

# Contents

## Azure Sphere Documentation

### Overview

#### What is Azure Sphere

### Install

#### Overview

#### Set up Azure Sphere device

#### Update the OS on an early, unused device

#### Set up an account

#### Claim your device

#### Configure networking for development

#### Subscribe to notifications

#### Manage tenant and application access

##### Create an Azure Sphere tenant

##### Limit access to your tenant

##### Obtain admin approval

#### Troubleshoot installation

### Quickstarts

#### Overview

#### Build a high-level application

#### Build a real-time capable application

#### Deploy an application over the air

### Networking

#### Connect to Wi-Fi

#### Connect to Ethernet

#### Use network services

#### Beta topic - Perform service discovery

#### Networking requirements

### Application Development

#### Concepts

[Overview of Azure Sphere applications](#)

[Development environment](#)

[Application manifest](#)

[Lifecycle of an application](#)

[Available memory](#)

[How-to Guides](#)

[Develop high-level applications](#)

[Overview](#)

[Use Beta APIs](#)

[Manage target hardware dependencies](#)

[Defer device updates](#)

[Use GPIO](#)

[Use I2C, SPI, and UART](#)

[Use I2C](#)

[Use SPI](#)

[Use UART](#)

[Use ADC](#)

[Use PWM](#)

[Connect to web services](#)

[Understand memory use](#)

[Manage time and use the real-time clock](#)

[Use device storage](#)

[Communicate with a real-time capable application](#)

[Develop without Visual Studio](#)

[Best practices for high-level applications](#)

[Initialization and termination](#)

[Pass arguments](#)

[Periodic tasks](#)

[Asynchronous events, threads, and concurrency](#)

[Watchdog timers](#)

[Handle and log errors](#)

[Develop real-time capable applications - Beta](#)

## Overview

Develop and debug a real-time capable application

Port an existing application

Use peripherals in a real-time capable application

Manage memory and latency considerations

Communicate with a high-level app

Build and debug an RTApp from the command line

## Troubleshoot

Load and unload applications during testing

Sideload for testing

Remove an application

Use Azure IoT with Azure Sphere

Set up IoT hub

Set up IoT Central

Use an Azure IoT Hub

## Over-the-Air Deployment

### Overview

### Concepts

Deployment basics

Deployment summary

Deployment history

Device software updates

OS update feeds

### How-to Guides

Deploy an application

Prepare the device for updates

Link to feed

Set up device groups for OS evaluation

Update a deployment

Recover the system software

Update an external MCU

## Reference

[azsphere command-line utility](#)

[component](#)

[device](#)

[device-group](#)

[feed](#)

[get-support-data](#)

[hardware-definition](#)

[image](#)

[image-set](#)

[login](#)

[logout](#)

[show-version](#)

[sku](#)

[tenant](#)

[Application Libraries reference](#)

[Base APIs](#)

[application.h](#)

Functions

[Application\\_Socket](#)

[adc.h](#)

Functions

[ADC\\_GetSampleBitCount](#)

[ADC\\_Open](#)

[ADC\\_Poll](#)

[ADC\\_SetReferenceVoltage](#)

Typedefs

[ADC\\_ChannelId](#)

[ADC\\_ControllerId](#)

[eventloop.h](#)

Functions

[EventLoop\\_Close](#)

[EventLoop\\_Create](#)

[EventLoop\\_GetWaitDescriptor](#)  
[EventLoop\\_RegisterIo](#)  
[EventLoop\\_Run](#)  
[EventLoop\\_Stop](#)  
[EventLoop\\_UnregisterIo](#)  
[EventLoopIoCallback](#)

**Enumerations**

[EventLoop\\_IoEvents](#)  
[EventLoop\\_Run\\_Result](#)

**Structs**

[EventLoop](#)  
[EventRegistration](#)

**gpio.h**

**Functions**

[GPIO\\_GetValue](#)  
[GPIO\\_OpenAsInput](#)  
[GPIO\\_OpenAsOutput](#)  
[GPIO\\_SetValue](#)

**Enumerations**

[GPIO\\_OutputMode](#)  
[GPIO\\_Value](#)

**Typedefs**

[GPIO\\_Id](#)  
[GPIO\\_OutputMode\\_Type](#)  
[GPIO\\_Value\\_Type](#)

**i2c.h**

**Functions**

[I2CMaster\\_Open](#)  
[I2CMaster\\_Read](#)  
[I2CMaster\\_SetBusSpeed](#)  
[I2CMaster\\_SetDefaultTargetAddress](#)  
[I2CMaster\\_SetTimeout](#)

[I2CMaster\\_Write](#)

[I2CMaster\\_WriteThenRead](#)

[Typedefs](#)

[I2C\\_DeviceAddress](#)

[I2C\\_InterfacId](#)

[log.h](#)

[Functions](#)

[Log\\_Debug](#)

[Log\\_DebugVarArgs](#)

[networking.h](#)

[Functions](#)

[Networking\\_DhcpServer\\_Start](#)

[Networking\\_DhcpServerConfig\\_Destroy](#)

[Networking\\_DhcpServerConfig\\_Init](#)

[Networking\\_DhcpServerConfig\\_SetLease](#)

[Networking\\_DhcpServerConfig\\_SetNtpServerAddresses](#)

[Networking\\_GetInterfaceConnectionStatus](#)

[Networking\\_GetInterfaceCount](#)

[Networking\\_GetInterfaces](#)

[Networking\\_GetNtpState](#)

[Networking\\_InitDhcpServerConfiguration](#)

[Networking\\_InitStaticIpConfiguration](#)

[Networking\\_IpConfig\\_Apply](#)

[Networking\\_IpConfig\\_Destroy](#)

[Networking\\_IpConfig\\_EnableDynamicIp](#)

[Networking\\_IpConfig\\_EnableStaticIp](#)

[Networking\\_IpConfig\\_Init](#)

[Networking\\_IsNetworkingReady](#)

[Networking\\_SetInterfaceState](#)

[Networking\\_SetNtpState](#)

[Networking\\_SetStaticIp](#)

[Networking\\_SntpServer\\_Start](#)

[Networking\\_SntpServerConfig\\_Destroy](#)

[Networking\\_SntpServerConfig\\_Init](#)

[Networking\\_StartDhcpServer](#)

[Networking\\_StartSntpServer](#)

[Networking\\_TimeSync\\_GetEnabled](#)

[Networking\\_TimeSync\\_SetEnabled](#)

## Structs

[Networking\\_DhcpServerConfig](#)

[Networking\\_DhcpServerConfiguration](#)

[Networking\\_IpConfig](#)

[Networking\\_NetworkInterface](#)

[Networking\\_SntpServerConfig](#)

[Networking\\_StaticIpConfiguration](#)

## Enumerations

[Networking\\_InterfaceConnectionStatus](#)

[Networking\\_InterfaceMedium](#)

[Networking\\_IpConfiguration](#)

[Networking\\_IpType](#)

## Typedefs

[Networking\\_InterfaceMedium\\_Type](#)

[Networking\\_IpConfiguration\\_Type](#)

[Networking\\_IpType\\_Type](#)

pwm.h

## Functions

[PWM\\_Apply](#)

[PWM\\_Open](#)

## Enumerations

[PWM\\_Polarity](#)

## Structs

[PwmState](#)

## Typedefs

[PWM\\_ChannelId](#)

PWM\_ControllerId  
rtc.h

Functions

clock\_systohc

spi.h

Enumerations

SPI\_BitOrder

SPI\_ChipSelectPolarity

SPI\_Mode

SPI\_TransferFlags

Functions

SPIMaster\_InitConfig

SPIMaster\_InitTransfers

SPIMaster\_Open

SPIMaster\_SetBitOrder

SPIMaster\_SetBusSpeed

SPIMaster\_SetMode

SPIMaster\_TransferSequential

SPIMaster\_WriteThenRead

Structs

SPIMaster\_Config

SPIMaster\_Transfer

Typedefs

SPI\_ChipSelectId

SPI\_InterfacId

storage.h

Functions

Storage\_DeleteMutableFile

Storage\_GetAbsolutePathInImagePackage

Storage\_OpenFileInImagePackage

Storage\_OpenMutableFile

sysevent.h

## Functions

[SysEvent\\_DeferEvent](#)  
[SysEvent\\_EventsCallback](#)  
[SysEvent\\_Info\\_GetUpdateData](#)  
[SysEvent\\_RegisterForEventNotifications](#)  
[SysEvent\\_ResumeEvent](#)  
[SysEvent\\_UnregisterForEventNotifications](#)

## Enumerations

[SysEvent\\_Events](#)  
[SysEvent\\_Status](#)  
[SysEvent\\_UpdateType](#)

## Structs

[SysEvent\\_Info](#)  
[SysEvent\\_Info\\_UpdateData](#)

## uart.h

### Functions

[UART\\_InitConfig](#)  
[UART\\_Open](#)

### Structs

[UART\\_Config](#)

### Enumerations

[UART\\_BlockingMode](#)  
[UART\\_DataBits](#)  
[UART\\_FlowControl](#)  
[UART\\_Parity](#)  
[UART\\_StopBits](#)

### Typedefs

[UART\\_BaudRate\\_Type](#)  
[UART\\_BlockingMode\\_Type](#)  
[UART\\_DataBits\\_Type](#)  
[UART\\_FlowControl\\_Type](#)  
[UART\\_Id](#)

[UART\\_Parity\\_Type](#)

[UART\\_StopBits\\_Type](#)

[wificonfig.h](#)

[Functions](#)

[WifiConfig\\_AddNetwork](#)

[WifiConfig\\_ForgetAllNetworks](#)

[WifiConfig\\_ForgetNetwork](#)

[WifiConfig\\_ForgetNetworkById](#)

[WifiConfig\\_GetCurrentNetwork](#)

[WifiConfig\\_GetScannedNetworks](#)

[WifiConfig\\_GetStoredNetworkCount](#)

[WifiConfig\\_GetStoredNetworks](#)

[WifiConfig\\_PersistConfig](#)

[WifiConfig\\_SetNetworkEnabled](#)

[WifiConfig\\_SetPSK](#)

[WifiConfig\\_SetSecurityType](#)

[WifiConfig\\_SetSsid](#)

[WifiConfig\\_SetTargetedScanEnabled](#)

[WifiConfig\\_StoreOpenNetwork](#)

[WifiConfig\\_StoreWpa2Network](#)

[WifiConfig\\_TriggerScanAndGetScannedNetworkCount](#)

[Structs](#)

[WifiConfig\\_ConnectedNetwork](#)

[WifiConfig\\_ScannedNetwork](#)

[WifiConfig\\_StoredNetwork](#)

[Enumerations](#)

[WifiConfig\\_Security](#)

[Typedefs](#)

[WifiConfig\\_Security\\_Type](#)

[Terminology](#)

[Hardware Design and Manufacturing](#)

[Overview](#)

## [MT3620 information](#)

[MT3620 support status](#)

[MT3620 Hardware Notes](#)

[MT3620 development board user guide](#)

[MT3620 reference board design](#)

[MCU programming and debugging interface](#)

## [RF test tools](#)

## [Guardian modules](#)

## [Manufacturing guide](#)

[Factory floor tasks](#)

[Cloud configuration tasks](#)

## [Hardware abstraction files for samples](#)

## [Resources](#)

### [Samples and Reference Solutions](#)

### [Release Notes](#)

[Release Notes 19.09](#)

[Release Notes 19.05](#)

[Release Notes 19.02](#)

[Release Notes 18.11](#)

[Release Notes TP 4.2.1](#)

## [Support](#)

## [Additional Resources](#)

# Azure Sphere Documentation

5/30/2019 • 2 minutes to read

Azure Sphere is a secured, high-level application platform with built-in communication and security features for internet-connected devices. It comprises an Azure Sphere microcontroller unit (MCU), tools and an SDK for developing applications, and the Azure Sphere Security Service, through which applications can securely connect to the cloud and web.

For more information, [read the overview](#).

## Get started with your development kit

If you already have an Azure Sphere development kit, complete the installation tasks:

- [Install Azure Sphere](#)
- [Set up an account](#) to use with Azure Sphere
- [Claim your device](#)
- [Configure networking](#)
- [Subscribe to notifications](#)

Then follow these quickstarts to build and deploy an application:

- [Build your first high-level application](#)
- [Build your first real-time capable application](#)
- [Deploy your application over the air](#)

## Concepts

- [Azure Sphere product overview](#)
- [Applications](#)
- [Deployment model](#)

## Need a development kit?

[Order one!](#)

# What is Azure Sphere?

5/30/2019 • 12 minutes to read

Azure Sphere is a secured, high-level application platform with built-in communication and security features for internet-connected devices. It comprises a secured, connected, crossover microcontroller unit (MCU), a custom high-level Linux-based operating system (OS), and a cloud-based security service that provides continuous, renewable security.

The Azure Sphere MCU integrates real-time processing capabilities with the ability to run a high-level operating system. An Azure Sphere MCU, along with its operating system and application platform, enables the creation of secured, internet-connected devices that can be updated, controlled, monitored, and maintained remotely. A connected device that includes an Azure Sphere MCU, either alongside or in place of an existing MCU(s), provides enhanced security, productivity, and opportunity. For example:

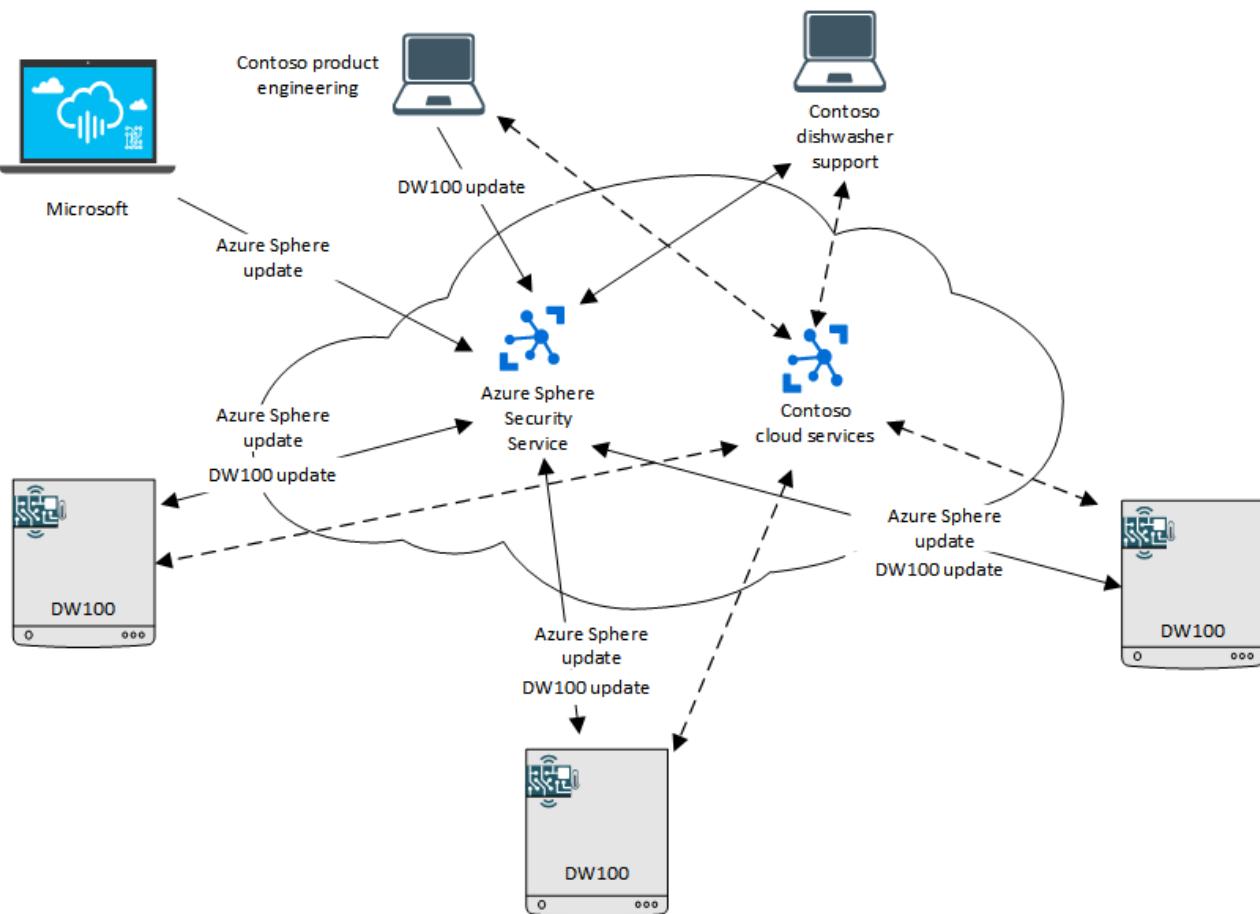
- A secured application environment, authenticated connections, and opt-in use of peripherals minimizes security risks due to spoofing, rogue software, or denial of service attacks, among others.
- Software updates can be automatically deployed over the air to any connected device to fix problems, provide new functionality, or counter emerging methods of attack, thus enhancing the productivity of support personnel.
- Product usage data can be reported to the cloud over a secured connection to help in diagnosing problems and designing new products, thus increasing the opportunity for product service, positive customer interactions, and future development.

The Azure Sphere Security Service is an integral aspect of Azure Sphere. Using this service, Azure Sphere MCUs safely and securely connect to the cloud and web. The service ensures that the device boots only with an authorized version of genuine, approved software. In addition, it provides a secured channel through which Microsoft can automatically download and install OS updates to deployed devices in the field to mitigate security issues. Neither manufacturer nor end-user intervention is required, thus closing a common security hole.

## Azure Sphere scenario

To understand how Azure Sphere works in a real-world setting, consider this scenario.

Contoso, Ltd., is a white-goods product manufacturer who embeds an Azure Sphere MCU into its dishwashers. The DW100 dishwasher couples the MCU with several sensors and an onboard high-level application that runs on the Azure Sphere MCU. The application communicates with the Azure Sphere Security Service and with Contoso's cloud services. The following diagram illustrates this scenario:



### Contoso network-connected dishwashers

Starting from the top left and moving clockwise:

- Microsoft releases updates for the Azure Sphere OS through the Azure Sphere Security Service.
- Contoso product engineering releases updates to its DW100 application through the Azure Sphere Security Service.
- The Azure Sphere Security Service securely deploys the updated OS and the Contoso DW100 application software to the dishwashers at end-user locations.
- Contoso dishwasher support communicates with the Azure Sphere Security Service to determine which version of the Azure Sphere software and the DW100 application software should be running on each end-user device and to glean any error-reporting data that has been reported to the service. Contoso dishwasher support also communicates with the Contoso cloud service for additional information.
- Contoso cloud services support applications for troubleshooting, data analysis, and customer interaction. Contoso's cloud services may be hosted by Microsoft Azure, by another vendor's cloud service, or by Contoso's own cloud.
- Contoso DW100 models at end-user locations download updated OS and application software over their connection to the Azure Sphere Security Service. They can also communicate with Contoso's cloud service application to report additional data.

For example, sensors on the dishwasher might monitor water temperature, drying temperature, and rinse agent level and upload this data to Contoso's cloud services, where a cloud service application analyzes it for potential problems. If the drying temperature seems unusually hot or cool—which might indicate a failing part—Contoso runs diagnostics remotely and notifies the customer that repairs are needed. If the dishwasher is under warranty, the cloud service application might also ensure that the customer's local repair shop has the replacement part, thus reducing maintenance visits and inventory requirements. Similarly, if the rinse agent is low, the dishwasher might signal the customer to purchase more rinse agent directly from the manufacturer.

All communications take place over secured, authenticated connections. Contoso support and engineering personnel can visualize data by using the Azure Sphere Security Service, Microsoft Azure features, or a Contoso-specific cloud service application. Contoso might also provide customer-facing web and mobile applications, with which dishwasher owners can request service, monitor dishwasher resource usage, or otherwise interact with the company.

Using Azure Sphere deployment tools, Contoso targets each application software update to the appropriate dishwasher model, and the Azure Sphere Security Service distributes the software updates to the correct devices. Only signed and verified software updates can be installed on the dishwashers.

## Azure Sphere and the seven properties of highly secured devices

A primary goal of the Azure Sphere platform is to provide high-value security at a low cost, so that price-sensitive, microcontroller-powered devices can safely and reliably connect to the internet. As network-connected toys, appliances, and other consumer devices become commonplace, security is of utmost importance. Not only must the device hardware itself be secured, its software and its cloud connections must also be secured. A security lapse anywhere in the operating environment threatens the entire product and, potentially, anything or anyone nearby.

Based on Microsoft's decades of experience with internet security, the Azure Sphere team has identified [seven properties of highly secured devices](#). The Azure Sphere platform is designed around these seven properties:

**Hardware-based root of trust.** A hardware-based root of trust ensures that the device and its identity cannot be separated, thus preventing device forgery or spoofing. Every Azure Sphere MCU is identified by an unforgeable cryptographic key that is generated and protected by the Microsoft-designed Pluto security subsystem hardware. This ensures a tamper-resistant, secured hardware root of trust from factory to end user.

**Small trusted computing base.** Most of the device's software remains outside the trusted computing base, thus reducing the surface area for attacks. Only the secured Security Monitor, Pluto runtime, and Pluto subsystem—all of which Microsoft provides—run on the trusted computing base.

**Defense in depth.** Defense in depth provides for multiple layers of security and thus multiple mitigations against each threat. Each layer of software in the Azure Sphere platform verifies that the layer above it is secured.

**Compartmentalization.** Compartmentalization limits the reach of any single failure. Azure Sphere MCUs contain silicon counter-measures, including hardware firewalls, to prevent a security breach in one component from propagating to other components. A constrained, "sandboxed" runtime environment prevents applications from corrupting secured code or data.

**Certificate-based authentication.** The use of signed certificates, validated by an unforgeable cryptographic key, provides much stronger authentication than passwords. The Azure Sphere platform requires every software element to be signed. Device-to-cloud and cloud-to-device communications require further certificate-based authentication.

**Renewable security.** The device software is automatically updated to correct known vulnerabilities or security breaches, requiring no intervention from the product manufacturer or the end user. The Azure Sphere Security Service updates the Azure Sphere OS and your applications automatically.

**Failure reporting.** Failures in device software or hardware are typical in emerging security attacks; device failure by itself constitutes a denial-of-service attack. Device-to-cloud communication provides early warning of potential failures. Azure Sphere devices can automatically report operational data and failures to a cloud-based analysis system, and updates and servicing can be performed remotely.

## Azure Sphere architecture

Working together, the Azure Sphere hardware, software, and Security Service enable unique, integrated approaches to device maintenance, control, and security.

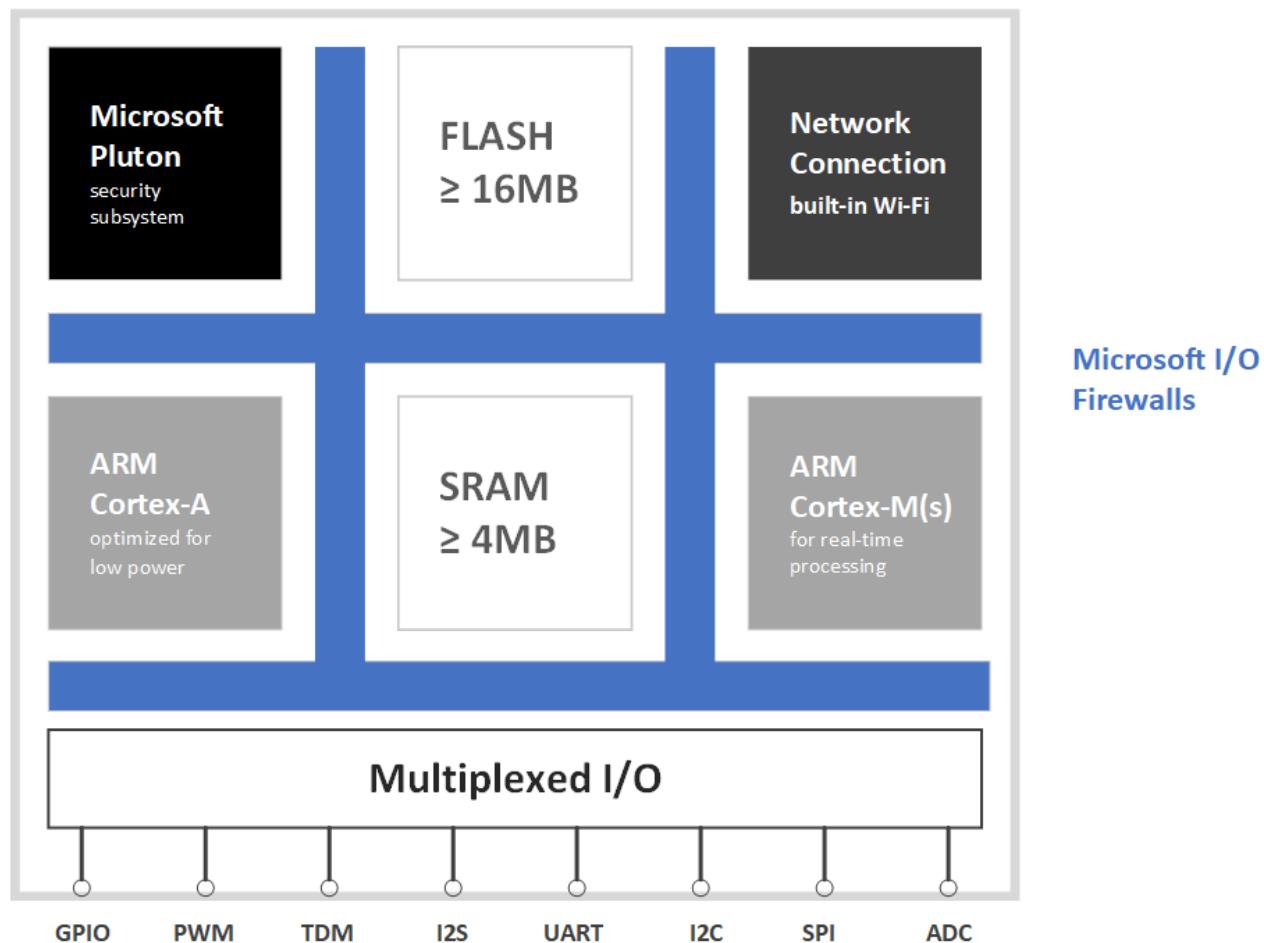
The hardware architecture provides a fundamentally secured computing base for connected devices, allowing you to focus on your product.

The software architecture, with a secured custom OS kernel running atop the Microsoft-written Security Monitor, similarly enables you to concentrate your software efforts on value-added IoT and device-specific features.

The Azure Sphere Security Service supports authentication, software update, and failure reporting over secured cloud-to-device and device-to-cloud channels. The result is a secured communications infrastructure that ensures that your products are running the most up-to-date Azure Sphere OS.

### Hardware architecture

An Azure Sphere crossover MCU consists of multiple cores on a single die, as the following figure shows.



### Azure Sphere MCU hardware architecture

Each core, and its associated subsystem, is in a different trust domain. The root of trust resides in the Pluto security subsystem. Each layer of the architecture assumes that the layer above it may be compromised. Within each layer, resource isolation and compartmentalization provide added security.

#### Microsoft Pluto security subsystem

The Pluto security subsystem is the hardware-based (in silicon) secured root of trust for Azure Sphere. It includes a security processor core, cryptographic engines, a hardware random number generator, public/private key generation, asymmetric and symmetric encryption, support for elliptic curve digital signature algorithm (ECDSA) verification for secured boot, and measured boot in silicon to support remote attestation with a cloud service, as well as various tampering counter-measures including an entropy detection unit.

As part of the secured boot process, the Pluto subsystem boots various software components. It also provides runtime services, processes requests from other components of the device, and manages critical components for other parts of the device.

#### High-level application core

The high-level application core features an ARM Cortex-A subsystem that has a full memory management unit (MMU). It enables hardware-based compartmentalization of processes by using trust zone functionality and is responsible for executing the operating system, high-level applications, and services. It supports two operating environments: Normal World (NW), which executes code in both user mode and supervisor mode, and Secure World (SW), which executes only the Microsoft-supplied Security Monitor. Your high-level applications run in NW user mode.

#### **Real-time core(s)**

The real-time core(s) feature an ARM Cortex-M I/O subsystem that can run real-time capable applications as either bare-metal code or a real-time operating system (RTOS). Such applications can map peripherals and communicate with high-level applications but cannot access the internet directly.

#### **Connectivity and communications**

The first Azure Sphere MCU provides an 802.11 b/g/n Wi-Fi radio that operates at both 2.4GHz and 5GHz. High-level applications can configure, use, and query the wireless communications subsystem, but they cannot program it directly. In addition to or instead of using Wi-Fi, Azure Sphere devices that are properly equipped can communicate on an Ethernet network.

#### **Multiplexed I/O**

The Azure Sphere platform supports a variety of I/O capabilities, so that you can configure embedded devices to suit your market and product requirements. I/O peripherals can be mapped to either the high-level application core or to a real-time core.

#### **Microsoft firewalls**

Hardware firewalls are silicon countermeasures that provide “sandbox” protection to ensure that I/O peripherals are accessible only to the core to which they are mapped. The firewalls impose compartmentalization, thus preventing a security threat that is localized in the high-level application core from affecting the real-time cores’ access to their peripherals.

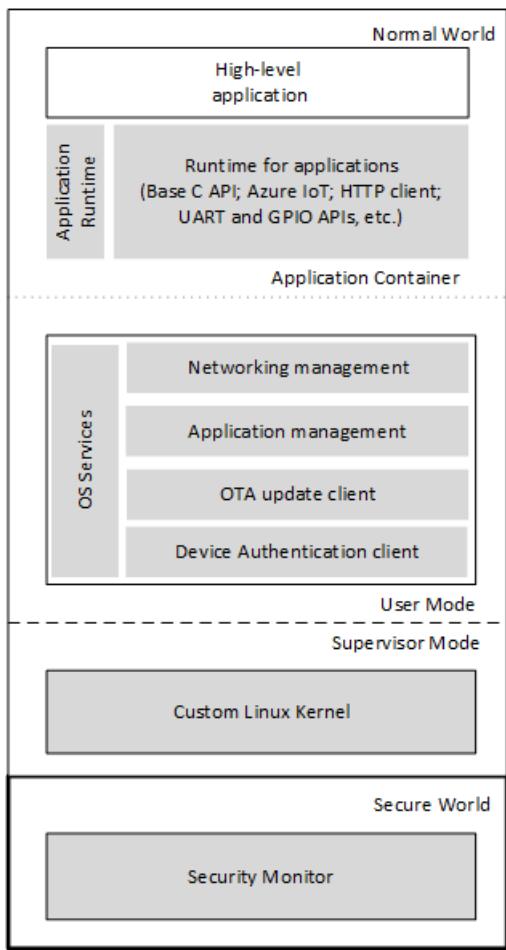
#### **Integrated RAM and flash**

Azure Sphere MCUs include a minimum of 4MB of integrated RAM and 16MB of integrated flash memory.

#### **Software architecture and OS**

The high-level application platform runs the Azure Sphere OS along with a device-specific high-level application that can communicate both with the internet and with real-time capable applications that run on the real-time cores. The following figure shows the elements of this platform.

Microsoft-supplied elements are shown in gray.



## High-level Application Platform

Microsoft provides and maintains all software other than your device-specific applications. All software that runs on the device, including the high-level application, is signed by the Microsoft certificate authority (CA). Application updates are delivered through the trusted Microsoft pipeline, and the compatibility of each update with the Azure Sphere device hardware is verified before installation.

### Application runtime

The Microsoft-provided application runtime is based on a subset of the POSIX standard. It consists of libraries and runtime services that execute in NW user mode. This environment supports the high-level applications that you create.

Application libraries support networking, storage, and communications features that are required by high-level applications but do not support direct generic file I/O or shell access, among other constraints. These restrictions ensure that the platform remains secured and that Microsoft can provide security and maintenance updates. In addition, the constrained libraries provide a long-term stable API surface so that system software can be updated to enhance security while retaining binary compatibility for applications.

### OS services

OS services host the high-level application container and are responsible for communicating with the Azure Sphere Security Service. They manage network authentication and the network firewall for all outbound traffic. During development, OS services also communicate with a connected PC and the application that is being debugged.

### Custom Linux kernel

The custom Linux-based kernel runs in supervisor mode, along with a boot loader. The kernel is carefully tuned for the flash and RAM footprint of the Azure Sphere MCU. It provides a surface for preemptable execution of user-space processes in separate virtual address spaces. The driver model exposes MCU peripherals to OS services and applications. Azure Sphere drivers include Wi-Fi (which includes a TCP/IP networking stack), UART, SPI, I2C, and GPIO, among others.

## **Security Monitor**

The Microsoft-supplied Security Monitor runs in SW. It is responsible for protecting security-sensitive hardware, such as memory, flash, and other shared MCU resources and for safely exposing limited access to these resources. The Security Monitor brokers and gates access to the Pluton Security Subsystem and the hardware root of trust and acts as a watchdog for the NW environment. It starts the boot loader, exposes runtime services to NW, and manages hardware firewalls and other silicon components that are not accessible to NW.

## **Azure Sphere Security Service**

The Azure Sphere Security Service comprises three components: certificate-based authentication, update, and failure reporting.

- **Certificate-based authentication.** The authentication component provides remote attestation and certificate-based authentication. The remote attestation service connects via a challenge-response protocol that uses the measured boot feature on the Pluton subsystem. It guarantees not merely that the device booted with the correct software, but with the correct version of that software.

After attestation succeeds, the authentication service takes over. The authentication service communicates over a secured TLS connection and issues a certificate that the device can present to a web service, such as Microsoft Azure or a company's private cloud. The web service validates the certificate chain, thus verifying that the device is genuine, that its software is up to date, and that Microsoft is its source. The device can then connect safely and securely with the online service.

- **Update.** The update service distributes automatic updates for the Azure Sphere OS and for applications. The update service ensures continued operation and enables the remote servicing and update of application software.
- **Failure reporting.** The failure reporting service, currently under development, provides simple crash reporting for deployed software. To obtain richer data, use the reporting and analysis features that are included with a Microsoft Azure subscription.

# Install Azure Sphere

10/9/2019 • 2 minutes to read

If you have an Azure Sphere development kit that has not yet been used, complete these steps first to get up and running:

- [Install the Azure Sphere SDK](#) and set up your development kit
- [Update the OS](#) if you have an early Seeed MT3620 development kit that has never been used
- [Set up an account](#) to authenticate with Microsoft Azure
- [Claim your device](#)
- [Configure networking](#)
- [Subscribe to Azure Sphere notifications](#) so that you receive the latest information about OS updates and other important news

# Set up your device and install the Azure Sphere SDK

10/9/2019 • 2 minutes to read

Before you can develop or deploy applications for Azure Sphere, you must first set up your device and install the software.

Prerequisites:

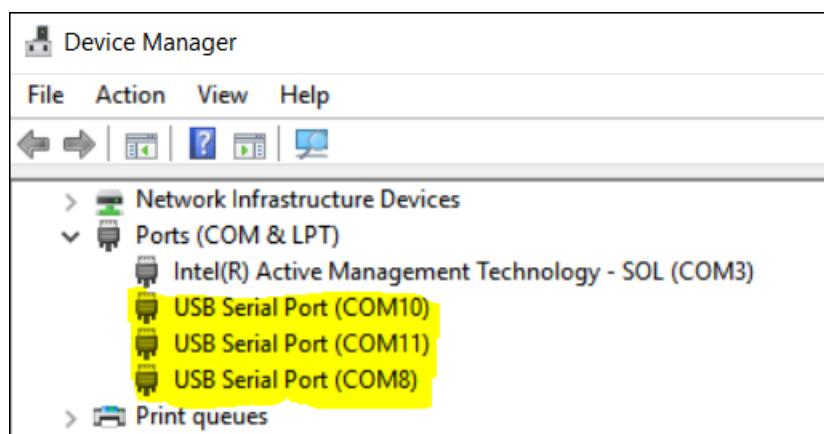
- An [Azure Sphere Development Kit](#)
- A PC running Windows 10 Anniversary Update or later
- An unused USB port on the PC
- Visual Studio 2019 Enterprise, Professional, or Community version 16.04 or later; or Visual Studio 2017 version 15.9 or later. To [install Visual Studio](#), select the edition to install and then run the installer. You can install any workloads, or none.

## Connect the Azure Sphere device

Connect your device to your PC through USB. When plugged in, the device exposes three COM ports.

The first time you plug in the device, the drivers should be automatically downloaded and installed. Installation can be slow; if the drivers are not installed automatically, right-click on the device name in Device Manager and select **Update driver**. Alternatively, you can download the drivers from [Future Technology Devices International \(FTDI\)](#). Choose the driver that matches your Windows installation (32- or 64-bit).

To verify installation, open **Device Manager** and look for three COM ports. The numbers on your COM ports may be different from those in the figure.



If other errors occur, or if you see fewer than three COM ports, see [Troubleshoot Installation](#) for help.

## Install the Azure Sphere SDK Preview for Visual Studio

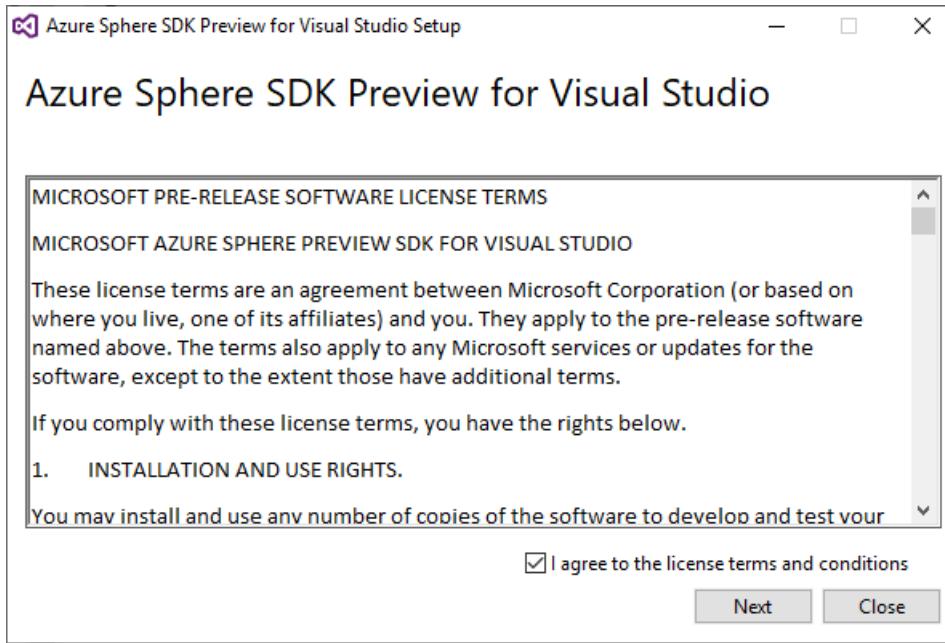
The Azure Sphere SDK Preview for Visual Studio includes:

- A custom Azure Sphere Developer Command Prompt, which is available in the **Start** menu under **Azure Sphere**
- The **azsphere** command-line utility for managing devices, images, and deployments
- Libraries for application development
- Visual Studio extensions to support Azure Sphere development

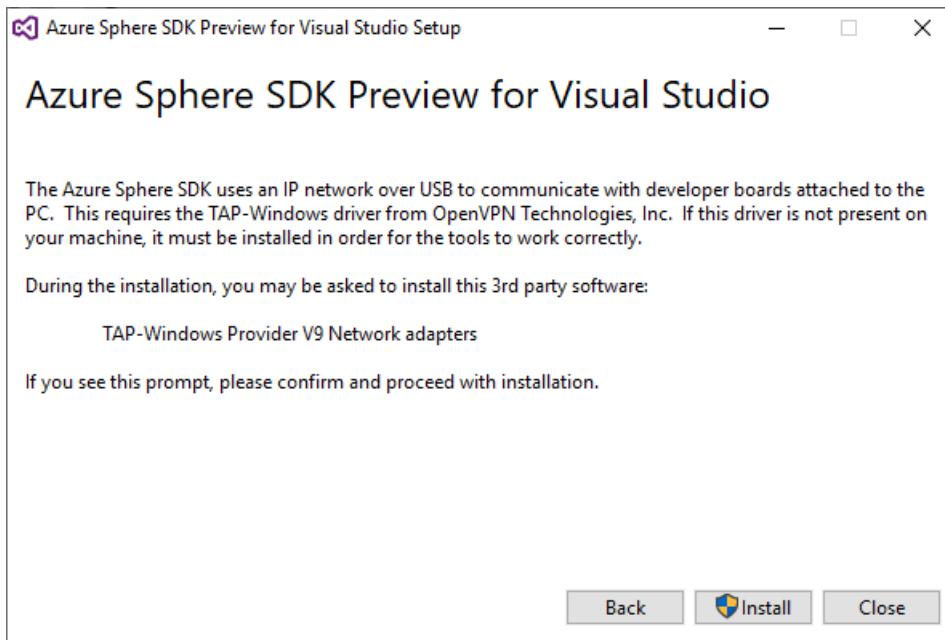
Azure\_Sphere\_SDK\_Preview\_for\_Visual\_Studio.exe installs the complete Azure Sphere software development kit (SDK).

To install the SDK:

1. [Download the Azure Sphere SDK Preview for Visual Studio](#) from Visual Studio Marketplace if you have not already done so. Save the downloaded file on your PC. The SDK requires Visual Studio 2017, version 15.9 or later, or Visual Studio 2019, version 16.04 or later.
2. Run Azure\_Sphere\_SDK\_Preview\_for\_Visual\_Studio.exe from the download to install the SDK. Agree to the license terms and select **Next**.



3. Click **Install** to begin installation. Installation may take up to 10 minutes.



The message "No product to install SDK on" may appear if the installed version of Visual Studio is too old, or if you have just installed Visual Studio for the first time. If you see this message, either update your Visual Studio installation if necessary or restart your PC and return to this step.

4. Accept the elevation prompt if one appears.
5. When setup completes, restart your PC if the setup application requests it. The SDK is installed to all compatible editions of Visual Studio on your PC.

If the installer returns errors, try uninstalling and then reinstalling the tools. To uninstall the tools, rerun the installer or use **Add and Remove Programs** in **Settings**.

## Next Steps

- [Update the OS](#) if you have an older MT3620 reference development board (RDB) that has never been used

# Update the OS

10/9/2019 • 2 minutes to read

All Azure Sphere devices are shipped from the manufacturer with the Azure Sphere OS installed. Most devices can be updated over the internet after they are connected to Wi-Fi. Some early Seeed MT3620 development kits, however, may require manual update if they have never been used.

## IMPORTANT

If your device is already in use and is running the 18.11 OS release (or more recent), it should receive the updated OS after it connects to the internet. You do not need to do anything to update the OS.

Skip this section and proceed to [Set up an account](#) to use with Azure Sphere.

## To update the OS on an unused board

Currently, a few early Seeed MT3620 development kits require manual update. If you have such a board that has never been used, follow these steps to update the OS manually.

1. Connect the board to the PC by USB.
2. Open an Azure Sphere Developer Command Prompt. The command prompt appears in the **Start** menu under **Azure Sphere**. Note that sometimes the **Start** menu index is not updated immediately after installation, so searching for the prompt might fail. If so, browse to **Azure Sphere** and open it.
3. Update your board by using the recovery procedure:

```
azsphere device recover
```

You should see output similar to this, though the number of images downloaded might vary:

```
Starting device recovery. Please note that this may take up to 10 minutes.  
Board found. Sending recovery bootloader.  
Erasing flash.  
Sending images.  
Sending image 1 of 16.  
.  
.  
.  
Sending image 16 of 16.  
Finished writing images; rebooting board.  
Device ID: <GUID>  
Device recovered successfully.  
Command completed successfully in 00:02:37.3011134.
```

## Next Steps

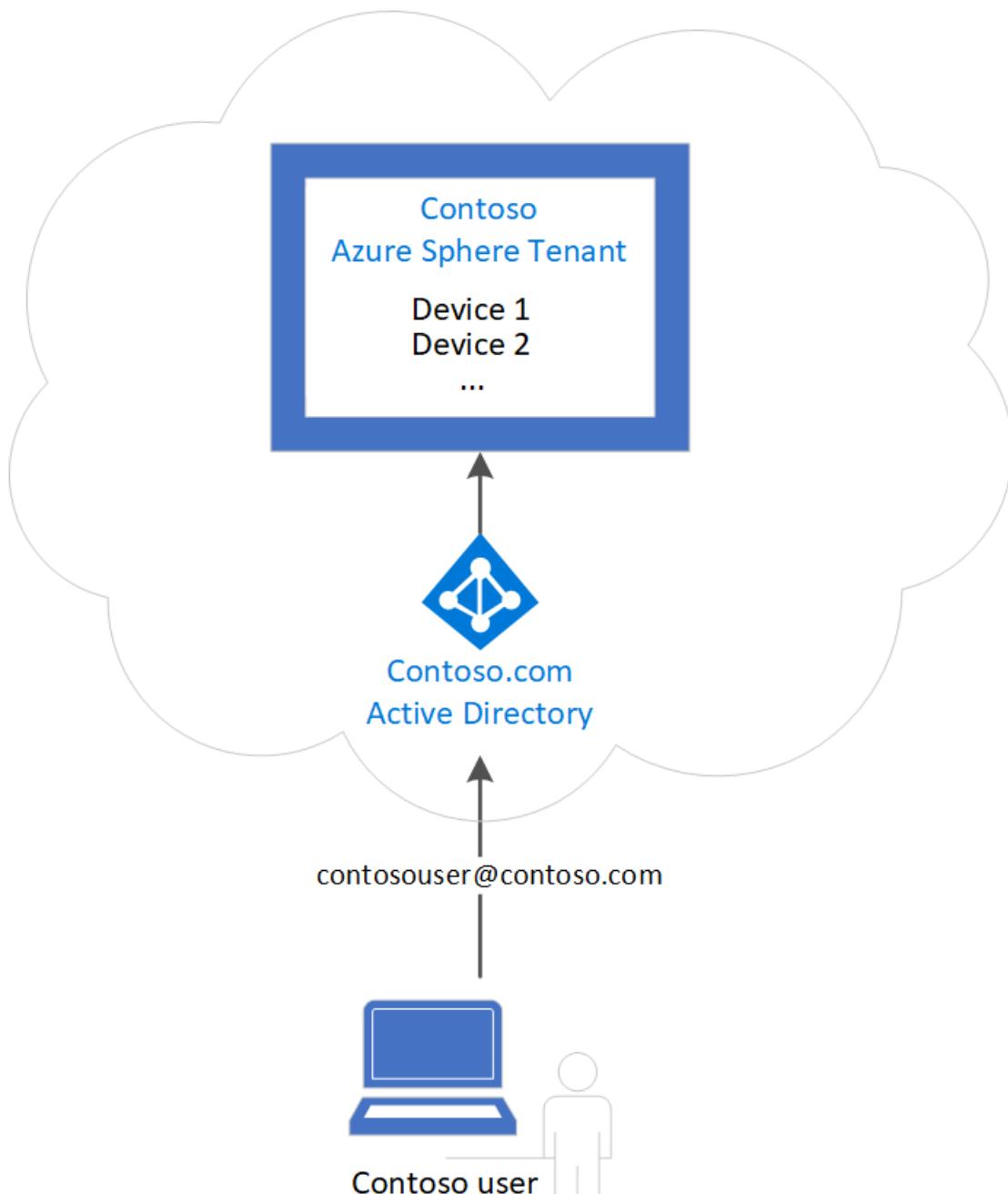
- [Set up an account](#) to use with Azure Sphere

# Set up an account for Azure Sphere

10/9/2019 • 4 minutes to read

Azure Sphere uses Azure Active Directory (AAD) to enforce enterprise access control. Therefore, to use Azure Sphere, you need a Microsoft work or school account (sometimes called an organizational account) that is associated with an AAD.

If you already use Microsoft Azure through work or school, or if you or your employer/school subscribes to any other Microsoft Online services (for example, Office 365 for Business, OneDrive for Business, or InTune), you probably have an account and directory that you can use with Azure Sphere. A personal account (also called an MSA), such as an account that is associated with an Office 365 Home subscription, a personal OneDrive account, or an outlook.com email address, does not provide the necessary AAD.



In the figure, Contoso Corp. has an AAD, so Contoso users can sign in to use Azure Sphere with their Contoso work accounts. After Azure authenticates the sign-in, the Contoso user can create an *Azure Sphere tenant* if

Contoso does not already have one. The Azure Sphere tenant isolates Contoso's Azure Sphere devices from those of all other Azure Sphere customers and enables Contoso personnel to manage them. The Azure Sphere tenant is strictly used for Azure Sphere; it is not the same as an Azure AD tenant.

**TIP**

For help with Azure directories, accounts, tenants, and identities, see [What is Azure Active Directory](#).

## Find out whether your existing account works with Azure Sphere

To find out whether you have an account, open an Azure Sphere Developer Command prompt (on the **Start** menu under **Azure Sphere**) and sign in to Azure Sphere with your work or school account:

```
azsphere login
```

In response, **azsphere** prompts you to pick an account. Choose your work/school account and type your password if required. If you see a dialog box requesting that an admin grant permission to use the Azure Sphere Utility, you'll need to log in as an administrator or [obtain admin approval](#).

If login succeeds, you can use this account with Azure Sphere; proceed to [Next steps](#).

If login fails, the account is not associated with an AAD. If you have another account, try it; if not, you can create a new account. Choose the option that describes your situation:

- I want to [create a new account and directory that are not associated with any other account](#)
- I have an existing personal/MSA account that I use with Azure, and I want to [create a work/school account that is associated with it](#)

## Create a new account and directory that are not associated with any other account

If you don't have a work or school account that you want to use with Azure Sphere, and you have no other account with Microsoft or Azure, you can create a new directory that has a new work/school account. (The Azure documentation refers to this directory as an Azure AD tenant; we call it a "directory" to distinguish it from the Azure Sphere tenant.)

To create a new directory that has a work/school account, visit the [Microsoft Azure Get started page](#).

Fill in the requested information and create a domain name, a user ID, and a password. Provide the details necessary to verify your information. When you click Continue, you will be prompted to sign up for an Azure subscription that is associated with the directory. If you don't want to sign up for an Azure subscription, you can leave the web page.

**IMPORTANT**

An Azure subscription is not required to use Azure Sphere; however, a subscription is required to use Azure IoT Hub or Azure IoT Central. Although you can create an Azure subscription for no charge, the sign-up process requires you to enter a credit card number. Azure provides several levels of subscription service. The Free tier includes the services required to use your device with an IoT Hub.

If you plan to use an Azure IoT Hub, follow the instructions to create an Azure subscription. If prompted, sign into your newly created directory as `userID@domainname.onmicrosoft.com`. Then follow the prompts to sign up for a free Azure subscription. You will need to enter credit card details for verification only.

# Create a work/school account that is associated with the personal/MSA account that you use with Azure

If you have a personal/MSA account that you use with Azure, you can create an associated user identity and directory to use with Azure Sphere.

1. Log in to the [Azure portal](#) using your existing personal/MSA account.
2. Create a user in the directory. In the Azure Portal, click Azure Active Directory on the left side menu and Users on the pane to its right.

The screenshot shows the Microsoft Azure portal interface. On the left, there is a dark sidebar with various service icons and links: 'Create a resource', 'All services', 'FAVORITES' (Function Apps, SQL databases, Azure Cosmos DB, Virtual machines, Load balancers, Storage accounts, Virtual networks), 'Azure Active Directory' (which is highlighted with a red box), and 'Monitor'. The main content area is titled 'default directory - Overview' under 'Azure Active Directory'. It features a search bar at the top. Below the search bar are two sections: 'OVERVIEW' (with 'Overview' and 'Getting started' options) and 'MANAGE' (with 'Users' highlighted with a red box, and other options like 'Groups', 'Roles and administrators', 'Enterprise applications', 'Devices', 'App registrations', 'Application proxy', and 'Licenses').

3. Click **+New User** at the top of the Users pane and then fill in the information to create a new user. Specify the `username@directoryname.onmicrosoft.com` as the login.
4. Set the role for the user. If this user will manage access to your Azure Sphere applications and devices, select the Global Administrator role.
5. Select Show Password to display the auto-generated password so that you can note it for future use, and then click Create. This is the account you'll use to log in to Azure Sphere.

## IMPORTANT

Record the auto-generated password and the user name. You will need them both to log in so that you can use your Azure Sphere device.

Home > default directory > Users - All users > User

User

default directory

\* Name ⓘ  
Example: 'Chris Green'

\* User name ⓘ  
Example: chris@contoso.com

---

Profile ⓘ >  
Not configured

---

Properties ⓘ >  
Default

---

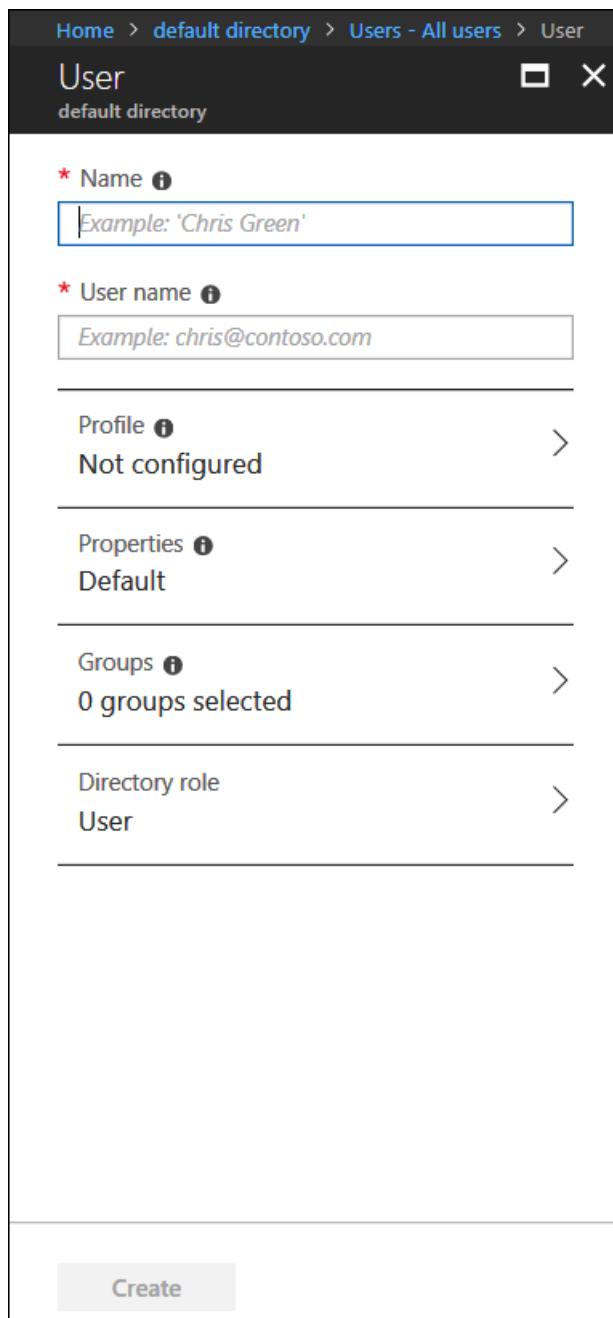
Groups ⓘ >  
0 groups selected

---

Directory role >  
User

---

Create



## Next steps

- [Claim your device](#)

# Claim your device

10/9/2019 • 2 minutes to read

Every device must be "claimed" by an Azure Sphere tenant. Claiming the device associates its unique, immutable device ID with your Azure Sphere tenant. The Azure Sphere Security Service uses the device ID to identify and authenticate the device.

We recommend that each company or organization create only one Azure Sphere tenant.

## IMPORTANT

Claiming is a one-time operation that you cannot undo even if the device is sold or transferred to another person or organization. A device can be claimed only once. Once claimed, the device is permanently associated with the Azure Sphere tenant.

Before you claim your device, [complete these steps](#) to ensure that you use the right work/school account to create and access your Azure Sphere tenant. Your device must be connected to your PC before you create the tenant, and you can only use the device to create a single tenant.

To claim your device:

1. Connect your device to your PC.
2. Open an Azure Sphere Developer Command Prompt, which is available in the **Start** menu under **Azure Sphere**.
3. Sign in to Azure Sphere, using [your work or school account](#):

```
azsphere login
```

If you have not yet logged in as this user, you will need to use the password that was auto-generated during sign-up and you will be prompted to change your password. You might also be prompted to [obtain admin consent](#) to use the Azure Sphere Utility.

4. If login succeeds, the command returns a list of the Azure Sphere tenants that are available for you.
  - If you are the first in your organization to sign in, you will not see any tenants, and you'll need to [create a tenant](#).
  - If multiple tenants exist, you'll need to select one. See [azsphere tenant select](#) for details.
  - If a single tenant is available, it will be selected by default, and you can proceed to the next step.

The Azure Sphere Security Service currently enables all members of the organization to manage all devices in an Azure Sphere tenant. If you want greater control over access to your Azure Sphere devices, you or your IT administrator can [limit access to your tenant](#).

5. Claim your device. After you claim your device into a tenant, you cannot move it to a different tenant.

```
azsphere device claim
```

You should see output like this:

```
Claiming device.  
Successfully claimed device ID '<device ID>' into tenant '<name>' with ID 'd343c263-4aa3-4558-adbb-d3fc34631800'.  
Command completed successfully in 00:00:05.5459143.
```

If **azsphere** returns an error, see [Troubleshoot Installation](#) for help.

## Next steps

- [Configure networking](#)

# Configure networking

10/9/2019 • 3 minutes to read

After you claim your Azure Sphere device, configure it for networking so that it can receive over-the-air (OTA) updates from the Azure Sphere Security Service and communicate with services such as an Azure IoT Hub.

Before you can configure networking, you must:

- [Install the SDK and set up the development board](#)
- [Update the OS](#) if you have an older MT3620 reference development board (RDB) that has never been used
- [Claim the device](#)

## IMPORTANT

This topic describes how to connect your Azure Sphere device to a Wi-Fi network. If your device supports a different networking mechanism and does not support Wi-Fi, connect it to the internet using that mechanism and proceed to [Receive device update](#).

## Set up Wi-Fi on your Azure Sphere device

Follow these steps to configure Wi-Fi on your Azure Sphere device:

1. Connect your Azure Sphere board to your PC over USB.
2. Open an Azure Sphere Developer Command Prompt.
3. Register the device's MAC address if your network environment requires it. Use the following command to get the MAC address, and then register it according to the procedure for your environment:

```
azsphere device wifi show-status
```

4. Add your Wi-Fi network to the device by using the **azsphere device wifi add** command as follows:

```
azsphere device wifi add --ssid <yourSSID> --psk <yourNetworkKey>
```

Replace <yourSSID> with the name of your network. Network SSIDs are case-sensitive. If the SSID is hidden, add --targeted-scan to try to connect to it anyway. You can use [azsphere device wifi scan](#) to get a list of available Wi-Fi networks.

Replace <yourNetworkKey> with your WPA/WPA2 key. Azure Sphere devices do not support WEP. To add an open network, omit --psk.

If your network SSID or key has embedded spaces, enclose the SSID or key in quotation marks. If the SSID or key includes a quotation mark, use a backslash to escape the quotation mark. Backslashes do not require escape if they are part of a value. For example:

```
azsphere device wifi add --ssid "New SSID" --psk "key \"value\" with quotes"
```

It typically takes several seconds for networking to be ready on the board, but might take longer, depending on your network environment.

#### NOTE

The Azure Sphere device checks for software updates each time it boots, when it initially connects to the internet, and at 24-hour intervals thereafter. If an Azure Sphere OS update is available, download and installation could take as much as 15 minutes and might cause the device to restart.

5. Use the **azsphere device wifi show-status** command to check the status of the connection. During update, the **azsphere device wifi show-status** command may temporarily show an unknown configuration state. The following example shows successful results for a secure WPA2 connection:

```
azsphere device wifi show-status

SSID : NETGEAR21
Configuration state : enabled
Connection state : connected
Security state : psk
Frequency : 2442
Mode : station
Key management : WPA2-PSK
WPA State : COMPLETED
IP Address : 192.168.1.15
MAC Address : 52:cf:ff:3a:76:1b
Command completed successfully in 00:00:01.3976308.
```

The **azsphere device wifi** command supports several additional options. Type **azsphere device wifi** for a complete list, or **azsphere device wifi option --help** for help on an individual option.

If you encounter Wi-Fi problems, first ensure that your Wi-Fi network uses 802.11b/g/n; Azure Sphere devices do not support 802.11a.

## Receive device update

When networking initially becomes available, the device checks for over-the-air (OTA) updates for the Azure Sphere operating system (OS) and the current application (if one exists). If updates are available, download should complete within 15-20 minutes.

To check on update status, type the following command in an Azure Sphere Developer Command Prompt:

```
azsphere device show-ota-status
```

#### NOTE

Azure Sphere OS update is a staged process over a period that may be less than 15 minutes but can be longer depending on the internet connectivity. Between stages, the board will appear functional while the next group of updates is downloaded, but during the update you should expect the board to be unresponsive for several minutes at a time.

## Next Steps

- [Subscribe to Azure Sphere notifications](#) so that you receive the latest information about OS updates

# Azure Sphere notifications

10/9/2019 • 2 minutes to read

Microsoft notifies customers about Azure Sphere service interruptions and planned outages on the [Azure Health Service Issues](#) page, which is available through the Azure Portal. You can set up alerts and subscribe to Azure Health Status Issues by using Azure Monitor. [Learn about alerts](#).

Notification of software updates, termination of support for older preview OS versions, and similar information is posted on the [Azure Updates](#) website. We also post technical information in the IoT blog on the [Internet of Things website](#).

## IMPORTANT

We strongly encourage you to subscribe to the Azure Updates RSS feed, so that you receive timely and essential information about Azure Sphere.

To subscribe to Azure Updates through the RSS feed:

1. Go to the [Azure Updates](#) website.
2. In the Products search box, enter Azure Sphere, and then select Azure Sphere from the search results.
3. Right-click the RSS feed button and select Copy link.
4. Use this link in any RSS reader to subscribe to the RSS feed.

If you use the Microsoft Outlook RSS reader, Azure Sphere notifications can be delivered to your email inbox. To subscribe to the RSS feed through Outlook, see [these instructions](#).

## Next Steps

Azure Sphere setup is now complete. Next, build and deploy a simple application or explore the use of IoT:

- [Build a high-level application](#)
- [Deploy an application over the air](#)
- Explore the [Azure Sphere samples](#) on GitHub

# Create an Azure Sphere tenant

3/20/2019 • 2 minutes to read

An Azure Sphere tenant isolates your organization's Azure Sphere devices from all other Azure Sphere devices and enables your organization to manage them. The Azure Sphere tenant is strictly used for Azure Sphere; it is not the same as an Azure AD tenant.

If your Azure Sphere login indicates that no Azure Sphere tenant exists, you can create one:

1. Connect an Azure Sphere device to your PC by USB.
2. Open an Azure Sphere Developer Command prompt and enter the following command:

```
azsphere tenant create --name <my-tenant>
```

Replace <my-tenant> with a name that others in your organization will recognize, such as "Contoso Ltd" or "Contoso Dishwasher Division." If the name includes spaces, enclose it in quotation marks.

You will be prompted to log in again. Be sure to log in with the account that you will use to manage your Azure Sphere devices.

This command creates a tenant but does not claim the device into the tenant. You can [claim your device](#) after you have created the tenant.

When you create a tenant, the Azure Sphere Security Service records the device ID of the attached device. Each device ID can be used to create only one Azure Sphere tenant.

If a tenant already exists, do not create another one unless your company or organization requires more than one tenant. We recommend that each company or organization create only one Azure Sphere tenant. If you are absolutely certain that you want to create another one, use the following command:

```
azsphere tenant create --force --name <my-tenant>
```

## NOTE

By default, everyone who can log in with a work or school account to your Azure Active Directory can access your Azure Sphere tenant and push new or modified applications to your Azure Sphere devices. To ensure greater security, you or your IT administrator can [limit access to your tenant](#) by setting enterprise application permissions for the Azure Sphere Utility.

# Limit access to your tenant

2/14/2019 • 2 minutes to read

By default, everyone who can log in to your Azure Active Directory (AAD) has access to your Azure Sphere tenant and can push new or modified applications to your Azure Sphere devices. To ensure greater security, you can limit access by setting enterprise application permissions for the Azure Sphere API.

You must be a Global Administrator or Application Administrator for the AAD to set enterprise application permissions. By default, the person who signs up for the Azure subscription has this role. [Learn more about AAD roles](#).

Follow these steps to limit access to specific users or groups:

1. Open the [Azure Portal](#) and sign in with your AAD credentials.
2. Ensure that you are a Global Administrator or Application Administrator for the directory. In the left panel, select Azure Active Directory, then choose Roles and Administrators in the center. Your role is displayed at the top of the rightmost panel:

The screenshot shows the 'default directory - Roles and administrators' page in the Azure Active Directory section of the Azure Portal. On the left, there's a sidebar with various service icons. The 'Azure Active Directory' icon is highlighted with a red box. In the center, under 'MANAGE', the 'Roles and administrators' link is also highlighted with a red box. On the right, a table lists various roles with their descriptions. At the top of the table, 'Your Role: Global administrator' is displayed in a box with a red border.

ROLE	DESCRIPTION
Application administrator	Can create and manage all aspects of app registrations and enterprise apps.
Application developer	Can create application registrations independent of the 'Users can register applications' setting.
Billing administrator	Can perform common billing related tasks like updating payment information.
Cloud application administrator	Can create and manage all aspects of app registrations and enterprise apps except App P...
Compliance administrator	Can read and manage compliance configuration and reports in Azure AD and Office 365.
Conditional access administrator	Can manage conditional access capabilities.
Customer LockBox access approver	Can approve Microsoft support requests to access customer organizational data.
Dynamics 365 administrator	Can manage all aspects of the Dynamics 365 product.

3. Select Enterprise Applications in the center panel, below Roles and Administrators.
4. In the Enterprise Applications screen, select Azure Sphere API.

The screenshot shows the 'Enterprise applications - All applications' page in the Azure Active Directory section of the Azure Portal. On the left, the 'Manage' section has 'All applications' selected, which is highlighted with a red box. In the main area, a table lists enterprise applications. The first entry, 'Azure Sphere API', is highlighted with a red box.

