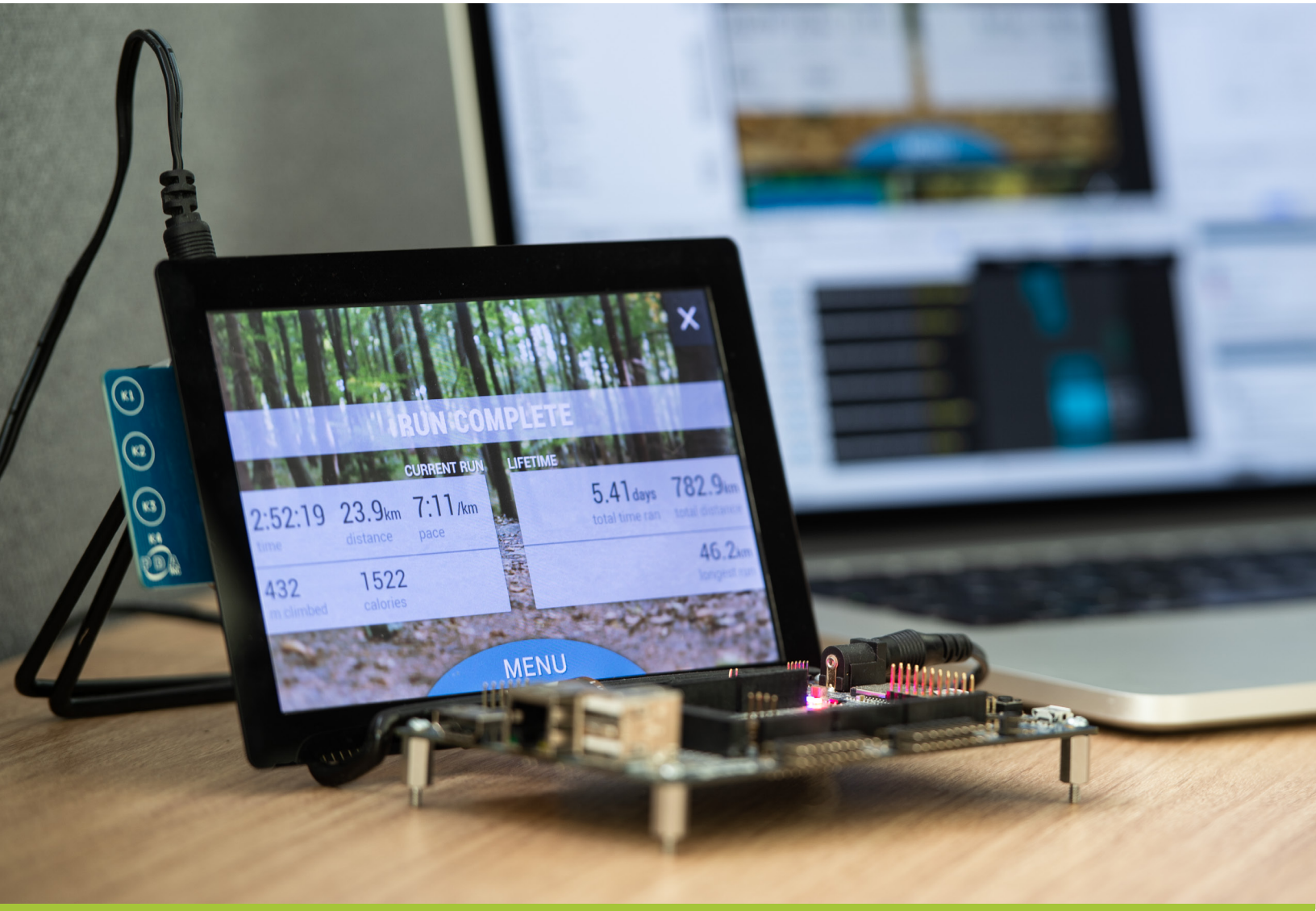
CRANK SOFTWARE

Choosing The Correct Hardware Platform For Embedded GUI Applications



Best Practices Using Storyboard™

Vol. 1

CRANK SOFTWARE

Selecting your Applications Hardware Platform

When building an embedded system there are many choices you will face when deciding the correct hardware and software combinations to best get your product to market. Hardware capabilities, performance, power requirements, memory types, OS implications, packaging options, platform scalability and interoperability are just some of the considerations you will have to work through in the early stages. Most of these decisions can be answered with research, testing, and peer discussion. When it comes to your GUI application however, we're committed to ensuring that it is created using all the best practice principles our customers have learnt along the way.

In this *Best Practices Using Storyboard* guide, our goal is to ensure your Storyboard application is created from the onset with these best practices in mind, and optimized for the hardware platform you've chosen. As a

platform-agnostic framework, Storyboard can be used to build applications on a wide range of hardware platforms, and as such, your decision to use Storyboard should not influence which hardware you finally select.

Nevertheless, it's critical that you understand the changing landscape of the MCU and MPU market, their capabilities and OS implications. In this document, you will learn the questions you need to consider when assessing your embedded UI potential against your desired user experience.

The three considerations for optimal embedded GUI hardware selection

Embedded systems designers typically have three areas of consideration they must break down and understand when trying to decide which hardware platform suits their embedded GUI requirements: They include:

1. **Is the platform powerful enough to run the GUI?**
2. **Does the system have the memory it needs?**
3. **What will be the inevitable overall storage requirements?**

Once you have satisfactory answers, you'll be in a better position to ensure you select the most suitable hardware. You'll also feel confident in reassuring other team members that the platform selected is capable of achieving your end goal: a brilliant user experience on your embedded device.

Determine Performance Requirements

Test to understand potential hardware performance

One of the first aspects you should address when striving to make a device with a great user experience is determining if the system being evaluated is powerful enough to render the graphics needed while still performing all the other tasks the system needs. Over the years we have seen many embedded project teams that encountered issues due to assumptions that they had a system with the horsepower they required. When it's late in the development cycle, you may end up having to make compromises in design in order to meet deadlines and therefore risk going to market with an inferior product.

As a first step, it's generally best to get a feeling for the capabilities of the hardware you're considering. Since Storyboard is platform-agnostic, you can run the same embedded application across multiple platforms to test the experience live and gather metrics to aid in your decision.

