GETTING STARTED GUIDE

Apollo4 Graphics

Ultra-Low Power Apollo SoC Family

A-SOCAP4-GGNA02EN v1.0

Legal Information and Disclaimers

Revision History

Revision	Date	Description
1.0	September 7, 2021	Initial release

Reference Documents

Document ID	Description
D-API	NEMA|GFX - API Library Manual
	NEMA|GFX Debug Application Note
D-API	NEMA|DC - API Library Manual
PG-A4-6p0	Apollo4 SoC Family Programmer's Guide

Table of Contents

1. Introduction 8

2. Quick Start Guide 9

2.1 Initialization 9
2.2 GPU Composing Frames 10
2.3 DC Rendering Display 11

3. Portable Implementation 12

3.1 Interrupts Handling 12
3.1.1 GPU Interrupt 12
3.1.2 DC VSync 13
3.2 Graphics Memory Management 13

4. Graphics APIs 14

4.1 Platform APIs 14
4.2 Command Lists APIs 14
4.3 Binding Textures APIs 15
4.4 Command Graphics APIs 16
4.5 Blending APIs 17
4.5.1 Predefined Blending Modes 17
4.5.2 User Defined Modes 18
4.5.3 Additional Operations 18
4.6 Font APIs 19
4.7 Display Initialization and Timing APIs 19
4.8 Display Video Layers APIs 20
4.9 Display Refreshing APIs 20
4.10 DSI ULPS APIs 20

5. Graphics Examples 21

6. Additional Software Tools 22

6.1 NEMA|GUI-Builder 22
6.2 NEMA|PIX-Presso 23

7. Graphics Troubleshooting 24

List of Tables

Table 4-1 Color Mode Descriptions 15
Table 4-2 Texture Slot and Usage 16

List of Figures

Figure 4-1 Predefined Blending Modes 17
Figure 4-2 User Defined Blending Modes 18
Figure 4-3 Additional Operations Example 18
Figure 4-4 Vector and Bitmap Fonts 19
Figure 6-1 NEMA|GUI-Builder 22
Figure 6-2 NEMA|PIX-Presso 23

SECTION 1

Introduction

The graphics subsystem in Apollo4 provides 2D/2.5D acceleration and CPU offload for a rich graphics user interface. The integrated Graphics Process Unit (GPU) provides 2D drawing, blit support, text rendering support, full alpha blending, image transformation with compression schemes, and rich color formats to bring high quality graphical user interfaces. The display subsystem provides support for interfacing to display panels. Several display interface types are supported: SPI, QSPI, and MIPI DSI.

The AmbiqSuite SDK provides a low overhead software graphics library and APIs that interface directly with Apollo4 graphics and display subsystems. This provides the end users a simple and flexible tool for rapid graphical user interface (GUI) development, tailored for ultra-low power systems. Its small memory footprint, command lists features (allowing optimal CPU-GPU decoupling), low overhead features, and lack of any external dependencies make it an ideal API for developing vivid graphics on ultra-low power devices.

SECTION

2

Quick Start Guide

This section demonstrates the NEMA graphics API. Refer to the graphics examples in the AmbiqSuite SDK for the implementation details of steps below, such as **nemagfx_rotating_crate**.

2.1 Initialization

The first thing is the GPU and display controller (DC) initialization using:

```
int nema_init ( void )
int nemadc_init ( void )
```

If using a MIPI DSI panel, some additional DSI initialization is needed.

```
uint32_t am_hal_dsi_init(void)

uint32_t
am_hal_dsi_para_config(uint8_t ui8LanesNum, uint8_t ui8DBI-
BusWidth, uint32_t ui32FreqTrim)
```

If using a QSPI display driven by RM67162, initialize the display panel driver:

```
uint32_t

am_devices_nemadc_rm67162_init(uint32_t mode, uint32_t pixel_for-
mat, uint16_t resx, uint16_t resy, uint16_t minx, uint16_t miny)
```

If using a MIPI DSI display driven by RM67162, initialize the display panel driver:

```
uint32_t

am_devices_dsi_rm67162_init(uint32_t ui32PixelFormat, uint16_t
ui16ResX, uint16_t ui16ResY, uint16_t ui16MinX, uint16_t
ui16MinY)
```

2.2 GPU Composing Frames

The GPU is programmed via its configuration registers. Using Command Lists (CLs) is more efficient than writing directly to the configuration registers. CLs are capable of drawing complicated scenes while keeping the CPU workload to a minimum.

