Containers

Cloud tech comes to embedded

Till Adam | Chief Commercial Officer
Sven Knebel | Software Engineer

KDAB

Containerization is starting to appear in embedded, but it is definitely not yet commonplace.

Containers in embedded

Software developed for embedded applications is often distinct from its desktop and cloud cousins due to the constraints of embedded hardware and the integration of non-mainstream devices. But problem-solving technologies developed in other places tend to migrate to embedded systems once the hardware catches up. Containers are one of these – and they're not just for cloud developers anymore.

While containerization is starting to appear in embedded, it is definitely not yet commonplace. At KDAB, we see the increasing interest in embedded containers come with a host of questions. What exactly are they, are they the right fit for my product, what advantages do they bring, what are the disadvantages, and are those disadvantages something we can live with?

This white paper provides a general overview of containers in embedded from our own knowledge, research, and experience, as well as that from our partner Toradex, a hardware vendor that is also an expert in embedded containers. Hopefully it will help you answer if now is the right time to add this technology to your embedded tool chest.

Containers don't require anywhere near as much disk space as VM images; this is handy on the developer workstation but critical on the embedded target.

What is a container?

Containers are sometimes called "packaging abstractions" since they can wrap up a program along with all its dependences (binary executables, libraries, configuration files, etc) into a single, isolated executable environment. Although this description is true, it doesn't communicate the true value of the technology.

People also often compare containers to lightweight virtual machines (VMs). This underscores one of the main use cases of containers: isolating multiple applications running on the same machine. While that description feels close to how containers are used in practice, it fails to deliver the right architectural sense. VMs are associated with hypervisors, full OS stacks, hard disk images, virtualized devices, and hardware-dependent isolation, all things a container doesn't have.

Containers have a lot of similarities to VMs without the comparatively heavy drain on resources, ideal for embedded.

A more appropriate analogue would be comparing a container to a super-powered `chroot` since a container bundles an application with all of the filesystem pieces it needs to operate. In other words, all the libraries, utilities, data, and configuration files that the app depends on during runtime. This part is like `chroot` – or a BSD jail, which is in fact the original genesis of the concept. In addition to the filesystem part, the container also gives the bundle dedicated namespace, memory, and networking views, insulating it from the rest of the system. This is the container's "superpower", and it's reliant on Linux namespace technology. (Although containers can operate on Windows too, Linux containers are much more common especially in embedded applications, and throughout this paper we'll be describing containers in Linux terms.)

What is `chroot`?

A shortened form of "change root", `chroot` changes the top-level root directory for a specified process to be somewhere else in the directory tree. The ancestor of today's containers, `chroot` originally comes from Unix and allows the system to restrict the process to a limited subset of the computer's file system. This prevents the process from being able to tamper with unrelated files even if it has (or is given) elevated privileges. This technique was commonly employed with applications such as web browsers, so that hackers would do limited damage if they exploited a vulnerability.

Benefits of containers

Containers have a lot of similarities to VMs without a comparatively heavy drain on resources, which is ideal from an embedded point-of-view. What are some practical benefits of this technology in day-to-day use?

Containers can be used to great effect in development environments, even if they are never deployed on targets.

There are many ways to use containers in development, testing, deployment, and cybersecurity.

In development

Although containers can be used on the embedded target (as we discuss later), embedded developers can also use them to great effect even if the containers are only used within the development environment and are never deployed on the target hardware. Here are several ways they're useful.

Guaranteed toolkits. Developers can place their development tools into a container, which allows them to use tools on multiple platforms without an installation. This helps ensure all team members are using the exact same tools and build environment. It also makes it trivial to spin up a new development environment for new developers or new machines. And it can guarantee that the build server is always using the same tools as the development team.

Different tool chains. Separate containers can hold different variants of a tool chain. This lets developers test an application against multiple tool chains without worrying about how those tool chains can co-exist, or whether they're installed correctly. It allows developers to tinker with experimental software in a controlled way without polluting their development environment. And developers can make software patches against an older version of tools when the mainline development has already moved to a later release.

