GUIDE TO

Development principles for building sophisticated wearable user experiences

Inside

- Comparing memory types
- Framebuffer tips
- Avoiding frame rate overruns
- Understanding fonts, images & animations

> “Worldwide end-user spending on wearable devices will total $81.5 billion in 2021.”
>
> – Gartner, Inc., 2021

Overview

Smartphones changed the way we interact with technology, giving us intelligent experiences in everything from Fitbits to freezers. More recently, a perfect storm of innovations in microchips, software development, and energy efficiency has set the stage for the next revolution in computing’s history: wearables and personal devices.

The way we receive, use, and share data is changing dramatically as a result. We’re using digital means to verify who we are, move us into action, reflect our wellbeing back to us, communicate with each other, record the world around us, and control our environments. This is no fad — wearable technology use has more than tripled in the last four years and global spending on wearable devices will total $81.5 billion in 2021.

While this wearable revolution amounts to many new business opportunities, it also constitutes a paradigm shift in how users interact with technology. Battery life becomes even more important, as does building a vivid, intuitive user experience (UX) that consumers and professionals crave.

Understanding how memory and graphics work to deliver those experiences is key to winning the wearables market and building user loyalty.

This e-book explains seven embedded graphical user interface (GUI) concepts that are key to getting the most out of your target system’s memory and graphics capabilities. From framebuffers to GIFs, the topics presented here help guide your software design decisions and offer useful reference sheets for when things get busy.

We’ve also included two sections on wearables software best practices and an in-depth analysis of how frames per second really matters to your application.

Get ready to fire up your wearables UX!

Best software practices for improving wearables UX

How do you approach designing and building a new wearables GUI? Rather than keep UX design separate from development, it's important to understand where they intersect to build applications that users love and want to interact with.

Let's look at the four ways in which effective wearable GUI development and design must adapt to address the new challenges of this booming market segment.

Step 1: Optimize for easier navigation on small screens

To accommodate increasingly smaller devices, screens are decreasing in size. This affects the user's ability to access information, including locating it, understanding it, and remembering it.

1 Simplify navigation options

The first rule of thumb is to simplify and declutter. Users want to get quickly in and out of the functions they need. For this, they need easy navigation. It's a good idea to get rid of anything in a wearable GUI that isn't absolutely necessary. To do this: Use simple buttons and swipes, and avoid any text input. The user should be able to gain access to the information they want in ideally two clicks and/or swipes.

2 Use scrolling or wrapped menus

These help declutter the GUI since users only see one to two menu items on-screen at a time.

3 Focus on glanceability

Users need to be able to glance at their wearables while in motion and get the information they need.

4 Avoid high contrast and bright colors

The best way to ensure users don't have to stare at their screens for long lengths of time is to use minimal text in high contrast and bright colors.

5 Avoid serif and script fonts

Keep text highly legible by avoiding the use of serif and script fonts when pixels are limited.

6 Icons instead of text

Simple icons instead of text can have a positive impact too — it's much faster to recognize an icon versus reading a text string.

Step 2: Design UX based on hardware limitations

The chip or board running a wearable device is going to be more limited in performance than a desktop or even a typical embedded processor. Plus, to be useful on a person's body, it must stay cool and use little power. And for that, choosing the right hardware is critical.

Not all GUI development teams have input into hardware choice, but they should have a say in its functional requirements, which may include everything from 3D gestures through haptic feedback to Wi-Fi connectivity.

It's best to carefully consider processor-intensive tasks as needed. While animations can provide an effective and rich UX, they may consume more cycles and power than you realize. It's good practice to use CPU and memory performance measurement tools to determine exactly how resources are being consumed and tweak animations accordingly (such as by reducing the number of frames or the complexity of the images). Something else to keep in mind: these high-intensive tasks can also make a device run extra hot. No one wants users burned – literally.

Also consider using hardware acceleration, where available, to offload the CPU from performing these high-running tasks.

Step 3: Use memory judiciously

Memory and flash storage in wearable devices is a tricky balancing act between capacity and price. The bigger the chip, the more expensive it is. Both resources must be used judiciously, and GUI development teams (as one of the biggest consumers of both RAM and flash) should be highly considerate of their usage.

1 **Consider replacing high-definition images with smaller resolution images**

You can also adopt higher compression formats or, better yet, use vector graphics that designers can scale easily during the UX design process and developers can rasterize to consume much less memory and processor cycles on the target. GUIs can still be attractive with solid shapes and fills or gradients instead of custom-crafted PNGs.

Also consider dropping down screen and GUI asset color fidelity, if further memory is required.

2 **Ensure the application frees up as much memory as possible**

Free data lists, terminate applications, and flush caches as soon as they're not needed, but first profile your memory use and run user UX tests to determine which of these techniques won't impact the device's perceived "snappiness".