A typical routine for drawing using CL includes the following functions:

1. Create a CL and bind it.

   ```
   nema_cmdlist_t cl = nema_cl_create (); // Create a new CL
   nema_cl_bind_cmdlist (& cl); // Bind it
   ```

2. Set clipping rectangle.

   ```
   void nema_set_clip(...)
   ```

3. Bind destination texture (frame buffer).

   ```
   void nema_bind_dst_tex(...)
   ```

4. Bind source texture (if needed).

   ```
   nema_bind_src_tex(…)
   ```

5. Set blending mode.

   ```
   nema_set_blend_fill(…)/nema_set_blend_blit(…)
   ```

6. Draw/Fill/Blit.

   ```
   nema_draw_(…)/nema_fill_(…)/nema_blit*(…)
   ```

7. Submit CL for execution and wait for its completion.

   ```
   nema_cl_submit(&cl);
   nema_cl_wait(&cl);
   ```

The frame buffer (destination texture) is then updated by the GPU and the information is sent to the display.

For more details, refer to *NEMA|GFX - API Library Manual Doc Part: D-API*.

2.3 DC Rendering Display

DC uses *Layers* which are visual elements stacked to compose the final displayed image and send to the panel.

A typical routine for DC updating the display would be the following:

1. Set the DC layer.

   The layers can be scaled, cropped, positioned, blended and composed on the final display. They are completely independent to each other, therefore must be set separately.

   But in most of cases, 1 layer is enough. A **nemadc_layer_t struct** contains information regarding the layer, such as resolution, stride, format, and more. Layers are set as follows:

   ```
   void nemadc_set_layer (int layer_no , nemadc_layer_t * layer)
   ```

2. Send the frame to the display panel.

   ```
   void nemadc_send_frame_single(void)
   ```

   Use the alternative function below if a MIPI DSI panel is used.

   ```
   void dsi_send_frame_single_start(uint32_t ui32Mode)
   ```

3. Wait VSYNC for the completion of frame refreshing.

   ```
   void nemadc_wait_vsync ( void )
   ```

For more details, refer to *NEMA|DC - API Library Manual Doc Part: D-API*.

SECTION
3

Portable Implementation

The Graphics APIs are compatible among different Apollo4 SoCs and AmbiqSuite SDKs. However, each implementation of these APIs may be different and is released in the binary form with corresponding backend configurations.

- For Apollo4 RevB, the backend configurations are at: **third_party/ThinkSi/config/nemagfx_apollo4b**

Developers can further adjust these configurations for their RTOS and software architecture.

3.1 Interrupts Handling

3.1.1 GPU Interrupt

The users UI task calls **nema_cl_wait** to wait for GPU to finish current operation. GPU notifies CPU by an interrupt when the current graphics operation completes. Its interrupt handler should:

- De-assert hardware interrupt
- Wake/notify tasks waiting for interrupt

As the UI task is suspended at waiting CL (**nema_cl_wait**) by the **nema_wait_irq()**, the AmbiqSuite SDK users need to implement waking/notifying UI task for their RTOS in the interrupt handler **prvNemaInterruptHandler** in **nema_hal.c**.

3.1.2 DC VSync

The **nemadc_wait_vsync** function waits for a vsync signal to avoid screen tearing or visual artifacts. The UI task (or display rendering task) is suspended at waiting VSync after requesting DC refreshing the display.