Test GUI samples live on your hardware

To help you get a visual understanding of the capabilities of the various hardware platforms, our own development team has included a number of sample GUI applications within Storyboard and others within a public SVN repository, specifically built and optimized for unique hardware models, for you to download and try. Once you've found a suitable application you can run it on your hardware to see how it performs and behaves. Doing this will allow you to obtain a better understanding of what the platform is

As a cross-platform framework for embedded UI development, Storyboard works with MCUs and MPUs on multiple hardware platforms. Download free interactive UI samples for your hardware from cranksoftware.com/platforms/demo-images.

capable of with minimal effort. If you are considering a range of hardware platforms, this step should help you narrow your hardware choices to a select few from which you can start creating some more test cases specific to the requirements of your final product.

Ideally, you can find a demo image that matches or is close to your hardware platform since this will provide you with the easiest path forward. Our demo images are SD card images or binaries that can be deployed straight to the hardware. They are also configured to allow easy deployment and testing via SCP transfer so you can quickly drop new applications onto the hardware for your evaluation process.

Understand performance benchmarks

Once you have a general idea of what your hardware is capable of, the next step should be to create some realistic benchmarks tailored to your embedded project. The goal of this isn't to build a production-ready application, but to pick a handful of UI elements that you envision being resource intensive. This way, you'll see how they run on the hardware and gather additional system metrics to further influence your hardware selection.

Even with low-frequency devices, like wearable devices, consumers expect strong user experiences. Benchmarks focused on resource-intensive elements of the design can help ensure smooth UI interactions and performance.

For example, consider a smartwatch or a fitness tracker. There are plenty of display modes that display static information or data with lower update intervals. However scrolling menu lists or the plotting of "vitals" data might be more resource-intensive, and therefore great candidates for building some targeted benchmarks around. This way you can make sure that your system has the resources it needs to run any critical tasks in the background, while still providing smooth UI interactions and performance for the user.

Pro Tip: Use scrolling content to benchmark performance

Scrolling content is a great benchmark for testing performance. It impacts a larger portion of the display and requires constant performance to maintain smooth movement (more pixels to be redrawn means more CPU usage time). Plus any delays in movement / motion will show up quickly when performing the dragging movement of the scrolling content.

RESOURCES

Where to start when building a test case or benchmark?

Here are some recommended resources you should consider when building a test case or benchmark for the hardware you are considering.

Import sample project from Storyboard sample

These snippets are designed to demonstrate how to use features of Storyboard including render extensions, actions, animations, or other visual effects.

LEARN MORE

Import sample project from public SVN

Access to additional samples that might not be included with your installation of Storyboard.

LEARN MORE

Download a sample Storyboard GUI demo images

Ready made sample UI applications for the popular hardware platforms to download and flash it straight to your board.

LEARN MORE

Measure application performance

Measuring performance of your GUI application is an important next step in the hardware evaluation process because it provides real data to back up any assumptions, and provides more quantitative data on which to make final decisions. When doing this there you will want to answer three questions:

1. **How is the Storyboard GUI application performing?**
2. **What's the load on the system?**
3. **How much, if any, hardware acceleration is required?**

Capture performance logs and metrics, including redraw times and loading times, with a push of a button within Storyboard, speeding up the process of measuring application performance.

Evaluate application performance

When you begin to interact with the Storyboard application, you begin to develop a sense of the performance based on how the UI feels and reacts.

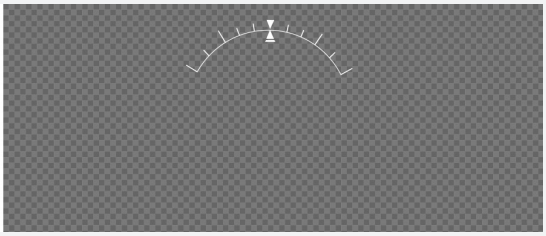
- Are animations smooth and snappy?
- Is scrolling stutter-free and locked to my finger?
- Does the UI react quickly to my input?

Assessing these criteria will help you start to understand the impact of these interactions on the app performance. Generally, the cause of any poor performance can be boiled down to two keys aspects:

- rendering time (a major influencer on framerates), and
- response time (how quickly the UI can react to events).

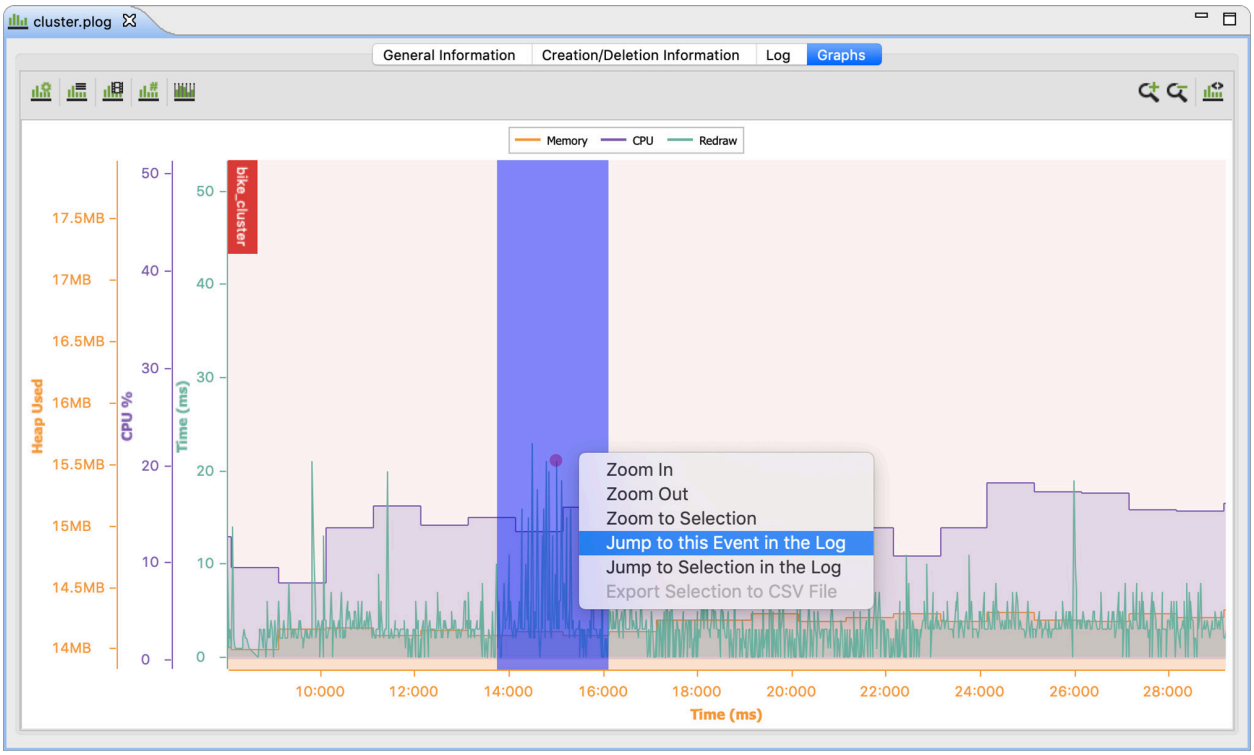
Pro Tip: Use of partial or full transparency in images