Step 4: Make every minute of battery life count

While the modest form factor of a wearable device makes it easier to keep on the body, a small device contains a small battery, which makes energy a precious resource. Embedded GUI development teams must pay careful attention to make every minute of battery life count.

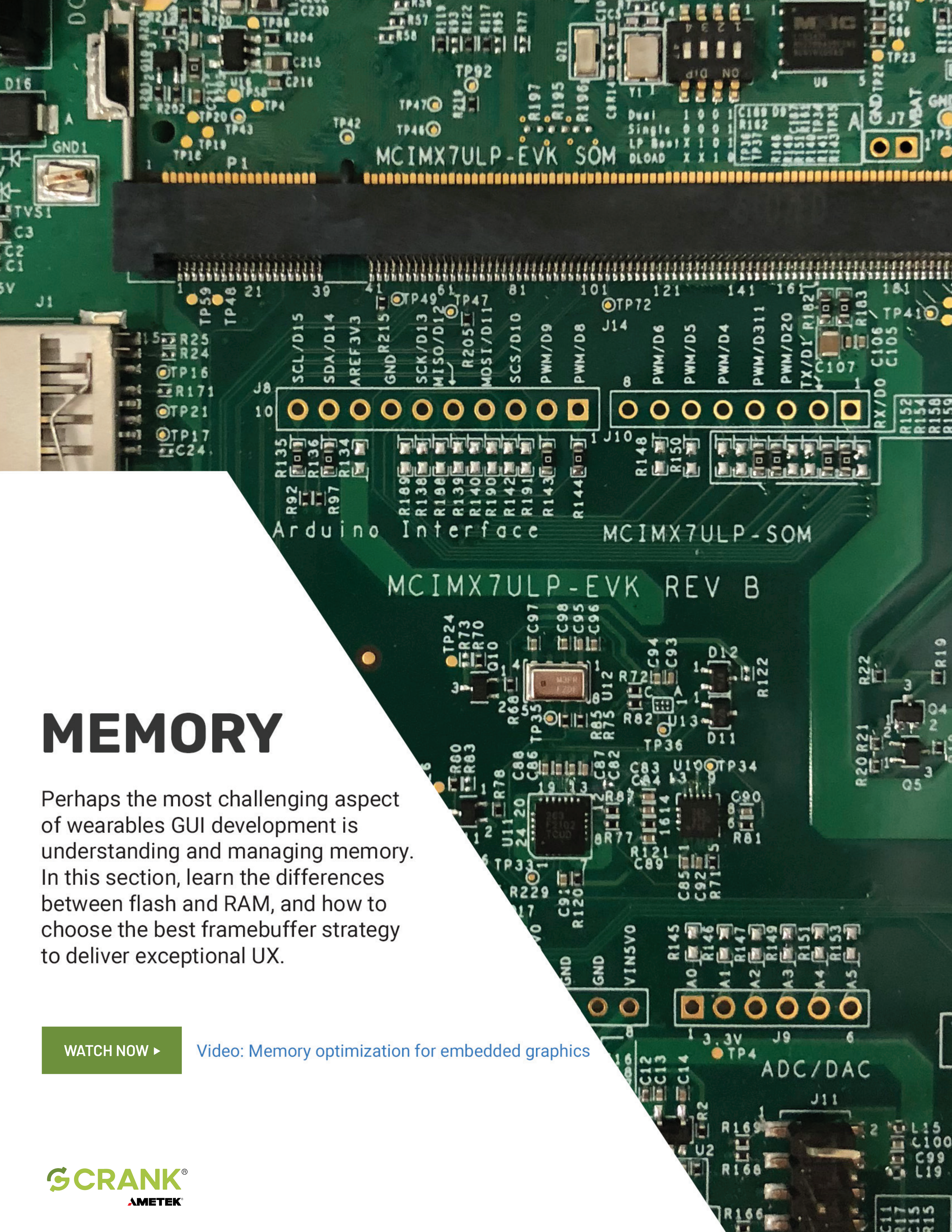
1 **Use sleep mode as much as possible**

The most obvious strategy for saving power is to use sleep mode as much as possible, only waking a device when there's an event, such as a button push or physical movement. It's important to have a GUI framework that can create code that's close to the metal, for the sake of battery life as well as a device's responsiveness and memory consumption. The more efficient the code, the less power it will draw.

2 **Dim the backlight when not active or in low light**

Another thing to keep in mind is how a GUI interacts with screen technology. With an LCD display, the most controllable power draw is the screen's backlight. Dimming the backlight when not active or in low light can save a lot of juice. With an OLED or AMOLED screen, there is no backlight. Instead, each pixel is its own LED (light-emitting diode). That means white and bright colors draw much more power than black and dark colors.

