USER'S GUIDE

Apollo4 Display Kit NEMA®| GUI-Builder

Ultra-Low Power Apollo SoC Family

A-SOCAP4-UGGA03EN v1.0

Legal Information and Disclaimers

Revision History

Revision	Date	Description
1.0	August 27, 2021	Initial release

Reference Documents

Document ID	Description
A-SOCAP4-PBGA02EN	Apollo4 Display Kit Product Brief
A-SOCAP4-GGGA01EN	Apollo4 Display Kit Getting Started Guide
A-SOCAP4-PGGA01EN	Apollo4 Programmer's Guide
A-SOCAP4-DSGA01EN	Apollo4 Datasheet

Table of Contents

List of Tables

List of Figures

SECTION 1 Introduction

A partnership with Think Silicon Ltd enables Ambiq to provide the NEMA®|GUI-Builder, a product based on Think Silicon's NemaGFX graphic library. NEMA|GUI-Builder is one of the main software tools available in NEMA GPU's ecosystem. Its main purpose is to provide 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.

NOTE: Unless specifically noted, all the examples mentioned in this document are included in the AmbiqSuite SDK release under directory: apollo4b_evb_disp_shield\examples\graphics

1.1 System Requirements

NEMA|GUI-Builder System Requirements:

- Operating System: Windows® (10), Linux® (Ubuntu 32/64-bit)
- Screen Resolution: 800×600 or higher
- RAM: At least 256MB (minimum)
- Hard Drive: 100MB available space.

1.2 Terminology

Table 1-1 shows the terminology used throughout this document.

Table 1-1: Terminology

Term	Definition
Blending	Blending is the process by which two or more images are combined in a weighted manner per-pixel to create merged images.
Fragment	A fragment is the data representing a single-pixel primitive. This includes raster position, color, and texture coordinates.
Interpolation	Interpolation is the process of generating intermediate values between two known data points to give the appearance of continuity and smooth transition. Several distinct interpolation techniques are used in both computer graphics and animation such as linear, bilinear, spline and polynomial interpolation.
Pixel	A pixel is the smallest addressable element in a digital display device. A pixel is generally considered as the smallest single component of a digital image and is often used as a measurement unit.
Primitives	Primitives in computer graphics are the simplest geometric objects a system can handle. Common sets of two-dimensional primitives include lines, points, triangles, and polygons while all other geometric elements are built up from these primitives. In three-dimensions, properly positioned triangles or polygons can be used as primitives to model more complex forms.
Raster image (or bitmapped image)	A raster image (or bitmapped image) is a matrix data structure representing the actual image content.
Rasterization	Rasterization is the process of converting an image described in a vector graphics format to a raster image consisting of pixels for output to a video display or for storage in bitmap format.
Texel	A texel is the fundamental unit of texture space. Textures are represented by arrays of texels in the same way that pictures are represented by arrays of pixels.
Texture	Texture is the digital representation of an object's surface. In addition to two-dimensional qualities such as color and transparency, a texture also incorporates three-dimensional ones such as reflectiveness.
Texture mapping	Texture mapping is the process of wrapping a pre-defined texture around any two or three-dimensional object.
Vector graphics	Vector graphics is a technique of using polygons, plane figures bound by a finite chain of straight-line segments closing a loop, to represent images. Vector graphics have inherent scale up abilities, only depending on the rendering device capability.
Vertex	A vertex is a data structure that describes the location of an object by defining its corners as positions of points in two or three-dimensional space.

SECTION

2

Access and Licensing

The Think Silicon Contributor License Agreement (CLA) can be found at:
https://www.think-silicon.com/?section=2358&language=en_US

2.1 Product IP Licensing

The Apollo4 Display Kit includes a 45-day evaluation period with the PRO Edition of NEMA|GUI-Builder licensed through Think Silicon S.A.. A user agreement must be executed between Ambiq Inc. and the customer.

A free version of NEMA|GUI-Builder can be downloaded on the Think Silicon website at https://www.think-silicon.com/?section=2183&language=en_US

2.2 Partner Contacts

Table 2-1: Partner Contacts

Company	Contact	Email Address
Ambiq Inc.	Marc Miller	mmiller@ambiq.com
Think Silicon S.A.	Ulli Mueller	u.mueller@think-silicon.com

SECTION

3

Supported Boards and Examples

Table 3-1: Supported Boards

Board ID	Description
AMAP4DISP	Apollo4 EVB and Display Shield ▪ 454x454 Pixel MIPI/SPI/QSPI AMOLED display ▪ Display laminated ambient light sensor – TSL2540 ▪ Display laminated capacitive touch sensor – TMA525C ▪ Touch Screen and ALS Support ▪ 256MB Octal-SPI double-data-rate (DDR) enabled PSRAM – APS256XXN ▪ 32MB Octal-SPI DDR enabled flash memory – ATXP032 ▪ 4GB x1/x4/x8 e-MMC module – THGBMNG5D1LBAIT ▪ 3-Axis MEMS accelerometer – ADXL362

The Apollo4 Display Kit comes with example projects, which can be found inside the installation directory, in the examples folder. The goal of these examples is to familiarize the user with the NEMA|GUI-Builder environment and the way it handles the GUI development process. They can be instantly simulated to evaluate their behavior at runtime.

The examples include:

- **Animated screens:** Contains six screens that are divided into two screen groups. It depicts the functionality of swiping inside a screen group, along with how a transition from one group to another is performed (button-triggered transition).
- **Gauge:** This example contains a button that is connected to a gauge and a digital meter. Clicking the button will set the value of the gauge and the digital meter using a transition event.
- **Watch:** This example contains a watch face and graphics. Transitions and animations may be used to demonstrate performance of the features of a typical watch example on the hardware.

In each of these examples, the events used to perform their functionality can be inspected in the Event Manager as described in *Section 7 Event Manager on page 27* of this document. It is recommended not to modify these examples, as their goal is to act as guides for GUI development projects. Modifying these examples and saving the changes will overwrite them and their initial structure cannot be recovered.

NOTE: Unless specifically noted, all the examples mentioned in this document are included in the AmbiqSuite SDK release under directory:
apollo4b_evb_disp_shield\examples\graphics

SECTION

4

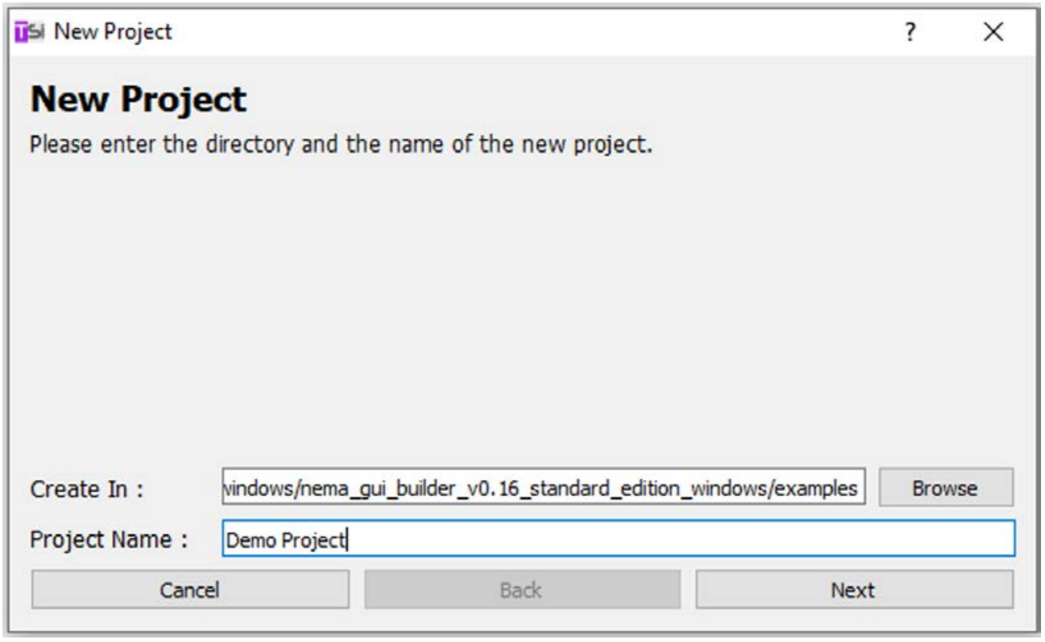
Getting Started

4.1 Creating a New Project Using NEMA|GUI-Builder

Use the following procedure to create a new project using NEMA|GUI Builder:

1. Open the NEMA|GUI Builder application.
2. Click **File**, then select **New Project**.
3. Complete the following in the New Project window (Figure 4-2):
   a. Click **Browse** and select the appropriate project directory.
   b. Type the project name in the **Project Name** box.
   c. Click **Next**.

Figure 4-1: New Project Window


New Project

New Project

Please enter the directory and the name of the new project.

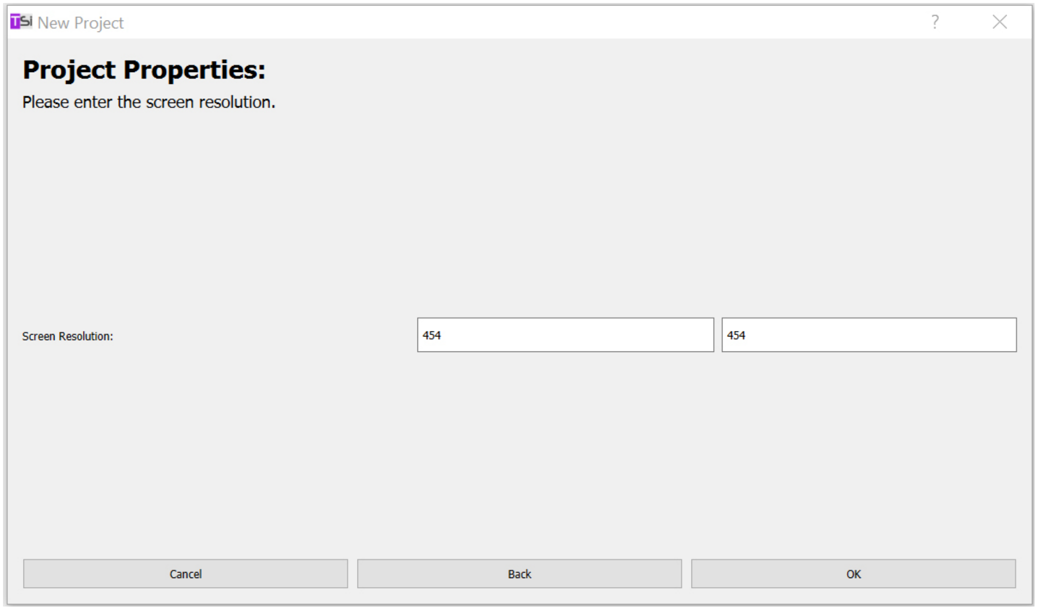
Create In : vindows/nema_gui_builder_v0.16_standard_edition_windows/examples

Project Name : Demo Project

4. Complete the following in the Properties window (Figure 4-2):
   a. Type the appropriate resolution in the **Screen Resolution** boxes.
   b. Click **OK**.

      Note: The wizard will allocate the directory that has been entered for the project related files (assets, generated files, etc).

Figure 4-2: Project Properties Window



New Project

Project Properties:

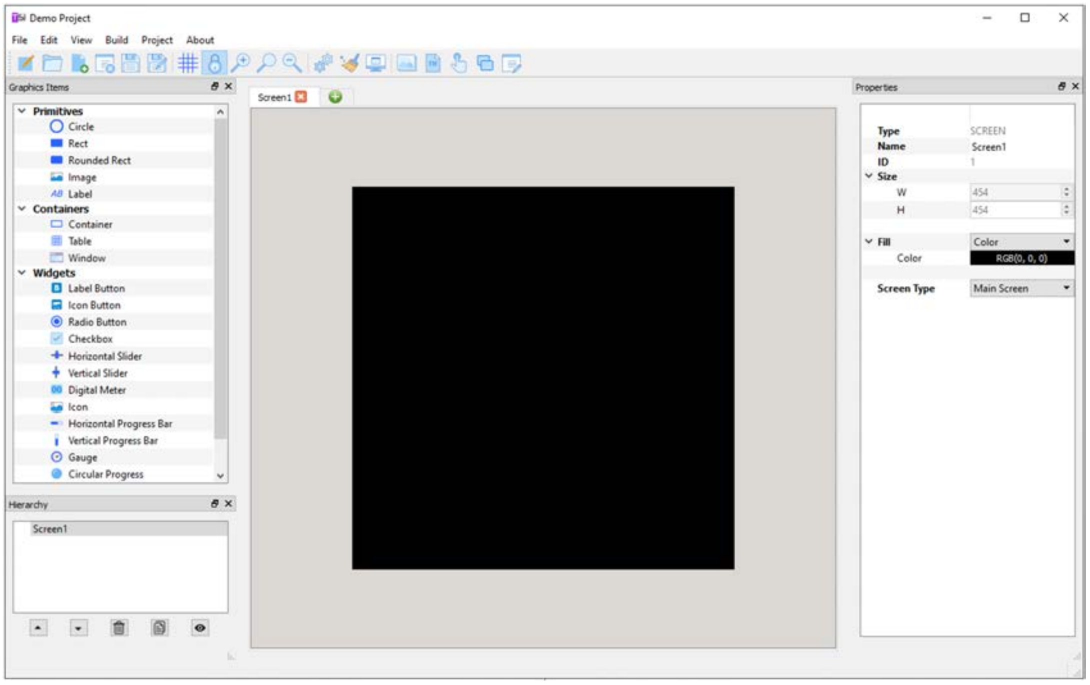
Please enter the screen resolution.

Screen Resolution: 454 454

4.2 Using GUI in the Design Area

NOTE: You must complete the steps in *Section 4.1 Creating a New Project Using NEMA|GUI-Builder on page 13* in order to start designing the GUI in the Design Area (Figure 4-3)

Figure 4-3: GUI in the Design Area



Type	SCREEN
Name	Screen1
ID	1
Size	
W	454
H	454
Fill	Color
Color	RGB(0, 0, 0)
Screen Type	Main Screen

4.3 Adding Screens

Use the following procedure to add screens in the GUI:

1. Click the **+** (plus) icon in the design area.
2. Set the screen as **Main Screen** or **Secondary** in the **Screen Type** drop-down option.

NOTE: This affects the way they are displayed at the application runtime.

- **Main Screens** have fixed resolution and when they are displayed, they occupy the whole area of the framebuffer.
- **Secondary** screens have variable resolution so that they can be used along with a Window item, or they can be displayed as pop-up screens

4.4 Editing the Background of the Screen

The background of the screen, which is either a plain color or an image, can be edited in the Properties panel. Images must be added in the project's assets before they can be used in the project. See *Section 4.5 Importing Images in the Project Assets on page 16*.