When DC VSync interrupt arrives, the current refresh cycle finishes. Its interrupt handler should:

- De-assert hardware interrupt
- Wake/notify tasks waiting for interrupt

The AmbiqSuite SDK users need to implement waking/notifying action for their RTOS in the interrupt handler **prvVsyncInterruptHandler** in **nema_dc_hal.c**.

3.2 Graphics Memory Management

The graphics memory for the Apollo4 resides in shared SRAM (SSRAM). The memory pool of the graphics memory is defined in **nema_hal.c** as the following.

```
static AM_SHARED_RW uint64_t tsi_buffer[VMEM_SIZE/8];
```

The size **VMEM_SIZE** can be adjusted based on application need.

SECTION 4

Graphics APIs

The graphics APIs include a set of higher level calls, forming a complete standalone Graphics API for applications in systems where no other APIs are needed. This API is able to carry out draw operations from as simple as lines, triangles, and quadrilaterals to more complex ones like blitting and perspective correct texture mapping. They can also be used as a back-end to existing third-party GUI solutions.

4.1 Platform APIs

- GPU register access:
  - **nema_reg_read**: read GPU registers
  - **nema_reg_write:** write GPU registers
- DC register access:
  - **nemadc_reg_read**: read DC registers
  - **nemadc_reg_write**: write DC registers
- Graphics buffers management:
  - **nema_buffer_create/destroy()**

4.2 Command Lists APIs

A Command List (CL) is considered amongst the most important features of the NEMA® series.

- **nema_cl_create** allocates and initializes a new CL
- **nema_cl_bind** sets the referred CL as active. From that point on, each subsequent drawing call will incrementally be incorporated in the active CL. At any time, all drawing operations should be called when there is a bound CL.

- **nema_cl_unbind** unbinds the currently bound CL. The CL shall be unbound before submission.
- **nema_cl_submit** submits the referred CL for execution. When a CL is submitted for execution, it should never be altered until it finishes execution. Writing in such a CL results in undefined behavior. If this CL is currently the one that is bound, this call unbinds it.
- **nema_cl_wait** blocks till the CL completes.

4.3 Binding Textures APIs

The color modes listed in Table 4-1 can be used for both source and destination textures.

Table 4-1: Color Mode Descriptions

Color Mode	Description
RGBX8888	32-bit color with no transparency
RGBA8888	32-bit color with transparency
XRGB8888	32-bit color with no transparency
ARGB8888	32-bit color with transparency
BGRA8888	32-bit color with transparency
BGRX8888	32-bit color with transparency
RGBA5650	16-bit color with no transparency
RGBA5551	16-bit with 1-bit transparency
RGBA4444	16-bit color with transparency
RGBA3320	8-bit color with no transparency
L8	8-bit gray scale (luminance) color
A8	8-bit translucent color
L2	2-bit grayscale (luminance) color
L4	4-bit grayscale (luminance) color
BW1	1-bit color (black or white)
UYVY	UYVY color
TSC™4	4-bit proprietary compressed
YUV	YUV
Z24_8	32-bit (24+4) depth and stencil
Z16	16-bit depth

The NEMA® Graphics Core is a programmable very long instruction word (VLIW) processor, which allows rapid calculations between colors. It is programmed through instructions in binary form, called Shaders. NEMA|p incorporates four texture slots, allowing four textures to be bound simultaneously. Accordingly, the hardware allows a single Shader to read from and/or write to four different textures as seen in Table 4-2.

