma_ui_card(grid, 0, 0, 0, 0, AROMA_CARD_OUTLINED);
}
```

10.5 Vertically scrollable list

```
AromaNode *list = aroma_ui_listview(
    root,
    0, 56,      /* x, y  offset below a 56 px toolbar   */
    400, 544,   /* width, height  viewport dimensions    */
    on_item_click, NULL, font
);

for (int i = 0; i < 50; i++) {
    char label[32];
    snprintf(label, sizeof(label), "Item %d", i);
    aroma_listview_add_item(list, label);
}
```

DPI Conversion

1. Overview

The AromaUI DPI System provides density-aware utilities for Android builds.
It ensures consistent layout scaling across different screen densities, physical sizes, and user font preferences.

This subsystem allows you to:

- Query density metrics
- Convert between DP, PX, and SP units
- Detect physical screen size
- Retrieve available window dimensions
- Respect user font scaling
- Access true hardware DPI values

2. Core Density Concepts

Density (Scale Factor)

Density represents logical scaling relative to the baseline **160 DPI (mdpi)**.

DPI Category	Density	Example DPI
mdpi	1.0	160
hdpi	1.5	240
xhdpi	2.0	320
xxhdpi	3.0	480
xxxhdpi	4.0	640

Retrieve density scale factor:

```
float density = aroma_android_get_density();
```

Retrieve raw density DPI:

```
int dpi = aroma_android_get_density_dpi();
```

3. Text Scaling (Scaled Density)

Scaled density accounts for the user's font size setting.

Use for text scaling (SP units).

```
float scaledDensity = aroma_android_get_scaled_density();
```

Never use plain density for text scaling.

4. Unit Conversions

DP -> PX

Density-independent pixels to physical pixels.

```
int px = aroma_android_dp_to_px(16);
```

Conceptual formula:

```
px = dp × density
```

PX -> DP

```
int dp = aroma_android_px_to_dp(48);
```

SP -> PX (Text)

```
int px = aroma_android_sp_to_px(14);
```

Conceptual formula:

```
px = sp × scaledDensity
```

PX -> SP

```
int sp = aroma_android_px_to_sp(28);
```

5. Window Size (DP Units)

Retrieve available layout space excluding system UI (status bar, navigation bar).

```
int width_dp;
int height_dp;

aroma_android_get_available_size_dp(&width_dp, &height_dp);
```

This should be used for responsive layout logic instead of raw pixel values.

6. Physical Screen Size

Width and Height in Inches

```
float width_in;
float height_in;

aroma_android_get_screen_size_inches(&width_in, &height_in);
```

Diagonal Size in Inches

```
float diagonal = aroma_android_get_screen_diagonal_inches();
```

Useful for differentiating:

- Phone
- Tablet
- Large tablet
- Embedded display

7. Screen Size Category

Returns a logical size classification string:

```
const char* category = aroma_android_get_screen_size_category();
```

Possible values:

Value	Meaning
small	Small phone
normal	Standard phone
large	Phablet / small tab
xlarge	Tablet
xxlarge	Large tablet / TV

8. True Physical DPI

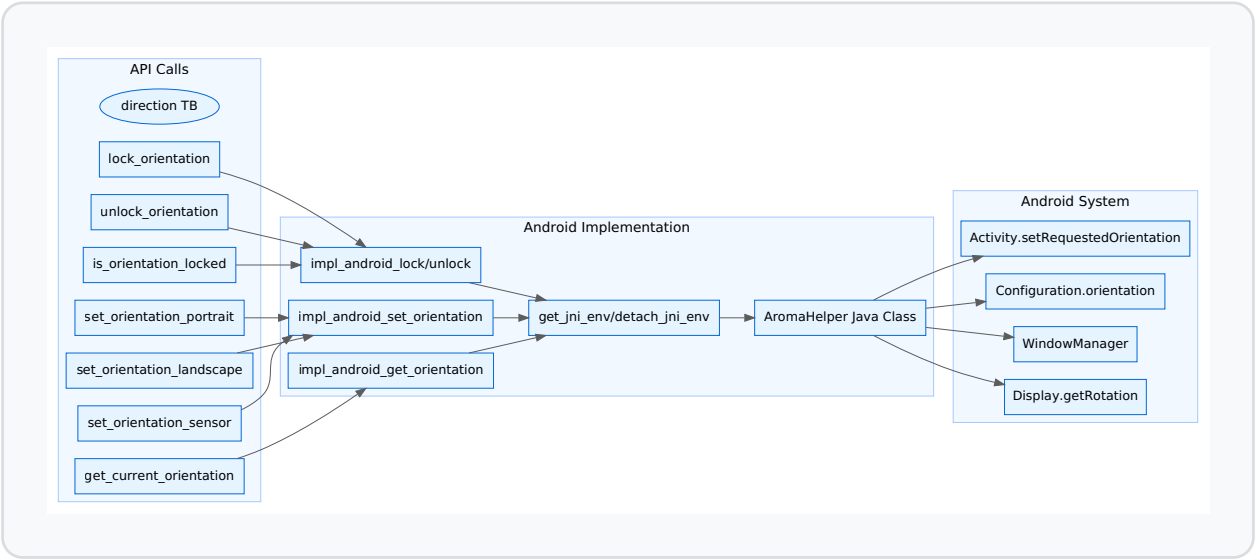
Retrieve hardware DPI along each axis:

```
float xdpi = aroma_android_get_xdpi();
float ydpi = aroma_android_get_ydpi();
```

These values represent the actual physical dots-per-inch of the panel and may differ from logical density.

Orientation API

This guide provides a structured, comprehensive explanation of the AromaUI Screen Orientation Control API for Android.



API Capabilities

Feature	Description
Orientation locking	Prevent screen rotation
Orientation unlocking	Allow sensor-based rotation
Force portrait	Lock screen to portrait mode
Force landscape	Lock screen to landscape mode
Sensor-based rotation	Enable automatic rotation
Current orientation query	Get device orientation state
Orientation lock status	Check if orientation is locked

Limitations

Limitation	Status
Portrait mode	Supported
Landscape mode	Supported
Reverse portrait	Not directly
Reverse landscape	Not directly
Sensor-based	Supported
User-sensor	Not directly

> ⚠ All APIs are valid only when compiling with `ANDROID` defined.

1. Android Manifest Configuration

To use orientation control, ensure your Android manifest includes appropriate configuration:

```
<activity
    android:name=".MainActivity"
    android:configChanges="orientation|screenSize"
    android:screenOrientation="unspecified">
</activity>
```

The `configChanges` attribute allows your app to handle orientation changes manually rather than letting Android recreate the activity.

2. Basic Orientation Control

Lock Current Orientation

Prevent the screen from rotating based on sensor input:

```
aroma_android_lock_orientation();
```

This locks the screen to whatever orientation the device is currently in (portrait or landscape).

Unlock Orientation

Allow the screen to auto-rotate based on device sensors:

```
aroma_android_unlock_orientation();
```

This returns the device to sensor-based rotation behavior.

3. Forcing Specific Orientations

Force Portrait Mode

Lock the screen to portrait orientation regardless of device position:

```
aroma_android_set_orientation_portrait();
```

This forces the display to portrait mode (taller than wide).

Force Landscape Mode

Lock the screen to landscape orientation regardless of device position:

```
aroma_android_set_orientation_landscape();
```

This forces the display to landscape mode (wider than tall).

Enable Sensor-Based Rotation

Set the orientation to follow device sensors:

```
aroma_android_set_orientation_sensor();
```

This is equivalent to unlocking orientation but explicitly sets sensor mode.

4. Querying Orientation State

Get Current Orientation

Retrieve the current screen orientation:

```
int orientation = aroma_android_get_current_orientation();

if (orientation == 1) {
    // Portrait mode
} else if (orientation == 2) {
    // Landscape mode
}
```

Return Values

Value	Description
1	Portrait orientation
2	Landscape orientation
-1	Unknown/unavailable

Check if Orientation is Locked

Determine if orientation locking is currently active:

```
bool is_locked = aroma_android_is_orientation_locked();

if (is_locked) {
    // Orientation is locked to a specific mode
} else {
    // Orientation can auto-rotate
}
```

5. Complete Usage Example

