# Use containers to build Azure Sphere apps

Azure Sphere provides a container that packages the SDK in a stand-alone Linux environment. By using a container with a pre-defined build environment, you can avoid the steps of installing (or uninstalling and then reinstalling) the correct SDK build environment. You can also modify the build environment to suit your own needs and replicate that environment to all your host machines at the same time with uniform results.

## What is a container?

Containers are portable packages that come with their own lightweight environments that run on a host machine's kernel. Containers are lightweight because they use shared layers. These layers can be shared slices of an operating system or shared applications. Layers avoids the overhead of a virtual machine, which contains an entire operating system and all associated applications. Sharing allows containers to be small and boot quickly.

Containers can be downloaded from a container registry such as the Microsoft Container Registry.

## What containers bring to Azure Sphere

The container for the Microsoft Azure Sphere SDK build environment provides you with a pre-made development environment. The container provides the following:

- The version of Ubuntu Linux for the current Azure Sphere release
- The current version of the Azure Sphere SDK for Linux
- Additional tools needed by the SDK such as CMake and Ninja

Azure Sphere uses Docker containers that are configured with Dockerfile text files. You can author Dockerfiles that use a base container image to create a custom container for building Azure Sphere applications. Running the customized container downloads the latest base image if it is not on your host machine, builds the new customized container if needed, builds the specified application, and exits. You can then copy the output of the application build to a host machine that has the Azure Sphere SDK installed and

sideload the application to a device. The custom build container is not typically used interactively, but it can be, for example, to diagnose build issues.

The Azure Sphere container cannot be used to recover devices, send Azure Sphere commands to a device, or debug applications, because USB pass-through from a host to a device is not supported.

## Install Docker Desktop for Windows

This topic describes how to use Docker Desktop for Windows to build Azure Sphere applications in a container. To build apps in a Docker container on Linux, you can use the same azurespheresdk container from the Microsoft Container Registry.

Before you can download and run a Docker container, you must install the Docker Desktop on Windows. Make sure you enable Hyper-V Windows features. You may need to reboot after the installation.

After you have installed Docker Desktop for Windows, you must start it from the Windows Start menu. Look for the whale icon in the Notifications area to alert you to the status of Docker Desktop. When the initialization is complete, Docker Desktop will launch an onboarding tutorial that may be helpful.

Linux is the default container type for Docker Desktop on Windows. Azure Sphere uses Linux containers. Make sure that Linux containers are the default by checking the Docker Desktop menu. Open the menu from the Docker Desktop icon and make sure that one of the options is "Switch to Windows containers...". This means that Linux is the current default type of container.

Wait until the Docker Desktop whale icon animation stops. The icon may be in the hidden Notifications area. Hover over the icon to see the Docker Desktop status.

## Use the Azure Sphere SDK build environment container to build sample apps

You can use a container interactively by entering it and issuing command; however, it is more efficient to capture the steps necessary for building your applications in a file that Docker can use to build a custom image based on the original Azure Sphere image. This ensures the build process is repeatable and consistent. By default this file must be named Dockerfile and be in the $PATH where the docker command is run.

The following steps provide an outline for creating Dockerfile instructions to build Azure Sphere samples. You can adjust these steps for your own needs.

1. Create a new container based on the mcr.microsoft.com/azurespheresdk container.
2. Clone the Azure Sphere samples repo from GitHub.
3. Create a directory to store your sample in when it is built.
4. Create an environment variable to specify the sample you want to build.
5. Run CMake to build the sample and place it in the specified directory.

## Create a Dockerfile for building samples

To build a Docker image based on the Azure Sphere image but with custom build functionality, create a text file with the following Docker instructions:

```
FROM mcr.microsoft.com/azurespheresdk AS azsphere-samples-repo

RUN git clone https://github.com/Azure/azure-sphere-samples.git

FROM azsphere-samples-repo AS azsphere-sampleapp-build

RUN mkdir /build
WORKDIR /build

ENV sample=HelloWorld/HelloWorld_HighLevelApp

CMD cmake -G "Ninja" \
-DCMAKE_TOOLCHAIN_FILE="/opt/azurespheresdk/CMakeFiles/AzureSphereToolchain.cmake" \
-DAZURE_SPHERE_TARGET_API_SET="latest-lts" \
-DCMAKE_BUILD_TYPE="Debug" \
/azure-sphere-samples/Samples/${sample} && \
ninja
```

This file uses the **ENV** environment variable to specify the sample to be built. Set a new value for **ENV** to build a sample different from **HelloWorld/HelloWorld_HighLevelApp**.