Table 4-2: Texture Slot and Usage

Texture Slot	Texture Usage
NEMA_TEX0	Destination/Background Texture
NEMA_TEX1	Foreground Texture
NEMA_TEX2	Background Texture
NEMA_TEX3	Depth Buffer

- **nema_bind_dst_tex** binds a texture to serve as destination. Every drawing operation should have an effect on a given destination texture. The texture’s attributes (GPU address, width, height, format, and stride) are written inside the bound CL. Each subsequent drawing operation will have an effect on this destination texture.
- **nema_bind_src_tex** binds a texture to be used as foreground. Most common graphics operations include some kind of image blitting (copying), like drawing a background image, GUI icons or even font rendering. Besides the texture's attributes, It has one extra argument, **NEMA_tex_mode_t** mode, that determines how to read a texture (point/bilinear sampling, wrapping mode etc).
- **nema_bind_src2_tex** binds a background texture to **NEMA_TEX2** slot. This is needed when the Blending Mode to be used does not use the destination texture (**NEMA_TEX0**) as background texture at the blending operation.

4.4 Command Graphics APIs

Drawing APIs can draw geometry primitives. Based on these primitives, the Programmable Core can do simple operations such as filling and blitting.

- Filling APIs fills a primitive with a color.
- Blitting APIs copy source texture to destination texture to fit a specified shape with rotation or not.
- Clipping APIs set the drawing area's clipping rectangle.
- The Programmable Core can do more advanced ones such as blurring and edge detection.

4.5 Blending APIs

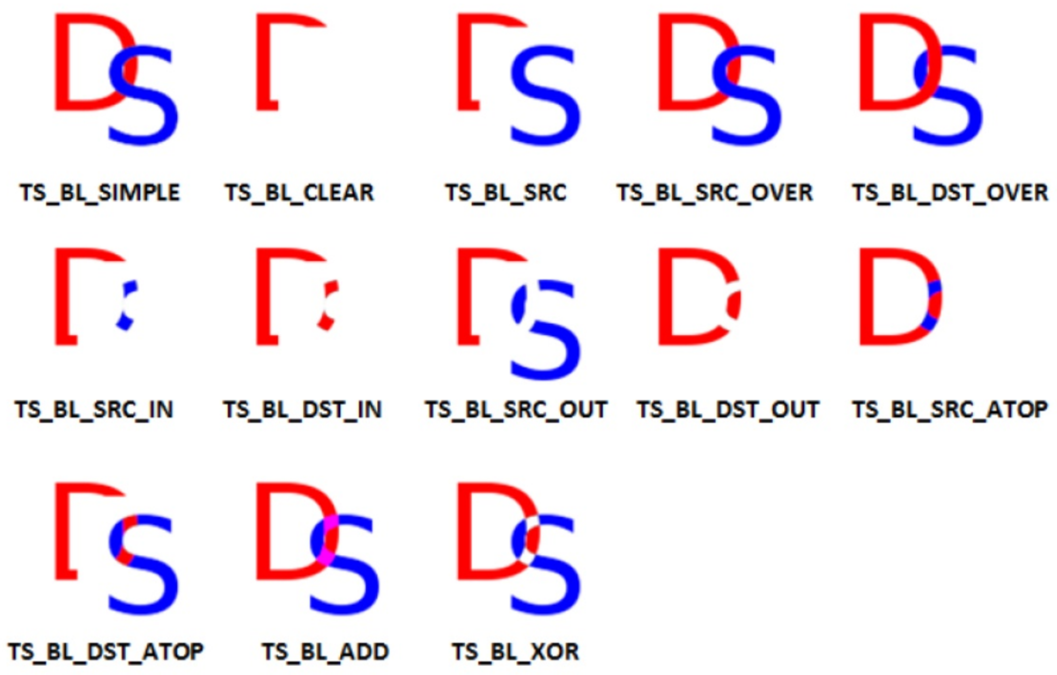
Blending refers to a convex combination of two colors – a translucent source (foreground) and a destination (background), allowing transparency effects. The basic blending algorithms define a set of mathematical operations for the Color channels (RGB) and the Alpha (transparency) channel of a fragment. The blending process is essential for rendering fonts and/or creating GUIs. In the NEMA graphics pipeline, blending is carried out in the Graphics Core.