A dimming layer or muted design palette can help achieve the same effect, to reduce the brightness of the GUI when the screen isn't being actively used. Both backlight and dimming strategies have the added bonus of avoiding display burn-in, which happens when high-contrast images in full-power mode end up leaving a ghost shadow.



MEMORY

Perhaps the most challenging aspect of wearables GUI development is understanding and managing memory. In this section, learn the differences between flash and RAM, and how to choose the best framebuffer strategy to deliver exceptional UX.

WATCH NOW ▸ Video: Memory optimization for embedded graphics

CRANK® AMETEK®

Flash vs. RAM

Flash memory is used primarily for storage while RAM is typically used for dynamic operations on data retrieved from storage. Flash is considered non-volatile memory, in that data is stored even when the device's power is off. RAM, on the other hand, is volatile, maintaining data only when power is applied to the memory cells. Understanding and accounting for the different memory requirements of your wearable application is critical. Your choice between using system-level memory (heap, stack, and static code) and hardware-level memory impacts graphics performance, directly affecting UX and the market success of your GUI.

Comparing memory types

- **Flash** memory typically has speeds fast enough to manage the storage of image assets, font files, and videos but not fast enough to perform dynamic operations like graphics rendering and manipulation of application data. Flash memory is made of solid-state chips that run slower than RAM but do not require battery power to maintain data persistence.
- **RAM** has very fast access times and is used for dynamic content, such as animations, framebuffer, font rendering, and read/write operations from the CPU. RAM uses storage cells that require a fresh electronic charge every few milliseconds to maintain data, directly impacting the battery life of your wearable device.
- **Cache** memory is a specific application of RAM that's located very close to the CPU. This is very fast, and typically very expensive memory that's used to store instructions and data that the CPU is likely to use repeatedly in the immediate future. Cache operation is usually outside your control but it's a good idea to understand its behavior on your system to optimize its use.

Memory type	Nature	Read access time	Write access time	Power consumption	Use cases
Flash	Persistent	Fast	Slow	Low	Application code, persistent data, asset storage
RAM	Dynamic	Very fast	Very fast	High	Application stack, heap, variables, framebuffers
Cache	Dynamic	Very fast	Very fast	High	Usually controlled by the OS to speed up memory access

Framebuffer

In embedded devices, framebuffers are a contiguous block of memory that contain the color 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 color depth (16 or 32-bit) determine how much memory you require to allocate for your buffer(s).

Framebuffer memory is usually allocated to the heap in RAM.

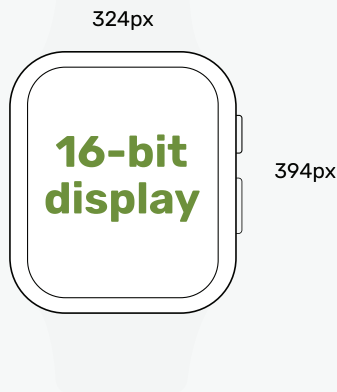
16-bit color depth = 2 bytes per pixel

32-bit color depth = 4 bytes per pixel

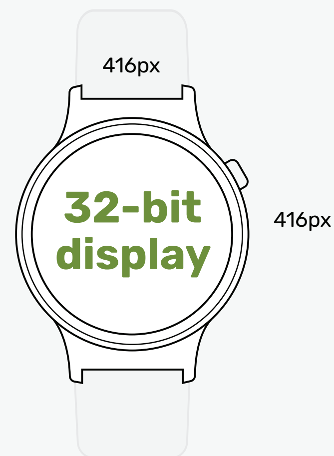
Calculating memory requirements

Framebuffer memory required = display resolution x bit color

(bit color is the number of bytes per pixel)



(324 x 394) x 2 = 255,312 bytes = **255 kb**



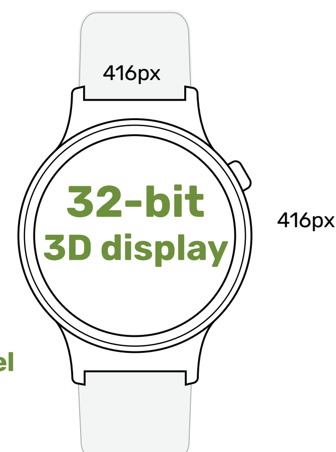
(416 x 416) x 4 = 692,224 bytes = **692 kb**

3D graphics

Wearables that require 3D graphics commonly use two additional buffers: Depth (or Z-buffer, used to specify depth of objects) and stencil (used to limit render areas).

In these cases, you need to include additional bytes per pixel for each buffer:

32-bit color buffer + 32-bit depth/stencil buffer = 8 bytes per pixel



(416 x 416) x 8 = 1,384,448 bytes = **1,384 kb**

Selecting framebuffer memory

Many MCUs offer different types of RAM, each with their own characteristics in terms of size and speed. Your framebuffer strategy should take these characteristics into account, to improve UX.