```
#include "aroma_android.h"

// Example: Toggle between portrait and landscape
void toggle_orientation(void) {
    int current = aroma_android_get_current_orientation();

    if (current == 1) {
        // Currently in portrait, switch to landscape
        aroma_android_set_orientation_landscape();
        printf("Switched to landscape mode\n");
    } else if (current == 2) {
        // Currently in landscape, switch to portrait
        aroma_android_set_orientation_portrait();
        printf("Switched to portrait mode\n");
    }
}

// Example: Lock/unlock orientation
void handle_orientation_lock(bool should_lock) {
    if (should_lock) {
        aroma_android_lock_orientation();
        printf("Orientation locked\n");
    } else {
        aroma_android_unlock_orientation();
        printf("Orientation unlocked\n");
    }
}

// Example: Initialize with desired orientation
void initialize_with_orientation(void) {
    // Force portrait mode at startup
    aroma_android_set_orientation_portrait();

    // Later, allow rotation
    // aroma_android_set_orientation_sensor();
}

// Example: Respond to device state
void on_device_rotated(void) {
    if (!aroma_android_is_orientation_locked()) {
        int orientation = aroma_android_get_current_orientation();
```

```
        switch(orientation) {
            case 1:
                // Adapt UI for portrait
                break;
            case 2:
                // Adapt UI for landscape
                break;
        }
    }
}
```

6. Best Practices

When to Lock Orientation

Use Case	Recommendation
Video playback	Lock to landscape
Document reading	Allow auto-rotation
Games	Lock to preferred orientation
Camera apps	Lock to landscape
E-books	Allow portrait/landscape
Media viewers	Follow sensor

Handling Configuration Changes

When your app handles orientation changes manually:

```
// In your Android activity's onCreate or native code
aroma_android_set_orientation_portrait();

// The layout will update through aroma_refresh_layout()
```

Performance Considerations

- Orientation changes trigger surface recreation
- Layout recalculation occurs automatically
- Consider debouncing rapid orientation changes
- Cache orientation-dependent resources

7. Error Handling

```
// Always check if orientation functions are available
if (!aroma_android_get_current_orientation) {
    printf("Orientation API not available\n");
    return;
}

// Check return values where applicable
int orientation = aroma_android_get_current_orientation();
if (orientation < 0) {
    printf("Failed to get orientation\n");
}

// Verify lock state before assuming
if (aroma_android_is_orientation_locked()) {
    // Safe to assume fixed orientation
}
```

Theming System

A Baked-In set of themes is included in AromaUI, which can be used as-is or customized to fit the needs of your application. The available themes include:

Theme Name	Description
Default Theme	A balanced default theme suitable for most applications.
High Contrast Theme	A theme with high contrast colors for improved readability and accessibility.
Material Theme Presets	A set of themes based on the Material Design color palette, including presets for purple, blue, teal, green, orange, and pink.
Black OLED Theme	A theme with darker colors for a more subdued and modern look, suitable for low-light environments.
Material Dark Theme Presets	A set of dark themes based on the Material Design color palette, including presets for purple, blue, teal, green, orange, and pink.

Customization

In addition to the built-in themes, developers can create their own custom themes by defining their own color palettes, spacing, typography, and other style properties.

- The `aroma_theme_create_custom` function allows developers to create a custom theme by providing their own values for the various theme properties.
- Developers can also modify the global theme using the `aroma_theme_set_global` function, which will affect all components that use the global theme for styling.

- The `aroma_style_apply_theme_colors` function can be used to apply the colors from a theme to a specific style, allowing for more granular control over the appearance of individual components.

Color Manipulation

The theming system also includes utility functions for manipulating colors, such as adjusting brightness, blending colors, and converting between different color formats. These functions can be useful for creating dynamic themes or for implementing features such as dark mode or user-customizable themes.

For example, the `aroma_color_adjust` function can be used to adjust the brightness of a color by a specified factor, while the `aroma_color_blend` function can be used to blend two colors together based on a specified blend factor. The `aroma_color_rgb` and `aroma_color_rgba` functions can be used to create colors from RGB or RGBA values, while the `aroma_color_extract_rgb` function can be used to extract the RGB components from a color value.

Shadows

The theming system also includes support for shadows, which can be used to add depth and visual interest to UI components. - The `AromaShadow` struct defines the properties of a shadow, including the blur radius, offset, color, and opacity.

- The `aroma_shadow_create_*` functions provide predefined shadow styles, while the `aroma_shadow_create_custom` function allows developers to create their own custom shadow styles.

- The `aroma_style_apply_shadow` function can be used to apply a shadow to a specific style, allowing for more control over the appearance of individual components. Shadows can be particularly effective when used in combination with themes, as they can help to create a sense of hierarchy and focus within the UI.

For example, a component with a deeper shadow may be perceived as being more important or interactive than a component with a softer shadow. Additionally, shadows can be used to create a sense of depth and layering within the UI, which can enhance the overall visual appeal and user experience of the application.

Camera API

Author: AHMED ALI Mohamed Yassine

Provides access to Android media intents for capturing and selecting images.

`void aroma_android_launch_camera();`

Launches the default camera application.

Example