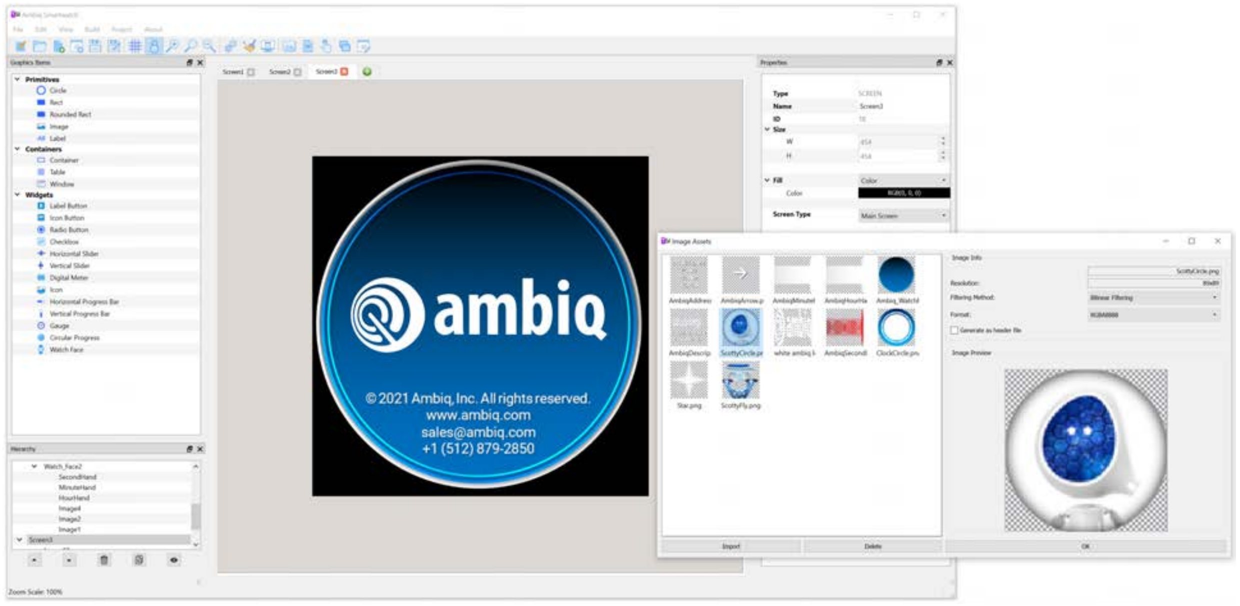
4.5 Importing Images in the Project Assets

Use the following procedure to import images in the project assets:

1. Complete one of the following to add images in the project asset:
   - Option 1: Click **Import** in the Properties panel.
   - Option 2: Click **Project**, then select **Assets/Images**.
2. Inspect and modify the project's images as shown in Figure 4-4.

NOTE: The imported images can now be used to customize image compatible graphics items (e.g., images, containers, and more).

Figure 4-4: Importing Images as Project Assets



4.6 Adjusting the Design Area Zoom Scale

Zooming into or out of the design area is performed by pressing the **Ctrl** key and scrolling the mouse wheel, while the mouse cursor is within the design area, or alternatively by using the keyboard shortcut keys, as shown in Table 4-1:

Table 4-1: Keyboard Shortcut for Zooming in/out

Keyboard Shortcut	Actions
Ctrl+= (or Ctrl++)	Zoom in
Ctrl+-	Zoom out
Ctrl+0	Reset zoom to 100%

The zoom scale is displayed at the status bar at the bottom of the design area.

4.7 Using the Grid to Align Graphic Items

The Grid is a feature that eases the alignment of graphic items. It can be shown or hidden by checking the respective checkbox under the **View** menu. In addition, graphic items can be snapped to the grid by selecting the corresponding option while the grid parameters (width, height, horizontal and vertical offset) can be modified under the same menu. This guarantees that the items will be aligned on the desired line of the grid, thus avoiding the manual alignment by setting each of the coordinates of every item. Note that snapping is available only for the top-left corner of each item.

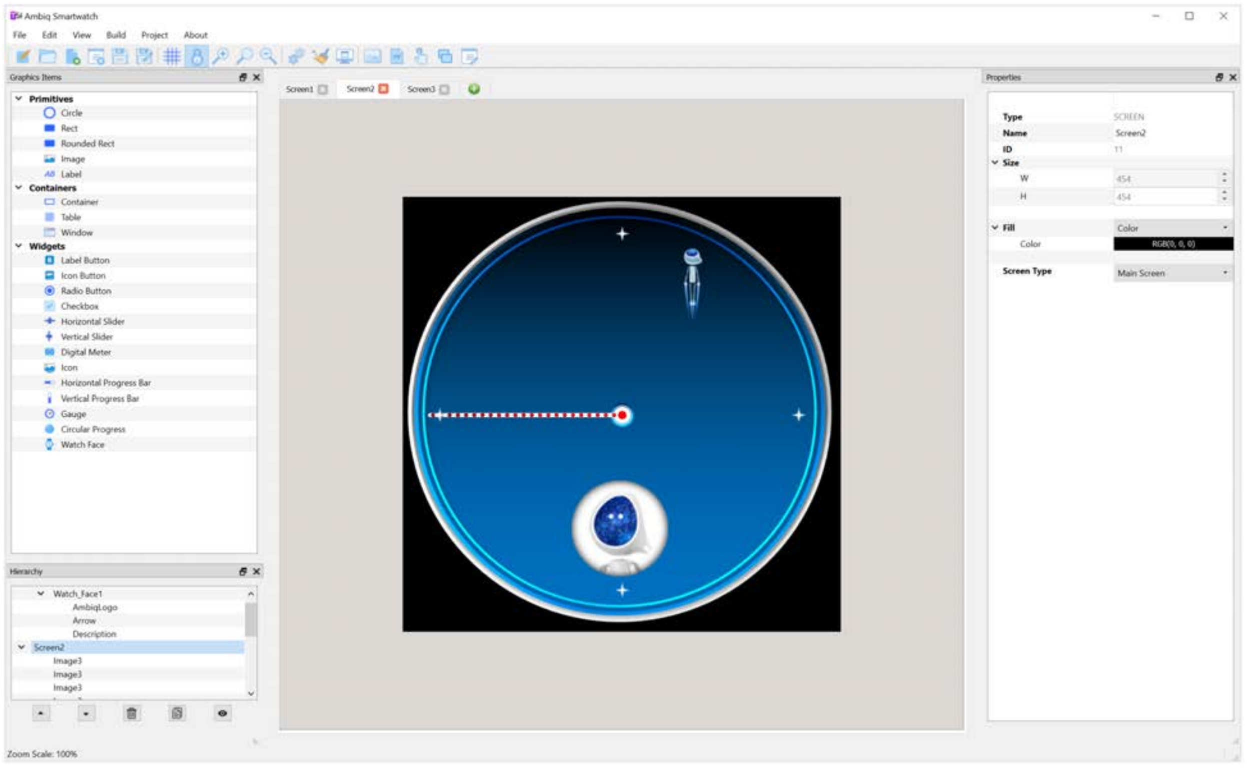
4.8 Understanding Graphic Items

Graphic items support generic capabilities such as copy, paste, cut, delete, bring to front, and send to back. Whenever such actions take place in Containers, the same action takes place for the contained graphic items.

These items consist of a set of basic items (e.g., sliders that are made of two rectangles and a container). Manipulating the basic items can be performed in the Hierarchy window, or by double clicking on them in the Design Area. The user can select these items in the Hierarchy window and edit their properties in the Properties window. In addition, supported drag-and-drop operations in such items can also be performed in the Hierarchy window by dragging the name of the desired item and dropping it over the name of the desired parent item.

The functionality of complex items is extended as they come with some default features and can be customized to the demands of each application (e.g., dropping a circle inside a slider’s container changes the look of the slider’s indicator).