NAME	HOMEPAGE URL	OBJECT ID	APPLICATION ID
AS Azure Sphere API	<a href="https://www.microsoft.com/en-us/azure-sphere/">https://www.microsoft.com/en-us/azure-sphere/</a>	52e51a2f-59fc-46bf-86b2-9f870b2028f	7a663687-509b-4592-ab89-e20...
AS Azure Sphere Utility	<a href="https://www.microsoft.com/en-us/azure-sphere/">https://www.microsoft.com/en-us/azure-sphere/</a>	d1b6bbf8-452c-47d0-a7d6-84104e7dcf54	0b1c8f7e-28d2-4378-97e2-7d7d...

5. In Azure Sphere API Properties, set **Enabled for users to sign in?** and **User assignment required?** to Yes and select **Save**.

## Azure Sphere API - Properties

Enterprise Application

[Overview](#)
[Save](#) [Discard](#) [Delete](#)

Enabled for users to sign-in? [?](#)

Yes  No

Name [?](#)

Homepage URL [?](#)

Logo [?](#) 

Application ID [?](#)  [Download](#)

Object ID [?](#)  [Download](#)

User assignment required? [?](#)  Yes  No

Visible to users? [?](#)  Yes  No

6. Under Manage, select Users and Groups, and then select **Add** to add users.

In the left panel, select Users or Groups (if your Active Directory plan supports groups) to add individual users or groups to your Azure Sphere tenant. Then, in the pane to the right, select the users or groups to whom you want to grant access and choose **Select**.

The screenshot shows the 'Add Assignment' dialog box. On the left, under 'None Selected', there are three users: PO, PE, and TE. Each user has a small circular icon next to their name. On the right, a list of 'Selected members:' is shown, which is currently empty. At the bottom, there are two buttons: 'Assign' on the left and 'Select' on the right.

- Click **Assign** to add the selected members to the tenant. You should see a list of the users in the panel to the right.

The screenshot shows the 'Users' list view. The sidebar on the left shows 'Users and groups' is selected. The main area displays a table with three rows, each representing a user: PO, PE, and TE. The table columns are 'DISPLAY NAME', 'OBJECT TYPE', and 'ROLE ASSIGNED'. All three users are listed as 'User' type with 'Default Access' assigned.

DISPLAY NAME	OBJECT TYPE	ROLE ASSIGNED
PO	User	Default Access
PE	User	Default Access
TE	User	Default Access

# Obtain admin approval

8/30/2019 • 2 minutes to read

Some corporate networks have policies that require an administrator to grant permission to use applications in the Azure Active Directory (AAD). In such cases, you may see a dialog box similar to the following when you try to use the Azure Sphere utility:



.com

## Need admin approval

Azure Sphere Utility

Azure Sphere Utility needs permission to access resources in your organization that only an admin can grant. Please ask an admin to grant permission to this app before you can use it.

[Have an admin account? Sign in with that account](#)

[Return to the application without granting consent](#)

If you are not an admin for the AAD, you'll need to request that an administrator grant you permission to use the application.

The administrator should follow these steps:

1. Sign in [here](#) with an AAD administrator account.
2. In the dialog box that appears, check the box and click **Accept**.

This link gives consent for all users in the directory to use the Azure Sphere utility. If necessary, you can [limit access to Azure Sphere](#) to certain AAD users.

See the AAD documentation for more information about how to [configure user consent](#).

# Troubleshooting installation and setup

10/9/2019 • 7 minutes to read

Here are some troubleshooting steps for common installation and setup problems.

## Failure to connect to device

Probably the most common problem is a failure to connect to the Azure Sphere device. This problem can occur for many reasons and may trigger any of several error messages, depending on which tools or applications encounter it. The following error messages may indicate a failed connection:

- An error occurred. Please check your device is connected and your PC has been configured correctly, then retry.  
Could not connect to the device. Check if your device is connected to the PC. The device may be unresponsive if it is applying an Azure Sphere operating system update; Wait a few minutes and then retry. If this issue persists, try uninstalling and reinstalling the Azure Sphere SDK.
- An unexpected problem occurred. Please try again; if the issue persists, please refer to [aka.ms/azurespheresupport](https://aka.ms/azurespheresupport) for troubleshooting suggestions and support.
- Failed to retrieve device ID from attached device: 'Could not connect to the device; please ensure it is attached.'
- Failed to establish communication with device after recovery.

Failure to connect may occur for any of these reasons:

- The board is not connected by USB to the PC.
- The board is not powered.
- The TAP-Windows adapter is not installed or is not configured correctly.
- The Azure Sphere Device Communication Service has not yet started.

To resolve the problem, start with these basic steps:

1. Ensure that the device is connected by USB to the PC.
2. If the device is connected, press the Reset button on the device. Wait ten seconds or so for the device to restart, and then try the failed command again.
3. If the error recurs, unplug the device from the USB port, plug it in again, and wait for it to restart.
4. If the error recurs after restart, use the **View Network Connections** control panel to check that the adapter exists and is configured to use IP address 192.168.35.1. To access this control panel, press Start and type "ncpa.cpl". Right-click on the Azure Sphere device to view its properties. If Azure Sphere is not visible or does not have the correct IP address, reinstall the Azure Sphere SDK.

If none of these steps solves the problem, try the solutions that follow.

### Check the power to the board

If the MT3620 board appears "dead" or unresponsive, the problem could be a lack of power.

Make sure that the real-time clock (RTC) is powered, by either the main 3V3 power supply or a coin cell battery, as described in the [MT3620 development board user guide](#).

MT3620 development boards are shipped from the factory with a jumper header across pins 2 and 3 of J3, which powers the clock from the main power supply. Check that the header has not been dislodged or removed.

## Failure to create three COM ports

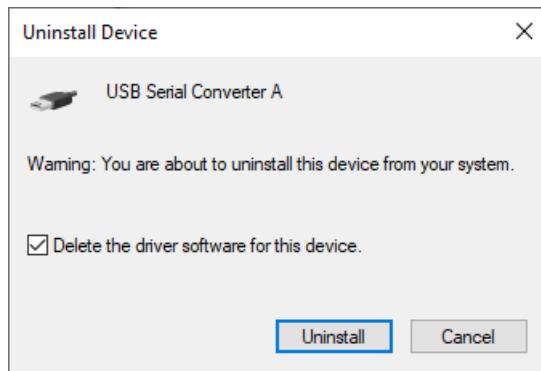
After you set up an MT3620 development board, you should see three COM ports in Device Manager. If you see only one or two, as in the following screenshot, you may have a problem with the FTDI driver:



If the FTDI driver is not correctly installed, the COM ports might appear in the wrong location, such as Other devices, or might not appear at all.

To resolve this problem:

1. Open **Device Manager** by clicking **Start** and typing "Device Manager."
2. Under Other devices, check for COM ports and delete any you find.
3. Under Universal Serial Bus controllers, select "USB Serial Converter A." Right-click the name, select **Uninstall Device**, and delete the driver if given the option:



Repeat this step for "USB Serial Converter B" through "USB Serial Converter D." As you uninstall the USB Serial Converters, you should see the USB Serial Ports also disappear from the Device Manager window.

4. Unplug your development board from your PC and plug it back in again. "MSFT MT3620 Std Interface" should appear with a triangle warning icon, which indicates no driver is available.
5. Right-click on one of the "MSFT MT3620 Std Interface" devices and select **Update driver**. Choose "Search automatically for updated driver software." Updating one should fix them all. You should now see three USB Serial Ports in the Ports section. If all three ports do not appear, repeat this step for each port.

#### Check the TAP-Windows adapter configuration

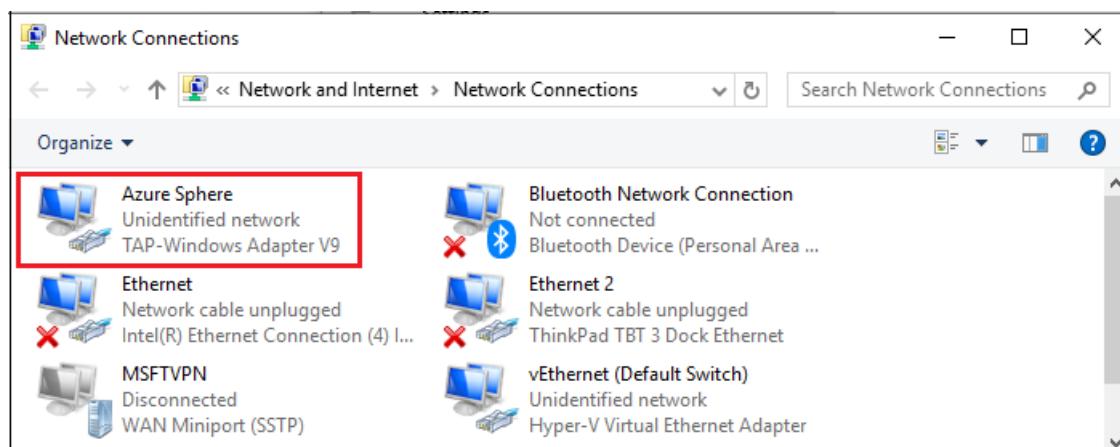
Azure Sphere tools communicate with attached development boards by using an IP network over USB. This requires the TAP-Windows adapter from OpenVPN Technologies. The Azure Sphere SDK installation procedure installs this adapter on your PC if it is not already present.

However, if a different version of the TAP-Windows adapter is already installed, or if the Azure Sphere device is not connected to the first instance of the TAP-Windows adapter, the Azure Sphere tools may fail to connect to your device.

To determine whether the problem is related to the TAP adapter, first find out how many TAP adapters are installed on your PC, and then modify the installation if necessary.

To determine how many TAP adapters are installed on your PC:

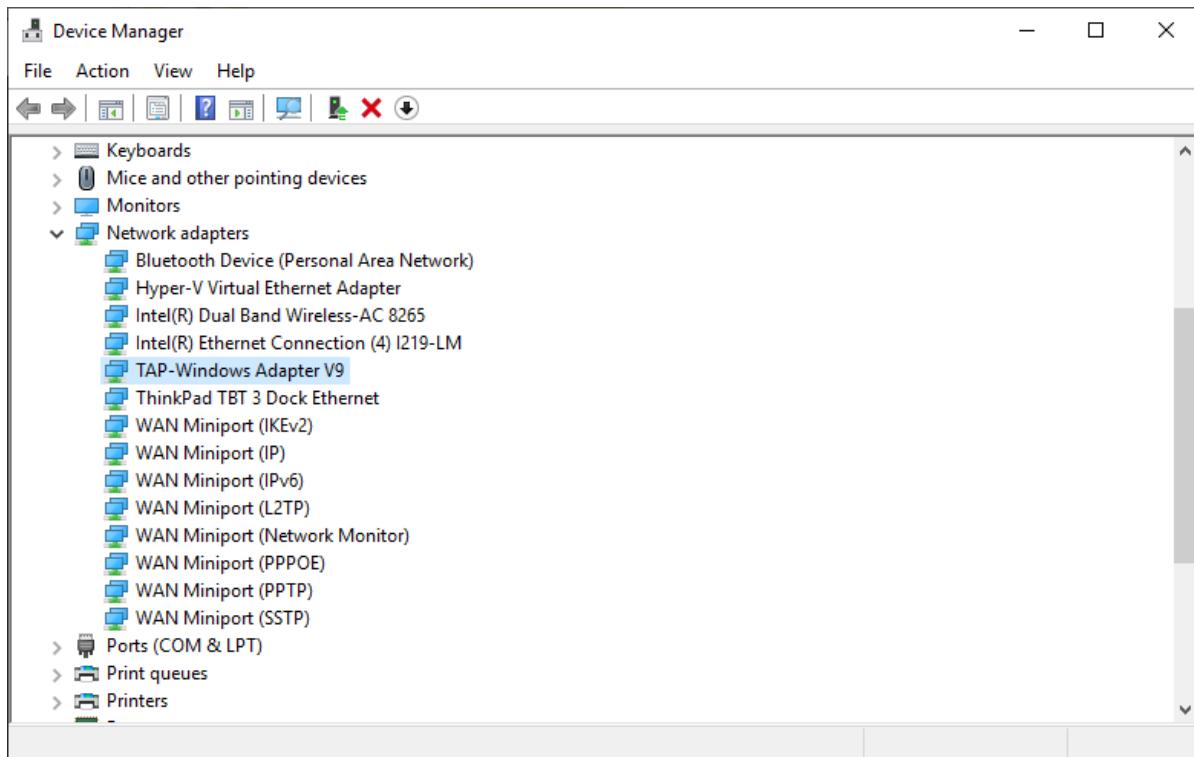
1. Open Control Panel.
2. In Windows Settings, select Network & Internet, and then select Change Adapter Options. You should see only one TAP adapter, as in the screenshot:



If you see more than one TAP adapter, or if you see only one TAP adapter but its name is not Azure Sphere, follow these steps to uninstall all TAP adapters and reinstall the SDK. If you see no TAP adapters, reinstall the SDK.

To uninstall the TAP adapters:

1. Click **Start** and type "Device Manager".
2. In Device Manager, open Network Adapters and select TAP-Windows adapter:



3. Right-click TAP-Windows adapter and select **Uninstall Device**. In the dialog box, check Delete the driver software for this device, then click **Uninstall**.
4. Open a command prompt as Administrator and run the following Powershell installer script:

```
powershell -ExecutionPolicy RemoteSigned -File "%ProgramData%\Microsoft\AzureSphere\TapDriverInstaller\TapDriverInstaller.ps1" Install
```

5. If the installation succeeds, restart the Azure Sphere Device Communication Service:

```
net stop AzureSphereDeviceCommunicationService
```

```
net start AzureSphereDeviceCommunicationService
```

6. Reinstall the Azure Sphere SDK Preview for Visual Studio.

## Device does not respond

One or more of the following errors from an **azsphere** command may indicate that the Azure Sphere Device Communication Service failed to start:

- warn: Device is not responding. Could not perform version check.
- Device is not responding. Cannot get device ID.  
error: Could not connect to the Azure Sphere Device Communication Service. If this issue persists, try
- uninstalling and reinstalling the Azure Sphere SDK.  
error: The device is not responding. The device may be unresponsive if it is applying an Azure Sphere operating system update; please retry in a few minutes.

The service may fail to start after Windows update or in cases where one of the internal JSON settings files has become corrupted.

### Failure after Windows Update

These errors can occur after you've updated Windows on your PC. Sometimes Windows Update uninstalls the FTDI drivers required for the communication service.

To resolve the problem:

1. Unplug the Azure Sphere device from USB and plug it in again. Upon replugging the device, the correct drivers should reinstall.
2. If unplugging and replugging the device fails to fix the problem, uninstall and reinstall the Azure Sphere SDK.

### JSON file

If you have not recently updated Windows, the cause of the error might be the `restore.json` file that is used for the service.

To resolve this problem:

1. Save a copy of the following file:  
`c:\windows\serviceprofiles\localservice\appdata\local\Azure Sphere Tools\restore.json`
2. Delete the file from its original location.
3. Stop and then restart the Azure Sphere Device Communication Service:

```
net stop AzureSphereDeviceCommunicationService
```

```
net start AzureSphereDeviceCommunicationService
```

## USB errors

If the following error occurs after you attach an MT3620 development board, you may have a USB error:

```
Windows has stopped this device because it has reported problems. (Code 43) A USB port reset request failed.
```

To resolve this error, try these steps:

1. Plug the device into a different USB port on the PC. If the error occurred when the device was plugged into a USB hub, plug it into a port on the PC instead.
2. If the problem recurs, plug the device into a powered USB hub. In some cases, USB ports do not provide adequate power for the board, so a powered hub may solve the problem.

## PC drops connection to Azure Sphere

If your PC can connect to the Azure Sphere device, but often drops the connection, you may have a conflict in the IP subnet.

Azure Sphere uses subnet 192.168.35.\*. If you have other software that uses the same subnet, either disable that software or limit the range of IP addresses that it uses.

Currently, you cannot change the range of IP addresses that Azure Sphere uses.

# Quickstarts for Azure Sphere

5/30/2019 • 2 minutes to read

After you [install Azure Sphere](#), complete these quickstarts to build and deploy a simple application.

The quickstarts guide you through:

- [Building your first high-level application](#) by using the Azure Sphere SDK Preview for Visual Studio
- [Building a real-time capable application](#) by using CMake with the Azure Sphere SDK Preview for Visual Studio
- [Deploying an application over the air](#)

# Quickstart: Build the Blink application

10/9/2019 • 4 minutes to read

This quickstart shows how to enable application development on an Azure Sphere device and how to build and debug a high-level application. It uses the Blink sample, which is part of the Azure Sphere SDK. The Blink sample blinks an LED, so that you can verify that the Azure Sphere device and tools are installed and set up correctly.

## Prerequisites

The steps in this quickstart assume that:

- Your Azure Sphere device is connected to your PC
- You have completed all the steps to [install Azure Sphere](#)

### IMPORTANT

These instructions assume that you are using hardware that follows the [MT3620 reference board design \(RBD\)](#), such as the Seeed MT3620 Development Kit. If you are using different Azure Sphere hardware, consult the manufacturer's documentation to find out whether the GPIO is exposed and how to access it. You might need to update the sample code and the "Gpio" field of the app\_manifest.json file to use a different GPIO.

## Prepare your device for development and debugging

Before you can build a sample application on your Azure Sphere device or develop new applications for it, you must enable development and debugging. By default, Azure Sphere devices are "locked"; that is, they do not allow applications under development to be loaded from a PC, and they do not allow debugging of applications.

Preparing the device for debugging removes this restriction.

The [azsphere device prep-debug](#) command configures the device to accept applications from a PC for debugging and loads the debugging server onto the device. It also assigns the device to a [device group](#) that does not allow over-the-air (OTA) application updates. During application development and debugging, you should leave the device in this group so that OTA application updates do not overwrite the application under development.

### To prep your device

1. Make sure that your Azure Sphere device is connected to your PC, and your PC is connected to the internet.
2. In an Azure Sphere Developer Command Prompt window, type the following command:

```
azsphere device prep-debug
```

You should see output similar to the following:

```
Getting device capability configuration for application development.
Downloading device capability configuration for device ID '<device ID>'.
Successfully downloaded device capability configuration.
Successfully wrote device capability configuration file 'C:\Users\user\AppData\Local\Temp\tmpD732.tmp'.
Setting device group ID 'a6df7013-c7c2-4764-8424-00cbacb431e5' for device with ID '<device ID>'.
Successfully disabled over-the-air updates.
Enabling application development capability on attached device.
Applying device capability configuration to device.
Successfully applied device capability configuration to device.
The device is rebooting.
Installing debugging server to device.
Deploying 'C:\Program Files (x86)\Microsoft Azure Sphere SDK\DebugTools\gdbserver.imagepackage' to the
attached device.
Image package 'C:\Program Files (x86)\Microsoft Azure Sphere SDK\DebugTools\gdbserver.imagepackage' has
been deployed to the attached device.
Application development capability enabled.
Successfully set up device '<device ID>' for application development, and disabled over-the-air
updates.
Command completed successfully in 00:00:38.3299276.
```

The device remains enabled for debugging and closed to OTA application updates until you explicitly change it. To disable debugging and allow application updates, use the [azsphere device prep-field](#) command.

## Build and run the Blink sample

1. If you are new to Visual Studio, consider the [Quickstart](#) or [Guided Tour](#) to learn about navigating and using it.
2. In Visual Studio, create a new project based on the Azure Sphere Blink template. Enter a name and location for the project and select **OK**. The project opens with main.c in the editor.
3. If you are using hardware on which GPIO 9 is not accessible, change the GPIO. For example, on the Seeed Mini Dev Board, change GPIO 9 to GPIO 7:
  - In app\_manifest.json, change `"Gpio": [ 9 ],` to `"Gpio": [ 7 ],`
  - In main.c, edit the call to **GPIO\_OpenAsOutput** to specify GPIO 7 instead of GPIO 9:

```
int fd = GPIO_OpenAsOutput(7, GPIO_OutputMode_PushPull, GPIO_Value_High);
```
4. In Visual Studio, select **View>Output** to display the Output window.
5. Ensure that your device is connected to your PC by USB. Then select **Remote GDB Debugger** from the menu bar or press **F5**.



6. If you are prompted to build the project, select **Yes**. Visual Studio compiles the application, creates an image package, *sideloads* it onto the board, and starts it in debug mode. *Sideload*ing means that the application is delivered directly from the PC over a wired connection, rather than delivered over the air (OTA).

### TIP

Note the path in the Build output, which indicates the location of the output image package on your PC. You'll use the image package later in the [Deployment Quickstart](#).

7. By default, the Output window shows output from **Device Output**. To see messages from the debugger, select **Debug** from the **Show output from:** dropdown menu. You can also inspect the program disassembly, registers, or memory through the **Debug>Windows** menu.

8. When you run the program, you should see an LED blink.

## Next steps

This quickstart showed how to prepare your Azure Sphere device for debugging and how to build and debug a high-level application.

Next:

- Learn how to [deploy the application over the air](#)
- [Create a real-time capable application](#)
- Explore [additional samples](#) for the high-level core
- Learn about [Azure Sphere applications](#)

# Quickstart: Build a real-time capable application

10/9/2019 • 4 minutes to read

This quickstart shows how to build a sample application for the real-time capable cores on an Azure Sphere device. It uses the HelloWorld\_RTApp\_MT3620\_Baremetal sample from the Azure Sphere samples repository on GitHub and sends output over the dedicated UART on the real-time core. Build, deploy, and debug this application to verify that the Azure Sphere device and tools are installed and set up correctly.

## IMPORTANT

These instructions assume you are using an MT3620 reference development board (RDB). If you are using different Azure Sphere hardware, consult the manufacturer's documentation to find out whether the UART is exposed and how to access it. You might need to [set up hardware to display output](#) differently, and update the sample code and the "Uarts" field of the `app_manifest.json` file to use a different UART.

## Prerequisites

The steps in this quickstart assume that:

- Your Azure Sphere device is connected to your PC
- You have completed all the steps to [install Azure Sphere](#)

## Hardware and software requirements

Currently, each real-time core supports a TX-only UART. The HelloWorld\_RTApp\_MT3620\_Baremetal sample uses this UART to send log output from the device. To read the output, this Quickstart requires a way to read and display the output.

Use a USB-to-serial adapter such as the [FTDI Friend](#), to connect the UART on the real-time core to a USB port on your PC. You will also need a terminal emulator such as Telnet or [PuTTY](#) to display the output.

## TIP

To enable Telnet, open Control Panel, click **Programs>Programs and Features>Turn Windows features on or off**. Scroll down, ensure that Telnet client is checked, and click OK. The Windows documentation provides detail about [telnet commands](#).

## Prepare your device for development and debugging

Before you can build a sample application on your Azure Sphere device or develop new applications for it, you must enable development and debugging. By default, Azure Sphere devices are "locked"; that is, they do not allow applications under development to be loaded from a PC, and they do not allow debugging of applications. Preparing the device for debugging removes this restriction and loads software required for debugging.

To debug on the real-time cores, use the **azsphere device prep-debug** command with the `--enablerterebugging` option. This command configures the device to accept applications from a PC for debugging, loads the debugging servers and required drivers for each type of core onto the device, and assigns the device to a [device group](#) that does not allow over-the-air (OTA) application updates. During application development and debugging, you should leave the device in this group so that OTA application updates do not overwrite the application under development.

## To prep your device

1. Make sure that your Azure Sphere device is connected to your PC, and your PC is connected to the internet.
2. Right-click the Azure Sphere Developer Command Prompt shortcut and select **More>Run as administrator**. When the window opens, issue the following command:

```
azsphere dev prep-debug --enablertcoredebugging
```

The **azsphere** command must be run as administrator when you enable real-time core debugging because it installs USB drivers for the debugger.

3. Close the window after the command completes because administrator privilege is no longer required. As a best practice, you should always use the lowest privilege that can accomplish a task.

## Set up hardware to display output

To set up the hardware to display output from the HelloWorld sample, follow these steps. [Interface headers](#) may help you to determine the pin locations.

1. Connect GND on the USB-to-serial adapter to Header 3, pin 2 (GND) on the MT3620 RDB.
2. Connect RX on the USB-to-serial adapter to Header 3, pin 6 (IOM4-0 TX) on the MT3620 RDB.
3. Attach the USB-to-serial adapter to your PC.
4. Determine which COM port the adapter uses on the PC. Start Device Manager. Select **View>Devices by container**, and look for FT232R USB UART.
5. On the PC, start the terminal emulator and open a 115200-8-N-1 terminal to the COM port that the adapter uses.

## Build and run the HelloWorld sample

1. Open an Azure Sphere Developer Command prompt and clone the Azure Sphere Samples repository from GitHub:

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

2. Start Visual Studio. On the **File** menu, select **Open>CMake...**, navigate to the HelloWorld folder, and open the folder to find the HelloWorld\_RTApp\_MT3620\_Baremetal sample. If you are not using an MT3620 RDB, update the app\_manifest.json file and the sample code to specify the correct UART, for example ISU0.
3. If CMake generation does not start automatically, select the CMakeLists.txt file.
4. In the Visual Studio Output window, the CMake output should show the messages **CMake generation started.** and **CMake generation finished.**
5. On the CMake menu (if present), click Build All. If the menu is not present, open Solution Explorer, right-click the CMakeLists.txt file, and select Build. The output location of the HelloWorld\_RTApp\_MT3620\_Baremetal application appears in the Output window.
6. On the Select Startup Item menu, select GDB Debugger (RTCore).
7. Press F5 to deploy the application.
8. The connected terminal emulator should display output from the HelloWorld\_RTApp\_MT3620\_Baremetal program. The program sends the following words at one-second intervals:

Tick

Tock

9. Use the debugging options in Visual Studio to set breakpoints, inspect variables, and try other debugging tasks.

## Next steps

- Explore [additional samples](#) for the real-time capable cores
- Learn about [Azure Sphere applications](#)

# Quickstart: Deploy an application over the air

10/9/2019 • 6 minutes to read

This quickstart shows how to create your first over-the-air (OTA) application deployment. OTA deployment delivers an application through a [feed](#) to the devices in a [device group](#) that match the target [stock-keeping unit \(SKU\)](#) for the feed.

## Prerequisites

The steps in this quickstart assume that:

- Your Azure Sphere device is connected to your PC and to the internet
- You have completed all the steps to [install Azure Sphere](#)
- You have completed [Quickstart: Build the Blink application](#) and retained the image package for the application

## Prepare your device for OTA deployment

Before you test the OTA deployment process, your Azure Sphere device must be ready to accept OTA application updates. In effect, you "lock" the device and enable OTA updates, which is how it will operate at a customer site.

The **azsphere device prep-field** command is the simplest way to do this. This command:

- Disables the ability for Visual Studio to load applications onto the device, so that only OTA applications can be loaded
- Assigns a new [product SKU](#) to the device
- Assigns the device to a new [device group](#) that enables OTA application updates

The **azsphere device prep-field** command works on the device that is connected to your PC. It has the following form:

```
azsphere device prep-field --newdevicegroupname <unique-dg-name> --newsuname <unique-sku-name>
```

The `--newdevicegroupname` flag specifies a name for the new device group that the command creates. Learn about device groups [here](#). All device groups created by this command support automatic OTA application updates. Supply a descriptive name that is unique among the device group names in your Azure Sphere tenant.

The `--newsuname` flag specifies a name for the new product SKU that the command creates. A [product SKU](#) identifies a model of the connected device that contains an Azure Sphere chip. Each SKU in an Azure Sphere tenant must have a unique name.

The Azure Sphere Security Service uses the device group and the SKU to determine whether to [update the application](#) on a device.

### TIP

Enclose any strings that include spaces in double quotation marks, like "this string".

For example, the following command creates a new device group named ota-test-3 that enables OTA application updates and assigns the attached device to it. It also creates a new product SKU named test-sku-3 and assigns that SKU to the device.

```
azsphere device prep-field --newdevicegroupname ota-test-3 --newskuusername test-sku-3
```

```
Removing applications from device.  
Component '68c95ce8-e06c-4713-95f2-00cc1861a996' deleted or was not present beforehand.  
Removing debugging server from device.  
Component '8548b129-b16f-4f84-8dbe-d2c847862e78' deleted or was not present beforehand.  
Successfully removed applications from device.  
Locking device.  
Downloading device capability configuration for device ID <deviceID>.  
Successfully downloaded device capability configuration.  
Applying device capability configuration to device.  
Successfully applied device capability configuration to device.  
The device is rebooting.  
Successfully locked device.  
Creating a new device group with name 'ota-test-3'.  
Setting device group ID 'fee1a5fe-59a3-44fc-8fc1-ab6848e09490' for device with ID <deviceID>.  
Creating a new SKU with name 'test-sku-3'.  
Setting product SKU to 'ae7927d0-1cbd-4e87-995a-495964e7facf' for device with ID <deviceID>.  
Successfully set up device <deviceID> for OTA loading.  
Command completed successfully in 00:00:15.5644027.
```

You aren't required to create a new device group and SKU every time you prepare a device for field use. Typically, you would create one SKU for each product model, and one device group for each collection of devices that you want to update together. For additional details, see [prep-field](#).

## Link the device to a feed

The next step in deployment is to link your device to a [feed](#) that delivers the high-level Blink sample that you build from the template. You must supply:

- The ID of Azure Sphere OS feed on which the application depends
- The path to the image package file that Visual Studio created for the Blink application
- A name for the feed that will deliver the application

### To link to a feed

1. Get the feed ID for the Retail Azure Sphere feed, which delivers the Azure Sphere OS. The feeds that you see might differ from those in the example.

```
azsphere feed list
```

```
Listing all feeds.  
Retrieved feeds:  
--> [3369f0e1-dedf-49ec-a602-2aa98669fd61] 'Retail Azure Sphere OS'  
--> [82bacf85-990d-4023-91c5-c6694a9fa5b4] 'Retail Evaluation Azure Sphere OS'  
Command completed successfully in 00:00:03.0017019.
```

Copy the ID for the Retail Azure Sphere OS feed to use in the next step.

2. Issue the **azsphere device link-feed** command to create a feed and associate it with the Blink image package that you created in [Quickstart: Build the Blink application](#).

```
azsphere device link-feed --dependentfeedid 3369f0e1-dedf-49ec-a602-2aa98669fd61 --imagepath <path-to-image> --newfeedname <unique-name>
```

The --dependentfeedid flag supplies the ID of the Retail feed.

The `--imagepath` flag provides the path to the image package file for the Blink application. As noted in [Quickstart: Build the Blink application](#), the full path to the image file is displayed in the Visual Studio Build Output window. The **azsphere device link-feed** command uploads the image package file to the Azure Sphere Security Service and creates an image set with a unique name.

The `--newfeedname` flag provides a name for the feed that the command creates. Feed names must be unique in an Azure Sphere tenant, so specify a name that distinguishes this feed from any others.

For example:

```
azsphere device link-feed --dependentfeedid 3369f0e1-dedf-49ec-a602-2aa98669fd61 --imagepath  
"C:\Users\user\Documents\Visual Studio  
2017\Projects\AzureSphereBlink3\AzureSphereBlink3\bin\ARM\Debug\AzureSphereBlink3.imagepackage" --  
newfeedname blink-feed-test-3
```

```
Getting the details for device with ID <deviceID>.  
Uploading image from file 'C:\Users\user\Documents\Visual Studio  
2017\Projects\AzureSphereBlink3\AzureSphereBlink3\bin\ARM\Debug\AzureSphereBlink3.imagepackage':  
--> Image ID: 5fae38c5-48ee-4934-96b0-445b4f3e68fa  
--> Component ID: 68c95ce8-e06c-4713-95f2-00cc1861a996  
--> Component name: 'AzureSphereBlink3'  
Removing temporary state for uploaded image.  
Create a new feed with name 'blink-feed-test-3'.  
Adding feed with ID '21b69146-2d6d-4aec-b626-e70b6a19dc2c' to device group with ID 'fee1a5fe-59a3-44fc-  
8fc1-ab6848e09490'.  
Creating new image set with name 'ImageSet-AzureSphereBlink3-2019.05.07-18.30.00-07:00' for images with  
these IDs: 5fae38c5-48ee-4934-96b0-445b4f3e68fa.  
Adding image set with ID '607a5a24-c648-40c5-b496-6c5338a263fd' to feed with ID '21b69146-2d6d-4aec-  
b626-e70b6a19dc2c'.  
Successfully linked device <deviceID> to feed with ID '21b69146-2d6d-4aec-b626-e70b6a19dc2c'.  
Command completed successfully in 00:00:33.3283634.
```

This command creates a feed that is linked to the device group to which the attached device belongs—that is, the ota-test-2 group created [earlier in this quickstart](#). It will deliver the AzureSphereBlink3 application to all Azure Sphere devices in the group whose product SKU is test-sku-3.

If you see an error like the following, someone else who uses your Azure Sphere tenant has already uploaded a component with the same name.

```
error: A component with the same name ('Mt3620Blink1') already exists. Change the name in the  
application manifest and create a new image package. Trace ID: 38749b6a-c64e-4adb-80f4-4dcac8c5be77  
error: Command failed in 00:00:05.1624969.
```

Change the name of the component in the `app_manifest.json` file, rebuild the application to create a new image package, and issue the **azsphere device link-feed** command again.

## Trigger the deployment

The previous steps set up all the required deployment elements. To trigger the download immediately, press the **Reset** button on the Azure Sphere device. The application should download and start within several minutes, and you should see the LED start to blink.

To verify that the application was installed on your device:

```
azsphere device image list-installed
```

## Enable debugging

As you continue to develop and test applications, you will probably want to use Visual Studio to load them until you're ready to deploy them more broadly. To reverse the **device prep-field** command and enable the device for

development and debugging, use **device prep-debug**, as you did when you built the application:

```
azsphere device prep-debug
```

This command:

- Moves the device to a device group that disables OTA application updates
- Enables the device capability to accept applications from Visual Studio for debugging

## Next steps

- Learn about developing applications for Azure Sphere by checking out the [Azure Sphere samples](#) on GitHub
- Understand Azure Sphere [deployment basics](#)
- [Update the deployment](#)

# Connect Azure Sphere to Wi-Fi

10/9/2019 • 2 minutes to read

Azure Sphere devices rely on network connectivity to receive over-the-air OS and application updates. During development, it's easy to [configure Wi-Fi](#) for a device that's connected to your PC. When you incorporate Azure Sphere into a manufactured product, however, your customers must be able to set up Wi-Fi at their location.

You might accomplish this by providing a physical control panel through which the customer can configure their own Wi-Fi connection, or you might provide a mobile app to connect to the Azure Sphere device and configure Wi-Fi connectivity, via an additional Bluetooth Low Energy (BLE) chip. In either case, your Azure Sphere app will need to use the Azure Sphere Wi-Fi configuration API (`wificonfig.h`) to find available networks, then accept the user's network selection and Wi-Fi credentials.

## BLE-based Wi-Fi Setup - reference solution

The [BLE-based Wi-Fi setup and device control](#) reference solution demonstrates how to connect Azure Sphere over UART to a Nordic nRF52 BLE Development Kit. It also includes a sample Windows companion app that uses BLE to view and modify the Wi-Fi settings of the Azure Sphere device, and control attached device behavior.

## Enable targeted scanning

Azure Sphere supports targeted scanning, which allows devices to connect to Wi-Fi networks that don't broadcast their SSID, or are located in a crowded wireless network environment.

### IMPORTANT

Targeted scanning causes the device to transmit probe requests that may reveal the SSID of the network to other devices. This should only be used in controlled environments, or on networks where this is an acceptable risk.

You can enable targeted scanning through the CLI or an Applibs API. To enable targeted scanning through the CLI, run the `azsphere device wifi add` command with the `--targeted-scan` parameter. An application can enable targeted scanning by calling the [WifiConfig\\_SetTargetedScanEnabled](#) function with the `enabled` parameter set to *true*.

# Connect Azure Sphere to Ethernet

10/9/2019 • 5 minutes to read

An Azure Sphere device can communicate on a 10-Mbps Ethernet network through standard TCP or UDP networking. This topic describes how to configure an Azure Sphere device to use an external Ethernet controller by deploying a board configuration image to your device and then enabling the Ethernet interface. You can connect an Ethernet-enabled device to a public (internet-connected) network and communicate with [Azure IoT](#) or your own cloud services, or you can connect it to a private network and [use network services](#). In addition to communicating through Ethernet, you can [connect Azure Sphere to Wi-Fi](#).

**Caution**

Azure Sphere doesn't have a low-level filter for broadcast messages, so a quiet network is required if your device is connected through Ethernet. If the device is connected to a noisy network through Ethernet, it can lead to poor performance on the device.

## Hardware instructions

[Connecting Ethernet adapters to the MT3620 development board](#) describes how to connect an Ethernet adapter to the MT3620. The adapter referenced in those instructions uses the board configuration described here.

## Board configuration

Use of Ethernet requires a *board configuration image* in addition to the application image. The board configuration image contains information that the Azure Sphere Security Monitor requires to add support for Ethernet to the Azure Sphere OS. Microsoft supplies a board configuration for the Microchip ENC286J60 Ethernet chip, which is attached via SPI to ISU0 with interrupts on GPIO5. For development, we recommend the [Olimex ENC28J60-H module](#).

To use Ethernet on Azure Sphere, you package a board configuration image for the ENC28J60 and deploy this image package in addition to your application image package.

### Create a board configuration image package

In an Azure Sphere Developer Command Prompt, use **azsphere image package-board-config** to create a board configuration image package for the ENC28J60:

```
azsphere image package-board-config --preset lan-enc28j60-isu0-int5 --output enc28j60-isu0-int5.imagepackage
```

The --preset flag identifies the Microsoft-supplied board configuration to package, and the --output flag specifies a name for the package.

### Sideload a board configuration image package

You can sideload a board configuration image package for development and debugging, or you can deploy such a package over the air (OTA) along with your Azure Sphere application for field use.

To use a board configuration image package during development and debugging:

1. Prepare the device for development and debugging:

```
azsphere device prep-debug
```

2. Delete any existing applications from the device, using the **azsphere device sideload delete** command.

It's important to delete existing applications before you load the board configuration image package to avoid resource conflicts between existing applications and the board configuration.

3. Sideload the board configuration image package. For example, the following command sideloads the ENC28J60 Ethernet board configuration image package:

```
azsphere device sideload deploy --imagepackage enc28j60-isu0-int5.imagepackage
```

4. Sideload the application, either by using Visual Studio or by using the **azsphere device sideload deploy** command.

### OTA deploy a board configuration image package

OTA deployment of the board configuration image requires the 18.11.1 or newer SDK. To deploy a board configuration image package OTA:

1. Prepare the device for field use:

```
azsphere device prep-field --devicegroupid <devicegroup-GUID> --skuid <sku-GUID>
```

Replace <devicegroup-GUID> and <sku-GUID> with the device group and product SKU IDs, respectively, for the devices that should receive the OTA deployment. To create a new device group and a new product SKU, use the --newdevicegroupname and --newskuuname parameters instead, with appropriate names for the device group and product SKU.

2. Deploy the board configuration image along with the application image in a single over-the-air deployment. Use the **azsphere device link-feed** command to create a new feed that contains both images. For example:

```
azsphere device link-feed --newfeedname MyEthernetAppFeed --dependentfeedid 3369f0e1-dedf-49ec-a602-2aa98669fd61 --imagepath=enc28j60-isu0-int5.imagepackage,EthernetSample.imagepackage
```

This command creates a new feed named MyEthernetAppFeed, which depends on the Retail Azure Sphere software feed, and supplies software to devices in the same device group as the attached device. The feed delivers two image packages: the enc28j60-isu0-int5 board configuration image package and the EthernetSample application image package.

### Remove a sideloaded board configuration

If you sideload a board configuration during development, you might later need to remove that configuration so that other applications can use the resources that the board reserves. For example, the lan-enc28j60-isu0-int5 board configuration reserves ISU0 and GPIO 5. If you try to run an application that uses these resources while the board configuration is loaded on the Azure Sphere device, pin conflict errors will occur.

To remove a board configuration, follow these steps:

1. List the images installed on the device:

```
azsphere device image list-installed
```

2. Find the component ID for the board configuration in the list:

```
--> lan-enc28j60-is  
--> Image type: Board configuration  
--> Component ID: 75a3dbfe-3fd2-4776-b7fe-c4c91de902c6  
--> Image ID: a726b919-bdbe-4cf4-8360-2e37af9407e1
```

3. Delete the board configuration image package by specifying its component ID:

```
azsphere device sideload delete --componentid 75a3dbfe-3fd2-4776-b7fe-c4c91de902c6
```

4. Restart the device by either pressing the Reset button or issuing the **azsphere device restart** command.

### Remove a board configuration OTA

To remove a board configuration OTA, you must create a new feed that only delivers the application image package

and doesn't include the board config image package, and then associate your device with the feed.

**Caution**

Don't remove a board configuration OTA if your device relies on Ethernet for internet connectivity. This will take the device offline and prevent internet connectivity from being restored OTA. To restore connectivity you'll need physical access to the device to sideload the board config; or to setup alternative internet access, such as Wi-Fi.

## Enable the Ethernet interface

To use Ethernet, your high-level application must enable the interface by calling the [Networking\\_SetInterfaceState](#) function, which is part of the network configuration API. Use of network configuration functions requires the **NetworkConfig** capability in the [application manifest](#).

For example, the following enables the "eth0" interface:

```
err = Networking_SetInterfaceState("eth0", true);
```

All interfaces use dynamic IP addresses by default. See [use network services](#) for details about the types of services an Azure Sphere application can use.

## Samples

- The [Private Network Services](#) sample demonstrates how to connect Azure Sphere to a private network and use several network services.
- The [AzureIoT](#) sample demonstrates how to use the Azure IoT SDK C APIs in an Azure Sphere application to communicate with Azure IoT Central or Azure IoT Hub.
- The [HTTPS](#) samples demonstrate how to use the cURL APIs with Azure Sphere over a secure HTTPS connection.

# Use network services

10/9/2019 • 3 minutes to read

Azure Sphere can run a static IP address, [dynamic host configuration protocol \(DHCP\) server](#), and a [simple network time protocol \(SNTP\) server](#) for a network interface. The DHCP server enables Azure Sphere applications to configure network parameters for an external device on the network. The external device can use the SNTP server to synchronize its time with Azure Sphere.

## Network configuration

You can configure an Ethernet and a Wi-Fi network interface to run simultaneously on an Azure Sphere device. Ethernet and Wi-Fi network interfaces can be connected to public (internet-connected) or private networks. At least one interface must be connected to a public network. Only one Ethernet interface can be configured at a time.

### Private and public network interfaces

If you use both public and private network interfaces, such as private Ethernet with public Wi-Fi, the Azure Sphere device will not act as a router. It won't automatically pass packets received on the Ethernet network to the Wi-Fi network, or vice versa. Your application must implement all logic that sends and receives information on both networks.

### Dual public network interfaces

If you use two network interfaces that have dynamic IP addressing enabled, the OS attempts to use the first interface that connects to the network when it selects DNS server addresses for host name resolution. If an interface disconnects from the network, the DNS server addresses from the other connected interface are automatically used.

## Static IP address

You can configure a static IP address on a private network interface, but this is not supported on a public network interface. If a static IP address is configured for Ethernet, a DNS resolver can't be configured for the interface.

To set up a static IP address configuration, your application must use the [applibs networking API](#), and the [application manifest](#) must enable the **NetworkConfig** capability.

## DHCP server

An external client device that is connected to Azure Sphere through an Ethernet interface must be assigned an IP address and other network parameters so that it can communicate with a server application on the Azure Sphere device. However, some external devices do not support a way to configure these parameters. Azure Sphere supports a DHCP server through which an application can provide this configuration. The application must enable the **DhcpService** capability in its [application manifest](#).

The Azure Sphere application calls [Networking\\_DhcpServerConfig\\_Init](#) to configure the server to provide an IP address, subnet mask, gateway address, lease duration, and up to three NTP server addresses to a client device. Only one IP address can be configured in the current release. It then calls [Networking\\_DhcpServer\\_Start](#) to start the server on a particular network interface. After the DHCP server starts, the client device can send out broadcast DHCP messages to discover and request IP addresses from the DHCP server on the specified subnet.

## SNTP server

The SNTP server enables client devices to synchronize their system time with that of the Azure Sphere device. To use the server, the Azure Sphere application must enable the **SntpService** capability in its [application manifest](#).

To start the server, the Azure Sphere application calls [Networking\\_SntpServer\\_Start](#) and specifies the network interface on which the server will run. The client device and the Azure Sphere device must be in the same local subnet of the network on which the server is running. The Azure Sphere device must be connected to at least one public network, so that it can get the current time from a public network time protocol (NTP) server. The SNTP server does not respond to queries until it has the current time.

**NOTE**

Although an application can set the system time directly, this is not recommended because the time does not persist when the device loses power. [Manage system time and the RTC on Azure Sphere](#) has more information.

## Listening ports

If the Azure Sphere application listens for incoming TCP or UDP connections, the [application manifest](#) must specify the ports that the application uses. Incoming UDP connections are allowed only on the Ethernet network, and not on the Wi-Fi network.

## Samples

- The [Private Network Services](#) sample demonstrates how to connect Azure Sphere to a private network and use several network services.
- The [DNS Service Discovery](#) sample demonstrates how to query and process responses from a DNS server.

# Perform service discovery

7/5/2019 • 3 minutes to read

## BETA feature

High-level applications on Azure Sphere can perform service discovery by using DNS service discovery ([DNS-SD](#)). Applications can use service discovery to find network services and perform host name resolution so they can interact with the service through the Azure Sphere firewall. Multicast DNS ([mDNS](#)) can also be used to perform peer-to-peer discovery on a local network, which is especially useful when the IP addresses and host names of the destination endpoint aren't known at design time.

Applications use DNS-SD queries to retrieve DNS records from non-local DNS servers or over a multicast link. If the name being queried is under the **.local** top-level domain (TLD), the query is multicast on the local network through all enabled network interfaces; otherwise, unicast service discovery is performed. The [Service discovery sample](#) demonstrates how to perform service discovery on Azure Sphere.

### NOTE

The Azure Sphere firewall prevents applications from communicating with unauthorized services. However, allowing outbound connections to **.local** TLDs in the application manifest may increase the security risk to a device by allowing an application to connect with unauthorized services that are advertised on the local network. Applications should only allow outbound connections to **.local** TLDs in secured environments that prevent unauthorized parties from advertising services. To provide additional protection in this scenario, Azure Sphere requires that services that are discovered on the local network also reside on the local subnet.

## Include header files

Applications that perform service discovery must include the resolv header file:

```
#include <resolv.h>
```

## Allow a service connection

Before you perform a DNS-SD query, you must add the service to the AllowedConnections capability of the [Application manifest](#). The Azure Sphere firewall will then allow the application to connect to the discovered service instances using their associated host names and IP addresses. If a **.local** TLD service is specified, the firewall will only allow connections to discovered resources on the local subnet.

The following types of service names are supported in the AllowedConnections capability:

- Local DNS service name, such as "`_sample._tcp.local`"
- Non-local DNS service name, such as "`_sampleinstance._tcp.dns-sd.org`"
- Local service instance name, such as "`_sampleinstance._tcp.hostname.local`"
- Domain name, such as "`samplehost.contoso.com`"
- IP address

Here's an excerpt from an application manifest that includes a non-local service name.

```
"AllowedConnections": [ "_http._tcp.dns-sd.org" ]
```

## Perform a DNS-SD query

To perform a DNS-SD query, you need to request several types of [DNS records](#):

- **PTR** records that enumerate instances of a DNS service.
- **SRV** and **TXT** records that contain details of the service instances, such as host name and port.
- **A** records that contain the IP addresses of the retrieved host names.

Before you send the query, you need to create and initialize it, and then add a query message that requests the DNS record. You can create and initialize a DNS-SD query by calling the [POSIX function res\\_init\(\)](#). You can create a message for the query by calling the POSIX function [res\\_mkquery\(\)](#).

### Send a unicast DNS query

When performing unicast service discovery, you can send the DNS-SD query and retrieve the response by calling the POSIX function [res\\_send\(\)](#).

### Send the query over a multicast link

To send a DNS-SD query over a multicast link, the application must open a socket and send the request over the socket to the loopback IP address 127.0.0.1 (destination port 53). After the request is sent, multiple responses may be returned. The application should wait and listen for several seconds to collect all responses. This is demonstrated in the [Service discovery sample](#).

#### IMPORTANT

This loopback IP address is a Beta feature that will be deprecated and then replaced in future releases. This will be a breaking change for applications that rely on the address.

## Allowed connections for hosts with multiple IP addresses

The Azure Sphere firewall only allows connections to one IP address per host name. If a host has multiple IP addresses, the Azure Sphere firewall only allows connections to one of the addresses. An application can use [curl](#) to make HTTPS requests to a host that has multiple IP addresses; curl will try to connect to each IP address until the allowed address is found. However, this may cause a delay while the application finds the allowed address.

# Azure Sphere OS networking requirements

10/9/2019 • 2 minutes to read

The Azure Sphere OS and services communicate with devices, Azure IoT Hub, and other services using various endpoints, ports, and protocols. Some are required only by certain features and others are expected only on the local network. This topic lists the internet and public endpoints with which Azure Sphere devices must communicate for basic operation.

Azure Sphere tools use the 192.168.35.*n* subnet for a serial line IP connection to the device over the Service UART. Currently, you cannot change this.

Protocol	Port	URLs or IP Addresses	Purpose
MQTT over TCP	8883	global.azure-devices-provisioning.net	Device provisioning and communication with IoT Hub
MQTT over TCP	443 (WebSocket)	global.azure-devices-provisioning.net	Device provisioning and communication with IoT Hub
HTTP over TCP	80	<a href="http://www.msftconnecttest.com">www.msftconnecttest.com</a> , prod.update.sphere.azure.net	Internet connection checks, certificate file downloads, and similar tasks
HTTPS over TCP	443	prod.device.core.sphere.azure.net, prod.deviceauth.sphere.azure.net, prod.releases.sphere.azure.net, prod.core.sphere.azure.net, eastus-prod-azuresphere.azure-devices.net, prodmsimg.blob.core.windows.net, prodptimg.blob.core.windows.net	Communication with web services and Azure Sphere Security service
UDP	53		Communication with domain name servers (DNS)
UDP	123	prod.time.sphere.azure.net, time.sphere.azure.net	Communication with NTP server
UDP	124		Inbound communication with NTP client

High-level applications may also use additional networking resources. In particular, applications that use an Azure IoT Hub require ports 8883 and/or 443 to communicate with their hub at the domain name(s) created during Azure IoT setup. The Azure IoT Hub documentation lists other [Azure IoT Hub port and protocol requirements](#).

# Overview of Azure Sphere applications

7/2/2019 • 5 minutes to read

Azure Sphere devices can run two types of applications:

- *High-level applications* run containerized on the Azure Sphere OS
- *Real-time capable applications (RTApps)* run on bare metal or with a real-time operating system (RTOS) on the real-time cores

A high-level application is required for every Azure Sphere device; RTApps are optional.

## High-level applications

Every Azure Sphere device has a high-level application, which runs on the Azure Sphere OS and uses the application libraries. A high-level application can:

- Configure and interact with Azure Sphere peripherals, such as the general-purpose input/output (GPIO) pins, universal asynchronous receiver/transmitters (UARTs), and other interfaces
- Communicate with RTApps
- Communicate with the internet and cloud-based services
- Broker trust relationships with other devices and services via certificate-based authentication

A high-level application runs in a container in Normal World user mode, as described in [What is Azure Sphere?](#). The application container supports a subset of the POSIX environment and a set of application libraries (Applibs) that are specific to the Azure Sphere OS. The libraries and functions that are available to high-level applications are restricted to ensure that the platform remains secure and can be easily updated. Applications can access only the libraries and run-time services that Microsoft provides; neither direct file I/O nor shell access are available, among other constraints. The [Development environment](#) topic describes the base API set and introduces the Azure Sphere application libraries that support device-specific features.

High-level applications are expected to run continuously and are automatically restarted if they stop or fail.

[Overview of high-level application development](#) provides more information about features.

## Real-time capable applications

An Azure Sphere device may also have one or more real-time capable applications in addition to its high-level application. An RTApp can:

- Configure and interact with peripherals integrated into the Azure Sphere MCU, such as the GPIO pins and UARTs
- Communicate with high-level applications

RTApps can run either on bare metal or with a real-time operating system (RTOS). The [Azure Sphere samples repo](#) on GitHub includes bare-metal samples that you can start with. If you choose instead to use an RTOS, you will need to port the RTOS to Azure Sphere.

Each RTApp runs isolated on a particular I/O core and can communicate only with a high-level application; it cannot use the internet, the Azure Sphere applibs, or other features of the Azure Sphere OS.

[Overview of RTApp development](#) provides more information about the features and development process for

## Features common to all applications

Despite the significant differences between high-level and RTApps, all Azure Sphere applications have some things in common. The Azure Sphere SDK Preview for Visual Studio enables you to use the Visual Studio integrated development environment (IDE) to develop, build, and debug both types of applications. The SDK preview also supports IntelliSense for both types of applications.

In addition, the following security features apply to both high-level and RTApps:

- Application capabilities
- Device capabilities
- Signing and deployment requirements

### Application capabilities

Regardless of where it runs, every Azure Sphere application must specify the external services and interfaces that it requires—for example, its I/O and network requirements—to prevent any unauthorized or unexpected use.

*Application capabilities* are the resources that an application requires. Application capabilities include the peripherals that the application uses, the internet hosts to which a high-level application connects, and permission to change the network configuration, among others. Every application must have an [application manifest](#), which identifies these resources.

### Device capabilities

A *device capability* enables a device-specific activity. Device capabilities are granted by the Azure Sphere Security Service and are stored in flash memory on the Azure Sphere chip. By default, Azure Sphere chips have no device capabilities.

The `appDevelopment` device capability changes the type of signing that the device trusts. By default, Azure Sphere devices trust production-signed image packages but do not trust SDK-signed image packages. As a result, you cannot sideload an SDK-signed image package to an Azure Sphere device that does not have this capability. When the `appDevelopment` capability is present, however, the device trusts SDK-signed image packages. In addition, it enables you to start, stop, debug, or remove an application from the device. In summary, the application development capability must be present on the device before you can:

- Sideload an image package that was built by Visual Studio or the [azsphere image package](#) command.
- Start, stop, debug, or remove an image package from the Azure Sphere device, regardless of how the image package is signed.

The [azsphere device prep-debug](#) command creates and applies the `appDevelopment` capability and prevents the device from receiving over-the-air (OTA) application updates.

### Signing and deployment requirements

All image packages deployed to an Azure Sphere device must be signed. The Visual Studio Extension for Azure Sphere Preview and the [azsphere image package](#) command sign image packages for testing by using an SDK signing key. Azure Sphere devices trust this key only if the `appDevelopment` device capability is also present.

The Azure Sphere Security Service production-signs image packages when you upload them to the cloud. Production-signed image packages can be sideloaded or loaded over the air.

To prevent the installation of rogue software, applications can be loaded on an Azure Sphere device in only two ways:

- *Sideload*ing for software development and testing. Sideload requires direct access to the device and a device capability that allows test-signed software. Visual Studio sideloads applications during development

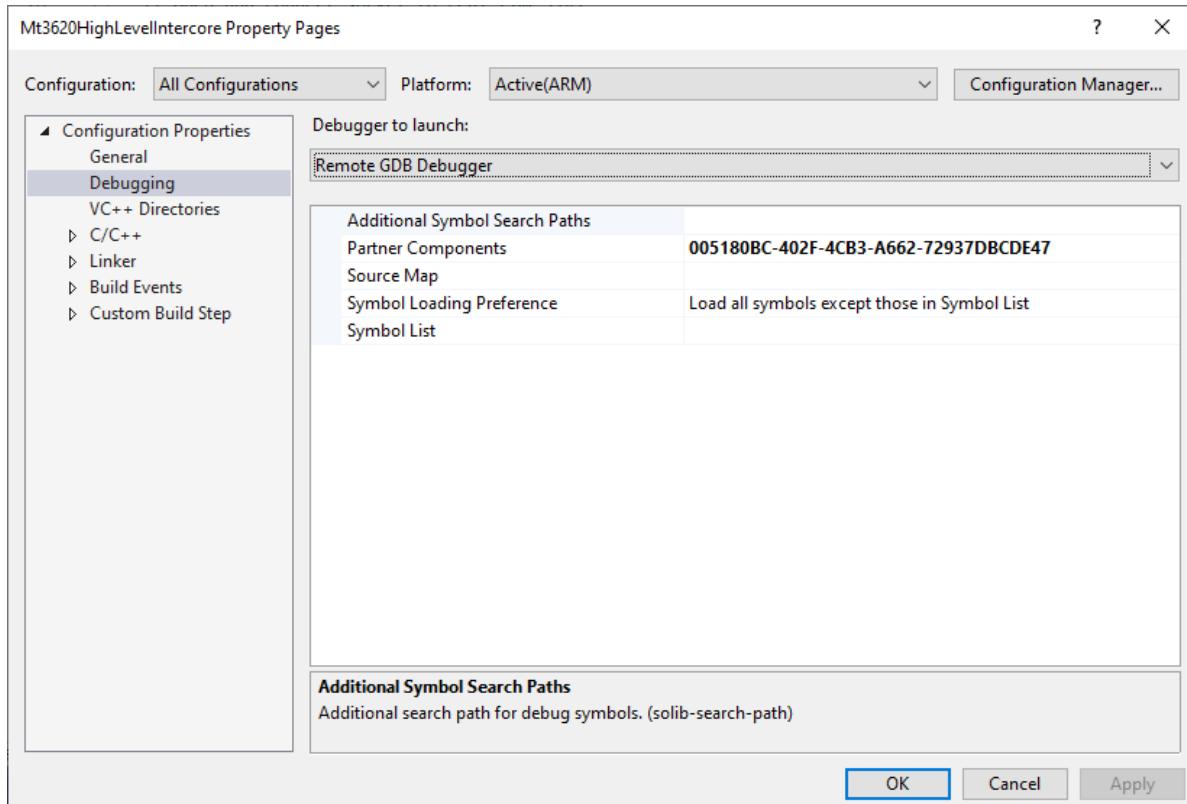
and debugging; you can also sideload manually by using the [azsphere](#) command-line interface (CLI).

- *Over-the-air (OTA) update*, which can be performed only by the Azure Sphere Security Service. Use the [azsphere](#) CLI to create and manage OTA deployments.

## Partner applications

Applications that work together can be considered *partner applications* and then can be sideloaded separately. When you sideload an application that has a partner, the partner application remains on the Azure Sphere device if it has already been deployed. Each application declares a list of its partners in its project properties:

- Applications that use Visual Studio projects (.vcxproj) specify the component ID of the partner app on the **Project Properties>Debugging page**:



- Applications that use CMake specify the component ID of the partner app in the **partnerComponents** field of the **configurations** section of the launch.vs.json file:

```
"partnerComponents": [ "25025d2c-66da-4448-bae1-ac26fcdd3627" ]
```

# Development environment

10/9/2019 • 2 minutes to read

The Azure Sphere SDK Preview for Visual Studio includes header files, libraries, and tools for application development on Azure Sphere devices. Currently, the Azure Sphere SDK only supports application development in C. The SDK is installed in C:\Program Files(x86)\Microsoft Azure Sphere SDK.

## NOTE

The [Quickstarts](#) walk you through building and deploying your first application. We also provide [samples and reference solutions](#) that demonstrate how to use our APIs.

## Requirements

The requirements for application development on Azure Sphere:

- Windows 10 version 1607 or later
- Visual Studio Enterprise, Professional, or Community 2017 version 15.9 or later, or 2019 version 16.04 or later.
- [Azure Sphere SDK Preview](#) for Visual Studio
- An Azure Sphere development board that is connected to your PC by USB

## Azure Sphere Application Runtime

The Azure Sphere Application Runtime provides two sets of libraries for high-level application development, the base APIs and the applibs APIs. The base APIs are based on libraries that don't exclusively target Azure Sphere devices, while the applibs APIs specifically target Azure Sphere devices. High-level applications built with the Azure Sphere SDK compile and link against these interfaces. These APIs can't be used in real-time capable applications.

### Base APIs

LIBRARY	DESCRIPTION
C standard library	POSIX-based C development library.
curl library	Transfers data over HTTP/HTTPS.
Azure IoT C SDK library	Interacts with an IoT Hub.
TLS utilities library	Supports mutual authentication over TLS.

The header files are installed in the `Sysroots\API set\usr\include` folders of the Azure Sphere SDK installation directory. See [base APIs](#) for documentation.

### Applibs

LIBRARY	DESCRIPTION
ADC	Interacts with analog-to-digital converters (ADCs).
Application	Communicates with and controls real-time capable applications.
EventLoop	Monitors and dispatches events.
GPIO	Interacts with GPIOs (general-purpose input/output).
I2C	Interacts with I2C (Inter-Integrated Circuit) bus devices.
Log	Logs debug messages that are displayed when you debug an application through the Azure Sphere SDK. Applications do not have stdout or stdin.
Networking	Manages network connectivity.
PWM	Interacts with pulse-width modulators (PWMs).
RTC	Interacts with the real-time clock (RTC).
SPI	Interacts with SPI (Serial Peripheral Interface) devices.
Storage	Supports the use of on-device storage.
SysEvent	Interacts with system event notifications.
UART	Interacts with UARTs (Universal Asynchronous Receiver/Transmitter).
WiFiConfig	Manages Wi-Fi network connectivity.

The header files are installed in the `Sysroots\API set\usr\include\applibs` folder of the Azure Sphere SDK installation directory. See [Azure Sphere application libraries](#) for reference documentation.

## Tools

The Azure Sphere SDK includes the **azsphere** command-line tool for managing devices, developing and deploying applications, and working with cloud services.

# Application manifest

10/9/2019 • 9 minutes to read

The application manifest describes the resources, also called *application capabilities*, that an application requires. Every application has an application manifest.

Applications must opt in to use capabilities by listing each required resource in the **Capabilities** section of the application manifest; no capabilities are enabled by default. If an application requests a capability that is not listed, the request fails. If the application manifest file contains errors, attempts to sideload the application fail. Each application's manifest must be stored as app\_manifest.json in the root directory of the application folder on your PC.

The Azure Sphere template automatically creates a default application manifest when you create an application. You must edit the default manifest to list the capabilities that your application requires. Each Azure Sphere sample also includes an application manifest. If you base your application on a sample, you will probably also need to edit the manifest.

Different Azure Sphere devices may expose features of the chip in different ways. As a result, the value you use in the manifest to identify a particular feature, such as a GPIO pin, may vary depending on the hardware you're developing for. [Manage target hardware dependencies](#) provides more information about hardware targets for a high-level application. In the application manifest for a [high-level application](#), use the constants that are defined in the JSON file in the [target hardware folder](#). In a real-time capable application (RTApp), use the raw values that are listed in [Contents of the application manifest](#).

When any application is sideloaded or deployed to the device, the Azure Sphere runtime reads the application manifest to ascertain which capabilities the application is allowed to use. Attempts to access resources that are not listed in the manifest will result in API errors such as EPERM (permission denied).

## Contents of the application manifest

The application manifest includes the following items:

NAME	DESCRIPTION
<b>SchemaVersion</b>	Version of the application manifest schema in use. Currently must be 1. Required.
<b>Name</b>	Name of the component. At project creation, this value is set to the name of the project. The name can be any length, but only the first 15 characters are stored in the image package; thus the name appears truncated in inquiries about the image package. If you do not use Visual Studio, see <a href="#">How to manually build and load an application</a> for information on adding a name. Required.

NAME	DESCRIPTION
<b>ComponentId</b>	ID of the component. Visual Studio creates this ID when you build the application. If you do not use Visual Studio, see <a href="#">How to manually build and load an application</a> for information on creating the ID. Required.
<b>EntryPoint</b>	Name of the executable together with the relative path in the application's file system image, which is created when the application is built. Visual Studio currently uses /bin/app for this value. Required.
<b>CmdArgs</b>	Arguments to pass to the application at startup. Enclose each argument in double quotation marks and separate arguments with a comma. Optional.
<b>TargetApplicationRuntimeVersion</b>	Version of the Azure Sphere application runtime that the application requires. This field is automatically added during the build process. Optional. See <a href="#">Use beta features</a> for details.
<b>TargetBetaApis</b>	Specifies that the application requires Beta APIs and identifies the set of Beta APIs used. This field is automatically added during the build process if the application is built using Beta APIs. Optional. See <a href="#">Use beta features</a> for details.
<b>ApplicationType</b>	Type of application. Optional. Set to Debugger only if you are building a replacement for gdbserver.
<b>Capabilities</b>	List of key/value pairs that specify application resource requirements. Required.