```
aroma_android_launch_camera();
```

Notes

- Requires camera permission.
- Result must be handled on the Java side.

`void aroma_android_launch_gallery();`

Launches image picker.

Example

```
aroma_android_launch_gallery();
```

Notes

- Used for selecting existing media.
- Result is delivered asynchronously.

Font Module

Overview

The **Aroma Font Module** provides functionality for loading, managing, and querying font data within AromaUI. It supports loading fonts from both file systems and memory buffers, and exposes essential typography metrics such as line height, ascender, descender, and text width.

This module is designed to be lightweight, flexible, and backend agnostic, making it suitable for desktop, mobile, and embedded environments.

Header

```
#include "aroma_font.h"
```

Data Types

`AromaFont`

```
typedef struct AromaFont AromaFont;
```

An opaque structure representing a loaded font instance. The internal implementation is hidden and may vary depending on the backend (for example FreeType).

API Reference

Font Creation

`aroma_font_create`

```
AromaFont* aroma_font_create(const char* font_path, int size_px);
```

Loads a font from a file on disk.

Parameters:

- `font_path`: Path to the font file (for example `.ttf`, `.otf`)
- `size_px`: Desired font size in pixels

Returns:

- Pointer to `AromaFont` on success
- `NULL` on failure

`aroma_font_create_from_memory`

```
AromaFont* aroma_font_create_from_memory(const unsigned char* data, unsigned int data_len,
```

Loads a font from a memory buffer.

Parameters:

- `data`: Pointer to raw font data
- `data_len`: Size of the data buffer in bytes
- `size_px`: Desired font size in pixels

Returns:

- Pointer to `AromaFont` on success

- `NULL` on failure

> AromaUI includes embedded font data for the Ubuntu font family, which can be loaded using `aroma_font_create_from_memory` with the provided `aroma_ubuntu_ttf` and `aroma_ubuntu_ttf_len` variables.

Font Destruction

`aroma_font_destroy`

```
void aroma_font_destroy(AromaFont* font);
```

Frees all resources associated with a font.

Parameters:

- `font`: Font instance to destroy

Font Metrics

`aroma_font_get_line_height`

```
int aroma_font_get_line_height(AromaFont* font);
```

Returns the total height of a line of text.

`aroma_font_get_ascender`

```
int aroma_font_get_ascender(AromaFont* font);
```

Returns the ascender height (distance above the baseline).

`aroma_font_get_descender`

```
int aroma_font_get_descender(AromaFont* font);
```

Returns the descender height (distance below the baseline).

`aroma_font_get_px_size`

```
int aroma_font_get_px_size(AromaFont* font);
```

Returns the configured pixel size of the font.

Text Measurement

`aroma_font_get_line_width`

```
int aroma_font_get_line_width(AromaFont* font, const char* text);
```

Calculates the width of a string when rendered with the font.

Parameters:

- `font`: Font instance
- `text`: Null terminated string

Returns:

- Width in pixels

Low Level Access

`aroma_font_get_face`