If only one framebuffer was used and allocated to a single block of RAM, users would see elements being composited in real-time as the screen pulls from the same location that the application is using — an undesirable user experience. To avoid this, wearable applications typically use multiple framebuffers to render to the screen, flipping between those that are used for compositing (not visible) and the one that's being directly used by the screen.

There are many strategies for flipping between framebuffers (such as back buffering and triple buffering) but in terms of memory usage, two important considerations must be factored into your memory model:

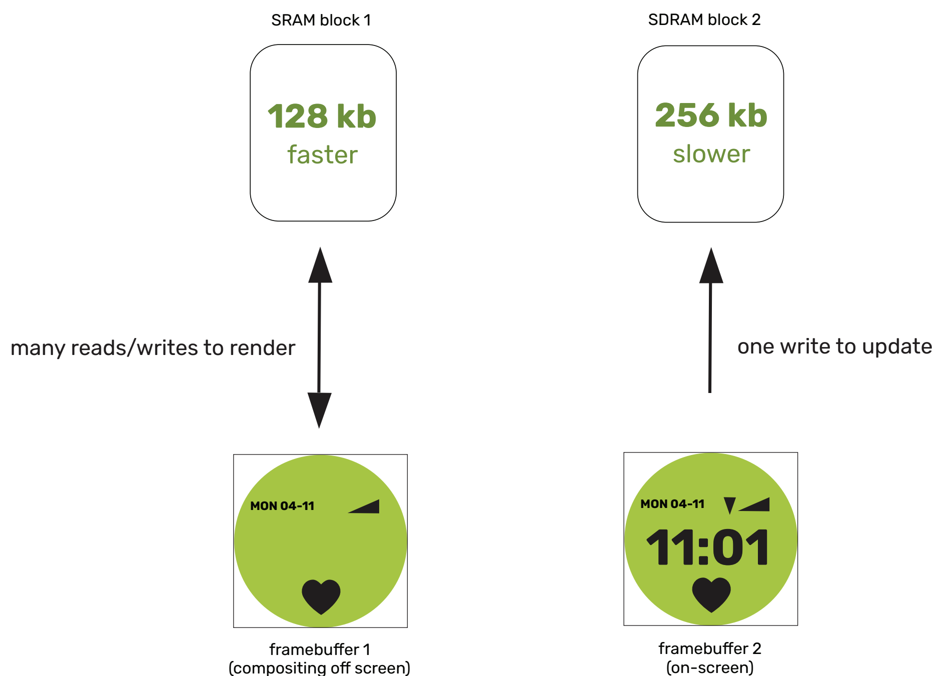
1 **More buffers require more memory space**

2 **Compositing requires many read and write cycles, impacting performance and battery life**

Consider this simplified example of two memory blocks (faster SRAM and slower SDRAM), and two framebuffers, one for compositing and one for the screen to pull from.

As compositing requires many reads/write cycles from memory to assemble the screen, and pushing to the screen requires only one write cycle, the best allocation for performance (assuming no other constraints) is:

- Framebuffer 1 maps to SRAM block 1
- Framebuffer 2 maps to SDRAM block 2



There are other considerations in real systems, such as whether the framebuffer fits into the available RAM block, choosing to allocate the stack to the fastest RAM, etc., but the key to remember is:

Fast RAM is better for frequent and repetitive data read/writes, such as the stack, framebuffer, and common images and fonts.

GRAPHICS

Many aspects of a wearable GUI application contribute to the UX, but none more so than the design choices and development decisions surrounding graphics. In this section, learn how frame rate affects UX and how to choose and optimize three types of visual elements: images, fonts, and animations

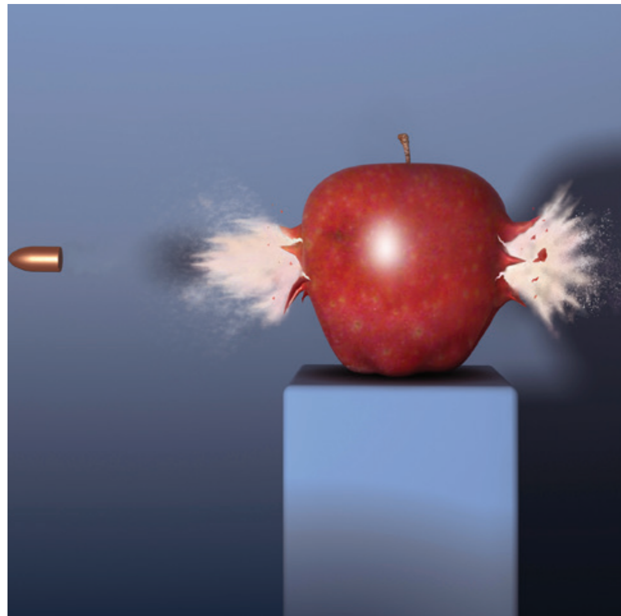
WATCH NOW ▸ Video: Best practices for fonts on MCU systems

How important is 60 frames per second?