Blending APIs employs pre-assembled commands to create a powerful set of blending algorithms in-steading of running a compiler for creating Shaders.

- **nema_set_blend_fill** refers to blending when filling a primitive with a color.
- **nema_set_blend_blit** refers to blending when blitting a texture.

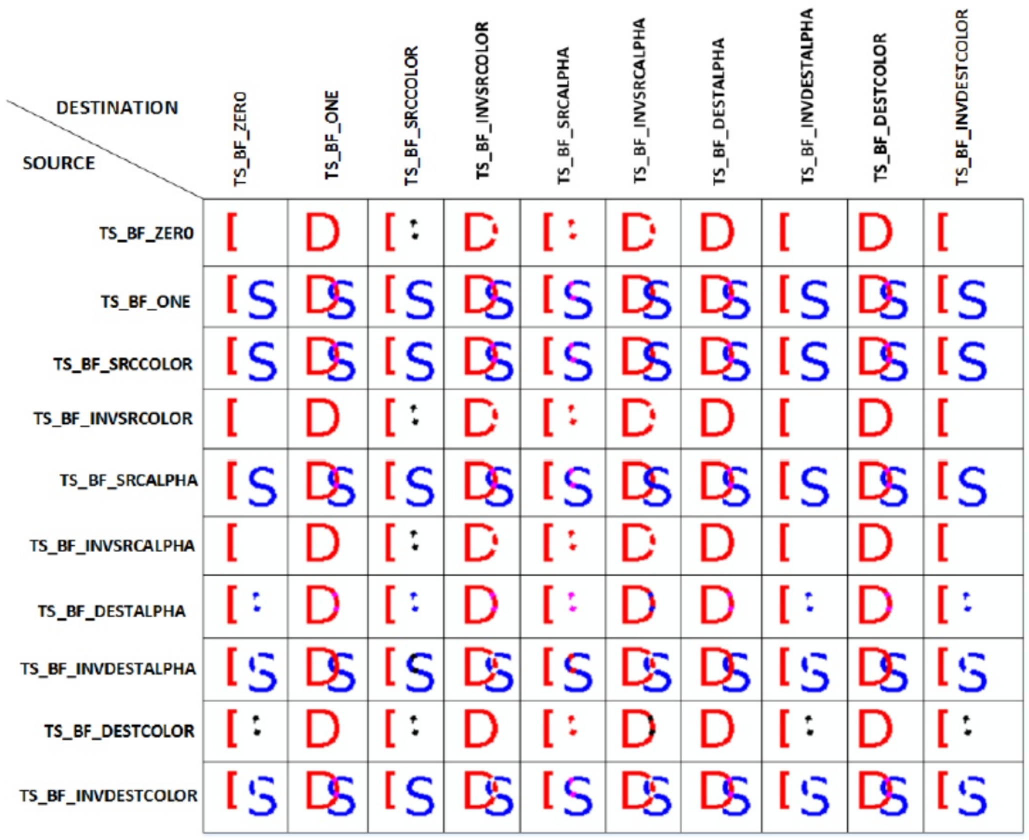
4.5.1 Predefined Blending Modes

Figure 4-1: Predefined Blending Modes



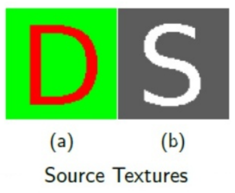
4.5.2 User Defined Modes

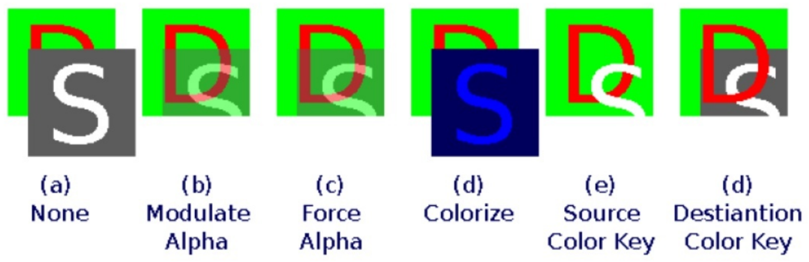
Figure 4-2: User Defined Blending Modes



4.5.3 Additional Operations

Figure 4-3: Additional Operations Example

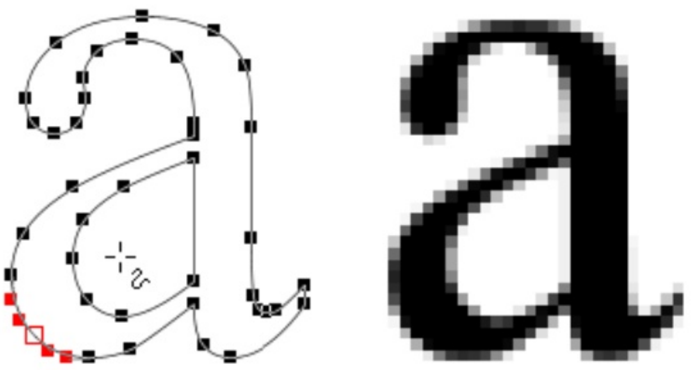




4.6 Font APIs

Drawing text on the screen is an important element of any Graphical User Interface. To draw a string, you will need a Typeface, the text to be drawn and some attributes on how the text is to be displayed. Typefaces are sourced in TrueType (TTF) file type, which contains scalable representations of typefaces described as vector curves. Scalable fonts are converted to raster fonts (bitmap fonts) by rasterization to a particular size and format. Raster fonts are drawn on the screen as a series of images with each letter drawn after the other using the correct letter width. To facilitate this process, NEMA|GFX Library handles text display and alignment using special functions.

Figure 4-4: Vector and Bitmap Fonts



This first step is to convert a TrueType font to Bitmap font offline by *NemaGFX_FontUtil* tool.

The **nema_bind_font** is to bind the data structure of a typeface and nema_draw_-text is to draw the text.

4.7 Display Initialization and Timing APIs

- **nemadc_init** initializes the NEMA|dc and retrieves its configuration by getting access to its register file.