```
void* aroma_font_get_face(AromaFont* font);
```

Provides access to the underlying font face object (for example `FT_Face` in FreeType).

Use Cases:

- Advanced glyph rendering
- Custom text shaping
- Integration with external rendering systems

Returns:

- Pointer to backend specific object
- `NULL` if unavailable

Usage Example

```
#include "aroma_font.h"
#include <stdio.h>

int main() {
    AromaFont* font = aroma_font_create("assets/Roboto-Regular.ttf", 16);
    if (!font) {
        printf("Failed to load font\n");
        return 1;
    }

    int width = aroma_font_get_line_width(font, "Hello, AromaUI!");
    int height = aroma_font_get_line_height(font);

    printf("Text width: %d px\n", width);
    printf("Line height: %d px\n", height);

    aroma_font_destroy(font);
    return 0;
}
```

Notes

- Always call `aroma_font_destroy` to prevent memory leaks.
- Font size is specified in pixels, not points.
- Behavior of `aroma_font_get_face` depends on the backend implementation.
- Text measurement assumes simple layout unless the backend provides advanced shaping.

Preferences API

The **Preferences API** provides persistent key-value storage for user settings and application configuration. It offers a lightweight abstraction over platform-specific storage systems.

On Android, the implementation maps directly to **SharedPreferences**, while the public API remains consistent through the **AromaPlatformInterface**.

Features

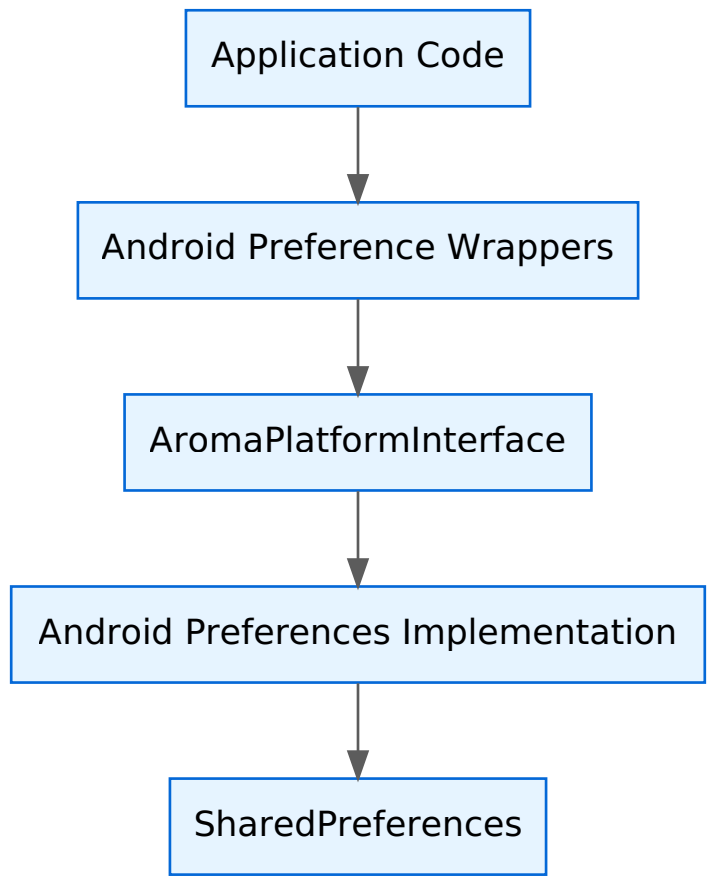
- Persistent key-value storage
- Lightweight and fast access
- Platform abstraction layer
- Type-safe preference access
- Default value fallback when a key does not exist

Supported types:

- `string`
- `int`
- `float`
- `bool`
- `long`

Architecture

The Preferences system is implemented through the platform abstraction layer. Application calls are routed through the platform interface and handled by the platform-specific implementation.



Components

Android Preference Wrappers Inline helper functions exposed in the Aroma SDK that provide type-safe access to preferences.

AromaPlatformInterface Platform abstraction layer that allows Aroma to call platform-specific implementations without coupling the core framework to Android APIs.

Android Preferences Implementation Implements preference storage using Android's native storage system.

SharedPreferences Android's built-in persistent key-value storage used to store application settings.

Typed Preferences API

The Android preferences layer exposes **type-safe accessors** for storing and retrieving values. These functions internally forward calls through the `AromaPlatformInterface`, which connects the core framework to the Android implementation.

All getters support a **default value**, which is returned when the key does not exist or when the platform interface is unavailable.

String Preferences