The **Capabilities** section supports the following:

NAME	DESCRIPTION
<b>Adc</b>	<p><b>BETA feature</b> The analog-to-digital conversion (ADC) controller that is used by the application. This capability reserves the entire ADC controller (which comprises an 8-pin block), not just pin 0 in the block. In a high-level application, specify the peripheral name that is declared in the hardware definition header file. In an RTApp, specify the <i>AppManifestValue</i> that is declared in the hardware definition JSON file.</p> <p><b>High-level example:</b></p> <pre>"Adc": [ "MT3620_RDB_ADC_CONTROLLER0" ]</pre> <p><b>RTApp Example:</b></p> <pre>"Adc": [ "ADC-CONTROLLER-0" ]</pre> <p><b>API reference:</b> <a href="#">Applibs adc.h</a></p> <p><b>Conceptual:</b> <a href="#">Using ADCs on Azure Sphere</a></p>

NAME	DESCRIPTION
<b>AllowedApplicationConnections</b>	<p><b>BETA feature</b> A list of application component IDs to which the application is allowed to connect.</p> <p><b>Example:</b></p> <pre>"AllowedApplicationConnections": [ "005180BC-402F-4CB3-A662-72937DBCDE47" ]</pre> <p><b>API reference:</b> <a href="#">Applibs application.h</a></p> <p><b>Conceptual:</b> <a href="#">Communicate with a high-level application</a></p>
<b>AllowedConnections</b>	<p>A list of DNS host names or IP addresses (IPv4) to which the application is allowed to connect. If the application uses an Azure IoT Hub, the list must include the IP address or DNS host name for the hub, typically <i>hub-name.azure-devices.net</i>. Port numbers and wildcard characters in names and IP addresses are not accepted.</p> <p><b>Example:</b></p> <pre>"AllowedConnections" : [ "my-hub.example.net", "global.azure-devices-provisioning.net" ]</pre>
<b>AllowedTcpServerPorts</b>	<p><b>BETA feature</b> A list of ports that allow incoming TCP traffic. You can include up to 10 ports, and each port must be listed individually. The supported ports are 1024 to 65535. You can specify the same ports for both TCP and UDP. However, if you specify the same port for more than one app on the device, the second app to run will fail to load.</p> <p><b>Example:</b></p> <pre>"AllowedTcpServerPorts": [ 1024, 65535 ]</pre>
<b>AllowedUdpServerPorts</b>	<p><b>BETA feature</b> A list of ports that allow incoming UDP traffic. You can include up to 10 ports, and each port must be listed individually. The supported ports are 1024 to 65535. You can specify the same ports for both TCP and UDP. However, if you specify the same port for more than one app on the device, the second app to run will fail to load.</p> <p><b>Example:</b></p> <pre>"AllowedUdpServerPorts": [ 1024, 50000 ]</pre>
<b>DeviceAuthentication</b>	<p>A string that specifies the UUID of the Azure Sphere tenant to use for device authentication. This field is automatically added by the Connected Service in Visual Studio when the Device Provisioning Service is used for device authentication.</p> <p><b>Example:</b></p> <pre>"DeviceAuthentication": "77304f1f-9530-4157-8598-30bc1f3d66f0"</pre>
<b>DhcpService</b>	<p><b>BETA feature</b> A boolean that indicates whether the application has permission to configure the DHCP service. True if the application has the capability; otherwise, False.</p> <p><b>Example:</b></p> <pre>"DhcpService" : true</pre> <p><b>API reference:</b> <a href="#">Applibs networking.h</a></p> <p><b>Conceptual:</b> <a href="#">Use network services</a></p>

NAME	DESCRIPTION
<b>Gpio</b>	<p>A list of GPIOs the application uses.</p> <p>In a high-level application, specify the GPIO name that is declared in the hardware definition header file, such as <code>\$MT3620_RDB_LED1_RED</code>.</p> <p>In an RTApp, specify the integers that correspond to the GPIO numbers in the hardware definition JSON file. For example, 8 specifies GPIO 8.</p> <p><b>High-level example:</b></p> <pre>"Gpio": [ "\$MT3620_RDB_HEADER1_PIN6_GPIO", "\$MT3620_RDB_LED1_RED", "\$MT3620_RDB_BUTTON_A" ]</pre> <p><b>RTApp Example:</b> <code>"Gpio": [ 8, 12 ]</code></p> <p><b>API reference:</b> <a href="#">Applibs gpio.h</a></p> <p><b>Conceptual:</b> <a href="#">Using GPIOs on Azure Sphere</a></p>
<b>I2cMaster</b>	<p>A list of I2C master interfaces that are used by the application.</p> <p>In a high-level application, specify the peripheral name that is declared in the hardware definition header file.</p> <p>In an RTApp, specify the <i>AppManifestValue</i> that is declared in the hardware definition JSON file.</p> <p><b>High-level example:</b></p> <pre>"I2cMaster": [ "\$MT3620_RDB_HEADER2_ISU0_I2C", "\$MT3620_RDB_HEADER4_ISU1_I2C" ]</pre> <p><b>RTApp example:</b> <code>"I2cMaster": [ "ISU0", "ISU1" ]</code></p> <p><b>API reference:</b> <a href="#">Applibs i2c.h</a></p> <p><b>Conceptual:</b> <a href="#">Using I2C with Azure Sphere</a></p>
<b>I2sSubordinate</b>	<p>The Inter-IC Sound (I2S) subordinate interface used by an RTApp. This capability isn't available to high-level applications.</p> <p><b>RTApp example:</b> <code>"I2sSubordinate": [ "I2S0", "I2S1" ]</code></p>
<b>MutableStorage</b>	<p>Mutable storage settings that allow the application to use persistent storage.</p> <p><b>Example:</b> <code>"MutableStorage" : { "SizeKB": 64, }</code></p> <p><b>API reference:</b> <a href="#">Applibs storage.h</a></p> <p><b>Conceptual:</b> <a href="#">Using storage on Azure Sphere</a></p>
<b>NetworkConfig</b>	<p><b>BETA feature</b> A boolean that indicates whether the application has permission to configure a network interface. True if the application has the capability; otherwise, False.</p> <p><b>Example:</b> <code>"NetworkConfig" : true</code></p> <p><b>API reference:</b> <a href="#">Applibs networking.h</a></p> <p><b>Conceptual:</b> <a href="#">Use network services</a></p>
<b>Pwm</b>	<p><b>BETA feature</b> The pulse-width modulator (PWM) that is used by the application.</p> <p>In a high-level application, specify the peripheral name that is declared in the hardware definition header file.</p> <p>In an RTApp, specify the <i>AppManifestValue</i> that is declared in the hardware definition JSON file.</p> <p><b>High-level example:</b></p> <pre>"Pwm": [ "\$MT3620_RDB_LED_PWM_CONTROLLER2" ]</pre> <p><b>RTApp example:</b> <code>"Pwm": [ "PWM-CONTROLLER-0" ]</code></p> <p><b>API reference:</b> <a href="#">Applibs pwm.h</a></p> <p><b>Conceptual:</b> <a href="#">Using ADCs on Azure Sphere</a></p>

NAME	DESCRIPTION
<b>SntpService</b>	<p><b>BETA feature</b> A boolean that indicates whether the application has permission to configure the SNTP service. True if the application has the capability; otherwise, False.</p> <p><b>Example:</b> "SntpService" : true</p> <p><b>API reference:</b> <a href="#">Applibs networking.h</a></p> <p><b>Conceptual:</b> <a href="#">SNTP server</a></p>
<b>SoftwareUpdateDeferral</b>	<p><b>BETA feature</b> A boolean that indicates whether the application has permission to defer software updates for a limited period. True if the application has the capability; otherwise, False.</p> <p><b>Example:</b> "SoftwareUpdateDeferral" : true</p> <p><b>API reference:</b> <a href="#">Applibs eventloop.h</a></p> <p><b>Conceptual:</b> <a href="#">Defer device updates</a></p>
<b>SpiMaster</b>	<p>A list of SPI master interfaces that are used by the application.</p> <p>In a high-level application, specify the peripheral name that is declared in the hardware definition header file.</p> <p>In an RTApp, specify the <i>AppManifestValue</i> that is declared in the hardware definition JSON file.</p> <p><b>High-level example:</b></p> <pre>"SpiMaster": [ "\$MT3620_RDB_HEADER2_ISU0_SPI",     "\$MT3620_RDB_HEADER4_ISU1_SPI" ]</pre> <p><b>RTApp example:</b></p> <pre>"SpiMaster": [ "ISU0", "ISU1" ]</pre> <p><b>API reference:</b> <a href="#">Applibs spi.h</a></p> <p><b>Conceptual:</b> <a href="#">Using SPI with Azure Sphere</a></p>
<b>SystemEventNotifications</b>	<p><b>BETA feature</b> A boolean that indicates whether the application has permission to receive system event notifications. True if the application has the capability; otherwise, False.</p> <p><b>Example:</b> "SystemEventNotifications" : true</p> <p><b>API reference:</b> <a href="#">Applibs sysevent.h</a></p> <p><b>Conceptual:</b> <a href="#">Defer device updates</a></p>
<b>SystemTime</b>	<p><b>BETA feature</b> A boolean that indicates whether the application has permission to configure the system time. True if the application has the capability; otherwise, False.</p> <p><b>Example:</b> "SystemTime" : true</p> <p><b>API reference:</b> <a href="#">Applibs rtc.h</a></p> <p><b>Conceptual:</b> <a href="#">Manage system time and the RTC on Azure Sphere</a></p>
<b>TimeSyncConfig</b>	<p><b>BETA feature</b> A boolean that indicates whether the application has permission to configure the time-sync service. True if the application has the capability; otherwise, False.</p> <p><b>Example:</b> "TimeSyncConfig" : true</p>

NAME	DESCRIPTION
<b>Uart</b>	<p>A list of UART peripherals that the application uses. This capability doesn't enable the dedicated UART on an MT3620 development board. For information about the dedicated UART, see <a href="#">Develop and debug a real-time capable application</a>.</p> <p>In a high-level application, specify the peripheral name that is declared in the hardware definition header file. In an RTApp, specify the <i>AppManifestValue</i> that is declared in the hardware definition JSON file.</p> <p><b>High-level example:</b></p> <pre>"Uart": [ "\$MT3620_RDB_HEADER2_ISU0_UART", "\$MT3620_RDB_HEADER4_ISU1_UART" ]</pre> <p><b>RTApp example:</b> "Uart": [ "ISU0", "ISU1" ]</p> <p><b>API reference:</b> <a href="#">Applibs uart.h</a></p> <p><b>Conceptual:</b> <a href="#">Use UART on Azure Sphere</a></p>
<b>WifiConfig</b>	<p>A boolean that indicates whether the application has permission to use the WifiConfig API to change the Wi-Fi configuration. True if the application has the capability; otherwise, False.</p> <p><b>Example:</b> "WifiConfig" : true</p> <p><b>API reference:</b> <a href="#">Applibs wificonfig.h</a></p> <p><b>Conceptual:</b> <a href="#">Configure networking</a></p>

The **MutableStorage** section supports the following:

NAME	DESCRIPTION
<b>SizeKB</b>	An integer that specifies the size of mutable storage in kibibytes. The maximum value is 64. A value of 0 is equivalent to not having the mutable storage capability.

## Example

The following shows a sample app\_manifest.json file for a high-level application that targets the [MT3620 RDB hardware](#):

```
{
  "SchemaVersion": 1,
  "Name": "MyTestApp",
  "ComponentId": "072c9364-61d4-4303-86e0-b0f883c7ada2",
  "EntryPoint": "/bin/app",
  "CmdArgs": ["-m", "262144", "-t", "1"],
  "Capabilities": {
    "AllowedConnections" : [
      "my-hub.example.net",
      "contoso.azure-devices.net",
      "global.azure-devices-provisioning.net" ],
    "AllowedTcpServerPorts": [ 1024, 65535 ],
    "AllowedUdpServerPorts": [ 1024, 50000 ],
    "DeviceAuthentication": "77304f1f-9530-4157-8598-30bc1f3d66f0",
    "Gpio": [ "$MT3620_RDB_HEADER1_PIN6_GPIO", "$MT3620_RDB_LED1_RED", "$MT3620_RDB_BUTTON_A"
  ],
    "I2cMaster": [ "ISU2" ],
    "MutableStorage" : {
      "SizeKB": 64,
    },
    "SpiMaster": [ "ISU1" ],
    "SystemTime" : true,
    "Uart": [ "ISU0" ],
    "WifiConfig" : true
  }
}
}
```

The sample app\_manifest.json file for MyTestApp does the following:

- Passes four command-line arguments to the app.
- Only allows connections to the DNS hosts my-hub.example.net, contoso.azure-devices.net, and global.azure-devices-provisioning.net.
- **BETA feature** Allows incoming TCP traffic on ports 1024 and 65535.
- **BETA feature** Allows incoming UDP traffic on ports 1024 and 50000.
- Specifies an Azure Sphere tenant to use for device authentication and allow connections to the Device Provisioning Service.
- Specifies the use of three GPIOs.
- Specifies the use of one UART peripheral.
- Enables mutable storage with 64 kibibytes of storage space.
- Enables the app to use the WifiConfig API to change the Wi-Fi configuration.
- Specifies the use of one SPI master interface.
- Specifies the use of one I2C master interface.

# Lifecycle of an application

5/30/2019 • 2 minutes to read

All Azure Sphere applications, whether for the high-level or real-time cores, should be written to run continuously. If an application exits unexpectedly, system software automatically restarts it.

High-level applications should exit only upon receiving a SIGTERM signal from the Azure Sphere OS. The [Azure Sphere samples](#) show how to handle this signal and terminate neatly. If an application fails to exit after it receives the SIGTERM signal, the Azure Sphere OS terminates with the SIGKILL signal.

# Memory available on Azure Sphere

10/9/2019 • 2 minutes to read

The following table lists the memory available to Azure Sphere applications that are running on an MT3620 chip.

MEMORY TYPE	AMOUNT	AVAILABILITY
Flash	1 MiB	Shared
RAM	256 KiB	High-level applications
SYSRAM	64 KB per real-time core	Real-time core
Tightly-coupled memory (TCM)	192 KB per real-time core	Real-time core

Azure Sphere provides 1 MiB of flash memory that is shared between high-level and real-time capable applications (RTApps). The Azure Sphere OS uses this space for application image packages and to map RTApps, either for execute in place (XIP) or for loading into TCM.

In addition, 256 KiB of RAM is available for the high-level core. Up to 1 KiB of this space may be allocated for each shared buffer channel through which high-level applications and RTApps communicate.

Each real-time core also has 64 KB of SYSRAM and 192 KB of TCM. Typically, the TCM is used for fast code execution and the SYSRAM is used for data.

## For more information

- For details about memory use in high-level applications, see [Memory available for high-level applications](#)
- For details about memory use in RTApps, see [Manage memory and latency considerations](#)

# Overview of high-level application development

6/20/2019 • 2 minutes to read

Build high-level applications that run on Azure Sphere devices, or add new features to your application. To build real-time capable applications, see [Overview of real-time capable application development](#).

## Set up your application and development environment

TOPIC	DESCRIPTION
<a href="#">Development environment</a>	Describes the components of the Azure Sphere development environment.
<a href="#">Build and load an application with Visual Studio</a>	Describes how to build, load, and debug your application with Visual Studio.
<a href="#">Build and load an application with the command line</a>	Describes how to build, load, and debug your application with the command line.
<a href="#">Remove an application</a>	Describes how to remove your application from a device with Visual Studio or the command line.

## Add features to your high-level application

Overviews and tutorials that describe how to add features to your high-level application using the Azure Sphere SDK. Most of these features include a tutorial and [application samples](#).

TOPIC	DESCRIPTION
<a href="#">Enable beta APIs</a>	Describes how to enable beta APIs so you can test new Applibs APIs while they are still in development. Until you perform this task, you can only use production APIs.
<a href="#">Add resources with the application manifest</a>	Describes how to add resources to an application manifest, which is included with every Azure Sphere Application. Many Azure Sphere features and APIs require that you add certain resources to the application manifest.
<a href="#">Connect to web services with curl</a>	Describes how to integrate HTTP and HTTPS web services with the libcurl file transfer library.
<a href="#">Use UART, I2C, and SPI</a>	Describes how to integrate serial communication formats with your high-level application, such as UART, I2C, and SPI so you can connect to peripherals.
<a href="#">Manage system time</a>	Describes how to manage system time with and without power on a device using the NTP service and the real-time clock.

TOPIC	DESCRIPTION
<a href="#">Use read-only and mutable storage</a>	Describes how to use read-only storage in your high-level application, which can only be updated when your application is updated; and use mutable storage, which can be updated with changes that persist when your device reboots.
<a href="#">Use Azure IoT</a>	Describes how to integrate Azure IoT with your application by using an Azure IoT hub or Azure IoT Central.
<a href="#">Connect to Ethernet</a>	Describes how to communicate with devices on an Ethernet network through standard TCP or UDP networking.

## Perform common and recommended development tasks

- [Pass arguments to an application](#)
- [Error handling and logging](#)
- [Initialization and termination](#)
- [Determine application memory usage](#)
- [Asynchronous events and concurrency](#)
- [Use a system timer as a watchdog](#)
- [Periodic tasks](#)
- [Application lifecycle](#)
- [Manage target hardware dependencies\]](#)

## Command line and API reference

- [Azure Sphere command-line reference](#)
- [Azure Sphere Applibs reference](#)

# Use Beta API features

10/9/2019 • 4 minutes to read

An Azure Sphere SDK release may contain both production APIs and Beta APIs. Beta APIs are still in development and may change in or be removed from a later release. In most cases, new APIs are marked Beta in their first release and moved to production in a subsequent release. Beta APIs provide early access to new features, enabling prototyping and feedback before they are finalized. Applications that use Beta APIs will generally require modifications after future Azure OS and SDK releases to continue to work correctly.

Beta features are labeled **BETA feature** in the documentation. Every Azure Sphere high-level application specifies whether it targets only production APIs or both production and Beta APIs.

## Target API sets, ARV, and sysroots

The target API set indicates which APIs the application uses: either production APIs only or production and Beta APIs. The Target API Set value is either an integer that represents the application runtime version (ARV) or the ARV plus a string that identifies the Beta API release. At the 19.05 release, the ARV was incremented from 1 to 2, and at the 19.09 release it was increased again to 3. The numeric value alone specifies only the production APIs in the ARV, whereas the "value+BetaNumber" specifies the production and Beta APIs in a particular release. For example, ARV 2 indicates the 19.05 release, and "2+Beta1905" specifies the production and Beta APIs in the 19.05 release.

The Azure Sphere SDK implements multiple API sets by using *sysroots*. A sysroot specifies the libraries, header files, and tools that are used to compile and link an application that targets a particular API set. The sysroots are installed in the Microsoft Azure Sphere SDK directory in the sysroots subfolder.

## Develop applications that use Beta APIs

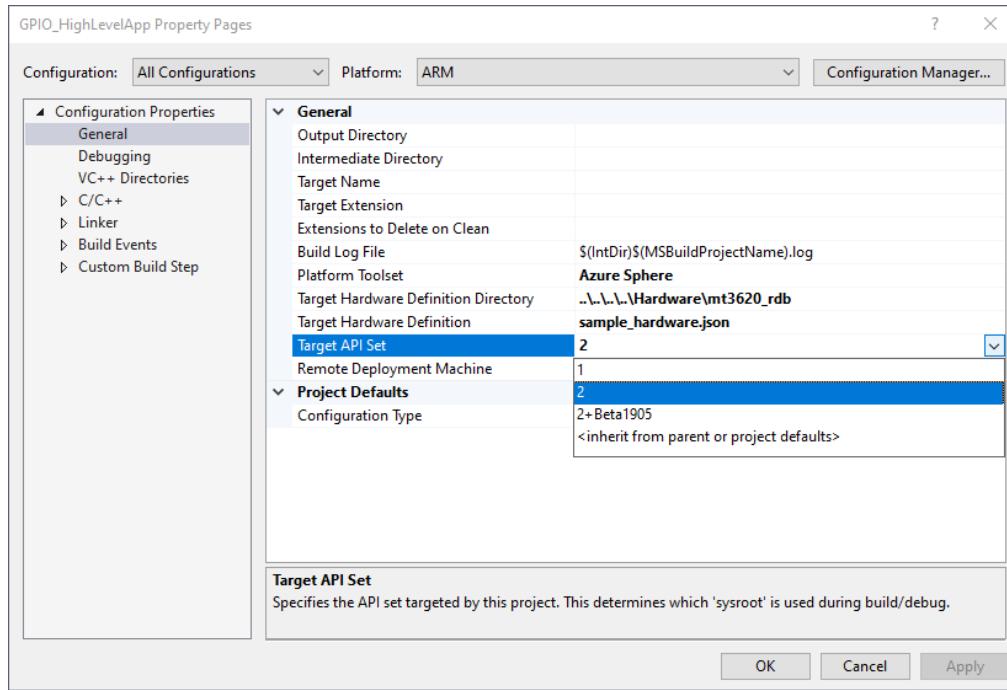
If you start with an Azure Sphere sample that uses a Beta API, the Target API Set by default is set to the production and Beta APIs for the current release.

If you do not base your application on one of the samples, or if you base it on a sample that uses only production APIs, follow these steps to use Beta APIs.

### Use Beta APIs in a Visual Studio project

To set the target API set in a Visual Studio project, set the Project Properties.

1. Open the Azure Sphere project in Visual Studio.
2. On the **Project** menu, select **Project Properties...**
3. Ensure that the Configuration is set to either All Configurations or Active (Debug).
4. In the list of General properties, select **Target API Set**.
5. In the drop-down menu, select the target API set and then click OK.



## Use Beta APIs in a CMake project

To set the target API set in a CMake project, update **AzureSphereTargetApiSet** in CMakeSettings.json. The easiest way to do this is to edit the JSON file directly and change the following field:

**"AzureSphereTargetApiSet": API-set**

Replace *API-set* with the required ARV value: either an integer or an integer plus a string. For example, to target the production and Beta APIs in the 19.09 release, you would update the JSON as follows:

```
"AzureSphereTargetApiSet": "3+Beta1909"
```

## Update applications that use Beta APIs

If you've already created an application that uses Beta APIs, you should update it when the Target API Set options change, which typically occurs at each feature release. Beta APIs may be moved directly from Beta status to production, resulting in a new ARV, or they might be changed and remain in Beta. If you update an application that uses Beta APIs to target a more recent target API set, you may encounter errors or warnings about removed or deprecated APIs.

## Beta APIs and OS compatibility

The compatibility of an application with the Azure Sphere OS depends on the target API set with which the application was built and the latest ARV that the OS version supports. A *down-level* application or OS uses an older ARV (which has a lower number), and an *up-level* application or OS uses a more recent ARV (which has a higher number). The following sections describe what to expect in each possible scenario.

### Down-level applications with up-level OS

Existing down-level images that use only production APIs are guaranteed to work successfully with up-level versions of the Azure Sphere OS. For example, an application that was built with Target API Set 1 runs successfully on an Azure Sphere OS that supports ARV 2. Thus, your existing deployed applications will continue to operate properly after OTA Azure Sphere OS

updates. You can either sideload or OTA-deploy down-level, production-only images to an up-level OS without error.

Down-level images that use Beta APIs are not guaranteed to work successfully with up-level versions of the Azure Sphere OS. For example, an application that was built with Target API Set 1+Beta1902 might fail to run on an Azure Sphere OS that has ARV 2. Attempts to sideload such an image return an error unless you use the --force flag on the **azure sphere device sideload deploy** command. Similarly, the **azsphere device link-feed**, **azsphere component publish**, and **azsphere component image add** commands require the --force flag to upload such an image. No current checks subsequently prevent a previously uploaded down-level image that uses Beta APIs from being deployed alongside an up-level OS that no longer supports those Beta APIs.

### **Up-level applications with down-level OS**

Up-level applications cannot be deployed to down-level versions of the Azure Sphere OS, regardless of whether they use Beta APIs. Attempts to sideload such an image will fail with an error. Attempts to deploy over-the-air are not currently possible because the up-level SDK and OS are released simultaneously.

# Manage target hardware dependencies

10/9/2019 • 6 minutes to read

Azure Sphere hardware is available from multiple vendors, and each vendor may expose features of the underlying chip in different ways. Azure Sphere applications manage hardware dependencies by using *hardware definition files*, which define the features that a particular Azure Sphere-compatible chip, board, or module exposes. The Azure Sphere samples repository on GitHub includes [hardware definition files](#) for most current Azure Sphere hardware. Check with your hardware manufacturer if you need more information.

This topic describes how to set the target hardware for your high-level application and provides general information about hardware definition files. It also explains how to develop a hardware abstraction for your own products.

## NOTE

The information in this topic applies only to high-level applications.

## Set the hardware target for an application

To set the target hardware for an application, you need to set the project properties to use the corresponding hardware definition and include the appropriate header file. You can use the hardware definition files that are supplied with the samples or other files supplied by your hardware vendor.

The Azure Sphere samples use an abstraction that enables them to run on any hardware. When you build a sample application, you can use the sample hardware abstraction files. If you're creating a new application that targets specific hardware, you need to follow these steps:

1. Copy the JSON and .h files for your hardware and add them to your project. A hardware platform may involve a board, a module, and a chip, so be sure to copy them all. For example, if you're using the MT 3620 reference development board (RDB), copy the files from the mt3620\_rdb and mt3620 folders in the samples repository. Do not copy sample\_hardware.json or sample\_hardware.h.
2. Set the Target Hardware Definition Directory and the Target Hardware Definition for your hardware.

The Target Hardware Definition Directory identifies the folder that contains the hardware definition files for the target hardware. This path is relative to the workspace of the project. The Target Hardware Definition identifies the JSON file in the Target Hardware Definition Directory that defines the mappings from the sample application code to the target hardware.

- For a Visual Studio project, use **Project Properties** to set the Target Hardware Definition Directory and the Target Hardware Definition for your hardware. For example:

```
![Target Hardware Definition properties](../media/targethardwareproperties.png)
```

In the example, the ..\Hardware\mt3620\_rdb folder contains the mt3620\_rdb.json file, which maps peripherals on an MT3620 reference design board, such as the Seeed MT3620 Development Kit, to features of the underlying MT3620 chip.

- For a CMake project, edit the **AzureSphereTargetHardwareDefinitionDirectory** and **AzureSphereTargetHardwareDefinition** fields in the CMakeSettings.json file. For example:

```
```json
"AzureSphereTargetHardwareDefinitionDirectory": "...\\..\\..\\Hardware\\mt3620_rdb",
"AzureSphereTargetHardwareDefinition": "mt3620_rdb.json"
```

```

3. In your application code, include the header file for your target hardware. The following includes the header for the MT3620 RDB hardware, assuming a folder structure that is similar to the one used in the Azure Sphere samples:

```
#include "..\Hardware\mt3620_rdb\inc\hw\mt3620_rdb.h"
```

4. In the JSON file for your board or module, find the name of each peripheral that you plan to use in the application. Use these names in your application code for the corresponding hardware. For example, for MT3620 RDB hardware, your application would use `MT3620_RDB_LED1_RED` to identify the red channel for LED 1, which maps to GPIO 8. For example:

```
// Open red channel for LED1 on MT3620 RDB
err = GPIO_OpenAsInput (MT3620_RDB_LED1_RED);
```

To target the Avnet MT3620 Starter Kit, the application would specify `AVNET_MT3620_SK_USER_LED_RED`:

```
// // Open red channel for LED1 on Avnet MT3620 Starter Kit
err = GPIO_OpenAsInput (AVNET_MT3620_SK_USER_LED_RED);
```

5. In the `app_manifest.json` file, use `$name-in-code` to represent the identifiers for the peripherals. The following example shows how to target LED 1 on MT3620 RDB hardware:

```
"Gpio": [ "$MT3620_RDB_LED1_RED" ]
```

The next example shows how to specify this GPIO for an application that targets the Avnet MT3620 Starter Kit:

```
"Gpio": [ "$AVNET_MT3620_SK_USER_LED_RED" ]
```

## Hardware definition files

A *physical hardware definition file* defines the features that a particular chip or Azure Sphere-compatible board or module exposes.

The base for all hardware definition files is a pair of JSON and header files that identify the features that a particular Azure Sphere chip exposes. The JSON file lists the peripherals that are available on the chip along with the identifiers and application manifest values that are required to use each peripheral. For example, the `mt3620.json` file associates the name `MT3620_GPIO8` with pin 8 on the header and GPIO 8 in the application manifest. In turn, the `mt3620.h` header file defines this pin as follows:

```
// MT3620 GPIO 8
#define MT3620_GPIO8 (8)
```

For each type of board or module, a board-specific JSON file maps the features exposed by that board to the features of the underlying chip. Thus, the MT3620 reference board design exposes the red channel on LED1 via

GPIO 8, so mt3620\_rdb.json defines `MT3620_RDB_LED1_RED` as follows:

```
{"Name": "MT3620_RDB_LED1_RED", "Type": "Gpio", "Mapping": "MT3620_GPIO8", "Comment": "LED 1 Red channel uses GPIO8."}
```

This mapping associates the identifier `MT3620_RDB_LED1_RED` with `MT3620_GPIO8`, which mt3620.json associates with pin 8. In the resulting mt3620\_rdb.h file, `MT3620_RDB_LED1_RED` is defined as follows:

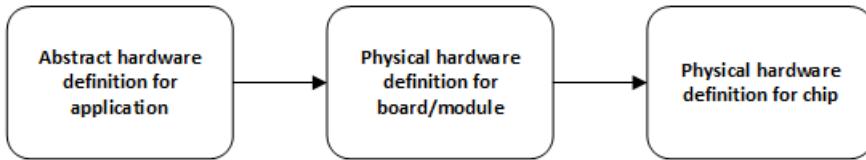
```
// LED 1 Red channel uses GPIO8.  
#define MT3620_RDB_LED1_RED MT3620_GPIO8
```

As a result, applications that run on an MT3620 RDB can use `MT3620_RDB_LED1_RED`, and the name will resolve to `MT3620_GPIO8` at run time and to `8` when the code is compiled, the application manifest is processed, and the image package is generated.

## Develop a custom hardware abstraction for your product

You can develop a custom hardware abstraction for your product by using a pattern similar to the one used in the Azure Sphere samples repository. Use the abstraction across the different hardware options supported in your product line.

A hardware abstraction requires an *abstract hardware definition file*, which maps the hardware and corresponding identifiers used by an application to the definitions in one or more physical hardware definition files. The following diagram shows the relationships among these files:



The sample\_hardware.json files in the samples repository are abstract hardware definition files for the sample applications. These files map the identifiers used in the sample code to the same peripherals, abstracting how they are exposed differently on each board or module.

### Create the abstraction files

To create a hardware abstraction, follow these steps:

1. Identify the peripherals that your applications use. For example, COFFEE MAKER STATUS LED indicates that the application requires an LED; DISHWASHER WASHING SPI MOTOR indicates that the application requires an SPI.
2. Create a JSON file for each hardware platform that your application targets. Use the files in the Azure Sphere samples repository as a guide. In the **Imports** section of the JSON file, specify the path to the hardware definition file for the hardware platform. For example, the following imports the definitions from mt3620\_rdb.json:

```
"Imports" : [ {"Path": "../mt3620_rdb/mt3620_rdb.json"} ],
```

3. For each common peripheral, the **Peripherals** section of the JSON file should contain a line in the following form:

```
{"Name": "<name-in-code>", "Type": "<type>", "Mapping": "<name-in-imported-definition>", "Comment": "<helpful info>"}
```

- Replace *name-in-code* and *name-in-imported-definition* with the identifiers used for the peripheral in your application and in the imported physical hardware definition file, respectively
- Replace *type* with the type of peripheral. See the hardware definition files for examples.
- Replace *helpful-info* with information to appear in the generated header file

For example, the following line in a JSON file maps COFFEE MAKER STATUS LED to the red channel of LED 1 on an MT3620 RDB:

```
{"Name": "COFFEE_MAKER_STATUS_LED", "Type": "Gpio", "Mapping": "MT3620_RDB_LED1_RED", "Comment": "MT3620 RDB: LED 1 (red channel)"}
```

4. Generate a header file from each JSON file. In an Azure Sphere Developer Command Prompt, run **azsphere hardware-definition generate-header**, and supply the JSON file as the --input parameter.

For example:

```
azsphere hardware-definition generate-header --input CoffeeMaker.json
```

This example generates the CoffeeMaker.h file and places it in the inc\hw subdirectory of the folder that contains the input file.

### Add abstraction support to your application

After you create a hardware abstraction, follow these steps to use it in your application:

1. In your application code, include the header file for the hardware abstraction, such as CoffeeMaker.h.
2. In the Project Properties, [set the hardware target](#) for the application.

# Defer device updates

10/9/2019 • 3 minutes to read

## BETA feature

A high-level application can temporarily defer updates to the Azure Sphere OS and to application images to prevent the update from interrupting critical processing. An Azure Sphere device in a kitchen appliance, for example, could defer updates during use. To have the opportunity to defer updates, the app registers for update notifications. After the OS downloads the update, it notifies the application, which can get details about the update and request deferral.

Real-time capable applications (RTApps) cannot receive update notifications or request deferrals. A high-level app is responsible for managing update deferral on behalf of RTApps on the device.

## Deferral requirements

Applications that defer updates must enable beta APIs, include the appropriate header files, and add deferral settings to the [application manifest](#).

### Enable beta APIs

Complete the steps in [beta API features](#) to enable your app to use beta APIs for Azure Sphere.

### Header files

Include the eventloop and sysevent headers in your project:

```
#include <applibs/eventloop.h>
#include <applibs/sysevent.h>
```

### Application manifest settings

To be notified about software update events and to have the opportunity to defer such updates, an application includes two capabilities in the [application manifest](#) file:

- **SystemEventNotifications**
- **SoftwareUpdateDeferral**

Set the **SystemEventNotifications** field to `true` in the `app_manifest.json` file to receive notifications of system update events. The application must also register for event notifications, as described in [Notification mechanism](#). Set **SoftwareUpdateDeferral** to `true` to enable the app to defer and resume updates.

The following shows the `app_manifest.json` settings required to enable both notification and deferral:

```
"Capabilities": {
    "SystemEventNotifications" : true,
    "SoftwareUpdateDeferral" : true
}
```

## Notification mechanism

Azure Sphere supports update deferral through an event notification and event loop mechanism. The application creates an `EventLoop`, which is a single-threaded object through which the application is notified of pending events.

To receive notifications, an app calls **SysEvent\_RegisterForEventNotifications**, passing these parameters:

- A pointer to the event loop
- The **SysEvent\_Events\_Update** enumeration value, which indicates that the app requests notification for update events
- A pointer to an app-defined callback function
- An optional context pointer that is passed to the callback

Only one `EventLoop` object can be used with **SysEvent\_RegisterForEventNotifications** per thread. A second call to **SysEvent\_RegisterForEventNotifications** with a different `EventLoop` object will fail.

After registration, the application calls **EventLoop\_Run**, which invokes the callback function if an event has changed status. The callback function receives a **SysEvent\_Events** value, which identifies the type of event. In turn, the callback calls **SysEvent\_Info\_GetUpdateData** to find out whether the event is an OS or application update and how long the update can be deferred. The app can then determine how to handle the event notification.

Azure Sphere may send several status notifications for each update event:

| STATUS                          | DESCRIPTION  |
|---------------------------------|--|
| <b>SysEvent_Status_Pending</b>  | A 10-second warning that an update event will occur, with the opportunity to defer.                    |
| <b>SysEvent_Status_Final</b>    | A 10-second warning that an update event will occur, without the opportunity for deferral.             |
| <b>SysEvent_Status_Rejected</b> | The previously pending event has been deferred and will occur later.                                   |
| <b>SysEvent_Status_Complete</b> | The software update process is complete. This event notification is sent only for application updates. |

An app can request deferral only after it receives a **SysEvent\_Status\_Pending** notification. To allow the update to occur immediately, the application can ignore the notification.

To defer the update, the application calls **SysEvent\_DeferEvent**, passing the number of minutes to defer the update. For an OS update, the maximum deferral is 1440 minutes (24 hours). For an application update, the maximum deferral period is 10,020 minutes (167 hours).

An application can end an update deferral prematurely by calling **SysEvent\_ResumeEvent**. For an application or OS update, a successful call to **SysEvent\_ResumeEvent** restarts the notification process and thus sends a new **SysEvent\_Status\_Pending** notification. The app should not call **SysEvent\_DeferEvent** again until it has received such a notification.

When the application receives the **SysEvent\_Status\_Final** notification, it should prepare for the update. For an OS update, the application should do whatever cleanup is required before device reboot. For an application update, the high-level application should do whatever is necessary before it or any other application on the device is restarted. In the initial Beta release, application notification does not currently specify which application is being updated.

When notification is no longer required, the app should call **SysEvent\_UnregisterForEventNotifications** and then **EventLoop\_Close** to release the memory allocated for the event loop object. Note that after all event notifications have been unregistered, the app can use a new `EventLoop` object.

# Using GPIOs on Azure Sphere

10/9/2019 • 3 minutes to read

Azure Sphere supports GPIOs (general-purpose input/output). A GPIO is a type of programmable digital pin on an integrated circuit. GPIOs don't have predefined functionality and their behavior can be customized by an application. Some common uses for GPIOs are to change the state of hardware devices, control LEDs, and read the state of switches.

## NOTE

This topic describes how to use GPIOs in a high-level application. See [Use peripherals in a real-time capable application](#) for information about GPIO use in RTApps.

Azure Sphere high-level applications can communicate with GPIOs by calling Applibs [GPIO APIs](#). The [GPIO\\_HighLevelApp sample](#) demonstrates how to communicate with GPIOs on an MT3620 device.

The following operations are supported for GPIO:

- Read input
- Set output to high or low
- Polling / software interrupts

## GPIO requirements

Applications that communicate with GPIOs must include the appropriate header files for GPIO and add GPIO settings to the [application manifest](#).

All applications must [set their target hardware](#) and include the corresponding hardware definition header file.

### Header Files

```
#include <applibs/gpio.h>
#include "path-to-your-target-hardware.h"
```

Replace "path-to-your-target-hardware.h" with the path to the header file for your hardware.

### Application manifest settings

The GPIO settings in the application manifest list the GPIOs that are accessed by the application. Only one application can use a GPIO at a time. To configure these settings, add the `Gpio` capability to the application manifest, and then add each GPIO to the capability. The [Azure Sphere application manifest](#) topic has more details.

In your code, use the constants that are defined for your hardware to identify the GPIOs. The compiler will translate these values to raw values when you build the app.

For example, here's an excerpt from an application manifest for an application that targets an [MT3620 reference development board \(RDB\)](#) and acquires three of its GPIOs (1, 8, and 12):

```
"Gpio": [ "$MT3620_RDB_HEADER1_PIN6_GPIO", "$MT3620_RDB_LED1_RED", "$MT3620_RDB_BUTTON_A" ]
```

The following excerpt shows how to specify the same GPIOs in an application that targets the [Avnet MT3620 Starter Kit](#):

```
"Gpio": [ "$AVNET_MT3620_SK_GPIO1", "$AVNET_MT3620_SK_USER_LED_RED", "$AVNET_MT3620_SK_USER_BUTTON_A" ]
```

## Open a GPIO as input

If you need to read from a GPIO but not write to it, you can open it as an input. Call [GPIO\\_OpenAsInput](#) to open a GPIO and set it to input. This will retrieve a file descriptor for operations on the GPIO. You can read from a GPIO while it is set to input, but you can't write to it. If a GPIO is set to input, you must [close](#) it before you can set it to output.

## Open a GPIO as output

If you need to write to a GPIO, you must open it as an output. Call [GPIO\\_OpenAsOutput](#) to open a GPIO and set it to output. This will retrieve a file descriptor for operations on the GPIO, set the [output mode](#), and the initial [value](#). When a GPIO is set to output, you can write to it and read from it. If a GPIO is set to output, you must [close](#) it before you can set it to input.

## Poll a GPIO

When the GPIO is open, you can monitor it for events, such as a button press. To do so, you need to set up a timer to poll the GPIO. Hardware interrupts for GPIOs are not supported on Azure Sphere, so you need to use polling. The [GPIO sample](#) demonstrates how to poll a GPIO.

## Read from a GPIO

To read from the GPIO, call [GPIO\\_GetValue](#).

## Write to a GPIO

To write to a GPIO, call [GPIO\\_SetValue](#).

## Close a GPIO

To close the GPIO, call the POSIX function `close()`.

## MT3620 support

The supported GPIO features for the MT3620 chip are listed in [MT3620 Support Status](#). The [MT3620 development board user guide](#) describes the pin layout and pin functions on the MT3620 RDB.

The Azure Sphere Samples repo on GitHub contains header and JSON files that define GPIOs for the [MT3620 chip](#) and [MT3620 RDB](#), along with [other MT3620 hardware](#).

# Use I2C, SPI, and UART

7/2/2019 • 2 minutes to read

Azure Sphere supports several serial communication formats for application development through [Applibs](#) and standard APIs. The supported formats include UART (universal asynchronous receiver-transmitter), I2C (Inter-Integrated Circuit), and SPI (Serial Peripheral Interface). UART, I2C, and SPI support allows you to integrate a wide range of peripherals into your Azure Sphere applications.

| TOPIC                    | DESCRIPTION  |
|--------------------------|--|
| <a href="#">Use I2C</a>  | Develop applications that communicate with peripherals through I2C. This topic provides an overview of I2C on Azure Sphere, and it describes how to setup and use I2C in your application. |
| <a href="#">Use SPI</a>  | Develop applications that communicate with peripherals through SPI. This topic provides an overview of SPI on Azure Sphere, and it describes how to setup and use SPI in your application. |
| <a href="#">Use UART</a> | Develop applications that communicate with UART. This topic provides an overview of UART on Azure Sphere, and it describes how to set up and use UART in your application.                 |

## Samples

The following I2C, SPI, and UART samples for Azure Sphere are available on Github.

| TOPIC                              | DESCRIPTION   |
|------------------------------------|---|
| <a href="#">LSM6DS3 I2C sample</a> | Demonstrates how to display data from an accelerometer that is connected to an Azure Sphere device through I2C. |
| <a href="#">LSM6DS3 SPI sample</a> | Demonstrates how to display data from an accelerometer that is connected to an Azure Sphere device through SPI. |
| <a href="#">UART sample</a>        | Demonstrates how to send and receive data over UART.  |

## Hardware documentation

The topics above include device wiring information and feature overviews for application developers. For information about I2C, SPI, and UART for hardware manufacturers and designers, go to the [Hardware and manufacturing overview](#).

# Using I2C with Azure Sphere

10/9/2019 • 3 minutes to read

Azure Sphere supports Inter-Integrated Circuit (I2C) in master mode. I2C is a serial bus that connects lower-speed peripherals to microcontrollers. I2C uses a multi-master/multi-subordinate model where a master device controls a set of subordinate devices. I2C is often used with peripherals that only require simple lightweight communication with a microcontroller, such as setting controls, power switches, and sensors.

Applications can access peripherals through I2C by calling Applibs [I2C APIs](#) to perform operations on an I2C master interface. The [LSM6DS3 I2C sample](#) describes how to configure the hardware for I2C on an MT3620 device and use I2C in an application.

## I2C Requirements

Applications that use I2C must include the appropriate header files for I2C, and add I2C settings to the [application manifest](#).

All applications must [set their target hardware](#) and include the corresponding hardware definition header file.

### Header Files

```
#define I2C_STRUCTS_VERSION 1
#include <applibs/i2c.h>
#include "path-to-your-target-hardware.h"
```

Declare the `I2C_STRUCTS_VERSION` preprocessor definition before including the header file. This specifies the struct version that is used by the application.

Replace "path-to-your-target-hardware.h" with the path to the header file for your hardware.

### Application manifest settings

To use the I2C APIs, you must add the `I2cMaster` capability to the application manifest, and then add each I2C master interface to the capability. This enables the application to access the interface. The [Azure Sphere application manifest](#) topic has more details about the application manifest.

In your code, use the constants that are defined for your hardware to identify the I2C interfaces. The compiler will translate these values to raw values when you build the app.

For example, here's an excerpt from an application manifest that targets an [MT3620 reference development board \(RDB\)](#) and configures two I2C master interfaces:

```
"I2cMaster": [ "$MT3620_RDB_HEADER2_ISU0_I2C", "$MT3620_RDB_HEADER4_ISU1_I2C" ],
```

The following excerpt shows how to specify the same I2C master interfaces in an application that targets the [Avnet MT3620 Starter Kit](#):

```
"I2cMaster": [ "$AVNET_MT3620_SK_ISU0_I2C", "$AVNET_MT3620_SK_ISU1_I2C" ]
```

## Open an I2C master interface

Before you perform operations on an I2C master interface, you must open it by calling the [I2CMaster\\_Open](#)

function.

## Update the settings for an I2C master interface

After you open the master interface, you can change the settings:

- To change the bus speed for operations on the master interface, call [I2CMaster\\_SetBusSpeed](#)
- To change the timeout for operations, call [I2CMaster\\_SetTimeout](#)

## Perform read and write operations on the I2C master interface

Azure Sphere supports several options for performing read and write operations with I2C. These options are all blocking, synchronous operations.

For one-way write or read operations you can call [I2CMaster\\_Write](#) or [I2CMaster\\_Read](#). This is the simplest way to perform operations on an I2C master interface because it only specifies one operation and it includes the address of the subordinate device in the function call.

You can call [I2CMaster\\_WriteThenRead](#) to perform a combined write then read operation in a single bus transaction without interruption from another transaction.

For interoperability with some POSIX interfaces, you can call POSIX read(2) and write(2) functions to perform one-way transactions. You must call [I2CMaster\\_SetDefaultTargetAddress](#) to set the address of the subordinate device before you call read(2) or write(2).

You can call these functions to perform 0-byte write operations in order to verify the presence of a subordinate device. If a read or write operation fails, your application must handle reissuing the request.

## Close the I2C interface

To close the interface, you must call the standard POSIX function `close()`.

### MT3620 support

This section describes the I2C options that only apply when running Azure Sphere on the MT3620.

The I2C specifications for the MT3620 chip are listed in [MT3620 Support Status](#). The [MT3620 development board user guide](#) describes the pin layout and functions for wiring.

The Azure Sphere Samples repo on GitHub contains header and JSON files that define I2C master interfaces for the [MT3620 chip](#) and [MT3620 RDB](#), along with [other MT3620 hardware](#).

- When you configure the MT3620 dev board, you can use any [ISU](#) port as an I2C master interface. When you use an ISU port as an I2C master interface, you can't use the same port as an SPI or UART interface.
- 10-bit subordinate device addresses are not supported on the MT3620; only 7-bit addresses are supported.
- The MT3620 supports 100 KHz, 400 KHz, and 1 MHz bus speeds, but not 3.4 Mhz.
- 0-byte I2C reads are not supported on the MT3620.

# Using SPI with Azure Sphere

10/9/2019 • 4 minutes to read

Azure Sphere supports Serial Peripheral Interface (SPI) in master mode. SPI is a serial interface used for communication between peripherals and integrated circuits. SPI uses a master/subordinate model where a master device controls a set of subordinate devices. In contrast to [I2C](#), SPI can be used with more complex higher speed peripherals.

Applications can access peripherals through SPI by calling Applibs [SPI APIs](#) to perform operations on an SPI master interface. The [LSM6DS3 SPI sample](#) describes how to configure the hardware for SPI on an MT3620 device and use SPI in an application.

## Chip select

Chip select manages the connection between an SPI master interface and a set of subordinate devices; and allows the master interface to send and receive data to each subordinate device independently. Azure Sphere supports the active-low and active-high settings for chip select, with active-low as the default setting. Each SPI master interface can be used exclusively by one application. The application must open the SPI master interface and identify each connected subordinate device before performing read and write operations on the interface. The SPI read and write operations on Azure Sphere use blocking APIs.

## SPI requirements

Applications that use SPI must include the appropriate header files for SPI, and add SPI settings to the [application manifest](#).

All applications must [set their target hardware](#) and include the corresponding hardware definition header file.

### Header files

```
#define SPI_STRUCTS_VERSION 1
#include <applibs/spi.h>
#include "path-to-your-target-hardware.h"
```

Declare the `SPI_STRUCTS_VERSION` preprocessor definition before including the header file. This specifies the struct version that is used by the application.

Replace "path-to-your-target-hardware.h" with the path to the header file for your hardware.

### Application manifest settings

To use the SPI APIs, you must add the `SpiMaster` capability to the application manifest, and then add each SPI master controller to the capability. This enables the application to access the controller. The [Azure Sphere application manifest](#) topic has more details about the application manifest.

In your code, use the constants that are defined for your hardware to identify the SPI master interfaces. The compiler will translate these values to raw values when you build the app.

For example, here's an excerpt from an application manifest that targets an [MT3620 reference development board \(RDB\)](#) and configures two SPI master interfaces:

```
"SpiMaster": [ "$MT3620_RDB_HEADER2_ISU0_SPI", "$MT3620_RDB_HEADER4_ISU1_SPI" ],
```

The following excerpt shows how to specify the same SPI master interfaces in an application that targets the [Avnet MT3620 Starter Kit](#):

```
"SpiMaster": [ "$AVNET_MT3620_SK_ISU0_SPI", "$AVNET_MT3620_SK_ISU1_SPI" ]
```

## Configure chip select and open an SPI master interface

Before you perform operations on an SPI master interface, you must configure chip select and open the interface. To configure chip select, call the [SPIMaster\\_InitConfig](#) function to initialize the [SPIMaster\\_Config](#) struct. After you initialize [SPIMaster\\_Config](#), update the `csPolarity` field with the [SPI\\_ChipSelectPolarity\\_ActiveLow](#) or [SPI\\_ChipSelectPolarity\\_ActiveHigh](#) setting.

To open an SPI master interface, call the [SPIMaster\\_Open](#) function. This will apply the default settings to the interface and apply your chip select settings:

- `SPI_Mode_0` for the SPI bit order
- `SPI_BitOrder_MsbFirst` for the communication mode

## Update the settings for an SPI master interface

After initialization you can change the settings for the interface:

- To change the bit order, call [SPIMaster\\_SetBitOrder](#)
- To change the SPI bus speed, call [SPIMaster\\_SetBusSpeed](#)
- To change the communication mode, call [SPIMaster\\_SetMode](#)

## Perform read and write operations on the SPI master interface

Azure Sphere supports several options for performing read and write operations with SPI. For one-way read or write operations and to maintain interoperability with some POSIX APIs, you can call the POSIX `read(2)` and `write(2)` functions.

You can call the [SPIMaster\\_WriteThenRead](#) function to perform a combined write then read operation in a single bus transaction without interruption from another transaction.

Call the [SPIMaster\\_TransferSequential](#) function when you need more precise control over the timing between read or write operations. This allows you to issue multiple read and write operations between a pair of CS enable and disable states.

## Close the SPI interface

To close the interface, call the standard POSIX function `close()`.

## MT3620 support

This section describes the SPI options that only apply when running Azure Sphere on the MT3620 development board.

The SPI specifications for the MT3620 are listed in [MT3620 Support Status](#). The [MT3620 development board user guide](#) describes the pin layout and functions for wiring.

The Azure Sphere Samples repo on GitHub contains header and JSON files that define SPI master interfaces for the [MT3620 chip](#) and [MT3620 RDB](#), along with [other MT3620 hardware](#).

- When you configure the MT3620 dev board, you can use any [ISU](#) port as an SPI master interface. You can

connect up to two subordinate devices to each ISU. When you use an ISU port as an SPI master interface, you can't use the same port as an I<sup>2</sup>C or UART interface.

- The MT3620 supports SPI transactions that are up to 40 MHz.
- The MT3620 doesn't support simultaneous bidirectional read and write (full-duplex) SPI operations within a single bus transaction.

# Use UART on Azure Sphere

10/9/2019 • 3 minutes to read

Azure Sphere supports universal asynchronous receiver-transmitters (UARTs) for serial communication. A UART is a type of integrated circuit that is used to send and receive data over a serial port on a computer or peripheral device. UARTs are widely used and known for their simplicity. However, unlike [SPI](#) and [I2C](#), UARTs do not support multiple subordinate devices.

## NOTE

This topic describes how to use UARTs in a high-level application. See [Use peripherals in a real-time capable application](#) for information about UART use in RTApps.

Azure Sphere high-level applications can communicate with UARTs by calling Applibs [UART APIs](#). The [UART\\_HighLevelApp sample](#) demonstrates how to communicate with UARTs on an MT3620 device.

## UART requirements

Applications that communicate with UARTs must include the appropriate header files, and add UART settings to the [application manifest](#).

All applications must [set their target hardware](#) and include the corresponding hardware definition header file.

### Header Files

```
#define UART_STRUCTS_VERSION 1
#include <applibs/uart.h>
#include "path-to-your-target-hardware.h"
```

Declare the `UART_STRUCTS_VERSION` preprocessor definition before including the header file. This specifies the struct version that is used by the application.

Replace "path-to-your-target-hardware.h" with the path to the header file for your hardware.

### Application manifest settings

The UART settings in the application manifest list the UARTs that are accessed by the application. Only one application can use a UART at a time. To configure these settings, add the `Uart` capability to the application manifest, and then add each UART to the capability. The [Azure Sphere application manifest](#) topic has more details about the application manifest.

In your code, use the constants that are defined for your hardware to identify the UARTs. The compiler will translate these values to raw values when you build the app.

For example, here's an excerpt from an application manifest that targets an [MT3620 reference development board \(RDB\)](#) and configures two UARTs on an MT3620.

```
"Uart": [ "$MT3620_RDB_HEADER2_ISU0_UART", "$MT3620_RDB_HEADER4_ISU1_UART" ],
```

The following excerpt shows how to specify the same UARTs in an application that targets the [Avnet MT3620 Starter Kit]([https://github.com/Azure/azure-sphere-samples/tree/master/Hardware/avnet\\_mt3620\\_sk](https://github.com/Azure/azure-sphere-samples/tree/master/Hardware/avnet_mt3620_sk)):

```
```json
"Uart": [ "$AVNET_MT3620_SK_ISU0_UART", "$AVNET_MT3620_SK_ISU1_UART" ],
```

## Configure and open a UART

Before you perform operations on a UART, you must configure the settings and open the UART. When you open a UART, a file descriptor is returned that you can pass to functions that perform operations on the UART.

To configure the settings, call the [UART\\_InitConfig](#) function to initialize the [UART\\_Config](#) struct. After you initialize the [UART\\_Config](#) struct, you can change the UART settings in the struct.

To open the UART and apply the settings, call the [UART\\_Open](#) function and pass the [UART\\_Config](#) struct.

## Perform read and write operations on a UART

You can use POSIX functions to perform read and write operations on a UART. To perform a read operation on a UART, call the [read\(\)](#) function. To perform a write operation on a UART, call the [write\(\)](#) function.

## Close a UART

To close the UART, call the POSIX function [close\(\)](#).

## MT3620 support

This section describes the UART options that only apply when running Azure Sphere on an MT3620.

The UART specifications for the MT3620 are listed in [MT3620 Support Status](#). The [MT3620 development board user guide](#) describes the pin layout and functions for wiring.

The Azure Sphere Samples repo on GitHub contains header and JSON files that define UARTs for the [MT3620 chip](#) and [MT3620 RDB](#), along with [other MT3620 hardware](#).

The following UART settings are supported. 8N1 (8 data bits, 1 stop bit, and no parity) is the default setting:

- When you configure the MT3620 dev board, you can use any [ISU](#) port as a UART interface. When you use an ISU port as a UART interface, you can't use the same port as an I2C or SPI interface.
- baud rate : 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200, 230400, 460800, 500000, 576000, 921600, 1000000, 1152000, 1500000, and 2000000.
- Data bit: 5, 6, 7, and 8.
- Stop bit: 1 and 2.
- Parity: odd, even, and none.
- Flow control mode: RTS/CTS, XON/XOFF, and no flow control.
- Hardware receive buffer: 32-byte.

# Using ADCs on Azure Sphere

10/9/2019 • 2 minutes to read

This topic describes how to use analog-to-digital converters (ADCs) in a high-level application. See [Use peripherals in a real-time capable application](#) for information about ADC use in RTApps.

Azure Sphere supports analog to digital conversion. An ADC converts an analog input to a corresponding digital value. The number of input channels and the resolution (as number of ADC output bits) are device dependent.

The [ADC\\_HighLevelApp sample](#) demonstrates how to access ADCs on an MT3620 device.

The MT3620 contains a 12-bit ADC with 8 input channels. The ADC compares an input voltage to a reference voltage and produces a value between 0 and 4095 as its output. The ADC input channels and the GPIO pins GPIO41 through GPIO48 map to the same pins on the MT3260. However, if your application uses the ADC then all 8 pins are allocated for use as ADC inputs. None of them can be used for GPIO.

## ADC requirements

Applications that communicate with ADCs must include the adc.h header file and add ADC settings to the [application manifest](#).

All applications must [set their target hardware](#) and include the corresponding hardware definition header file.

### Header Files

```
#include <applibs/adc.h>
#include "path-to-your-target-hardware.h"
```

Replace "path-to-your-target-hardware.h" with the path to the header file for your hardware.

### Application manifest settings

The ADC setting in the application manifest lists the ADC controllers that are accessed by the application. To configure these settings, add the `Adc` capability to the application manifest, and then add each ADC controller to the capability. The Azure Sphere [application manifest](#) topic has more details.

In your code, use the constants that are defined for your hardware to identify the ADC controllers. The compiler will translate these values to raw values when you build the app.

For example, here's an excerpt from an application manifest for an application that targets an [MT3620 reference development board \(RDB\)](#).

```
"Adc": [ "$MT3620_RDB_ADC_CONTROLLER0" ]
```

The following excerpt shows how to specify the same ADC controller in an application that targets the [Avnet MT3620 Starter Kit](#):

```
"Adc": [ "$AVNET_MT3620_SK_ADC_CONTROLLER0" ]
```

**NOTE:** Currently, on the MT3620 you can only access ADC controller 0.

## ADC access

Azure Sphere high-level applications can access ADCs by calling Applibs [ADC APIs](#).

## Open an ADC controller

To open an ADC controller for access, call [ADC\\_Open](#) and pass the ID of the controller as a parameter. A file descriptor will be returned if the call is successful. Otherwise, an error value will be returned.

```
int ADC_Open(ADC_ControllerId id);
```

## Read from an ADC

To read from the ADC, call [ADC\\_Poll](#). You pass the following parameters to ADC\_Poll: the file descriptor returned by ADC\_Open, the ADC channel, and a pointer to where the ADC value will be stored.

To get the number of valid bits (resolution) returned by the ADC\_Poll function, call [ADC\\_GetSampleBitCount](#).

## Set the ADC reference voltage

Call [ADC\\_SetReferenceVoltage](#) to set the ADC reference voltage.

The MT3620 has an on-chip, 2.5 volt reference. Alternatively, it can be connected to an external voltage reference that is less than or equal to 2.5 volts.

## MT3620 support

The supported ADC features for the MT3620 are listed in [MT3620 Support Status](#). The [MT3620 development board user guide](#) describes the pin layout and pin functions.

The Azure Sphere Samples repo on GitHub contains header and JSON files that define ADCs for the [MT3620 chip](#) and [MT3620 RDB](#), along with [other MT3620 hardware](#).

# Using PWMs on Azure Sphere

10/9/2019 • 2 minutes to read

This topic describes how to access device pulse-width modulators (PWMs) for use in Azure Sphere high-level applications.

Pulse-width modulation is achieved by varying the duty cycle (ratio of on time to off time) of a pulsed digital signal. Pulse-width modulators are used in a wide range of applications such as digital motor control, communication, and digital-to-analog conversion.

To use PWMs in your Azure Sphere applications you must include the required header files and `Pwm` capability, and specify the controller for the PWM channel(s) that your application will access.

The [PWM\\_HighLevelApp sample](#) demonstrates how to use PWM in a simple digital-to-analog conversion application on an MT3620 device.

## PWM requirements

Applications that use PWMs must include the appropriate header file and add PWM settings to the [application manifest](#).

All applications must [set their target hardware](#) and include the corresponding hardware definition header file.

### Header Files

```
#include <applibs/pwm.h>
#include "path-to-your-target-hardware.h"
```

Replace "path-to-your-target-hardware.h" with the path to the header file for your hardware.

### Application manifest settings

The PWM setting in the application manifest lists the PWM controllers that are accessed by the application. To configure these settings, add the `Pwm` capability to the application manifest, and then add each PWM controller to the capability. The Azure Sphere [application manifest](#) topic has more details.

In your code, use the constants that are defined for your hardware to identify the PWM controllers. The compiler will translate these values to raw values when you build the app.

For example, here's an excerpt from an application manifest for an application that targets an [MT3620 reference development board \(RDB\)](#).

```
"Pwm": [ "$MT3620_RDB_LED_PWM_CONTROLLER2" ]
```

The following excerpt shows how to specify the same PWM controller in an application that targets the [Avnet MT3620 Starter Kit](#):

```
"Pwm": [ "$AVNET_MT3620_SK_PWM_CONTROLLER2" ]
```

## PWM access

Azure Sphere high-level applications can access a PWM by calling Applibs [PWM APIs](#).

## **Open a PWM controller**

To open a PWM controller for access, call [PWM\\_Open](#) and pass as a parameter the ID of the controller to open. A file descriptor will be returned if the call is successful. Otherwise, -1 will be returned.

[!NOTE] The MT3620 has 12 PWM channels, PWM0 - PWM11. They are organized into 3 groups of 4 channels. Each group is associated with a PWM controller (PWM-CONTROLLER-0, PWM-CONTROLLER-1, PWM-CONTROLLER-2). The PWM channels and GPIO pins GPIO0 through GPIO11 map to the same pins on the MT3260. If your application uses a PWM controller then all of the pins associated with that controller are allocated for use as PWM outputs. None of them can be used for GPIO.

## **Set the state of a PWM channel**

To set or update the state of a PWM channel, call [PWM\\_Apply](#). You pass the following parameters to PWM\_Apply:

- The file descriptor returned by PWM\_Open
- The PWM channel to update; this value is platform dependent
- The period, duty cycle, and polarity to apply to the channel
- Whether to enable or disable the channel

[!NOTE] Minimum and maximum limits for period and duty cycle are device dependent. For example, the MT3620 PWMs run at 2 MHz with 16 bit on/off compare registers. This imposes a duty cycle resolution of 500 ns, and an effective maximum period of approximately 32.77 ms. Consult your specific device's data sheet for details.

## **MT3620 support**

The supported PWM features for the MT3620 are listed in [MT3620 Support Status](#). The [MT3620 development board user guide](#) describes the pin layout and pin functions.

# Connect to web services

10/9/2019 • 7 minutes to read

The Azure Sphere SDK includes the libcurl library, which high-level applications can use to connect and authenticate with HTTP and HTTPS web services. Both server and client authentication are supported, so that applications can verify that they are communicating with the expected server and can prove to the server that their device and Azure Sphere tenant are legitimate. *Mutual authentication* combines the two.

The Azure Sphere samples repo on GitHub includes the following curl samples:

- [HTTPS\\_Curl\\_Easy](#) uses a synchronous (blocking) API for server authentication.
- [HTTPS\\_Curl\\_Multi sample](#) uses an asynchronous (non-blocking) API for server authentication.

Although the synchronous approach to server authentication in `HTTPS_Curl_Easy` is quite simple, Azure Sphere applications should generally use the more complex asynchronous technique shown in the `HTTPS_Curl_Multi` sample, along with an epoll-based, single-threaded event-driven pattern.

The [libcurl website](#) provides thorough documentation of the [libcurl C API](#) and many [examples](#).

## Requirements for applications that use curl

Applications that use the curl library must include the appropriate header files, add a curl dependency to the linker, and provide tenant and internet host information in the [application manifest](#).

### Header files

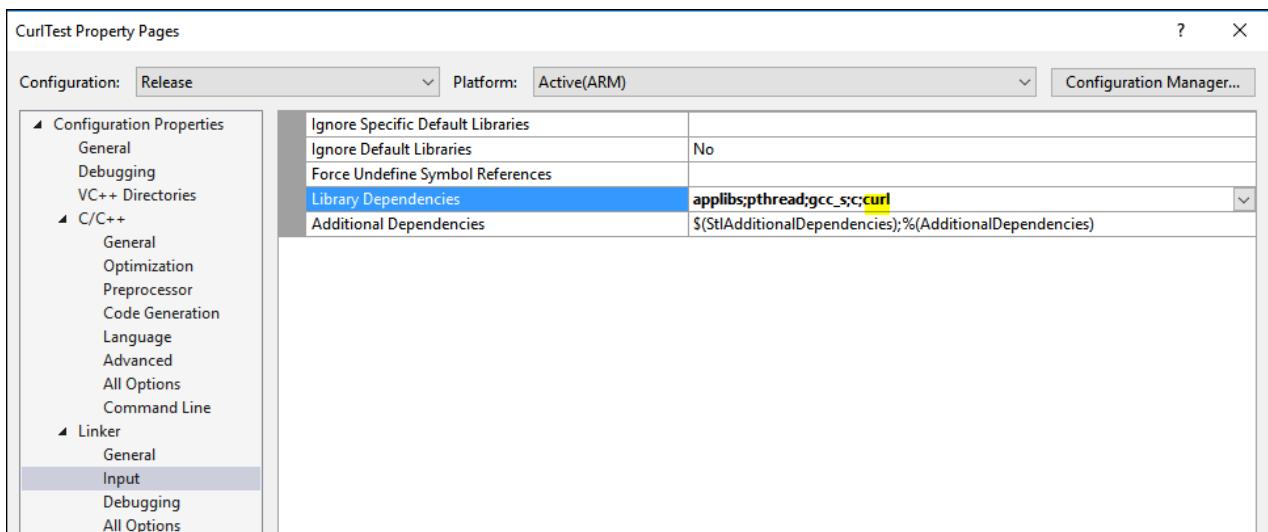
To use curl, include these header files in your application:

```
#include <applibs/storage.h> // required only if you supply a certificate in the image package
#include <tlsutils/deviceauth_curl.h> // required only for mutual authentication
#include <curl/curl.h>
```

The `storage.h` header file is required only if you supply one or more certificates in the application image package. The `deviceauth_curl.h` header is required for performing mutual authentication. Neither is required if you use only default certificates and do not implement mutual authentication.

### Linker

To link an application that uses curl, add a linker dependency on the curl library. In Visual Studio, select **Properties** for your project, then select **Linker>Input**. Add **curl** to the Library Dependencies list and click **Apply**.



## Application manifest

The **AllowedConnections** field of the application manifest must specify the hosts to which the application connects. It must also contain the name of each domain that the connection may encounter if redirected. For example, both microsoft.com and [www.microsoft.com](http://www.microsoft.com) are required for an application that connects to the Microsoft home page.

If the application uses mutual authentication, the **DeviceAuthentication** field of the manifest must include the Azure Sphere tenant ID. Device authentication certificates are issued only if the device's tenant ID matches the tenant ID in the application manifest. This restriction provides defense in depth: an application running on a device in a different tenant (say, that of a different customer or a rogue entity) cannot authenticate to the server.

During development, you can find the ID of the current Azure Sphere tenant by using the `azsphere tenant show-selected` command in an Azure Sphere Developer Command Prompt.

In the following example, the **AllowedConnections** field specifies that the application connects only to [www.example.com](http://www.example.com), and the **DeviceAuthentication** field specifies the Azure Sphere tenant ID and thus enables the application to use the device certificate for mutual authentication.