Multistage containers. With multistage containers, developers can create a container structure that layers the development environment on top of the production environment. That way, all of the tools that developers rely on to create software are stripped out safely without impacting the production build.

Complete snapshot. Because containers are easily versioned, it's easier to snapshot the entire runtime environment for branches, forks, and releases. This tight control also helps when certifying software, allowing external auditors to identify exactly what software and libraries are in use.

Namespaces are the Linux technology that container software uses to implement process isolation.

Distributed deployment. It's easier to build complex distributed systems with containers since they support modular software building and deployment. This is becoming especially important in IoT, industrial, and automation applications, where you need to orchestrate many different components over a long period of time at scale.

The power behind containers: namespaces

Namespaces are the Linux technology that container software uses to implement process isolation. They provide the ability to control which resources an application can see and how they appear. These resources are divided into different domains; here are some of the most common.

mnt – mount points of the namespace; in other words, the process' view of the file system. Mounts give the container the ability to use its own files (from the host system) or share files between containers

pid – process IDs (PIDs), independent for each container. The pid namespace follows the same behavior as Linux in general, such as the first process within the namespace is assigned a PID of 1 and terminating this process will end all other processes within the namespace

net – a virtualized network stack that has its own devices, IP addresses, routing, firewall, sockets, etc. The network resources within a namespace may be redirected back to network resources within the parent namespace

ipc – inter process communication, such as messages, semaphores, or shared memory, letting processes in different namespaces use the same names for objects without having them collide at a system level

user – the creation of independent non-conflicting user ids for each name space

With targets

Using containers on an embedded target provides additional benefits beyond the development sphere.

Containers make it much easier to develop software that's hardware independent, minimizing the pain of switching between hardware

Containers broaden the embedded development talent pool by allowing developers to use a system right away.

platforms or hardware vendors. This hardware flexibility supports a development workflow and configuration management system that's needed to deliver multiple product variants or scalable product families from a single code base.

There are other benefits too. Placing the target environment into a container helps with provisioning. With a container, the setup of a fresh target environment becomes much faster than provisioning a raw target and much more reliable than cleaning off a target that has already been used. And with proper configuration, container behavior differences between the development and embedded systems can be minimized. This allows much of the development to proceed even in the absence of target hardware. The software team then becomes less dependent on sufficient hardware availability or affected by delays in hardware-production timing.

Containers make it easier to ramp up in embedded development, making it possible to expand your team with web and cloud developers.

Containers also help broaden the talent pool necessary for embedded development. Rather than requiring a lot of complex configuration procedures to bring up an embedded target, a pre-containerized target makes it possible for nearly any developer to start using an embedded system right away. While arguably that might also be true for any hardware vendor who's provided a sufficiently complete and well-documented BSP, containers can provide a consistent interface between vendors that's also well-understood outside the embedded space. That means that cloud and web developers who are moving into the embedded space and already have experience with containers can easily bring up a target that's out-of-the-box. Containers can open up embedded and IoT programming to a whole new talent pool and partially side-step the need for specialized hardware bring-up expertise.

In testing

Container start up is fast enough to allow a new container to be spun up for each test, making test reproducibility much more reliable and consistent. They are also lightweight enough to allow several containers running at the same time for parallel test sessions.

The hardware vendor can provide quick patches and support new hardware while the application remains stable.

With containers shared between a development machine and embedded target, some tests can be run on the faster development host instead of the slower embedded target. And for applications that need to talk between peers, clients, and servers, a container supports the creation of multiple nodes within a virtual network on the same machine, drastically simplifying the creation of a test environment.

Containers aren't a security panacea, but they do help to additionally insulate your product from attackers.