One of the key performance issues related to image rendering and composition is the use of full or partial transparency. Large areas of full transparency can result in many CPU or GPU cycles consumed needlessly traversing pixel areas. Similarly images that are fully opaque but in a format that indicates they may have some level of per pixel level transparency can be optimized at design into a different format.

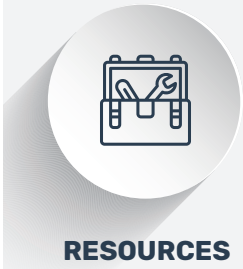


The image **bank_angle.png** shown on the right uses dimensions that match the display size of the cockpit application which unnecessarily uses large areas of full transparency which results in a negative impact on performance.

With the performance logging, Storyboard provides the tools to assist in getting greater visibility into the causation of performance issues. The Storyboard engine has instrumentation in place that, when enabled, provides a detailed breakdown of the event flow and the associated timing for each action that is executed. Data metrics from the log can also be plotted in graphical form, allowing you to easily spot problem areas in need of optimization.



Graphical representation of data captured within Storyboard's performance logs makes it easier to identify the valuable insight required for making smart UI design choices that do not impact the user experience.



RESOURCES

Where to start with determining Storyboard application performance?

Here are some recommended resources you should consider when looking to leverage Storyboard's built-in performance tools.

Creating and importing a performance log

A video explaining the available command line options and demonstrating how to add the logging command line option to sbengine.

LEARN MORE

Visualizing Storyboard's performance logs in a graphical format

Details on how to provide context to the captured record tracing being performed within the Storyboard engine.

LEARN MORE

With the ability to gather data on the application's performance, you can start to look at the performance of the system as a whole. Since your Storyboard application needs to run in conjunction with other platform systems you should ensure that there are extra system resources available. With the techniques described above you should be able to run some tests to benchmark how your system performs with multiple applications running at the same time. If you aren't happy with the overall performance, you can then make adjustments to the UI implementation, design and other applications running on your system.

Measure the system load

Once you are satisfied with the GUI and hardware performance benchmarks, it's time to look at the system as a whole. This will enable you to understand how the rest of your software will fit alongside your device's GUI application. While this tends to be specific to the hardware and operating system (OS) selection, the general goal is to gauge CPU usage. Most OS' will provide tooling to help you measure the total amount of CPU cycles used, and which processes or tasks are responsible for that system draw.

For example, if you are running Linux, you have access to the top utility. Or if you are running QNX, there are different options available to you in the Momentics tool suite. The same applies to the real time executives like FreeRTOS. Your software and compiler choice will determine the specific tooling solutions available to you.

Regardless of the platform tooling you use to assess system load, your end goal is to be in a position where your GUI and other application tasks can all run concurrently at the performance levels you require.

RESOURCES

Where to start when measuring system performance?

Here are some recommended resources you should consider when leveraging Storyboard to help measure the performance of your hardware.

Using real-time insight from Storyboard and Percepio to enhanced UI development

This webinar demonstrates how Storyboard along with Percepio's Tracealyzer can be used for common scenarios in UI performance analysis and optimization.

LEARN MORE

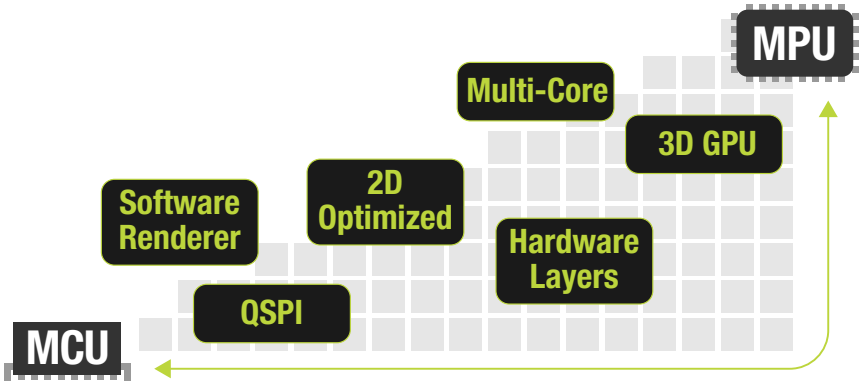
Building Embedded GUIs for MCUs versus MPUs

This video discusses the differences in developing GUI applications for MCUs vs. MPUs and what to consider when evaluating embedded hardware options.

LEARN MORE

Determine your graphics acceleration requirements

Storyboard lets you choose how you want to render graphics, from writing directly into the framebuffer to taking advantage of any 2D or 3D hardware rendering available on the board.



Storyboard's unique architecture allows it to be compatible with MCUs, crossover processors and MPUs, and a wide range of graphic rendering technologies, providing the freedom to choose the best hardware the embedded product.

The addition of some form of hardware acceleration can help offset your CPU usage since graphic-based tasks, normally handled by the CPU, can be offloaded to the graphics processing unit (GPU). This hand-off of graphics processing will free up your CPU for other system tasks while providing a boost in performance. However, if your chosen hardware doesn't have an on-board GPU you can still leverage Storyboard's support for software rendering to use the CPU to draw content on the screen.

Should you select a platform with a GPU built-in, Storyboard supports a wide range of render engines from ChromeART (DMA2D) to OpenGL and many other choices in between. But be aware that with GPUs you will require more power from your device. For battery powered devices, this

should be carefully considered as you will want to ensure maximum battery life while still getting the best UI performance possible. In this case a GPU may or may not make sense for your product.

It is not necessary to have a GPU if your GUI application consists of only 2D images. It is really only necessary to add hardware acceleration when you have 3D content, or when you notice your graphics are struggling during rendering.