```

"Capabilities": {
  "AllowedConnections": [ "www.example.com" ],
  "Gpio": [],
  "Uart": [],
  "WifiConfig": false,
  "DeviceAuthentication": "00000000-0000-0000-0000-000000000000"
}

```

## Supported functionality

Libcurl for Azure Sphere supports only the HTTP and HTTPS protocols. In addition, the Azure Sphere OS does not support some functionality, such writable files (cookies) or UNIX sockets. Features that will not be supported in future libcurl releases, such as the **mprintf()** family, are not available.

Applications can call **curl\_version\_info** or check the return code from **curl\_easy\_setopt** to determine whether a particular feature is supported.

## Server authentication

Azure Sphere supports server authentication through libcurl. The server's certificate must be signed by a Certificate Authority (CA) that the device trusts. Several common CAs are built into the Azure Sphere device. In addition, you can add one or more certificates to your application image package.

For libcurl to authenticate a server, the application must provide the path to the CA file. This path is set by default to the directory that contains the [default CAs](#). If you use only the default CAs, no special code is required to provide this path.

To [use additional certificates](#), add them to your project and set their location in [CURLOPT\\_CAINFO](#). Use [Storage\\_GetAbsolutePathInImagePackage](#) to retrieve the absolute path to the certificates in the image package and then call [curl\\_easy\\_setopt](#):

```
char *path = Storage_GetAbsolutePathInImagePackage("certs/mycertificates.pem");
curl_easy_setopt(curl_handle, CURLOPT_CAINFO, path);
```

This code tells curl to trust any CAs that are defined in the mycertificates.pem file, in addition to CAs that are defined in the directory set in CURLOPT\_CAPATH.

If your application requires none of the default certificates, you can change [CURLOPT\\_CAPATH](#) to point to the folder that contains your certificates. For example, the following code tells curl to look only in the certs folder in the image:

```
char *path = Storage_GetAbsolutePathInImagePackage("certs");
curl_easy_setopt(curl_handle, CURLOPT_CAPATH, path);
```

## Default CA certificates

The Azure Sphere OS includes the following common CA certificates for your convenience in authenticating a server:

- BaltimoreCyberTrustRoot
- Equifax Secure
- GeoTrust Global
- GeoTrust Primary
- GlobalSign Root
- GoDaddy Class2 Certification Authority
- Thawte Primary Root
- VeriSign Class3 Public Primary Certification Authority G5

The curl library is configured by default to look for these certificates on the Azure Sphere device.

## Additional CA certificates

To use one or more CAs that are not among the defaults, you must add the certificates to your project and bundle them with your image package. Each certificate must be base-64 encoded. The simplest approach is to create a single file that contains all the additional certificates. The file must have the .pem filename extension. Add the certificate file to your project as a resource:

1. Create a certs folder in the project folder for your application. The project folder contains the .vcxproj or CMakeLists file for your application.
2. In the certs folder, create a text file with the .pem extension, copy each certificate into it, and save the file.
3. In Solution Explorer, right-click on the Resources folder, select **Add>New Item...**, and add the .pem file.
4. Right-click on the .pem file in Solution Explorer and select **Properties**.
5. In the General tab in the Properties dialog box, set Content to Yes.

The certificate file should now appear in the certs folder in the image package.

## Mutual authentication

Mutual authentication is a **BETA feature**.

Mutual authentication verifies that both the server and the client device are legitimate. It's a two-step process:

1. The application authenticates the server using a CA certificate, as described in [Server authentication](#).
2. The application presents an x509 client authentication certificate to the server so that the server can authenticate the device.

An application can set up the device-authentication side of mutual authentication in either of two ways:

- Configure the Azure Sphere **DeviceAuth\_CurlSslFunc** function as the SSL function that performs authentication.
- Create a custom SSL function that calls the Azure Sphere **DeviceAuth\_SslCtxFunc** function for authentication.

## Use DeviceAuth\_CurlSslFunc

The simplest way to perform device authentication is to configure **DeviceAuth\_CurlSslFunc** as the callback function for curl SSL authentication:

```
// Set DeviceAuth_CurlSslFunc to perform authentication
CURLcode err = curl_easy_setopt(_curl, CURLOPT_SSL_CTX_FUNCTION, DeviceAuth_CurlSslFunc);
if (err) {
    // Set ssl function failed
    return err;
}
```

**DeviceAuth\_CurlSslFunc** function retrieves the certificate chain for the current Azure Sphere tenant and sets up the curl connection to perform mutual authentication. If authentication fails, the function returns CURLE\_CERTPROBLEM.

## Write a custom callback function

If your application requires more detailed error handling, you can write a custom callback function instead of using **DeviceAuth\_CurlSslFunc**. Your custom callback function must call **DeviceAuth\_SslCtxFunc** to perform the authentication, but may also do other tasks related to authentication. **DeviceAuth\_SslCtxFunc** performs the same authentication tasks as **DeviceAuth\_CurlSslFunc**, but returns a value of the `DeviceAuthSslResult` enumeration, which provides more information about the failure than the standard CURLcode. For example:

```
static CURLcode MyCallback(CURL *curl, void **sslctx, void *userCtx)
{
    int err = DeviceAuth_SslCtxFunc(sslctx);
    Log_Debug("ssl func callback error %d\n", err);
    if (err) {
        // detailed error handling code goes here
    }
    return CURLE_OK;
}
...

err = curl_easy_setopt(curl, CURLOPT_SSL_CTX_FUNCTION, MyCallback);
if (err) {
    goto cleanupLabel;
}
```

## Additional tips for using curl

Here are some additional tips for using curl in an Azure Sphere application.

- If you plan to store page content in RAM or flash, keep in mind that storage on the Azure Sphere device is

[limited.](#)

- To ensure that curl follows redirects, add this to your code:

```
curl_easy_setopt(curl_handle, CURLOPT_FOLLOWLOCATION, 1L);
```

- To add verbose information about curl operations that might be helpful during debugging:

```
curl_easy_setopt(curl_handle, CURLOPT_VERBOSE, 1L);
```

- Some servers return errors if a request does not contain a user agent. To set a user agent:

```
curl_easy_setopt(curl_handle, CURLOPT_USERAGENT, "libcurl-agent/1.0");
```

# Memory use in high-level applications

7/31/2019 • 3 minutes to read

This topic provides details about memory use in high-level applications. See [Manage memory and latency considerations](#) for information about the memory available for real-time capable applications (RTApps).

High-level applications have access to the following memory and storage:

- 256 KiB RAM on the high-level core, reserved entirely for high-level application use. Up to 1 KiB of this space may be allocated for each shared buffer channel through which high-level applications and RTApps communicate.
- 1 MiB read-only flash memory, which is shared between the high-level and real-time cores.
- Read/write (mutable) storage, which persists when a device reboots. For information about mutable storage, see [Using storage on Azure Sphere](#).

## Determine flash memory usage

To determine your flash memory usage, consider only the size of the image package file, which includes the image metadata and application manifest in addition to the executable image. You don't need to account for the storage required by Microsoft-provided components such as the Azure Sphere OS or the run-time services and shared libraries that control peripherals and enable connection to an Azure IoT Hub. Likewise, you don't need to include the size of a full backup copy of your application or the components that enable failover or rollback in case of corruption or problems with over-the-air update.

During development and debugging, however, the size of the debugger does count against the limit. The debugger is automatically added by **azsphere device prep-debug** and removed by **azsphere device prep-field**. You can find the size of the debugger used by your SDK by looking for gdbserver.imagepackage in the DebugTools folder of the [Microsoft Azure Sphere SDK installation directory](#).

The **azsphere device sideload** command returns an error if the application image package, plus the debugger (if present), exceeds the 1 MiB total limit. The following commands, which upload applications to the Azure Sphere Security Service, also return an error if the image package exceeds 1MiB:

- **azsphere component image add ... --filepath filename**
- **azsphere component publish ... --filepath filename**
- **azsphere device link-feed ... --filepath filename**

The 256 KiB RAM limit applies to the application alone; you do not need to allow for RAM used by the debugger.

The available flash and RAM may increase, but will never decrease, for applications written for the current Azure Sphere chip (MT3620). Future Azure Sphere chips may have different limits.

## Out of memory conditions

If your application uses too much memory, the Azure Sphere OS terminates it with a SIGKILL signal. For example, in the debugger you'll see the following:

```
Child terminated with signal = 0x9 (SIGKILL)
```

The SIGKILL signal also occurs if a high-level application fails to handle the SIGTERM request; see [Lifecycle of an application](#) for details.

## Determine run-time application memory usage

You can get information about your application's memory usage during debugging with Visual Studio by issuing commands to the Visual Studio MI Debug Engine.

1. Open a Command Window in Visual Studio by selecting **View > Other Windows > Command Window**.
2. Pause the application.
3. Type the following command in the Command Window prompt:

```
Debug.MIDebugExec info proc status
```

This command returns the equivalent of `proc/self/stat` in Linux.

The following sample shows information from a sample Azure Sphere application. Note the `VmPeak` and `VmSize` entries, which list the peak and average virtual memory used by the application:

```
>Debug.MIDebugExec info proc status
process 101
Name: app
Umask: 0022
State: t (tracing stop)
Tgid: 101
Ngid: 0
Pid: 101
PPid: 98
TracerPid: 98
Uid: 1007 1007 1007 1007
Gid: 1007 1007 1007 1007
FDSize: 32
Groups: 5 10
NStgid: 101
NSpid: 101
NSpgid: 101
NSSid: 0
VmPeak: 1728 kB
VmSize: 1728 kB
VmLck: 0 kB
VmPin: 0 kB
VmHWM: 100 kB
VmRSS: 100 kB
RssAnon: 100 kB
RssFile: 0 kB
RssShmem: 0 kB
VmData: 76 kB
VmStk: 100 kB
VmExe: 40 kB
VmLib: 1508 kB
VmPTE: 6 kB
VmPMD: 0 kB
VmSwap: 0 kB
Threads: 1
SigQ: 1/55
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: 000000200001000
SigCgt: 000000000004000
CapInh: 0000000000000000
CapPrm: 0000000000000000
CapEff: 0000000000000000
CapBnd: 0000003fffffff
CapAmb: 0000000000000000

Speculation_Store_Bypass: unknown
Cpus_allowed: 1
Cpus_allowed_list: 0
voluntary_ctxt_switches: 5914
nonvoluntary_ctxt_switches: 380
```

# Manage system time and the RTC on Azure Sphere

5/30/2019 • 2 minutes to read

## BETA feature

The RTC (real-time clock) is used to keep time on an Azure Sphere device when the device loses power and has no access to a network connection after the device reboots. This allows the device maintain time during a power loss even if it doesn't have access to an NTP server.

If you set the system time, it does not persist when the device loses power. To persist the time during power loss, you must call the Applibs function [clock\\_systohc](#). When `clock_systohc` is called, the system time is pushed to the RTC.

## RTC requirements

Applications that use the RTC must enable beta APIs, include the appropriate header files, and add RTC settings to the [application manifest](#).

### Enable beta APIs

Complete the steps in [beta API features](#) to enable your app to use beta APIs for Azure Sphere.

### Header files

Include the `rtc` header in your project:

```
#include <applibs\rtc.h>
```

### Application manifest settings

To use the RTC and standard clock APIs, you must add the `SystemTime` application capability to the application manifest and then set the value to `true`. The [Azure Sphere application manifest](#) topic has more details about the application manifest.

```
{
  "SchemaVersion": 1,
  "Name" : "Mt3620App3_RTC",
  "ComponentId" : "bb267cbd-4d2a-4937-8dd8-3603f48cb8f6",
  "EntryPoint": "/bin/app",
  "CmdArgs": [],
  "Capabilities": {
    "AllowedConnections": [],
    "AllowedTcpServerPorts": [],
    "AllowedUdpServerPorts": [],
    "Gpio": [],
    "Uart": [],
    "WifiConfig": false,
    "NetworkConfig": false,
    "SystemTime": true
  }
}
```

## Get the system time

To get the system time, call the standard `clock_gettime` function.

## Set the system time

To set the system time, call the standard `clock_settime` function.

## Synchronize the system time with the RTC

When the system time is set, it does not persist when the device loses power. To persist the time during power loss, call the Applibs function `clock_systohc`. When `clock_systohc` is called the system time is pushed to the RTC.

## Configure the NTP client service

The NTP client service is enabled by default. If you set the system time while the NTP client service is enabled, it will overwrite the UTC time when the device has internet connectivity. You can [disable the NTP client service](#); however, this can cause OTA updates on the device to fail if the difference between the system time and the NTP server time is too great.

## Set the time zone

The system time and the RTC time are stored in GMT/UTC. You can change the time zone used by your application by calling the `setenv` function to update the [TZ environment variable](#), and then calling the `tzset` function.

The Azure Sphere OS supports some, but not all, possible formats for the [TZ environment variable](#):

- You can set the current time zone with or without Daylight Saving Time (DST). Examples: "EST+5", "EST+5EDT". This value is positive if the local time zone is west of the Prime Meridian and negative if it is east.
- You can't specify the date and time when DST should come into effect.
- You can't set the offset in minutes or seconds, only in hours.
- You can't specify a timezone file/database.

To maintain the timezone settings during a power loss, you can use [mutable storage](#) to store the timezone in persistent storage and then recall the setting when the device reboots.

## System time sample

The [System Time sample](#) shows how to manage the system time and use the hardware RTC. The sample application sets the system time and then uses the `clock_systohc` function to synchronize the system time with the RTC.

# Using storage on Azure Sphere

5/30/2019 • 3 minutes to read

This topic describes how to use storage on an Azure Sphere device. Azure Sphere provides two types of storage, read-only flash storage and mutable storage.

Read-only storage is used to store application image packages on a device so the contents can't be modified without [updating the application](#). This can include any data such as user interface assets, static configuration data, binary resources including firmware images used to update external MCUs, or initialization data for mutable storage. [Memory available for applications](#) provides additional details about the amount of storage available.

Mutable storage stores data that persists when a device reboots. For example, if you want to [manage system time](#) using the local time zone, you can store the time zone settings in mutable storage. Some other examples are settings a user can modify, or downloaded configuration data. The [mutable storage sample](#) shows how to use mutable storage in an application.

## Using read-only storage

### BETA feature

You can use these Applibs functions to manage read-only storage. For an example that uses these functions see [Connect to web services using curl](#).

- [Storage\\_OpenFileInImagePackage](#)
- [Storage\\_GetAbsolutePathInImagePackage](#)

### Read-only storage requirements

Applications that use read-only storage must enable beta APIs and include the appropriate header files.

#### Enable beta APIs

Complete the steps in [beta API features](#) to enable your app to use beta APIs for Azure Sphere.

#### Header files

Include the storage and unistd headers in your project:

```
#include <applibs/storage.h>
```

### Add a file to an image package

To include a file in an image package for read-only storage you can add the file to your project as a resource:

1. In Solution Explorer, right-click on the Resources folder, select **Add>New Item...**, and add the files.
2. Right-click on the file in Solution Explorer and select Properties.
3. In the General tab in the Properties dialog box, set Content to Yes.

The file should now appear in the image package.

## Using mutable storage

When you configure mutable storage for your application, it is assigned to the component ID of the application and can't be accessed by an application that has a different component ID. If the component ID of the application

changes, the new application will not have access to the mutable storage of the previous application.

If you delete an application from a device, the mutable storage assigned to the application is also deleted. If the same application is then loaded back onto the device, the mutable storage will be empty. However, if you update the application without deleting it, the mutable storage contents are maintained.

The **azsphere device sideload show-quota** command displays the amount of mutable storage currently in use.

You can use these Applibs functions to manage mutable storage data:

- [Storage\\_OpenMutableFile](#)
- [Storage\\_DeleteMutableFile](#)

### Mutable storage requirements

Applications that use mutable storage must include the appropriate header files and add mutable storage settings to the [application manifest](#).

#### Header files

Include the storage and unistd headers in your project:

```
#include <applibs/storage.h>
```

#### Application manifest

To use the APIs in this topic, you must add the `MutableStorage` capability to the [application manifest](#) and then set the `SizeKB` field. The `SizeKB` field is an integer that specifies the size of your mutable storage in kilobytes. The maximum value is 64 and the storage is allocated according to the erase block size of the device. The allocation is done by rounding up the `SizeKB` value to the next block size if the value isn't a whole multiple of the block size of the device.

#### NOTE

The MT3620 has an erase block size of 8 KB, so any values that are not multiples of 8 will be rounded up. For example, if you specify 12 KB in the 'MutableStorage' capability, you will receive 16 KB on an MT3620.

In the example below, the `MutableStorage` storage capability is added to the application manifest with a size of 8 KB.

```
{
  "SchemaVersion": 1,
  "Name" : "Mt3620App_Mutable_Storage",
  "ComponentId" : "9f4fee77-0c2c-4433-827b-e778024a04c3",
  "EntryPoint": "/bin/app",
  "CmdArgs": [],
  "Capabilities": {
    "AllowedConnections": [],
    "AllowedTcpServerPorts": [],
    "AllowedUdpServerPorts": [],
    "MutableStorage": { "SizeKB": 8 },
    "Gpio": [],
    "Uart": [],
    "WifiConfig": false,
    "NetworkConfig": false,
    "SystemTime": false
  }
}
```

### Write persistent data

To write data to persistent storage, start by calling the Applibs function [Storage\\_OpenMutableFile](#) to retrieve a file descriptor for a persistent data file. Next call the `write` function to write the data to the persistent data file. If the amount of data you attempt to write exceeds your mutable storage allocation, the write function might succeed; however, the only data written will be the portion that doesn't exceed the storage allocation. To ensure all the data is written you must check the return value of the `write` function call.

#### **Read persistent data**

To read data from persistent storage call [Storage\\_OpenMutableFile](#) to retrieve a file descriptor for the persistent data file, and then call the `read` function to read the data.

#### **Delete persistent data**

To delete data from persistent storage call [Storage\\_DeleteMutableFile](#).

# Communicate with a real-time capable application

6/20/2019 • 2 minutes to read

Azure Sphere supports inter-core communication between high-level applications and real-time capable applications (RTApp). High-level applications can use the applibs [Application\\_Socket](#) function to send and receive data when communicating with an RTApp. Application\_Socket retrieves a file descriptor to a socket that is used to communicate with the RTApp. While the file descriptor is open, you can perform operations on it, such as send/recv.

The [IntercoreComms samples](#) demonstrate how to use a high-level application to communicate with an RTApp on an MT3620. [Communicate with a high-level application](#) describes programming considerations for the RTApp.

## Requirements

High-level applications that use inter-app communication must enable beta APIs, include the appropriate header files, and add application settings to the [application manifest](#).

### Enable Beta APIs

Complete the steps in [beta API features](#) to enable your app to use beta APIs for Azure Sphere.

### Header files

```
#include <sys/socket.h>
#include <applibs/application.h>
```

### App manifest settings

To communicate with an RTApp, both applications need to include the AllowedApplicationConnections capability in the [application manifest](#):

- The high-level application must list the component ID of the real-time capable application in the AllowedApplicationConnections capability.
- The real-time capable application must list the component ID of the high-level application in the AllowedApplicationConnections capability.

```
"AllowedApplicationConnections": [ "005180BC-402F-4CB3-A662-72937DBCDE47" ]
```

## Open the socket

Before you communicate with an RTApp, you must open a socket by calling the [Application\\_Socket](#) function. The file descriptor returned by this function call is used to perform send/recv operations on the socket.

## Send data

You can send a message to an RTApp by calling the POSIX send() function. The maximum message size is 1 KB.

## Receive data

You can receive a message from an RTApp by called the POSIX recv() function.

## Close the socket

To close the socket, call the POSIX `close()` function.

# How to manually build and load a high-level application

5/30/2019 • 8 minutes to read

This section shows how to compile, link, package, and debug an application without using Visual Studio. To build your application you will need to find the correct compilation tools, headers, and libraries—collectively called the *sysroot*—on your PC. The Azure Sphere SDK ships with multiple sysroots so that applications can target different API sets, as described in [Beta API features](#). The sysroots are installed in the Azure Sphere SDK installation folder (C:\Program Files (x86)\Microsoft Azure Sphere SDK) under Sysroots.

The instructions that follow use the GPIO\_HighLevelApp sample application as an example. By default, the source and header files required to build the GPIO\_HighLevelApp sample are in the GPIO\_HighLevelApp folder for the project.

## Compile the application

1. Open an Azure Sphere Developer Command Prompt.
2. Create a folder and copy the source, header, and application manifest files that are required for the application to that folder. For the GPIO\_HighLevelApp sample, the following files are required:
  - main.c
  - applibs\_versions.h
  - epoll\_timerfd\_utilities.h
  - epoll\_timerfd\_utilities.c
  - mt3620\_rdb.h
  - app\_manifest.json
3. Run **gcc** to compile the application. Make sure you change the names of the source and output files in the `-x` and `-o` options, and replace *sysroot* in the search (`-I`) and Sysroots paths with the sysroot that the application uses.

For the GPIO\_HighLevelApp sample, compile the main.c and epoll\_timerfd\_utilities.c source files. This sample uses only production APIs, so the commands specify *sysroot* "2".

```
gcc.exe -c -x c main.c -I "C:\Program Files (x86)\Microsoft Azure Sphere SDK\SysRoots\2\usr\include" -g2 -gdwarf-2 -o "main.o" -Wall -O0 -fno-strict-aliasing -fno-omit-frame-pointer -D "_POSIX_C_SOURCE" -fno-exceptions -std=c11 -march=armv7ve -mthumb -mfpu=neon -mfloat-abi=hard -mcpu=cortex-a7 --sysroot="C:\Program Files (x86)\Microsoft Azure Sphere SDK\SysRoots\2\"
```

```
gcc.exe -c -x c epoll_timerfd_utilities.c -I "C:\Program Files (x86)\Microsoft Azure Sphere SDK\SysRoots\2\usr\include" -g2 -gdwarf-2 -o "epoll_timerfd_utilities.o" -Wall -O0 -fno-strict-aliasing -fno-omit-frame-pointer -D "_POSIX_C_SOURCE" -fno-exceptions -std=c11 -march=armv7ve -mthumb -mfpu=neon -mfloat-abi=hard -mcpu=cortex-a7 --sysroot="C:\Program Files (x86)\Microsoft Azure Sphere SDK\SysRoots\2\"
```

The example explicitly sets **gcc** compile options to match the Visual Studio options that are preconfigured by the Azure Sphere developer tools. The following table summarizes the **gcc** options on the compile command line:

OPTION	DESCRIPTION
-c	Compiles but doesn't link the program.
-x c	Explicitly identifies the source language as C.
-I	Adds the specified directory to the list of directories to be searched for header files during preprocessing.
-g2	Generates level 2 (default) debugging information.
-gdwarf-2	Generates debugging information in DWARF format version 2.
-o main.o	Explicitly specifies main.o as the output file.
-Wall	Enables all warnings.
-O0	Sets code optimization level 0 to reduce compilation time and make debugging produce the expected results. Level 0 is the default.
-fno-strict-aliasing	Does not allow the compiler to assume the strictest C-language aliasing rules. This option disables optimizations based on the type of expressions.
-fno-omit-frame-pointer	Stores the frame pointer in a register whether or not it is required.
-D "_POSIX_C_SOURCE"	Defines _POSIX_C_SOURCE as a macro.
-fno-exceptions	Disables exception handling.
-std=c11	Uses the 2011 revision of the ISO C standard.
-march=armv7ve	Specifies the armv7 architecture with virtualization exceptions.
-mthumb	Generates code that executes in Thumb state.
-mfpu=neon	Specifies NEON-vfpv3 as the floating-point hardware.
-mfloat-abi=hard	Specifies the 'hard' floating-point ABI. The 'hard' ABI allows generation of floating-point instructions and uses FPU-specific calling conventions.
-mcpu=cortex-a7	Specifies the name of the target ARM processor.
--sysroot= <i>dir</i>	Uses <i>dir</i> as the logical root directory for headers and libraries.

## Link the image

After compilation succeeds, run **gcc** again to link the image. As in the previous example, the command that follows specifies sysroot 2.

```
gcc.exe -o "Blink.out" --sysroot="C:\Program Files (x86)\Microsoft Azure Sphere SDK\SysRoots\2" -Wl,--no-undefined -nodefaultlibs -B "C:\Program Files (x86)\Microsoft Azure Sphere SDK\SysRoots\2\tools\gcc" -march=armv7ve -mcpu=cortex-a7 -mthumb -mfpu=neon -mfloat-abi=hard main.o epoll_timerfd_utilities.o -lapplibs -lpthread -lgcc_s -lc
```

As with the compiler options, the example explicitly sets **gcc** linker options to match the Visual Studio options that

are preconfigured by the Azure Sphere developer tools. The following table summarizes the **gcc** options on the link command line:

>

OPTION	DESCRIPTION
<code>-o filename</code>	Specifies the name of the output file.
<code>--sysroot=dir</code>	Uses <i>dir</i> as the logical root directory for headers and libraries.
<code>-Wl,--no-undefined</code>	Passes --no-undefined as a linker option, so that the linker reports any references that are unresolved after linking.
<code>-nodefaultlibs</code>	Does not use the standard system libraries when linking. Only the libraries you specify are passed to the linker.
<code>-B path</code>	Specifies the path to the gcc compiler.
<code>-march=armv7ve</code>	Specifies the armv7 architecture with virtualization exceptions.
<code>-mcpu=cortex-a7</code>	Specifies the name of the target ARM processor.
<code>-mthumb</code>	Generates code that executes in Thumb state.
<code>-mfpu=neon</code>	Specifies NEON-vfpv3 as the floating-point hardware.
<code>-mfloat-abi=hard</code>	Specifies the 'hard' floating-point ABI. The 'hard' ABI allows generation of floating-point instructions and uses FPU-specific calling conventions.
<code>-lapplibs</code>	Searches the applibs library during linking.
<code>-lpthread</code>	Searches the pthread library during linking.
<code>-lgcc_s</code>	Searches the gcc_s library during linking.
<code>-lc</code>	Searches the C library during linking.

## Package the application image

1. Create an `approot\bin` folder in your working folder and copy `blink.out` into it. In this example, the copied file is named `app`. Also, copy the `app_manifest.json` file for the application into this folder.

```
mkdir approot\bin  
copy Blink.out approot\bin\app  
copy app_manifest.json approot
```

2. Run **uuidgen** to generate a UUID for the package. This utility is installed in the Windows Kits folder.

```
"C:\Program Files (x86)\Windows Kits\10\bin\10.0.17134.0\x64\uuidgen.exe"
```

The **uuidgen** program returns a UUID in the form `nnnnnnnn-nnnn-nnnn-nnnnnnnnnnnn`.

3. Copy the `app_manifest.json` file from the Blink sample application to the `approot` directory, and open it in a text editor. In the `app_manifest.json` file:

- Set **Name** to "app"
- Set **ComponentId** to the UUID you created in the previous step

```
{
  "SchemaVersion": 1,
  "Name": "GPIO_HighLevelApp",
  "ComponentId": "dc7f135c-6074-4d49-aa3a-160e4eed884f",
  "EntryPoint": "/bin/app",
  "CmdArgs": [],
  "Capabilities": {
    "Gpio": [ "$SAMPLE_BUTTON_1", "$SAMPLE_LED" ]
  },
  "ApplicationType": "Default"
}
```

#### 4. Run the **azsphere image package-application** command to package the image:

```
azsphere image package-application --input approot --output manual.imagepackage --sysroot 2 --verbose --
hardwaredefinition "../../../Hardware/mt3620_rdb"
```

The --input flag specifies the Windows path to the folder that represents the root of the filesystem for the image package. The --output flag specifies the Windows path to the output image package. The --sysroot flag specifies a string that identifies the sysroot that the application was compiled with. The --verbose flag requests verbose output. The --hardwaredefinition value provides the relative path to the hardware definition for the MT3620 RDB.

This command modifies the app\_manifest.json file in the approot folder by adding the **TargetApplicationRuntimeVersion** field. If the application uses Beta APIs, the command also adds the **TargetBetaApis** field.

## Sideload and debug

Make sure your device has the appDevelopment capability so you can sideload the application and ensure that the debugging server is present. Use the [azsphere device prep-debug](#) command if necessary.

#### 1. If your device is already running an application, delete the application:

```
azsphere device sideload delete
```

#### 2. Run the **azsphere device sideload deploy** command to sideload the application onto the device and start it:

```
azsphere device sideload deploy --imagepackage manual.imagepackage
```

You should see LED L1 start to blink.

To debug the application, stop it and then restart it with the --debug option:

```
azsphere device sideload stop --componentid <ComponentId>
```

```
azsphere device sideload start --debug --componentid <ComponentId>
```

You should see:

```
<ComponentId>
App state : debugging
GDB port   : 2345
Output port : 2342

Command completed successfully in 00:00:00.9121174.
```

3. Open a command prompt and use any Windows terminal client to read the output stream from the process. Specify 192.168.35.2 as the IP address and 2342 as the port.

Windows terminal clients include PuTTY, Tera Term and many others. Windows includes the Windows Telnet client as an optional feature. To enable the Windows Telnet client, open **Control Panel** and click **Programs**. In **Programs and Features**, click **Turn Windows features on or off**. Scroll down to **Telnet client**, click the check box, and then click **OK**.

4. In the Azure Sphere Developer Command Prompt window, start the **gdb** command-line debugger:

```
arm-poky-linux-musleabi-gdb.exe Blink.out
```

Issue whatever **gdb** commands you choose. For example:

```
target remote 192.168.35.2:2345  
break main  
c
```

The `target` command specifies remote debugging to IP address 192.168.35.2 on port 2345. The `break` and `c` commands set a breakpoint upon entry to `main()` and then continue execution after the breakpoint, respectively. Numerous sources of [documentation](#) are available for **gdb**.

## Compile and build with CMake

CMake can be used in Visual Studio and from the command line to build applications. CMake is added to Visual Studio as an optional component. To add Cmake:

1. Open the installer for your installed version of Visual Studio, and select **Modify**.
2. Select the **Individual components** tab.
3. In the **Compilers, build tools, and runtimes** section, select **Visual C++ tools for CMake**.
4. Select **Modify** to install the components.

Using CMake to build projects in Visual Studio will vary by the version you are using. In Visual Studio 2017, CMake has a separate menu with various options for build. Visual Studio 2019 does not have a separate menu, but CMake build options are added to the context menus when you right-click build files in Solution Explorer.

If you are using Visual Studio 2019, build 16.04 or higher is required to use CMake to build Azure Sphere applications.

You can view the CMake command generated by Visual Studio by enabling verbose output.

1. In Visual Studio 2019, under the **Tools** menu, select **Options**.
2. In the options for CMake, select **Enable verbose CMake output**.
3. Click **OK**.

To build the CMake sample application from the command line, assuming that Visual Studio is installed in the default location and that your path includes CMake and Ninja:

```
cmake.exe ^
-G "Ninja" ^
-DCMAKE_INSTALL_PREFIX:PATH="."
-DCMAKE_TOOLCHAIN_FILE="C:/Program Files (x86)/Microsoft Azure Sphere
SDK/CMakeFiles/AzureSphereToolchain.cmake"
-DAZURE_SPHERE_TARGET_APPLICATION_RUNTIME_VERSION="2"
-DAZURE_SPHERE_TARGET_BETA_APIS=""
-DAZURE_SPHERE_TARGET_HARDWARE_DEFINITION_DIRECTORY="%CD%/.//./././Hardware/mt3620_rdb"
-DAZURE_SPHERE_TARGET_HARDWARE_DEFINITION="sample_hardware.json"
--no-warn-unused-cli
-DCMAKE_BUILD_TYPE="Debug"
-DCMAKE_MAKE_PROGRAM="ninja.exe"
```

Use the path to the location where you installed the samples.

## CMake parameters

-DCMAKE\_TOOLCHAIN\_FILE

This is a standard CMake flag to configure the use of a compiler toolchain. The example above passes the default location of the toolchain file that is installed with the Azure Sphere SDK.

-DAZURE\_SPHERE\_TARGET\_APPLICATION\_RUNTIME\_VERSION

This flag is provided by the Azure Sphere toolchain file. The value is the required Azure Sphere runtime version. In the example the value for the runtime version is 1.

-DAZURE\_SPHERE\_TARGET\_BETA\_APIS

This flag is provided by the Azure Sphere toolchain file. The value is the desired Azure Sphere Beta APIs to use such as "2+Beta1905". Here it is not set.

-DCMAKE\_BUILD\_TYPE

This flag indicates the build type. Possible values are debug and release.

-DCMAKE\_MAKE\_PROGRAM

This flag indicates the make program to use, and in the sample is set to use ninja.exe.

-DAZURE\_SPHERE\_TARGET\_HARDWARE\_DEFINITION\_DIRECTORY

Sets the location of the hardware definition folder.

-DAZURE\_SPHERE\_TARGET\_HARDWARE\_DEFINITION

Indicates the hardware definition to use for the application.

Once you have compiled and built your application with CMake, the steps to deploy and debug are the same as those found in Sideload and debug.

# Initialization and termination

11/15/2018 • 2 minutes to read

At start-up, every Azure Sphere application should perform some initialization:

- Register a SIGTERM handler for termination requests. The Azure Sphere device OS sends the SIGTERM termination signal to indicate that that application must exit. As part of its initialization code, the application should register a handler for such requests. For example:

```
#include <signal.h>
...
// Register a SIGTERM handler for termination requests
struct sigaction action;
memset(&action, 0, sizeof(struct sigaction));
action.sa_handler = TerminationHandler;
sigaction(SIGTERM, &action, NULL);
```

In the termination handler, the application can perform whatever shutdown tasks it requires. Termination handlers must be POSIX async-signal-safe. The sample programs exit on error as well as on receipt of the termination signal. Therefore, these programs simply set a Boolean in the termination handler and then perform cleanup and shutdown tasks after exiting the main loop.

- Initialize handles for GPIO peripherals.
- If the application uses Azure IoT Hub, connect to the IoT client and register callback functions for IoT features such as cloud-to-device messages, device twin status, and direct method calls.

At termination, the application should close peripherals, destroy handles, and free allocated memory.

# Pass arguments to an application

8/13/2018 • 2 minutes to read

Azure Sphere applications receive command-line arguments through the [application manifest](#). The application must follow the C language conventions by specifying argc and argv as arguments to **main()**.

In the application manifest, the command-line arguments appear as an array in the "CmdArgs" field. This example passes four arguments to the application:

```
{  
  "SchemaVersion": 1,  
  "Name": "MyTestApp",  
  "ComponentId": "072c9364-61d4-4303-86e0-b0f883c7ada2",  
  "EntryPoint": "/bin/app",  
  "CmdArgs": ["-m", "262144", "-t", "1"],  
  "TargetApplicationRuntimeVersion": 1,  
  "Capabilities": {  
    ...  
  }  
}
```

# Periodic tasks

9/13/2018 • 2 minutes to read

After start-up, the application is always running; it should not run, exit, and then restart. Therefore, your application should perform its ongoing, operational tasks in a continuous loop until it receives a termination signal, as the samples do.

As an application runs, it should call **Networking\_IsNetworkingReady()** before each use of networking. This function checks that Internet connectivity is available and that the Azure Sphere device clock is synchronized with a set of common network time protocol (NTP) servers. **Networking\_IsNetworkingReady()** is defined in networking.h. If networking is not available, the application must handle the error gracefully—for example, by waiting until the network is available or by queuing requests to try later. The application must not fail or become unresponsive if networking is unavailable.

# Asynchronous events and concurrency

8/13/2018 • 2 minutes to read

The Azure Sphere platform supports several common POSIX and Linux mechanisms to handle asynchronous events and concurrency.

The samples demonstrate how to use the epoll and timerfd system functions to pause execution until one of several types of events occurs. For example, the UART sample pauses until the device receives data over UART or until a button is pressed to send data over UART.

For applications that require threads, the Azure Sphere platform supports POSIX pthreads. It is the responsibility of the application to ensure thread-safe execution. Application calls to some applibs functions are thread-safe, but others are not, as indicated in the header files. If the header file does not mention thread safety, you should assume that the relevant function or library is not thread-safe.

# Use a system timer as a watchdog

11/15/2018 • 2 minutes to read

An Azure Sphere application can use a system timer as a watchdog to cause the OS to terminate and restart that application if it becomes unresponsive. When the watchdog expires, it raises a signal that the application doesn't handle, which in turn causes the OS to terminate the application. After termination, the OS automatically restarts the application.

To use a watchdog timer:

- Define the timer
- Create and arm the timer
- Reset the timer regularly before it expires

To define the timer, create an **itimerspec** structure and set the interval and initial expiration to a fixed value, such as one second.

```
#include <time.h>

const struct itimerspec watchdogInterval = { { 1, 0 },{ 1, 0 } };
timer_t watchdogTimer;
```

Set a notification event, signal, and signal value for the watchdog, call **timer\_create** to create it, and call **timer\_settime** to arm it. In this example, `watchdogTimer` raises the SIGALRM event. The application doesn't handle the event, so the OS terminates the application.

```
void SetupWatchdog(void)
{
    struct sigevent alarmEvent;
    alarmEvent.sigev_notify = SIGEV_SIGNAL;
    alarmEvent.sigev_signo = SIGALRM;
    alarmEvent.sigev_value.sival_ptr = &watchdogTimer;

    int result = timer_create(CLOCK_MONOTONIC, &alarmEvent, &watchdogTimer);
    result = timer_settime(watchdogTimer, 0, &watchdogInterval, NULL);
}
```

Elsewhere in the application code, reset the watchdog periodically. One technique is to use a second timer, which has a period shorter than the `watchdogInterval`, to verify that the application is operating as expected and, if so, reset the watchdog timer.

```
// Must be called periodically
void ExtendWatchdogExpiry(void)
{
    //check that application is operating normally
    //if so, reset the watchdog
    timer_settime(watchdogTimer, 0, &watchdogInterval, NULL);
}
```

# Error handling and logging

9/13/2018 • 2 minutes to read

Most functions in the Azure Sphere custom application libraries (applibs) return -1 to indicate failure and zero or a positive value to indicate success. In case of failure, the function sets the value of the **errno** variable to the POSIX error that corresponds to the failure. Applications must include the errno.h header file, which defines this variable. The **errno** variable is global per thread.

Applications can log errors in the following ways:

- During debugging, use the **Log\_Debug()** or **Log\_DebugVarArgs()** function to write a debug message to the Device Output window in Visual Studio, when using the Azure Sphere SDK Preview for Visual Studio. The format for the message is the same as that for **printf**.
- During execution, send messages to an IoT Hub. See [Using Microsoft Azure IoT](#) for details.

# Overview of real-time capable application development

10/9/2019 • 2 minutes to read

Build real-time capable applications (RTApps) that run on Azure Sphere devices. Use of the real-time cores is currently a **Beta** feature.

## NOTE

For details about programming the M4 cores on the MT3620, see the [MT3620 datasheet](#) published by Mediatek. If the datasheet does not contain all the information you need, please email Avnet ([Azure.Sphere@avnet.com](mailto:Azure.Sphere@avnet.com)) to request the full datasheet.

## Develop and debug RTApps

TOPIC	DESCRIPTION
<a href="#">Develop and debug an RTApp</a>	Describes how to create and debug a new RTApp using Visual Studio
<a href="#">Port an existing application</a>	Describes how to port an existing bare-metal application to run on Azure Sphere
<a href="#">Manage memory and latency considerations</a>	Describes memory contention and latency issues that might occur in RTApps
<a href="#">Sideload an application</a>	Describes how to load an app for testing
<a href="#">Remove an application</a>	Describes how to remove your application from a device
<a href="#">Build and debug an RTApp from the command line</a>	Describes how to build, load, and debug an RTApp from the command line
<a href="#">Troubleshoot RTApps</a>	Provides troubleshooting suggestions for RTApp problems

## Add features to your RTApps

Add features to your RTApps. These topics are associated with [application samples](#).

TOPIC	DESCRIPTION
<a href="#">Use peripherals in RTApps</a>	Describes how to map peripherals such as GPIO and UART for use by an RTApp
<a href="#">Communicate with a high-level app</a>	Describes how to communicate with a high-level application on the Azure Sphere device

## Command line reference

- [Azure Sphere command-line reference](#)

# Develop and debug a real-time capable application

10/9/2019 • 3 minutes to read

You can use Visual Studio to develop and debug real-time capable applications (RTApps) in much the same way as high-level applications. The primary difference is that RTApps projects currently must use CMake, instead of vcxproj files.

## Create a new RTApp

The easiest way to create a new real-time capable application (RTApp) is to start with the [HelloWorld\\_RTApp\\_MT3620\\_BareMetal sample](#) and adjust the configuration to your project by following these steps:

1. Clone the [samples repo](#) if you haven't already done so. Copy the HelloWorld\_RTApp\_MT3620\_BareMetal folder and rename it for your project.
2. Using Visual Studio, open the folder.
3. In the CMakeLists.txt file, change the project name to the name of your new folder. For example:

```
PROJECT(NewRTApp C)
```

4. Write your code, using the Azure Sphere RTApp samples as guides. The following topics describe specific implementation scenarios:
  - [Use peripherals in a real-time capable application](#)
  - [Manage memory and latency considerations](#)
5. In the app\_manifest.json file:
  - Set `Name` to your project name,
  - Set `ApplicationType` to `"RealTimeCapable"`
  - Add any application-specific capabilities that your code requires, such as hardware resources or connections. If the RTApp communicates with a high-level app, add the component ID of the high-level application to the `AllowedApplicationConnections` capability.
6. If you want Visual Studio to deploy your RTApp alongside a high-level [partner app](#), add the component ID of the partner to the `partnerComponents` field of the `configurations` section of the launch.vs.json file:

```
"partnerComponents": [ "25025d2c-66da-4448-bae1-ac26fcdd3627" ]
```

## Build the RTApp

To build an RTApp using Visual Studio:

1. Connect your Azure Sphere device to your PC by USB.
2. Open a new Azure Sphere Developer Command Prompt with Administrator privileges and issue the following command:

```
azsphere device prep-debug --enablertcoredebugging
```

3. In Visual Studio, open the **File** menu, select **Open>CMake...** and navigate to the folder that contains the sample.
4. If CMake generation does not start automatically, select the CMakeLists.txt file.

5. In the Visual Studio Output window, the CMake output should show the messages

CMake generation started. and CMake generation finished.

6. On the CMake menu (if present), click Build All. If the menu is not present, open Solution Explorer, right-click the CMakeLists.txt file, and select Build. The output location of the Azure Sphere application appears in the Output window.

7. Press F5 to deploy the application.

By default, the RTApp is deployed to the first available real-time core on the device; you cannot currently specify a particular core. To find out which core the application is running on, use the **azsphere device sideload** command to stop and then restart the application. Supply the component ID for the application in the commands. For example:

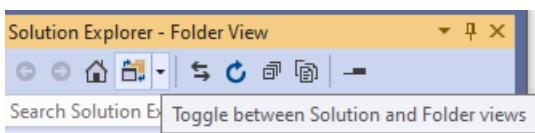
```
azsphere device sideload stop --componentid d3b80666-feaf-433a-b294-6a5846853b4a`  
d3b80666-feaf-433a-b294-6a5846853b4a: App state: stopped
```

```
azsphere device sideload start --componentid d3b80666-feaf-433a-b294-6a5846853b4a  
d3b80666-feaf-433a-b294-6a5846853b4a  
App state: running  
Core : 0  
  
Command completed successfully in 00:00:01.8273541.
```

## Visual Studio features for RTApps

Visual Studio provides Intellisense for Azure Sphere RTApps by gathering data from its CMake cache. Visual Studio updates the cache whenever the CMakeLists.txt or CMakeSettings.json file in the RTApp changes.

By default, Visual Studio uses the Folder view. If you prefer a logical view of the CMake project, you can change to the CMake Targets view. In Solution Explorer, select the folder toggle icon:



From the drop-down menu, select CMake Targets View.

## Log output from an RTApp

Each real-time core on the MT3620 has a dedicated UART that is intended for logging output. (The real-time cores can also access the ISU UARTs.) The MT3620 RDB exposes only the TX pin, which you can use to display logging output from the application. Other hardware may expose this differently, or not at all. To see the log output, set up your hardware to display this output as described in the [quickstart](#). The dedicated UART doesn't require the **uart** application manifest requirement; however, it shouldn't be used for purposes other than logging output for an RTApp.

## Visual Studio debugging

RTApps are debugged using GDB and OpenOCD, which are installed with the Azure Sphere SDK. Visual Studio uses information from the launch.vs.json file in your project to configure the debugging tools. If you start with one of the samples, you already have such a file. If this file is not present, Visual Studio creates it when you set Debug and Launch settings:

- In Solution Explorer, right-click the target application, select Debug and Launch settings, and then Launch

for Azure Sphere RTApps (gdb).

- Choose Select to create the launch.vs.json file for your application.
- In the launch.vs.json file, set the `project` value in the `"configurations"` field:

```
"project": "CMakeLists.txt"
```

Visual Studio sets up connections between the GDB server and OpenOCD so that you can use the standard Visual Studio debugging interface (F5, F6, F9 for breakpoints and so forth) on an RTApp, in the same way as on a high-level app.

## Troubleshooting

If you encounter problems, see [Troubleshooting real-time capable applications](#).

# Port an existing real-time capable application

6/20/2019 • 2 minutes to read

Most existing ARM Cortex-M4 applications can be ported to run as real-time capable applications (RTApps) on the MT3620, if they do not have specific hardware dependencies. You can also port applications that run with an existing real-time operating system (RTOS). Porting involves the following tasks:

1. Import the project into Visual Studio
2. Configure the project for the target processor
3. Configure device-specific parameters
4. Create ELF files

## Import a CMake project into Visual Studio

If your existing application uses CMake, you can import it by cloning the repository directly from within Visual Studio or by copying the source to your PC. Then, in Visual Studio, select **File>Open Folder** and navigate to the root folder that contains the CMakeLists.txt file for the application.

## Configure the project for the target processor

If the project is not already configured for the Azure Sphere real-time cores (ARM Cortex M4 processors on the MT3620), you will need to configure it.

1. Copy the launch.vs.json and CMakeSettings.json files from one of the Azure Sphere samples.
2. Edit the CMakeSettings.json file if necessary. By default, the project is configured to use the ARM GCC compilers that are supplied by Visual Studio. You can use a different compiler by changing the value of the ARM\_GNU\_PATH variable in CMakeSettings.json.

## Configure device-specific parameters

Existing code typically requires a few changes to run on Azure Sphere.

To modify your code to use Azure Sphere peripherals, see [use peripherals in a real-time capable application](#).

You can also add support to [communicate with a high-level application](#).

## Create ELF files

RTApp images for Azure Sphere must be created as ELF images, not as raw binaries. Visual Studio creates the application image in the correct form.

The entry point for the ELF file can be either:

- A pointer to the entry point function, if the least significant bit (LSB) of the address is set; this corresponds to the use of Thumb-2. In this case, the stack pointer is undefined.
- Pointer to a (SP, PC) dword pair, similar to a Cortex-M reset vector, if the LSB is 0.

Toolchains typically create an ELF image that includes debugging information. Because these files can be large, use the `arm-none-eabi-strip` utility to remove any unnecessary information from the ELF image before it is embedded into the image package. Keep the original image that contains the debug information, because it is useful when using GDB to debug.

If you are porting an existing CMake project from a different platform and are not using Visual Studio, you might need to modify the ELF image before you can deploy it. [Manage memory and latency considerations](#) provides details about ELF file layout and how Azure Sphere loads applications.

## Troubleshooting

If you encounter problems, see [Troubleshooting real-time capable applications](#).

# Use peripherals in a real-time capable application

10/9/2019 • 6 minutes to read

Real-time capable applications (RTApps) can map peripherals for their exclusive use. To use one or more peripherals in an RTApp:

- Add the peripheral to the [application manifest](#).
- Add an interrupt handler (if required) and other supporting code to the application.

## IMPORTANT

For hardware-specific information about how to program peripherals for an RTApp, see the documentation from your hardware manufacturer. For information about the MT3620, see the published [MT3620 Datasheet from Mediatek](#); if questions remain, you can request the MT3620 M4 Datasheet from Avnet by emailing Azure.Sphere@avnet.com.

All resources that an RTApp uses must be specified in the application manifest. Use the "AppManifestValue" that is defined for the resource in the JSON file for the target chip. For MT3620 hardware, use the mt3620.json file from the Azure Sphere Samples [hardware folder](#). Find the entry that defines the resource you want to use. For example, to use I2C on ISU0, the application manifest would specify "ISU0", which is defined as follows in the JSON file:

```
{"Name": "MT3620_ISU0_I2C", "Type": "I2cMaster", "MainCoreHeaderValue": "(0)", "AppManifestValue": "ISU0",  
"Comment": "MT3620 ISU 0 configured as I2C"},
```

Concurrently loaded applications cannot share resources; resource use is exclusive to a single application. The Azure Sphere OS performs several important tasks when it loads the application:

- Configures multiplexing, so that the pins associated with the peripheral are configured for the requested functionality.
- Sets up *core mapping*, which involves configuring firewalls so that the application has access to the registers associated with the requested peripherals.
- Checks the manifest and fails to load the application if it claims resources that have already been claimed by another app.

The remainder of this topic provides details about using specific peripherals.

## UARTs

The ISU UARTs on the Azure Sphere device are available for use by RTApps. The [UART\\_RTApp\\_MT3620\\_BareMetal sample](#) shows how to use interrupts to asynchronously send data to, and receive data from, an ISU UART. This RTApp is based on the [equivalent high-level sample application](#). Use this sample as a starting point for developing RTApps that use ISU UARTs.

To use an ISU UART, an RTApp must list it in the **Capabilities** section of the application manifest. Identify the UART by using the "AppManifestValue" that is defined for it in the JSON file for the target chip. For example, the following line reserves the UART on ISU0 on an MT3620 chip:

```
"Capabilities": {  
    "Uart": [ "ISU0" ]  
}
```

Your application code needs a way to identify the UART, its register base address, and its interrupt number, and must include an interrupt handler for the UART. You can find the base address and interrupt number for the UART in the manufacturer's hardware documentation.

### MT3620 support for URTs on the real-time cores

This section describes the UART options that apply for real-time cores on MT3620 hardware. For general information about MT3620 support for URTs, see [MT3620 support](#). For information about register base addresses, interrupt numbers, and similar details, request the [MT3620 M4 Datasheet](#).

Each real-time core has a dedicated UART, which is separate from the ISU URTs and is intended for logging. Because each such URT is dedicated for use only by the application on its core, the RTApp is not required to list it in the application manifest. The [UART\\_RTApp\\_MT3620\\_BareMetal sample](#) demonstrates how to use this URT in polling mode. RTApps are assigned to the first real-time core that is available; it is not currently possible to choose which real-time core—and thus which dedicated URT—an application uses. However, the output of the **azsphere device sideload start** command reports which core has been assigned to the application.

The mt3620-uart.c file in the [UART\\_RTApp\\_MT3620\\_BareMetal sample](#) shows how to set up and communicate over ISU URTs on the MT3620 RDB hardware. It provides functions that can be used to asynchronously write strings or print integers via a URT.

## GPIOs

An RTApp can use any of the GPIOs on the Azure Sphere device. The [GPIO\\_RTApp\\_MT3620\\_BareMetal sample](#) uses GPIOs to blink an LED and change the blink rate in response to a button press. It is based on the [high-level GPIO sample application](#). Use this sample as a starting point for your own applications.

To use a GPIO, an RTApp must list it in the **Capabilities** section of the application manifest. Identify the GPIO by using the "AppManifestValue" that is defined for it in the JSON file for the target hardware. For example, the following line reserves GPIOs 8 and 12:

```
"Capabilities": {  
    "Gpio": [ 8, 12 ]  
}
```

Your application code needs a way to identify the GPIO and its registers. You can find the register base address in the manufacturer's hardware documentation. The Azure Sphere GPIO samples use polling, rather than interrupts, for GPIOs. An RTApp can use interrupts to handle timers and check GPIO status from the interrupt handler.

### MT3620 support for GPIOs on the real-time cores

This section describes GPIO features that apply only for real-time cores on MT3620 hardware. For information about register base addresses, interrupt numbers, and similar details, request the [MT3620 M4 Datasheet](#).

On the MT3620, most GPIOs are mapped to real-time cores in blocks of four, starting with GPIO 0; however, ISU block 0 has five (GPIO26-GPIO30). As a result, RTApp GPIO requests may fail if a high-level application (or another RTApp) requests a GPIO in the same block. For example, if a high-level app requests GPIO 8 and an RTApp requests GPIO 9, the second app returns an error at deployment.

Before your RTApp can use a GPIO, it must initialize the block. Initialization is required only once for each block that the app uses.

Each LED on the MT3620 RDB maps to three GPIOs: one each for red, green, and blue channels.

The GPIO\_RTApp\_MT3620\_BareMetal sample includes MT3620-specific functions that configure a GPIO as input or output, and read or set the value of that GPIO respectively. These functions work in the same way as the Applibs GPIO functions that are available to high-level applications.

## General-purpose timers

Each real-time core on the MT3620 supports five general-purpose timers (GPT). GPT0, GPT1, and GPT3 are interrupt-based, and GPT2 and GPT4 are free-run timers. The [GPIO\\_RTApp\\_MT3620\\_BareMetal sample](#) uses GPT0 and GPT1, and the [ADC\\_RTApp\\_MT3620\\_BareMetal sample](#) uses GPT3.

### NOTE

GPT0 and GPT1 should use a 32 KHz clock source, but currently they do not run at the correct frequency. Consequently, applications that use these timers will get inaccurate timeouts. For example, a request for a 1-second delay could actually delay for 1.5 seconds. GPT3 has higher resolution, but if your scenario requires more than one timer, you may need to use GPT0 or GPT1. You can work around this issue by implementing a timer queue and running all the timers from GPT3.

## ADC

The [ADC\\_RTApp\\_MT3620\\_BareMetal sample](#) shows how to use an analog-to-digital converter in an RTApp.

To use an ADC, an RTApp must list it in the **Capabilities** section of the application manifest. For an RTApp that runs on the MT3620 chip, identify the ADC by using the "AppManifestValue" that is defined for it in mt3620.json:

```
"Capabilities": {  
    "Adc": [ "ADC-CONTROLLER-0" ]  
}
```

Your application code needs a way to identify the ADC, its control registers, and its channels. You can find the register base address in the manufacturer's hardware documentation.

### MT3620 support for ADCs on the real-time cores

This section describes the UART options that apply for real-time cores on MT3620 hardware. For information about register base addresses, interrupt numbers, and similar details, request the [MT3620 M4 Datasheet](#).

The mt3620-adc.c file in the ADC\_RTApp\_MT3620\_BareMetal sample shows how to set up and use an ADC on the MT3620 RDB hardware. It provides functions that can be used to enable and read data from an ADC channel.

# Manage memory and latency considerations

10/9/2019 • 7 minutes to read

This topic describes basic memory use and latency considerations for real-time applications that run on the MT3620 chip.

## NOTE

For more detail about memory configuration or DMA, see the published [MT3620 Datasheet from Mediatek](#); if questions remain, you can request the MT3620 M4 Datasheet from Avnet by emailing Azure.Sphere@avnet.com.

## Memory layout on the real-time cores

The following table summarizes the memory available on the real-time cores:

MEMORY TYPE	BASE ADDRESS
TCM	0x00100000
XIP flash	0x10000000
SYSRAM	0x22000000

Each real-time core has 192 KB of tightly-coupled memory (TCM), which is mapped in three banks of 64 KB starting at 0x00100000. TCM accesses are fast, but only the real-time core can access the memory. TCM cannot be shared with a high-level application or with a real-time capable application (RTApp) that runs on a different core.

Each real-time core also has 64 KB of SYSRAM, which is mapped starting at 0x22000000. The DMA controller can also target SYSRAM, so that peripherals can access it. Accesses to SYSRAM from the real-time core are slower than accesses to TCM. As with TCM, SYSRAM cannot be shared with another application.

Execute-in-place (XIP) flash memory is shared with high-level applications. A window into the XIP mapping of the flash is visible to each core at address 0x10000000. The OS configures the XIP mapping before it starts the application if the application's ELF file contains a segment that has the following properties:

- Load address (as specified in the VirtAddr column of the Program Header) is equal to 0x10000000
- File offset and size (as specified in the FileSiz and MemSiz fields in the Program Header) fit in the application's ELF file

If a program header with these properties is present in the application's ELF file, the XIP window will be positioned so that the segment is visible at 0x10000000. The file can have no more than one XIP segment, and it must point to 0x10000000; it cannot specify any other address.

## ELF deployment

RTApp images must be ELF files. The ELF image is wrapped in an Azure Sphere image package and deployed as an application. To load the application, the Azure Sphere OS starts an ELF loader that runs on the real-time core. The loader processes each LOAD segment in the ELF file and loads it into the type of memory indicated by the virtual address in the program header.

Use `arm-none-eabi-readelf.exe -S`, which is part of the GNU Arm Embedded Toolchain, to display the program headers for your application. The virtual address column (VirtAddr) that appears in the header indicates the destination address for the load segment. It does not mean that the processor itself performs any additional translation. The Azure Sphere ELF loader doesn't use the physical address (PhysAddr).

Consider this example:

Program Headers:							
Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
LOAD	0x0000098	0x00100000	0x00100000	0x00000	0x00e78	RW	0x8
LOAD	0x00000a0	0x10000000	0x10000000	0x03078	0x03078	RWE	0x10
LOAD	0x003118	0x00100e78	0x10003078	0x000f0	0x000f0	RW	0x4

- The segment at 0x00100000 is targeted at tightly-coupled memory (TCM). The loader either copies data from the image package into RAM or zero-initializes the TCM as required.
- The segment at 0x10000000 is mapped to the XIP window for the core. At run time, accesses to `0x10000000 + offset` are translated to `<address-of-XIP-segment-in-flash> + offset` when they leave the real-time core.
- The data segment at virtual address 0x00100e78 is mapped to TCM.

### ELF runtime considerations

The ELF loader performs some of the tasks that a raw binary (or chained bootloader) would perform at start-up. Specifically, it zero-initializes block-started-by-symbol (BSS) data and copies initialized but mutable data from read-only flash into RAM, according to the program headers. The application then starts and runs its own initialization functions. In most cases, changes to existing applications aren't required. Zeroing the BSS data in the application is unnecessary but harmless, because the loader has already zeroed the memory.

Copying mutable data from flash to RAM may in some circumstances result in problems, depending on how the ELF file is laid out. The ELF loader processes the program headers sequentially, without changing the overall layout of the segments in the file. It then maps not only the XIP segment itself to 0x10000000, but also any subsequent segments in order. If the segments in the ELF file are in sequential order without any alignment or gaps, OS startup code can use pointer arithmetic to find the start of the data segment. If the ELF file has a different layout, however, pointer arithmetic does not result in the correct address, so application startup code must not try to copy the data section. This may cause problems if the application or RTOS uses a chained bootloader or needs to set up a stack canary before zeroing BSS or initializing mutable data.

### Memory targets

You can target code at TCM, XIP flash, or SYSRAM by editing the linker.ld script for your application. The [Azure Sphere sample applications](#) run from TCM, but the linker.ld script file for each application describes how to target XIP flash instead. As the following example shows, you can change a sample to run on XIP by aliasing CODE\_REGION and RODATA\_REGION to FLASH instead of the default TCM:

```
REGION_ALIAS("CODE_REGION", FLASH);
REGION_ALIAS("RODATA_REGION", FLASH);
```

To determine whether a compiled application runs from TCM or XIP flash, use `arm-none-eabi-readelf.exe`, which is part of the GNU Arm Embedded Toolchain. Run it on the .out file, which is in the same directory as the image package, and specify the `-l` (lower-case L) flag to see where the code and read-only data have been placed. Code and read-only data that are in flash memory are loaded at address 0x10000000; code and data in TCM are loaded in the TCM region.

The following example shows an application that runs from flash memory.

```

arm-none-eabi-readelf.exe -l UART_RTApp_MT3620_BareMetal.out

Elf file type is EXEC (Executable file)
Entry point 0x10000000
There are 2 program headers, starting at offset 52

Program Headers:
Type          Offset      VirtAddr      PhysAddr      FileSiz MemSiz Flg Align
LOAD          0x000074 0x00100000 0x00100000 0x00284 0x003c0 RW 0x4
LOAD          0x000300 0x10000000 0x10000000 0x013b9 0x013b9 R E 0x10

Section to Segment mapping:
Segment Sections...
00      .data .bss
01      .text .rodata

```

## Vector table location

On ARMv7-M devices, the vector table must be aligned on a power-of-two boundary that is at least 128 bytes and no less than the size of the table, as noted in the [ARMv7-M Architecture Reference Manual](#). Each I/O M4 core on the MT3620 supports 100 external interrupts. Therefore, including the stack pointer and 15 standard exceptions, the table has 116 4-byte entries, for a total size of 464 bytes, which rounds up to 512 bytes.

When the code is run from XIP flash, the vector table must be placed at 0x10000000 and must be aligned on a 32-byte boundary within the ELF file. When the code is not run from XIP flash, the table is typically placed at the start of TCM0, which is 0x100000. In either case, to ensure that the table's virtual address is correctly aligned, put the vector table in a dedicated section and set CODE\_REGION to the appropriate address.

The MT3620 BareMetal samples in the Azure Sphere Samples repository show how to do this. The declaration of the vector table in main.c sets its `section` attribute to `.vector_table`. The linker script aliases CODE\_REGION to the start of either TCM or XIP, and the ALIGN attribute sets the alignment of the text section within the ELF file as follows:

```

SECTIONS
{
    .text : ALIGN(32) {
        KEEP(*(.vector_table))
        *(.text)
    } >CODE_REGION
    ...
}

```

## Real-time and latency considerations

RTApps and high-level applications contend for access to flash memory, even if they don't communicate with each other. As a result, RTApps that are running from XIP flash may encounter high and unpredictable latency. Writes to flash, such as during an update, may involve latency spikes up to several hundred milliseconds. Depending on your application's requirements, you can manage this in several ways:

- Put all code and data in TCM. Code that runs from TCM is not vulnerable to contention for flash.
- Split code into critical and non-critical sections, and run the non-critical code from flash. Code that has real-time requirements, such as a watchdog timer, should not have to run when other code is accessing the flash. [Memory targets](#) describes how to target XIP flash instead of TCM.
- Use cache. An application can use the lowest 32KB of TCM as XIP cache. This approach does not provide hard real-time guarantees in the event of a cache miss, but improves typical performance without requiring you to move all the code into RAM. Refer to the MT3620 M4 Datasheet for information about XIP cache

configuration.

# Communicate with a high-level application

6/20/2019 • 2 minutes to read

Real-time capable applications (RTApps) communicate with high-level applications through a ring buffer that is mapped to a shared region of memory.

The [IntercoreComms samples](#) show how RTApps can communicate with high-level applications. Use the IntercoreComms\_RTApp\_MT3620\_BareMetal sample as a starting point for developing your own RTApp communication code. See [Communicate with a real-time capable application](#) for information about the code required in a high-level application.

## IMPORTANT

For information about register addresses and other hardware-specific details, request the MT3620 M4 Programmer's Guide by emailing Azure.Sphere@avnet.com.

## Operation

From the perspective of the RTApp, communication with a high-level application involves the following basic operations:

- Reading set-up information from the mailbox it shares with the high-level core on the MT3620
- Reading and writing data to shared memory

### Initialize buffer sizes and pointers

The RTApp receives information about the ring buffer via three messages in the mailbox that the real-time and high-level cores share. Each message contains a command and data. The first two messages provide the base addresses of the read and write buffers that the applications will use to communicate; the third message contains an "end-of-transmission" code.

COMMAND VALUE	DATA
0xba5e0001	Inbound (read) buffer descriptor
0xba5e0002	Outbound (write) buffer descriptor
0xba5e0003	End of transmission

The RTApp calculates the size of the read and write buffers and sets up pointers to the initial position of each buffer based on the data in the buffer descriptors.

See the IntercoreComms\_RTApp\_MT3620\_BareMetal sample for additional details about how to use the mailbox.

### Read from and write to the buffers

After initialization, the RTApp can read from and write to the buffers. Because communication is implemented with a ring buffer, both reads and writes can wrap around to the beginning of the buffer. The message header is always aligned on 16-byte boundary. The message content is left to the individual applications. Currently, messages can be at most 1 KB in size; the underlying buffer is somewhat larger to account for headers and other overhead.

After each read or write operation to the buffer, the RTApp uses the shared mailbox to notify the high-level app that it has received or transmitted a message.

## Application manifest settings

The application manifests for both the RTApp and the high-level app must list the component IDs of the applications with which they communicate. To configure this setting, add the `AllowedApplicationConnections` capability to the application manifest, and then add each component ID to the capability. The [Azure Sphere application manifest](#) topic has more details. Here's an excerpt from an application manifest that configures an RTApp to communicate with one other application.

```
"AllowedApplicationConnections": [ "25025d2c-66da-4448-bae1-ac26fcdd3627" ]
```

# Build and debug an RTApp from the command line

6/20/2019 • 2 minutes to read

This section shows how to build, sideload, and debug a real-time capable application (RTApp) without using Visual Studio. The instructions that follow use the [HelloWorld\\_RTApp\\_MT3620\\_BareMetal sample](#) as an example.

The instructions in this section assume that CMake.exe and ninja.exe are included in your PATH environment variable.

1. Open an Azure Sphere Developer Command Prompt.
2. Gather the source and application manifest files along with the CMake files that are required for the application. For the HelloWorld\_RTApp\_MT3620\_BareMetal sample, the following files are required:
  - main.c
  - app\_manifest.json
  - linker.ld
  - CMakeLists.txt
  - CMakeSettings.json
3. Run cmake.exe to compile and link the application. Use the `-S` option to specify the source directory. See [How to manually build and load an application](#) for information about the Azure Sphere build variables.