In deployment

Containers allow application development to be decoupled from the hardware vendor's Linux stack. That is, the hardware vendor uses a Yocto base system while the application developer implements a container-based environment on top. This lets a hardware vendor provide quick patches and support new hardware while the application remains stable and unperturbed, even when the release cycles of the two sides are mismatched (as they often are). Eliminating issues caused by swapping out the application's underlying OS and drivers helps ensure that products will get the latest security updates, bug fixes, and performance improvements.

Updating containers with an OTA (over-the-air) solution takes advantage of an already modular architecture and can provide software updates with very little disruption to the underlying application.

For security

Containers provide an additional layer of protection for applications and services, making it even harder for hackers to misuse the underlying system or other applications. For one, containers can be signed so that unsigned containers (which are untrusted) can be prevented from running. They enable modular updates to system components, making it easier to keep security-critical software updated without interfering with the application. They also decrease the attack surface exposed by an application, which helps reduce the risk of vulnerabilities being exploited.

Architecting with containers

All these benefits are great, but how do you actually build an embedded application using a container? There are at least three common

Breaking the application into independent containers can help insulate it from internal dependencies.

approaches, depending on the requirements of your embedded device and whether you're starting from scratch or have an existing code base.

Headless. Headless and IoT edge devices don't need a display and are very easy to containerize. They're already similar in many ways to cloud or web applications and in fact may use web services to communicate, making them a natural fit. A containerized application will still probably need to access sensors, peripherals, or other hardware on the embedded device. This will require an embedded-savvy container environment, since you're very unlikely to see web or cloud containers that are able to write to hardware.

Headless embedded systems or those with specialized displays are a good fit because there's no need to share the UI framework and the screen between containers.

Monolithic. A monolithic container encapsulates the embedded application into a single container for use on a target. This is often the case for devices that have a display and need to run a graphical framework such as Qt. However, the structure of the container itself is very basic; this is probably where most people with existing embedded applications would start.

Microservices. By decomposing the embedded software into microservices, each independent service can be placed into its own container. This is a very modular approach, and it allows changes to modules individually without impacting the rest of the application. (In this model, external access could be provided by a web server that can host browser-based apps for remote access.)

Even if you don't believe the microservice model lives up to its hype, breaking the embedded application into independently containerized components can help insulate the application from its internal dependencies. For example, placing a third-party binary into a separate container can allow the rest of the system to be updated without breaking the app when the binary is static and cannot be changed. Similarly, any component with a different update cadence that your application requires (like partner apps, open source libraries, or protocol stacks) can be containerized with the precise version of the libraries, frameworks, and languages it has been tested against.

You get a lot of bang for the buck in provisioning, versioning, testing, and building, making your team more effective.

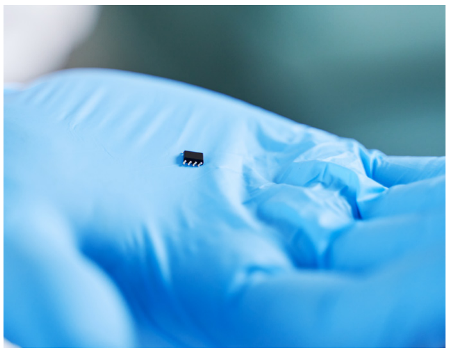
When not to use containers

With all of these positive attributes, are there any downsides to using containers for embedded development?

If we're looking exclusively on developer, build, and QA machines, there seems to be little reason not to use containers. You get a lot of bang for the buck in provisioning, versioning, testing, and building, and once you're past the relatively quick learning curve, you can make your entire team more effective and your results more reproducible.

What about using containers on an embedded target?

Clearly, you'll never see containers in deeply embedded applications where the target has an 8- or 16-bit processor, less than 1MB RAM, or is incapable of running Linux. But for the rest of SOCs that are 32-bit, understanding exactly where and when you might deploy a target with containers is a more nuanced question. Our recommendation would be to start by incorporating containers in your normal development workflow. Once you've accumulated some expertise in them, you'll be better able to recognize if they're viable for your project. Your team will also have acquired the skills necessary to deploy them successfully in an embedded context.



