Digi Containers: Rapidly Deploy, Monitor, and Manage Applications at Scale

Introduction

Digi Containers are available as an add-on service that simplifies and centralizes the process of building, deploying and running custom applications on devices managed with Digi Remote Manager® (Digi RM). With a Digi RM license and Digi Containers, you can deploy containerized programs or Python applications at scale on any device running DAL OS.

Implemented via Lightweight Linux Containers (LXC), Digi Containers make the process portable, scalable, secure, fast and efficient. Digi empowers companies to orchestrate and manage a complex series of containers in various structures and configurations across enterprise, industrial, transportation and other use cases.

Theory

LXC is a combination of Linux cgroups and chroot. A chroot (CHange ROOT) is a method by which an application sees a specified location as its root file system. For example, if you had a directory in your home called rootfs, and under that had a directory called bin containing applications, you could specify ~/rootfs as the root for the chroot. This gives:

Native system
~/rootfs
~/rootfs/bin
~/rootfs/bin/ls
~/rootfs/bin/sh
.....

Setting the chroot to ~/rootfs would mean any process running in the chroot would just see:

/bin
/bin/ls
/bin/sh
.....

This allows a chroot environment to look like a different device — a virtual device. The problem is that a process running in the chroot still has access to the entire device and is in no way partitioned/sandboxed from the running system. An application inside the container can easily take over the system.