```
cmake.exe -G "Ninja" -S . -DCMAKE_INSTALL_PREFIX:PATH=".\\install\\ARM-Release-2Beta1905" -  
DCMAKE_TOOLCHAIN_FILE:STRING="C:/Program Files (x86)/Microsoft Azure Sphere  
SDK/CMakeFiles/AzureSphereRTCoreToolchain.cmake" -  
DAZURE_SPHERE_TARGET_APPLICATION_RUNTIME_VERSION:STRING="2" -  
DAZURE_SPHERE_TARGET_BETA_APIS:STRING="Beta1905" -DARM_GNU_PATH:STRING="C:/Program Files (x86)/Microsoft  
Visual Studio/2019/Enterprise/Linux/gcc_arm" --no-warn-unused-cli -DCMAKE_BUILD_TYPE="Release" -  
DCMAKE_MAKE_PROGRAM="C:\\PROGRAM FILES (X86)\\MICROSOFT VISUAL  
STUDIO\\2019\\ENTERPRISE\\COMMON7\\IDE\\COMMONEXTENSIONS\\MICROSOFT\\CMAKE\\Ninja\\ninja.exe"
```

4. Run `ninja.exe` from the output folder to create the image package.

## Sideload and debug

1. Open a new Azure Sphere Developer Command Prompt with Administrator privileges and run **azsphere device prep-debug** for the RT core to prepare the device for application development and install the USB drivers required for debugging:

```
azsphere device prep-debug --enabltcoredebugging
```

Close the window after the command completes.

2. Set up your hardware to display log output from the dedicated UART, as described in the [quickstart](#).
3. Deploy the image package that `ninja.exe` created:

```
azsphere device sideload deploy --imagepackage <package-name>
```

4. To start the application for debugging:

```
azsphere device sideload start --componentid <component id> --debug
```

To get the component ID, use `azsphere image show --filepath <imagepackage file>`

5. Navigate to the Openocd folder for the sysroot that the application was built with. For the HelloWorld\_RTApp\_MT3620\_BareMetal sample used here, the sysroot is 2+Beta1905. The sysroots are installed in the Azure Sphere SDK installation folder:

```
cd C:\Program Files (x86)\Microsoft Azure Sphere SDK\Sysroots\2+Beta1905\tools\openocd
```

6. Run `openocd.exe` as follows:

```
openocd.exe -f mt3620-rdb-ftdi.cfg -f mt3620-io0.cfg -c use_swd -c "gdb_memory_map disable" -c "gdb_breakpoint_override hard" -c init -c "targets io0" -c halt -c "targets"
```

7. In a different Azure Sphere Developer Command Prompt window, navigate to the folder that contains the application .out file and start arm-none-eabi-gdb, which is part of the GNU Arm Embedded Toolchain:

```
arm-none-eabi-gdb HelloWorld_RTApp_MT3620_BareMetal.out
```

8. The OpenOCD server provides a GDB server interface on :3333. Set the target for debugging.

```
target remote :3333
```

9. Issue whatever gdb commands you choose.

## Troubleshooting

If you encounter problems, see [Troubleshooting real-time capable applications](#).

# Troubleshooting real-time capable applications

5/30/2019 • 2 minutes to read

Here are some troubleshooting steps for problems that may occur during development of real-time capable applications (RTApps).

## Debugging issues

Debugging may fail for the following reasons:

- Windows Update replaces the USB driver for the real-time cores with a generic driver. To restore debugging functionality for the real-time cores, open an Azure Sphere Developer Command Prompt with administrator privileges and re-enable real-time core debugging. The required USB driver is installed when you run `azsphere device prep-debug --enablertcoredebugging`.
- The debugging settings file contains errors. In this situation, the debugger may fail to start. If you have edited the debugger settings, make sure the syntax of the file is correct.
- If you try to debug a high-level app and two RTApps at once, Visual Studio fails to work properly. To work around this problem, close all instances of Visual Studio and debug only one RTApp at a time.

## CMake issues

You may encounter the following problems when using CMake with an RTApp:

- The first time you open a CMake project in Visual Studio, CMakeLists generation automatically runs and sets the component ID for the application. If you then delete the component ID—for example, because the application is in a GitHub repo and you've done a hard reset—the build fails. To resolve this problem, regenerate the CMake cache in Visual Studio.
- The CMakeLists.txt startup item frequently regenerates. The CMakeLists task runs whenever Visual Studio regenerates the CMake cache. As a result, it may run often.
- CTRL-F5 fails to start the RTApp in Visual Studio 2017, although F5 successfully starts the application for debugging. This is a known problem that will be fixed in a future release.
- Visual Studio hangs when you build an RTApp with CMake. To avoid this problem use Visual Studio 2019 update 1 or later.

As a general note, if errors occur when you're working with CMake, try regenerating the CMake cache. This often resolves problems.

# Sideload an application image package for testing

5/30/2019 • 4 minutes to read

When you build your application with Visual Studio, the Visual Studio Extension for Azure Sphere Preview packages the application image for you. If you have direct access to an Azure Sphere device, the tools can also load it onto the Azure Sphere device, start it, and enable debugging.

You must first enable the application development capability for the device and add the device to a [device group](#) that does not support over-the-air application update. Assigning devices to such a group ensures that your sideloaded applications will not be overwritten by OTA deployments. To prepare your device, you can either:

- [Create a device group](#), assign your device to it, and [add the capability](#)

OR

- Use the **azsphere device prep-debug** command as described here.

The **azsphere device prep-debug** command has the following form:

```
azsphere device prep-debug --devicegroupid <devicegroup> --enablertcoredebugging
```

This command:

- Queries the cloud to get the appDevelopment capability for the device. Only the user identity with which the device was claimed can use the capability.
- Assigns the device to the specified device group. If the device group is omitted, the command assigns the device to a Microsoft-created device group that does not allow OTA application deployments.
- Applies the appDevelopment capability to the device.
- Loads the debugging server on the device to enable debugging of high-level applications.
- If the **--enablertcoredebugging** flag is included, the command will load necessary USB drivers and the debugging server for the real time cores. Note that because **--enablertcoredebugging** installs USB drivers, the command must be run as administrator.

For example, to enable appDevelopment capability for high level applications and real time applications:

```
azsphere device prep-debug

Getting device capability configuration for application development.
Downloading device capability configuration for device ID <device ID>.
Successfully downloaded device capability configuration.
Successfully wrote device capability configuration file 'C:\Users\user\AppData\Local\Temp\tmpD732.tmp'.
Setting device group ID <group ID> for device with ID <device ID>.
Successfully disabled over-the-air updates.
Enabling application development capability on attached device.
Applying device capability configuration to device.
Successfully applied device capability configuration to device.
The device is rebooting.
Installing debugging server to device.
Installation started.
Application development capability enabled.
Successfully set up device <device ID> for application development, and disabled over-the-air updates.
Command completed successfully in 00:00:17.1861625.
```

For example, to enable appDevelopment capability for both high level applications and real time applications, and debugging for real time applications:

```
azsphere device prep-debug --enablertcoredebugging

Installing USB drivers to enable real-time core debugging.
Drivers installed for all attached devices
Getting device capability configuration for application development.
Downloading device capability configuration for device ID <device ID>.
Successfully downloaded device capability configuration.
Successfully wrote device capability configuration file 'C:\Users\user\AppData\Local\Temp\tmpD732.tmp'.
Setting device group ID <group ID> for device with ID <device ID>.
Successfully disabled over-the-air updates.
Enabling application development capability on attached device.
Applying device capability configuration to device.
Successfully applied device capability configuration to device.
The device is rebooting.
Installing debugging server to device.
Installation started.
Application development capability enabled.
Successfully set up device <device ID> for application development, and disabled over-the-air updates.
Command completed successfully in 00:00:17.1861625.
```

You can now sideload the image.

## Sideload and debug an image packaged by Visual Studio

When you use Visual Studio to build an application, the Visual Studio Extension for Azure Sphere Preview creates an image package that contains the application. If you then begin debugging, Visual Studio also deletes any existing applications that aren't marked as partner applications from the device and sideloads the image package onto the device.

To sideload the image package outside Visual Studio, use the **azsphere device sideload** command. In an Azure Sphere Developer Command Prompt, issue the following commands to delete all existing applications on the device, sideload the new image package, and start the application:

```
azsphere device sideload delete
azsphere device sideload deploy --imagepackage <imagepackagepath>
```

Replace *imagepackagepath* with the path to the image package. Depending on the project configuration, you can find the image package in the bin\ARM\Debug or bin\ARM\Release subfolder of the application's Visual Studio project folder. By default, the command starts the application after deploying it.

To debug the application, add the -m flag to the **azsphere device sideload deploy** command to suppress automatic start, and then start the application for debugging:

```
azsphere device sideload deploy --manualstart --imagepackage <imagepackagepath>
azsphere device sideload start --debug --componentid <ComponentID>
```

The command displays the output and debug ports for high-level applications:

```
Output Port: 2342
GDB Port: 2345
```

To stop the application and delete it, use the **azsphere device sideload stop** and **azsphere device sideload delete** commands with the --componentid option, as follows:

```
azsphere device sideload stop --componentid <ComponentID>
azsphere device sideload delete --componentid <ComponentID>
```

To stop and delete all applications on the device, omit the --componentid option.

## Sideload more than one application

Applications that are related to another application can be marked as partner applications. There are many situations where you would load multiple applications such as a high-level application that provides communication for a real-time capable application. Marking the applications are partners will prevent one from being deleted when the second is loaded.

To mark an application as a partner using Visual Studio:

1. In Visual Studio, under the **Project** menu select **Properties**.
2. Under **Configuration Properties** select **Debugging**.
3. Set the value of **Partner Components** to the GUID of the partner application.
4. Click OK.

To mark an application as a partner using CMake:

1. Edit the launch.vs.json file for the application.
2. Add the following line:

```
"partnerComponents": [ "<PartnerGUID>" ]
```

Substitute the GUID of the partner application for `<PartnerGUID>` and save the file.

# Remove an application

10/9/2019 • 2 minutes to read

You can remove applications that are installed on your device in two ways:

- In Visual Studio, right-click on the project in **Solution Explorer** and then select **Remove the application from device**. This applies only for high-level applications.
- From an Azure Sphere Developer Command Prompt, issue the following command to delete all apps from the device:

```
azsphere device sideload delete
```

To remove only one application, use the **--componentid** parameter to specify the component ID of the application to remove. Use the **azsphere device sideload show-status** command to show all currently loaded applications with their component IDs.

After you delete the applications, the device will not run any application until you sideload an application or trigger deployment. See [When do updates occur](#) for details.

# Use Azure IoT with Azure Sphere

6/20/2019 • 2 minutes to read

Your Azure Sphere devices can communicate with the Azure IoT by using an [Azure IoT Hub](#) or by using [Azure IoT Central](#).

No matter which you use, you'll need an Azure subscription. If your organization does not already have a subscription, you can set up a [free trial](#).

## IMPORTANT

Although you can create an Azure subscription for no charge, the sign-up process requires you to enter a credit card number.

## Authenticate your Azure Sphere tenant

After you have an Azure subscription, you must establish trust between Azure Sphere and your Azure IoT Central application or IoT hub. You do this by downloading a certificate authority (CA) certificate from the Azure Sphere Security Service and validating it using a code generated by Azure IoT Hub or Azure IoT Central. The validation process authenticates your Azure Sphere tenant. You need to perform the validation only once.

The authentication process is slightly different for an IoT hub and Azure IoT Central:

- [Set up an IoT Hub](#)
- [Set up Azure IoT Central](#)

## Next Steps

You can now run the [Azure IoT sample application](#) from GitHub, which connects to either Azure IoT Central or your Azure IoT hub.

# Set up an Azure IoT Hub for Azure Sphere

5/30/2019 • 4 minutes to read

To use your Azure Sphere devices with the IoT, you can set up an [Azure IoT Hub](#) to work with your Azure Sphere tenant. After you have completed the tasks in this section, any device that is claimed by your Azure Sphere tenant will be automatically enrolled in your IoT hub when it first comes online and connects to the [Device Provisioning Service \(DPS\)](#). Therefore, you only need to complete these steps once.

## Prerequisites

The steps in this section assume that:

- Your Azure Sphere device is connected to your PC by USB
- You have an Azure subscription

## Overview

Setting up an Azure IoT Hub to work with Azure Sphere devices requires a multi-step process:

1. Create an Azure IoT Hub and DPS in your Azure subscription.
2. Download the authentication CA certificate for your Azure Sphere tenant from the Azure Sphere Security Service.
3. Upload the CA certificate to DPS to tell it that you own all devices whose certificates are signed by this CA. In return, the DPS presents a challenge code.
4. Generate and download a validation certificate from the Azure Sphere Security Service, which signs the challenge code. Upload the validation certificate to prove to DPS that you own the CA.
5. Create a device enrollment group, which will enroll any newly claimed Azure Sphere device whose certificate is signed by the validated tenant CA.

### IMPORTANT

Although you can create an Azure subscription for no charge, the sign-up process requires you to enter a credit card number. Azure provides several levels of subscription service. By default, the Standard Tier, which requires a monthly service charge, is selected when you create an IoT hub. To avoid a monthly charge, select the Free tier. The Free tier includes the services required to use your device with an IoT hub, including the Device Twin.

If you choose to test an Azure IoT-based application that uses the Device Provisioning Service (DPS), be aware that DPS charges \$0.10 per 1000 transactions (ten U.S. cents per one thousand transactions). We expect that the free credit that applies to many new subscriptions will cover any DPS charges, but we recommend that you check the details of your subscription agreement.

## Step 1. Create an IoT hub and DPS and link them

Create an [Azure IoT Hub and DPS](#) and link them. Do not clean up the resources created in that Quickstart.

## Step 2. Download the tenant authentication CA certificate

1. Open an Azure Sphere Developer Command Prompt, which is available in the **Start** menu under **Azure Sphere**.

2. Sign in with the user for your Azure Active Directory:

```
azsphere login
```

3. Download the Certificate Authority (CA) certificate for your Azure Sphere tenant:

```
azsphere tenant download-CA-certificate --output CAcertificate.cer
```

The output file must have the .cer extension.

## Step 3. Upload the tenant CA certificate to DPS and generate a verification code

1. Open the [Azure portal](#) and navigate to the DPS you created in [Step 1](#).
2. Open **Certificates** from the menu. You might have to scroll down to find it.

AzureSphereDocs-DPS  
Device Provisioning Service

Resource group (change)  
AzureSphereDocResources

Status  
Active

Location  
West US

Subscription (change)  
Free Trial

Subscription ID  
8504fd31-51e7-445c-be42-8a10750ef9da

Service endpoint  
AzureSphereDocs-DPS.azure-devices-prov...

Global device endpoint  
global.azure-devices-provisioning.net

ID Scope  
One0002161E

Pricing and scale tier  
S1

Quick Links

- Azure IoT Hub Device Provisioning Service Documentation
- Learn more about IoT Hub Device Provisioning Service

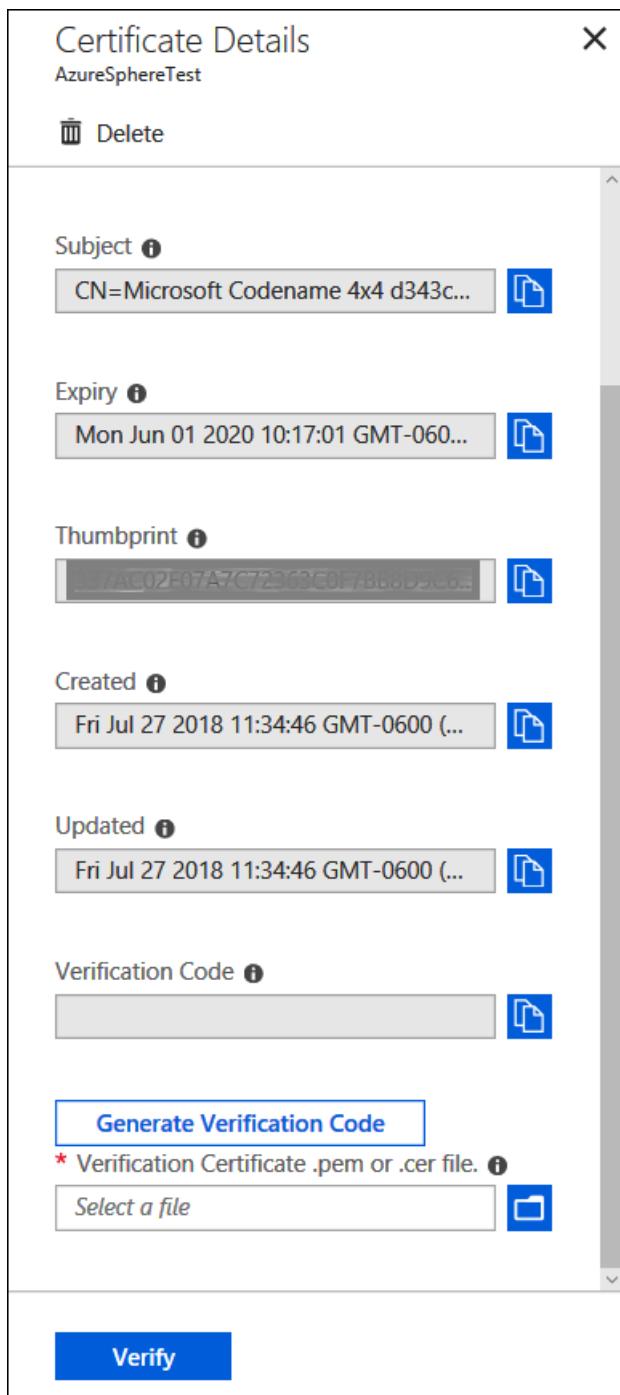
3. Click **Add** to add a new certificate and enter a friendly display name for your certificate.
4. Browse to the .cer file you downloaded in [Step 2](#). Click **Upload**.
5. After you are notified that the certificate uploaded successfully, click **Save**.

The screenshot shows the AzureSphereDocs-DPS - Certificates blade. On the left, there's a sidebar with options like Access control (IAM), Tags, Properties, Locks, Automation script, Quick Start, Shared access policies, Linked IoT hubs, and Certificates (which is selected and highlighted in blue). The main area displays a table of certificates. One row is selected, showing 'AzureSphere...' with a green checkmark icon, 'Verified' status, and an expiry date of 'Thu Jul 23 2020'. A modal window titled 'Add Certificate' is overlaid on the page, asking for a 'Certificate Name' (set to 'AzureSphereTest') and a 'Certificate .pem or .cer file' (set to 'CAcertificate.cer'). A 'Save' button is at the bottom of the modal.

6. The **Certificate Explorer** list shows your certificates. Note that the **STATUS** of the certificate you just created is *Unverified*. Click on this certificate.

The screenshot shows the same AzureSphereDocs-DPS - Certificates blade. The sidebar and main table structure are identical to the previous screenshot. The table now lists two entries: 'AzureSphereTena...' with a green checkmark and 'Verified' status, and 'AzureSphereTest' with a grey circle and 'Unverified' status. The 'Unverified' entry is highlighted with a red border. The 'Save' button from the previous modal is no longer visible.

7. In **Certificate Details**, click **Generate Verification Code**. The DPS creates a **Verification Code** that you can use to validate the certificate ownership. Copy the code to your clipboard for use in the next step.



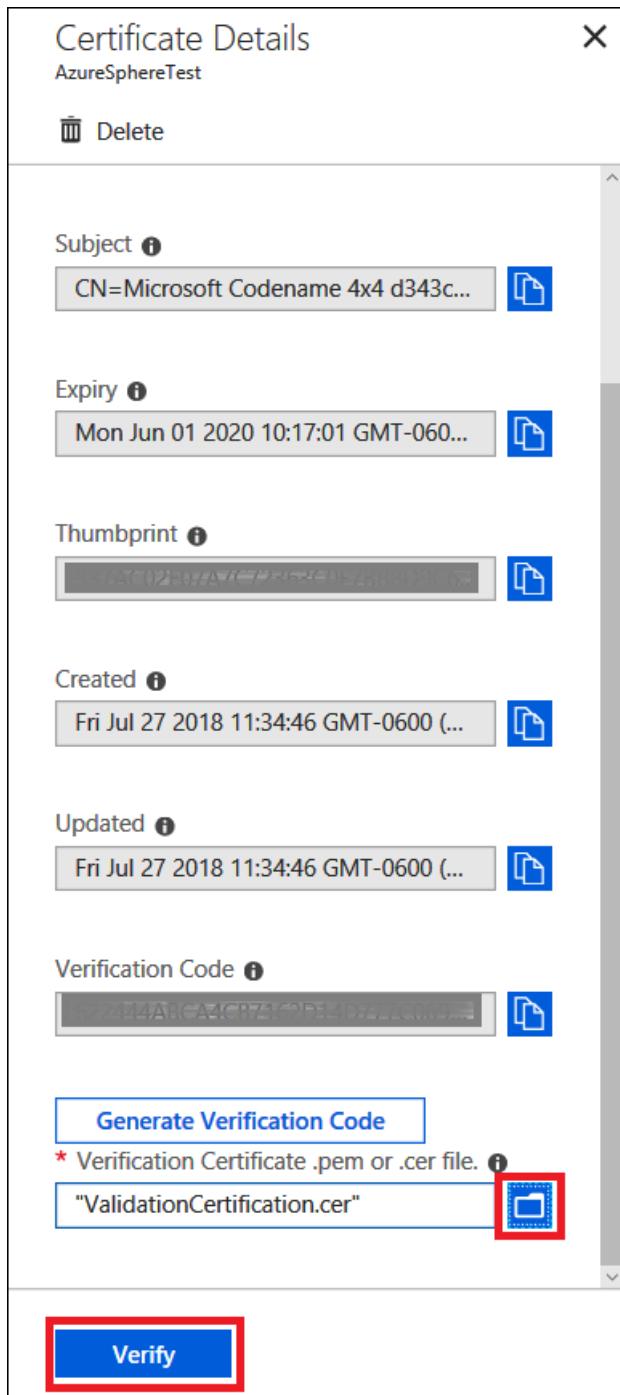
## Step 4. Verify the tenant CA certificate

1. Return to the Azure Sphere Developer Command Prompt. Download a validation certificate that proves that you own the tenant CA certificate. The Replace *code* in the command with the verification code from the previous step.

```
azsphere tenant download-validation-certificate --output ValidationCertification.cer --verificationcode<code>
```

The Azure Sphere Security Service signs the validation certificate with the verification code to prove to DPS that you own the CA.

2. Return to the Azure Portal to upload the validation certificate to DPS. In **Certificate Details** on the Azure portal, use the *File Explorer* icon next to the **Verification Certificate .pem or .cer file** field to upload the signed verification certificate. When the certificate is successfully uploaded, click **Verify**.



3. The **STATUS** of your certificate changes to **Verified** in the **Certificate Explorer** list. Click **Refresh** if it does not update automatically.

## Step 5. Use the validation certificate to add your device to an enrollment group

1. In the Azure portal, select **Manage enrollments** and then click **Add enrollment group**.
2. In the Add Enrollment Group pane, create a name for your enrollment group, select CA Certificate as the **Certificate type**, and select the certificate that you validated in the previous step.

 Add Enrollment Group

**Save**

\* Group name  
test-enrollment-group

Attestation Type ⓘ  
**Certificate** Symmetric Key

Certificate Type ⓘ  
**CA Certificate** Intermediate Certificate

Primary Certificate ⓘ  
AzureSphereTest

Secondary Certificate ⓘ  
No certificate selected

Select how you want to assign devices to hubs ⓘ  
Evenly weighted distribution

Select the IoT hubs this group can be assigned to: ⓘ  
POTestHub.azure-devices.net

**Link a new IoT hub**

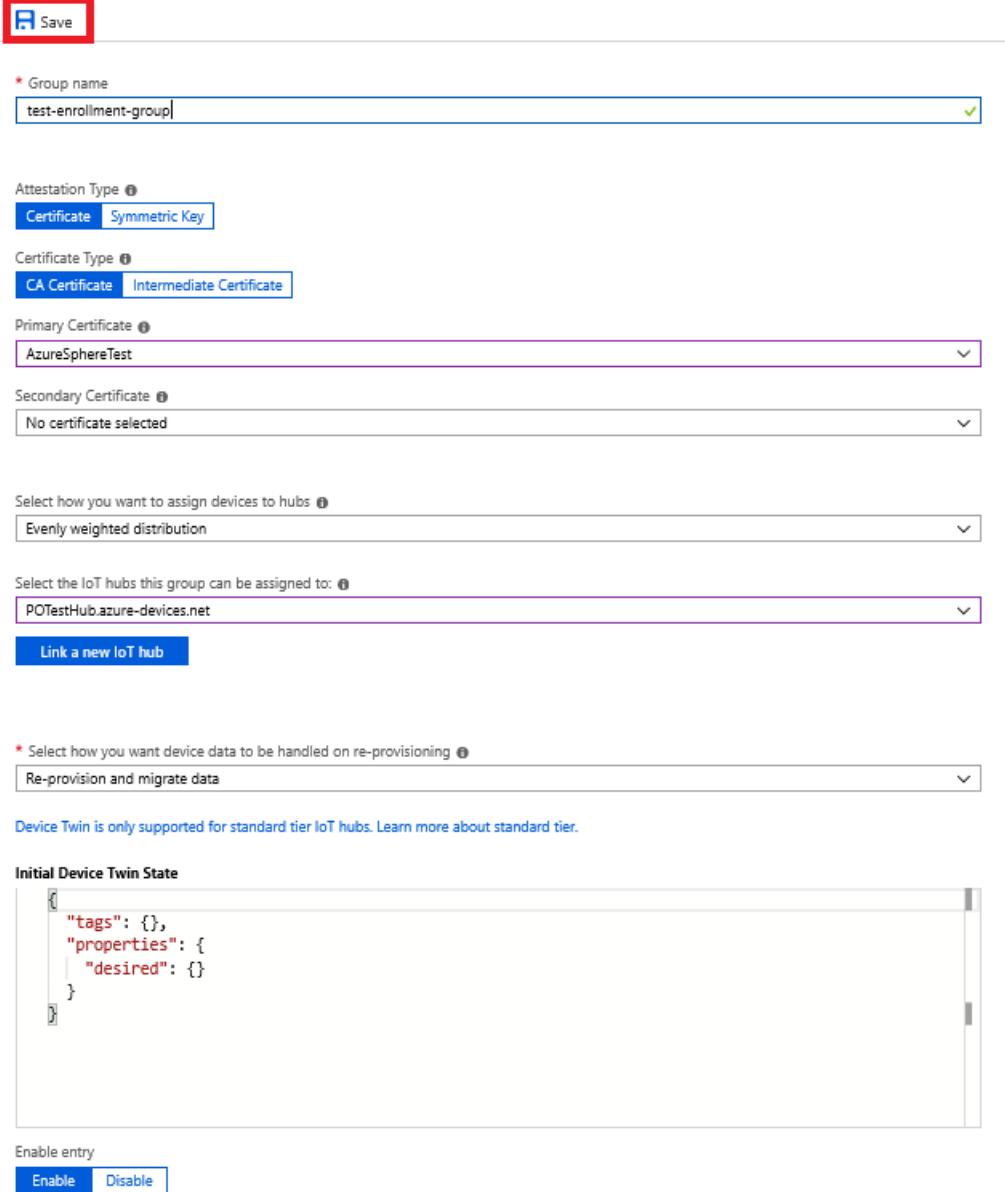
\* Select how you want device data to be handled on re-provisioning ⓘ  
Re-provision and migrate data

Device Twin is only supported for standard tier IoT hubs. Learn more about standard tier.

Initial Device Twin State

```
{  
  "tags": {},  
  "properties": {  
    "desired": {}  
  }  
}
```

Enable entry  
**Enable** Disable



3. Click **Save**. On successful creation of your enrollment group, you should see the group name appear under the **Enrollment Groups** tab.

## Next steps

After you complete these steps, any device that is claimed into your Azure Sphere tenant will be automatically enrolled in your IoT hub when it first connects to your DPS.

You can now run the [Azure IoT sample](#) or build your own applications that use your IoT hub.

# Set up Azure IoT Central to work with Azure Sphere

7/31/2019 • 3 minutes to read

After you have completed the tasks in this section, any device that is claimed by your Azure Sphere tenant will be automatically enrolled when it first connects to your Azure IoT Central application. Therefore, you only need to complete these steps once.

## Prerequisites

The steps in this section assume that:

- Your Azure Sphere device is connected to your PC by USB
- You have an Azure subscription.

## Overview

Setting up Azure IoT Central to work with Azure Sphere devices requires a multi-step process:

1. Create an Azure IoT Central application.
2. Download the authentication CA certificate for your Azure Sphere tenant from the Azure Sphere Security Service.
3. Upload the CA certificate to Azure IoT Central to tell it that you own all devices whose certificates are signed by this CA. In return, Azure IoT Central returns a verification code.
4. Generate and download a validation certificate from the Azure Sphere Security Service, which signs the verification code.
5. Upload the validation certificate to prove to Azure IoT Central that you own the CA.

## Step 1. Create an Azure IoT Central application

1. Sign in to [Azure IoT Central](#) with your Azure credentials.
2. Follow the steps in [create an Azure IoT Central application](#) if you do not already have an application.

### IMPORTANT

Azure IoT Central offers a 7-day free trial application. After 7 days, applications incur charges based on the number of devices and messages. The [Azure IoT Central pricing page](#) provides details.

## Step 2. Download the tenant authentication CA certificate

1. Open an Azure Sphere Developer Command Prompt, which is available in the **Start** menu under **Azure Sphere**.
2. Sign in with the login identity for your Azure Active Directory:

```
azsphere login
```

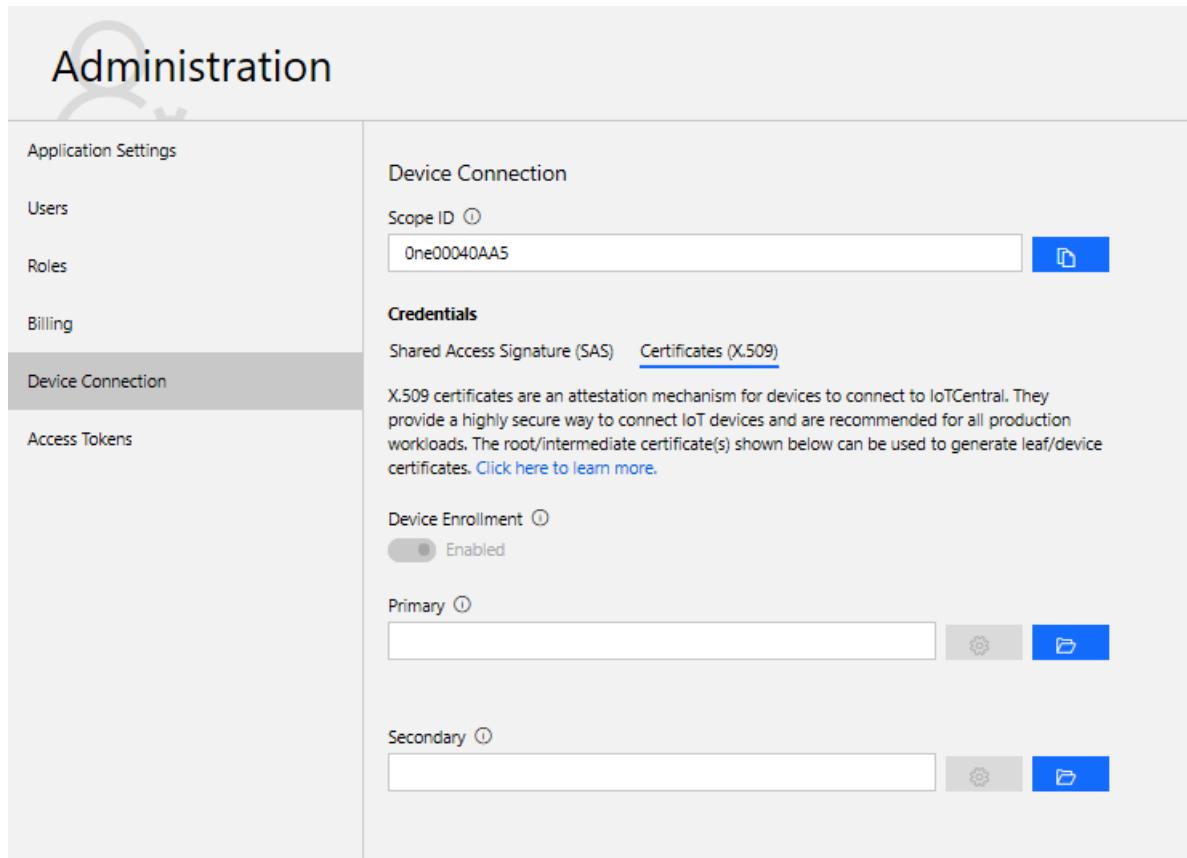
3. Download the Certificate Authority (CA) certificate for your Azure Sphere tenant:

```
azsphere tenant download-CA-certificate --output CAcertificate.cer
```

The output file must have the .cer extension.

## Step 3. Upload the tenant CA certificate to Azure IoT Central and generate a verification code

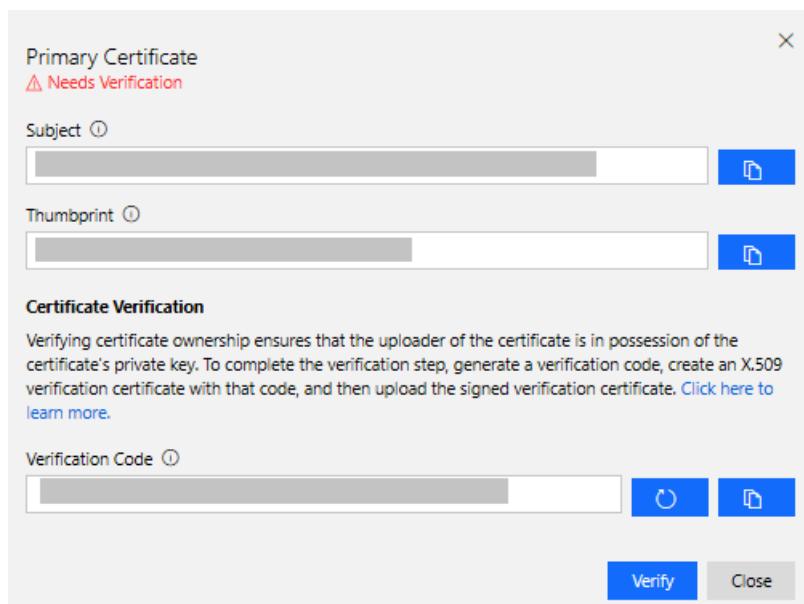
1. In [Azure IoT Central](#), go to **Administration > Device Connection > Certificates (X.509)**:



The screenshot shows the Azure IoT Central Administration interface. On the left, a sidebar lists 'Application Settings', 'Users', 'Roles', 'Billing', 'Device Connection' (which is selected and highlighted in grey), and 'Access Tokens'. The main area is titled 'Device Connection' and contains a 'Scope ID' input field with the value 'One00040AA5' and a 'Certificates (X.509)' link. Below this, a note explains X.509 certificates as an attestation mechanism for devices to connect to IoTCentral. It mentions they provide a highly secure way to connect IoT devices and are recommended for all production workloads. It also notes that root/intermediate certificate(s) shown below can be used to generate leaf/device certificates, with a link to learn more. Under 'Certificates (X.509)', there are two sections: 'Primary' and 'Secondary'. Each section has a text input field for the certificate file, a gear icon for settings, and a folder icon for uploading the file. The 'Primary' section's folder icon is highlighted with a blue border.

2. Click the folder icon next to the Primary box and navigate to the certificate you downloaded in [Step 2](#). If you don't see the .cer file in the list, make sure that the view filter is set to All files (\*). Select the certificate and then click the gear icon next to the Primary box.
3. The Primary Certificate dialog box appears. The Subject and Thumbprint fields contain information about the current Azure Sphere tenant and primary root certificate.

Click the Refresh icon to the right of the Verification Code box to generate a verification code. Copy the verification code to the clipboard.



The screenshot shows the 'Primary Certificate' dialog box. It starts with a note: 'Primary Certificate' and '⚠ Needs Verification'. It has two fields: 'Subject' and 'Thumbprint', each with a placeholder text box and a folder icon. Below these is a 'Certificate Verification' section with the following text: 'Verifying certificate ownership ensures that the uploader of the certificate is in possession of the certificate's private key. To complete the verification step, generate a verification code, create an X.509 verification certificate with that code, and then upload the signed verification certificate.' It includes a link to learn more. At the bottom, there is a 'Verification Code' input field with a placeholder, a refresh icon, and a folder icon. There are also 'Verify' and 'Close' buttons at the bottom right.

## Step 4. Verify the tenant CA certificate

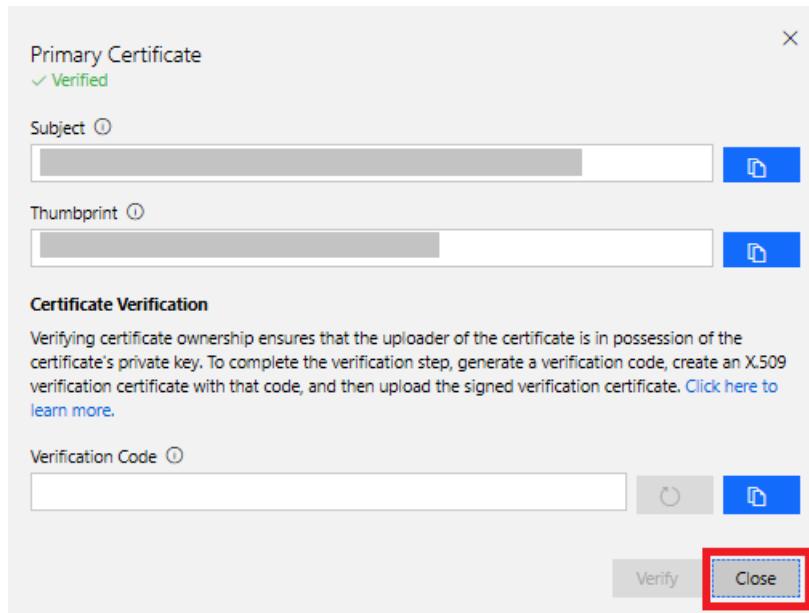
1. Return to the Azure Sphere Developer Command Prompt.
2. Download a validation certificate that proves that you own the tenant CA certificate. Replace *code* in the command with the verification code from the previous step.

```
azsphere tenant download-validation-certificate --output ValidationCertification.cer --verificationcode  
<code>
```

The Azure Sphere Security Service signs the validation certificate with the verification code to prove that you own the CA.

## Step 5. Use the validation certificate to verify the tenant identity

1. Return to Azure IoT Central and click **Verify**.
2. When prompted, navigate to the validation certificate that you downloaded in the previous step and select it. When the verification process is complete, the Primary Certificate dialog box displays the Verified message. Click **Close** to dismiss the box.



## Next steps

After you complete these steps, any device that is claimed into your Azure Sphere tenant will automatically be accessible to your Azure IoT Central application.

You can now run the [Azure IoT sample](#) or use Azure IoT Central to monitor and control any of your Azure Sphere devices.

# Use Microsoft Azure IoT Hub

6/20/2019 • 2 minutes to read

An Azure Sphere application can interact with an IoT Hub to send and receive messages, manage a device twin, and receive direct method calls from an Azure IoT service application. To use these features, you need a Microsoft Azure subscription and an IoT Hub.

## Azure IoT hub setup

Use of an Azure IoT hub with an Azure Sphere device involves a multi-step authentication process:

- Download an authentication CA certificate from the Azure Sphere Security Service, which validates your Azure Sphere tenant's certificate authority.
- Upload the CA certificate to the Azure IoT Hub Device Provisioning Service (DPS) to register the device in your Azure IoT hub.
- Validate the CA certificate to prove ownership of the Azure Sphere tenant. In return, receive a second certificate—the validation certificate—with which you can register your device in the IoT hub.

This process helps to:

- Safeguard against spoofing the device identity, so that an untrusted device cannot be used
- Prevent the use of compromised or untrusted Azure Sphere OS
- Ensure that only an authorized entity can register the device in an IoT hub

Follow the steps in [Set up IoT Hub for Azure Sphere](#) to complete the authentication process.

### IMPORTANT

Although it is possible to use a connection string with a device-specific shared access key to authenticate an application to the IoT Hub, such a solution is less secure than using certificate-based authentication. Anyone who has access to the shared access key can send and receive messages on behalf of that device. To ensure the security of your devices and applications, always use the certificate-based authentication procedure that is described in this topic.

## Using IoT Hub

The Azure Sphere SDK Preview for Visual Studio includes a Connected Service extension that facilitates the setup and use of the Azure IoT Hub SDK with an Azure Sphere application.

When you add the extension to your project, you identify the IoT Hub that you plan to use. See [Azure IoT sample application](#) on GitHub for a walkthrough of Azure IoT Hub setup.

## IoT Hub SDK

The [Azure IoT Device SDK for C](#) includes an IoT Hub client library that you can use in Azure Sphere applications.

## To learn more about IoT Hub

These tools can help you manage devices in IoT Hub:

- [Microsoft Azure Portal](#)

- [Visual Studio Cloud Explorer](#)
- [Device Explorer](#) lets you manage devices, send cloud-to-device messages, and monitor device-to-cloud messages.
- [Iohub-explorer](#) is a command-line tool that does the same tasks as Device Explorer but also lets you query and set information in the device twin.
- [Azure IoT Toolkit](#) is a cross-platform, open-source Visual Studio Code extension that helps you manage Azure IoT Hub and devices in VS Code.

For extended IoT scenarios using other Azure services and tools, check out these tutorials:

- [Manage devices with Visual Studio Cloud Explorer](#)
- [Manage IoT Hub messages](#)
- [Manage your IoT device](#)
- [Save IoT Hub messages to Azure storage](#)
- [Visualize sensor data](#)

# Over-the-air application deployment

9/28/2018 • 2 minutes to read

When you are ready to distribute an application over the air to a group of devices, you create a deployment for the application. A *deployment* delivers software over the air to the Azure Sphere devices on which it runs. Only one deployment is active for a particular device at any given time.

The Azure Sphere deployment model provides the ability to specify which software package should be delivered to any individual connected device. Every connected device is uniquely identified by the device ID of its integrated Azure Sphere MCU. [Deployment basics](#) describes the deployment model.

Deploying an application involves uploading it to the Azure Sphere Security Service, which then feeds it to the devices to which it is targeted. Before your application can access the Azure Sphere Security Service, you must have a [work or school account](#) and an [Azure Sphere tenant](#).

To deploy an application:

- [Prepare the device for OTA updates](#)
- [Link the device to a feed](#)

## TIP

If you have never deployed an application, we recommend that you build the Blink application and deploy it as described in the deployment [Quickstart](#). It introduces the deployment commands and guides you through the basic tasks involved in deploying an application to a single device.

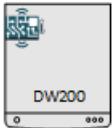
# Deployment basics

5/30/2019 • 9 minutes to read

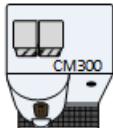
We will use three appliance models from the fictitious Contoso Corporation as an example throughout this discussion.



Dishwasher DW100



Dishwasher DW200



Coffee Maker CM300

## Device IDs

Every Azure Sphere chip has a unique and immutable Azure Sphere device ID. The Azure Sphere device ID uniquely identifies the individual chip. The device ID is stored on the device itself. All the other elements of a deployment are stored with the Azure Sphere Security Service.

## SKUs and SKU sets

A stock keeping unit (SKU) is a GUID that identifies a model of physical device. A *product SKU* identifies a connected device product that incorporates an Azure Sphere MCU. As the manufacturer, you create a product SKU for each model of connected device, such as a dishwasher or coffeemaker. For example, Contoso creates a product SKU for its DW100 dishwashers and assigns this product SKU to each DW100 dishwasher during manufacturing.

Every connected device has a single product SKU, but a single product SKU can be associated with many devices. Each product SKU has a name and a description, which provide a human-readable way to distinguish one product SKU from another.

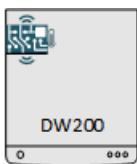
In addition, every Azure Sphere chip has a *chip SKU* that identifies a particular type of Azure Sphere-compatible MCU. The chip SKU is assigned by Microsoft and cannot be changed. Microsoft uses this SKU to deliver the correct system software updates to each Azure Sphere device.

A *SKU set* identifies all the hardware that is incorporated into a connected device, and thus identifies all the software that is compatible with the device. The SKU set for a connected device includes both its manufacturer-assigned product SKU and the Microsoft-assigned chip SKU.

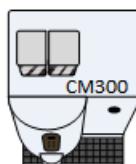
For example:



DW100 Product SKU  
MT3620 Chip SKU



DW200 Product SKU  
MT3620 Chip SKU



CM300 Product SKU  
MT3620 Chip SKU

## SKU sets for Contoso appliances

The figure shows three Contoso appliances, all of which include an Azure Sphere MT3620 MCU:

- The DW100 dishwasher has the DW100 Product SKU. Its SKU set indicates that the device is compatible

with Contoso application software that targets the DW100 dishwasher.

- The DW200 dishwasher has additional features that are not available in the DW100 and thus has a different product SKU: the DW200 product SKU. Its SKU set indicates that the device is compatible with Contoso application software that targets the DW200 dishwasher. Currently, the DW200 runs the same application software as the DW100, but Contoso intends to release DW200-specific software soon.
- The CM300 coffeemaker has the CM300 product SKU.

## Components and applications

A *component* represents a software package that can be updated. The preceding Contoso example involves two components:

- DW100SW dishwasher application software, which runs on the DW100 and DW200 models
- CM300SW coffeemaker application software

An *application* is a component that performs tasks specific to certain connected devices. Each application targets one or more product SKUs that are associated with those connected devices. As a product manufacturer, you develop and manage applications, whereas Microsoft develops and manages system software components. System software components target chip SKUs.

## Images and image sets

An *image* represents a single version of a component. Images are immutable: you cannot modify an image after it has been uploaded. For an application, the image includes the binaries for the application along with its image metadata. Each image has a unique image ID, which is part of the image metadata. Every time the SDK builds or rebuilds an Azure Sphere image package, it uses a new unique image ID.

An *image set* is a group of images that represent interdependent components and therefore must be deployed and updated as a unit. Like a single image, an image set is immutable.

## Feeds

A *feed* delivers software to one or more devices. Microsoft creates feeds to deliver the Azure Sphere OS, and you create feeds to deliver your application software. Feeds are hierarchical: every application feed depends on an Azure Sphere OS feed.

When you create an application feed, you specify the system software feed that it depends on, the product and chip SKUs that it targets, and the component that it delivers. Currently, each application feed is associated with a single component.

When you are ready to deploy an application, you create an image that represents the version of the application you want to deploy. Then you add the image to an image set, and add the image set to a feed for the associated component. The most recently added image set becomes the current image set. The Azure Sphere Security Service delivers the current image set to the targeted devices. Although a feed has only one current image set, each feed maintains an audit list of all image sets that were ever added to it.

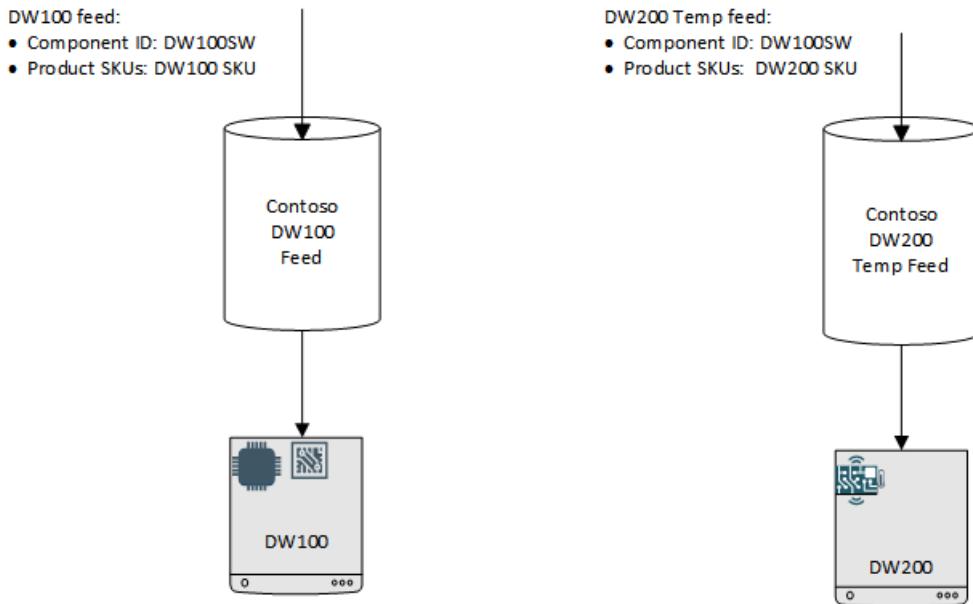
An application feed is analogous to a pipe that carries only a certain kind of material; that is, an application feed can only deliver image sets that represent a particular component. Therefore, Contoso would define one feed for the DW100SW application software and another for the CM300SW application software because the two devices use different applications.

A single feed can supply its software to one or more product SKUs. After you define a feed, you can add an image set, but you cannot add or remove a component, a chip SKU, or a product SKU from the feed. That is, the definition of the feed—the components it delivers and the SKUs it targets—is immutable.

In the Contoso example, the DW100 and DW200 dishwashers have different product SKUs but both currently use the DW100SW application software. Contoso has two options:

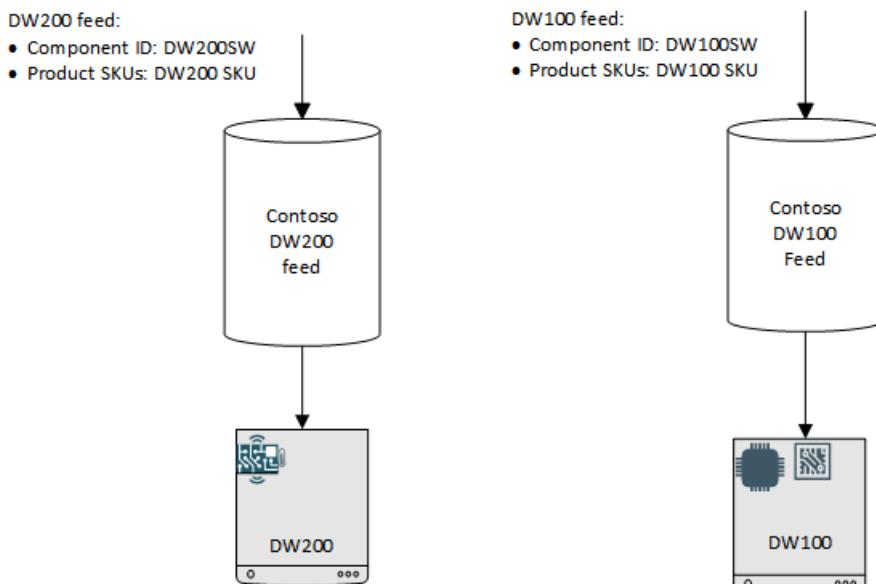
- Define a single feed that delivers the DW100SW application software to both the DW100 and DW200 product SKUs
- Define one feed that delivers the DW100SW application software to the DW100 product SKU and another feed that delivers the DW100SW application software to the DW200 product SKU

Contoso plans to release a separate application for the DW200 soon. If it defines a single feed that delivers the current application to both product SKUs, it must create two new feeds when the new software is ready: one for each SKU. Instead, it creates two feeds now. The DW100 feed delivers the DW100SW application to DW100 dishwashers, and a temporary DW200 feed delivers the DW100SW application to the DW200 dishwashers. The following figure shows these sample feeds:



### Contoso feeds for DW100 application

When Contoso is ready to test its new DW200-specific application, it creates a new component and image set to represent the test version of the DW200SW application. Contoso then sets up a new feed that delivers the DW200SW component to devices that have the DW200 product SKU, and it assigns the DW200 test application image set to this new feed. The existing DW100 feed continues to deliver the DW100SW application to DW100 dishwashers.



## Contoso feed for new DW200 application

As this example implies, a product manufacturer would typically define multiple feeds. For example, Contoso might define two feeds for each of its applications: a Test feed and a Retail feed. The Test feed delivers application software that is still in development, and the Retail feed delivers application software that is ready for field deployment. Both feeds target the same SKU set, but deliver different versions of the application. That is, they are associated with different image sets.

### DW100 Production feed:

- Component ID: DW100SW
- ImageSet: DW100SW v1.0
- Product SKUs: DW100 SKU

### DW100 Test feed:

- Component ID: DW100SW
- ImageSet: DW100SW v1.5
- Product SKUs: DW100 SKU



## Contoso retail and test feeds

The Test and Retail feeds in the figure are identical except for their image sets. The Retail feed is associated with the DW100SW v1.0 image set, and the Test feed is associated with the DW100SW v1.5 image set. To test a new version of its DW100SW application, Contoso simply creates a new image set that contains the updated software, using the component ID for its test software, and assigns that image set to the Test feed. Similarly, if Contoso discovers errors during testing, it can roll back the deployment to an earlier version by assigning an earlier image set to the Test feed.

Feeds and image sets determine which components—and which versions of those components—are deployed over the air to connected devices. Feeds link components and image sets with product SKUs, and device groups link feeds with individual connected devices.

## Device groups

Device groups provide a way to scale application deployment to many devices. A *device group* is a named collection of devices that have something in common, together with a list of feeds that deliver software to those devices. Each device belongs to exactly one device group. For example, Contoso might create one device group for the devices in its test lab and another for devices in the retail channel. Devices in the test lab group receive the test application feed, and the devices in the retail group receive the production application feed. Alternatively, Contoso might group devices by warranty status or geography. The grouping criteria are left completely to the discretion of the manufacturer.

A device group can contain products with different SKUs. For example, a single device group could contain several DW100 dishwashers, DW200 dishwashers, and CM300 coffeemakers. The only restriction is that every device in the group must belong to the same Azure Sphere tenant.

Each device group is associated with a list of the feeds for its member devices. Thus, a device group for the Contoso Test Lab might contain the following information:

**Devices:**

DW100 device 123  
DW200 device 456  
CM300 device 789  
CM300 device 0ab

**Feeds:**

DW100 Test feed, which targets the DW100 product SKU  
DW200 Temp feed, which targets the DW200 product SKU with DW100SW software  
CM300 Test feed, which targets the CM300 product SKU

This device group provides the DW100SW Test software to the DW100 dishwasher, DW100SW Test software to the DW200 dishwasher through the DW200 Temp feed previously described, and the CM300SW Test software to both the CM300 coffeemakers.

Each feed assigned to a device group must supply a unique SKU set. In this example, both CM300 coffeemakers have the CM300 product SKU, so both devices must use the version of the CM300SW that the CM300 Test feed supplies.

To deploy updated software to the devices in the test lab, Contoso creates a new image set and adds it to the appropriate feed. (Remember, image sets are immutable.) For example, when Contoso is ready to test a new version of the DW100SW application, it creates an image set that represents the new version and assigns it to the DW100 Test feed. To test its new DW200SW application, it creates a DW200 Test feed and links it to the Contoso Test Lab device group. The device group then contains the following information:

**Devices:**

DW100 device 123  
DW200 device 456  
CM300 device 789  
CM300 device 0ab

**Feeds:**

DW100 Test feed, which targets DW100 product SKU  
DW200 Test feed, which targets DW200 product SKU  
CM300 Test feed, which targets CM300 product SKU

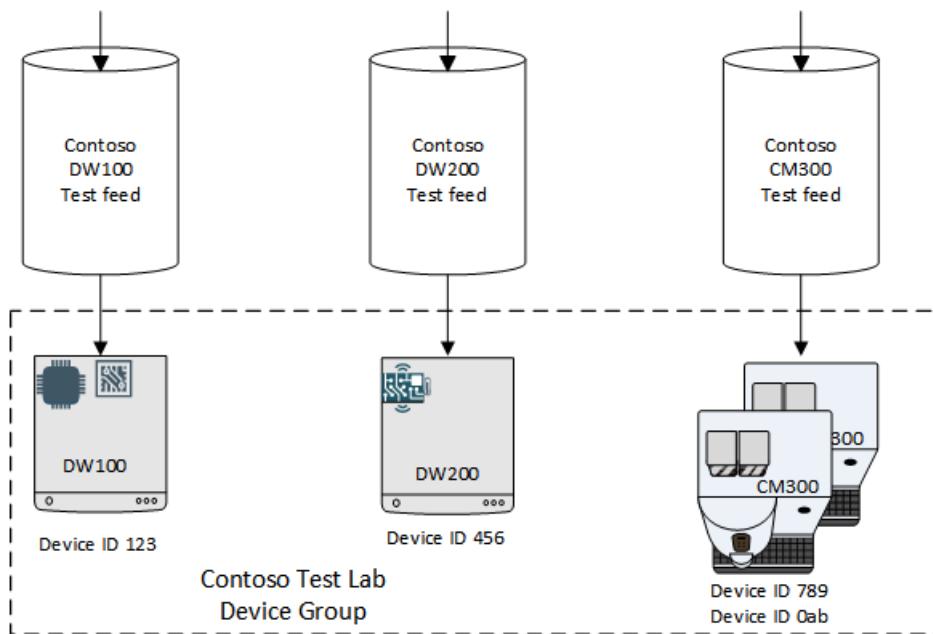
The next time device 456 in the Contoso Test Lab requests an update, it receives the DW200SW test software via the DW200 Test feed.

The following diagram shows how feeds, image sets, and device groups together determine the software that is deployed to each device:

- DW100 Test feed:
- Component ID: DW100SW
  - ImageSet: DW100SW v1.5
  - Product SKUs: DW100 SKU

- DW200 Test feed:
- Component ID: DW200SW
  - ImageSet: DW200SW v0.5
  - Product SKUs: DW200 SKU

- CM300 Test feed:
- Component ID: CM300SW
  - ImageSet: CM300SW v2.5
  - Product SKUs: CM300 SKU



### Contoso Test Lab device group and feeds

Keep in mind that the device group does not establish which components are deployed. The SKU set establishes which components can run on the device, the feed determines the current image set, and the image set represents the current version of the component. These elements taken together determine the deployment.

# Which software targets a device?

5/30/2019 • 2 minutes to read

To determine which version of any software runs on any particular connected device:

- Which device group contains the connected device? A connected device belongs to exactly one device group.
- Within that device group, which feed specifies the SKU set for the connected device? A device group can be associated with multiple feeds, but only one that matches the SKU set for any individual connected device.
- What is the current image set for that feed? The current image set provides the version of the application that the Azure Sphere Security Service will deploy to and run on the connected device.

# Deployment history and security

8/13/2018 • 2 minutes to read

The ability to track and reverse deployments is an essential part of the security built into Azure Sphere. The definition of SKU sets, feeds, image sets, and device groups makes it possible for the Azure Sphere Security Service to maintain a history of the image sets that have been added to each feed. The history is available through Azure Sphere utilities, so by determining the device group to which a particular device belongs, and the current feeds and image sets targeted at the device's SKU set, you can determine exactly which set of software should be running on the given device. The [azsphere device image list-targeted](#) and [azsphere feed image-set list](#) commands provide detailed information.

Immutable image sets enable predictable, repeatable, and reversible software deployment. If problems occur, support engineers can roll back a deployment by simply assigning a different image set to the relevant feed. All the images in a single image set are updated atomically.

# When do updates occur?

11/15/2018 • 2 minutes to read

Azure Sphere devices check for updates when they first connect to the internet after powering on or after a user presses the Reset button on an MT3620 development board. Thereafter, checks occur at regular intervals (currently 24 hours).

The Azure Sphere Security Service downloads an OTA application deployment if both the following are true:

- An existing OTA deployment targets the device's device group and SKU set
- The device group to which the device belongs allows OTA application updates

If a device is not up to date when it connects to the internet either for the first time or after an extended off-line period, you may see unexpected behavior as the device receives an OTA update that causes it to restart.

# Azure Sphere OS feeds

3/20/2019 • 2 minutes to read

Microsoft deploys OTA updates for the Azure Sphere OS through system software feeds. The **Retail Azure Sphere OS** feed provides our highest quality software that is ready for production use. The **Retail Evaluation Azure Sphere OS** feed provides OS software 14 days before its release to the Retail Azure Sphere OS feed, so that you can verify compatibility before broad deployment. Each time we release software to the Retail Evaluation feed or the Retail feed, we will [notify you](#) through the usual channels.

When you configure an OTA application deployment, you must specify the OS feed on which it depends. OTA application deployments to connected devices at end-user sites should always use the Retail feed.

Applications that are built with production APIs are compatible with updated OS releases. [Beta features](#), however, may change from one release to the next.

To verify your applications with the Retail Evaluation OS, we recommend that you [set up a separate retail Evaluation device group](#) that depends on the Retail Evaluation feed (feed ID 82bacf85-990d-4023-91c5-c6694a9fa5b4).

## IMPORTANT

Contact your Microsoft Technical Account Manager (TAM) immediately if you encounter any compatibility issues so that we can assess and address the issue before we release the OS software to the Retail feed.

# Deploy an application over the air

3/14/2019 • 2 minutes to read

Although the **azsphere** command-line utility can perform each individual deployment task, you can deploy an application to a locally attached Azure Sphere device in two steps by using **azsphere** commands that combine several other commands:

- [Prepare the device for OTA updates](#)
- [Link the device to a feed](#)

If you want to perform each over-the-air (OTA) application deployment task separately for learning or troubleshooting purposes, you can use **azsphere** to do the following series of tasks:

- Removing any current sideloaded application from the device
- Removing any **appdevelopment** capability, so that only production-signed applications can be loaded on the device
- Assigning a [product SKU](#) to the device
- Assigning the device to a [device group](#) that provides OTA application updates
- Adding the application image to a [component](#)
- Creating a [feed](#) to deliver the component
- Adding the feed to the device group to which the device belongs
- Creating an [image set](#) that represents the application
- Adding the image set to the feed

After your initial deployment, you can [update a deployment](#) by assigning a different image set to the existing feed.

# Prepare a device for OTA updates

11/15/2018 • 3 minutes to read

When you are ready to deploy a production application, prepare your device by:

- Removing the current sideloaded applications from the device, including the customer application and the debugging server
- Removing the appDevelopment capability, so that only production-signed applications can be loaded
- Assigning a [product SKU](#) to the device
- Assigning the device to a [device group](#) that enables OTA application updates

The **azsphere device prep-field** command performs these tasks in a single step.

## Create a new product SKU and device group

If you have not already created a product SKU or a device group for the device, use an **azsphere device prep-field** command like the following to create a new product SKU and device group:

```
azsphere device prep-field --newdevicegroupname <UniqueGroupNameOTA> --newskuname <UniqueSKUName> --skudescription <FriendlyDescription>
```

The --newdevicegroupname parameter specifies a name for the new device group that the command creates. All device groups created by this command support automatic OTA application updates. The device group name must be unique in your Azure Sphere tenant.

The --newskuname parameter specifies a name for the new product SKU that the command creates. The SKU name must be unique in your Azure Sphere tenant.

The --skudescription parameter provides an optional friendly description of the product SKU. Enclose the string in quotation marks if it includes spaces.

For example:

```
azsphere device prep-field --newdevicegroupname POTestGroup --newskuvalue POTestSKU --skudescription "Test MT3620 docs"

Removing applications from device.
Component 'ae4714aa-03aa-492b-9663-962f966a9cc3' deleted or was not present beforehand.
Removing debugging server from device.
Component '8548b129-b16f-4f84-8dbe-d2c847862e78' deleted or was not present beforehand.
Successfully removed applications from device.
Locking device.
Downloading device capability configuration for device ID
'ABCDEF82513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A5588B42085
1EE4F3F1A7DC51399ED'.
Successfully downloaded device capability configuration.
Applying device capability configuration to device.
Successfully applied device capability configuration to device.
The device is rebooting.
Successfully locked device.
Creating a new device group with name 'POTestGroup'.
Setting device group ID 'f90189d0-f7ec-4e76-8d4d-b826fde85cf' for device with ID
'ABCDEF82513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A5588B42085
1EE4F3F1A7DC51399ED'.
Creating a new SKU with name 'POTestSKU'.
Setting product SKU to '0efa3fdd-cba9-4eae-a75d-05b6f043de6b' for device with ID
'ABCDEF82513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A5588B42085
1EE4F3F1A7DC51399ED'.
Successfully set up device
'ABCDEF82513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A5588B42085
1EE4F3F1A7DC51399ED' for OTA loading.
Command completed successfully in 00:00:14.5977871.
```

## Use an existing device group

If you have already deployed OTA application updates for a similar device, you have most likely already created the device group and product SKU. If so, include them on the command line:

```
azsphere device prep-field --devicegroupid <GUID> --skuid <GUID>
```

Replace in the example with the IDs for the device group and product SKU for the device, respectively. To list the device groups in your tenant, use the **azsphere device-group list** command. Use **azsphere sku list** to list the SKUs.

The device group must support all updates—that is, Azure Sphere OS and application updates. All device groups support Azure Sphere OS updates. To find out whether a particular device group supports application updates, use **azsphere device-group show**. The Update Policy line indicates whether the device group supports all updates or only Azure Sphere OS updates:

```
azsphere device-group show --devicegroupid 1c5d6515-8a77-4a5e-81d9-b7dc1fbfaef
Getting device group with ID '1c5d6515-8a77-4a5e-81d9-b7dc1fbfaef'.
Successfully retrieved the device group:
--> ID:          '1c5d6515-8a77-4a5e-81d9-b7dc1fbfaef'
--> Name:        'Popcorn Makers - North America'
--> Update Policy: Accept all updates from the Azure Sphere Security Service.
Command completed successfully in 00:00:06.0729682.
```

The following example assigns an existing device group and SKU to the attached device:

```
azsphere device prep-field --devicegroupid f90189d0-f7ec-4e76-8d4d-b826fde85cf --skuid 0efa3fdd-cba9-4eae-a75d-05b6f043de6b
Getting the details of device group with ID 'f90189d0-f7ec-4e76-8d4d-b826fde85cf'.
Removing applications from device.
Removing debugging server from device.
Component '8548b129-b16f-4f84-8dbe-d2c847862e78' deleted.
Successfully removed applications from device.
Locking device.
Downloading device capability configuration for device ID
'ABCDEF82513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A5588B42085
1EE4F3F1A7DC51399ED'.
Successfully downloaded device capability configuration.
Applying device capability configuration to device.
Successfully applied device capability configuration to device.
The device is rebooting.
Successfully locked device.
Setting device group ID 'f90189d0-f7ec-4e76-8d4d-b826fde85cf' for device with ID
'ABCDEF82513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A5588B42085
1EE4F3F1A7DC51399ED'.
Setting product SKU to '0efa3fdd-cba9-4eae-a75d-05b6f043de6b' for device with ID
'ABCDEF82513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A5588B42085
1EE4F3F1A7DC51399ED'.
Successfully set up device
'ABCDEF82513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A5588B42085
1EE4F3F1A7DC51399ED' for OTA loading.
Command completed successfully in 00:00:18.0674049.
```

# Link the device to a feed

2/14/2019 • 2 minutes to read

After you [prepare a device for OTA updates](#), you link it to a feed that delivers your application. This operation involves:

- Adding the application image to a [component](#)
- Creating or specifying a [feed](#)
- Adding the feed to the device group you specified in the previous step
- Creating an [image set](#) that represents the application
- Adding the image set to the feed

The **azsphere device link-feed** operation is the easiest way to do this. You can create a new feed or link to an existing feed.

## Create and link to a new feed

If you have not previously deployed this component, create a new feed. Use a command in the following form:

```
azsphere device link-feed --dependentfeedid 3369f0e1-dedf-49ec-a602-2aa98669fd61 --imagepath <path-to-image> --newfeedname <unique-feed-name>
```

By default, the command applies to the Azure Sphere device that is attached to the PC. To set up a different device, use the `--deviceid` parameter instead and specify the ID of the device to set up. The device ID is used to infer the product SKU and chip SKU that the new feed should target. The new feed is associated with the device group that contains the device.

The `--dependentfeedid` parameter supplies the ID of the Azure Sphere OS feed on which the application depends. To create a new feed, you must supply the ID of the dependent feed. All application feeds depend on an Azure Sphere OS feed. Currently, the Azure Sphere OS feed is named Retail Azure Sphere and the feed ID is 3369f0e1-dedf-49ec-a602-2aa98669fd61.

The `--imagepath` parameter provides the path to the image package file for the application that the newly created feed will distribute. As a result, the image package file is uploaded to the Azure Sphere Security Service and is added to the new image set that the command creates. If the component ID of the image package does not match the component ID that the feed delivers, the command returns an error. By default, the command creates a unique name for the image set, based on the name of the image package. To override the default, use the `--newimagesetname` parameter and specify a unique name.

The `--newfeedname` parameter provides a name for the new feed that the command creates. The feed name must be unique within your Azure Sphere tenant. The feed distributes the component whose component ID is specified in the image package file.

**azsphere** displays progress information about each step, as the following example shows:

```
azsphere device link-feed --dependentfeedid 3369f0e1-dedf-49ec-a602-2aa98669fd61 --imagepath  
"C:\Users\Test\Documents\Visual Studio  
2017\Projects\Mt3620BlinkTestImage\Mt3620BlinkTestImage\bin\ARM\Debug\Mt3620BlinkTestImage.imagepackage" --  
newfeedname P0TestFeed  
  
Getting the details for device with ID 'device-id'.  
  