When it comes to embedded GUI development, the 60 frames per second (FPS) watermark is synonymous with `smooth´, `beautiful´, and `fluid´. Developers and tech-savvy users talk about 60 FPS as the gold standard for screen updates while vendors promise that only their platforms will be the ones to help you achieve it.

When was the last time an end-user actually noticed the frame rate of their device? More likely, they complained about slow screen transitions or jittery animations and they may have attributed it to FPS without knowing the actual causes. Most users don't care about frames per second specifically; they care about user experience (UX) and how often they need to plug in their devices.

These factors make 60 FPS an incomplete goal. It's not just about achieving the number, it's about managing CPU workloads without the user noticing it.



Can the human eye see 60 FPS?

There's debate over whether we can actually tell the difference between higher frame rates. Some say that the human eye can perceive only 30 frames per second while others state that human physiology is capable of detecting up to 1000 FPS. The behavior that most agree on is best explained by these two quotes from PC Gamer:

> *"The first thing to understand is that we perceive different aspects of vision differently. Detecting motion is not the same as detecting light. Another thing is that different parts of the eye perform differently. The centre of your vision is good at different stuff than the periphery. And another thing is that there are natural, physical limits to what we can perceive. It takes time for the light that passes through your cornea to become information on which your brain can act, and our brains can only process that information at a certain speed."*
>
> *"The whole of what we perceive is greater than what any one element of our visual system can achieve. This point is fundamental to understanding our perception of vision."*
>
> *- PC Gamer, 2017*

Consider the reason why we love 60 FPS — it's essentially an historical artifact from many years ago. Older NTSC televisions ran at 60 Hz because it was easier to synchronize that type of video signal to the 60 Hz AC power the devices ran on. People got used to the look and feel of this frame rate and now, users expect it on every screen they interact with.

Understanding the psychology of why we want 60 FPS for our embedded devices is part of the story. The other is to consider what vendors and manufacturers do to achieve it.

Is 60 a good FPS?

More often than not, high frame rate claims are taken from applications designed to achieve an arbitrary watermark, not reflect the reality of real-world systems.

Here are some examples of the design decisions made in the name of achieving 60 FPS:

1. Raw file formats used for images and animations versus the compressed formats that many developers use to save on storage space
2. Minimal inter-process communication or events handling between the backend software and the GUI
3. No complex calculations or algorithms that are usually required for applications such as maps and voice assistants
4. Use of hardware and peripherals with fast access times that aren't necessarily available to all projects and budgets (although this is becoming less of an issue in the MCU space as silicon vendors are releasing more powerful products, like this NXP webinar explains)
5. Measurements look at the screen's refresh rate, not the number of times the application rendered to the screen.

Does refresh rate affect FPS?

Refresh rate can dramatically impact frames per second. First, a screen running at a 60 Hz refresh rate has little to do with the application workloads put upon the CPU and GPU. The screen runs as it runs. Second, if the application is refreshing the entire framebuffer at the same rate as the screen, serious consideration must be made as to why this is so. Consuming precious processor and memory cycles should not be taken lightly, and many applications do not actually require a full-screen refresh at 60 Hz.

Is it better to have a smooth, efficient, and slick looking GUI, or is it better to make sacrifices to hit 60 FPS?

How to optimize FPS

The "right" FPS for embedded applications is the one that creates the best user experience and minimizes power consumption within every frame. The best embedded UX is achieved by rendering the designer's intent for screen elements and behavior within the boundaries of a frame. In the 60 FPS case, this means completing the necessary processing and rendering within 16 milliseconds (ms). The definition of "necessary" depends on the application but generally, this does not require the entire screen to be re-rendered every frame. As workloads consume processor cycles and power, only changing elements should be rendered.

Always ask yourself:

What is required of the system and when must it be done?

From there, you can tune up or down to achieve the desired FPS.

Here are two steps to consider when determining your FPS optimization strategy:

1

Only render when required

Here are two examples of how to minimize workloads for common GUI elements:

- **Button up/down** — if the design requires a two-state button effect, the application can render the down state in one frame and the up state in another. This requires only two frames in total.
- **Animations** — if a screen update only requires a 500 ms animation, an application running at 60 FPS doesn't need to consume cycles for the whole second. It can render that element in 30 frames and either sleep or do other work for 30 frames (500 ms = half a second = 30 frames).