Many of the available platforms come with two runtime versions; OpenGL and Software Renderer. If you aren't sure which runtime is best for the application, you can experiment on both and then compare the performance and resource consumption. Performing this extra step might uncover the fact you can get away without the OpenGL hardware after all, saving you on additional hardware costs (and extending the life of that battery-powered device).

If you find you require acceleration, or have screens with a mix of 3D and 2D content, look for a hardware platform that offers different tiers of GPU inside. In instances where this occurs, Storyboard's Hybrid Rendering functionality can be leveraged here, providing the ability for the GUI to automatically toggle between 2D and 3D GPU rendering on the fly, while conserving power consumption.

RESOURCES

How to enhance UI applications with 3D content?

Here are some recommended resources to provide a better understanding of how you can leverage the capabilities of hardware acceleration to help elevate your UI.

Learning Storyboard 3D rendering fundamentals

Understanding the underlying concepts to help with making informed decisions when configuring 3D Model render extensions in Storyboard Designer.

LEARN MORE

How to create a 3D model application in Storyboard

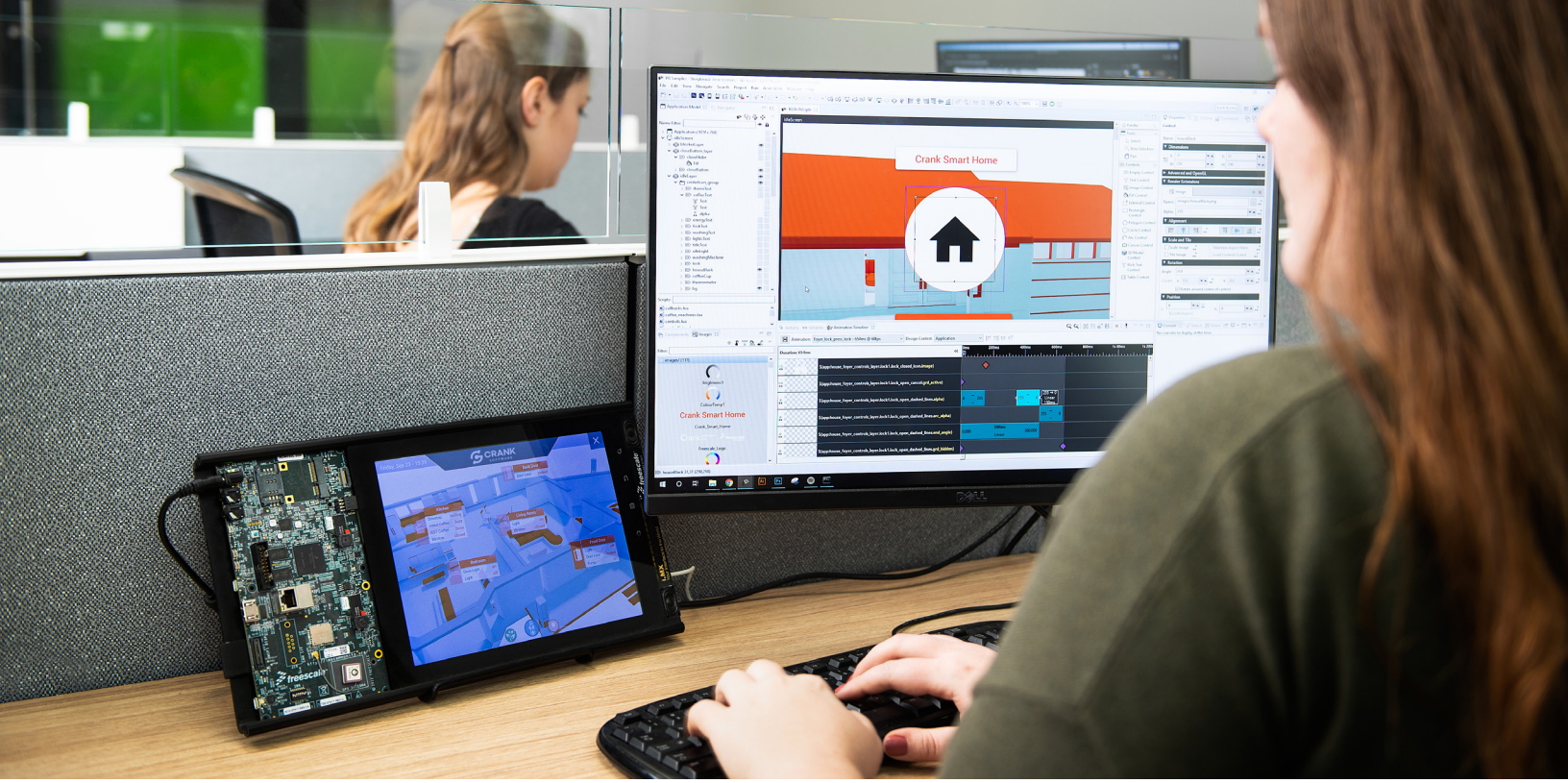
A tutorial showing how to use the 3D Model Control with a new Storyboard project.

LEARN MORE

Working with OpenGL shaders to display 3D model objects

Leveraging OpenGL hardware to transform model elements, create custom GLSL shader effects and to display 3D model objects.

LEARN MORE



Estimating System Memory Needs

Now that you have gauged the expectation of the CPU and GPU, the next step is to understand how much memory your GUI application will require. Similar to how important it is to understand how much processing power your product requires, it is equally important to be aware of the amount of memory needed. Too little and you'll find that you will have to make design concessions. Too much and your bill of materials cost will be unnecessarily high.

When it comes to determining the amount of required memory there are four items to consider:

1. **Display resolution**
2. **Buffering**
3. **Fonts**
4. **Number of images**
5. **Location of where images will be loaded from**

Display resolution and framebuffers

In embedded devices, framebuffers are a contiguous block of memory that contain the colour data for each pixel of the display's resolution. A minimum of one framebuffer is required to display graphics to the device's screen. The display resolution and configuration (colour depth) will determine how much memory you require to allocate for your buffer(s).

In general, most framebuffers use either full 32- or 16-bit color depths for a higher color depth equates to a more visually appealing image. To determine how much memory is required for the framebuffer, you would need to multiply the display resolution by the required color depth. Since it would most likely be 32- or 16-bit color you can multiply the pixels by 4 or 2, respectively, to learn how many bytes of total space are required.