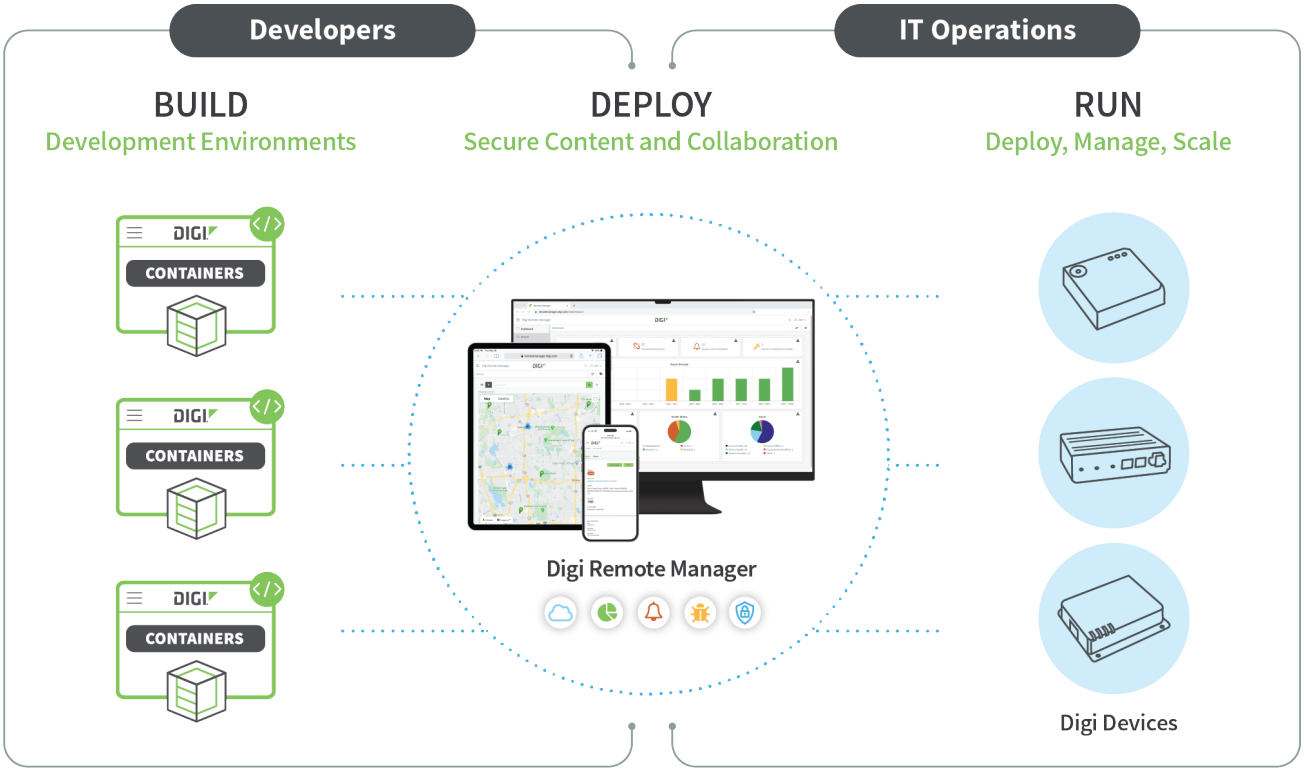
Introducing Containers

Linux cgroups, or Control Groups, can define and control what kind of access various processes have on the system. They can stop processes from accessing hardware (such as CPUs, devices, RAM, disk, or I/O), or other processes being run by the system. This is like sandboxing your process so it can't harm the running system. By combining cgroups and chroot, we can have a device that has its own root file system and can't interfere or harm the device it is running on. **This is called a container.**

LXC is a set of tools that help create and manage the container. For all intents and purposes, it is a virtual machine. The only thing in common with the physical device is the running kernel. This means that processes running inside a container run at native speed, as they are actually running directly on the host device.

If access to a host device/service is required, cgroups can be instructed to allow access to that device/service. For example, the container may be able to access the network of the host device, or a serial port. An added security measure is that cgroups can map user IDs such that inside the cgroup a user ID may be 0, but on the real system it may be 100000. LXC maps the user IDs so that even if a process managed to escape the container, it is mapped to a user with little to no rights on the host device.

For more information, visit:
www.digi.com
877-912-3444 | 952-912-3444



Containers in DAL OS

A container consists of two parts: the root file system, and the configuration file. The root file system is either an archive, file system image, or a directory tree that becomes the roots of the container. DAL OS accepts a tgz (gzipped tar file), sqfs (squashfile system), or a directory tree as the root file system. The configuration file defines the access the container has on the host system. It defines network access, common directories, devices and user mappings. A process running inside the container will see these resources as "native" to the container — just like a virtual machine.

In order to protect the system security, the configuration file is generated as part of the Digi device's configuration; we don't allow user-defined configuration files. This means users can't request features that could defeat DAL OS security. This configuration file is written by the container action script. The user can select several settings:

- **Clone DAL** — The container will mount standard DAL OS directories like bin, lib, etc.
- **Network device** — Containers can attach to a network bridge on the DAL OS device for independent networking.
- **Serial device access** — The container can access one or more serial ports on the DAL OS device.

Resources

The host can share resources with the container. The resources can be devices (such as eth1 or ttyS1), files (such as /etc/accns.schema), or directories (such as /var/log or /var/run). Any shared resource becomes available inside the container and appears as a native resource. By this, we mean a process running inside the container will believe the device/file/directory is part of its virtual machine. These resources are configured in the container configuration file. In DAL, the configuration file is generated and not under direct user influence. This is so that access to system resources can be tightly controlled.

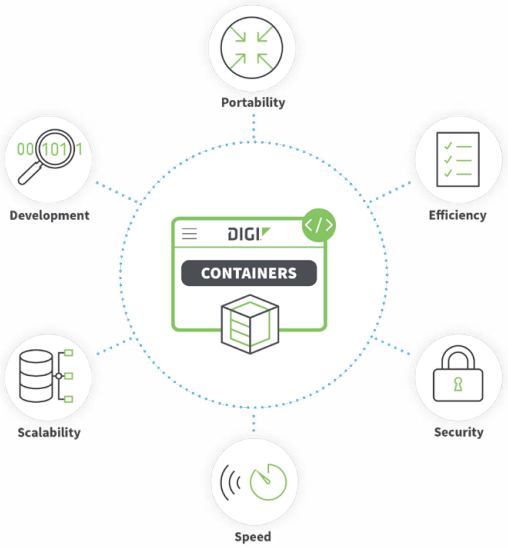
Privileged vs. Unprivileged Containers

Containers that map the user ID to provide another layer of protection are called unprivileged containers. They are the standard container, and any third-party applications would be run in an unprivileged container. This protects the system from unauthorized access. Privileged containers, or system containers, do not map the user ID, and as such, all access to the system is based on the user ID outside the container. By this, we mean if you are root in the container, you are root outside the container. The container still provides protection, as cgroups can limit access to system resources. If access is granted to a resource, the container has the same rights as the equivalent user outside the container. For example:

For more information, visit:
www.digi.com
877-912-3444 | 952-912-3444

Container config -> mount the host /var/log as /var/log in the container
 # ls -l /var/log
-rw-r----- 1 root root 10866 Mar 2 05:10 /var/log/messages

An unprivileged container has its user mapped from 0 to 100000, so the messages file above would be inaccessible either for reading or writing even for the user ID 0 (root) inside the container. A privileged container doesn't map user IDs, so it would have the same access as the root user in the host, and hence would be able to read and write the messages file. This access applies to every resource shared by the host in the container config file.



Persistent vs. Non-Persistent Containers

A non-persistent container is a container that is loaded from the archive each time, and only exists in a RAM disk. When shut down, all data "saved" in the container is lost. This means that each invocation of the container is from a known state and can't be modified. A persistent container creates a file system on the DAL OS device which is used as the root file system. All files saved in the persistent file system will be retained in the DAL OS file system location.

The next invocation will present the same file system that was present when the container was last shut down. In other words, the container behaves like a conventional device with non-volatile storage. If the container root file system is contained in a squash file system (sqfs), then a hybrid model is used. The root file system is mounted from the squash file system as read only, and writable directories are mounted on top of this. That allows, for example, for the /bin directory to be read only, but /home to be writable. This is how the Python container works.

Loading a Digi Container

In order to utilize containers on a Digi device, you must first have the **Digi Containers subscription** enabled in your Digi RM account and added to your Digi device. To do so, order the Digi Containers license (SKU name: **DIGI-RM-PRM-CS**). You will need to provide your customer ID for your Digi RM account and order a license for each device you would like to deploy the container(s) onto.

To load a container on a Digi DAL OS device, you simply need the root file system in either squashfile (.sqfs) or gzipped tar file (.tgz) format. This can be loaded via Digi Remote Manager, the web UI, or the admin CLI (command line interface). See the **documentation link here** for instructions on loading the container.

When you add the container, the configuration is automatically generated. This configuration can be edited to enable the required features for this container. If a container is run as persistent, the root file system is written to the DAL OS flash, and is fully writable inside the container. Writing to the flash should be minimized to extend the life of the flash. Running a container in non-persistent mode will extract a clean file system each time the container is run. Non-persistent file systems are based in RAM and will be lost when the container is stopped. This means an external actor can't compromise security on the DAL device as each time the container is run, it starts from a clean state.

Running Containers in DAL OS

All containers run using the DAL OS generated configuration. Containers are started, stopped and queried using the lxc command. Container root file systems can be as simple as a single, statically linked file, or a complete operating system. The syntax of the lxc command is as follows:

lxc <container_name> [-s] [-p] [command [arg1 [arg2 ...]]

The container name is taken from the root file system filename, minus the extension. For example, a root file system file my_container.tgz would create a container with the name my_container. Supplying no parameters will list the available containers and show their current state. Containers can be RUNNING or STOPPED and will show associated network addresses and container type. Python containers are privileged (flagged as unprivileged=false), with all others being unprivileged. Supplying just the container name will, by default, look for and run (if found) /bin/sh inside the container. Providing a **-s** will start the container in the background like a virtual machine rather than running a specific command. The **-p** option runs the container as a persistent container. If a file system has already been created, it will use this as the root file system. Any changes from the previous invocation will be saved and available in this invocation. If the file system doesn't exist, a new one is created using the supplied file system as a base.

For more information, visit:
www.digi.com
877-912-3444 | 952-912-3444

If you want to run a specific command inside a container, you can supply a command with parameters, and the container will start and immediately execute the given command. For example, if you have a container with a custom command you want to run in /bin/my_command, for a container named my_container, you would run it with:

lxc my_container /bin/my_command

This would start a non-persistent container (i.e. a fresh clone of the installed root file system) and run the command my_command in the container's bin directory. Containers can be used anywhere scripts can be used in DAL, for example in scheduled tasks. Instead of running a script stored on the DAL device, you run the lxc command associated with the application inside an installed container.