- **nemadc_set_bgcolor** sets the background color once the DC is initialized.
- **nemadc_timing** set different timing parameters include setting display resolution and blanking the front and back porch.

  Note: The DSI interface must be initialized if using MIPI DSI mode.

- **am_hal_dsi_init** initializes the DSI interface.
- **am_hal_dsi_para_config** configures the DSI interface. We recommend 1 lane, 16-bit DBI bus and 240MHz mode (ui8LanesNum = 1, ui8DbiWidth = 16, ui32-FreqTrim = 10).

4.8 Display Video Layers APIs

NEMA|dc400 supports up to four layers. These layers can be scaled, clipped, positioned, and composed on the final display. They are completely independent to each other, therefore must be set separately.

- **nemadc_layer_enable** and **nemadc_layer_disable** enable and disable layers at runtime.
- **nemadc_set_layer** sets a layer's information such as resolution, stride, format.
- **nemadc_set_layer_addr** sets its physical address.
- Besides the above, NEMA|dc implements a programmable global and per-layer Gamma Look Up Table (LUT) for gamma correction. Gamma LUT consists of a 3x256x8 memory array that holds the RGB values for each of the 256 colors in the palette. In order to program it, NEMA|DC-API implements getter and setter calls for both the global and the per layer Gamma LUT.

4.9 Display Refreshing APIs

The **nemadc_send_frame_single()** sends the frame to the display panel. If you're using DSI interface, use **dsi_send_frame_single(NEMADC_OUTP_OFF)** instead.

The **nemadc_wait_vsync** waits for Vertical Synchronization (VSync) which means the DC is done with the frame refresh. While the DC is busy scanning the current frame from memory, front frame should never be altered. Any modifications to the layer information or address should be done after VSync.