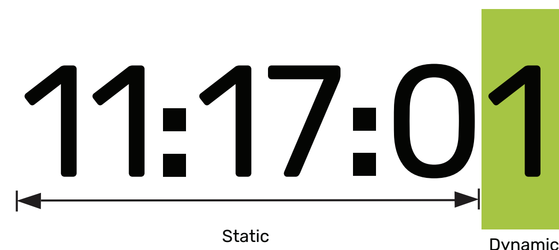
2

Render only what's required

Developers have a choice as to what to update in the framebuffer: The entire screen or just the changed elements. It may seem easier to render the whole screen every frame but this requires CPU cycles and power to perform.

The clock to the right changes every second but only the rightmost digit changes for every nine second period. Rather than update both the static and dynamic elements every second, it's far more efficient to update the framebuffer location that contains the rightmost digit only, saving time and CPU power.

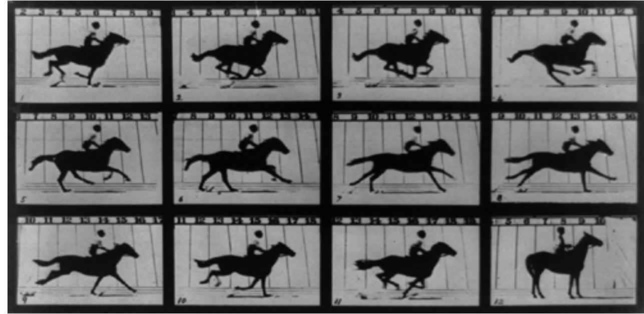
These examples demonstrate a "minimum rendering workload" philosophy, allowing the system to perform other tasks during the frame and second, and enriching the UX.



Frame rate

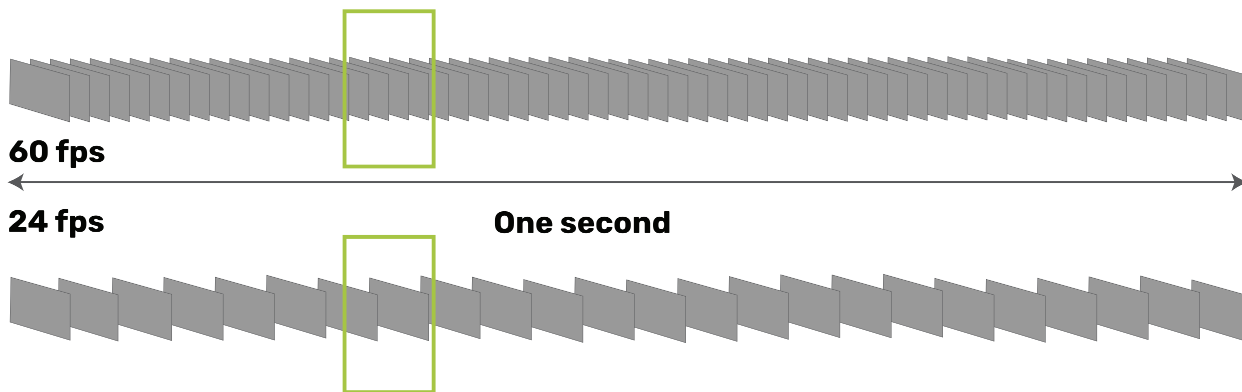
Frame rate, measured in frames per second (FPS), is the speed at which the wearable device's screen is updated by the system. Frame rate has a significant impact over user experience as the smoothness of image rendering, video, and animations is directly dependent on how well the system maintains visual updates per frame.

60 FPS is typically the goal for wearables but there are common misconceptions around this number discussed below.



How it works

Updating the screen requires processor resources and time:



If the amount of processor time needed to update the screen exceeds the frame time (overrun), the graphics might jitter, freeze, or not update at all. A common misconception is that a higher frame rate makes animations and transitions run smoother, however, if the system can't keep pace, it's likely to drop frames.