Running User-generated Code

Containers act as virtual machines. All libraries, applications and files need to be available to the container. You can either link to the native DAL firmware libraries for convenience or supply your own. A user who has their own proprietary code can compile for a target architecture without any knowledge of DAL, provided they supply all required libraries.

As an example, consider customer A. They have an application they have developed that can update firmware on a device via a serial port. The firmware is downloaded from a central site. They want to be able to manage the device using a DAL router. They don't want or need any specific DAL features or services other than to run the application every day. We would create a container with their application, using their tools. The application is called our_app and requires one library — our_library. The configuration file for the app is in /etc.

/bin/our_app
/lib/our_libraries
/etc/our_config_files

Running ldd on "our_app" will show what libraries are required to run the application. A standard C library may be required. If it is, it must be included in the file system. An alternative is to clone DAL, which mounts the standard binary and library directories. To use these libraries, the application will need to be compiled using the DAL tool chain. We compress the file system to firmware_writer.tgz. We add this container to the system, and edit the configuration to allow serial port access, and network access.

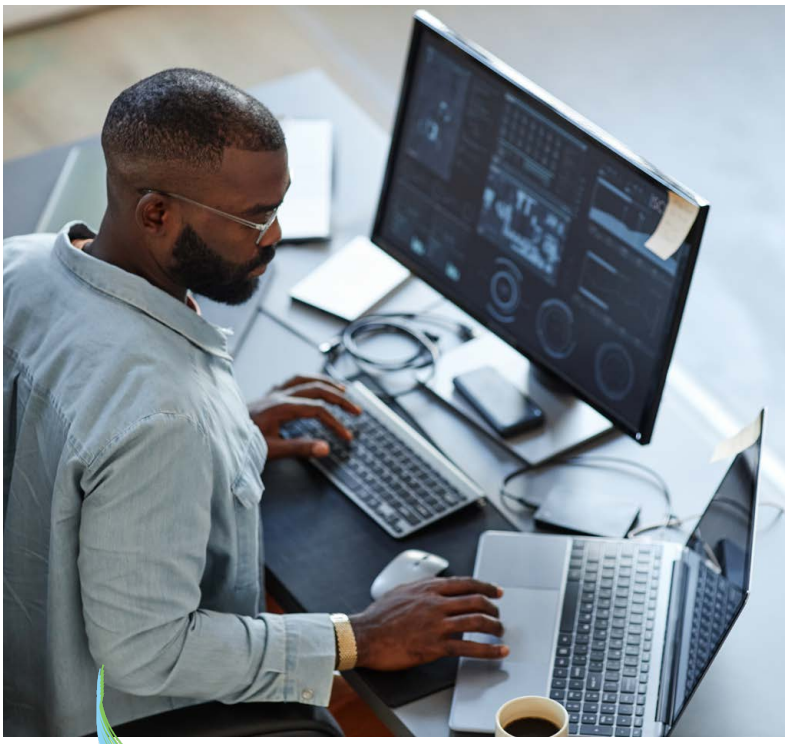
We now add the container command to the scheduled task in the normal DAL way. We don't need to tick "sandbox" as containers are by default sandboxed. We would add the command to the scheduled task:

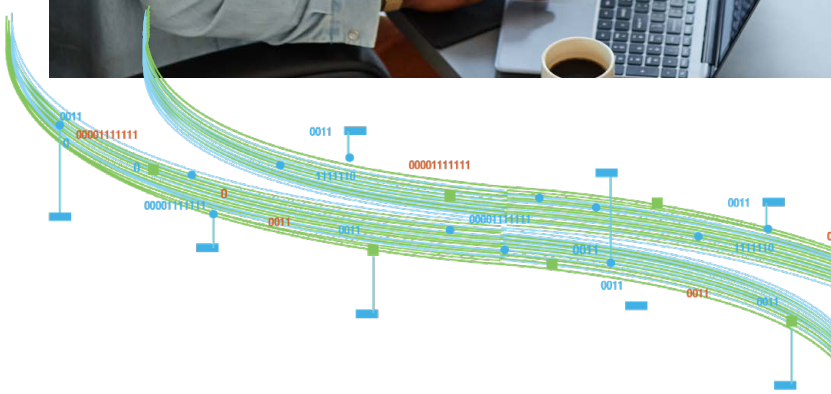
lxc firmware_writer /bin/our_app

If a parameter was required, for example "upload," we would use:

lxc firmware_writer /bin/our_app upload

When the scheduled time arrives, our_app will be run inside the container. It will have access to the network, so it can download the firmware, and has access to the serial port, so it can upload the firmware to the third-party device connected by serial. We have successfully used a proprietary application compiled with no knowledge of DAL in a DAL container.