See Line-by-line discussion of the Dockerfile instructions for more details about the Dockerfile instructions.

## Build the default sample app using the Dockerfile

There are three steps needed to build a sample app using a custom Dockerfile:

1. Build the image from the Dockerfile using the Azure Sphere Developer Command prompt:

   ```
   docker build --target azsphere-sampleapp-build --tag azsphere-sampleapp-build .
   ```

   The `--target` option specifies which part of a multi-stage build is to be used. The `--tag` option specifies a name of the image and must be lowercase only. Docker images must always use lowercase letters only. If you do not specify a name with `--tag`, the image will have a 12-digit number that isn't easy to work with. Don't forget the period at the end of the command. You can list the images with the `docker images` command.

   Docker will build an image named **azsphere-sampleapp-build** based on the file named "Dockerfile". If your Dockerfile is named something else, use the `--file` option to specify the name.

2. Give the container a simpler name using the `--name` option. The `run` command will enter the container and build the sample specified by the **ENV** environment variable. Use the Azure Sphere Developer Command Prompt to give this command:

   ```
   docker run --name hello_hl azsphere-sampleapp-build
   ```

   The sample app (**HelloWorld/HelloWorld_HighLevelApp**) will be built and will be placed in the `/build` directory inside the container. When the container is finished running, it will exit and take you back to the Azure Sphere Developer Command Prompt.

   **Note**

   This command builds the app without any interaction and exits the container after the build is finished. The container is still active after you exit. This is because you did not specify the `-it` or `--rm` options. You can later use the `docker run` command again on the container without rebuilding it, as long as Docker Desktop is running.

3. Copy the results of your build from inside your container to your host machine environment. Use the Azure Sphere Developer Command Prompt to give this command:

   ```
   docker cp hello_hl:/build .
   ```

This command copies the contents of the `/build` directory inside the **hello_h1** container to the directory on your host machine that you issue the command from. The `/build` directory is specified as the working directory (WORKDIR) that the sample is to be compiled to. Note that you are still outside the container but issuing commands to it using the docker **cp** command. Don't forget the period at the end of the command.

## Build a different sample using the custom Dockerfile

To build a different sample, for example, the GPIO sample, provide the path to the GPIO sample.

```
docker run --name gpio_hl --env sample=GPIO/GPIO_HighLevelApp azsphere-sampleapp-build
```

After the build is complete, copy the result from inside your container to your host machine environment:

```
docker cp gpio_hl:/build .
```

Don't forget the period at the end of the command.

After your package has been copied to your host machine environment, you can use Azure Sphere CLI commands from Windows or Linux to deploy your application. For more information, see Deploy the Application.

Device interaction via USB from a container is not supported.

## Line-by-line discussion of the Dockerfile instructions

Each part of the Dockerfile created in Create a Dockerfile for building samples is explained below.

### Prepare for multiple builds

```
FROM mcr.microsoft.com/azurespheresdk AS azsphere-samples-repo
```

This line sets up a new build, **azsphere-samples-repo**, based on the original **microsoft.com/azurespheresdk** container.

### Download the Azure Sphere samples

```
RUN git clone https://github.com/Azure/azure-sphere-samples.git
```

This line clones all the samples from the Azure Sphere samples repo.

### Add another targetable multi-stage build

```
FROM azsphere-samples-repo AS azsphere-sampleapp-build
```

This line adds a new build based on the **azsphere-samples-repo** build.

### Set the working directory inside the container

```
RUN mkdir /build
WORKDIR /build
```

These lines create a new working directory.

### Create a default environment variable to specify sample

```
ENV sample=HelloWorld/HelloWorld_HighLevelApp
```

This line creates an environment variable that specifies the sample to be built. In this case, it is the HelloWorld_HighLevelApp sample. The environment variable can be overridden to specify any sample name and path.

### Run CMake and Ninja to build a package

```
CMD cmake -G "Ninja" \
-DCMAKE_TOOLCHAIN_FILE="/opt/azurespheresdk/CMakeFiles/AzureSphereToolchain.cmake" \
-DAZURE_SPHERE_TARGET_API_SET="latest-lts" \
-DCMAKE_BUILD_TYPE="Debug" \
/azure-sphere-samples/Samples/${sample} && \
ninja
```

This section uses CMake to specify parameters used when invoking Ninja to build the package.