How to avoid frame overruns

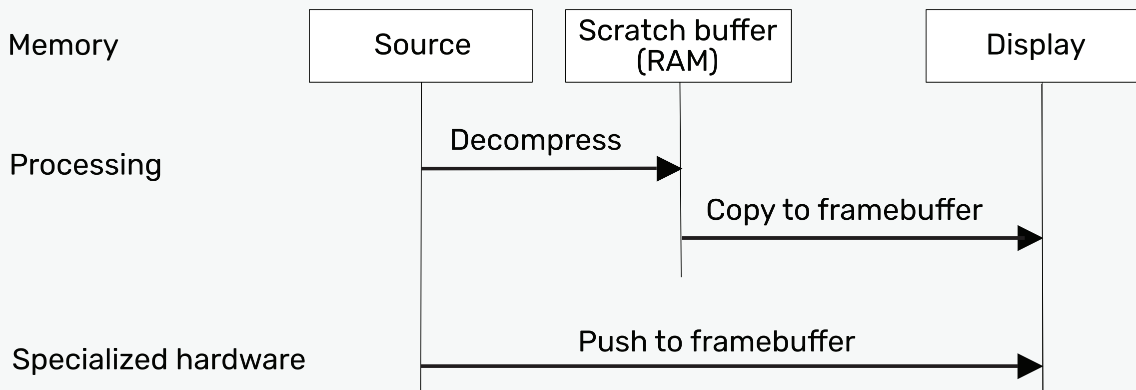
1. Update the screen only when necessary. For example, if the user action is to press and release a button, this usually requires a two-frame update only (down state, up state) with no need to redraw up to 60 frames.
2. Avoid full screen updates when you can and update only those elements that change. For example, a digital clock only requires the seconds digits to update frequently while the rest of the display can remain unchanged.
3. Reduce the amount of alpha blending, which can be very processor intensive.

Image formats

Different image file formats have different implications on storage space and computational overhead at runtime. Raw formats (like TGA) contain all the image data without any type of compression algorithm applied to it, while compressed formats (like PNG, JPEG, BMP) utilize encoding to reduce the file size.

How it works

As raw files contain all the uncompressed image data, the screen can pull these directly from storage. Compressed images require decompression into a format usable by the screen, incurring a processor cost and the need for a temporary, or scratch buffer to store the intermediate artifacts:



In some cases, systems use hardware acceleration or other specialized features to reduce or eliminate the need for your own decompression path. To avoid repeated overhead costs, systems typically cache decompressed images (scratch buffer) for the screen to pull from as needed.

There also tends to be a difference between MCU and MPU deployments: As there is no filesystem to manage access, MCU applications commonly use raw image formats versus MPU-based applications that can take advantage of the optimized access times and capabilities of a filesystem.

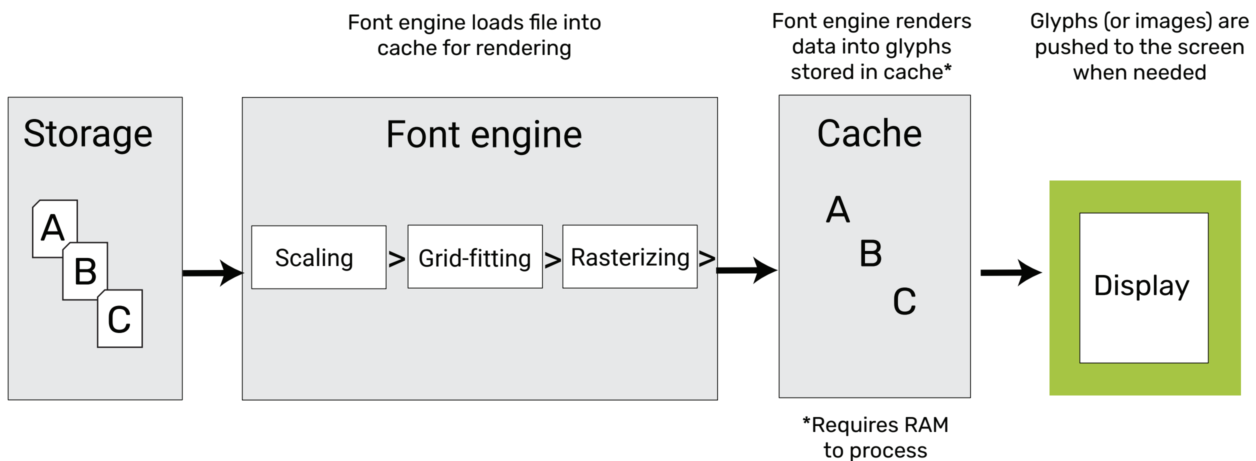
Balancing the costs of RAM vs. flash depends on the project but in general:

Image format	Memory source	Scratch buffer (cache)
Raw	Flash, as these files can be copied directly to the screen in one operation	N/A
Compressed	RAM, as these files require multiple operations to decompress from storage (writes to the scratch buffer) and be retrieved by the screen (reads from the screen buffer)	

Rendering fonts

Strings are rendered to the screen as a series of glyphs (or small images) extracted from the characters in a font file. There are two types of font files, true type (TTF) and open type (OTF). A font engine is typically used to render the font file data into glyphs that are stored in cache and pushed to the screen. These glyphs remain in cache and used when needed, avoiding the need for re-rendering.

How it works



Avoiding the use of RAM

Caching font data requires RAM and computational costs. To avoid consuming resources on low-RAM systems:

1. Before deployment, pre-render your font glyphs into Bitmap images
2. Store the Bitmap images in Flash or other type of non-RAM storage
3. Push the Bitmap images directly from storage, bypassing the need to use RAM

Pre-rendered Bitmap images require storage space, with one glyph needed for every font style and every size:

Roboto font at 18-pt, 48-pt, and 70-pt = 5.6 MB flash storage (approx.)