16-bit (RGB565) color means you will have a reduced spectrum of color and lack an alpha channel support. It's important to note that the color depth used needs to align with what the LCD display supports. You can use your desired colour fidelity to help select a compatible display or have the determine your framebuffer configuration.

Pro Tip: 16-bit color for MCUs

In general, the only time 16-bit color is noticeable to the human eye is in gradients where an effect called banding can show up. Most MCU systems use this color space since it reduces the required memory footprint in half, yet still provides good visual quality for the user.

Formula for calculating framebuffer memory requirements for different display resolutions

Framebuffer Memory Required = Display Resolution x Bit Color

Quotient (4 = for 32-Bits, 2 = 16-Bits)

Example Worksheet For 480 x 272 Display:

32-Bit memory calculation	**16-Bit memory calculation**
(480 x 272) x 4 = 522,240 Bytes	(480 x 272) x 2 = 261,120 Bytes
~522 Kilobytes framebuffer memory required	~261 Kilobytes framebuffer memory required

While one framebuffer is required for graphics, for fluid animations and smooth blending within GUI applications, it is recommended you have two framebuffers. Two buffers can also help avoid a visual effect known as "tearing" where a display device shows information from multiple frames in a single screen at the same time causing the image to look like it is torn. By using two framebuffers instead, the display reads from one buffer while the engine writes to the other. When the frame is done, Storyboard then flips the buffer it is writing to and tells the display to read from the buffer which is no longer being written to. Having two framebuffers doubles the amount of memory you require to support the graphic rendering of your GUI application but it can improve visual fidelity and improve performance.

For example, the resolution of 480 pixels wide by 272 pixels high, using a 32-bit color gamut would need 522,240 Bytes x 2 = 1,044,480 Bytes ~1.044 Megabytes.

Framebuffers can be configured in other ways than the double buffering method (also referred to as page flipping) that was just discussed. In cases where a double framebuffer is required, but the hardware can't support the flipping of buffers, Storyboard supports a uniquely designed version of double framebuffer called "back buffering". With back buffering two framebuffers are still utilized; however, instead of flipping between the two buffers with backbuffers the information is written to a buffer in the background (i.e. the backbuffer) and then copied to the front buffer from which the screen displays from.

If memory is not an issue with the hardware you have selected, there is a buffer configuration known as triple buffering. Triple buffering is a combination of page flipping and backbuffering that leverages two "backbuffers" as page flips to minimize latency between copying content between the back buffer(s) and the front buffer.

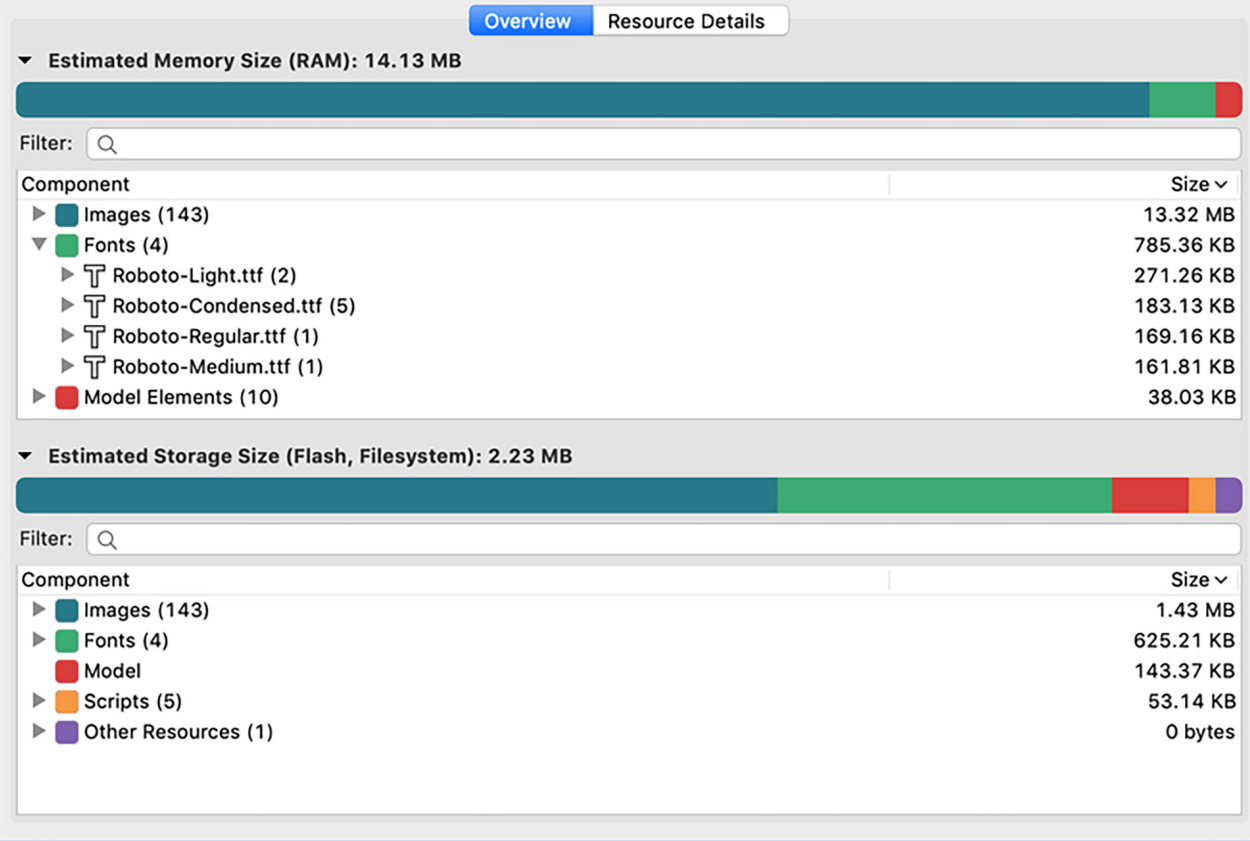
Regardless of which buffer technique that you choose to use in your embedded device, the Storyboard engine supports all standard buffer configurations enabling your UI to be optimized for your chosen hardware.

Visual assets (fonts and images)

Storyboard will do it's best to minimize resource consumption; however, it's important to understand that the visual complexity of your application will impact your memory consumption.