Uploading image from file 'C:\Users\Test\Documents\Visual Studio  
2017\Projects\Mt3620BlinkTestImage\Mt3620BlinkTestImage\bin\ARM\Debug\Mt3620BlinkTestImage.imagepackage':  
--> Image ID: deee01f7-e636-45a3-8faa-a6c75694b241  
--> Component ID: d89a72b5-8481-45f9-99b6-58cf3cc93a5b  
--> Component name: 'Mt3620BlinkTestImage'  
Removing temporary state for uploaded image.  
Create a new feed with name 'P0TestFeed'.  
Adding feed with ID 'f1cd9099-1bd2-46ca-b523-774298823d86' to device group with ID 'b2f63faf-5bb3-4f75-b4f1-  
a4f0ba698fc2'.  
Creating new image set with name 'ImageSet-Mt3620BlinkTestImage-2018.07.22-18.29.35+01:00' for image with ID  
'deee01f7-e636-45a3-8faa-a6c75694b241'.  
Adding image set with ID 'c0c2c974-f84d-44fa-a74f-b31227db5c43' to feed with ID 'f1cd9099-1bd2-46ca-b523-  
774298823d86'.  
Successfully linked device 'device-id' to feed with ID 'f1cd9099-1bd2-46ca-b523-774298823d86'.  
Command completed successfully in 00:00:34.1718633.
```

## Link to an existing feed

If you already have a feed for this component, use a command like the following:

```
azsphere device link-feed --imagepath <path-to-image> --feedid <GUID>
```

This command links the feed identified by the GUID to the device group that contains the attached device and uploads the specified image package.

The next time the device [checks for updates](#), the Azure Sphere Security Service downloads and starts the application.

# Set up a device group for OS evaluation

3/14/2019 • 2 minutes to read

Microsoft deploys the updated Azure Sphere OS on the Retail Evaluation Azure Sphere OS feed before releasing it on the Retail Azure Sphere OS feed, as described in [Azure Sphere OS feeds](#). By using the Retail Evaluation feed, you can verify production-signed applications against the new release before we deploy it broadly on the Retail feed.

The instructions in this topic assume:

- The device that is attached to your PC will be used for evaluation.
- Your device already has an OTA application deployment. If not, follow the steps in [Deploy an application over the air](#) to set one up.

To verify your applications against the Retail Evaluation Azure Sphere OS, you need to create both a new device group and a new feed, as follows:

1. Connect your device to your PC.
2. Create a new device group that enables application updates:

```
azsphere device-group create --name "Retail Evaluation Group with OTA app"
```

You will use the device group ID in a later step.

3. Get information about the current OTA configuration for the attached device, so that you can set up a new feed with the same configuration:

```
azsphere device image list-targeted
```

You can find the image set ID and the component IDs in the output. You will use these in later steps.

```
Successfully retrieved the current image set for device with ID '<device-ID>' from your Azure Sphere tenant:  
--> Image set ID: 188a93b8-d304-4230-a6af-b58ac33af819  
--> Image set name: ImageSet-lan-enc28j60-isu0-int5_4add93-PrivateEthernet-2018.11.27-09.59.42+00:00  
Images to be installed:  
--> lan-enc28j60-isu0-int5_4add93  
--> Image type: Board configuration  
--> Component ID: 4add937b-5f32-4b9d-b5fa-568007b26e6a  
--> Image ID: 015ecc67-7667-4fe0-8ed6-c7587c9554b4  
--> PrivateEthernet  
--> Image type: Application  
--> Component ID: 283b16c8-d6a2-4d56-8abb-f390b0d3dda9<  
--> Image ID: f0b357e0-8a9f-4037-8116-4b4037ff7562  
Command completed successfully in 00:00:03.2752069.
```

4. Add the attached device to the new device group:

```
azsphere device update-device-group --devicegroupid <device-group-ID>
```

To add more devices, use the same command but add the `--deviceid <deviceID>` option.

5. Create a Retail Evaluation application feed that depends on the Retail Evaluation Azure Sphere OS feed and delivers the same components as your current OTA configuration. The following command creates and names the feed and assigns it to the device group that contains the attached device.

```
azsphere device link-feed --dependentfeedid 82bacf85-990d-4023-91c5-c6694a9fa5b4 --newfeedname "Eval Feed for OTA App" --componentid <component-ID>, <component-ID>, ...
```

After you link this feed, all devices in the same device group as the attached device will get the new feed.

You will use the new feed ID in the next step.

6. Add the existing image set to the new feed:

```
azsphere feed image-set add --feedid <new-feedID> --imagesetid <image-set-ID>
```

7. Verify that the attached device is in the intended device group:

```
azsphere device show-ota-config
```

8. Verify that the device will continue to receive the images you expect:

```
azsphere device image list-targeted
```

9. Within 24 hours, your device should receive the Retail Evaluation OS OTA. To trigger OTA update in the next 10 minutes, press the Reset button on the device. To verify that the device has received the Retail Evaluation OS:

```
azsphere device show-ota-status
```

#### IMPORTANT

Contact your Microsoft Technical Account Manager (TAM) immediately if you encounter any compatibility issues so that we can assess and address the issue before we release the OS software to the Retail feed.

# Update a deployment

11/15/2018 • 2 minutes to read

After your initial deployment, you can update a deployment by assigning a different image set to the existing feed.

## Deploy an updated image set

To update a feed with a new version of your application, use the **azsphere component publish** command. This command uploads a new image package, creates a new image set, and adds the new image set to an existing feed.

The following example uploads a new version of the MyIoTHubApp image package, creates a new image set that contains this image package, and adds it to the existing feed identified by the --feedid parameter. The new image set then becomes the current image set for the feed.

```
azsphere component publish --feedid a48bb8cf-bee5-439c-a3fa-889c5f1c9807 --imagepath  
"C:\Users\User\Documents\Visual Studio  
2017\Projects\Mt3620Uart1\Mt3620Uart1\bin\ARM\Debug\Mt3620Uart1.imagepackage"
```

Although **azsphere component publish** supports the --newimagesetname parameter, which supplies a name for the new image set, this example does not use it. If the parameter is not present, the command generates a unique name for the image set, as the output shows:

```
Publishing images to feed with ID 'a48bb8cf-bee5-439c-a3fa-889c5f1c9807'.  
Getting details for feed with ID 'a48bb8cf-bee5-439c-a3fa-889c5f1c9807'.  
Uploading image from file 'C:\Users\User\Documents\Visual Studio  
2017\projects\Mt3620Uart1\Mt3620Uart1\bin\ARM\Debug\Mt3620Uart1.imagepackage':  
--> Image ID: f177ead4-1bbb-4f2a-9364-8613b06fa764  
--> Component ID: 07c1c908-df57-44cc-8315-6edebac203e1  
--> Component name: 'Mt3620Uart1'  
Removing temporary state for uploaded image.  
Creating new image set with name 'ImageSet-Mt3620Uart1-2018.11.13-16.22.28-08:00' for images with IDs  
'f177ead4-1bbb-4f2a-9364-8613b06fa764'.  
Adding image set with ID '0fe78fad-b611-4d9d-9ed8-c3308eba613e' to feed with ID 'a48bb8cf-bee5-439c-a3fa-  
889c5f1c9807'.  
Successfully published the following images to feed with ID 'a48bb8cf-bee5-439c-a3fa-889c5f1c9807':  
-> 'C:\Users\User\Documents\Visual Studio  
2017\projects\Mt3620Uart1\Mt3620Uart1\bin\ARM\Debug\Mt3620Uart1.imagepackage'  
Command completed successfully in 00:00:17.8908297.
```

## Redeploy an image set

The Azure Sphere Security Service maintains information about all the image sets that are associated with a feed, along with the image sets themselves. Consequently, you can easily redeploy an earlier image set by reassigning its image set ID to the feed, thus making it the current image set for the feed.

Use **azsphere feed image-set list** to get a list of the previously assigned image sets. For example:

```
Listing all image sets in feed 'a48bb8cf-bee5-439c-a3fa-889c5f1c9807'.
Retrieved 2 image sets for feed 'a48bb8cf-bee5-439c-a3fa-889c5f1c9807':
--> {
    "Id": "0fe78fad-b611-4d9d-9ed8-c3308eba613e",
    "FriendlyName": "ImageSet-Mt3620Uart1-2018.11.13-16.22.28-08:00"
}
--> {
    "Id": "d994d535-7ea0-45b1-a790-8e6c8e37b15b",
    "FriendlyName": "ImageSet-Mt3620Uart1-2018.11.13-16.18.27-08:00"
}
Command completed successfully in 00:00:01.6562708.
```

The current image set is first in the list. Additional image sets are listed in the order in which they were added to the feed.

To redeploy an earlier image set, reassign the image set to the feed by using the **azsphere feed image-set add** command.

```
azsphere feed image-set add --feedid a48bb8cf-bee5-439c-a3fa-889c5f1c9807 --imagesetid d994d535-7ea0-45b1-
a790-8e6c8e37b15b

Adding image set with ID 'd994d535-7ea0-45b1-a790-8e6c8e37b15b' to feed with ID 'a48bb8cf-bee5-439c-a3fa-
889c5f1c9807'.

Successfully added image set with ID 'd994d535-7ea0-45b1-a790-8e6c8e37b15b' to feed with ID 'a48bb8cf-bee5-
439c-a3fa-889c5f1c9807'.

Command completed successfully in 00:00:02.2672744.
```

This example reassigns the image set named ImageSet-Mt3620Uart1-2018.11.13-16.18.27-08:00 to the feed, replacing ImageSet-Mt3620Uart1-2018.11.13-16.22.28-08:00. The contents of each image set are stored by the Azure Sphere Security Service, so you are not required to supply a path to the image or the image set.

# Recover the system software

5/30/2019 • 2 minutes to read

Recovery is the process of replacing the system software on the device using a special recovery bootloader instead of OTA update. The recovery process erases the contents of flash, replaces the system software, and reboots the device. As a result, application software and configuration data, including Wi-Fi credentials, are erased from the device. After you recover, you must reinstate the credentials (if any) and reconnect the device to the internet.

## IMPORTANT

Perform the recovery procedure only if instructed to do so by Microsoft. Recovery is required only when OTA download is not available.

To recover the system software:

1. Open an Azure Sphere Developer Command Prompt.
2. Ensure that your device is connected by USB to your computer.
3. Issue the **azsphere device recover** command:

```
azsphere device recover
```

You should see output similar to the following, although the number of images may differ.

```
azsphere device recover

Starting device recovery. Please note that this may take up to 10 minutes.
Board found. Sending recovery bootloader.
Erasing flash.
Sending images.
Sending image 1 of 16.
Sending image 2 of 16.
Sending image 3 of 16.
...
Sending image 16 of 16.
Finished writing images; rebooting board.
Device ID:
ABCDEF64D649176A4C5F26FE01EAD92F01BA0C50A20E9F6E441F7C5B66DF193E775524D70C7176AF2592F94729C9936FE4ABDDA9
B45B8B76123682509ABCDEF
    Device recovered successfully.
Command completed successfully in 00:02:39.9343076.
```

4. To continue using your device for development, run **azsphere device prep-debug** to re-enable application sideload and debugging.
5. To reconnect your device to Wi-Fi, [replace the Wi-Fi credentials](#) on the device.

# External MCU update reference solution

11/15/2018 • 2 minutes to read

If your product incorporates an Azure Sphere chip and another MCU you can use Azure Sphere to deploy updates to the external MCU. This makes use of an Azure Sphere application's [read-only resources](#) which can include firmware images for external MCUs.

The [External MCU Update nRF52 reference solution](#) demonstrates how to use Azure Sphere to update firmware on a Nordic nRF52 over a UART interface.

# Overview of azsphere

5/30/2019 • 2 minutes to read

The **azsphere.exe** command-line utility supports commands that manage Azure Sphere elements:

- [Components](#)
- [Devices](#)
- [Device groups](#)
- [Feeds](#)
- [Get support data](#)
- [Hardware definition](#)
- [Images](#)
- [Image sets](#)
- [Skus](#)
- [Show-version](#)
- [Tenants](#)

In addition, **azsphere.exe** provides [login](#) and [logout](#) commands to control access to your Azure Sphere tenant.

The **azsphere** command-line has the following format:

```
azsphere [command] [subcommand] operation [parameters]
```

In general, *command* and *subcommand* are nouns and *operation* is a verb, so that the combination identifies both an action and the object of the action. Most commands and operations have both a full name and an abbreviation. For example, the **device-group** command is abbreviated **dg**.

Most *parameters* have both a long name and an abbreviation. On the command line, introduce the long name with two hyphens and the abbreviation with a single hyphen. For example, the following two commands are equivalent:

```
azsphere device wifi add --ssid MyNetwork --key mynetworkkey
```

```
azsphere device wifi add -s MyNetwork -k mynetworkkey
```

Some commands allow multiple values for a single parameter, in which case you can either supply a parameter with each value, or a single parameter followed by a list of values separated by commas and no intervening spaces. For example, the following two commands are equivalent:

```
azsphere component publish --feedid ID --imagepath filepath-1 --imagepath filepath-2
```

```
azsphere component publish --feedid ID --imagepath filepath-1,filepath-2
```

# component, com

6/20/2019 • 6 minutes to read

Manages components and images for the Azure Sphere Security Service.

OPERATION	DESCRIPTION
<b>create</b>	Creates a new component
<b>image, img</b>	Manages images for a component
<b>list</b>	Lists all components
<b>publish</b>	Uploads a new image, adds it to an image set, and adds the image set to an existing feed

## create

Creates a new component, given a component ID and a name.

### Required parameters

PARAMETER	DESCRIPTION
-i, --componentid <i>GUID</i>	Specifies the ID of the component. This value appears in the <b>ComponentId</b> field of the app_manifest.json file for the application.
-n, --name <i>string</i>	Specifies a name for the component. Component names must be unique within an Azure Sphere tenant.

### Optional parameters

PARAMETER	DESCRIPTION
-t, --imagetype <i>ComponentImageType</i>	The image type for this component. If not provided, defaults to the Application image type. Values can be either <b>Application</b> or <b>BoardConfiguration</b> .

### ► Global parameters

#### Example

```
azsphere component create --componentid 83fac70c-072f-4f58-96bb-1be5c3557819 --name MT3620Uart1

Creating new component 'MT3620Uart1'.
Successfully created component 'MT3620Uart1' with ID '83fac70c-072f-4f58-96bb-1be5c3557819'.
Command completed successfully in 00:00:07.7393213.
```

## image, img

Manages the images that are part of a component.

OPERATION	DESCRIPTION
<b>add</b>	Uploads a new image and adds it to a component
<b>download</b>	Downloads an existing image
<b>show</b>	Displays information about an existing image

## image add

The **image add** operation uploads a new image to the cloud and adds it to a component. The component ID is not required because **azsphere component** reads it from the application manifest. Use the --autocreatecomponent (-c) flag to create the component if it does not already exist.

### Required parameters

PARAMETER	DESCRIPTION
-f, --filepath <i>path</i>	Specifies the path to the image file to upload.
--force	Forces the deployment of an image that uses a Beta API that may no longer be supported.

### Optional parameters

PARAMETER	DESCRIPTION
-c, --autocreatecomponent	Creates a new component for the image if one does not already exist.
-t, --temporary	Marks an image as temporary. This option is intended for use only during factory testing.

## ► Global parameters

### Example

```
azsphere component image add --autocreatecomponent --filepath "C:\Users\User\Documents\Visual Studio 2017\Projects\Mt3620BlinkEx\Mt3620BlinkEx\bin\ARM\Debug\MT3620BlinkEx.imagepackage"

Uploading image from file 'C:\Users\User\Documents\Visual Studio 2017\Projects\Mt3620BlinkEx\Mt3620BlinkEx\bin\ARM\Debug\MT3620BlinkEx.imagepackage':
--> Image ID: dc59be07-1feb-4be9-a5dc-42664dba4871
--> Component ID: 4275ecb3-5cf8-4147-9bfb-a7e8f3955e96
--> Component name: 'Mt3620BlinkEx'
Successfully uploaded image with ID 'dc59be07-1feb-4be9-a5dc-42664dba4871' to component 'Mt3620BlinkEx' with ID '4275ecb3-5cf8-4147-9bfb-a7e8f3955e96'.
Command completed successfully in 00:00:20.4555421.
```

## image download

The **image download** operation downloads a copy of an image that has already been added to a component.

### Required parameters

PARAMETER	DESCRIPTION
-i, --imageid <i>GUID</i>	Specifies the image ID of the image to download.

PARAMETER	DESCRIPTION
-o, --output <i>path</i>	Specifies the path and filename in which to save the image. Path can be relative to the current directory.

## ► Global parameters

### Example

```
azsphere component image download --imageid dc59be07-1feb-4be9-a5dc-42664dba4871 --output
"BlinkEx.imagepackage"

Getting the image with ID 'dc59be07-1feb-4be9-a5dc-42664dba4871'.
Successfully downloaded image to location 'BlinkEx.imagepackage'.
Command completed successfully in 00:00:08.7115126.
```

## image show

The **image show** operation downloads the metadata for an image that has already been added to a component.

### Required parameters

PARAMETER	DESCRIPTION
-i, --imageid <i>GUID</i>	Specifies the image ID of the image for which to show metadata.

## ► Global parameters

### Example

```
azsphere component image show --imageid 73093019-617d-4458-a8bc-d0c7a3c75a11

Getting the metadata for image with ID '73093019-617d-4458-a8bc-d0c7a3c75a11'.
Successfully retrieved image metadata and status:
-> Image ID: 73093019-617d-4458-a8bc-d0c7a3c75a11
-> Component ID: 54acba7c-7719-461a-89db-49c807e0fa4d
-> Name: Mt3620Blink1
-> Description:
-> Signing status: "Succeeded"
Command completed successfully in 00:00:08.3633986.
```

## list

Displays the names, IDs, and types of all components in the Azure Sphere tenant.

## ► Global parameters

### Example

```

azsphere component list
Listing all components.
Retrieved components:
--> [11be994e-30a9-48c2-ba89-88b0d2987d70] 'Trusted Keystore'
    --> Image Type: System software image type 19
--> [123455cc-fa2b-4e5d-95fe-b7fda7ef6e2d] 'Update Certificate Store'
    --> Image Type: System software image type 22
--> [16bf62d0-f47e-11e6-839c-00155d9f1e00] 'Pluton Runtime'
    --> Image Type: System software image type 2
--> [318811ba-76df-4851-b668-ccfa1683690f] 'Base System Update Manifest'
    --> Image Type: System software image type 23
--> [32fc880c-f31f-471b-a4b5-91585b66b37e] 'Secure World Kernel'
    --> Image Type: System software image type 4
--> [48a22e96-d078-4e34-9d7a-91b3404031da] 'azured'
    --> Image Type: System software image type 9
--> [4ec06498-0025-4443-87d9-b10ca7e563e7] 'Recovery Manifest'
    --> Image Type: System software image type 26
--> [641f94d9-7600-4c5b-9955-5163cb7f1d75] 'gatewayd'
    --> Image Type: System software image type 9
--> [6904e268-2627-5ae4-92f2-96176db30269] 'Security Monitor Policy'
    --> Image Type: System software image type 20
--> [7ba05ff7-7835-4b26-9eda-29af0c635280] 'networkd'
    --> Image Type: System software image type 9
--> [89ecd022-0bdd-4767-a527-d756dd784a19] 'rng-tools'
    --> Image Type: System software image type 9
--> [960f32d4-38a3-47cc-82a8-3dc30dac95b4] 'Update Manifest Set'
    --> Image Type: System software image type 27
--> [a87d9f43-e240-4dab-8a85-54512ddffe00] 'N9 Wifi Firmware'
    --> Image Type: System software image type 3
--> [ae01da67-d18b-44f4-9cf0-5dc81d51fdc7] 'A7 NW Loader'
    --> Image Type: System software image type 5
--> [c1390819-1058-4e2e-ab7d-23f4079cfa0b] 'Firmware Update Manifest'
    --> Image Type: System software image type 24
--> [c43a90e4-cfa5-4b50-af28-b611032fd4d5] 'NW Device Tree'
    --> Image Type: System software image type 6
--> [c92ff430-2613-4077-a084-d79cc7c15e3b] '1BL'
    --> Image Type: System software image type 1
--> [d3cbbdfb-4e4d-465d-a297-12a6a7f46c25] 'Customer Update Manifest'
    --> Image Type: System software image type 25
--> [ec96028b-080b-4ff5-9ef1-b40264a8c652] 'NW Kernel'
    --> Image Type: System software image type 7
--> [f7fd0c88-d005-45c6-ac4b-88afdbf2dc6a] 'NW Root Filesystem'
    --> Image Type: System software image type 8
--> [9fd529ac-9aa0-4aa5-a657-631a445db317] 'Mt3620Blink1'
    --> Image Type: Application
Command completed successfully in 00:00:01.9113890.

```

## publish

Publishes a new image set for an existing component to an existing feed.

The **publish** operation uploads a new image, adds it to a new image set, and adds the new image set to an existing feed. It combines the tasks of several other **azsphere** commands:

- **azsphere component image add**
- **azsphere image-set create**
- **azsphere feed image-set add**

### Required parameters

PARAMETER	DESCRIPTION
-----------	-------------

PARAMETER	DESCRIPTION
-f, --feedid <i>GUID</i>	Specifies the GUID of the feed to which to add the image set. The feed must already exist and must deliver a component that has the same component ID as the specified image.
--force	Forces the deployment of an image that uses a Beta API that may no longer be supported.
-i, --imagepath <i>filepath</i>	Specifies the path and filename of the image to upload. The command auto-generates a component ID based on the information in the application manifest if a component with that ID does not already exist. If you are publishing multiple images to the image set, either use one --imagepath <i>filepath</i> for each image, or use a single --imagepath and separate the paths with commas and no intervening spaces. At least one image path is required.

#### Optional parameters

PARAMETER	DESCRIPTION
-n, --newimagesetname <i>string</i>	Specifies a name for the new image set to be created. The name must be unique within the tenant. If omitted, a name is generated from the information in the application manifest. Optional.

### ► Global parameters

#### Example

The following example uploads a new image for the existing Mt3620Blink3 component, creates an image set, and adds the image set to a feed that delivers the Mt3620Blink3 component.

```
azsphere component publish --feedid d755a1b9-192d-4769-aad5-5d579178242f --imagepath
"C:\Users\User\Source\Repos\Mt3620Blink3\Mt3620Blink3\bin\ARM\Debug\Mt3620Blink3.imagepackage"

Publishing image
'C:\Users\User\Source\Repos\Mt3620Blink3\Mt3620Blink3\bin\ARM\Debug\Mt3620Blink3.imagepackage' to feed with ID
'd755a1b9-192d-4769-aad5-5d579178242f'.
Uploading image from file
'C:\Users\User\Source\Repos\Mt3620Blink3\Mt3620Blink3\bin\ARM\Debug\Mt3620Blink3.imagepackage':
--> Image ID:      '4eb71b48-16a2-4f31-a338-bb8c0e5f7386'
--> Component ID:  '16995a70-377f-4bd2-b29d-1b0fffcbe287'
--> Component name: 'Mt3620Blink3'
Creating new image set with name 'ImageSet-Mt3620Blink3-2018.04.26-14.33.07' for image with ID '4eb71b48-16a2-
4f31-a338-bb8c0e5f7386'.
Adding image set with ID 'adfd80e1-43d9-4359-99fe-31df0c834d7a' to feed with ID 'd755a1b9-192d-4769-aad5-
5d579178242f'.
Successfully published image
'C:\Users\User\Source\Repos\Mt3620Blink3\Mt3620Blink3\bin\ARM\Debug\Mt3620Blink3.imagepackage' to feed with ID
'd755a1b9-192d-4769-aad5-5d579178242f'.
Command completed successfully in 00:00:11.3067837.
```

# device, dev

10/9/2019 • 34 minutes to read

Manages Azure Sphere devices.

OPERATION	DESCRIPTION
<b>capability, cap</b>	Manages device capabilities
<b>claim</b>	Claims a previously unclaimed device to the Azure Sphere tenant
<b>image, img</b>	Manages images for a device
<b>link-feed</b>	Links a device to a feed
<b>manufacturing-state, mfg</b>	Manages the manufacturing state of the attached device
<b>prep-debug</b>	Sets up a device for local debugging
<b>prep-field</b>	Sets up a device to disable debugging and receive over-the-air (OTA) updates
<b>recover</b>	Uses special recovery mode to load new firmware onto the device
<b>restart</b>	Restarts the attached device
<b>show-attached</b>	Displays details about the attached device from the device itself
<b>show-ota-config</b>	Displays details about the OTA update configuration of the device
<b>show-ota-status</b>	Displays the status of the most recent OTA update
<b>sideload, sl</b>	Loads an application onto the attached device or changes the status of the application
<b>update-device-group</b>	Moves a device into a device group
<b>update-sku</b>	Sets the product SKU for a device
<b>wifi</b>	Manages the Wi-Fi configuration for the device

## capability, cap

Manages device capabilities.

Currently, the only capability is **appdevelopment**, which enables you to sideload SDK-signed image packages to the device and to start, stop, debug, or delete any image package from the device.

OPERATION	DESCRIPTION
<b>download</b>	Downloads a device capability configuration from the Azure Sphere Security Service
<b>show-attached</b>	Displays the capability configuration for the attached device
<b>update</b>	Applies a device capability configuration to the attached device

## capability download

Downloads a device capability from the Azure Sphere Security Service and applies it to a device.

The **appdevelopment** capability lets you sideload SDK-signed applications and start, stop, debug, or delete any application on the device. Without the **appdevelopment** capability, only OTA-deployed applications can be loaded and start, stop, debug, and delete are prohibited.

### Required parameters

PARAMETER	DESCRIPTION
<code>-t, --type capability-type</code>	Specifies the type of capability, either <b>appdevelopment</b> or <b>none</b> . Use <b>none</b> to remove the <b>appdevelopment</b> capability.

### Optional parameters

PARAMETER	DESCRIPTION
<code>-i, --deviceid id</code>	Specifies the ID of the device for which to get the capability configuration. If you specify a device ID, you must also use <code>--output</code> . If omitted, gets a capability for the attached device.
<code>-ip, --deviceip</code>	[Multi-Device] Specifies the IP address of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.
<code>-l, --devicelocation</code>	[Multi-Device] Specifies the FTDI location ID of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.
<code>-o, --output path</code>	Specifies the path and filename at which to save the capability configuration. Include a path, even for the current directory. Required with the <code>--deviceid</code> parameter.
<code>-p, --apply</code>	Applies the device capability configuration to the attached device. Do not use with <code>--deviceid</code> or <code>--output</code> .

## ► Global parameters

### Example

```
azsphere device capability download --type appdevelopment --output ./appdevcap

Downloading device capability configuration for device ID <device ID>.
Successfully downloaded device capability configuration.
Successfully wrote device capability configuration file './appdevcap'.
Command completed successfully in 00:00:07.7393213.
```

## capability show-attached

Displays the capability configuration for the attached device.

### Optional parameters

PARAMETER	DESCRIPTION
-ip, --deviceip	[Multi-Device] Specifies the IP address of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.
-l, --devicelocation	[Multi-Device] Specifies the FTDI location ID of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.

### ► Global parameters

#### Example

```
azsphere device capability show-attached
Device Capabilities:
    Enable App development
Command completed successfully in 00:00:00.8746160.
```

## capability update

Applies a device capability configuration to the attached device.

### Required parameters

PARAMETER	DESCRIPTION
-f, --filepath <i>path</i>	Specifies the path and name of the device capability file to apply.

### Optional parameters

PARAMETER	DESCRIPTION
-ip, --deviceip	[Multi-Device] Specifies the IP address of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.
-l, --devicelocation	[Multi-Device] Specifies the FTDI location ID of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.

### ► Global parameters

#### Example

```
azsphere device capability update --filepath appdevcap

Applying device capability configuration to device.
Successfully applied device capability configuration to device.
The device is rebooting.
Command completed successfully in 00:00:07.7393213.
```

## claim

Claims a previously unclaimed device for the current Azure Sphere tenant.

### IMPORTANT

Before you claim the device, ensure that you are signed in to the correct Azure Sphere tenant. A device can be claimed only once. Once claimed, a device cannot be moved to a different tenant.

#### Optional parameters

PARAMETER	DESCRIPTION
-i, --deviceid <i>GUID</i>	Specifies the ID of the device to claim. If omitted, <b>azsphere</b> claims the attached device.
-ip, --deviceip	[Multi-Device] Specifies the IP address of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.
-l, --devicelocation	[Multi-Device] Specifies the FTDI location ID of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.

#### ► Global parameters

##### Example

```
azsphere device claim

Claiming device.
Claiming attached device ID <device ID> into tenant ID 'd343c263-4aa3-4558-adbb-d3fc34631800'.
Successfully claimed device ID <device ID> into tenant 'Microsoft' with ID 'd343c263-4aa3-4558-adbb-d3fc34631800'.
Command completed successfully in 00:00:05.5459143.
```

## image, img

Returns information about the images that are installed on or targeted to the attached device.

OPERATION	DESCRIPTION
<b>list-installed</b>	Provides details about the images that are currently installed on the attached device
<b>list-targeted</b>	Provides details about the images that are targeted to the attached device.

## image list-installed

Lists the images that are installed on the attached device. The list of information includes the component and image IDs.

### Optional parameters

PARAMETER	DESCRIPTION
-f, --full	Lists both customer and system software images that are installed on the device. By default, lists only customer application images, debuggers, and board configuration images.
-ip, --deviceip	[Multi-Device] Specifies the IP address of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.
-l, --devicelocation	[Multi-Device] Specifies the FTDI location ID of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.

### ► Global parameters

#### Example

```
azsphere device image list-installed
Installed images:
--> gdbserver
--> Image type: Application
--> Component ID: 8548b129-b16f-4f84-8dbe-d2c847862e78
--> Image ID: 43d2707f-0bc7-4956-92c1-4a3d0ad91a74
--> Mt3620Blink4
--> Image type: Application
--> Component ID: 970d2ff1-86b4-4e50-9e80-e5af2845f465
--> Image ID: e53ce989-0ecf-493d-8056-fc0683a566d3
Command completed successfully in 00:00:01.6189314.
```

## image list-targeted

Lists the images that have been uploaded to the Azure Sphere Security Service and will be installed the next time the device is updated.

### Optional parameters

PARAMETER	DESCRIPTION
-f, --full	Lists both customer and system software images that will be installed on the device. By default, lists only customer application images.
-i, --deviceid <i>GUID</i>	Specifies the ID of the device for which to list targeted images. By default, lists images for the attached device.
-ip, --deviceip	[Multi-Device] Specifies the IP address of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.

PARAMETER	DESCRIPTION
-l, --devicelocation	[Multi-Device] Specifies the FTDI location ID of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.

## ► Global parameters

### Example

```
azsphere device image list-targeted

Successfully retrieved the current image set for device with ID <device ID> from your Azure Sphere tenant:
--> ID: [6e9cdc9d-c9ca-4080-9f95-b77599b4095a]
--> Name: 'ImageSet-Mt3620Blink1-2018.07.19-18.15.42'
Images to be installed:
--> [ID: 116c0bc5-be17-47f9-88af-8f3410fe7efa]
Command completed successfully in 00:00:04.2733444.
```

## link-feed

Links a device to a feed and optionally adds a new image to the feed.

### Optional parameters

PARAMETER	DESCRIPTION
-c, --componentid <i>GUID</i>	Specifies the ID of an existing component that the feed will deliver. Either --componentid or --imagepath is required with --newfeedname. You may add multiple components to the feed by either using this flag multiple times to specify multiple components, or once and separate the component IDs with commas and no intervening spaces.
-d, --dependentfeedid <i>GUID</i>	Specifies the ID of the system software feed that the new feed depends on. Required with --newfeedname.
-f --feedid <i>GUID</i>	Specifies the ID of an existing feed to link to the device group for the specified device. If omitted, creates a new feed and assigns it the name in the optional --newfeedname parameter. If you already have a feed for this component, use it to avoid cluttering your tenant with redundant feeds. Either --feedid or --newfeedname is required.
--force	Forces the deployment of an image that uses a Beta API that may no longer be supported.
-i, --deviceid <i>GUID</i>	Specifies the ID of the device to which to link the feed. If omitted, links the feed to the attached device.
-ip, --deviceip	[Multi-Device] Specifies the IP address of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.
-l, --devicelocation	[Multi-Device] Specifies the FTDI location ID of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.

PARAMETER	DESCRIPTION
-p, --imagepath <i>path</i>	Specifies the path to an image package to upload to the feed and validates the component ID of the image package against the component ID for the feed. If omitted, creates a new feed but does not assign an initial image set. Either --imagepath or --componentid is required with --newfeedname. You may add multiple image paths to the feed by either using this flag multiple times to specify multiple image paths or using it once and separating the image paths with commas and no intervening spaces.
-n, --newfeedname <i>string</i>	Specifies a name for the feed. Requires --dependentfeedid. Feed names must be unique within an Azure Sphere tenant. Either --newfeedname or --feedid is required.
-s, --newimagesetname <i>string</i>	Specifies the name for the new image set to create. If omitted, generates an image set name based on the component name and a time stamp. Requires --imagepath.

## ► Global parameters

### Examples

This example creates a new feed and supplies an image package for the feed to deliver.

```
azsphere device link-feed --dependentfeedid 3369f0e1-dedf-49ec-a602-2aa98669fd61 --imagepath
"C:\Users\User\Documents\Visual Studio
2017\Projects\Mt3620Blink5\Mt3620Blink5\bin\ARM\Debug\Mt3620Blink5.imagepackage" --newfeedname
Model100AppFeedOTA --newimagesetname Model100Appv1.0

Getting the details for device with ID <device ID>.
Uploading image from file 'C:\Users\User\Documents\Visual Studio
2017\Projects\Mt3620Blink5\Mt3620Blink5\bin\ARM\Debug\Mt3620Blink5.imagepackage':
--> Image ID:      'b66f1398-4ad6-4f12-be84-8ad607676ec3'
--> Component ID:  '8fb105a3-35d0-423a-9427-a21852623965'
--> Component name: 'Mt3620Blink5'
Create a new feed with name 'Model100AppFeedOTA'.
Adding feed with ID '8e2d3b19-bb01-4e36-b974-5b2f8df502e9' to device group with ID 'c0346077-eb9e-4dbc-85ad-
03313867be69'.
Creating new image set with name 'Model100Appv1.0' for image with ID 'b66f1398-4ad6-4f12-be84-8ad607676ec3'.
Adding image set with ID '70207e9a-d080-42d7-899e-fb02822fbc32' to feed with ID '8e2d3b19-bb01-4e36-b974-
5b2f8df502e9'.
Successfully linked device <device ID> to feed with ID '8e2d3b19-bb01-4e36-b974-5b2f8df502e9'.
Command completed successfully in 00:00:25.5193828.
```

The next example creates a new feed for an existing component. The feed depends on the Retail Azure Sphere OS feed and is named BlinkLink. The feed services the devices in the same device group as the attached device.

```
azsphere device link-feed --componentid 16995a70-377f-4bd2-b29d-1b0fffcbe287 --newfeedname BlinkLink --
dependentfeedid 3369f0e1-dedf-49ec-a602-2aa98669fd61

Getting the details for device with ID <device ID>.
Create a new feed with name 'BlinkLink'.
Adding feed with ID '6daf66aa-5a3e-41f9-9659-6c30a2f7673f' to device group with ID 'd80bf785-acae-497c-a62c-
21a6ce65b81f'.
Successfully linked device <device ID> to feed with ID '6daf66aa-5a3e-41f9-9659-6c30a2f7673f'.
Command completed successfully in 00:00:07.1315217.
```

Manages the manufacturing state of the attached device.

OPERATION	DESCRIPTION
<b>show</b>	Displays the manufacturing state of the attached device
<b>update</b>	Updates the manufacturing state of the attached device if it is not already AllComplete.

**Caution**

Manufacturing state changes are permanent and irreversible.

### **manufacturing-state show**

Displays the manufacturing state of the attached device.

#### **Optional parameters**

PARAMETER	DESCRIPTION
-ip, --deviceip	[Multi-Device] Specifies the IP address of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.
-l, --devicelocation	[Multi-Device] Specifies the FTDI location ID of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.

## ► Global parameters

### **Example**

```
azsphere device manufacturing-state show
Manufacturing State: Blank
Command completed successfully in 00:00:00.9005143.
```

### **manufacturing-state update**

Updates the manufacturing state of the attached device if the state is not already AllComplete.

This command is intended for use during the manufacturing process.

**Caution**

Manufacturing state changes are permanent and irreversible.

#### **Optional parameters**

PARAMETER	DESCRIPTION
-ip, --deviceip	[Multi-Device] Specifies the IP address of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.
-l, --devicelocation	[Multi-Device] Specifies the FTDI location ID of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.
-s, --state SettableManufacturingStates	Specifies the manufacturing state to set for the device. Currently the only settable state is AllComplete.

## ► Global parameters

### Example

```
azsphere device manufacturing-state update --state AllComplete
Manufacturing State: AllComplete
Command completed successfully in 00:00:00.9005143.
```

## prep-debug

Sets up the attached device for local debugging and disables over-the-air application updates.

Specifically, **prep-debug**:

- Downloads and applies the **appdevelopment** capability for the attached device
- Assigns the device to a device group that does not enable over-the-air application updates
- Reboots the device

If the command reports an error, it may suggest that you claim the device. First, use **azsphere tenant show-selected** to ensure that you are logged in to the intended Azure Sphere tenant, and then **azsphere login** to log into a different tenant if necessary. To claim the device, use **azsphere device claim**. If you see the error but have already claimed the device, make sure that you are logged in to the tenant in which you claimed the device.

### Optional parameters

PARAMETER	DESCRIPTION
-d, --devicegroupid <i>GUID</i>	Specifies the ID of a device group that does not apply over-the-air application updates. If omitted, assigns the device to a default group.
-ip, --deviceip	[Multi-Device] Specifies the IP address of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.
-l, --devicelocation	[Multi-Device] Specifies the FTDI location ID of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.
-r, --enablertcoredebugging	Installs drivers required to debug applications running on a real-time core. This option requires administrator permissions.

## ► Global parameters

### Example

```

azsphere device prep-debug --enablertcoredebugging

Installing USB drivers to enable real-time core debugging.
Drivers installed for all attached devices
Getting device capability configuration for application development.
Downloading device capability configuration for device ID <device ID>.
Successfully downloaded device capability configuration.
Successfully wrote device capability configuration file
'C:\Users\Administrator\AppData\Local\Temp\tmpD15E.tmp'.
Setting device group ID '63bbe6ea-14be-4d1a-a6e7-03591d882b42' for device with ID <device ID>.
Successfully disabled over-the-air updates.
Enabling application development capability on attached device.
Applying device capability configuration to device.
Successfully applied device capability configuration to device.
The device is rebooting.
Application development capability enabled.
Successfully set up device <device ID> for application development, and disabled over-the-air updates.
Command completed successfully in 00:00:07.7393213.

```

## prep-field

Readies the attached device for field use by disabling application development, deleting any existing applications, and enabling over-the-air application updates.

It requires a device to be attached to the PC and operates only on the attached device.

The specific tasks that **prep-field** performs depend on the whether a product SKU and device group have already been assigned for this device. If the product SKU or the device group does not already exist, the command creates it and assigns it to the device, provided that a SKU name or a device group name is supplied with the appropriate parameter.

### Optional parameters

PARAMETER	DESCRIPTION
-r, --devicegroupid <i>GUID</i>	Specifies the ID of the device group to which to assign the device. If omitted, assigns the device to a default group that enables application updates. Cannot be used with --newdevicegroupname.
-g, --newdevicegroupname <i>string</i>	Creates a new device group that allows over-the-air application updates. Device group names must be unique within an Azure Sphere tenant. Cannot be used with --devicegroupid.
-ip, --deviceip	[Multi-Device] Specifies the IP address of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.
-l, --devicelocation	[Multi-Device] Specifies the FTDI location ID of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.
-n, --newskuname <i>string</i>	Creates a new product SKU and assigns it to the device. SKU names must be unique within an Azure Sphere tenant. Cannot be used with --skuid.

PARAMETER	DESCRIPTION
-d, --skudescription <i>string</i>	Provides a friendly description of the new SKU. Requires --newskuname.
-s, --skuid <i>GUID</i>	Specifies the ID of an existing product SKU to apply to the device. Cannot be used with --newskuname.

## ► Global parameters

### Examples

Example 1. Create a product SKU and device group for device

This example creates a new product SKU and a new device group, and assigns both to the attached device.

```
azsphere device prep-field --newdevicegroupname AppUpdateGroup --newskuname TestSKU
```

As the output shows, the command deletes the existing application from the device, removes the **appdevelopment** capability, creates a device group named AppUpdateGroup, creates a product SKU named TestSKU, and assigns both the device group and the product SKU to the attached device. The new device group is enabled for OTA loading of application updates.

```
Removing applications from device.
Component '54acba7c-7719-461a-89db-49c807e0fa4d' deleted.
Locking device.
Downloading device capability configuration for device ID <device ID>.
Successfully downloaded device capability configuration.
Applying device capability configuration to device.
Successfully applied device capability configuration to device.
The device is rebooting.
Successfully locked device.
Creating a new device group with name 'AppUpdateGroup'.
Setting device group ID 'bbc91f02-1de4-43a3-bcf2-f6f0994ac723' for device with ID <device ID>.
Creating a new SKU with name 'TestSKU'.
Setting product SKU to '5d88f658-be0c-4814-8319-473f21f4f88f' for device with ID <device ID>.
Successfully set up device <device ID> for OTA loading.
Command completed successfully in 00:00:18.0072081.
```

Example 2. Assign existing product SKU and device group to device

This example assigns an existing product SKU and device group to the attached device.

```
azsphere device prep-field --skuid 2bc8c605-6f8f-4802-ba69-c57d63e9c6dd --devicegroupid d80bf785-acae-497c-a62c-21a6ce65b81f

Getting the details of device group with ID ''.
Removing applications from device.
No app present.
Successfully removed applications from device.
Locking device.
Downloading device capability configuration for device ID <device ID>.
Successfully downloaded device capability configuration.
Applying device capability configuration to device.
Successfully applied device capability configuration to device.
The device is rebooting.
Successfully locked device.
Setting device group ID 'd80bf785-acae-497c-a62c-21a6ce65b81f' for device with ID <device ID>.
Setting product SKU to '2bc8c605-6f8f-4802-ba69-c57d63e9c6dd' for device with ID <device ID>.
Successfully set up device <device ID> for OTA loading.
Command completed successfully in 00:00:12.0638169.
```

### Example 3. Assign device to different device group

This example is similar to the preceding example, but retains the existing product SKU for the device. Here the **prep-field** operation changes the device group to which the device belongs and removes the appDevelopment capability. This command is useful for moving a device from a development environment that does not enable OTA application updates to a production environment that does.

```
azsphere device prep-field --devicegroupid 655d7b12-07ad-4e8a-b104-c0ec494b8489

Getting the product SKU for device with ID <device ID>.
Getting the details of device group with ID ''.
Removing applications from device.
Component '54acba7c-7719-461a-89db-49c807e0fa4d' deleted.
Successfully removed applications from device.
Locking device.
Downloading device capability configuration for device ID <device ID>.
Successfully downloaded device capability configuration.
Applying device capability configuration to device.
Successfully applied device capability configuration to device.
The device is rebooting.
Successfully locked device.
Setting device group ID '655d7b12-07ad-4e8a-b104-c0ec494b8489' for device with ID <device ID>.
Successfully set up device <device ID> for OTA loading.
Command completed successfully in 00:00:11.1988981.
```

## recover

Replaces the system software on the device.

### Optional parameters

PARAMETER	DESCRIPTION
-c, --capability <i>filename</i>	Specifies the filename of the device capability image to apply to the device. For Microsoft use only.
-i, --images <i>folder</i>	Specifies the path to a folder that contains the image packages to write to the device. By default, recovery uses the images in the SDK unless an alternate path is provided with this flag.
-ip, --deviceip	[Multi-Device] Specifies the IP address of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.
-l, --devicelocation	[Multi-Device] Specifies the FTDI location ID of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.

### ► Global parameters

#### Example

```
azsphere device recover --images .\recovery`
```

## restart

Restarts the attached device.

#### Optional parameters

PARAMETER	DESCRIPTION
-ip, --deviceip	[Multi-Device] Specifies the IP address of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.
-l, --devicelocation	[Multi-Device] Specifies the FTDI location ID of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.

#### ► Global parameters

##### Example

```
azsphere device restart
Restarting device.
Device restarted successfully.
Command completed successfully in 00:00:10.2668755.
```

## show-attached

Displays information about the attached device from the device itself. These details differ from those that the Azure Sphere Security Service stores for the device.

#### Optional parameters

PARAMETER	DESCRIPTION
-ip, --deviceip	[Multi-Device] Specifies the IP address of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.
-l, --devicelocation	[Multi-Device] Specifies the FTDI location ID of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.

#### ► Global parameters

##### Example

```
Device ID: <device ID>
IP address: 192.168.35.2
Connection path: 1423
Command completed successfully in 00:00:03.3654153.
```

## show-ota-config

Displays information that the Azure Sphere Security Service stores for a device. These details differ from those that the device itself stores.

#### Optional parameters

PARAMETER	DESCRIPTION
-----------	-------------

PARAMETER	DESCRIPTION
-i, --deviceid <i>GUID</i>	Specifies the ID of the device. By default, shows information about the attached device.
-ip, --deviceip	[Multi-Device] Specifies the IP address of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.
-l, --devicelocation	[Multi-Device] Specifies the FTDI location ID of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.

## ► Global parameters

### Example

```
azsphere device show-ota-config
Retrieved the over-the-air update configuration for device with ID <device ID>:
--> Device group: 'System Software' with ID '63bbe6ea-14be-4d1a-a6e7-03591d882b42'
--> SKU: '9d606c43-1fad-4990-b207-554a025e0869' of type 'Chip'
--> SKU: '946410a0-0057-4b11-af68-d56a684f6681' of type 'Product'
Command completed successfully in 00:00:03.0219123.
```

## show-ota-status

Displays the status of the most recent OTA update for the device.

Use this command to find out which version of the Azure Sphere OS your device is running or whether the current OTA update has completed.

### Optional parameters

PARAMETER	DESCRIPTION
-ip, --deviceip	[Multi-Device] Specifies the IP address of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.
-l, --devicelocation	[Multi-Device] Specifies the FTDI location ID of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.

## ► Global parameters

### Example

```
azsphere device show-ota-status
The Azure Sphere Security Service is targeting this device with OS version <versionNumber>
Your device has the expected version of the Azure Sphere OS: <versionNumber>.
Command completed successfully in 00:00:03.8184016.
```

In this example, <versionNumber> represents the current operating system version and is a changeable value.

## sideload, sl

Manages the application on the device.

Many of the sideload options require the **appdevelopment** capability, which can be acquired by using **prep-debug**. To sideload an SDK-signed application, or to start, stop, debug, or delete an SDK-signed application or a production-signed application, the device must have the **appdevelopment** capability.

You can use **sideload stop** and **sideload start** to restart a running real-time capable application to determine which real-time core it is running on.

OPERATION	DESCRIPTION
<b>delete</b>	Deletes the current application from the device.
<b>deploy</b>	Loads an application onto the device.
<b>show-quota</b>	Displays the amount of storage used by the current application on the device.
<b>show-status</b>	Returns the status of the current application on the device.
<b>start</b>	Starts the application that is loaded on the device.
<b>stop</b>	Stops the application that is running on the device.

## **sideload delete**

Deletes applications from the device.

### Optional parameters

PARAMETER	DESCRIPTION
<code>-i, --componentid <i>GUID</i></code>	Specifies the ID of the component to delete. If omitted, deletes all applications.
<code>-ip, --deviceip</code>	[Multi-Device] Specifies the IP address of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.
<code>-l, --devicelocation</code>	[Multi-Device] Specifies the FTDI location ID of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.

## ► Global parameters

### Example

```
azsphere device sideload delete
Component '54acba7c-7719-461a-89db-49c807e0fa4d' deleted.
```

## **sideload deploy**

Loads an application onto the attached device and starts the application.

This command fails if the application manifest requests a resource that is being used by an application that is already on the device. In this case, use **azsphere device sideload delete** to delete the existing application and then try sideloading again. The **azsphere device sideload delete** command will delete both high level applications and real time applications; to delete one application and leave the other specify the component ID of the application to delete.

The same command is used to deploy both high-level applications and real-time capable applications.

#### Required parameters

PARAMETER	DESCRIPTION
-p, --imagepackage <i>path</i>	Specifies the path and filename of the image package to load on the device. Sideload deployment will fail if the device does not have the <b>appdevelopment</b> capability.

#### Optional parameters

PARAMETER	DESCRIPTION
--force	Forces the deployment of an image that uses a Beta API that may no longer be supported.
-ip, --deviceip	[Multi-Device] Specifies the IP address of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.
-l, --devicelocation	[Multi-Device] Specifies the FTDI location ID of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.
-m, --manualstart	Does not start the application after loading it.

### ► Global parameters

#### Example

```
azsphere device sideload deploy --imagepackage "C:\Users\Test\Documents\Visual Studio 2017\Projects\Mt3620Blink6\Mt3620Blink6\bin\ARM\Debug\Mt3620Blink6.imagepackage"
Deploying 'C:\Users\Test\Documents\Visual Studio 2017\Projects\Mt3620Blink6\Mt3620Blink6\bin\ARM\Debug\Mt3620Blink6.imagepackage' to the attached device.
Image package 'C:\Users\Test\Documents\Visual Studio 2017\Projects\Mt3620Blink6\Mt3620Blink6\bin\ARM\Debug\Mt3620Blink6.imagepackage' has been deployed to the attached device.
Command completed successfully in 00:00:03.0567304.
```

## sideload show-quota

Displays the amount of mutable storage allocated and in use on the attached device.

You set the **mutable storage** quota in the application manifest, and the Azure Sphere OS enforces quotas when it allocates sectors for the file. As a result, if you decrease the **MutableStorage** value, the amount of storage in use will not change, but the allocated value reported will be different. For example, if the application has already used 16 KB and you change the **MutableStorage** value to 8, the command reports that the application uses 16 KB of 8 KB allocated. The data remains on the device.

#### Optional parameters

PARAMETER	DESCRIPTION
-i, --componentid <i>GUID</i>	Specifies the ID of the component for which to return storage information. If omitted, displays information for all applications.

PARAMETER	DESCRIPTION
-ip, --deviceip	[Multi-Device] Specifies the IP address of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.
-l, --devicelocation	[Multi-Device] Specifies the FTDI location ID of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.

## ► Global parameters

### Example

```
C:\>azsphere device sideload show-quota
8548b129-b16f-4f84-8dbe-d2c847862e78: No mutable storage allocated.
Command completed successfully in 00:00:02.3003524.

C:\> azsphere.exe device sideload deploy -p Mt3620Blink1 Mutable.imagepackage
Deploying 'Mt3620Blink1 Mutable.imagepackage' to the attached device.
Image package 'Mt3620Blink1 Mutable.imagepackage' has been deployed to the attached device.
Command completed successfully in 00:00:04.8939438.

C:\>azsphere device sideload show-quota
8548b129-b16f-4f84-8dbe-d2c847862e78: No mutable storage allocated.
ee8abc15-41f3-491d-a4c7-4af49948e159: 0KB out of 16KB of mutable storage used.
Command completed successfully in 00:00:02.0410841.
```

## sideload show-status

Displays the current status of the applications on the device.

### Optional parameters

PARAMETER	DESCRIPTION
-i, --componentid <i>GUID</i>	Specifies the ID of the component for which to display status. If omitted, shows status of all components.
-ip, --deviceip	[Multi-Device] Specifies the IP address of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.
-l, --devicelocation	[Multi-Device] Specifies the FTDI location ID of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.

## ► Global parameters

### Example

```
azsphere device sideload show-status
54acba7c-7719-461a-89db-49c807e0fa4d: App state: running

Command completed successfully in 00:00:01.1103343.
```

## sideload start

Starts applications on the device.

## Optional parameters

PARAMETER	DESCRIPTION
-i, --componentid <i>GUID</i>	Specifies the ID of the component to start. If omitted, starts all applications.
-d, --debug	Starts the application for debugging. Requires --componentid.

## ► Global parameters

### Example

```
azsphere device sideload start  
54acba7c-7719-461a-89db-49c807e0fa4d: App state: running  
  
Command completed successfully in 00:00:01.1183407.
```

## sideload stop

Stops the applications on the device.

## Optional parameters

PARAMETER	DESCRIPTION
-i, --componentid <i>GUID</i>	Specifies the ID of the component to stop. If omitted, stops all applications.

## ► Global parameters

### Example

```
azsphere device sideload stop  
54acba7c-7719-461a-89db-49c807e0fa4d: App state: stopped  
  
Command completed successfully in 00:00:01.1210256.
```

## update-device-group

Moves the device into a different device group in your Azure Sphere tenant.

## Required parameters

PARAMETER	DESCRIPTION
-d, --devicegroupid <i>GUID</i>	Specifies the ID of the device group to which to move the device.

## Optional parameters

PARAMETER	DESCRIPTION
-i, --deviceid <i>GUID</i>	Specifies the ID of the device to move. By default, moves the attached device.
-ip, --deviceip	[Multi-Device] Specifies the IP address of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.

PARAMETER	DESCRIPTION
-l, --devicelocation	[Multi-Device] Specifies the FTDI location ID of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.

## ► Global parameters

### Example

```
azsphere device update-device-group --devicegroupid 63bbe6ea-14be-4d1a-a6e7-03591d882b42
Successfully moved device <device ID> to device group '63bbe6ea-14be-4d1a-a6e7-03591d882b42' in your Azure Sphere tenant.

Command completed successfully in 00:00:02.7316538.
```

## update-sku

Sets the product SKU for a device.

### Required parameters

PARAMETER	DESCRIPTION
-s, --skuid <i>GUID</i>	Specifies the ID of the SKU to assign to the device.

### Optional parameters

PARAMETER	DESCRIPTION
-i, --deviceid <i>GUID</i>	Specifies the ID of the device. By default, assigns the SKU to the attached device.
-ip, --deviceip	[Multi-Device] Specifies the IP address of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.
-l, --devicelocation	[Multi-Device] Specifies the FTDI location ID of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.

## ► Global parameters

### Example

```
azsphere device update-sku --skuid 319725fd-1591-4a92-be9a-2f8cf90707f1
Successfully set SKU '319725fd-1591-4a92-be9a-2f8cf90707f1' for device ID <device ID> in your Azure Sphere tenant.