Figure 4-5: Design area after the addition of graphic items



Type	SCREEN
Name	Screen2
ID	11
Size	
W	454
H	454
Fill	Color
Color	RGB(0, 0, 0)
Screen Type	Main Screen

4.9 Saving a Project

A project can be saved in the designated location (project directory) as a ***.tsg** file. This will save:

- The project’s structure (in xml format)
- Assets (images and fonts)
- Events (explained in more detail in *Section 7 Event Manager on page 27*)

A saved project with these features (structure, assets, and events) can then be opened by the application for further modifications. The NEMA|GUI-Builder is configured to auto-save the current project silently every 2 minutes.

WARNING: When a project has been saved, be cautious when modifying the saved files outside the NEMA|GUI-Builder. This could break associations between these files by breaking the project’s structure or even by making the project incompatible with NEMA|GUI-Builder.

4.10 Recovering a Project from a Backup File

The backup files are located inside the project directory in the folder labeled **Backup**. In order to recover the project from the Backup files, copy the backup files to the project sources manually.

SECTION 5

Library Description

This section describes the basic structural elements and features of the NEMA|GUI Builder and Nema|GFX libraries. Examples of proper library usage and general information on the overall operation scheme are included as provided by Think Silicon S.A.

5.1 2D Graphics/Primitives

Primitives (circle, rectangle, rounded rectangle, image, and label) are elementary items that perform basic drawing operations:

Table 5-1: Primitive Graphic Items

Graphic Items	Description
Circle	Circle of desired radius. Can be either filled with a specified color or not.
Rectangle	Rectangle of desired size (width/height). Filled or not.
Rounded Rectangle	Same as Rectangle, corners are rounded according to a desired radius value.
Image	Image item that must be associated to an image asset.
Label	Label for displaying text information. Must be associated to a font asset.

In the NEMA architecture, these primitives are generated by the Rasterizer module. The Rasterizer generates the fragments contained inside the Primitive and feeds them to the Programmable Core for processing. The NEMA GPUs can draw the following Geometry Primitives:

- Points
- Lines
- Filled Triangles
- Filled Rectangles
- Filled Quadrilaterals

All of the aforementioned Primitives can be processed by the Programmable Core to do simple operations (e.g., filling with a constant color or gradient, blitting, etc.) or more advanced ones (e.g., blurring, edge detection, etc.). This functionality can be extended through software to draw:

- Triangles
- Rectangles
- Polygons
- Filled Polygons
- Triangle Fans
- Triangle Strips
- Circles
- Filled Circles
- Arcs
- Rounded Rectangles

5.2 Containers

Containers (container, table, window) are more complex items than primitives as they can be used in order to group together several other items. More specifically, items inside a container are grouped so that they can be moved together along with their parent item (container). Tables are used to group together several containers for the creation of list-like tables.

Table 5-2: Container Graphic Items

Container Items	**Description**
Container	Containers act as parent items to the items they contain. They can be filled with a selected color or an image.

Table 5-2: Container Graphic Items *(Continued)*

Container Items	Description
Table	Tables consist of several sub-items (containers) in a tabular layout. The number of rows and columns, their dimensions and the distance between them can be configured. When adding new items, the last added item is copied to the new ones.
Window	The Window can display any screen, other than its parent, within its area (this can be selected by its corresponding Source Screen property). The displayed content is scrollable at runtime and therefore a Window can be used to create scrollable items (e.g., scrolling tables).

5.3 Widgets

Widgets (label button, icon button, radio button, horizontal slider, vertical slider, digital meter, icon, progress bar, gauge, circular progress, watch face, and digital clock) are the graphic items that comprise the GUI during application runtime. Widgets are able to send or receive events to and from other widgets. Some primitives can also act as widgets (e.g., when an image needs to be displayed at the press of a button, thus the image receives an event). This functionality is achieved by setting the graphic item’s Interactive property in the Properties window.

Table 5-3: Widgets Graphic Items

Widget Items	Description
Label Button	Button containing text. The background of the text (image or color) can be configured when the button is pressed and released.
Icon Button	Button containing an icon. The background of the icon (image or color) can be configured when the button is pressed and released.
Radio Button	Radio buttons must be placed inside of a table for grouping multiple radio buttons. Checking a radio button will uncheck all the other radio buttons that belong to the same group or table.
Vertical/Horizontal Slider	The slider consists of two rectangles (filled and empty) and a container as its indicator. The properties of each sub-item can be edited by selecting the respective item in the Hierarchy window.
Digital Meter	The digital meter is a widget for displaying numerical values. The background color, precision (number of decimal digits) and initial value can be edited.

Table 5-3: Widgets Graphic Items *(Continued)*

Widget Items	Description
Icon	Icon consists of an image and a label. It can be used in cases whereby pressing it activates a specific action.
Vertical/Horizontal Progress Bar	The progress bar is widget for displaying the progress attribute. Respective events about setting its value must be manually configured.

For the full list of widgets and associated impact on memory usage, see *Section 9.2 Widget Footprint on page 40*.

5.4 Images

Images that are used in the GUI must be imported to it before becoming available for graphic item customization. Images can be imported under the **Project** menu. Current supported formats are png, jpg, and svg image formats.

After importing an image to the assets, the target format to be used during runtime can be modified. The imported images must be converted into a format suitable for low power devices. Such formats require direct pixel mapping in the memory subsystem of the device. NEMA|GUI-Builder currently supports the following formats:

- RGBA8888 (32 bits-per-pixel)
- RGBA5650 (16 bpp)
- RGBA5551 (16 bpp)
- RGBA4444 (16 bpp)
- L8 (luminance-only 8bpp)
- A8 (transparency-only 8bpp)
- Think Silicon’s proprietary and patented formats:
  - TSC™4 (4 bpp)
  - TSC™6 (6 bpp)
  - TSC™6a (6 bpp with alpha channel support).

The default format for newly imported images is set to RGBA8888.

The format of an associated image asset to an item with an opacity value lower than 255 must support opacity, otherwise these items will not be displayed properly during the application runtime. Think Silicon’s formats (TSC4 and TSC6) as well as RGBA5650 do not support opacity. If an image asset’s format is A8, a default color can be selected in order to colorize the displayed image.