You're not going to containerize microcontrollers and deeply embedded applications; you need a 32-bit memory-protected version of Linux at a minimum.

How are a container and a virtual machine different?

Few embedded developers would consider using a full VM on target hardware boards – they are rarely powerful enough. However, because many engineers are familiar with VMs through other development work, they can provide a useful comparative technology for understanding containers in more detail.

Specifically, this comparison can help you start developing the criteria needed to understand whether containers are right for your embedded target. By examining a few key areas where these technologies differ, you can get an idea of where and why containers may be sufficiently efficient to use in embedded applications.

Each container has its own files, which can be transparently mixed with files from shared volumes and host folders.

	Containers	Virtual machines (VMs)
Architecture	• Containers are application-centric • OS-level virtualization is achieved through user-space abstractions and namespace isolation • Hypervisors are unnecessary; container services use OS features	• VMs are hardware-centric • The guest machine is emulated with a complete stack of virtualized hardware • Hardware assistance and/or hyper-visors are required
Operating system (OS)	• The OS and kernel are shared between all applications and the host OS	• It's possible to host a unique OS (Windows IoT, Linux, QNX, etc) in each VM
File system	• Each container has its own files, which can be transparently mixed with files from shared volumes and host folders • Containers have more flexibility on how their files are managed	• VM filesystems reside completely within VM images • VMs can only access their host's files through virtual networks and server apps • VM filesystems do not have to be the same as host filesystems
Memory	• Memory consumption for containers is nearly the same as for standard applications • App memory allocations come directly from operating systems • Containers can use any available memory or be constrained • Containers share read-only portions, so running multiple containers only pays a one-time hit on largest RAM use	• VMs contain entire OS memory layouts so their memory consump-tion is much more than single applications • App memory allocations come from the VM OS, which requests pages from the host OS (or are reserved by the host OS) • VM memory use is generally fixed and instantiated for each VM
Start-up time	• Since the OS is already booted, only con-tainer initialization is required, generally taking much less time	• A full OS boot is required; optimiz-ing this takes additional work but regardless is generally longer
Image size	• Containers are generally smaller than VMs	• VMs are generally larger than containers

Containers are designed for easy versioning of images with built in version management.

	Containers	Virtual machines (VMs)
Layering/ versioning	• Containers can be built from multiple layers (called multistage builds), allowing for clean isolation of developer and production environments, environment portability, and other build configurations • Designed for easy versioning of container images with built-in version management • Docker images are stateless and do not contain modified files	• Layering must be done manually if needed • VM images are huge binary files not appropriate for version control • Without developer intervention, VM images aren't automatically state-less making them much harder to archive, duplicate, and test reliably
Security	• The barrier between containers and the OS is thinner • Containers make syscalls into the kernel, providing a larger attack surface	• Virtualized hardware isolates VMs from each other • VM calls the host OS services exclu-sively through the host hypervisor, so attack surfaces are much smaller

Embedded container products

Lots of companies are providing container solutions that will work in embedded systems. For starters companies like ARM, Daynix, Docker, Mentor, Toradex, Windriver, and Xilinx are investing in the technol-ogy. We'll take a quick look at a couple of the more notable ones here, but as products and features are constantly being updated, it's always best to check with the manufacturer for specifics.

Docker

This is the granddaddy of all container companies. As docker is the first and most comprehensive container (and it's open source), it forms the core of most implementations. If you're putting a container on your system, you're almost certainly going to have to learn the ins and outs of the docker command and common development patterns. You might also want to look at a quick "getting started with docker on ARM" guide.