When it comes to visual complexity, the biggest factors that impact memory are your images and fonts. Both these assets need to be loaded (in some cases decompressed) from the file system into RAM, then composited together with other content, and finally written to the

framebuffer. By default, Storyboard will cache images and font glyphs in order to improve rendering performance the next time the asset needs to be redrawn. However, that performance boost does come at the cost in terms of total consumed memory which will need to be accounted for. Plus, for those images that start off as a compressed file once they are decompressed this will add to the total consumed memory as well.



Storyboard's Metrics View shows you how much memory and storage will be used by the resources within your embedded application, and will help you save and redefine your touch optimized UI where needed.

For example, you might have a background image that is the entire size of your display (480 x 272) with a file system size showing ~90kb. Calculating the cost of this is similar to determining the cost of a framebuffer (width x height x color depth). When you consider how many images are being used to make up your UI, it is easy to see how this can quickly add up.

When it comes to fonts, while they are different from images, they too can have a sizable impact on the consumption of memory. This is because, with fonts, each font file contains the total definition of each and every glyph in the supported character set. When it comes to displaying the text on the screen, Storyboard will parse the string of characters and generate the required glyphs for each character required. These glyphs are based on the point size and style of the font and will be cached by Storyboard for rendering optimization purposes. So, if your text content is dynamic, or if you have multiple styles of fonts, or different font sizes to display at a single time then you'll be caching for each option and end up accumulating a large cache of glyphs.

By default the engine doesn't limit the amount of memory you can use to cache resources, as Storyboard will favor performance when resources are available. However, if you are running closer to the limit of what your system has for memory then you can always set limits to how much memory you can allocate for resource caching. Therefore if you should reach this defined upper limit, the Storyboard engine will release assets based on their frequency of use in order to make room for new visual assets that need to be rendered.

Calculating memory usage

Knowing how to calculate the memory usage of a particular resource is useful for it can guide in the making of smarter, more efficient GUI design and development choices. To make this easy, Storyboard's Metrics View panel keeps track of the resource requirements of your application. Within, you can get a breakdown of what you might expect the usage to be, based on the layout of the application. You can also create a resource export configuration that can be the basis of these calculations since the format in which you choose to store your assets will affect how the impact system memory. An excellent example of this is if your system support is after storing your application in Flash memory. You can configure Storyboard to export your images in flash memory uncompressed and then draw them directly from this location. This bypasses the need to decode a compressed image (jpg or png) into RAM and can save you resources and, in some cases, improve performance. To learn more about this, please refer to the following Crank Software support article.

However, not all aspects of the Storyboard built UI are displayed within the Metrics View panel. Lua scripting and dynamic behavior items, like animations, are outside the estimation scope provided by the Metrics View. It will still provide a pretty close estimation of the memory consumption, enabling a decent estimate of your memory requirements.

RESOURCES

How to determine general memory consumption?

Here is a recommended resource to provide a better understanding of how to measure memory usage in Storyboard.

Estimating UI memory and storage size

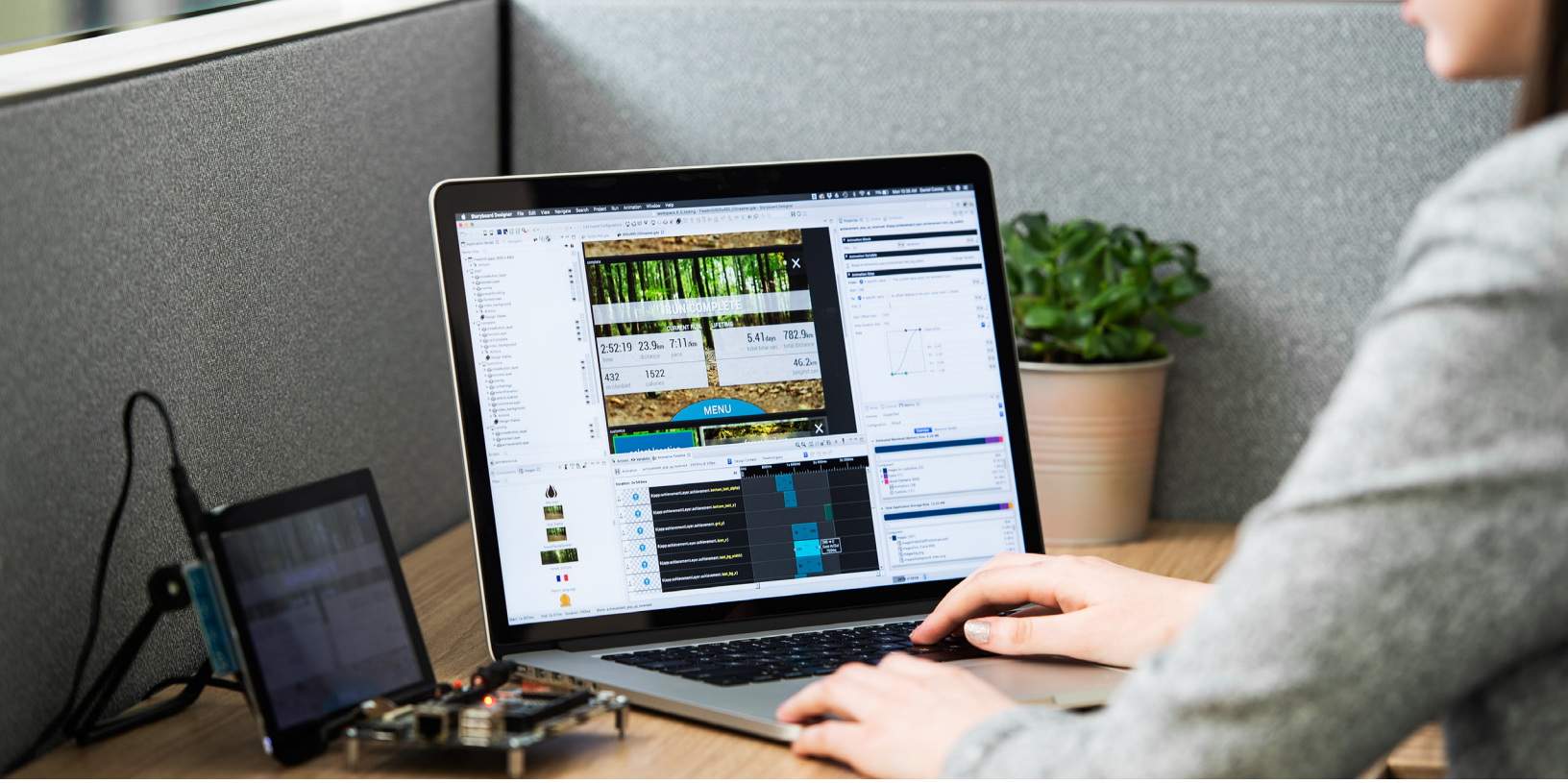
A video explaining where to access Storyboard's Metrics View, how to change the storage type, and preview the image footprint. LEARN MORE