4.10 DSI ULPS APIs

Two pairs of APIs for DSI ultra-low power state (ULPS) mode are provided:

- **am_hal_dsi_ulps_entry**/**am_hal_dsi_ulps_exit** enter/exit ULPS mode.
- **am_hal_dsi_napping**/**am_hal_dsi_wakeup** enter/exit ULPS and power off/on the DSI hardware to save more power.

For function details, refer to the *Apollo4 SoC Family Programmer's Guide Doc Part: PG-A4-6p0*.

SECTION

5

Graphics Examples

See Read Me file in the latest AmbiqSuite SDK for more information.

SECTION

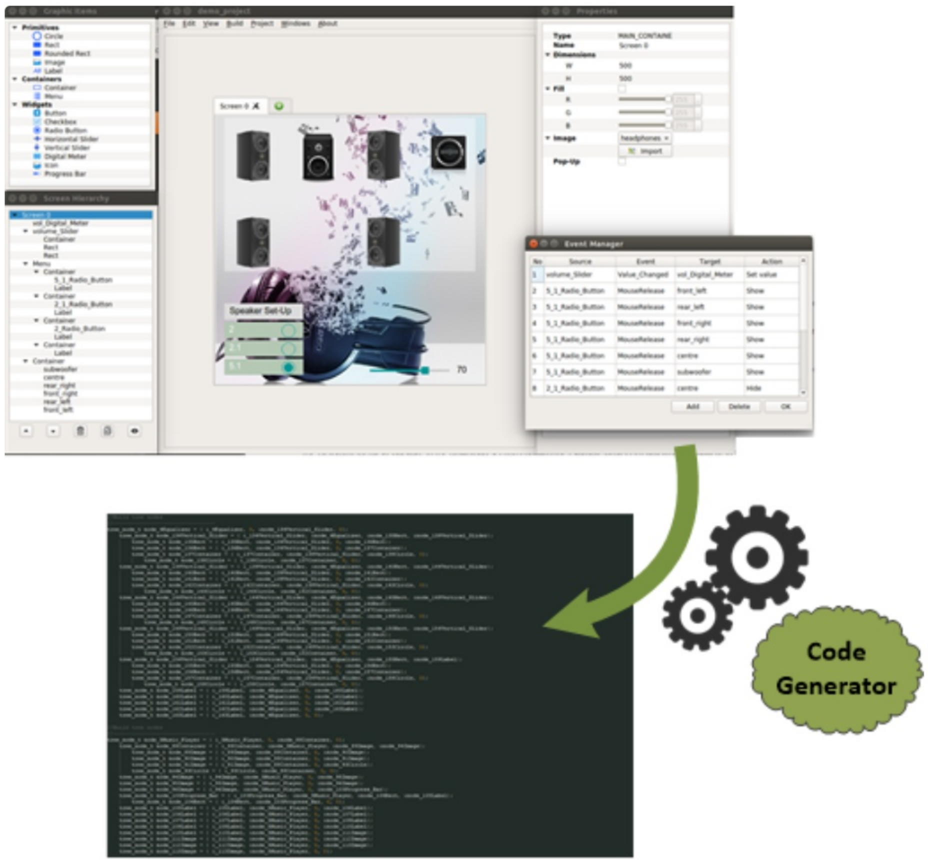
6

Additional Software Tools

6.1 NEMA|GUI-Builder

NEMA|GUI-Builder is a Rapid GUI Design Toolkit that allows drag-and-drop creation of advanced GUI. It is a simple and flexible tool for rapid graphical user interfaces development.