Toradex provides a family of embedded SoCs that are able to load the developer's containers out-of-the-box.

```
                    break
        return OutPlanList
    #-------------------------------
    # Main
    #-------------------------------
    if __name__ == "__main__":
        s_area = str(sys.argv[1]) # '440602' area code
        target_date = str(sys.argv[2]) # '20190901'

        year = int(target_date[0:4])
        month = int(target_date[4:6])
        day = int(target_date[6:8])
        target_date = datetime.date(year, month, day)
        wb = Workbook()
        ws = wb.active
        ws.title = "Hotel list"
        initializeSheet(ws)

        for term in range(0,7):
            for i in range(1, 7): #
                plan_list = apicall(s_area, target_date, str(1))
                outputOnSheet(ws,target_date,plan_list, i, row +
```

Don't want to figure out how to put a Python build on your target? Containers can be a perfect solution.

Toradex

With Torizon, Toradex provides a family of embedded SoCs that are already fully containerized. The embedded hardware ships with the Torizon image – a yocto kernel preloaded with Toradex tools and the docker command – so it's able to load the developer's containers out of the box. Toradex also provides many pre-built container images to enable quick and easy C/C++, Python, and .NET development as well as Qt and Crank Storyboard GUIs. Toradex also provides tutorials, videos, and articles focused on getting up to speed with embedded containers, making them a go-to resource for embedded developers.

Kubernetes

Also known as k8s for lazy typists, Kubernetes is a container management and orchestration system. With features to support scaling, storage orchestration, service discovery, load balancing, and automated rollouts, most embedded users (especially those who are new to containers) will find Kubernetes a bit of overkill. It's also not a good fit for the resource requirements of most embedded devices in its current incarnation.

However k3s or Kubernetes-light is a better fit. Having been specifically developed for IoT and Edge computing, k3s replace the complex parts of Kubernetes with more limited alternatives to trim down the footprint, making them much more suitable for deployment on embedded devices. Developers who work on embedded applications with complex distributed deployments, large setups that span multiple systems, or systems that require regular online updates might want to investigate k3s to see if they make sense for their product.

Container wrap-up

It can take some getting used to the idea of containers, especially for established embedded practitioners who, approaching everything with a mindset of optimal efficiency, initially might find them a bit excessive. When considering just the off-target environment, an

If you gradually incorporate containers into your development workflow, you'll be ready when the time comes to containerize your targets.

embedded development team using containers will see immediate gains from developer convenience, testing efficiency, and workflow streamlining.

Containers aren't for everyone; you'll want to assess your developers, build, and deployment needs first.

Whether containers also make sense on-target is at present still an open question. There are many benefits like dependency insulation, simple provisioning, and runtime security. But to understand if it makes sense for your project, you'll need to consider things like:

- Does your hardware vendor provide images that are pre-loaded with container software?
- Do you have reason to update different portions of your application independently (diverging dependencies, multiple suppliers, different update frequencies)?
- Can your product be cleanly divided into independent components that are updated independently?
- Do you have OTA requirements where a container-based solution could help you roll out new changes?
- Does your product have large-scale complex services (such as industrial or building automation) where modular updates are essential?

Embedded containers are an exciting and evolving technology. We've provided a lot of ideas of where you may be able to use containers in your development practice and products. If you gradually incorporate containers into your development workflow, you'll be experienced and ready when the time comes to containerize your targets. And as always, we're happy to help you understand the right fit if you're still looking for some answers after this brief introduction.

Have container questions? We've got container answers.

About the KDAB Group

The KDAB Group is the world's leading software consultancy for architecture, development and design of Qt, C++ and OpenGL applications across desktop, embedded and mobile platforms. KDAB is the biggest independent contributor to Qt and is the world's first ISO 9001 certified Qt consulting and development company. Our experts build runtimes, mix native and web technologies, solve hardware stack performance issues and porting problems for hundreds of customers, many among the Fortune 500. KDAB's tools and extensive experience in creating, debugging, profiling and porting complex applications help developers worldwide to deliver successful projects. KDAB's trainers, all full-time developers, provide market leading, hands-on, training for Qt, OpenGL and modern C++ in multiple languages.

www.kdab.com