Preserving storage space when pre-rendering fonts

1. Store less bits for the alpha map
2. Don't pre-render unnecessary characters (e.g., a numeric keypad doesn't need letters)

Animations: Flipbook vs. animated GIF vs. video

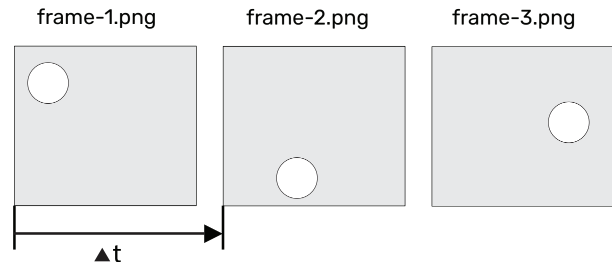
When it comes to running animations on wearable screens, there's a choice between loading separate image files in sequence or loading a single animated GIF. Each approach has implications and tradeoffs:

Flipbook style

A flipbook style animation requires pulling all the files from storage, decoding them if compressed, and pushing each file in sequence to the screen at the desired frame rate (▲t).

Benefit: This approach uses less memory to cache the files (RAM) over the application life cycle.

Disadvantage: The downside is that multiple files are required to be stored, potentially taking up more space.



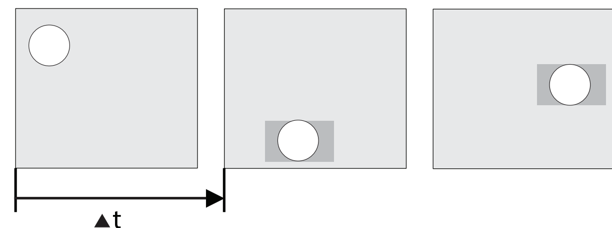
A flipbook style animation pushes each file in sequence

Animated GIF

An animated GIF stores a stream of images in a single file, with time delays specified between them (▲t). Once started, the animated GIF plays each frame once, unless the file contains looping data to repeat the animation a specified number of times.

Benefit: All image information is encapsulated in one file, generally consuming less space than if the files were split out individually.

Disadvantage: The downside is that this requires more memory to cache (to process all the frames at once) and the animation cannot be started from any frame other than the first.



An animated GIF loops data to repeat the animation

Video

An alternative to both these formats is to use video, however this has its own implications:

- Video requires the inclusion and management of a player within the application
- Tends to be more difficult to share between processes without consuming more memory
- Difficult to render the video to multiple locations on the screen at the same time without consuming too much memory

LEARN MORE

WATCH NOW ▸ Video: Memory optimization for embedded GUIs

READ NOW ▸ Blog: Optimizing image graphic memory in embedded GUIs

READ NOW ▸ Blog: Maintaining a responsive GUI with task prioritization

WATCH NOW ▸ Creating exceptional wearable UX with Storyboard and NXP's i.MX RT500 MCU

CRANK®
AMETEK®

Try smartwatch demo GUI images for yourself

Download and run these self-contained, sample executable images — built in Crank Storyboard — on your hardware, and see what your GUI possibilities are.

NXP

Processor: NXP i.MX RT 500

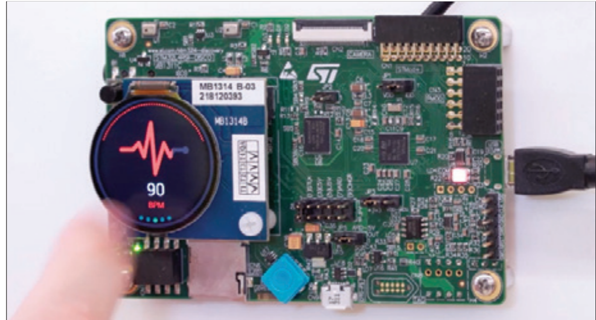
Operating system: FreeRTOS

Resolution: 390 x 390

Rendering: VGLite

DOWNLOAD ▸

STMicroelectronics



Processor: STMicroelectronics STM32L4R9

Operating system: FreeRTOS

Resolution: 390 x 390

Rendering: DMA2D

DOWNLOAD ▸

1000 Innovation Drive, Suite 100, Ottawa, ON, Canada K2K 3E7
1 (613) 595-1999 | info@cranksoftware.com | cranksoftware.com

© 2021 Crank Software Inc. All rights reserved. No part of this publication may be reproduced without the prior written permission of Crank Software Inc. While every precaution has been taken in the preparation of this document, Crank Software Inc. assumes no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein. Specifications subject to change without notice.