For more information, visit:

www.digi.com

877-912-3444 | 952-912-3444

Creating a Container

To create a container root file system, you need all files required by the application available to the container. This is either in the form of files in the root file system, or a link to a DAL OS file/directory. If the container is unprivileged, then the file system rights need to be set to the DAL user ID of the container user. The root user (0) in the container is mapped to the DAL OS user ID 165536. User ID 1 is mapped to 165537 and so on. Once the root file system has been created, the files need to be changed so that they are owned by user 165536 (or whichever user is required). Use the following command, where rootfs is the directory in which your container root file system has been created:

```
chown -R 165536:165536 rootfs
```

Once the permissions are updated, you can tar up the file with:

```
tar zcvf my_container.tgz rootfs
```

NOTE: You need to include the rootfs directory, and it must be named rootfs in the archive.

You can now install the container on the DAL OS device, select the resources it has access to, and run it. The simplest container is one that uses the DAL OS binaries and utilities (Clone DAL option). The following is a listing from a minimal container that provides a shell access based on the current DAL OS binaries/libraries.

```
# tar tvf test_lxc.tgz
drwxrwxrwx 165536/165536     0 2021-06-25 13:50 rootfs/
drwxr-x--- 165536/165536     0 2021-06-25 11:10 rootfs/etc/
-rw-r----- 165536/165536    80 2021-06-25 11:18 rootfs/etc/passwd
-rw-r----- 165536/165536    30 2021-06-25 11:18 rootfs/etc/group
-rw-r----- 165536/165536    99 2021-06-25 10:44 rootfs/etc/profile
drwxr-x--- 165536/165536     0 2021-06-25 10:36 rootfs/tmp/
```

The container configuration is set to “Clone DAL,” and network enable. This will give a fully functional shell environment that can use standard DAL OS tools to make network connections. Below is a simple example of using this container. One thing to note is that the profile of the container is set to provide the prompt “lxc #” when inside the container. This makes it easier to know if you are inside the container or on the normal DAL OS system.

```
# lxc test_lxc
lxc # ping -c 1 192.168.210.1
PING 192.168.210.1 (192.168.210.1) 56(84) bytes of data.
64 bytes from 192.168.210.1: icmp_seq=1 ttl=64 time=0.544 ms
--- 192.168.210.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.300/0.422/0.544/0.122 ms
lxc # date
Wed Mar  2 05:55:35 UTC 2022
lxc # cat /etc/profile
export PS1=”lxc # “
[ $(df /tmp | grep -c tmpfs) -ne 1 ] &&  mount -t tmpfs -o size=40M tmpfs /tmp
lxc # exit
#
#
#
# lxc test_lxc /bin/cat /etc/profile
export PS1=”lxc # “
[ $(df /tmp | grep -c tmpfs) -ne 1 ] &&  mount -t tmpfs -o size=40M tmpfs /tmp
#
#
#
# lxc
NAME          STATE   AUTOSTART GROUPS IPV4 IPV6 UNPRIVILEGED
test_lxc      STOPPED 0         -      -    -    true
#
```

Conclusion

Implementing Digi Containers via LXC (Lightweight Linux Containers) provides users of DAL OS-based Digi devices a secure environment to develop, distribute, and run custom programs or Python applications.

Further documentation and details on utilizing containers on DAL OS devices can be found in our **user guides** and **Containers SDK**.

If your team needs assistance, **Digi Professional Services** can help. Reach out if you want to get in touch and learn how our team can support your goals.

For more information, visit:

www.digi.com

877-912-3444 | 952-912-3444