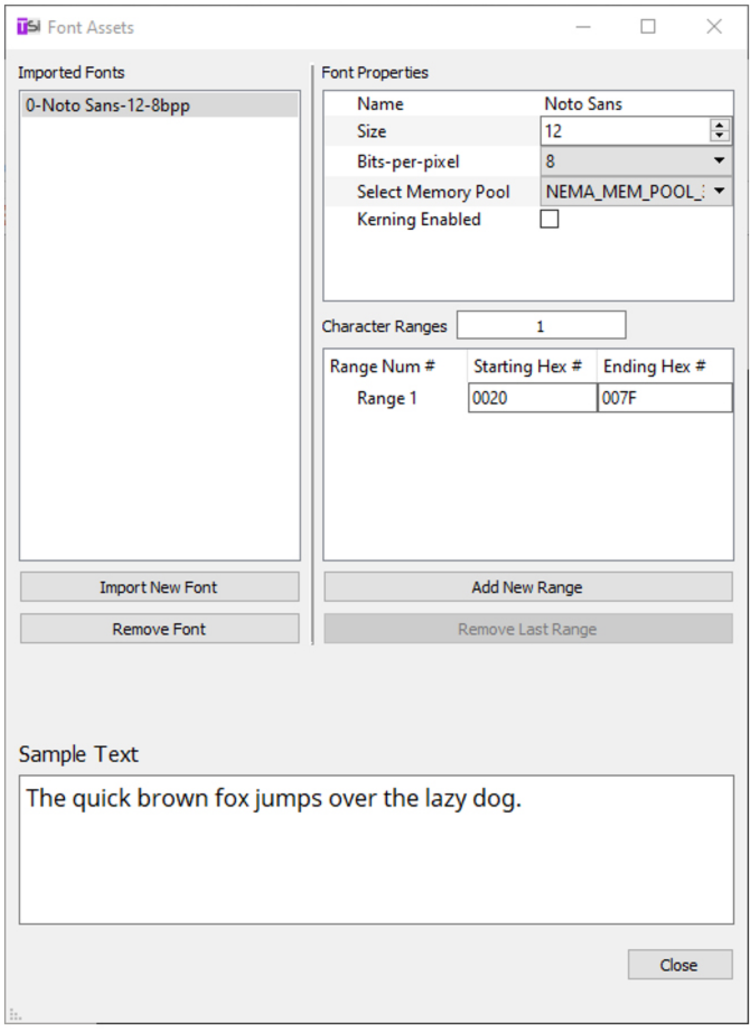
Besides the target format that affects the application memory requirements, the texture filtering method (the way that image pixels are rendered on the output screen pixels) can be selected. This method can either be "point-sampling filtering" (also known as "nearest-neighbor filtering") or "bilinear filtering". The former offers high performance versus poor rendering quality, while the latter trades performance for rendering quality.

At the runtime of a deployed project, the generated assets (images and fonts) are usually stored in a file system (e.g., in an SD card). These stored files are then loaded to the main memory of the system so that the application can start executing.

5.5 Fonts

Fonts are necessary for customizing any graphic items with text support such as labels, digital meters, and more. NEMA|GUI-Builder includes the NotoSans_Regular-12 (font family: NotoSans-Regular, font size: 12) as a default font. New fonts can be imported from the respective **Asset** menu. Only true-type-fonts are currently supported and, consequently **.ttf** files can only be imported. Figure 5-1 shows the form used for this purpose:

Figure 5-1: Font Assets Window



0-Noto Sans-12-8bpp

Name	Noto Sans
Size	12
Bits-per-pixel	8
Select Memory Pool	NEMA_MEM_POOL_
Kerning Enabled	☐

Character Ranges 1

Range Num #	Starting Hex #	Ending Hex #
Range 1	0020	007F

Sample Text

The quick brown fox jumps over the lazy dog.

When importing a font, its size, bits-per-pixel, range count, start and end values for each range need to be set. By using the default values, the imported font will have size 12, 8 bits-per-pixel and one range that spans from 32 to 127; this is the range for the ASCII character set.

In order to include characters beyond this range, the start and end values can be modified. However, as more characters are included, the memory footprint will increase. Each range can contain characters that belong to the Unicode character set: 0 up to 10FFFF (hexadecimal value). In addition, the size and bits-per-pixel parameters directly affect the memory size of the generated fonts as well. In order to minimize the memory consumption of the application, ensure that these parameters are fine-tuned before generating the project code.

SECTION

6

Features

Each project created in NEMA|GUI-Builder has properties that affect the code generation process as well as the generated code. They are as follows:

- The number of framebuffers that will be used (single, double, or triple buffering)
- The frame-buffer format (RGBA8888, RGBA5650, TSC4, TSC6)
- The number of back buffers that will be used (up to two). Two back buffers are necessary when performing animations such as screen transitions using a nonlinear animation effect. If the animation buffers are less than two, screen transitions will be performed using a linear effect and show/hide animations will be performed instantly, without animation.
- The animation-buffers format (RGBA8888, RGBA5650, TSC4, TSC6) To keep the memory usage to a minimum, the default format of these buffers is TSC4.
- The memory pool that will be used for the framebuffers and the back buffers.
- The animations frame rate (this sets the animation timer period used in the generated code)
- The project's resolution and the code generator options (whether the code generator will generate the fonts, images and if the generated images will be scaled to the minimum possible resolution as identified by NEMA|GUI-Builder).

The current project path can be navigated to by selecting the **Open Directory** option.

Only images that are assigned to graphic items can be automatically scaled. A project may include images that are not assigned to any graphic item. In this case, the tool cannot identify any suitable resolution for such images, and thus they will be generated at their default resolution. The images can be manually scaled by using any suitable application before using them in a NEMA|GUI-Builder project.

SECTION 7

Event Manager

NEMA|GUI-Builder utilizes the Event Manager for managing the events associated to a project. It can be found under the **Project** menu. This enables the user to inspect, add or remove events according to the needs of the project. Through the Event Manager, events can be added by setting following parameters:

- The Trigger (e.g., a button is released)
- The Source item (if applicable)
- The Action when the event is triggered

Depending on the Action that should be executed, more data can be added to an event. For instance, a Screen Transition action needs to know the duration and animation effect, while a Set Value action should know the value (absolute value or percentage) that will be set. These attributes are displayed when an event is created, or an existing one is inspected.

Custom events can also be created and their functionality tailored to the project requirements. This can be performed by selecting a Custom action when creating a new event (or attaching a new action to an existing event). In the generated code, Actions are handled as callback functions. Therefore, these callback functions needs to be completed with the desired code.

These functions can be found in the **custom_callbacks.c** file among the generated files.

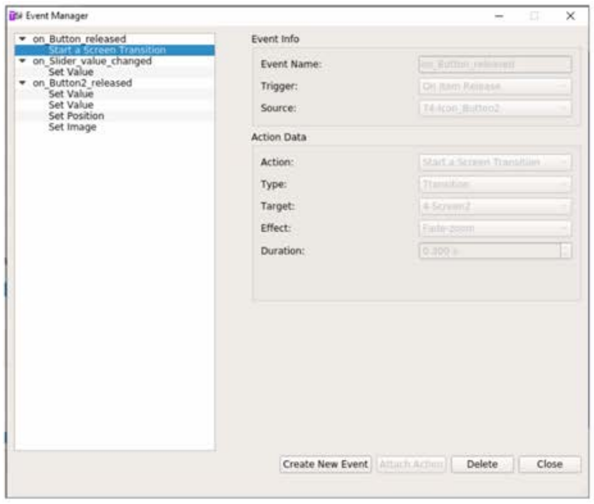
For ease of use, custom actions are divided into four categories:

- **One-Shot**: The callback function is executed instantly when the event is triggered.
- **Periodic**: Actions are performed in a periodic basis. For example, the callback function is executed periodically according to the defined period (e.g., a Digital Meter that needs to set its value every 10 seconds).

- **Transition**: This action can be used to change an attribute of the target item (e,g., opacity) in a continuous way. The Transition has a specific duration, defined by the user and during runtime, it keeps track of its progress.
- **Periodic Transition**: Transitions that are performed every set number of seconds. The user must define both the duration and the period of this action.

After a custom action has been created, it can be configured by changing its corresponding attributes (Type, Duration and Period) in the Event Manager. Figure 7-1 depicts how the events can be managed in the Event Manager.