Command completed successfully in 00:00:02.5983519.
```

## wifi

Changes the wireless configuration for the attached device.

To use the device on a wireless network, you must add information about the network and enable the network on the device. Although you can input non-ASCII characters in SSIDs, **azsphere** does not display them properly.

If your application uses the WifiConfig API, you must also include the WifiConfig capability in the application's app\_manifest.json file.

OPERATION	DESCRIPTION
<b>add</b>	Adds the details of a wireless network to the device.
<b>delete</b>	Removes the details of a wireless network from the device.
<b>disable</b>	Disables a wireless network on the device.
<b>enable</b>	Enables a wireless network on the device.
<b>list</b>	Lists the current Wi-Fi configuration for the device.
<b>scan</b>	Scans for available networks.
<b>show-status</b>	Displays the status of the wireless interface.

## wifi add

Adds information about a Wi-Fi connection to the device.

A device can have multiple Wi-Fi connections.

Although you can input non-ASCII characters in SSIDs, **azsphere** does not display them properly.

The **azsphere** command can connect to hidden SSIDs if you include the --targeted-scan parameter.

### Required parameters

PARAMETER	DESCRIPTION
<code>-s, --ssid string</code>	Specifies the SSID of the network. Network SSIDs are case-sensitive.

### Optional parameters

PARAMETER	DESCRIPTION
<code>-p, --psk string</code>	Specifies the WPA/WPA2 key for the new network. Omit to add this SSID as an open network. If your key contains an ampersand (&), enclose the key in quotation marks.
<code>-t, --targeted-scan</code>	Attempts to connect to a network even when it is not broadcasting or is hidden by network congestion.
<code>-ip, --deviceip</code>	[Multi-Device] Specifies the IP address of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.
<code>-l, --devicelocation</code>	[Multi-Device] Specifies the FTDI location ID of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.

## ► Global parameters

### Example

```

azsphere device wifi add --ssid MyNetwork --psk myKey123

Add network succeeded:
ID : 2
SSID : MyNetwork
Configuration state : enabled
Connection state : unknown
Security state : psk

Command completed successfully in 00:00:01.7039497.

```

If the network SSID or key has embedded spaces or an ampersand, enclose the SSID or key in quotation marks. If the SSID or key includes a quotation mark, use a backslash to escape the quotation mark. Backslashes do not require escape if they are part of a value. For example:

```
azsphere device wifi add --ssid "New SSID" --psk "key \"value\" with quotes"
```

## wifi delete

Deletes information about a wireless network from the device.

### Required parameters

PARAMETER	DESCRIPTION
-i, --id <i>integer</i>	Specifies the ID of the network to delete.

### Optional parameters

PARAMETER	DESCRIPTION
-ip, --deviceip	[Multi-Device] Specifies the IP address of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.
-l, --devicelocation	[Multi-Device] Specifies the FTDI location ID of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.

## ► Global parameters

### Example

```

azsphere device wifi delete --id 1
Successfully removed network.
Command completed successfully in 00:00:01.0055424.

```

## wifi disable

Disables a wireless network on the attached device.

### Required parameters

PARAMETER	DESCRIPTION
-i, --id <i>integer</i>	Specifies the ID of the network to disable.

### Optional parameters

PARAMETER	DESCRIPTION
-ip, --deviceip	[Multi-Device] Specifies the IP address of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.
-l, --devicelocation	[Multi-Device] Specifies the FTDI location ID of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.

## ► Global parameters

### Example

```
azsphere device wifi disable --id 2
Successfully disabled network:
ID          : 2
SSID        : NETGEAR21-5G
Configuration state : disabled
Connection state   : disconnected
Security state    : psk
Command completed successfully in 00:00:01.4166658.
```

## wifi enable

Enables a wireless network on the attached device.

To change from one network to another if both are within range, disable the currently connected network before you enable the new network.

### Required parameters

PARAMETER	DESCRIPTION
-i, --id <i>integer</i>	Specifies the ID of the network to enable.

### Optional parameters

PARAMETER	DESCRIPTION
-ip, --deviceip	[Multi-Device] Specifies the IP address of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.
-l, --devicelocation	[Multi-Device] Specifies the FTDI location ID of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.

## ► Global parameters

### Example

```

azsphere device wifi enable --id 2
Successfully enabled network:
ID          : 2
SSID        : NETGEAR21-5G
Configuration state : enabled
Connection state   : connected
Security state    : psk
Command completed successfully in 00:00:01.4063645.

```

## wifi list

Displays information about all the Wi-Fi connections on the device.

### Optional parameters

PARAMETER	DESCRIPTION
-ip, --deviceip	[Multi-Device] Specifies the IP address of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.
-l, --devicelocation	[Multi-Device] Specifies the FTDI location ID of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.

### ► Global parameters

#### Example

```

azsphere device wifi list
Network list:

ID          : 0
SSID        : NETGEAR21
Configuration state : enabled
Connection state   : connected
Security state    : psk
ID          : 1

SSID        : A_WiFi_SSID
Configuration state : enabled
Connection state   : disconnected
Security state    : open
ID          : 2

SSID        : NETGEAR21-5G
Configuration state : enabled
Connection state   : disconnected
Security state    : psk
Command completed successfully in 00:00:00.7698180.

```

## wifi scan

Scans for wireless networks within range of the device. The command will return up to 64 wireless network, but the number of wireless networks displayed may be limited by environmental factors such as the density of access points in range.

### Optional parameters

PARAMETER	DESCRIPTION
-ip, --deviceip	[Multi-Device] Specifies the IP address of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.
-l, --devicelocation	[Multi-Device] Specifies the FTDI location ID of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.

#### Example

```
azsphere device wifi scan

Scan results:
SSID : NETGEAR21
Security state : psk
BSSID : 44:94:fc:36:c8:65
Signal level : -66
Frequency : 2442

SSID : CenturyLink9303
Security state : psk
BSSID : 58:8b:f3:09:ae:d2
Signal level : -75
Frequency : 2412

SSID : NETGEAR21-5G
Security state : psk
BSSID : 44:94:fc:36:c8:64
Signal level : -86
Frequency : 5765

SSID : belkin.c32
Security state : psk
BSSID : 08:86:3b:0b:cc:32
Signal level : -86
Frequency : 2462
Command completed successfully in 00:00:07.4163320.
```

## wifi show-status

Displays information about the current Wi-Fi connection on the device.

#### Optional parameters

PARAMETER	DESCRIPTION
-ip, --deviceip	[Multi-Device] Specifies the IP address of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.
-l, --devicelocation	[Multi-Device] Specifies the FTDI location ID of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.

#### Example

```
azsphere dev wifi show-status
SSID : NETGEAR23
Configuration state : enabled
Connection state : connected
Security state : psk
Frequency : 2417
Mode : station
Key management : WPA2-PSK
WPA State : COMPLETED
IP Address : 192.168.1.9
MAC Address : be:98:26:be:0d:e0

Command completed successfully in 00:00:01.7341825.
```

# device-group, dg

6/20/2019 • 2 minutes to read

Creates and manages device groups.

OPERATION	DESCRIPTION
<b>create</b>	Creates a new device group
<b>feed</b>	Manages the feeds that a device group targets
<b>list</b>	Lists all device groups
<b>show</b>	Displays information about a device group

## create

Creates a new device group and assigns it a friendly name. Device group names must be unique within the Azure Sphere tenant.

By default, application software updates are enabled for all device groups, so that devices receive OTA deployments of application software automatically. To change this default, include the `--noapplicationupdates (-a)` flag when you create a group. Disabling updates means that the devices in the group will not receive OTA updates and must instead be updated by sideloading, either through Visual Studio or by using the [azsphere device sideload](#) command.

### Required parameters

PARAMETER	DESCRIPTION
<code>-n, --name String</code>	Specifies an alphanumeric name for the device group. If the name includes embedded spaces, enclose it in quotation marks. The device group name must be unique within the tenant.

### Optional parameters

PARAMETER	DESCRIPTION
<code>-a, --noapplicationupdates</code>	Disables application updates for this device group.

### ► Global parameters

### Examples

```
azsphere device-group create --name TestLabOTA

Creating device group with name 'TestLabOTA'.
Successfully created device group 'TestLabOTA' with ID '12786f5d-630a-49d5-974c-909bcff5b301',
and update policy: Accept all updates from the Azure Sphere Security Service.
Command completed successfully in 00:00:02.2233048.
```

# feed

Manages the feeds that a device group targets.

COMMAND	DESCRIPTION
<b>add</b>	Adds a feed to the device group
<b>list</b>	Lists all feeds that the device group targets

## feed add

Adds a feed to a device group.

### Required parameters

PARAMETER	DESCRIPTION
-i --devicegroupid <i>GUID</i>	Specifies the GUID that identifies the device group.
-f, --feedid <i>GUID</i>	Specifies the GUID that identifies the feed to add to the device group.

### ► Global parameters

#### Example

```
azsphere device-group feed add --devicegroupid 655d7b12-07ad-4e8a-b104-c0ec494b8489 --feedid ce680169-d893-49de-bb02-5f2c40c52932

Adding feed 'ce680169-d893-49de-bb02-5f2c40c52932' to device group '655d7b12-07ad-4e8a-b104-c0ec494b8489'.
Successfully added feed with ID 'ce680169-d893-49de-bb02-5f2c40c52932' to device group with ID '655d7b12-07ad-4e8a-b104-c0ec494b8489'.
Command completed successfully in 00:00:02.3648319.
```

## feed list

Lists all the feeds that target a specified device group.

### Required parameters

PARAMETER	DESCRIPTION
-i --devicegroupid <i>GUID</i>	Specifies the GUID that identifies the device group.

### ► Global parameters

#### Example

```
azsphere device-group feed list --devicegroupid 63bbe6ea-14be-4d1a-a6e7-03591d882b42

Get all supported feeds that target device group with ID '63bbe6ea-14be-4d1a-a6e7-03591d882b42'.
--> Device group '63bbe6ea-14be-4d1a-a6e7-03591d882b42' contains feed '3369f0e1-dedf-49ec-a602-2aa98669fd61',
    targeting SKU set '0d24af68-c1e6-4d60-ac82-8ba92e09f7e9'.
--> Device group '63bbe6ea-14be-4d1a-a6e7-03591d882b42' contains feed '3369f0e1-dedf-49ec-a602-2aa98669fd61',
    targeting SKU set '9d606c43-1fad-4990-b207-554a025e0869'.
Command completed successfully in 00:00:02.6278794.
```

## list

Lists all device groups in the current tenant.

## ► Global parameters

### Example

```
azsphere device-group list

Listing all device groups.
--> [ID: 19066e8f-c4a0-4b83-8436-73caf0656069] 'TestGroup1'
--> [ID: a56c666c-38fc-4aa5-a9c8-8172cd224c26] 'TestGroup1-no updates'
--> [ID: fbb064a6-df8d-4d21-8a45-d4ff0fb8de95] 'DocMT'
Command completed successfully in 00:00:05.5871572.
```

## show

Returns information about a device group.

### Required parameters

PARAMETER	DESCRIPTION
-i --devicegroupid <i>GUID</i>	Specifies the GUID that identifies the device group.

## ► Global parameters

### Example

```
azsphere device-group show --devicegroupid 63bbe6ea-14be-4d1a-a6e7-03591d882b42
Getting device group with ID '63bbe6ea-14be-4d1a-a6e7-03591d882b42'.
Successfully retrieved the device group:
--> ID:          '63bbe6ea-14be-4d1a-a6e7-03591d882b42'
--> Name:        'System Software'
--> Update Policy: Accept only system software updates from the Azure Sphere Security Service.
Command completed successfully in 00:00:02.6116279.
```

# feed

6/20/2019 • 3 minutes to read

Manages feeds in an Azure Sphere tenant.

OPERATION	DESCRIPTION
<b>create</b>	Creates a new feed
<b>image-set, ims</b>	Manages image sets in a feed
<b>list</b>	Lists all feeds in the Azure Sphere tenant
<b>list-device-groups</b>	Lists all device groups that a feed targets
<b>show</b>	Displays details about a feed

## create

Creates a new feed.

### Required parameters

PARAMETER	DESCRIPTION
<code>-s, --chipskuid <i>GUID</i></code>	Specifies one or more chip SKU IDs that this feed targets. You can either use this flag multiple times to specify multiple SKUs or use the flag once and separate multiple SKU IDs with commas and no intervening spaces.
<code>-c, --componentid <i>GUID</i></code>	Specifies the ID of the component that this feed delivers. The component must already exist. You may add multiple components to the feed by either using this flag multiple times to specify multiple components or once and separate the component IDs with commas and no intervening spaces.
<code>-f, --dependentfeedid <i>GUID</i></code>	Specifies the ID of the Azure Sphere OS feed on which this feed depends. To get a list of system software feeds and IDs, use the <b>azsphere feed list</b> command.
<code>-n, --name <i>String</i></code>	Specifies an alphanumeric name for the feed. Feed names must be unique within a tenant.
<code>-p, --productskuid <i>GUID</i></code>	Specifies one or more product SKU IDs that this feed targets. You can either use this flag multiple times to specify multiple SKUs or use the flag once and separate multiple SKU IDs with commas and no intervening spaces.

### ► Global parameters

#### Example

```

azsphere feed create --name NewDocTestFeed --componentid 4275ecb3-5cf8-4147-9fb-a7e8f3955e96 --chipSkuid
0d24af68-c1e6-4d60-ac82-8ba92e09f7e9 --productskuid ee4c1baa-1887-4da5-aaf9-76c0b59cda70 --dependentfeedid
3369f0e1-dedf-49ec-a602-2aa98669fd61

Creating feed with name 'NewDocTestFeed'.
Successfully created feed 'NewDocTestFeed' with ID 'fa1c6849-dd43-48bb-be91-199b731ea392'.
Command completed successfully in 00:00:08.0771186.

```

## image-set, ims

Manages the image sets in a feed.

COMMAND	DESCRIPTION
<b>add</b>	Adds an image set to a feed
<b>list</b>	Lists all image sets in a feed

### image-set add

Adds an image set to a feed.

#### Required parameters

PARAMETER	DESCRIPTION
-i, --feedid <i>GUID</i>	Specifies the GUID that identifies the feed to which to add the image set.
-s, --imagesetid <i>GUID</i>	Specifies the GUID that identifies the image set to add to the feed.

#### ► Global parameters

##### Example

```

azsphere feed image-set add --feedid 34b62e08-208e-4707-8ffa-f01613f74e2e --imagesetid 16d84454-08d0-4b35-
aa7c-4ebecea3664f

Adding image set with ID '16d84454-08d0-4b35-aa7c-4ebecea3664f' to feed with ID '34b62e08-208e-4707-8ffa-
f01613f74e2e'.
Successfully added image set with ID '16d84454-08d0-4b35-aa7c-4ebecea3664f' to feed with ID '34b62e08-208e-
4707-8ffa-f01613f74e2e'.
Command completed successfully in 00:00:06.1590998.

```

## image-set list

Lists all image sets that are assigned to a particular feed.

#### Required parameters

PARAMETER	DESCRIPTION
-i, --feedid <i>GUID</i>	Specifies the GUID that identifies the feed for which to list the image sets.

#### ► Global parameters

##### Example

```
azsphere feed image-set list --feedid 8d297fc2-4c1b-4b81-9332-94e09f2bf0dd

Listing all image sets in feed '8d297fc2-4c1b-4b81-9332-94e09f2bf0dd'.
Retrieved 1 image sets for feed '8d297fc2-4c1b-4b81-9332-94e09f2bf0dd':
--> {
    "Id": "e88b0fb2-fa0e-4f2c-a68e-8a8c4b9bffd1",
    "FriendlyName": "DocTestBlink"
}
Command completed successfully in 00:00:04.8182373.
```

## list

Lists all feeds in the current Azure Sphere tenant. The feeds in your tenant will be different from those shown in the example.

### ► Global parameters

#### Example

```
azsphere feed list
Listing all feeds.
Retrieved feeds:
--> [3369f0e1-dedf-49ec-a602-2aa98669fd61] 'Retail'
--> [c6f28227-ffff-408e-b5e3-77a4216a5ea3] 'Waffle Maker Feed'
--> [ce680169-d893-49de-bb02-5f2c40c52932] 'GardenCamBlink'
Command completed successfully in 00:00:02.0855618.
```

## list-device-groups

Lists all device groups targeted by a particular feed.

### Required parameters

PARAMETER	DESCRIPTION
-i, --feedid <i>GUID</i>	Specifies the GUID that identifies the feed for which to list device groups.

### ► Global parameters

#### Example

```
azsphere feed list-device-groups --feedid 8d297fc2-4c1b-4b81-9332-94e09f2bf0dd

Listing all device groups targeted by feed '8d297fc2-4c1b-4b81-9332-94e09f2bf0dd'.
Retrieved device groups targeted by feed '8d297fc2-4c1b-4b81-9332-94e09f2bf0dd':
--> Group 'DocMT' (ID: fbb064a6-df8d-4d21-8a45-d4ff0fb8de95) targets SKU set '9d606c43-1fad-4990-b207-554a025e0869, ee4c1baa-1887-4da5-aaf9-76c0b59cda70'
Command completed successfully in 00:00:04.6317958.
```

## show

Displays information about a feed.

### Required parameters

PARAMETER	DESCRIPTION
-i, --feedid <i>GUID</i>	Specifies the GUID that identifies the feed.

## ► Global parameters

### Example

```
azsphere feed show --feedid ce680169-d893-49de-bb02-5f2c40c52932
Getting feed with ID 'ce680169-d893-49de-bb02-5f2c40c52932'.
Retrieved feed 'GardenCamBlink' with ID 'ce680169-d893-49de-bb02-5f2c40c52932'.
- SKU sets supported by this feed:
  -> '9d606c43-1fad-4990-b207-554a025e0869, 946410a0-0057-4b11-af68-d56a684f6681'
- Targeted Component ID: '54acba7c-7719-461a-89db-49c807e0fa4d'.
- Feeds this feed depends on:
  -> 3369f0e1-dedf-49ec-a602-2aa98669fd61
- Image sets in feed:
  -> [6e9cdc9d-c9ca-4080-9f95-b77599b4095a] 'ImageSet-Mt3620Blink1-2018.07.19-18.15.42'.
Command completed successfully in 00:00:02.6813025.
```

# get-support-data

6/20/2019 • 2 minutes to read

Gathers diagnostic and configuration information from your computer, the cloud, and the attached Azure Sphere device to aid technical support.

OPERATION	DESCRIPTION
<b>get-support-data</b>	Collects configuration data from your computer and connected Azure Sphere device.

## Required parameters

PARAMETER	DESCRIPTION
<code>-o, --output <i>filename</i></code>	Name and path for the zip file to create. The file name is required. If you do not provide a path, the file will be stored in the current working folder with the specified name.

## ► Global parameters

### Example

```
azsphere get-support-data --output logs.zip

Gathering device data.
Gathering Azure Sphere Security Service data.
Gathering computer setup data.
Created the support log archive at 'logs.zip'.
Note: This archive contains information about your system including Wi-Fi scans, installation logs, attached USB devices, and Azure Sphere local and cloud configuration.
If you choose to send this data to Microsoft, it will be handled according to the Microsoft Privacy Statement:
go.microsoft.com/fwlink/?linkid=528096
Command completed successfully in 00:01:12.3263605.
```

### NOTE

The collected data might contain information you wish to remain private. Review the lists that follow to determine whether any of the data should remain private before you share the output file with anyone.

The **azsphere get-support-data** command tries to gather all the following information:

### Attached device

- Device ID
- Images installed on device
- Device capabilities installed on the device
- Wi-Fi networks saved to device
- Wi-Fi scan results from device
- Status of the current Wi-Fi setup on the device
- Manufacturing state of the device
- AzureSphere\_DeviceLog.bin log file from the device

- AzureSphere\_DeviceTechSupportData.bin data from the device

### **Azure Sphere Security Service**

- The current configuration, including the current Azure Sphere tenant
- List of Azure Sphere tenants in the current AAD
- OTA configuration, including device group and product SKU
- OTA status, including current device OS version and current OS version installable from cloud

### **Local PC**

- Azure Sphere internal configuration settings
- IP addresses of all local network adapters
- All details of the Azure Sphere network adapter
- The status of the Azure Sphere SLIP service
- The last week's logs of the Azure Sphere SLIP service event log
- The last week's install logs for the Azure Sphere installer
- The last week's install logs for Visual Studio
- The last week's install logs for all Visual Studio Extension installers

# hardware-definition

5/30/2019 • 2 minutes to read

Manages hardware definitions for Azure Sphere devices.

OPERATION	DESCRIPTION
<b>generate-header</b>	Generates a C header file corresponding to a hardware definition and places it in the folder <code>inc/hw</code> relative to the input JSON.
<b>test-header</b>	Test that the C header file in the <code>inc/hw</code> folder is up-to-date with respect to the input JSON.

## generate-header

Creates a C header file for a given hardware definition. By default, the generated file is placed in a `inc/hw` folder relative to the location of the supplied JSON file.

### Required parameters

PARAMETER	DESCRIPTION
<code>-i, --input path</code>	Name and path to a hardware definition JSON file. You may provide a relative or absolute path.

### ► Global parameters

#### Example

```
azsphere hardware-definition generate-header --input mt3620_rdb.json
Generated header file at inc\hw\mt3620_rdb.h based on hardware definition at mt3620_rdb.json
Command completed successfully in 00:00:00.4194122.
```

## test-header

Test that the C header file in the `inc/hw` folder is up-to-date with respect to the input JSON.

### Required parameters

PARAMETER	DESCRIPTION
<code>-i, --input path</code>	Name and path to a hardware definition JSON file. You may provide a relative or absolute path.

### ► Global parameters

#### Example

```
azsphere hardware-definition test-header --input mt3620_rdb.json
Hardware definition at mt3620_rdb.json is consistent with header at inc\hw\mt3620_rdb.h
Command completed successfully in 00:00:00.3894152.
```

# image, img

6/20/2019 • 3 minutes to read

Manages images.

OPERATION	DESCRIPTION
<b>package-application</b>	Creates an image package
<b>package-board-config</b>	Creates a board configuration image package
<b>show</b>	Displays details about an image package

## package-application

Creates an executable application from a compiled and linked image and an `app_manifest.json` file.

Visual Studio automatically renames the binary file for the application to `/bin/app`, so the Visual Studio project name can include punctuation without causing any errors.

Real-time capable applications (RTApps) are built as ELF or AXF files and not as raw binaries. Before packaging an RTApp, edit the application manifest file so that **ApplicationType** is set to "RealTimeCapable", and **EntryPoint** is set to the name of the ELF or AXF file, which must be in the root of the application directory.

### Required parameters

PARAMETER	DESCRIPTION
<code>-i, --input path</code>	Identifies the input directory, which is used as the system root for the Azure Sphere image file. The <code>app_manifest.json</code> file for the application must be in this directory.
<code>-o, --output file</code>	Specifies a filename for the output image package.

### Optional parameters

PARAMETER	DESCRIPTION
<code>-h, --hardwaredefinition path</code>	Optional path to a hardware definition JSON file that is used to map peripheral names in <code>app_manifest.json</code> to underlying values. You may provide a relative or absolute path. See <a href="#">hardware definition</a> for more information.
<code>-s, --sysroot sysroot-name</code>	Name of the sysroot used during compilation. The output binary image package will contain a modified application manifest and metadata that reflect the API set that this sysroot represents. See <a href="#">Use beta API</a> for more information.

PARAMETER	DESCRIPTION
-x, --executables <i>executable1,executable2 ...</i>	<p>Subpaths to one or more files to mark as executable in the image package. The <b>EntryPoint</b> listed in the app_manifest files is always marked as executable, so the -x flag is required only if other executables are present.</p> <p>By default, files are not executable when packaged into an image. The subpaths are relative to the --input <i>path</i>. The paths can use either Windows filename syntax (backslashes) or Linux filename syntax (forward slashes); spaces, commas, and semicolons are not allowed. You can either specify -x for each executable file, or use it only once and supply multiple paths separated by commas without intervening spaces.</p>

## ► Global parameters

### Example

```
azsphere image package-application --input bin --output myimage.imagepackage
```

## package-board-config

Creates a board configuration image package. You may either use a preset board configuration image or provide a custom configuration image.

### Required parameters

PARAMETER	DESCRIPTION
-o, --output <i>filename</i>	Specifies a filename for the output image package.

### Optional parameters

PARAMETER	DESCRIPTION
-c, --componentid <i>GUID</i>	Specifies the component ID for the configuration package. If the ID is not provided, it will be automatically generated.
-i, --input <i>path</i>	Identifies the path to the board configuration image. If this is included, --preset must not be used; the two parameters are mutually exclusive.
-n, --name <i>package-name</i>	Sets the image package name in the created file's metadata. If not provided, a new name will be generated based on the provided board configuration, incorporating part of the component ID for uniqueness.
-p, --preset <i>string</i>	Provides the ID of the preset board configuration image to apply. Either use this flag with the ID of a preset package, or provide --input for a custom board configuration image. The ID is an enumeration value and is currently fixed to the single value "lan-enc28j60-isu0-int5".

## ► Global parameters

### Example

```
azsphere image package-board-config --preset lan-enc28j60-isu0-int5 --output  
c:\output\myBoardConfig.imagepackage
```

## show

Displays information about an image package.

### Required parameters

PARAMETER	DESCRIPTION
-f, --filepath <i>filename</i>	Specifies the path to the image package.

### ► Global parameters

#### Example

```
azsphere image show --filepath "C:\Users\User\Documents\Visual Studio  
2017\Projects\Mt3620Blink1\Mt3620Blink1\bin\ARM\Debug\Mt3620Blink1.imagepackage"  
Image package metadata:  
  Section: Identity  
    Image Type: Application  
    Component ID: 6f68266c-b78b-405e-a47a-72b38b9517ed  
    Image ID: 06ea8ade-0773-4d54-a76a-41a567d46bbd  
  Section: Signature  
    Signing Type: ECDsa256  
    Cert: a8d5cc6958f48710140d7a26160fc1cf31f5df0  
  Section: Debug  
    Image Name: Mt3620Blink50  
    Built On (UTC): 23/10/2018 17:03:33  
    Built On (Local): 23/10/2018 18:03:33  
  Section: Temporary Image  
    Remove image at boot: False  
    Under development: True  
  Section: ABI Depends  
    Depends on: ApplicationRuntime, version 1  
  
Command completed successfully in 00:00:00.3099684..
```

# image-set, ims

6/20/2019 • 2 minutes to read

Manages image sets in an Azure Sphere tenant.

OPERATION	DESCRIPTION
<b>create</b>	Creates a new image set
<b>list</b>	Lists all image sets in the Azure Sphere tenant
<b>show</b>	Displays information about an image set

## create

Creates a new image set. The images must first be uploaded to the tenant using the **azsphere component image add** command.

### Required parameters

PARAMETER	DESCRIPTION
-m, --imageid <i>GUID</i>	Specifies one or more image IDs that identify the images to add to the image set. You can either use this flag multiple times to specify multiple images or use the flag once and separate multiple image IDs with commas and no intervening spaces. Currently, image sets for applications can include only one image.
-n, --name <i>String</i>	Supplies an alphanumeric name for the image set. Image set names must be unique within a tenant.

### ► Global parameters

#### Example

```
azsphere image-set create --name DocTestImageset --imageid dc59be07-1feb-4be9-a5dc-42664dba4871

Adding new image set.
Successfully created image set 'DocTestImageset' with ID '12a6b409-4bec-432b-bfe6-19dac5553ab5'.
Command completed successfully in 00:00:05.7898800.
```

## list

Lists all image sets in the current Azure Sphere tenant.

### ► Global parameters

#### Example

```
azsphere image-set list
Getting all image sets.
Successfully retrieved image sets:
--> [05ef6cb6-ac86-4355-ab46-9c50507b2e46] 'ImageSet-Mt3620Blink11-2018.06.27-17.24.07'
--> [36c1bea7-9dc0-4b01-b0e4-616c079804c4] 'ImageSet-Mt3620Blink15-2018.06.26-15.19.34'
--> [6e9cdc9d-c9ca-4080-9f95-b77599b4095a] 'ImageSet-Mt3620Blink1-2018.07.19-18.15.42'
--> [8e755922-9c11-4f47-ae65-79c87be5dc08] 'Mt3620DirectDHT v0.9'
--> [cf75f0e8-5b36-437f-9729-d37fdb02fab7] 'ImageSet-Mt3620Blink15-2018.06.29-00.02.29'
Command completed successfully in 00:00:04.1471739.
```

## show

Displays details about an image set.

### Required parameters

PARAMETER	DESCRIPTION
-i, --imagesetid <i>GUID</i>	Specifies the GUID that identifies the image set.

### ► Global parameters

#### Example

```
azsphere image-set show --imagesetid 6e9cdc9d-c9ca-4080-9f95-b77599b4095a
Getting image set with ID '6e9cdc9d-c9ca-4080-9f95-b77599b4095a'.
Successfully retrieved image set '6e9cdc9d-c9ca-4080-9f95-b77599b4095a':
--> ID: [6e9cdc9d-c9ca-4080-9f95-b77599b4095a]
--> Name: 'ImageSet-Mt3620Blink1-2018.07.19-18.15.42'
Images to be installed:
--> [ID: 116c0bc5-be17-47f9-88af-8f3410fe7efa]
Command completed successfully in 00:00:02.5532911.
```

# login

6/20/2019 • 2 minutes to read

Provides login to the Azure Sphere tenant. By default, all **azsphere** commands apply to the current user's login identity and tenant. The **login** command lets you use a different identity.

When you use **azsphere**, the Azure Sphere Security Service verifies your identity by using Microsoft Azure Active Directory (AAD). AAD uses Single Sign-On (SSO), which typically defaults to an existing identity on your PC. If this identity is not valid for use with your Azure Sphere tenant, **azsphere** commands may fail.

Use **login** to sign in explicitly to Azure Sphere services. Upon success, this identity is used for subsequent **azsphere** commands. In most cases, you should only have to sign in once.

## ► Global parameters

### Example

```
azsphere login
```

In response, you should see a dialog box that lists your credentials or prompts you to log in. If the list includes the identity that you use for Azure Sphere, choose that identity. If not, enter the appropriate credentials.

# logout

6/20/2019 • 2 minutes to read

Provides logout from the Azure Sphere tenant. By default, all **azsphere** commands apply to the current user's login identity and tenant. Use the **logout** command to log out of your current tenant.

When you use **azsphere**, the Azure Sphere Security Service verifies your identity by using Microsoft Azure Active Directory (AAD). AAD uses Single Sign-On (SSO), which typically defaults to an existing identity on your PC. If this identity is not valid for use with your Azure Sphere tenant, **azsphere** commands may fail.

Use **logout** to sign out of Azure Sphere Security Service. Upon success, you will be signed out and must sign in with **azsphere login** to continue.

## ► Global parameters

### Example

```
azsphere logout
Successfully logged out and cleared tenant selection.
Command completed successfully in 00:00:02.7803960.
```

# show-version

6/20/2019 • 2 minutes to read

Displays version of the current installed Azure Sphere SDK.

## ► Global parameters

### Example

```
azsphere show-version
19.05
Command completed successfully in 00:00:00.6230420.
```

# sku

6/20/2019 • 2 minutes to read

Manages SKUs in an Azure Sphere tenant.

OPERATION	DESCRIPTION
<b>create</b>	Creates a new SKU
<b>list</b>	Lists all SKUs in the Azure Sphere tenant
<b>show</b>	Displays details about a SKU

## create

Creates a new product SKU.

### Required parameters

PARAMETER	DESCRIPTION
<code>-n, --name SkuName</code>	Supplies an alphanumeric name for the SKU. SKU names are case sensitive and must be unique within a tenant.

### Optional parameters

PARAMETER	DESCRIPTION
<code>-d, --description String</code>	Describes the SKU.

### ► Global parameters

#### Example

```
azsphere sku create --name DW100SKU --description "Contoso DW100 models"

Created SKU 'DW100SKU' with ID '78d74a1b-1644-4231-9d3d-0649f4a27f08'.
Command completed successfully in 00:00:05.7252534.
```

## list

Lists all SKUs in the Azure Sphere tenant.

### ► Global parameters

#### Example

```
azsphere sku list
Listing all SKUs.
Retrieved SKUs:

ID                           Name                         SkuType
--                           ----
0d24af68-c1e6-4d60-ac82-8ba92e09f7e9 MT3620 A1 16MB   Chip
9d606c43-1fad-4990-b207-554a025e0869 MT3620 A0 16MB   Chip
78d74a1b-1644-4231-9d3d-0649f4a27f08 DW100SKU        Product
946410a0-0057-4b11-af68-d56a684f6681 GardenCamSKU    Product
adf44435-3c72-41d1-826d-4018359319b8 Waffle Maker SKU  Product
```

Command completed successfully in 00:00:03.1203830.

## show

Displays details about a SKU.

### Required parameters

PARAMETER	DESCRIPTION
-i, --skuid <i>GUID</i>	Specifies the ID of the SKU.

### ► Global parameters

#### Example

```
azsphere sku show --skuid 78d74a1b-1644-4231-9d3d-0649f4a27f08
Getting details for SKU with ID '78d74a1b-1644-4231-9d3d-0649f4a27f08'.
Retrieved SKU:
--> ID:          78d74a1b-1644-4231-9d3d-0649f4a27f08
--> Name:        'DW100SKU'
--> Description: 'Contoso DW100 models'
--> Type:         'Product'
Command completed successfully in 00:00:01.9224778.
```

# tenant

10/9/2019 • 3 minutes to read

Manages an Azure Sphere tenant.

OPERATION	DESCRIPTION
<b>create</b>	Creates a new tenant
<b>download-ca-certificate</b>	Downloads the CA certificate for the current tenant
<b>download-validation-certificate</b>	Downloads the validation certificate for the current tenant, based on the provided verification code
<b>list</b>	Lists the available Azure Sphere tenants
<b>select</b>	Selects the default Azure Sphere tenant to use on this PC
<b>show-selected</b>	Shows the default Azure Sphere tenant for this PC

## create

Creates a new Azure Sphere tenant.

By default, **azsphere** allows one tenant per Azure Active Directory (AAD). If you already have a tenant and are certain you want another one, use the --force parameter. Currently, you cannot delete an Azure Sphere tenant.

### Required parameters

PARAMETER	DESCRIPTION
-n, --name <i>string</i>	Specifies a name for the tenant.

### Optional parameters

PARAMETER	DESCRIPTION
--force	Forces creation of a new Azure Sphere tenant in the current user's Azure Active Directory (AAD).
-ip, --deviceip	[Multi-Device] Specifies the IP address of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.
-l, --devicelocation	[Multi-Device] Specifies the FTDI location ID of the device to use for this operation. This is only required when multiple devices are attached. You may specify either a device IP or device location.

### ► Global parameters

#### Example

```
azsphere tenant create --name "DocExample"
warn: The following Azure Sphere tenants already exist in this AAD. To add a new Azure Sphere tenant, call
this
command with the '--force' flag.
--> 'Microsoft' (d343c263-4aa3-4558-adbb-d3fc34631800)
error: Command failed in 00:00:01.7883424.
```

## download-ca-certificate

Downloads the certificate authority (CA) certificate for the current Azure Sphere tenant.

The CA certificate is required as part of the device authentication and attestation process.

### Required parameters

PARAMETER	DESCRIPTION
-o, --output <i>filepath</i>	Specifies the path and filename in which to store the CA certificate. The <i>filepath</i> can be an absolute or relative path but must have the .cer extension.

### ► Global parameters

#### Example

```
azsphere tenant download-ca-certificate --output CA-cert.cer
Saving the CA certificate to 'C:\Users\Test\Documents\AzureSphere\CA-cert.cer'.
Saved the CA certificate to 'CA-cert.cer'.
Command completed successfully in 00:00:03.0547491.
```

## download-validation-certificate

Downloads the validation certificate based on the provided verification code for the current Azure Sphere tenant.

The validation certificate is part of the device authentication and attestation process.

### Required parameters

PARAMETER	DESCRIPTION
-c, --verificationcode <i>string</i>	Provides the verification code required to get a validation certificate.
-o, --output <i>filepath</i>	Specifies the path and filename in which to store the validation certificate. The <i>filepath</i> can be an absolute or relative path but must have the .cer extension.

### ► Global parameters

#### Example

```
azsphere tenant download-validation-certificate --output validation.cer --verificationcode 123412341234
Saving the validation certificate to 'C:\Users\Test\Documents\AzureSphere\validation.cer'.
Saved the validation certificate to 'validation.cer'.
Command completed successfully in 00:00:01.7821834.
```

## list

Lists the Azure Sphere tenants in the current AAD.

## ► Global parameters

### Example

```
azsphere tenant list
ID           Name
--           ---
d343c263-4aa3-4558-adbb-d3fc34631800 Microsoft

Command completed successfully in 00:00:02.0344647.
```

## select

Selects the default Azure Sphere tenant to use on this PC. To display the current default tenant, use **azsphere tenant show-selected**.

### Required parameters

PARAMETER	DESCRIPTION
-i, --tenantid <i>GUID</i>	Specifies the ID of the Azure Sphere tenant to use.

## ► Global parameters

### Example

```
azsphere tenant select --tenantid d343c263-4aa3-4558-adbb-d3fc34631800
Default Azure Sphere tenant ID has been set to 'd343c263-4aa3-4558-adbb-d3fc34631800'.
Command completed successfully in 00:00:00.3808250.
```

## show-selected

Displays the ID of the default Azure Sphere tenant for the PC. This is the tenant selected with the **azsphere tenant select** command.

## ► Global parameters

### Example

```
azsphere tenant show-selected
Default Azure Sphere tenant ID is 'd343c263-4aa3-4558-adbb-d3fc34631800'.
Command completed successfully in 00:00:00.3425522.
```

# Azure Sphere Application Libraries

10/9/2019 • 2 minutes to read

The Azure Sphere SDK Application Libraries (Applibs) support device-specific APIs for Azure Sphere application development. These headers contain the Applibs functions and types:

HEADER	DESCRIPTION
<a href="#">applibs/adc.h</a>	Interacts with analog-to-digital converters (ADCs).
<a href="#">applibs/application.h</a>	Communicates with and controls real-time capable applications.
<a href="#">EventLoop</a>	Monitors and dispatches events.
<a href="#">applibs/gpio.h</a>	Interacts with GPIOs (general-purpose input/output).
<a href="#">applibs/i2c.h</a>	Interacts with Inter-Integrated Circuit (I2C) interfaces.
<a href="#">applibs/log.h</a>	Logs debug messages that are displayed when you debug an application through the Azure Sphere SDK.
<a href="#">applibs/networking.h</a>	Interacts with the networking subsystem to query the network state, and to get and set the network service configuration.
<a href="#">applibs/pwm.h</a>	Interacts with pulse-width modulators (PWMs).
<a href="#">applibs/rtc.h</a>	Interacts with the real-time clock (RTC).
<a href="#">applibs/spi.h</a>	Interacts with SPI (Serial Peripheral Interface) devices.
<a href="#">applibs/storage.h</a>	Interacts with on-device storage, which includes read-only storage and mutable storage.
<a href="#">SysEvent</a>	Interacts with system event notifications.
<a href="#">applibs/uart.h</a>	Opens and interacts with a Universal Asynchronous Receiver/Transmitter (UART) on a device.
<a href="#">applibs/wificonfig.h</a>	Manages Wi-Fi network configurations on a device.

# Base APIs

10/9/2019 • 3 minutes to read

The [Azure Sphere Application Runtime](#) includes a set of common libraries that define the base APIs that are available for high-level application development: a POSIX-based C standard library, a curl-based HTTP client library, and an Azure IoT C SDK library.

This topic describes how to determine which base APIs are included in Azure Sphere, and where to find reference documentation for the base APIs. For information about device-specific APIs, see [Applibs APIs](#).

## Unsupported functions

It is important to use only base API functions that are explicitly included in the API surface of the Azure Sphere Application Runtime. Applications that call unsupported functions may not be compatible with future releases of the Azure Sphere OS, can cause device instability, and will void our support guarantee. If you want to request support for additional functions, you can use the [Azure Sphere forum](#) to make the request.

## Verify functions

To determine whether a function call is supported, use auto-complete with IntelliSense in Visual Studio or verify that it is included in the Azure Sphere SDK header files. The locations of the header files for each library are listed in the sections below. If we add or modify functions in these libraries, we will list them in this topic.

## POSIX-based C standard library

The Azure Sphere Application Runtime provides a custom version of the POSIX C standard library that excludes significant portions of some library features. Functional differences between the custom library and the POSIX standard are listed in [this topic](#). The entire supported API surface of the library is defined in the Azure Sphere SDK header files. We don't guarantee the use of the current implementation indefinitely, so it may change in a future release.

[functional differences](#)

**API reference:** [POSIX specification](#)

**Header file location:** Sysroots\API set\usr\include folders of the Azure Sphere SDK installation directory.

### C standard library features

Significant portions of the following C standard library features are excluded:

- file system paths
- terminal support
- ioctl and fcntl functions
- authentication and authorization
- syscall functions
- System V (SysV)

## curl library

The Azure Sphere SDK includes a subset of the libcurl multi protocol transfer library. You can use this API to transfer data over HTTP/HTTPs. The other transfer protocols are not supported. The entire supported API surface

of the library is defined in the Azure Sphere SDK header files.

**API reference:** [libcurl website](#)

**Header file location:** Sysroots\API\set\usr\include\curl folder of the Azure Sphere SDK installation directory.

## Azure IoT C SDK library

The Azure Sphere SDK includes a subset of the Azure IoT C SDK library. You can use this API to connect your application to an Azure IoT Hub. The entire supported API surface of the library is defined in the Azure Sphere SDK header files.

**API reference:** [Azure IoT device SDK for C](#)

**Header file location:** Sysroots\API\set\usr\include\azureiot folder of the Azure Sphere SDK installation directory.

### Azure IoT C SDK library features

Azure Sphere is configured with these settings for [constrained devices](#):

- Access to the lower layer (LL) programming model only; no access to the convenience layer.
- Use of the MQTT protocol; no support for the AQMP or HTTP protocols.
- No support for blob storage.

### Azure IoT C SDK library changes

This section lists the added and modified Azure IoT C SDK functions.

NAME	CHANGE	HEADER	COMMENTS
IoTHubDeviceClient_LL_CreateWithAzureSphereDeviceAuthProvisioning	added	azure_sphere_provisioning.h	

## TLS utilities library

[!INCLUDE [beta-inline]

The TLS utilities library supports [mutual authentication](#) over TLS (Transport Layer Security).

**API reference:** The header files contain the documentation for this library.

**Header file location:** Sysroots\API\set\usr\include\tlsutils folder of the Azure Sphere SDK installation directory.

### TLS utilities library features

The TLS utilities library currently provides support for using [curl SSL functions](#) to perform mutual authentication for an Azure Sphere device. The **HTTPS\_Curl\_MutualAuth** demonstrates how to this API.

# Applibs application.h

5/30/2019 • 2 minutes to read

**Header:** #include <applibs/application.h>

The Applibs application header contains functions that communicate with and control real-time capable applications.

## Functions

FUNCTION	DESCRIPTION
<a href="#">Application_Socket</a>	Creates a socket that can communicate with a real-time capable application.

# Application\_Socket Function

5/30/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/application.h>

Creates a socket that can communicate with a real-time capable application. The socket is created in a connected state, and may be used with the send() and recv() functions to transfer messages to and from the real-time capable application. The message format is similar to a datagram.

```
int Application_Socket(const char *componentId);
```

## Parameters

- `componentId` The component ID of the real-time capable application.

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- EINVAL: An invalid `componentId` was specified.
- EACCES: The component ID of the real-time capable application was not listed in the AllowedApplicationConnections capability in the application manifest of `componentId`.

Any other errno may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

The file descriptor of the socket, or -1 for failure, in which case errno is set to the error value.

## Application manifest requirements

The [application manifest](#) of the current application must list the component ID of the real-time capable application in the AllowedApplicationConnections capability. In addition, the application manifest of the real-time capable application must list the component ID of the current application in the AllowedApplicationConnections capability.

# Applibs adc.h

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/adc.h>

The Applibs adc header contains functions and types that interact with analog-to-digital converters (ADC).

## Application manifest requirements

To access an ADC, your application must identify it in the `Adc` field of the [application manifest](#).

## Thread safety

ADC functions are thread-safe between calls to different ADC channels; however, it is the caller's responsibility to ensure thread safety when accessing the same ADC channel.

## Concepts and samples

- [Using ADC]
- [Sample: ADC]

## Functions

FUNCTION	DESCRIPTION
<a href="#">ADC_GetSampleBitCount</a>	Gets the bit depth of the ADC.
<a href="#">ADC_Open</a>	Opens an ADC controller, and returns a file descriptor to use for subsequent calls.
<a href="#">ADC_Poll</a>	Gets sample data for an ADC channel.
<a href="#">ADC_SetReferenceVoltage</a>	Sets the reference voltage for an ADC.

## Typedefs

TYPEDEF	DESCRIPTION
<a href="#">ADC_ChannelId</a>	The ID of an ADC channel.
<a href="#">ADC_ControllerId</a>	The ID of an ADC controller.

# ADC\_GetSampleBitCount Function

10/9/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/adc.h>

Gets the bit depth of the ADC.

```
int ADC_GetSampleBitCount(int fd, ADC_ChannelId channel);
```

## Parameters

- `fd` The file descriptor for the ADC controller. The file descriptor is retrieved by [ADC\\_Open](#).
- `channel` The [ID](#) of the ADC channel to access. The range of allowed values is controller-dependent and is typically a zero-based index.

## Returns

The bit depth of the ADC. If an error is encountered, returns -1 and sets errno to the error value.

## Remarks

An example return value is 12, which indicates that the ADC controller can supply 12 bits of data that range from 0 to 4095.

## Application manifest requirements

To access an ADC controller, your application must identify it in the `Adc` field of the [application manifest](#).

# ADC\_Open Function

10/9/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/adc.h>

Opens an ADC controller, and returns a file descriptor to use for subsequent calls.

```
int ADC_Open(ADC_ControllerId id);
```

## Parameters

- `id` The ID of the ADC controller to open. The ID is a zero-based index. The maximum value permitted depends on the platform.

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- EACCES: access to the ADC controller isn't permitted. Verify that the controller exists and is listed in the "Adc" field of the application manifest.

Any other errno may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

The file descriptor for the ADC controller if it was opened successfully, or -1 for failure, in which case errno is set to the error value.

## Application manifest requirements

To access an ADC controller, your application must identify it in the Adc field of the [application manifest](#).

# ADC\_Poll Function

10/9/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/adc.h>

Gets sample data for an ADC channel.

```
int ADC_Poll(int fd, ADC_ChannelId channel, uint32_t *outSampleValue);
```

## Parameters

- `fd` The file descriptor for the ADC controller. The file descriptor is retrieved by [ADC\\_Open](#).
- `channel` The ID of the channel to access. The range of allowed values is controller-dependent and is typically a zero-based index.
- `outSampleValue` A pointer to the `uint32_t` that receives the sample data. This parameter must not be set to `NULL`.

## Errors

If an error is encountered, returns -1 and sets `errno` to the error value. Such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

0 for success, or -1 for failure, in which case `errno` is set to the error value.

## Application manifest requirements

To access an ADC controller, your application must identify it in the `Adc` field of the [application manifest](#).

# ADC\_SetReferenceVoltage Function

10/9/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/adc.h>

Sets the reference voltage for the ADC.

```
int ADC_SetReferenceVoltage(int fd, ADC_ChannelId channel, float referenceVoltage);
```

## Parameters

- `fd` The file descriptor for the ADC controller. The file descriptor is retrieved by [ADC\\_Open](#).
- `channel` The [ID](#) of the channel to access. The range of allowed values is controller-dependent and is typically a zero-based index.
- `referenceVoltage` The reference voltage to use.

## Errors

If an error is encountered, returns -1 and sets errno to the error value. Such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

0 for success, or -1 for failure, in which case errno is set to the error value.

## Application manifest requirements

To access an ADC controller, your application must identify it in the Adc field of the [application manifest](#).

# ADC\_ChannelId Typedef

10/9/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/adc.h>

The ID of an ADC channel.

```
typedef uint32_t ADC_ChannelId;
```

## Remarks

ADCs often have multiple channels on a single chip. A channel corresponds to a single pin or input on the device. The range of allowed values for a channel ID is device-dependent, and is typically a zero-based index.

# ADC\_ControllerId Typedef

10/9/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/adc.h>

The ID of an ADC controller. This ID is a zero-based index.

```
typedef uint32_t ADC_ControllerId;
```

# Applibs eventloop.h

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/eventloop.h>

The Applibs eventloop header contains functions and types used to monitor and dispatch events.

## Functions

FUNCTION	DESCRIPTION
<a href="#">EventLoop_Close</a>	Closes an <i>EventLoop</i> object and releases its memory.
<a href="#">EventLoop_Create</a>	Creates an <i>EventLoop</i> object.
<a href="#">EventLoop_GetWaitDescriptor</a>	Gets a file descriptor for an <i>EventLoop</i> .
<a href="#">EventLoop_RegisterIo</a>	Registers an I/O event with an <i>EventLoop</i> .
<a href="#">EventLoop_Run</a>	Runs an <i>EventLoop</i> and dispatches pending events in the caller's thread of execution.
<a href="#">EventLoop_Stop</a>	Stops the <i>EventLoop</i> from running and causes <a href="#">EventLoop_Run</a> to return control to its caller.
<a href="#">EventLoop_UnregisterIo</a>	Unregisters an I/O event from an <i>EventLoop</i> object.
<a href="#">EventLoopIoCallback</a>	The callback invoked by an <i>EventLoop</i> object when a registered I/O event occurs.

## Structs

STRUCT	DESCRIPTION
<a href="#">EventLoop</a>	An object that monitors event sources and dispatches their events to handlers.
<a href="#">EventRegistration</a>	A handle returned when a callback is registered with an event source.

## Enums

ENUM	DESCRIPTION
<a href="#">EventLoop_IoEvents</a>	A bitmask of the I/O events that can be captured by the <i>EventLoop</i> object.
<a href="#">EventLoop_Run_Result</a>	The possible return values for the <a href="#">EventLoop_Run</a> function.

# EventLoop\_Close Function

10/9/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/eventloop.h>

Closes an [EventLoop](#) object and releases its memory.

```
void EventLoop_Close(EventLoop *el);
```

## Parameters

- `el` The *EventLoop* object to close.

## Remarks

The pointer to the *EventLoop* object becomes invalid when *EventLoop\_Close* returns. Any attempts to use the pointer will lead to undefined behavior.

# EventLoop\_Create Function

10/9/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/eventloop.h>

Creates an [EventLoop](#) object.

```
EventLoop *EventLoop_Create(void);
```

## Returns

A pointer to the new *EventLoop* object. To avoid memory leaks, the returned object must be closed with [EventLoop\\_Close](#) when it's no longer needed.

# EventLoop\_GetWaitDescriptor Function

10/9/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/eventloop.h>

Gets a file descriptor for an [EventLoop](#).

The file descriptor is signaled for input when the [EventLoop](#) has work ready to process. The application can wait or poll the file descriptor to determine when to process the *EventLoop* with [EventLoop\\_Run](#).

```
int EventLoop_GetWaitDescriptor(EventLoop *el);
```

## Parameters

- `el` The EventLoop.

## Returns

The waitable descriptor on success, or -1 on failure, in which case errno is set to the error value.

# EventLoop\_RegisterIo Function

10/9/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/eventloop.h>

Registers an I/O event with an [EventLoop](#).

```
EventRegistration *EventLoop_RegisterIo(EventLoop *el, int fd, EventLoop_IoEvents events,
   EventLoopIoCallback *callback, void *context);
```

## Parameters

- `el` The EventLoop on which to register the I/O event.
- `fd` The file descriptor for the I/O event.
- `events` The bitmask of events to monitor.
- `callback` A pointer to the callback function to call whenever a monitored event is raised.
- `context` An optional context pointer.

## Returns

A pointer to an [EventRegistration](#) object on success, otherwise NULL for failure, in which case errno is set to the error value.

## Remarks

If this function succeeds, it returns a pointer to an *EventRegistration* object that tracks the registration operation. The *EventRegistration* object remains active until the application calls [EventLoop\\_UnregisterIo](#) on the object or closes the object with [EventLoop\\_Close](#).

# EventLoop\_Run Function

10/9/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/eventloop.h>

Runs an [EventLoop](#) and dispatches pending events in the caller's thread of execution.

```
EventLoop_Run_Result EventLoop_Run(EventLoop *el, int duration_in_milliseconds,  
                                    bool process_one_event);
```

## Parameters

- `el` The EventLoop to run.
- `duration_in_milliseconds` The length of time to run the event loop. If zero, the loop will process one event if one is ready and break immediately, regardless of the value of the `process_one_event` parameter. If greater than zero, the loop will run for the specified duration unless it is interrupted. If less than zero, the loop will keep running until interrupted. See the [EventLoop\\_Stop](#) function and the `process_one_event` parameter for additional conditions.
- `process_one_event` Indicates whether to break the loop after the first event is processed. If false, the loop will keep running for the duration specified by the `duration_in_milliseconds` parameter, or until it is interrupted by [EventLoop\\_Stop](#). This parameter is ignored if `duration_in_milliseconds` is zero.

## Returns

An [EventLoop\\_Run\\_Result](#) value that indicates results of this function call.

## Remarks

An application can call `EventLoop_Run(el, -1, false)` to pass control of the calling thread to the *EventLoop*.

If the application calls `EventLoop_Run(el, -1, true)`, the loop will block and wait until the first event is ready, then it will process the event and return.

An application can call [EventLoop\\_Stop](#) to exit [EventLoop\\_Run](#) earlier.

# EventLoop\_Stop Function

10/9/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/eventloop.h>

Stops the [EventLoop](#) from running and causes [EventLoop\\_Run](#) to return control to its caller.

```
int EventLoop_Stop(EventLoop *el);
```

## Parameters

- `el` A pointer to the *EventLoop* object.

## Returns

0 for success, or -1 for failure, in which case `errno` is set to the error value.

## Remarks

This function can be called from an event callback or another thread to stop the current loop and return from [EventLoop\\_Run](#).

If called from a callback, [EventLoop\\_Run](#) will stop synchronously. Once [EventLoop\\_Stop](#) returns, no further events will be processed by [EventLoop\\_Run](#). [EventLoop\\_Run](#) will then stop processing events and return to its caller.

If called from another thread, [EventLoop\\_Run](#) will stop asynchronously and return to its caller. As a result, [EventLoop\\_Run](#) may still process a few events after [EventLoop\\_Stop](#) returns.

An *EventLoop* object is a single-threaded object. An attempt to use [EventLoop](#) from multiple threads simultaneously will result in undefined behavior. The only exception is a call to [EventLoop\\_Stop](#).

# EventLoop\_UnregisterIo Function

10/9/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/eventloop.h>

Unregisters an I/O event from an [EventLoop](#) object.

```
int EventLoop_UnregisterIo(EventLoop *el, EventRegistration *reg);
```

## Parameters

- `el` The *EventLoop* to which the I/O event is registered.
- `reg` The [EventRegistration](#) object returned by [EventLoop\\_RegisterIo](#).

## Returns

0 for success, or -1 for failure, in which case `errno` is set to the error value.

## Remarks

The [EventRegistration](#) object must be returned by the [EventLoop\\_RegisterIo](#) call for the same *EventLoop* object. Trying to unregister an *EventRegistration* object from a different *EventLoop* or another object will lead to undefined behavior.

An active *EventRegistration* object can be unregistered once. Trying to unregister it multiple times will lead to undefined behavior.

# EventLoopIoCallback Function

10/9/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/eventloop.h>

The callback invoked by an [EventLoop](#) object when a registered I/O event occurs.

```
typedef void EventLoopIoCallback(EventLoop *el, int fd, EventLoop_IoEvents events, void *context);
```

## Parameters

- `el` The [EventLoop](#) to which the callback is registered.
- `fd` The file descriptor for the new I/O event.
- `events` The bitmask of events raised for the *EventLoop* object.
- `context` The optional context pointer that was passed to [EventLoop\\_RegisterIo](#).

# EventLoop\_IoEvents Enum

10/9/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/eventloop.h>

A bitmask of the I/O events that can be captured by the EventLoop object.

```
typedef uint32_t EventLoop_IoEvents;
enum {
    EventLoop_Input = 0x01,
    EventLoop_Output = 0x04,
    EventLoop_Error = 0x08,
};
```

## Members

### **EventLoop\_Input**

The descriptor is available for read operations.

### **EventLoop\_Output**

The descriptor is available for write operations.

### **EventLoop\_Error**

The descriptor experienced an error. The EventLoop always reports this event independently of the bitmask passed to [EventLoop\\_RegisterIo](#).

# EventLoop\_Run\_Result Enum

10/9/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/eventloop.h>

The possible return values for the [EventLoop\\_Run](#) function.

```
typedef int EventLoop_Run_Result;
enum {
    EventLoop_Run_Failed = -1,
    EventLoop_Run_FinishedEmpty = 0,
    EventLoop_Run_Finished = 1,
};
```

## Members

### **EventLoop\_Run\_Failed**

*EventLoop\_Run* failed; errno has the specific error code.

### **EventLoop\_Run\_FinishedEmpty**

*EventLoop\_Run* finished without processing any events.

### **EventLoop\_Run\_Finished**

*EventLoop\_Run* finished after processing one or more events.

# EventLoop Struct

10/9/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/eventloop.h>

An object that monitors event sources and dispatches their events to handlers.

```
typedef struct EventLoop EventLoop;
```

## Remarks

An EventLoop object is single-threaded. An application can use one or more EventLoop objects per thread, but each object must only be used in one thread. [EventLoop\\_Stop](#) can be called for an object in another thread; however, the other EventLoop functions must be called from the same thread as the target object in order to prevent undefined behavior.

To dispatch the events that need processing, the application must call [EventLoop\\_Run](#). The event handlers are called in the same thread where *EventLoop\_Run* is called.

# EventRegistration Struct

10/9/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/eventloop.h>

A handle returned when a callback is registered with an event source. This handle is later used to unregister the callback with the same source.

```
typedef struct EventRegistration EventRegistration;
```

# Applibs gpio.h

4/4/2019 • 2 minutes to read

**Header:** #include <applibs/gpio.h>

The Applibs gpio header contains functions and types that interact with GPIOs.

## Application manifest requirements

To access individual GPIOs, your application must identify them in the Gpio field of the [application manifest](#).

## Thread safety

GPIO functions are thread-safe between calls to different GPIOs; however, it is the caller's responsibility to ensure thread safety for accesses to the same GPIO.

## Concepts and samples

- [Quickstart: Build the Blink sample application](#)
- [Sample: GPIO](#)
- [Sample: CurlMultiHttps](#)
- [Sample: System Time](#)
- [Sample: External MCU update - reference solution](#)

## Functions

FUNCTION	DESCRIPTION
<a href="#">GPIO_GetValue</a>	Gets the current value of a GPIO.
<a href="#">GPIO_OpenAsInput</a>	Opens a GPIO (General Purpose Input/Output) as an input.
<a href="#">GPIO_OpenAsOutput</a>	Opens a GPIO (General Purpose Input/Output) as an output.
<a href="#">GPIO_SetValue</a>	Sets the output value for an output GPIO.

## Enums

ENUM	DESCRIPTION
<a href="#">GPIO_OutputMode</a>	The options for the output mode of a GPIO.
<a href="#">GPIO_Value</a>	The possible read/write values for a GPIO.

## Typedefs

TYPEDEF	DESCRIPTION
<a href="#">GPIO_Id</a>	Specifies the type of a GPIO ID, which is used to specify a GPIO peripheral instance.
<a href="#">GPIO_OutputMode_Type</a>	Specifies the type of the <a href="#">GPIO output mode</a> .
<a href="#">GPIO_Value_Type</a>	Specifies the type of a GPIO value.

# GPIO\_GetValue Function

1/7/2019 • 2 minutes to read

**Header:** #include <applibs/gpio.h>

Gets the current value of a GPIO.

GPIO functions are thread-safe between calls to different GPIOs; however, it is the caller's responsibility to ensure thread safety for accesses to the same GPIO.

```
int GPIO_GetValue(int gpioFd, GPIO_Value_Type *outValue);
```

## Parameters

- `gpioFd` The file descriptor for the GPIO.
- `outValue` The [GPIO\\_Value](#) read from the GPIO - `GPIO_Value_High` or `GPIO_Value_Low`.

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- `EFAULT`: the `outValue` is NULL.
- `EBADF`: the `gpioFd` is not valid.

Any other errno may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

0 for success, or -1 for failure, in which case errno will be set to the error value.

## Application manifest requirements

To access individual GPIOs, your application must identify them in the Gpio field of the [application manifest](#).

# GPIO\_OpenAsInput Function

1/7/2019 • 2 minutes to read

**Header:** #include <applibs/gpio.h>

Opens a GPIO (General Purpose Input/Output) as an input.

- Call [GPIO\\_GetValue](#) on an open input GPIO to read the input value.
- A [GPIO\\_SetValue](#) call on an open input GPIO will have no effect.

GPIO functions are thread-safe between calls to different GPIOs; however, it is the caller's responsibility to ensure thread safety for accesses to the same GPIO.

```
int GPIO_OpenAsInput(GPIO_Id gpioId);
```

## Parameters

- `gpioId` A [GPIO\\_Id](#) that identifies the GPIO.

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- EACCES: access to *gpioId* is not permitted as the GPIO is not listed in the Gpio field of the application manifest.
- ENODEV: the provided *gpioId* is invalid.
- EBUSY: the *gpioId* is already open.

Any other errno may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

A file descriptor for the opened GPIO on success, or -1 for failure, in which case errno is set to the error value.

## Application manifest requirements

To access individual GPIOs, your application must identify them in the Gpio field of the [application manifest](#).

# GPIO\_OpenAsOutput Function

1/7/2019 • 2 minutes to read

**Header:** #include <applibs/gpio.h>

Opens a GPIO (General Purpose Input/Output) as an output.

An output GPIO may be configured as [push-pull](#), [open drain](#), or [open source](#). Call [GPIO\\_SetValue](#) on an open output GPIO to set the output value. You can also call [GPIO\\_GetValue](#) on an open output GPIO to read the current value (for example, when the output GPIO is configured as `GPIO_OutputMode_OpenDrain` or `GPIO_OutputMode_OpenSource`).

GPIO functions are thread-safe between calls to different GPIOs; however, it is the caller's responsibility to ensure thread safety for accesses to the same GPIO.

```
int GPIO_OpenAsOutput(GPIO_Id gpioId, GPIO_OutputMode_Type outputMode, GPIO_Value_Type initialValue);
```

## Parameters

- `gpioId` A [GPIO\\_Id](#) that identifies the GPIO.
- `outputMode` The [output mode](#) of the GPIO. An output may be configured as push-pull, open drain, or open source.
- `initialValue` The initial [GPIO\\_Value](#) for the GPIO - `GPIO_Value_High` or `GPIO_Value_Low`.

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- `EACCES`: access to `gpioId` is not permitted as the GPIO is not listed in the `Gpio` field of the application manifest.
- `EBUSY`: the `gpioId` is already open.
- `ENODEV`: the `gpioId` is invalid.
- `EINVAL`: the `outputMode` is not a valid [GPIO\\_OutputMode](#) or the `initialValue` is not a valid [GPIO\\_Value](#).

Any other errno may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

A file descriptor for the opened GPIO on success, or -1 for failure, in which case errno will be set to the error value.

## Application manifest requirements

To access individual GPIOs, your application must identify them in the `Gpio` field of the [application manifest](#).

# GPIO\_SetValue Function

1/7/2019 • 2 minutes to read

**Header:** #include <applibs/gpio.h>

Sets the output value for an output GPIO. Only has an effect on GPIOs opened as outputs.

GPIO functions are thread-safe between calls to different GPIOs; however, it is the caller's responsibility to ensure thread safety for accesses to the same GPIO.

```
int GPIO_SetValue(int gpioFd, GPIO_Value_Type value);
```

## Parameters

- `gpioFd` The file descriptor for the GPIO.
- `value` The [GPIO\\_Value](#) value to set - `GPIO_Value_High` or `GPIO_Value_Low`.

## Errors

If an error is encountered, returns -1 and sets `errno` to the error value.

- `EINVAL`: the `value` is not a [GPIO\\_Value](#).
- `EBADF`: the `gpioFd` is not valid.

Any other `errno` may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

0 for success, or -1 for failure, in which case `errno` will be set to the error value.

## Application manifest requirements

To access individual GPIOs, your application must identify them in the `Gpio` field of the [application manifest](#).

# GPIO\_OutputMode Enum

1/7/2019 • 2 minutes to read

**Header:** #include <applibs/gpio.h>

The options for the output mode of a GPIO.

```
typedef enum {
    GPIO_OutputMode_PushPull = 0,
    GPIO_OutputMode_OpenDrain = 1,
    GPIO_OutputMode_OpenSource = 2
} GPIO_OutputMode;
```

VALUES	DESCRIPTIONS
GPIO_OutputMode_PushPull	Sets the output mode to push-pull.
GPIO_OutputMode_OpenDrain	Sets the output mode to open drain.
GPIO_OutputMode_OpenSource	Sets the output mode to open source.

# GPIO\_Value Enum

1/7/2019 • 2 minutes to read

**Header:** #include <applibs/gpio.h>

The possible read/write values for a GPIO.

```
typedef enum {
    GPIO_Value_Low = 0,
    GPIO_Value_High = 1
} GPIO_Value;
```

VALUES	DESCRIPTIONS
GPIO_Value_Low	Low, or logic 0
GPIO_Value_High	High, or logic 1

# GPIO\_Id Typedef

1/7/2019 • 2 minutes to read

**Header:** #include <applibs/gpio.h>

Specifies the type of a GPIO ID, which is used to specify a GPIO peripheral instance.

```
typedef int GPIO_Id;
```

# GPIO\_OutputMode\_Type Typedef

1/7/2019 • 2 minutes to read

**Header:** #include <applibs/gpio.h>

Specifies the type of the [GPIO output mode](#).

```
typedef int GPIO_OutputMode_Type;
```

# GPIO\_Value\_Type Typedef

1/7/2019 • 2 minutes to read

**Header:** #include <applibs/gpio.h>

Specifies the type of a [GPIO value](#).

```
typedef int GPIO_Value_Type;
```

# Applibs i2c.h

2/14/2019 • 2 minutes to read

**Header:** #include <applibs/i2c.h>

The Applibs I2C header contains functions and types that interact with an I2C (Inter-Integrated Circuit) interface.

## Application manifest requirements

To access an I2C master interface, your application must identify it in the I2cMaster field of the [application manifest](#).

## Concepts and samples

- [Using I2C with Azure Sphere](#)
- [Sample: I2C](#)

## Functions

FUNCTION	DESCRIPTION
I2CMaster_Open	Opens and configures an I2C master interface for exclusive use by an application, and returns a file descriptor used to perform operations on the interface.
I2CMaster_Read	Performs a read operation on an I2C master interface.
I2CMaster_SetBusSpeed	Sets the I2C bus speed for operations on the I2C master interface.
I2CMaster_SetDefaultTargetAddress	Sets the address of the subordinate device that is targeted by calls to read(2) and write(2) POSIX functions on the I2C master interface.
I2CMaster_SetTimeout	Sets the timeout for operations on an I2C master interface.
I2CMaster_Write	Performs a write operation on an I2C master interface.
I2CMaster_WriteThenRead	Performs a combined write-then-read operation on an I2C master interface.

## Typedefs

TYPEDEF	DESCRIPTION
I2C_DeviceAddress	A 7-bit or 10-bit I2C device address, which specifies the target of an I2C operation.
I2C_Interfaceld	The ID of an I2C master interface instance.

# I2CMaster\_Open Function

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/i2c.h>

Opens and configures an I2C master interface for exclusive use by an application, and returns a file descriptor used to perform operations on the interface.

```
int I2CMaster_Open(I2C_InterfaceId id);
```

## Parameters

- `id` The [ID](#) of the I2C interface to open.

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- **EACCES:** access to the I2C interface is not permitted; verify that the interface exists and is in the I2cMaster field of the application manifest.

Any other errno may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

The file descriptor of the I2C interface, or -1 for failure, in which case errno is set to the error value.

## Application manifest requirements

To access an I2c interface, your application must identify it in the I2cMaster field of the [application manifest](#).

# I2CMaster\_Read Function

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/i2c.h>

Performs a read operation on an I2C master interface. This function provides the same functionality as the POSIX read(2) function except it specifies the address of the subordinate I2C device that is the target of the operation.

```
ssize_t I2CMaster_Read(int fd, I2C_DeviceAddress address, uint8_t *buffer, size_t maxLength);
```

## Parameters

- `fd` The file descriptor for the I2C master interface.
- `address` The [address](#) of the subordinate I2C device that is the source for the read operation.
- `buffer` The output buffer that receives data from the subordinate device. This buffer must contain enough space to receive `maxLength` bytes. This can be NULL if `maxLength` is 0.
- `maxLength` The maximum number of bytes to receive. The value can be 0.

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- EBUSY: the interface is busy or the I2C clock line (SCL) is being held low.
- ENXIO: the operation didn't receive an ACK from the subordinate device.
- ETIMEDOUT: the operation timed out before completing; you can use the [I2CMaster\\_SetTimeout](#) function to adjust the timeout duration.

Any other errno may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

The number of bytes successfully read; or -1 for failure, in which case errno will be set to the error value. A partial read operation, including a read of 0 bytes, is considered a success.

## Application manifest requirements

To access an I2C interface, your application must identify it in the I2cMaster field of the [application manifest](#).

# I2CMaster\_SetBusSpeed Function

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/i2c.h>

Sets the I2C bus speed for operations on the I2C master interface.

## NOTE

Not all speeds are supported on all Azure Sphere devices. See [Using I2C](#) for details.

```
int I2CMaster_SetBusSpeed(int fd, uint32_t speedInHz);
```

## Parameters

- `fd` The file descriptor for the I2C interface.
- `speedInHz` The requested bus speed, in Hz.

## Returns

0 for success, or -1 for failure, in which case errno will be set to the error value.

## Application manifest requirements

To access an I2C interface, your application must identify it in the I2CMaster field of the [application manifest](#).

# I2CMaster\_SetDefaultTargetAddress Function

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/i2c.h>

Sets the address of the subordinate device that is targeted by calls to read(2) and write(2) POSIX functions on the I2C master interface.

## NOTE

*I2CMaster\_SetDefaultTargetAddress* is not required when using , , or , and has no impact on the address parameter of those functions.

```
int I2CMaster_SetDefaultTargetAddress(int fd, I2C_DeviceAddress address);
```

## Parameters

- `fd` The file descriptor for the I2C master interface.
- `address` The [address](#) of the subordinate I2C device that is targeted by read(2) and write(2) function calls.

## Returns

0 for success, or -1 for failure, in which case `errno` will be set to the error value. This function doesn't verify whether the device exists, so if the address is well formed, it can return success for an invalid subordinate device.

## Application manifest requirements

To access an I2C interface, your application must identify it in the `I2cMaster` field of the [application manifest](#).

# I2CMaster\_SetTimeout Function

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/i2c.h>

Sets the timeout for operations on an I2C master interface.

```
int I2CMaster_SetTimeout(int fd, uint32_t timeoutInMs);
```

## Parameters

- `fd` The file descriptor for the I2C interface.
- `timeoutInMs` The requested timeout, in milliseconds. This value may be rounded to the nearest supported value.

## Returns

0 for success, or -1 for failure, in which case `errno` will be set to the error value.

## Application manifest requirements

To access an I2C interface, your application must identify it in the `I2cMaster` field of the [application manifest](#).

# I2CMaster\_Write Function

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/i2c.h>

Performs a write operation on an I2C master interface. This function provides the same functionality as the POSIX write() function, except it specifies the address of the subordinate I2C device that is the target of the operation.

```
ssize_t I2CMaster_Write(int fd, I2C_DeviceAddress address, const uint8_t *data, size_t length);
```

## Parameters

- `fd` The file descriptor for the I2C master interface.
- `address` The [address](#) of the subordinate I2C device that is the target for the operation.
- `data` The data to transmit to the target device. This value can be NULL if length is 0.
- `length` The size of the data to transmit. This value can be 0.

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- EBUSY: the interface is busy or the I2C line is being held low.
- ENXIO: the operation didn't receive an ACK from the subordinate device.
- ETIMEDOUT: the operation timed out before completing; you can call the [I2CMaster\\_SetTimeout](#) function to adjust the timeout duration.

Any other errno may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

The number of bytes successfully written, or -1 for failure, in which case errno will be set to the error value. A partial write, including a write of 0 bytes, is considered a success.

## Application manifest requirements

To access an I2C interface, your application must identify it in the I2cMaster field of the [application manifest](#).

# I2CMaster\_WriteThenRead Function

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/i2c.h>

Performs a combined write-then-read operation on an I2C master interface. The operation appears as a single bus transaction with the following steps:

1. start condition
2. write
3. repeated start condition
4. read
5. stop condition

```
ssize_t I2CMaster_WriteThenRead(int fd, I2C_DeviceAddress address, const uint8_t *writeData, size_t lenWriteData, uint8_t *readData, size_t lenReadData);
```

## Parameters

- `fd` The file descriptor for the I2C master interface.
- `address` The [address](#) of the target I2C device for this operation.
- `writeData` The data to transmit to the targeted device.
- `lenWriteData` The byte length of the data to transmit.
- `readData` The output buffer that receives data from the target device. This buffer must contain sufficient space to receive `lenReadData` bytes.
- `lenReadData` The byte length of the data to receive.

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- EBUSY: the interface is busy or the I2C line is being held low.
- ENXIO: the operation did not receive an ACK from the subordinate device.
- ETIMEDOUT: the operation timed out before completing; you can use the [I2CMaster\\_SetTimeout](#) function to adjust the timeout duration.

Any other errno may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

The combined number of bytes successfully written and read, or -1 for failure, in which case errno is set to the error value. A partial result, including a transfer of 0 bytes, is considered a success.

## Application manifest requirements

To access an I2c interface, your application must identify it in the I2cMaster field of the [application manifest](#).

# I2C\_DeviceAddress Typedef

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/i2c.h>

A 7-bit or 10-bit I2C device address, which specifies the target of an I2C operation. This address must not contain additional information, such as read/write bits.

## NOTE

Not all Azure Sphere devices support 10-bit addresses.

```
typedef uint I2C_DeviceAddress;
```

# I2C\_InterfaceId Typedef

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/i2c.h>

The ID of an I2C master interface instance.

```
typedef int I2C_InterfaceId;
```

# Applib log.h

1/7/2019 • 2 minutes to read

**Header:** #include <applibs/log.h>

The Applibs log header contains functions that log debug messages. Debug messages are displayed when you debug an application through the Azure Sphere SDK. These functions are thread safe.

## Concepts and samples

- [Error handling and logging](#)
- [Sample: CurlEasyHttps](#)
- [Sample: CurlMultiHttps](#)
- [Sample: System Time](#)

## Functions

FUNCTION	DESCRIPTION
<a href="#">Log_Debug</a>	Logs and formats a debug message with printf formatting.
<a href="#">Log_DebugVarArgs</a>	Logs and formats a debug message with vprintf formatting.

# Log\_Debug Function

1/7/2019 • 2 minutes to read

**Header:** #include <applibs/log.h>

Logs and formats a debug message with printf formatting. The caller needs to provide an additional parameter for every argument specification defined in the *fmt* string. This function is thread safe.