After the build finishes, the container will stop running.

## Docker tips

These tips may help you work with Docker more effectively.

## Use the docker run command to explore the base container interactively

Use the Azure Sphere Developer Command Prompt to enter this command:

```
docker run --rm -it mcr.microsoft.com/azurespheresdk
```

In this example, **mcr.microsoft.com/azurespheresdk** is the name of the image the container is created from. Note that the **--rm** option shuts down the container after it runs and the **-it** option specifies interactive access to the container.

The Azure Sphere SDK build environment Docker container is provided by the Microsoft Container Registry (MCR) and is available to the public.

If the container is already on your local machine, it will not be downloaded again.

The download and setup may take several minutes. The build environment includes everything needed to build a package using the Azure Sphere Linux SDK.

After the `run` command is complete, your command prompt will change to a "#" sign. You are now inside a Linux-based Docker container. Typing **ls** will show you the current Linux directory inside the container, similar to this listing:

```
bin   cmake-3.14.5-Linux-x86_64  etc   lib    makeazsphere.sh  mnt    opt   root
sbin  sys  usr
boot  dev                        home  lib64  media            ninja  proc  run   srv
tmp  var
```

Type `exit` to leave the container. The container will no longer be available to you and you will need to create it again with this command:

```
docker run --rm -it mcr.microsoft.com/azurespheresdk
```

If you don't use the `--rm` option, the container will not be deleted when you exit.

## Container identification

When you build a new container, it will have an ID such as `a250ade97090` (your ID will be different). For many Docker commands, you must use the ID instead of the **microsoft.com/azurespheresdk** address.

Here is a typical listing of basic information about the containers on your system using this command:

```
docker ps --all
```

The result will look similar to this:

```
CONTAINER ID        IMAGE                    COMMAND             CREATED
STATUS              PORTS               NAMES
a250ade97090        microsoft.com/azurespheresdk   "/bin/bash"         15 minutes ago
Up 9 seconds                            pedantic_kilby
```

Your ID will be different. Note that Docker makes up random names for the container owner. Note that in this example, there is only one container.

## Working inside the container

If you would like to work inside a container on your machine without using the **run** command, use the **exec** command with the container ID and the script in the container you want to run (**/bin/bash**) by typing:

```
docker exec -t a250ade97090 /bin/bash
```

Your command prompt will change to a "#" sign. You are now in a Linux-based Docker container. Typing **ls** will show you the current Linux directory inside the container:

```
bin   cmake-3.14.5-Linux-x86_64  etc   lib    makeazsphere.sh  mnt    opt   root
sbin  sys  usr
boot  dev                        home  lib64  media            ninja  proc  run   srv
tmp  var
```

To leave the container, type the `exit` command.

## Azure Sphere SDK build container limitations

The Azure Sphere SDK build container is designed to build Azure Sphere packages only. It is *not* designed for running Azure Sphere CLI commands, recovering or sideloading devices, or debugging. The container does not have access to USB functions.

## Docker Linux container limitations

A Docker Linux container is not the same as a full installation of Linux. For example, you cannot run Linux GUI applications in a Docker Linux container.

## Use multi-stage build containers to reduce dependencies

The Docker multi-stage build feature allows you to use multiple FROM statements in your Dockerfile to reduce dependencies. Each FROM instruction can use a different base, and each of them begins a new stage of the build.

For more information about Docker multi-stage builds, see Use multi-stage builds.

Multi-stage builds are a recommended by Docker as a best practice. For more information about Docker best practices, see Intro Guide to Dockerfile Best Practices.

## Add a meaningful name to your stage with the AS argument

By default, the stages are not named but do have an ID number. You can make your Dockerfile more readable by adding a meaningful name to the stage by appending **AS** and a name. For example:

```
FROM mcr.microsoft.com/azurespheresdk AS azsphere-samples-repo
```

For more information about using the AS argument in multi-stage commands, see Name your build stages.

## Build the target with a meaningful name as a best practice

When you build a target, you can give it a meaningful name by using the **--tag** option. Meaningful names are useful. For example:

```
docker build --target azsphere-sampleapp-build --tag azsphere-sampleapp-build .
```

For more information about using names with the Docker **build** command, see the Docker build reference.

https://docs.microsoft.com/en-us/azure-sphere/app-development/container-build