```
const char* aroma_android_get_preference_string(const char* key, const char* default_value)
void aroma_android_set_preference_string(const char* key, const char* value);
```

Example:

```
aroma_android_set_preference_string("theme", "dark");

const char* theme = aroma_android_get_preference_string("theme", "light");
```

Integer Preferences

```
int aroma_android_get_preference_int(const char* key, int default_value);
void aroma_android_set_preference_int(const char* key, int value);
```

Example:

```
aroma_android_set_preference_int("volume", 80);

int volume = aroma_android_get_preference_int("volume", 50);
```

Float Preferences

```
float aroma_android_get_preference_float(const char* key, float default_value);
void aroma_android_set_preference_float(const char* key, float value);
```

Example:

```
aroma_android_set_preference_float("ui_scale", 1.25f);

float scale = aroma_android_get_preference_float("ui_scale", 1.0f);
```

Boolean Preferences

```
bool aroma_android_get_preference_bool(const char* key, bool default_value);
void aroma_android_set_preference_bool(const char* key, bool value);
```

Example:

```
aroma_android_set_preference_bool("notifications_enabled", true);

bool enabled = aroma_android_get_preference_bool("notifications_enabled", false);
```

Long Preferences

```
long aroma_android_get_preference_long(const char* key, long default_value);
void aroma_android_set_preference_long(const char* key, long value);
```

Example:

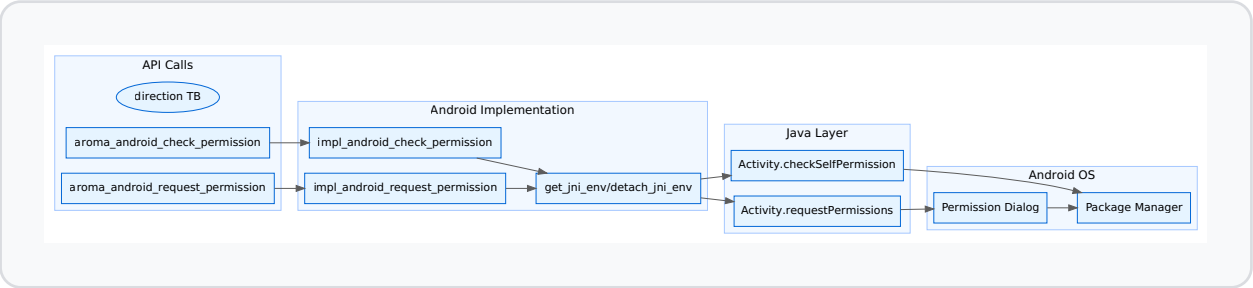
```
aroma_android_set_preference_long("last_sync", 1710000000);

long timestamp = aroma_android_get_preference_long("last_sync", 0);
```

Implementation Notes

- The functions are implemented as `static inline` **wrappers**.
- They forward calls to the `AromaPlatformInterface`.
- If the platform implementation is not available, the **default value is returned**.
- Preference values persist across application restarts.
- Returned strings are **owned by the platform implementation and should not be freed by the caller**.

Permissions API

The Permissions API provides runtime permission management in compliance with Android's security model. It supports permission grouping, Android version-specific behavior, and asynchronous permission handling.



1. Permission Checking

```
bool aroma_android_check_permission(const char* permission_name);
```

Checks whether a specific permission has been granted to the application.

Parameters

Parameter	Description	Example
`permission_name`	Fully qualified Android permission string	`"android.permission.CAMERA"`

Returns

Value	Description
`true`	Permission is granted
`false`	Permission is denied or not yet requested

Example

```
if (!aroma_android_check_permission("android.permission.CAMERA")) {
    const char* permissions[] = {"android.permission.CAMERA"};
    aroma_android_request_permissions(permissions, 1);
}
```

2. Permission Requesting

`void aroma_android_request_permissions(const char** permissions, int permCount);`

Requests one or more permissions from the user.

Parameters

Parameter	Description	Example
`permissions`	Array of permission strings	`{ "android.permission.CAMERA", "android.permission.RECORD_AUDIO" }`
`permCount`	Number of permissions in the array	`2`

Example