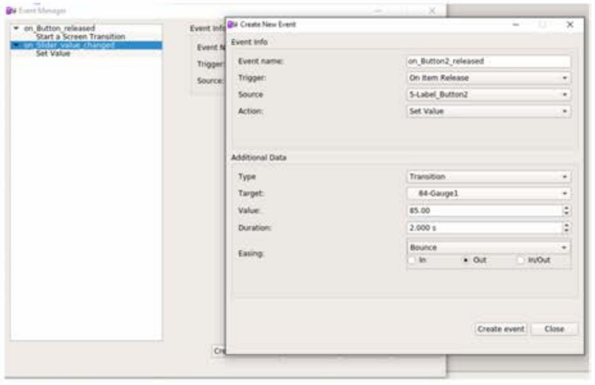
Figure 7-1: Manage Events in the Event Manager



on_Button_released
  Start a Screen Transition
on_Slider_value_changed
  Set Value
on_Button2_released
  Set Value
  Set Value
  Set Position
  Set Image

Event Info
Event Name: on_Button_released
Trigger: On Item Release
Source: T4-Icon_Button2

Action Data
Action: Start a Screen Transition
Type: Transition
Target: 4-Screen2
Effect: Fade-zoom
Duration: 0.300 s



on_Button_released
  Start a Screen Transition
on_Slider_value_changed
  Set Value

Event Info
Event name: on_Button2_released
Trigger: On Item Release
Source: 5-Label_Button2
Action: Set Value

Additional Data
Type: Transition
Target: 84-Gauge1
Value: 85.00
Duration: 2.000 s
Easing: Bounce
In [✓ Out] In/Out

SECTION

8

NEMA|GFX Architecture

NEMA|GFX Library is a low-level library that interfaces directly with the NEMA GPUs and provides a software abstraction layer to organize and employ drawing commands with ease and efficiency. The target of NEMA|GFX is to be able to be used as a back-end to existing APIs (such as OpenGL, DirectFB, or any proprietary one) but also to expose higher level drawing functions, so as to be used as a standalone Graphics API. Its small footprint, efficient design and lack of any external dependencies, makes it ideal for use in embedded applications. By leveraging NEMA's sophisticated architecture, Ambiq MCUs achieve great performance with minimum CPU usage and power consumption.

NEMA|GFX includes 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.

A developer may use only the lower layers of the architecture that provides communication to the NEMA hardware, synchronization and basic primitives drawing. The very thin Hardware Abstraction Layer allows for fast integration to the underlying hardware. The upper low level drawing API acts as a back-end interface for accelerating any higher third-party Graphics API.

NEMA|GFX is built on a modular architecture. These modules are generally stacked one over another, forming a layered scheme. This gives the developer the freedom to tailor the software stack according to ones needs.

The lowest layer is a thin Hardware Abstraction Layer (HAL). It includes some hooks for basic interfacing with the hardware such as register accessing, interrupt handling and more.

The layer above is the Command List Manager. It provides the appropriate API for creating, organizing and issuing Command Lists. This topic is discussed in detail in *Section 8.1 Command Lists on page 30* of this document.

Above the Command List Manager lies the Hardware Programming Layer (HPL). This is a set of helper functions that assemble commands for programming the NEMA GPU. These commands write to the NEMA’s Configuration Register File, which is used to program the submodules of the GPU.

Alongside the HPL resides the Blender module. This module programs NEMA’s Programmable Processing Core. It creates binary executables for the Core. These executables correspond to the various blending modes that are supported by the NEMA|GFX Library.

On top of the NEMA|GFX stack lies the NEMA|GFX Graphics API. This API offers function calls to draw geometry primitives (lines, triangles, quadrilaterals etc.), blit images, render text, transform geometry objects, perform perspective correct texture mapping etc. When using NEMA|GFX as a back-end for a third party Graphics API, much of the NEMA|GFX Graphics API may be disabled.

For additional information on NEMA|GFX features and architecture, please see the NEMA|GFX - API document on the Think Silicon website at:
https://www.think-silicon.com/?section=2889&language=en_US

8.1 Command Lists

In order to decouple CPU and GPU execution and achieve both better performance and lower power consumption, NEMA GPUs incorporate an advanced Command List Processor (CLP), capable of reading entire lists of commands from the main memory and relay them to the Configuration Register File. The CPU pre-assembles Command Lists (CL) prior to submitting them to the Command List Processor for execution, while a single Command List can be submitted multiple times. This approach alleviates the CPU from recalculating drawing operations for repetitive tasks, resulting in more efficient resource utilization. The steps for writing commands to the Configuration Registers through the Command List Processor are the following:

1. The CPU assembles a Command List, through the NEMA|GFX Library.
2. The CPU submits the Command List for execution. The Command List Processor is informed of a pending Command List.
3. The Command List Processor reads the Command List from the System Memory.
4. The Command List Processor relays the commands to the Configuration Register File.

Command List (CL) usage facilitates GPU and CPU decoupling, while its inherent re-usability greatly contributes to the decrease of the computational effort of the CPU. This approach renders the overall architecture capable of drawing complicated scenes while keeping the CPU workload to a minimum. The design principles of CLs allow developers to extend the features of their application while optimizing its functionality. CL is capable of jumping to another CL, thus forming a chain of seamlessly interconnected commands. In addition, a CL is able to branch to another CL and once the branch execution is concluded, resume its functionality after the branching point.

The NEMA|GFX Library helps developers to easily take advantage of all of these features through certain basic function calls that trigger the whole spectrum of CL capabilities. A list of the most fundamental subset of CLs is listed in the following subsections.

8.1.1 Create

The most straightforward command for initiating a simple coding example is the "Create" command which is listed below:

```
nema_cmdlist_t nema_cl_create (void)
```

This fundamental command allocates and initializes a new Command List for later use.

8.1.2 Bind

This command sets the referred Command List as active. Each subsequent drawing call will incrementally be incorporated in the active Command List. At any time, all drawing operations should be called when there is a bound Command List.

```
nema_cl_bind_cmdlist (nema_cmdlist_t *cl)
```

8.1.3 Unbind

Unbind the currently bound Command List.

```
nema_cl_unbind_cmdlist (void)
```

8.1.4 Submit

Submit the referred Command List for execution. If this CL is currently the one that is bound, this call unbinds it. When a CL is submitted for execution, it should never be altered until it finishes execution.

```
nema_cl_submit_cmdlist (nema_cmdlist_t *cl)
```

Writing in such a CL results in undefined behavior. A typical routine for drawing would be the following:

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

/* Drawing Operations */              // Draw scene

nema_cl_unbind_cmdlist ();            // Unbind CL (optional)
nema_cl_submit_cmdlist (& cl);        // Submit CL for execution
```

8.2 Textures

Texture is the digital representation of an object's surface. In addition to two-dimensional qualities such as color and transparency, a texture also incorporates three-dimensional ones such as reflectiveness.

- **Filtering**: pointer sampling, bilinear filtering
- **Wrapping Mode**: clamp, repeat, border, mirror

8.2.1 Binding Textures

The hardware allows a single shader to read from and/or write to 4 different textures. These textures must be bound before the Shader is submitted for execution. NEMA|GFX uses pre-assembled Shaders to perform blending operations.

Table 8-1: Shader Conventions

Textures	Description
NEMA_TEX0	Destination/Background Texture
NEMA_TEX1	Foreground Texture
NEMA_TEX2	Background Texture
NEMA_TEX3	Depth Buffer

```
void nema_bind_dst_tex (uint32_t baseaddr_phys,
                         uint32_t width , uint32_t height,
                         nema_tex_format_t format , int32_t stride)
```

The above function binds a texture to serve as a destination. The texture's attributes (GPU address, width, height, format and stride) are written inside the bound CL. Each subsequent drawing operation will influence this destination's texture. Most common graphics operations include image blitting (copying). This includes drawing a background image, GUI icons, and font rendering. The following command binds a texture to be used as foreground:

```
void nema_bind_src_tex (uint32_t baseaddr_phys ,
                        uint32_t width , uint32_t height,
                        nema_tex_format_t format , int32_t stride,
                        nema_tex_mode_tmode);
```

8.3 Color Formats

NEMA GPUs natively support a large set of texture formats. They are capable of performing fast read and write operations by executing on-the-fly color conversion/decompression. Native formats expand from full 32-bit RGBA to 1-bit black and white colors, together with an optional proprietary compressed 4-bit-per-pixel lossy format. These can all be used as source or destination textures. The list of all supported formats is presented in Table 8-2 below.

Table 8-2: Supported Color Formats

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 no transparency
RGBA5650	16-bit color with transparency
RGBA5551	16-bit color 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
TSC4	4-bit proprietary compressed
YUV	YUV
Z24_8	32-bit (24+4) depth and stencil
Z16	16-bit depth

8.4 Blending

Alpha 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.

The NEMA Graphics Core is a programmable VLIW processor, which allows rapid calculations between colors. Normally, such per-fragment calculations are an overwhelming computational burden for a CPU or MCU. The NEMA Graphics Core is programmed through instructions in binary form, called Shaders. However, in embedded applications, running a compiler for creating these kinds of Shaders is not a realistic scenario.

NEMA|GFX Library provides a lightweight and user-friendly interface that employs pre-assembled commands to create a powerful set of blending algorithms. NEMA Graphics Core can be programmed through the following functions:

```
void nema_set_blend_fill (nema_blend_mode_t blending_mode)
void nema_set_blend_blit (nema_blend_mode_t blending_mode)
```

These functions are defined in NemaGFX_blender.h file. They should be used on fill and blit operations respectively. The blending_mode argument is possible to be a predefined blending mode or a more refined User Defined Mode.

8.4.1 Notations and Conventions

Blending requires a series of calculations between the source and destination color fragments for producing the final color, which will be written in memory. The Color and Alpha channels are noted as follows:

- S_c: Source Color
- S_a: Source Alpha
- S_f: Source Blend Factor (multiplier)
- D_c: Destination Color
- D_a: Destination Alpha
- D_f: Destination Blend Factor (multiplier)
- F_c: Final Color
- F_a: Final Alpha
- C_c: Constant Color
- C_a: Constant Alpha

The Color and Alpha values range from 0 to 1. Therefore, each calculation result is also constrained to the same range. For consistency, Color and Alpha calculations are always described separately, since these calculations may not be identical. When a constant color is used (noted as C_c and C_a), it can be set using the following function:

```
void nema_set_const_color (uint32_t rgba)
```

8.4.2 Predefined Blending Modes

Predefined Blending Modes are a set of commonly used modes, each implying different calculations between the source and destination colors for Color (RGB) channel and Alpha channel respectively. Table 8-3 presents the complete list of the available Predefined Blending Modes along with the corresponding calculations that produce the final fragment color. Figure 8-1 shows the result for each Predefined Blending Mode.

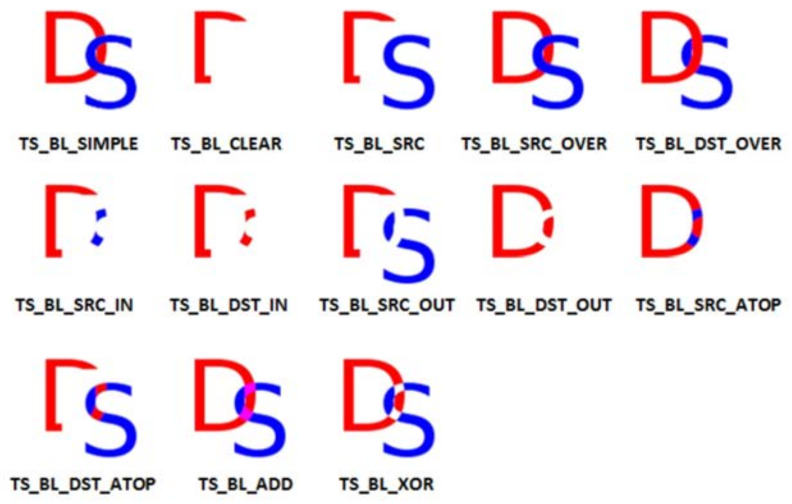
Table 8-3: Predefined Blending Modes

Predefined Blending Modes	RGB	ALPHA
NEMA_BL_SIMPLE	$Sc * Sa + Dc * (1 - Sa)$	$Sa * Sa + Da * (1 - Sa)$
NEMA_BL_CLEAR	0	0

Table 8-3: Predefined Blending Modes *(Continued)*

Predefined Blending Modes	**RGB**	**ALPHA**
NEMA_BL_SRC	*Sc*	*Sa*
NEMA_BL_SRC_OVER	*Sc* + *Dc* * (1 - *Sa*)	*Sa* + *Da* * (1 - *Sa*)
NEMA_BL_DST_OVER	*Sc* * (1 - *Da*) + *Dc*	*Sa* * (1 - *Da*) + *Da*
NEMA_BL_SRC_IN	*Sc* * *Da*	*Sa* * *Da*
NEMA_BL_DST_IN	*Dc* * *Sa*	*Da* * *Sa*
NEMA_BL_SRC_OUT	*Sc* * (1 - *Da*)	*Sa* * (1 - *Da*)
NEMA_BL_DST_OUT	*Dc* * (1 - *Sa*)	*Da* * (1 - *Sa*)
NEMA_BL_SRC_ATOP	*Sc* * *Da* + *Dc* * (1 - *Sa*)	*Sa* * *Da* + *Da* * (1 - *Sa*)
NEMA_BL_DST_ATOP	*Sc* * (1 - *Da*) + *Dc* * *Sa*	*Sa* * (1 - *Da*) + *Da* * *Sa*
NEMA_BL_ADD	*Sc* + *Dc*	*Sa* + *Da*
NEMA_BL_XOR	*Sc* * (1 - *Da*) + *Dc* * (1 - *Sa*)	*Sa* * (1 - *Da*) + *Da* * (1 - *Sa*)

Figure 8-1: Predefined Blending Modes



Drawing a translucent red rectangle would require the following calls:

```
nema_set_blend_fill(NEMA_BL_SIMPLE);
nema_fill_rect (52, 52, 14, 4, nema_rgba(0xff, 0, 0, 0x80));
```

TIP: When in doubt, usually the NEMA_BL_SIMPLE blending mode is the safest choice. The overall process starts with an empty Framebuffer and then performs blitting on the textures.

8.4.3 User Defined Blending Modes

Developers can create custom blending modes by using different factors for the source and destination color, through the following function:

```
nema_blend_mode_t nema_blending_mode(nema_blend_factor_t src,
                                      nema_blend_factor_t dst,
                                      nema_blend_op_t ops);
```

The ops argument of the above function refers to additional operations, should be set to zero (0). The calculations result to the final color:

$Fc = Sc \cdot Sf + Dc \cdot Df$
$Fa = Sa \cdot Sf + Da \cdot Df$

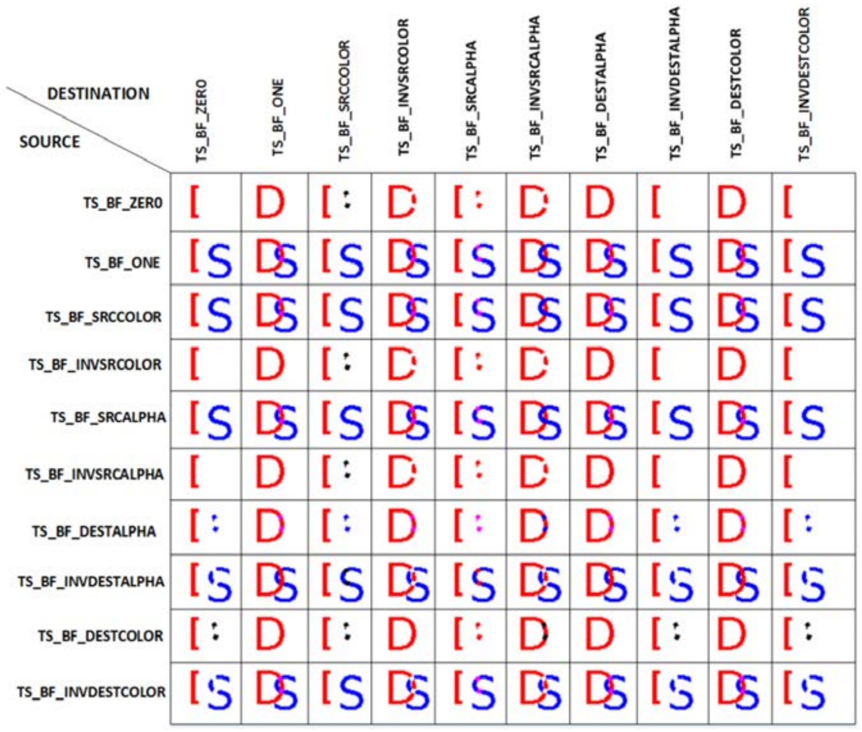
The Blend Factors are listed in Table 8-4. Figure 8-2 on page 37 shows the available custom blending modes. As a result, the previous example may be rewritten as:

```
nema_set_blend_fill(nema_blending_mode (NEMA_BF_SRCALPHA,
NEMA_BF_INVSRCALPHA , 0));
nema_fill_rect(52, 52, 14, 4, nema_rgba(0 xff , 0 , 0 , 0x80));
```

Table 8-4: Blend Factors

Blend Factors	Blend Factors (*Sf* or *Df*)
NEMA_BF_ZERO	0
NEMA_BF_ONE	1
NEMA_BF_SRCCOLOR	*Sc*
NEMA_BF_INVSRCCOLOR	(1 - *Sc*)
NEMA_BF_SRCALPHA	*Sa*
NEMA_BF_SRC_INVSRCALPHA	(1 - *Sa*)
NEMA_BF_DESTALPHA	*Da*
NEMA_BF_INVDESTALPHA	(1 - *Da*)
NEMA_BF_DESTCOLOR	*Dc*
NEMA_BF_INVDESTCOLOR	(1 - *Dc*)
NEMA_BF_CONSTCOLOR	*Cc*
NEMA_BF_CONSTALPHA	*Ca*

Figure 8-2: User Defined Blending Modes



8.4.4 Additional Operations

NEMA|GFX Library allows the following operations, which can be applied together with the previously mentioned blending modes through the function:

```
nema_blend_mode_t nema_blending_mode (nema_blend_factor_t src,
                                      nema_blend_factor_t dst,
                                      nema_blend_op_t ops)
```

The additional supported operations are listed in Table 8-5 below.

Table 8-5: ops Arguments

ops Arguments	Description
SRC_MODULATE_A	Multiply source alpha channel with *Ca* constant before blending. *Ca* is defined by calling NemaGFX_set_const_color().
SRC_FORCE_A	Replace source alpha channel with *Ca* before blending. Overrides SRC_MODULATE_A option. *Ca* is defined by calling NemaGFX_set_const_color().
SRC_COLORIZE	Multiply source color channels (RGB) with *Cc* before blending. *Cc* is defined by calling NemaGFX_set_const_color().
SRC_COLORKEY	Ignore fragment when source color matches the source color key, which is defined by calling NemaGFX_set_src_color_key().
DST_COLORKEY	Ignore fragment when destination color matches the destination color key, which is defined by calling NemaGFX_set_dst_color_key().

SECTION

9

Memory Footprint

9.1 Module Footprint

The memory requirement of an application depends on the features incorporated in it.

When the code is generated, a report file (report.txt) is created along the generated files that contains the memory requirements for each font and image of the project. This file can be reviewed and some project parameters can be revised (e.g., using compressed image formats instead of RGBA8888, use less bits per pixel in fonts etc.) to reduce the memory consumption.

The generated code consists of the project specific files (generated images, widgets, fonts, framebuffers and events) and a library that orchestrates the generated application's runtime. It contains the C implementation of all of the widgets, gestures, interactions, and screen transitions, that are common for every generated project. This library is labeled NemaGUI and it can be found in the generated files of a project.

NemaGUI consumes 25KB of memory with all of its features (widgets, animations, and event types) included. This can be split into 12KB for its core and 13KB for all the widget implementations. For example, a "Hello World" application with a simple screen (no images and fonts), and excluding the framebuffer, can use as little as 27KB. Additional features will increase the memory consumption of the application.

9.1.1 Framebuffers

The memory usage of framebuffers is application specific and depends on the following:

- The number of framebuffers an application makes use of (single framebuffer, double buffering, and animation buffers)
- The resolution (W×H)
- The format (compressed/uncompressed)

Table 9-1 lists the currently supported formats and associated memory usage.

Table 9-1: Supported Framebuffers

Framebuffers	Bytes/Pixel
RGBA8888	4
RGBA5650	2
TSC4	0.5
TSC6	0.75

9.1.2 Images

The memory usage of images is also application specific. The png, jpg, and svg image file types can be imported and used in the design. At code generation, they are converted to formats suitable for the GPU:

Table 9-2: Supported Images

Images	Bytes/Pixel
RGBA8888	4
RGBA5650	2
RGBA5551	2
RGBA4444	2
RGBA3320	1
TSC4	0.5
TSC6	0.75
TSC6A	0.75
L8	1
A8	1

A generated image consumes (in bytes) W*H*bytes_per_pixel. To minimize the memory footprint, big images are automatically scaled down if it is suitable. For example, if a large png image asset with a resolution of 1024×768 is used in a widget with actual resolution of 70×70, then the generated image will be 70×70 as well.

9.1.3 Fonts

The memory usage of fonts depends on how many fonts will be used. The footprint of a font depends on its:

- Size (the bigger the font the more memory it consumes)
- Bits per pixel (more bits per pixel result in fonts that consume more memory)
- Unicode Ranges (the default range is ASCII, more ranges contain more characters and therefore need more memory)
- Kerning (if kerning is enabled it requires memory to store the kerning pairs)

These parameters are configurable via the Font form (see *Section 5.5 Fonts on page 24*) in NEMA|GUI Builder.

9.2 Widget Footprint

Table 9-3: Widget Footprint

Widget	Bytes
Screen	60
Circle	56
Rectangle	60
Rounded Rectangle	60
Image	60
Label	84
Container	64
Table	56
Container Array	56
Window	56
Label Button	72
Icon Button	72
Radio Button	72
Checkbox	72
Horizontal Slider	68
Vertical Slider	68
Digital Meter	92
Icon	56
Horizontal Progress Bar	76
Vertical Progress Bar	76
Gauge	100
Circular Progress	76
Watch Face	76

SECTION

10

Frequently Asked Questions

This section shows some of the frequently asked questions.

1. Are all the NEMA|GUI Builder features included in the package?

   Yes. The Apollo4 Display Kit includes a 45-day evaluation period with the PRO Edition of NEMA|GUI-Builder licensed through Think Silicon S.A.. The Standard Edition which includes all features with limitations is available for download on the Think Silicon website.

2. The project is compiled without errors but, when running the application, the display does not work.

   This issue may be caused by one of the following:

   - Stack size is too low.
   - Wrong initialization of the display controller.
   - Wrong configuration of the display interface.

   For additional inquiries on the Apollo4 Display Kit, please visit our technical support page: https://support.ambiq.com/

A-SOCAP4-UGGA03EN v1.0
August 2021