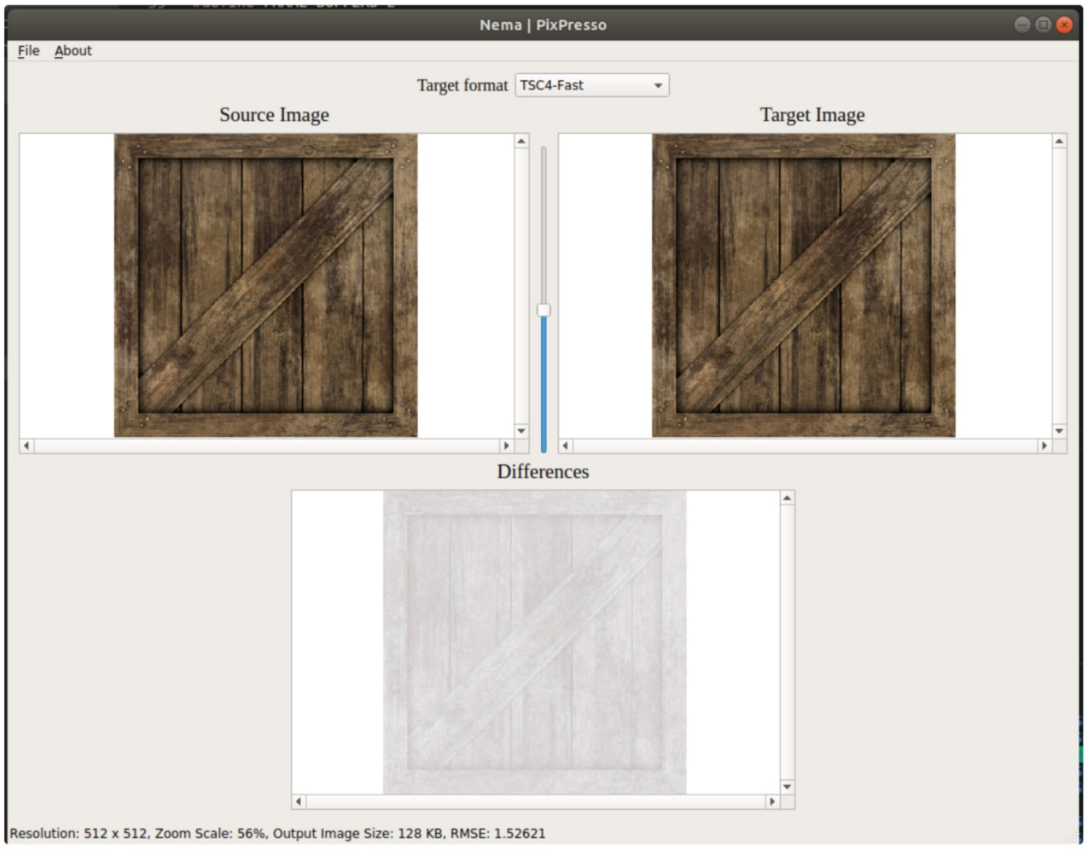
Figure 6-1: NEMA|GUI-Builder



No	Source	Event	Target	Action
1	volume_Slider	Value_Changed	vol_Digital_Meter	Set value
2	5_1_Radio_Button	MouseRelease	front_left	Show
3	5_1_Radio_Button	MouseRelease	rear_left	Show
4	5_1_Radio_Button	MouseRelease	front_right	Show
5	5_1_Radio_Button	MouseRelease	rear_right	Show
6	5_1_Radio_Button	MouseRelease	centre	Show
7	5_1_Radio_Button	MouseRelease	subwoofer	Show
8	2_1_Radio_Button	MouseRelease	centre	Hide

6.2 NEMA|PIX-Presso

Figure 6-2: NEMA|PIX-Presso



Resolution: 512 x 512, Zoom Scale: 56%, Output Image Size: 128 KB, RMSE: 1.52621

NEMA|PIX-Presso is a utility for converting images to formats suitable for low power embedded devices. The purpose of the Converter is to act as an easy to use companion for graphics developers in order to adapt images to applications requirements.

SECTION 7

Graphics Troubleshooting

Refer to *NEMA|GFX Debug Application Note v1.0*.

A-SOCAP4-GGNA02EN v1.0
September 2021