Storyboard Engine plugins

This document provides a complete list of plugins that can be used, such as reporting on process / task memory usage or heap allocator memory usage values in performance log metrics. LEARN MORE

Managing Storyboard engines resource memory

Ways to help fine tune the amount of memory used by sbengine to load assets that the application requires, such as images, fonts, scripts, etc. LEARN MORE



Estimating Storage Requirements

Understanding the storage requirements of your product will depend on several different aspects. Some of these, like graphical asset space and binary sizes, can be accounted for within the Metrics View tool mentioned in the section 'Measure application performance'. Other requirements, such as the size of the OS and other system dependencies, need to be accounted for manually to provide an all-encompassing view of the entire system.

It is important to understand that the hardware platform you are considering can limit the storage available for your GUI application. Even if the platform can support expandable memory, this will come at a cost to the bill of materials or the physical footprint of your device. By understanding the storage requirements of your application, you will be able to ensure you have the right amount of storage space on your device.

Calculating the GUI application storage requirements

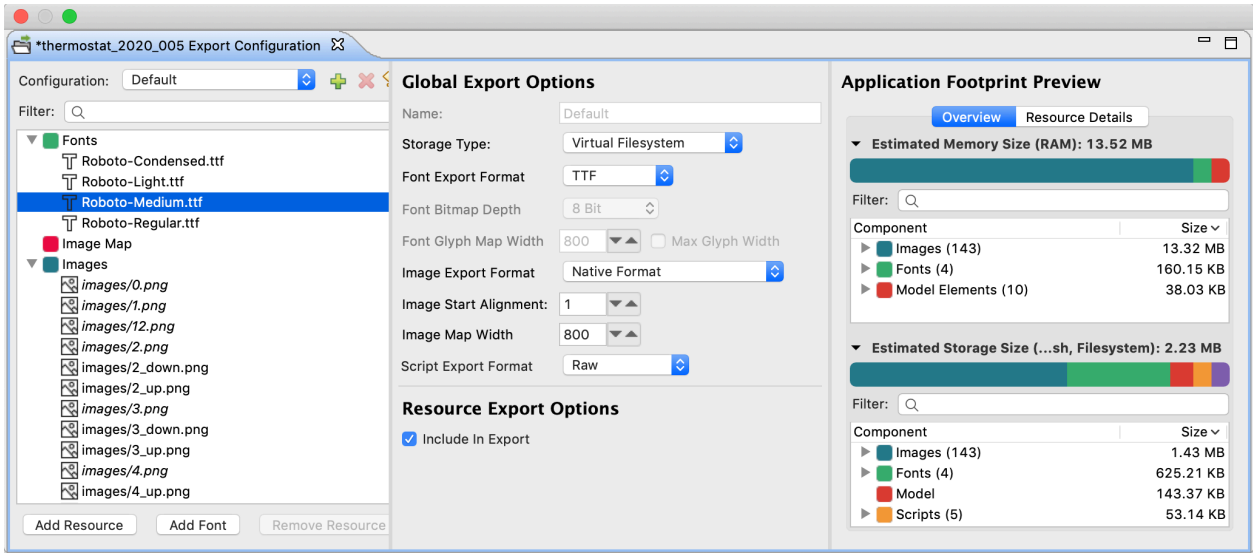
When calculating your filesystem requirements, there are two parts to your Storyboard GUI application:

1. **Storyboard runtime engine**
2. **Storyboard application**

The runtime is the binary portion of the Storyboard application and is fairly easy to calculate since it directly corresponds to the runtime package (hardware and Operating System type) you deploy to your system. On a Linux system, for instance, you'll deploy an archive that contains binary executables, made up of libraries and loadable plugins. The filesystem cost is simply the total size of this archive.

When looking at the Storyboard application, the same generally applies in the manner that you will deploy an application model file with the UI resources to your system. When deploying to an operating system, such as Linux or QNX, you can take the total size of your deployable project directory and use that as a benchmark. In fact, our Metrics View capability in Storyboard Designer will attempt to calculate this information for you (including memory).

While in many cases, the GUI application will make up the largest portion of the total filesystem footprint, due to the size of its graphical assets, you should also consider the dependencies of other aspects of your embedded device, such as its Operating System (OS). The OS and non-graphical applications will all require space. Therefore it is important that you are aware of what your base system image requirements are before the



Storyboard's Resource Export Configuration editor allows for the creation, deletion, and resetting of a configuration. This allows for fine grained control over how and when resources should be exported from the design environment.

Storyboard application and runtimes are added to the system. While these requirements will be highly specific to your configuration, they are essential in understanding what you will require in order to boot your hardware.

For example, if your Linux image requires 1.7GB of space on an SD card, then it would make sense to select a card with more than 2GB of space.

The same logic applies when you are flashing an MCU system. When building and flashing FreeRTOS, most of the tooling solutions will provide metrics on the size of the compiled binary being written to Flash memory. Whenever possible, it is best to give yourself a buffer in storage space since we know that change and iterations are inevitable in any embedded project. Therefore, it is always better to have a little bit more room so you can experiment with different concepts and future feature additions instead of being constrained by a hardware limitation.

How to decrease storage requirements

If the application is running on an OS without a traditional filesystem (for example, like most realtime executives running out of Flash memory on an MCU), there are some options available to help reduce the footprint:

1. Remove non-required duplicated images from the GUI

Ensure that you've consolidated any potential duplicate images and converted any solid color images to "Fill" render extensions using the consolidation tools outlined in Storyboard Documentation. This will ensure maximum efficiency and not waste space with duplicate images or flat colors that could be rendered programmatically.

2. Use of 9-Patch for scalable bitmap images

9-Patch is a technique used to scale an image in such a way that the four corners remain unscaled. You can quickly analyze and convert existing large or scaled image content to 9-Patch format to achieve immediate memory and runtime performance improvements. In order to be converted to 9-Patch, the source image must have full 32-bit color depth.