```
// Request camera and microphone permissions
const char* permissions[] = {
    "android.permission.CAMERA",
    "android.permission.RECORD_AUDIO"
};

aroma_android_request_permissions(permissions, 2);
```

3. Android Version Behavior

Android Version	Behavior
API < 23	Permissions granted at install time. Function returns immediately.
API 23+	Runtime permission dialog(s) shown to user.
API 31+	Automatically handles Bluetooth-Location dependency.

4. Automatic Permission Grouping

Permissions are grouped to comply with Android system requirements.

Group	Permissions
NEARBY_DEVICES	BLUETOOTH_SCAN, BLUETOOTH_ADVERTISE, BLUETOOTH_CONNECT
LOCATION	ACCESS_FINE_LOCATION, ACCESS_COARSE_LOCATION, ACCESS_BACKGROUND_LOCATION
NOTIFICATIONS	POST_NOTIFICATIONS
STORAGE	READ_EXTERNAL_STORAGE, WRITE_EXTERNAL_STORAGE, READ_MEDIA_IMAGES, READ_MEDIA_VIDEO, READ_MEDIA_AUDIO
CAMERA	CAMERA
MICROPHONE	RECORD_AUDIO

5. Common Permission Strings

Permission	String Constant
Camera	android.permission.CAMERA
Fine Location	android.permission.ACCESS_FINE_LOCATION
Coarse Location	android.permission.ACCESS_COARSE_LOCATION
Bluetooth Scan	android.permission.BLUETOOTH_SCAN
Bluetooth Connect	android.permission.BLUETOOTH_CONNECT
Record Audio	android.permission.RECORD_AUDIO
Read Storage	android.permission.READ_EXTERNAL_STORAGE
Write Storage	android.permission.WRITE_EXTERNAL_STORAGE
Post Notifications	android.permission.POST_NOTIFICATIONS

Building & Deployment

This document describes the complete deployment workflow for AromaUI projects using the `aroma` CLI. It focuses strictly on building, signing, packaging, and distributing applications for Linux and Android.

1. Build Targets Overview

AromaUI supports deployment to:

- Linux (native executable)
- Android Debug APK (testing)
- Android Release APK (signed, production)
- Android App Bundle (AAB, Play Store)

All deployment commands must be executed from the root of your Aroma project.

Linux Deployment

Build Linux Binary

```
aroma build linux
```

This command:

- Creates a `build/` directory
- Runs CMake configuration
- Compiles using Make

Output executable:

```
build/<project_name>
```

Run Linux Build

```
aroma run linux
```

This builds (if needed) and launches the executable.

For distribution, package the binary along with any required shared libraries.

Android Deployment

Android deployment supports debug and release builds.

Debug Build (Testing)

Build Debug APK

```
aroma build android
```

Output:

```
android/app/build/outputs/apk/debug/app-debug.apk
```

Install on Connected Device

```
aroma run android
```

Requirements:

- USB debugging enabled
- `adb devices` shows a connected device

Run in Emulator

```
aroma run android --emu
```

This will:

- Create an AVD if needed
- Start the emulator
- Wait for boot completion
- Install debug APK
- Launch the application

Release Deployment (Production)

Release builds require a signing keystore.

Step 1: Configure Signing

Run:

```
aroma sign
```

This process:

- Creates `android/keystore/release.keystore`
- Generates `android/keystore.properties`
- Updates `app/build.gradle` with signing configuration

The signing configuration is automatically connected to the Gradle `release` build type.

Important:

- Keep the keystore file secure
- Do not lose the password
- Do not commit the keystore to public repositories

Step 2: Build Release APK

```
aroma build android --release
```

Signed APK output:

```
android/app/build/outputs/apk/release/app-release.apk
```

If unsigned, the output will show:

```
app-release-unsigned.apk
```

Unsigned APKs cannot be installed or distributed.

Step 3: Build Android App Bundle (AAB)

For Google Play submission:

```
aroma build android --release --aab
```

Output:

```
android/app/build/outputs/bundle/release/app-release.aab
```

The AAB format is required for Play Store uploads.

Manual Installation of Release APK

To install a signed release APK manually:

```
adb install -r android/app/build/outputs/apk/release/app-release.apk
```

If upgrading an existing version, ensure the keystore matches the previously installed version.

Continuous Deployment Recommendations

For production workflows:

- Store keystore securely (offline backup)
- Use CI to run:

```
aroma build android --release --aab
```

- Archive generated AABs per version
- Tag releases in version control

Play Store Deployment Checklist

Before uploading to Google Play:

- Version code incremented
- Version name updated
- Release build signed
- AAB generated
- App tested on physical device
- Debug logs removed
- Min SDK and Target SDK verified

Upload file:

```
app-release.aab
```

Troubleshooting Deployment

Keystore Password Incorrect

Verify with:

```
keytool -list -keystore release.keystore
```

If the password is wrong, you must recreate signing and cannot update existing published apps.

No Device Detected

Check:

```
adb devices
```

Restart adb if needed:

```
adb kill-server
adb start-server
```

SDK Not Found

Run:

```
aroma doctor
```

Or install manually:

```
aroma install-sdk
```

Deployment Command Summary

Debug APK:

```
aroma build android
```

Release APK:

```
aroma build android --release
```

Release AAB:

```
aroma build android --release --aab
```

Run on device:

```
aroma run android
```

Run in emulator:

```
aroma run android --emu
```

Linux build:

```
aroma build linux
```

Linux run:

```
aroma run linux
```

Logging System

The logging system in AromaUI is designed to provide developers with a flexible and efficient way to log messages, errors, and other information during the development and debugging process. The logging system supports multiple log levels, allowing developers to categorize their log messages based on their importance and severity. The available log levels include:

Log Level	Description
DEBUG_LEVEL_INFO	General information about the application's state.
DEBUG_LEVEL_WARNING	Indications of potential issues or important events.
DEBUG_LEVEL_ERROR	Errors that have occurred in the application.
DEBUG_LEVEL_CRITICAL	Severe errors that may cause the application to crash.

> Note: You can set a minimum log level to filter out less important messages. For example, setting the log level to WARNING will only log WARNING, ERROR, and CRITICAL messages, while INFO messages will be ignored. `set_minimum_log_level(LogLevel level)` can be used to set the minimum log level for the logging system. You can also enable or disable logging `set_logging_enabled(bool enabled);`.

Logging Functions

The logging system provides a set of functions that developers can use to log messages at different log levels.

Function Name	Description
`LOG_INFO(const char* format, ...)`	Logs an informational message.
`LOG_WARNING(const char* format, ...)`	Logs a warning message.
`LOG_ERROR(const char* format, ...)`	Logs an error message.
`LOG_CRITICAL(const char* format, ...)`	Logs a critical error message.

Performance & Metrics

The logging system also includes a timer utility that allows developers to measure the time taken for specific operations or code blocks. This can be useful for performance profiling and optimization. The timer utility provides functions to start, stop, and reset timers, as well as retrieve the elapsed time in milliseconds.

Function Name	Description
`LOG_PERFORMANCE(NULL)`	Starts a performance timer.
`LOG_PERFORMANCE("Timer Name")`	Ends the specified performance timer.

Memory Debugging

In addition to logging messages and performance metrics, the logging system also includes a memory dump utility that allows developers to dump the contents of memory buffers for debugging purposes. This can be particularly useful for diagnosing issues related to memory management, such as buffer overflows or memory leaks. The memory dump utility provides a function to dump the contents of a specified memory buffer in a readable format.

Function Name	Description
`void dump_memory(const char *label, const void *buffer, size_t size);`	Dumps the contents of the specified memory buffer with a label for identification.

Exporting Logs

The logging system also provides functionality to export logs to a file for later analysis. This can be useful for debugging issues that occur in production environments or for keeping a record of application events. The log export function allows developers to specify the file path and name for the exported log file.

Function Name	Description
`void save_log_file(const char *path);`	Saves the current log messages to a file at the specified path.