3. Use only the plugins required for your GUI development

By default, sbengine uses as much memory as it requires to load all the assets that the application requires (images, fonts, scripts, etc.), but this can be tuned to save memory by removing any unused plugins from the plugins directory. The plugins that are available and being loaded will be shown by passing the -i option to the sbengine command-line utility.

4. Create a resource export configuration for your hardware

By default, the resource export configuration will assume a standard file system deployment (SD card with partitions and a filesystem like FAT or EXT4, etc.), from which you can tailor how to store the assets. If there is ample storage space, it is suggested that image assets are stored raw on Flash. Storyboard can render them directly from Flash memory without the need to decode them into RAM first. The big caveat here is that uncompressed images are bigger in size, therefore, requiring more storage space. Their size is directly proportional to their resolution multiplied by the color depth.

Pro Tip: Storyboard Metrics View updates with changes

When you play around with your settings you will notice the Metrics View updating with Storyboards estimate for space required in Flash vs Ram. Since Flash is generally cheaper than RAM, most systems include more of it. This results in many configurations weighted towards maximizing Flash storage in order to save on RAM.

RESOURCES

How to determine storage requirements?

Here is a recommended resource to provide a better understanding of how to measure memory usage in Storyboard.

Storyboard Resource Export Configuration Editor

This document provides details on Storyboard Resource Export Configuration editor which allows for fine grained control over how and when resources should be exported from the design environment.

LEARN MORE

Utilities to efficiently develop your embedded user interface

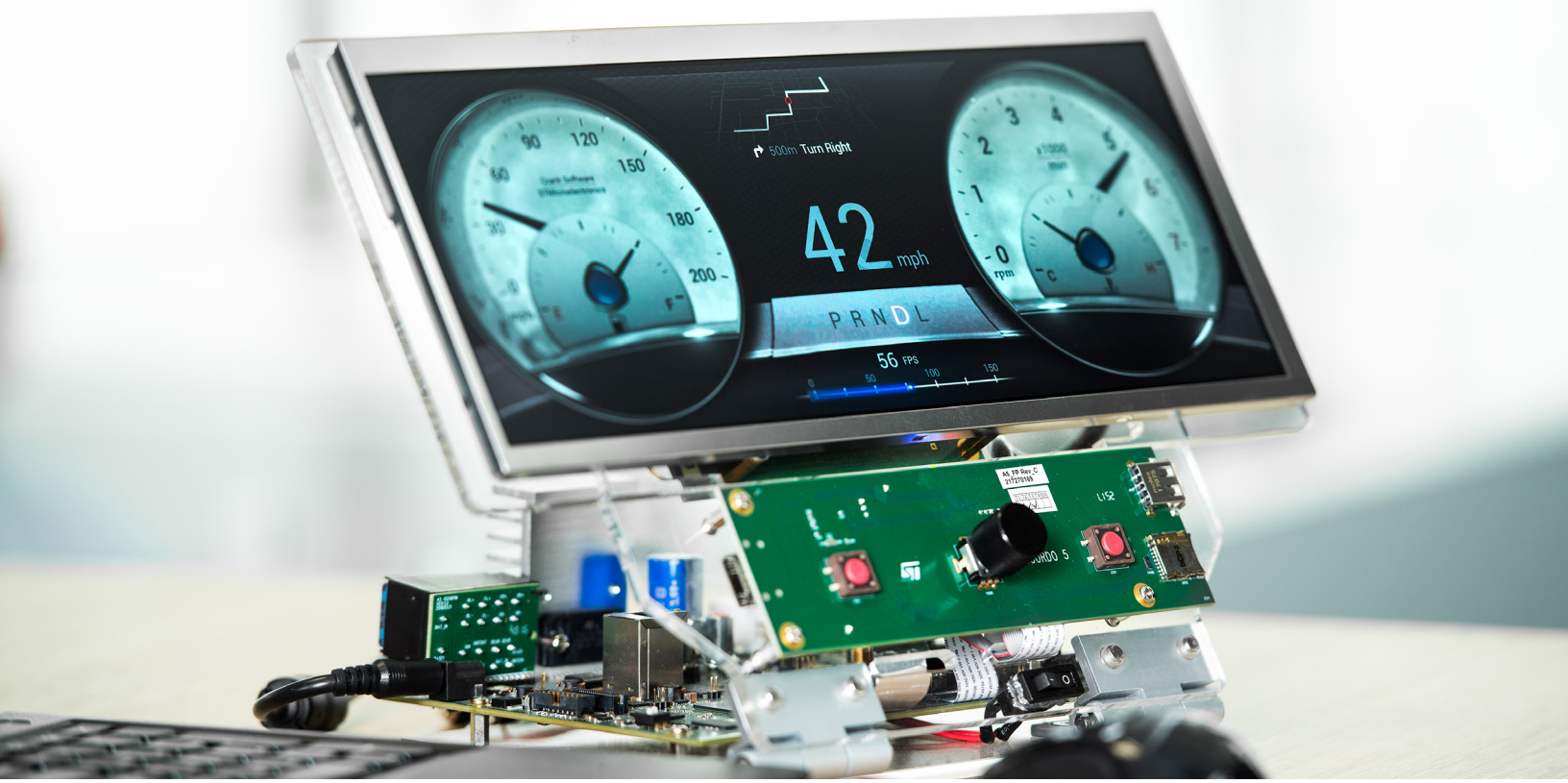
This document discusses features that can improve performance by reducing potential runtime inefficiencies and speed application development through greater insight into an existing application's user interface.

LEARN MORE

Setting up Storyboard Engine

This document reviews configuring the target system for Storyboard Engine (sbengine) and plugins required for the target application.

LEARN MORE



Conclusion

With some predetermined hardware performance testing and careful considerations during the early stages of the selection process, you can make educated decisions and select the most suited hardware for your embedded device. Following these guidelines can help you avoid making product compromises related to hardware limitations, costly hardware respins, or platform reselection that could put the project at risk of meeting deadlines.

Regardless of whether the final hardware choice is an MCU or MPU, you should now have a better understanding of how your decisions could impact your device's GUI application design (or vice versa). Through the use of pre-built Storyboard GUI demo applications, built-in performance metrics, and logging tools, you will be able to quickly gather data on system performance and application resources enabling you to make some educated decisions in hardware and GUI design.

Now go forward with confidence that your hardware platform can support the stunning, graphical user interface developed to enhance the user experience of your embedded device!