```
int Log_Debug(const char * fmt, ...);
```

## Parameters

- `fmt` The message string to log, with optional argument specifications.

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- EFAULT: the *fmt* is NULL.

Any other errno may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

0 for success, or -1 in the case of failure, in which case errno is set to the error.

# Log\_DebugVarArgs Function

1/7/2019 • 2 minutes to read

**Header:** #include <applibs/log.h>

Logs and formats a debug message with vprintf formatting. This function is thread safe.

The *args* va\_list parameter should be initialized with va\_start before this function is called, and should be cleaned up by calling va\_end afterwards. The caller needs to provide an additional parameter for every argument specification defined in the *fmt* string.

```
int Log_DebugVarArgs(const char * fmt, va_list args);
```

## Parameters

- `fmt` The message string to log.
- `args` An argument list that has been initialized with va\_start.

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- EFAULT: the *fmt* is NULL.

Any other errno may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

0 for success, or -1 for failure, in which case errno is set to the error value.

# Applibs networking.h

5/30/2019 • 2 minutes to read

**Header:** #include <applibs/networking.h>

The Applibs networking header contains functions and types that interact with the networking subsystem to query the network state, and to get and set the network service configuration.

## Concepts and samples

- [Sample: Private Network Services](#)
- [Sample: HTTPS](#)

## Functions

FUNCTION	DESCRIPTION
<a href="#">Networking_DhcpServer_Start</a>	Registers, configures, and starts the DHCP server for a network interface.
<a href="#">Networking_DhcpServerConfig_Destroy</a>	Destroys a <a href="#">Networking_DhcpServerConfig</a> struct.
<a href="#">Networking_DhcpServerConfig_Init</a>	Initializes a <a href="#">Networking_DhcpServerConfig</a> struct with the default DHCP Server configuration.
<a href="#">Networking_DhcpServerConfig_SetLease</a>	Applies lease information to a <a href="#">Networking_DhcpServerConfig</a> struct.
<a href="#">Networking_DhcpServerConfig_SetNtpServerAddresses</a>	Applies a set of NTP server IP addresses to a <a href="#">Networking_DhcpServerConfig</a> struct.
<a href="#">Networking_GetInterfaceConnectionStatus</a>	Gets the network connection status for a network interface.
<a href="#">Networking_GetInterfaceCount</a>	Gets the number of network interfaces in an Azure Sphere device.
<a href="#">Networking_GetInterfaces</a>	Gets the list of network interfaces in an Azure Sphere device.
<a href="#">Networking_GetNtpState</a>	<b>Obsolete:</b> This function is obsolete. Use <a href="#">Networking_TimeSync_GetEnabled</a> instead.
<a href="#">Networking_InitDhcpServerConfiguration</a>	<b>Obsolete:</b> This function is obsolete. Use <a href="#">Networking_DhcpServerConfig_Init</a> instead.
<a href="#">Networking_InitStaticIpConfiguration</a>	<b>Obsolete:</b> This function is obsolete. Use <a href="#">Networking_IpConfig_Init</a> instead.
<a href="#">Networking_IpConfig_Apply</a>	Applies an IP configuration to a network interface.

FUNCTION	DESCRIPTION
<a href="#">Networking_IpConfig_Destroy</a>	Destroys a <a href="#">Networking_IpConfig</a> struct.
<a href="#">Networking_IpConfig_EnableDynamicIp</a>	Enables dynamic IP and disables static IP for a <a href="#">Networking_IpConfig</a> struct.
<a href="#">Networking_IpConfig_EnableStaticIp</a>	Enables static IP and disables dynamic IP for a <a href="#">Networking_IpConfig</a> struct.
<a href="#">Networking_IpConfig_Init</a>	Initializes a <a href="#">Networking_IpConfig</a> struct with the default IP configuration.
<a href="#">Networking_IsNetworkingReady</a>	Verifies whether internet connectivity is available and time is synced.
<a href="#">Networking_SetInterfaceState</a>	Enables or disables a network interface.
<a href="#">Networking_SetNtpState</a>	<b>Obsolete:</b> This function is obsolete. Use <a href="#">Networking_TimeSync_SetEnabled</a> instead.
<a href="#">Networking_SetStaticIp</a>	<b>Obsolete:</b> This function is obsolete. To configure a static IP address, use <a href="#">Networking_IpConfig_EnableStaticIp</a> on an initialized <a href="#">Networking_IpConfig</a> struct, then apply it with <a href="#">Networking_IpConfig_Apply</a> .
<a href="#">Networking_SntpServer_Start</a>	Registers and starts an SNTP server for a network interface.
<a href="#">Networking_SntpServerConfig_Destroy</a>	Destroys a <a href="#">Networking_SntpServerConfig</a> struct.
<a href="#">Networking_SntpServerConfig_Init</a>	Initializes a <a href="#">Networking_SntpServerConfig</a> struct with the default SNTP Server configuration.
<a href="#">Networking_StartDhcpServer</a>	<b>Obsolete:</b> This function is obsolete. Use <a href="#">Networking_DhcpServer_Start</a> instead.
<a href="#">Networking_StartSntpServer</a>	<b>Obsolete:</b> This function is obsolete. Use <a href="#">Networking_SntpServer_Start</a> instead.
<a href="#">Networking_TimeSync_GetEnabled</a>	Indicates whether the time-sync service is enabled.
<a href="#">Networking_TimeSync_SetEnabled</a>	Enables or disables the time-sync service.

## Structs

STRUCT	DESCRIPTION
<a href="#">Networking_DhcpServerConfiguration</a>	The DHCP server configuration for a network interface.
<a href="#">Networking_DhcpServerConfig</a>	An opaque buffer that represents the DHCP server configuration for a network interface.
<a href="#">Networking_NetworkInterface</a>	The properties of a network interface.

STRUCT	DESCRIPTION
<a href="#">Networking_SntpServerConfig</a>	An opaque buffer that represents the SNTP server configuration for a network interface.
<a href="#">Networking_IpConfig</a>	An opaque buffer that represents the IP configuration for a network interface.
<a href="#">Networking_StaticIpConfiguration</a>	The static IP address configuration for a network interface.

## Enums

ENUM	DESCRIPTION
<a href="#">Networking_InterfaceConnectionStatus</a>	The connection status of a network interface.
<a href="#">Networking_InterfaceMedium</a>	The valid network technologies used by the network interface.
<a href="#">Networking_IpConfiguration</a>	<b>Renamed:</b> This enum was renamed <a href="#">Networking_IpType</a> .
<a href="#">Networking_IpType</a>	The IP configuration options for a network interface.

## Typedefs

TYPEDEF	DESCRIPTION
<a href="#">Networking_InterfaceMedium_Type</a>	Specifies the type for <a href="#">Networking_InterfaceMedium</a> enum values.
<a href="#">Networking_IpConfiguration_Type</a>	<b>Renamed:</b> This type was renamed <a href="#">Networking_IpType_Type</a> .
<a href="#">Networking_IpType_Type</a>	Specifies the type for <a href="#">Networking_IpType</a> enum values.

# Networking\_DhcpServer\_Start Function

5/30/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/networking.h>

Registers, configures, and starts the DHCP server for a network interface. The configuration specified by this function call overwrites the existing configuration.

```
int Networking_DhcpServer_Start(const char *networkInterfaceName, const Networking_DhcpServerConfig *dhcpServerConfig);
```

## Parameters

- `networkInterfaceName` The name of the network interface to configure.
- `dhcpServerConfig` A pointer to the [Networking\\_DhcpServerConfig](#) struct that represents the DHCP server configuration.

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- EACCES: the application manifest does not include the DhcpService capability.
- ENOENT: the `networkInterfaceName` parameter refers to an interface that does not exist.
- EPERM: this operation is not allowed on the network interface.
- EFAULT: the `networkInterfaceName` parameter is NULL.
- EFAULT: the `dhcpServerConfig` parameter is NULL.
- EAGAIN: the networking stack isn't ready.
- EINVAL: the configuration struct has invalid parameters.

Any other errno may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

0 for success, or -1 for failure, in which case errno will be set to the error value.

## Remarks

If the network interface is up when this function is called, the DHCP server will be shut down, configured, and started. If the interface is down, the server will start when the interface is up.

The interface must be configured with a static IP address before this function is called; otherwise, the EPERM error is returned.

## Application manifest requirements

The [application manifest](#) must include the DhcpService capability.

## Concepts and samples

- [Sample: Private Network Services](#)

# Networking\_DhcpServerConfig\_Destroy Function

5/30/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/networking.h>

Destroys a [Networking\\_DhcpServerConfig](#) struct.

```
void Networking_DhcpServerConfig_Destroy(Networking_DhcpServerConfig *dhcpServerConfig);
```

## Parameters

- `dhcpServerConfig` A pointer to the `Networking_DhcpServerConfig` struct to destroy.

## Remarks

It's unsafe to call `Networking_DhcpServerConfig_Destroy` on a struct that hasn't been initialized. After `Networking_DhcpServerConfig_Destroy` is called, the `Networking_DhcpServerConfig` struct must not be used until it is re-initialized with the [Networking\\_DhcpServerConfig\\_Init](#) function.

## Concepts and samples

- [Sample: Private Network Services](#)

# Networking\_DhcpServerConfig\_Init Function

5/30/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/networking.h>

Initializes a [Networking\\_DhcpServerConfig](#) struct with the default DHCP Server configuration.

```
void Networking_DhcpServerConfig_Init(Networking_DhcpServerConfig *dhcpServerConfig);
```

## Parameters

- `dhcpServerConfig` A pointer to the `Networking_DhcpServerConfig` struct that receives the default DHCP server configuration.

## Remarks

When the `Networking_DhcpServerConfig` struct is no longer needed, the [Networking\\_DhcpServerConfig\\_Destroy](#) function must be called on the struct to avoid resource leaks.

## Concepts and samples

- [Sample: Private Network Services](#)

# Networking\_DhcpServerConfig\_SetLease Function

5/30/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/networking.h>

Applies lease information to a [Networking\\_DhcpServerConfig](#) struct.

```
int Networking_DhcpServerConfig_SetLease(Networking_DhcpServerConfig *dhcpServerConfig, struct in_addr startIpAddress, uint8_t ipAddressCount, struct in_addr subnetMask, struct in_addr gatewayAddress, uint32_t leaseTimeInHours);
```

## Parameters

- `dhcpServerConfig` A pointer to the `Networking_DhcpServerConfig` struct to update.
- `startIpAddress` The starting IP address in the address range to lease.
- `ipAddressCount` The number of IP addresses the server can lease.
- `subnetMask` The subnet mask for the IP addresses.
- `gatewayAddress` The gateway address for the network interface.
- `leaseTimeInHours` The duration of the lease, in hours.

## Errors

If an error is encountered, returns -1 and sets `errno` to the error value.

- EFAULT: the `dhcpServerConfig` parameter is NULL.

Any other `errno` may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

0 for success, or -1 for failure, in which case `errno` will be set to the error value.

## Concepts and samples

- [Sample: Private Network Services](#)

# Networking\_DhcpServerConfig\_SetNtpServerAddresses Function

5/30/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/networking.h>

Applies a set of NTP server IP addresses to a [Networking\\_DhcpServerConfig](#) struct.

```
int Networking_DhcpServerConfig_SetNtpServerAddresses(Networking_DhcpServerConfig *dhcpServerConfig, const
   struct in_addr *ntpServerAddresses, size_t serverCount);
```

## Parameters

- `dhcpServerConfig` A pointer to the `Networking_DhcpServerConfig` struct to update.
- `ntpServerAddresses` A pointer to an array of NTP server IP addresses.
- `serverCount` The number of IP addresses in the `ntpServerAddresses` array.

## Errors

If an error is encountered, returns -1 and sets `errno` to the error value.

- EFAULT: the `dhcpServerConfig` parameter is NULL.
- EFAULT: the `ntpServerAddresses` parameter is NULL.
- EINVAL: More than three IP address were provided.

Any other `errno` may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

0 for success, or -1 for failure, in which case `errno` will be set to the error value.

## Concepts and samples

- [Sample: Private Network Services](#)

# Networking\_GetInterfaceConnectionStatus Function

5/30/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/networking.h>

Gets the network connection status for a network interface.

```
int Networking_GetInterfaceConnectionStatus(const char *networkInterfaceName,  
    Networking_InterfaceConnectionStatus *outStatus);
```

## Parameters

- `networkInterfaceName` The name of the network interface.
- `outStatus` A pointer to the `Networking_InterfaceConnectionStatus` enum that receives the network connection status.

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- EFAULT: the `outStatus` parameter is NULL.
- ENOENT: the `networkInterfaceName` interface does not exist.
- EAGAIN: the networking stack isn't ready.

Any other errno may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

0 for success, -1 for failure, in which case errno is set to the error value.

## Concepts and samples

- [Sample: Private Network Services](#)

# Networking\_GetInterfaceCount Function

5/30/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/networking.h>

Gets the number of network interfaces in an Azure Sphere device.

```
ssize_t Networking_GetInterfaceCount(void);
```

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- EAGAIN: the networking stack isn't ready yet.

Any errno may be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

The number of network interfaces, or -1 for failure, in which case errno is set to the error value.

## Concepts and samples

- [Sample: Private Network Services](#)

# Networking\_GetInterfaces Function

5/30/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/networking.h>

Gets the list of network interfaces in an Azure Sphere device. If *outNetworkInterfaces* is too small to hold all network interfaces in the system, this function fills the array and returns the number of array elements. The number of interfaces in the system will not change within a boot cycle.

```
ssize_t Networking_GetInterfaces(Networking_NetworkInterface *outNetworkInterfacesArray, size_t  
networkInterfacesArrayCount);
```

## Parameters

- `outNetworkInterfacesArray` A pointer to an array of `Networking_NetworkInterface` structs to fill with network interface properties. The caller must allocate memory for the array after calling `Networking_GetInterfacesCount` to retrieve the number of interfaces on the device.
- `networkInterfacesArrayCount` The number of elements *outNetworkInterfacesArray* can hold. The array should have one element for each network interface on the device.

## Errors

If an error is encountered, returns -1 and sets `errno` to the error value.

- `EFAULT`: the *outNetworkInterfacesArray* parameter is NULL.
- `ERANGE`: the *networkInterfacesArrayCount* parmaeter is 0.
- `EAGAIN`: the networking stack isn't ready yet.

Any other `errno` may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

The number of network interfaces added to the *outNetworkInterfaces* array. Otherwise -1 for failure, in which case `errno` is set to the error value.

## Concepts and samples

- [Sample: Private Network Services](#)

# Networking\_GetNtpState Function

5/30/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/networking.h>

### IMPORTANT

This function is obsolete. Use [Networking\\_TimeSync\\_GetEnabled](#) instead.

Indicates whether the NTP time-sync service is enabled.

```
int Networking_GetNtpState(bool *outIsEnabled);
```

## Parameters

- `outIsEnabled` Receives a pointer to a Boolean value that indicates whether the NTP time-sync service is enabled. The value is set to true if the NTP service is enabled, and false if it is disabled.

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- EFAULT: the output parameter (outIsEnabled) provided is null.
- EAGAIN: the networking stack isn't ready yet.

Any other errno may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

0 for success, or -1 for failure, in which case errno is set to the error value.

# Networking\_InitDhcpServerConfiguration Function

5/30/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/networking.h>

### IMPORTANT

This function is obsolete. Use [Networking\\_DhcpServerConfig\\_Init](#) instead.

Initializes a [Networking\\_DhcpServerConfiguration](#) struct with the default DHCP server parameters.

```
static inline void Networking_InitDhcpServerConfiguration(Networking_DhcpServerConfiguration
*dhcpServerConfiguration);
```

## Parameters

- `dhcpServerConfiguration` A pointer to a [Networking\\_DhcpServerConfiguration](#) struct that returns the default DHCP server parameters.

# Networking\_InitStaticIpConfiguration Function

5/30/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/networking.h>

### IMPORTANT

This function is obsolete. Use [Networking\\_IpConfig\\_Init](#) instead.

Initializes a [Networking\\_StaticIpConfiguration](#) struct with the default static IP configuration.

```
static inline void Networking_InitStaticIpConfiguration(Networking_StaticIpConfiguration  
*staticIpConfiguration);
```

## Parameters

- `staticIpConfiguration` A pointer to a [Networking\\_StaticIpConfiguration](#) struct that returns the default static IP parameters.

# Networking\_IpConfig\_Apply Function

5/30/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/networking.h>

Applies an IP configuration to a network interface.

```
int Networking_IpConfig_Apply(const char *networkInterfaceName, const Networking_IpConfig *ipConfig);
```

## Parameters

- `networkInterfaceName` The name of the network interface to configure.
- `ipConfig` A pointer to the [Networking\\_IpConfig](#) struct that contains the IP configuration to apply.

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- EACCES: the calling application doesn't have the NetworkConfig capability.
- ENOENT: the `networkInterfaceName` parameter refers to an interface that does not exist.
- EPERM: this operation is not allowed on the network interface.
- EFAULT: the `networkInterfaceName` or `ipConfig` parameter is NULL.
- EAGAIN: the networking stack isn't ready yet.

Any other errno may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

0 for success, or -1 for failure, in which case errno will be set to the error value.

## Remarks

This function does not verify whether the static IP address is compatible with the dynamic IP addresses received through an interface using a DHCP client.

This function does not verify whether a DHCP server is available on the network and if a dynamic IP address is configured.

If overlapping IP address configurations are present on a device, the behavior of this function is undefined.

## Application manifest requirements

The [application manifest](#) must include the NetworkConfig capability.

## Concepts and samples

- Sample: Private Network Services

# Networking\_IpConfig\_Destroy Function

5/30/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/networking.h>

Destroys a [Networking\\_IpConfig](#) struct.

```
void Networking_IpConfig_Destroy(Networking_IpConfig *ipConfig);
```

## Parameters

- `ipConfig` A pointer to the `Networking_IpConfig` struct to destroy.

## Remarks

It's unsafe to call `Networking_IpConfig_Destroy` on a struct that hasn't been initialized. After `Networking_IpConfig_Destroy` is called, the `Networking_IpConfig` struct must not be used until it is re-initialized with the [Networking\\_IpConfig\\_Init](#) function.

## Concepts and samples

- [Sample: Private Network Services](#)

# Networking\_IpConfig\_EnableDynamicIp Function

5/30/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/networking.h>

Enables dynamic IP and disables static IP for a [Networking\\_IpConfig](#) struct.

```
void Networking_IpConfig_EnableDynamicIp(Networking_IpConfig *ipConfig);
```

## Parameters

- `ipConfig` A pointer to the `Networking_IpConfig` struct to update.

# Networking\_IpConfig\_EnableStaticIp Function

5/30/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/networking.h>

Enables static IP and disables dynamic IP for a [Networking\\_IpConfig](#) struct.

```
void Networking_IpConfig_EnableStaticIp(Networking_IpConfig *ipConfig, struct in_addr ipAddress, struct in_addr subnetMask, struct in_addr gatewayAddress);
```

## Parameters

- `ipConfig` A pointer to the `Networking_IpConfig` struct to update.
- `ipAddress` The static IP address.
- `subnetMask` The static subnet mask.
- `gatewayAddress` The static gateway address.

## Concepts and samples

- [Sample: Private Network Services](#)

# Networking\_IpConfig\_Init Function

5/30/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/networking.h>

Initializes a [Networking\\_IpConfig](#) struct with the default IP configuration. The default IP configuration enables dynamic IP.

```
void Networking_IpConfig_Init(Networking_IpConfig *ipConfig);
```

## Parameters

- `ipConfig` A pointer to the Networking\_IpConfig struct that receives the default IP configuration.

## Remarks

When the Networking\_IpConfig struct is no longer needed, the [Networking\\_IpConfig\\_Destroy](#) function must be called on the struct to avoid resource leaks.

## Concepts and samples

- [Sample: Private Network Services](#)

# Networking\_IsNetworkingReady Function

5/30/2019 • 2 minutes to read

**Header:** #include <applibs/networking.h>

Verifies whether internet connectivity is available and time is synced.

```
int Networking_IsNetworkingReady(bool * outIsNetworkingReady);
```

## Parameters

- `outIsNetworkingReady` A pointer to a boolean that returns the result. This value is set to true if networking is ready, otherwise false.

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- EFAULT: the provided `outIsNetworkingReady` parameter is NULL.

Any other errno may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

0 for success, -1 for failure, in which case errno is set to the error value.

## Concepts and samples

- [Sample: CurlEasyHttps](#)

# Networking\_SetInterfaceState Function

5/30/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/networking.h>

Enables or disables a network interface.

```
int Networking_SetInterfaceState(const char *networkInterfaceName, bool isEnabled);
```

## Parameters

- `networkInterfaceName` The name of the network interface to update.
- `isEnabled` true to enable the interface, false to disable it.

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- EACCES: the application manifest does not include the NetworkConfig capability.
- ENOENT: the network interface does not exist.
- EPERM: this function is not allowed on the interface.
- EAGAIN: the networking stack isn't ready yet.
- EIO: the interface is inaccessible.

Any other errno may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

0 for success, or -1 for failure, in which case errno is set to the error value.

## Application manifest requirements

The [application manifest](#) must include the NetworkConfig capability.

## Concepts and samples

- [Sample: Private Network Services](#)

# Networking\_SetNtpState Function

5/30/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/networking.h>

### IMPORTANT

This function is obsolete. Use [Networking\\_TimeSync\\_SetEnabled](#) instead.

Enables or disables the NTP time-sync service. The changes take effect immediately without a device reboot and are persisted. The NTP service is then configured as requested at boot time. This function allows applications to override the default behavior, which is to enable the NTP service at boot time.

### NOTE

The NTP service is enabled by default. If you set the system time while the NTP service is enabled, it will overwrite the UTC time when the device has internet connectivity. You can disable the NTP service; however, this can cause OTA updates on the device to fail if the difference between the system time and the NTP server time is too great.

```
int Networking_SetNtpState(bool isEnabled);
```

## Parameters

- `isEnabled` true to enable the NTP service, false to disable it.

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- EACCES: the caller doesn't have the NetworkConfig capability.
- EAGAIN: the networking stack isn't ready yet.

Any other errno may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

0 for success, or -1 for failure, in which case errno is set to the error value.

# Networking\_SetStaticIp Function

5/30/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/networking.h>

### IMPORTANT

This function is obsolete. To configure a static IP address, use [Networking\\_IpConfig\\_EnableStaticIp](#) on an initialized [Networking\\_IpConfig](#) struct, then apply it with [Networking\\_IpConfig\\_Apply](#).

Sets the static IP configuration for a network interface.

### NOTE

This function doesn't verify whether the static IP address is compatible with dynamic IP addresses that are assigned to an interface by a DHCP client. If overlapping IP address configurations are present on a device, the behavior is not guaranteed.

```
int Networking_SetStaticIp(const char *networkInterfaceName, const Networking_StaticIpConfiguration *staticIpConfiguration);
```

## Parameters

- `networkInterfaceName` A string that contains the name of the network interface to configure.
- `staticIpConfiguration` A pointer to the [Networking\\_StaticIpConfiguration](#) struct that contains the static IP configuration.

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- EACCES: the application manifest does not include the NetworkConfig capability.
- ENOENT: the network interface does not exist.
- EPERM: this operation is not allowed on the network interface.
- EFAULT: the `staticIpConfiguration` parameter is NULL.
- EAGAIN: the networking stack isn't ready yet.

Any other errno may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

0 for success, -1 for failure, in which case errno is set to the error value.

# Networking\_SntpServer\_Start Function

5/30/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/networking.h>

Registers and starts an SNTP server for a network interface.

```
int Networking_SntpServer_Start(const char *networkInterfaceName, const Networking_SntpServerConfig *sntpServerConfig);
```

## Parameters

- `networkInterfaceName` The name of the network interface to configure.
- `sntpServerConfig` A pointer to the [Networking\\_SntpServerConfig](#) struct that represents the SNTP server configuration.

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- EACCES: the calling application doesn't have the SntpService capability.
- EFAULT: the `networkInterfaceName` parameter is NULL.
- EFAULT: the `sntpServerConfig` parameter is NULL.
- ENOENT: the `networkInterfaceName` parameter refers to an interface that does not exist..
- EPERM: this operation is not allowed on the network interface.
- EAGAIN: the networking stack isn't ready.

Any other errno may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

0 for success, or -1 for failure, in which case errno will be set to the error value.

## Remarks

If the SNTP server is already running and attached to the interface, this function returns success. If the `networkInterfaceName` interface is down or disabled, the SNTP server is registered for the interface but server isn't started.

## Application manifest requirements

The [application manifest](#) must include the SntpService capability.

## Concepts and samples

- [Sample: Private Network Services](#)

# Networking\_SntpServerConfig\_Destroy Function

5/30/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/networking.h>

Destroys a [Networking\\_SntpServerConfig](#) struct.

```
void Networking_SntpServerConfig_Destroy(Networking_SntpServerConfig *sntpServerConfig);
```

## Parameters

- `sntpServerConfig` A pointer to the `Networking_SntpServerConfig` struct to destroy.

## Remarks

It's unsafe to call `Networking_SntpServerConfig_Destroy` on a struct that hasn't been initialized. After `Networking_SntpServerConfig_Destroy` is called, the `Networking_SntpServerConfig` struct must not be used until it is re-initialized with the [Networking\\_SntpServerConfig\\_Init](#) function.

## Concepts and samples

- [Sample: Private Network Services](#)

# Networking\_SntpServerConfig\_Init Function

5/30/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/networking.h>

Initializes a [Networking\\_SntpServerConfig](#) struct with the default SNTP server configuration.

```
void Networking_SntpServerConfig_Init(Networking_SntpServerConfig *sntpServerConfig);
```

## Parameters

- `sntpServerConfig` A pointer to the `Networking_SntpServerConfig` struct that receives the default SNTP server configuration.

## Remarks

When the `Networking_SntpServerConfig` struct is no longer needed, the [Networking\\_SntpServerConfig\\_Destroy](#) function must be called on the struct to avoid resource leaks.

## Concepts and samples

- [Sample: Private Network Services](#)

# Networking\_StartDhcpServer Function

5/30/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/networking.h>

### IMPORTANT

This function is obsolete. Use [Networking\\_DhcpServer\\_Start](#) instead.

Registers and starts the DHCP server for a network interface.

The DHCP server configuration that is passed to this function call overwrites any existing configuration. If the network interface is running when this function is called, the DHCP server is restarted and reconfigured. If the network interface is not running, the DHCP server will start when the interface starts. The network interface must be configured with a static IP address before this function is called; otherwise, EPERM is returned.

```
int Networking_StartDhcpServer(const char *networkInterfaceName, const Networking_DhcpServerConfiguration *dhcpServerConfiguration);
```

## Parameters

- `networkInterfaceName` A string that contains the name of the network interface to configure.
- `dhcpServerConfiguration` A pointer to the [DhcpServerConfiguration](#) struct that contains the DHCP server configuration.

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- EACCES: the application manifest does not include the NetworkConfig capability.
- ENOENT: the network interface does not exist.
- EPERM: this operation is not allowed on the network interface.
- EFAULT: the `dhcpServerConfiguration` parameter is NULL.
- EAGAIN: the networking stack isn't ready yet.
- EINVAL: the configuration struct has invalid parameters.

Any other errno may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

0 for success, -1 for failure, in which case errno is set to the error value.

# Networking\_StartSntpServer Function

5/30/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/networking.h>

### IMPORTANT

This function is obsolete. Use [Networking\\_SntpServer\\_Start](#) instead.

Registers and starts the SNTP server for a network interface.

If the SNTP server is already running, this function returns success. If the network interface is shutdown or disabled, then the SNTP server is registered but will not start until the interface starts.

```
int Networking_StartSntpServer(const char *networkInterfaceName);
```

## Parameters

- `networkInterfaceName` A string that contains the name of the network interface of the SNTP server.

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- EACCES: the application manifest does not include the NetworkConfig capability.
- EFAULT: the `networkInterfaceName` parameter is NULL.
- ENOENT: the network interface does not exist.
- EPERM: this operation is not allowed on the network interface.
- EAGAIN: the networking stack isn't ready.

Any other errno may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

0 for success, -1 for failure, in which case errno is set to the error value.

# Networking\_TimeSync\_GetEnabled Function

5/30/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/networking.h>

Indicates whether the time-sync service is enabled.

```
int Networking_TimeSync_GetEnabled(bool *outIsEnabled);
```

## Parameters

- `outIsEnabled` A pointer to a Boolean value that receives the state of the time-sync service. The value is set to true if the time-sync service is enabled, otherwise false.

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- EFAULT: the `outIsEnabled` parameter provided is null.
- EAGAIN: the networking stack isn't ready.

Any other errno may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

0 for success, or -1 for failure, in which case errno will be set to the error value.

## Concepts and samples

- [Sample: System Time](#)

# Networking\_TimeSync\_SetEnabled Function

5/30/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/networking.h>

Enables or disables the time-sync service.

The changes take effect immediately without a device reboot and persist through device reboots. The time-sync service is then configured as requested at boot time. This function allows applications to override the default behavior, which is to enable time-sync at boot time.

```
int Networking_TimeSync_SetEnabled(bool enabled);
```

## Parameters

- `enabled` true to enable the time-sync service; false to disable it.

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- EACCES: the calling application doesn't have the TimeSyncConfig capability.
- EAGAIN: the networking stack isn't ready.

Any other errno may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

0 for success, or -1 for failure, in which case errno will be set to the error value.

## Application manifest requirements

The [application manifest](#) must include the TimeSyncConfig capability.

# Networking\_DhcpServerConfig Struct

5/30/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/networking.h>

An opaque buffer that represents the DHCP server configuration for a network interface.

```
typedef struct Networking_DhcpServerConfig Networking_DhcpServerConfig;
```

## Concepts and samples

- [Sample: Private Network Services](#)

# Networking\_DhcpServerConfiguration Struct

2/14/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/networking.h>

The DHCP server configuration for a network interface.

### NOTE

This is an alias to a versioned structure. Define NETWORKING\_STRUCTS\_VERSION to use this alias.

```
struct Networking_DhcpServerConfiguration {
    uint32_t z__magicAndVersion;
    struct in_addr startIpAddress;
    uint8_t ipAddressCount;
    struct in_addr netMask;
    struct in_addr gatewayAddress;
    struct in_addr ntpServers[3];
    struct uint32_t leaseTimeHours;
};
```

## Members

### **uint32\_t z\_magicAndVersion**

A magic number that uniquely identifies the struct version.

### **struct in\_addr startIpAddress**

The starting IP address. This parameter is in network byte order.

### **uint8\_t ipAddressCount**

The number of incrementing IP addresses that are supported. The only supported value is 1.

### **struct in\_addr netMask**

The netmask for the IP addresses. This parameter is in network byte order.

### **struct in\_addr gatewayAddress**

The gateway address for the interface. This parameter is in network byte order.

### NOTE

Azure Sphere does not support IP routing. This address can indicate an alternate gateway on a private network. All zeros indicate an unspecified value and the DHCP server will not return this option to the client. The gateway address must be in the same subnet as the IP address range specified by `startIpAddress` and `ipAddressCount`, and must not overlap with that range.

### **struct in\_addr ntpServers[3]**

The NTP server addresses in order of preference. Up to 3 addresses are supported. All zeros indicate an unspecified value and the DHCP server will not return this option to the client. This parameter is in network byte

order.

**uint32\_t leaseTimeHours**

The lease time for IP addresses, in hours. The minimum supported value is 1 and the maximum is 24.

# Networking\_IpConfig Struct

5/30/2019 • 2 minutes to read

## BETA feature

**Header:** #include <applibs/networking.h>

An opaque buffer that represents the IP configuration for a network interface.

```
typedef struct Networking_IpConfig Networking_IpConfig;
```

## Concepts and samples

- [Sample: Private Network Services](#)

# Networking\_NetworkInterface Struct

5/30/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/networking.h>

The properties of a network interface.

### NOTE

This is an alias to a versioned structure. Define NETWORKING\_STRUCTS\_VERSION to use this alias.

```
struct Networking_NetworkInterface {
    uint32_t z__magicAndVersion;
    bool isEnabled;
    char interfaceName[IF_NAMESIZE];
    uint32_t interfaceNameLength;
    Networking_IpType_Type ipConfigurationType;
    Networking_InterfaceMedium_Type interfaceMediumType;
};
```

## Members

### **uint32\_t z\_\_magicAndVersion**

A magic number that uniquely identifies the struct version.

### **bool isEnabled**

Indicates whether the network interface is enabled.

### **char interfaceName[IF\_NAMESIZE]**

The network interface name.

### **uint32\_t interfaceNameLength**

The length of the network interface name.

### **Networking\_IpType\_Type ipConfigurationType**

The [Networking\\_IpType](#) enum that contains the IP types for the interface.

### **Networking\_InterfaceMedium\_Type InterfaceMediumType**

The [Networking\\_InterfaceMedium](#) enum that contains the network types for the interface.

## Concepts and samples

- [Sample: Private Network Services](#)

# Networking\_SntpServerConfig Struct

5/30/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/networking.h>

An opaque buffer that represents the SNTP server configuration for a network interface.

```
typedef struct Networking_SntpServerConfig Networking_SntpServerConfig;
```

## Concepts and samples

- [Sample: Private Network Services](#)

# Networking\_StaticIpConfiguration Struct

2/14/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/networking.h>

The static IP address configuration for a network interface.

### NOTE

This is an alias to a versioned structure. Define NETWORKING\_STRUCTS\_VERSION to use this alias.

```
struct Networking_StaticIpConfiguration {  
    uint32_t z__magicAndVersion;  
    struct in_addr ipAddress;  
    struct in_addr netMask;  
    struct in_addr gatewayAddress;  
};
```

## Members

### **uint32\_t z\_\_magicAndVersion**

A magic number that uniquely identifies the struct version.

### **struct in\_addr ipAddress**

The Static IP address for the interface.

### **struct in\_addr netMask**

The netmask for the static IP address.

### **struct in\_addr gatewayAddress**

The gateway address for the interface. This should be set to 0.0.0.0

# Networking\_InterfaceConnectionStatus Enum

5/30/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/networking.h>

A bit mask that specifies the connection status of a network interface.

```
typedef enum Networking_InterfaceConnectionStatus {
    Networking_InterfaceConnectionStatus_InterfaceUp = 1 << 0,
    Networking_InterfaceConnectionStatus_ConnectedToNetwork = 1 << 1,
    Networking_InterfaceConnectionStatus_IpAvailable = 1 << 2,
    Networking_InterfaceConnectionStatus_ConnectedToInternet = 1 << 3
} Networking_InterfaceConnectionStatus;
```

## Values

### Networking\_InterfaceConnectionStatus\_InterfaceUp

The interface is enabled.

### Networking\_InterfaceConnectionStatus\_ConnectedToNetwork

The interface is connected to a network.

### Networking\_InterfaceConnectionStatus\_IpAvailable

The interface has an IP address assigned to it.

### Networking\_InterfaceConnectionStatus\_ConnectedToInternet

The interface is connected to the internet.

## Remarks

This enum is used by the [Networking\\_GetInterfaceConnectionStatus](#) function.

## Concepts and samples

- [Sample: Private Network Services](#)
- [Sample: Wi-Fi setup and device control via BLE - reference solution](#)

# Networking\_InterfaceMedium Enum

5/30/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/networking.h>

The valid network technologies used by the network interface.

```
typedef enum {
    Networking_InterfaceMedium_Unspecified = 0,
    Networking_InterfaceMedium_Wifi = 1,
    Networking_InterfaceMedium_Ethernet = 2
} Networking_InterfaceMedium;
```

VALUES	DESCRIPTIONS
Networking_InterfaceMedium_Unspecified = 0	The network technology is unspecified.
Networking_InterfaceMedium_Wifi = 1	Wi-Fi.
Networking_InterfaceMedium_Ethernet = 2	Ethernet.

## Concepts and samples

- [Sample: Private Network Services](#)

# Networking\_IpConfiguration Enum

5/30/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/networking.h>

### NOTE

This type was renamed [Networking\\_IpType](#).

# Networking\_IpType Enum

5/30/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/networking.h>

The IP configuration options for a network interface.

```
typedef enum {
    Networking_IpType_DhcpNone = 0,
    Networking_IpType_DhcpClient = 1
} Networking_IpType;
```

## Values

### Networking\_IpType\_DhcpNone

The interface doesn't have a DHCP client attached, so a static IP address must be [enabled](#) and then [applied](#) to the interface.

### Networking\_IpType\_DhcpClient

The interface has a DHCP client attached and is configured to use dynamic IP assignment.

## Remarks

This enum is used by the [Networking\\_NetworkInterface](#) struct.

## Concepts and samples

- [Sample: Private Network Services](#)

# Networking\_InterfaceMedium\_Type Typedef

5/30/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/networking.h>

Specifies the type of [Networking\\_InterfaceMedium](#) enum values.

```
typedef uint8_t Networking_InterfaceMedium_Type;
```

## Concepts and samples

- [Sample: Private Network Services](#)

# Networking\_IpConfiguration\_Type\_Typedef

5/30/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/networking.h>

### NOTE

This type was renamed [Networking\\_IpType\\_Type](#).

# Networking\_IpType\_Type Typedef

5/30/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/networking.h>

Specifies the type for [Networking\\_IpType](#) enum values.

```
typedef uint8_t Networking_IpType_Type;
```

# Applibs pwm.h

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/pwm.h>

The Applibs pwm header contains functions that interact with pulse-width modulators (PWM). PWM functions are thread-safe.

## Application manifest requirements

PWM functions are only permitted if the application has the Pwm capability in the [application manifest](#).

## Concepts and samples

- [Sample: PWM]

## Functions

FUNCTION	DESCRIPTION
<a href="#">PWM_Apply</a>	Sets the state of a PWM channel for a PWM controller.
<a href="#">PWM_Open</a>	Opens a PWM controller, and returns a file descriptor to use for subsequent calls.

## Structs

STRUCT	DESCRIPTION
<a href="#">PwmState</a>	The state of a PWM channel.

## Enums

ENUM	DESCRIPTION
<a href="#">PWM_Polarity</a>	The polarity of a PWM channel.

## Typedefs

TYPEDEF	DESCRIPTION
<a href="#">PWM_ControllerId</a>	The ID of a PWM controller.
<a href="#">PWM_ChannelId</a>	The ID of a PWM channel.

# PWM\_Apply Function

10/9/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/pwm.h>

Sets the state of a PWM channel for a PWM controller.

```
int PWM_Apply(int pwmFd, PWM_ChannelId pwmChannel, const PwmState *newState);
```

## Parameters

- `pwmFd` The file descriptor for the PWM controller. [PWM\\_Open](#) retrieves the file descriptor.
- `pwmChannel` The zero-based index that identifies the PWM channel to update. The maximum value permitted depends on the platform.
- `newState` A pointer to a [PwmState](#) struct that contains the new settings to apply. The pointer must remain valid for the duration of the call.

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- EBADF: the file descriptor is invalid.
- ENODEV: the `pwmChannel` parameter is invalid. Verify whether the channel is valid for the hardware platform.
- EINVAL: the `newState` parameter passed is invalid. Verify that the `newState` parameter isn't NULL, and contains valid settings.

## Returns

0 for success, or -1 for failure, in which case errno is set to the error value.

## Application manifest requirements

To access a PWM controller, your application must identify it in the Pwm field of the [application manifest](#).

# PWM\_Open Function

10/9/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/pwm.h>

Opens a PWM controller, and returns a file descriptor to use for subsequent calls.

```
int PWM_Open(PWM_ControllerId pwm);
```

## Parameters

- `pwm` The zero-based index of the PWM controller to access. The maximum value permitted depends on the platform.

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- EACCES: access to the PWM interface is not permitted because the *pwm* parameter is not listed in the Pwm field of the application manifest.

## Returns

The file descriptor for the PWM controller if it was opened successfully, or -1 for failure, in which case errno is set to the error value.

## Application manifest requirements

To access a PWM controller, your application must identify it in the Pwm field of the [application manifest](#).

# PWM\_Polarity Enum

10/9/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/pwm.h>

The polarity of a PWM channel.

```
typedef uint32_t PwmPolarity;

enum {
    PWM_Polarity_Normal,
    PWM_Polarity_Inversed,
};
```

VALUES	DESCRIPTIONS
PWM_Polarity_Normal	Normal polarity, which specifies a high signal during the duty cycle, and a low signal for the remainder of the period. For example, a duty cycle of 100 nanoseconds (ns) with a period of 300 ns results in a high signal for 100 ns and a low signal for 200 ns.
PWM_Polarity_Inversed	Inverse polarity, which specifies a low signal during the duty cycle, and a high signal for the remainder of the period. For example, a duty cycle of 100 ns with a period of 300 ns results in a high signal for 200 ns and a low signal for 100 ns.

# PwmState Struct

10/9/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/pwm.h>

The state of a PWM channel.

```
typedef struct PwmState {  
    unsigned int period_nsec;  
    unsigned int dutyCycle_nsec;  
    PwmPolarity polarity;  
    bool enabled;  
} PwmState;
```

## Members

### period\_nsec

The length of each period, in nanoseconds. This includes the total length of both the high and low states.

### dutyCycle\_nsec

The number of nanoseconds to spend in either a high or low state during a period. This value must be less than the period.

### polarity

The [PWM polarity](#) to apply. This specifies whether the *dutyCycle\_nsec* parameter applies to a high or a low state.

### enabled

True to enable the PWM functionality, false to disable it.

## Remarks

This struct is used by the [PWM\\_Apply function](#).

# PWM\_ChannelId Typedef

10/9/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/pwm.h>

The ID of a PWM channel.

```
typedef uint32_t PWM_ChannelId;
```

## Remarks

PWM controllers often have multiple channels on a single chip. An individual channel corresponds to a single pin or input on the device.

# PWM\_ControllerId Typedef

10/9/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/pwm.h>

The ID of a PWM controller.

```
typedef unsigned int PWM_ControllerId;
```

# Applibs rtc.h

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/rtc.h>

The Applibs rtc header contains functions that interact with the real-time clock (RTC).

## Application manifest requirements

RTC functions are only permitted if the application has the SystemTime capability in the [application manifest](#).

## Concepts and samples

- [Managing time and using the RTC](#)
- [Sample: System Time](#)

## Functions

FUNCTION	DESCRIPTION
<code>clock_systohc</code>	Synchronizes the real-time clock (RTC) with the current system time.

# clock\_systohc Function

1/7/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/rtc.h>

Synchronizes the real-time clock (RTC) with the current system time. The RTC only stores the time in UTC/GMT. Therefore, conversion from local time is necessary only if the local time zone isn't GMT.

```
int clock_systohc(void);
```

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- EACCES: the caller doesn't have the SystemTime capability.
- EBUSY: The RTC device was in use and couldn't be opened. The caller should try again periodically until it succeeds.

Any other errno may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

0 for success, or -1 for failure, in which case errno is set to the error value.

## Application manifest requirements

This function requires the SystemTime capability in the [application manifest](#).

## Concepts and samples

- [Managing time and using the RTC](#)
- [Sample: System Time](#)

# Applibs spi.h

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/spi.h>

The Applibs SPI header contains functions and types that access a Serial Peripheral Interface (SPI) on a device.

## NOTE

Define SPI\_STRUCTS\_VERSION to the appropriate version when using this header.

## Application manifest requirements

To access individual SPI interfaces, your application must identify them in the SpiMaster field of the [application manifest](#).

## Concepts and samples

- [Using SPI](#)
- [Sample: SPI](#)

## Functions

FUNCTION	DESCRIPTION
<a href="#">SpiMaster_InitConfig</a>	Initializes a <a href="#">SpiMaster_Config</a> struct with the default SPI master interface settings.
<a href="#">SpiMaster_InitTransfers</a>	Initializes an array of <a href="#">SpiMaster_Transfer</a> structs with the default SPI master transfer settings.
<a href="#">SpiMaster_Open</a>	Opens and configures an SPI master interface for exclusive use, and returns a file descriptor to use for subsequent calls.
<a href="#">SpiMaster_SetBitOrder</a>	Configures the order for transferring data bits on an SPI master interface.
<a href="#">SpiMaster_SetBusSpeed</a>	Sets the SPI bus speed for operations on an SPI master interface.
<a href="#">SpiMaster_SetMode</a>	Sets the communication mode for an SPI master interface.
<a href="#">SpiMaster_TransferSequential</a>	Performs a sequence of half-duplex read or write transfers using the SPI master interface.
<a href="#">SpiMaster_WriteThenRead</a>	Performs a sequence of a half-duplex writes immediately followed by a half-duplex read using the SPI master interface.

## Structs

<b>STRUCT</b>	<b>DESCRIPTION</b>
<a href="#">SPIMaster_Config</a>	The configuration options for opening an SPI master interface.
<a href="#">SPIMaster_Transfer</a>	The description of an SPI master transfer operation.

## Enums

<b>ENUM</b>	<b>DESCRIPTION</b>
<a href="#">SPI_BitOrder</a>	The possible SPI bit order values.
<a href="#">SPI_ChipSelectPolarity</a>	The possible chip select polarity values for an SPI interface.
<a href="#">SPI_Mode</a>	The possible communication mode values for an SPI interface.
<a href="#">SPI_Mode</a>	The possible <i>flags</i> values for a <a href="#">SPIMaster_Transfer</a> struct.

## Typdefs

<b>TYPEDEF</b>	<b>DESCRIPTION</b>
<a href="#">SPI_ChipSelectId</a>	An SPI chip select ID.
<a href="#">SPI_Interfaceld</a>	The ID for an SPI interface instance.

# SPI\_BitOrder Enum

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/spi.h>

The possible SPI bit order values.

```
typedef enum SPI_BitOrder {
    SPI_BitOrder_Invalid = 0x0,
    SPI_BitOrder_LsbFirst = 0x1,
    SPI_BitOrder_MsbFirst = 0x2
} SPI_BitOrder;
```

VALUES	DESCRIPTIONS
SPI_BitOrder_Invalid	An invalid bit order.
SPI_BitOrder_LsbFirst	The least-significant bit is sent first.
SPI_BitOrder_MsbFirst	The most-significant bit is sent first.

# SPI\_ChipSelectPolarity Enum

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/spi.h>

The possible chip select polarity values for an SPI interface.

```
typedef enum SPI_ChipSelectPolarity {
    SPI_ChipSelectPolarity_Invalid = 0x0,
    SPI_ChipSelectPolarity_ActiveLow = 0x1,
    SPI_ChipSelectPolarity_ActiveHigh = 0x2
} SPI_ChipSelectPolarity;
```

VALUES	DESCRIPTIONS
SPI_ChipSelectPolarity_Invalid	An invalid polarity.
SPI_ChipSelectPolarity_ActiveLow	Active low.
SPI_ChipSelectPolarity_ActiveHigh	Active high.

# SPI\_Mode Enum

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/spi.h>

The possible communication mode values for an SPI interface. The communication mode defines timings for device communication.

```
typedef enum SPI_Mode {  
    SPI_Mode_Invalid = 0x0,  
    SPI_Mode_0 = 0x1,  
    SPI_Mode_1 = 0x2,  
    SPI_Mode_2 = 0x3,  
    SPI_Mode_3 = 0x4  
} SPI_Mode;
```

VALUES	DESCRIPTIONS
SPI_Mode_Invalid	An invalid mode.
SPI_Mode_0	SPI mode 0: clock polarity (CPOL) = 0, clock phase (CPHA) = 0.
SPI_Mode_1	SPI mode 1: clock polarity (CPOL) = 0, clock phase (CPHA) = 1.
SPI_Mode_2	SPI mode 2: clock polarity (CPOL) = 1, clock phase (CPHA) = 0.
SPI_Mode_3	SPI mode 3: clock polarity (CPOL) = 1, clock phase (CPHA) = 1.

# SPI\_TransferFlags Enum

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/spi.h>

The possible *flags* values for a [SPIMaster\\_Transfer](#) struct.

```
typedef enum SPI_TransferFlags {
    SPI_TransferFlags_None = 0x0,
    SPI_TransferFlags_Read = 0x1,
    SPI_TransferFlags_Write = 0x2
} SPI_TransferFlags;
```

VALUES	DESCRIPTIONS
SPI_TransferFlags_None	No flags present.
SPI_TransferFlags_Read	Read from the subordinate device.
SPI_TransferFlags_Write	Write to the subordinate device.

# SPIMaster\_InitConfig Function

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/spi.h>

Initializes a [SPIMaster\\_Config](#) struct with the default SPI master interface settings.

```
static inline int SPIMaster_InitConfig(SPIMaster_Config *config);
```

## Parameters

- `config` A pointer to a SPIMaster\_Config struct that receives the default SPI master interface settings.

## Returns

0 for success, or -1 for failure, in which case errno will be set to the error value.

## Application manifest requirements

To access individual SPI interfaces, your application must identify them in the SpiMaster field of the [application manifest](#).

# SPIMaster\_InitTransfers Function

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/spi.h>

Initializes an array of [SPIMaster\\_Transfer](#) structs with the default SPI master transfer settings.

```
static inline int SPIMaster_InitTransfers(SPIMaster_Transfer *transfers, size_t transferCount);
```

## Parameters

- `transfers` A pointer to the array of [SPIMaster\\_Transfer](#) structs to initialize.
- `transferCount` The number of structs in the `transfers` array.

## Returns

0 for success, or -1 for failure, in which case errno will be set to the error value.

## Application manifest requirements

To access individual SPI interfaces, your application must identify them in the SpiMaster field of the [application manifest](#).

# SPIMaster\_Open Function

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/spi.h>

Opens and configures an SPI master interface for exclusive use, and returns a file descriptor to use for subsequent calls.

The interface is initialized with the default settings: *SPI\_Mode\_0*, *SPI\_BitOrder\_MsbFirst*. You can change these settings with SPI functions after the interface is opened.

```
static inline int SPIMaster_Open(SPI_InterfaceId interfaceId, SPI_ChipSelectId chipSelectId, const SPIMaster_Config *config);
```

## Parameters

- `interfaceId` The ID of the SPI master interface to open.
- `chipSelectId` The chip select ID to use with the SPI master interface.
- `config` The configuration for the SPI master interface. Before you call this function, you must call [SPIMaster\\_InitConfig](#) to initialize the [SPIMaster\\_Config](#) struct. You can change the settings after the struct is initialized. The `config` argument contains all settings that must be configured as part of opening the interface, and which may not be changed afterwards.

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- EACCES: access to this SPI interface is not permitted because the `interfaceId` parameter is not listed in the SpiMaster field of the application manifest.

Any other errno may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

The file descriptor of the SPI interface if it was opened successfully, or -1 for failure, in which case errno is set to the error value. You may use this descriptor with standard read(2) and write(2) functions to transact with the connected device. You may also use [SPIMaster\\_TransferSequential](#) to issue a sequence of transfers.

## Application manifest requirements

To access individual SPI interfaces, your application must identify them in the SpiMaster field of the [application manifest](#).

# SPIMaster\_SetBitOrder Function

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/spi.h>

Configures the order for transferring data bits on a SPI master interface.

```
int SPIMaster_SetBitOrder(int fd, SPI_BitOrder order);
```

## Parameters

- `fd` The file descriptor for the SPI master interface.
- `order` Specifies the desired [bit order](#) for data transfers.

## Returns

0 for success, or -1 for failure, in which case `errno` will be set to the error value.

## Application manifest requirements

To access individual SPI interfaces, your application must identify them in the `SpiMaster` field of the [application manifest](#).

# SPIMaster\_SetBusSpeed Function

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/spi.h>

Sets the SPI bus speed for operations on an SPI master interface.

```
int SPIMaster_SetBusSpeed(int fd, uint32_t speedInHz);
```

## Parameters

- `fd` The file descriptor for the SPI master interface.
- `speedInHz` The maximum speed for transfers on this interface, in Hz. Not all speeds are supported on all devices. The actual speed used by the interface may be lower than this value.

## Returns

0 for success, or -1 for failure, in which case `errno` will be set to the error value.

## Application manifest requirements

To access individual SPI interfaces, your application must identify them in the `SpiMaster` field of the [application manifest](#).

# SPIMaster\_SetMode Function

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/spi.h>

Sets the communication mode for an SPI master interface.

```
int SPIMaster_SetMode(int fd, SPI_Mode mode);
```

## Parameters

- `fd` The file descriptor for the SPI master interface.
- `mode` The [communication mode](#).

## Returns

0 for success, or -1 for failure, in which case errno will be set to the error value.

## Application manifest requirements

To access individual SPI interfaces, your application must identify them in the SpiMaster field of the [application manifest](#).

# SPIMaster\_TransferSequential Function

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/spi.h>

Performs a sequence of half-duplex read or write transfers using the SPI master interface. This function enables chip select once before the sequence, and disables it when it ends. This function does not support simultaneously reading and writing in a single transfer.

```
static inline ssize_t SPIMaster_TransferSequential(int fd, const SPIMaster_Transfer *transfers, size_t transferCount);
```

## Parameters

- `fd` The file descriptor for the SPI master interface.
- `transfers` An array of `SPIMaster_Transfer` structures that specify the transfer operations. You must call `SPIMaster_InitTransfers` to initialize the array with default settings before filling it.
- `transferCount` The number of transfer structures in the `transfers` array.

## Returns

The number of bytes transferred; or -1 for failure, in which case `errno` is set to the error value.

## Remarks

Each write transfer operation is limited to 4096 bytes. To transfer additional data, you need to use additional transfer operations.

## Application manifest requirements

To access individual SPI interfaces, your application must identify them in the `SpiMaster` field of the [application manifest](#).

# SPIMaster\_WriteThenRead Function

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/spi.h>

Performs a sequence of a half-duplex writes immediately followed by a half-duplex read using the SPI master interface. This function enables chip select once before the sequence, and disables it when it ends.

```
static inline ssize_t SPIMaster_WriteThenRead(int fd, const uint8_t *writeData, size_t lenWriteData, uint8_t *readData, size_t lenReadData);
```

## Parameters

- `fd` The file descriptor for the SPI master interface.
- `writeData` The data to write.
- `lenWriteData` The number of bytes to write.
- `readData` The output buffer that receives the data. This buffer must be large enough to receive up to `lenReadData` bytes.
- `lenReadData` The number of bytes to read.

## Returns

The number of bytes transferred; or -1 for failure, in which case errno is set to the error value.

## Application manifest requirements

To access individual SPI interfaces, your application must identify them in the SpiMaster field of the [application manifest](#).

# SPIMaster\_Config Struct

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/spi.h>

The configuration options for opening a SPI master interface. Call [SPIMaster\\_InitConfig](#) to initialize an instance.

## NOTE

This is an alias to a versioned structure. Define SPI\_STRUCTS\_VERSION to use this alias.

```
struct SPIMaster_Config {
    uint32_t z_magicAndVersion;
    SPI_ChipSelectPolarity csPolarity;
};
```

## Members

### **uint32\_t z\_magicAndVersion**

A unique identifier of the struct type and version. Do not edit.

### **SPI\_ChipSelectPolarity csPolarity**

The [chip select polarity](#).

# SPIMaster\_Transfer Struct

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/spi.h>

The description of an SPI master transfer operation. Call [SPIMaster\\_InitTransfer](#) to initialize an instance.

## NOTE

This is an alias to a versioned structure. Define SPI\_STRUCTS\_VERSION to use this alias.

```
struct SPIMaster_Transfer {
    uint32_t z__magicAndVersion;
    SPI_TransferFlags flags;
    const uint8_t *writeData;
    uint8_t *readData;
    size_t length;
};
```

## Members

### **uint32\_t z\_\_magicAndVersion**

A unique identifier of the struct type and version. Do not edit.

### **SPI\_TransferFlags flags**

The [transfer flags](#) for the operation.

### **const uint8\_t \*writeData**

The data for write operations. This value is ignored for half-duplex reads.

### **uint8\_t \*readData**

The buffer for read operations. This value is ignored for half-duplex writes.

### **size\_t length**

The number of bytes to transfer.

# SPI\_ChipSelectId Typedef

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/spi.h>

A SPI chip select ID.

```
typedef int SPI_ChipSelectId;
```

# SPI\_InterfaceId Typedef

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/spi.h>

The ID of an SPI interface instance.

```
typedef int SPI_InterfaceId;
```

# Applibs storage.h

1/7/2019 • 2 minutes to read

**Header:** #include <applibs/storage.h>

The Applibs storage header contains functions that interact with on-device storage, which includes read-only storage and mutable storage.

## Application manifest

Mutable storage functions are only permitted if the application has the **MutableStorage** capability in the [application manifest](#).

## Concepts and samples

Mutable storage:

- [Azure Sphere storage](#)
- [Sample: Mutable Storage](#)

On-device storage:

- [Sample: CurlEasyHttps](#)
- [Sample: CurlMultiHttps](#)

## Functions

FUNCTION	DESCRIPTION
<a href="#">Storage_DeleteMutableFile</a>	Deletes any existing data previously written to the mutable file when <a href="#">Storage_OpenMutableFile</a> is called.
<a href="#">Storage_GetAbsolutePathInImagePackage</a>	Gets a null-terminated string that contains the absolute path to a location within the image package of the running application, given a relative path inside the image package.
<a href="#">Storage_OpenFileInImagePackage</a>	Takes a relative path inside the image package and returns an opened read-only file descriptor.
<a href="#">Storage_OpenMutableFile</a>	Provides a file descriptor to a file placed in a location on disk where data will be persisted over device reboot.

# Storage\_DeleteMutableFile Function

5/30/2019 • 2 minutes to read

**Header:** #include <applibs/storage.h>

Deletes any existing data previously written to the mutable file when [Storage\\_OpenMutableFile](#) is called. This API assumes that all file descriptors on the mutable file have been closed.

```
int Storage_DeleteMutableFile(void);
```

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- EACCES: The application does not have the required application manifest capability (MutableStorage).
- EIO: An error occurred while trying to delete the data.
- ENOENT: There was no existing mutable storage file to delete.

Any other errno may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

0 for success, or -1 for failure, in which case errno is set to the error value.

## Application manifest requirements

The [application manifest](#) must include the MutableStorage capability.

## Concepts and samples

- [Azure Sphere storage](#)
- [Sample: Mutable Storage](#)

# Storage\_GetAbsolutePathInImagePackage Function

1/7/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/storage.h>

Gets a null-terminated string that contains the absolute path to a location within the image package of the running application, given a relative path inside the image package.

The location of the image package and the path returned by this function will not change while an application is running. However, the location may change between executions of an application.

This function allocates memory for the returned string, which should be freed by the caller using free().

This function does not check whether the path exists in the image package. The path cannot not begin with '/' or '.', and cannot contain '..'.

```
char *Storage_GetAbsolutePathInImagePackage(const char *relativePath);
```

## Parameters

- `relativePath` A relative path from the root of the image package. This value must not start with the directory separator character '/'.

## Errors

If an error is encountered, returns NULL and sets errno to the error value.

- EINVAL: `relativePath` begins with '/' or '.', or contains '..'.
- EFAULT: `relativePath` is NULL.
- ENOMEM: Out of memory.

Any other errno may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

The absolute path that includes the image package root, or -1 for failure, in which case errno is set to the error value.

## Concepts and samples

- [Sample: CurlEasyHttps](#)
- [Sample: CurlMultiHttps](#)

# Storage\_OpenFileInImagePackage Function

1/7/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/storage.h>

Takes a relative path inside the image package and returns an opened read-only file descriptor. The caller should close the returned file descriptor with the close function. This function should only be used to open regular files inside the image package.

```
int Storage_OpenFileInImagePackage(const char *relativePath);
```

## Parameters

- `relativePath` A relative path from the root of the image package. This value must not start with the directory separator character '/'.

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- EINVAL: `relativePath` begins with '/' or '.', or contains '..'.
- EFAULT: `relativePath` is NULL.
- ENOMEM: Out of memory.

Any other errno may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

The opened file descriptor, or -1 for failure, in which case errno is set to the error value.

## Concepts and samples

- [Sample: CurlEasyHttps](#)
- [Sample: CurlMultiHttps](#)

# Storage\_OpenMutableFile Function

5/30/2019 • 2 minutes to read

**Header:** #include <applibs/storage.h>

Provides a file descriptor to a file placed in a location on disk where data will be persisted over device reboot. This file will be retained over reboot as well as over application update.

The file will be created if it does not exist. Otherwise, this function will return a file descriptor to an existing object.

```
int Storage_OpenMutableFile(void);
```

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- EACCES: The application does not have the required application manifest capability (MutableStorage).
- EIO: An error occurred while trying to create the file.

Any other errno may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

A file descriptor to persistent, mutable storage. -1 on failure, in which case errno will be set to the error.

## Application manifest requirements

The [application manifest](#) must include the MutableStorage capability.

## Concepts and samples

- [Azure Sphere storage](#)
- [Sample: Mutable Storage](#)

# Applibs sysevent.h

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/sysevent.h>

The Applibs sysevent header contains functions and types for system event notifications. Applications can register for and unregister from update notifications. Apps can use these notifications to put themselves in a safe state before application shutdown, or can attempt to defer these events.

## Application manifest requirements

You can only call these functions if your application has the SystemEventNotifications capability configured in the [application manifest](#).

To call the [SysEvent\\_DeferEvent](#) or [SysEvent\\_ResumeEvent](#) functions, you must also configure the SoftwareUpdateDeferral capability in the application manifest.

## Functions

FUNCTION	DESCRIPTION
<a href="#">SysEvent_DeferEvent</a>	Attempts to defer an event for the specified duration.
<a href="#">SysEvent_EventsCallback</a>	This callback function is called when the status of a registered system event changes.
<a href="#">SysEvent_Info_GetUpdateData</a>	Retrieves application or OS update information.
<a href="#">SysEvent_RegisterForEventNotifications</a>	Registers the application with a set of events.
<a href="#">SysEvent_ResumeEvent</a>	Attempts to resume an event if it is deferred.
<a href="#">SysEvent_UnregisterForEventNotifications</a>	Unregisters from the system notifications that were registered by <a href="#">SysEvent_RegisterForEventNotifications</a> .

## Structs

STRUCT	DESCRIPTION
<a href="#">SysEvent_Info</a>	An opaque struct that contains information about a system event.
<a href="#">SysEvent_Info_UpdateData</a>	A struct that contains information about update events.

## Enums

ENUM	DESCRIPTION
<a href="#">SysEvent_Events</a>	Flags for system event types.

ENUM	DESCRIPTION
SysEvent_Status	The status of a system event.
SysEvent_UpdateType	The type of update to apply.

# SysEvent\_DeferEvent Function

10/9/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/sysevent.h>

Attempts to defer an event for the specified duration. This function should only be called when the event status is [SysEvent\\_Status\\_Pending](#).

```
int SysEvent_DeferEvent(SysEvent_Events event, uint32_t requested_defer_time_in_minutes);
```

## Parameters

- `event` The type of event to defer.
- `requested_defer_time_in_minutes` The duration of the deferral, in minutes. The maximum deferral time is 24 hours for OS updates and 167 hours for applications.

## Returns

0 for success, or -1 for failure, in which case errno is set to the error value.

## Application manifest requirements

The [application manifest](#) must include the SystemEventNotifications and SoftwareUpdateDeferral capabilities.

# SysEvent\_EventsCallback Function

10/9/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/sysevent.h>

This function is called when the status of a registered system event changes. Only one [SysEvent\\_Events](#) flag is set because there is only one callback call for each event status change.

```
typedef void SysEvent_EventsCallback(SysEvent_Events event, SysEvent_Status state,
                                     const SysEvent_Info *info, void *context);
```

## Parameters

- `event` The event.
- `state` The new status of the event.
- `info` Additional info about the status change. To retrieve the information, info needs to be passed to an event-specific function, such as [SysEvent\\_Info\\_GetUpdateData](#).
- `context` An optional context pointer that was passed to [SysEvent\\_RegisterForEventNotifications](#).

## Application manifest requirements

The [application manifest](#) must include the SystemEventNotifications capability.

# SysEvent\_Info\_GetUpdateData Function

10/9/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/sysevent.h>

Retrieves application or OS update information.

The info parameter must be retrieved from a [SysEvent\\_EventsCallback](#) call when the *event* parameter is set to [SysEvent\\_Events\\_Update](#).

```
int SysEvent_Info_GetUpdateData(const SysEvent_Info *info, SysEvent_Info_UpdateData *update_info);
```

## Parameters

- `info` A pointer to the [SysEvent\\_Info](#) struct that contains the system event information retrieved from the [SysEvent\\_EventsCallback](#) call.
- `update_info` A pointer to the [SysEvent\\_Info\\_UpdateData](#) structure that receives the software update information.

## Returns

0 for success, or -1 for failure, in which case errno is set to the error value.

## Application manifest requirements

The [application manifest](#) must include the SystemEventNotifications capability.

# SysEvent\_RegisterForEventNotifications Function

10/9/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/sysevent.h>

Registers the application with a set of events.

### NOTE

There must only be one active [EventRegistration](#) struct at a time for all system event notifications.

```
EventRegistration *SysEvent_RegisterForEventNotifications(EventLoop *el, SysEvent_Events eventBitmask  
SysEvent_EventsCallback callback_function, void *context);
```

## Parameters

- `el` The event loop to which the *EventRegistration* is registered.
- `eventBitmask` A bitmask that indicates the event types to listen to.
- `callbackFunction` A function handler that is called when the state of an event in `eventBitmask` changes.
- `context` An optional user context pointer that is passed to the events callback when an event occurs.

## Returns

A pointer to an [EventRegistration](#) struct, or nullptr, in which case errno is set to the error value.

## Remarks

The [EventRegistration](#) struct that is returned needs to be retained until it is passed to [SysEvent\\_UnregisterForEventNotifications](#).

## Application manifest requirements

The [application manifest](#) must include the SystemEventNotifications capability.

# SysEvent\_ResumeEvent Function

10/9/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/sysevent.h>

Attempts to resume an event if it is deferred.

This function can only be called if [SysEvent\\_DeferEvent](#) was called successfully, otherwise the call will fail. *SysEvent\_ResumeEvent* is not required after a *SysEvent\_DeferEvent* call. It should only be used if the update deferral is no longer needed.

```
int SysEvent_ResumeEvent(SysEvent_Events event);
```

## Parameters

- `event` The type of event to resume.

## Returns

0 for success, or -1 for failure, in which case errno is set to the error value.

The [application manifest](#) must include the SystemEventNotifications and SoftwareUpdateDeferral capabilities.

# SysEvent\_UnregisterForEventNotifications Function

10/9/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/sysevent.h>

Unregisters from the system notifications that were registered by [SysEvent\\_RegisterForEventNotifications](#).

The [EventRegistration](#) struct must already be registered, and each *EventRegistration* may only be unregistered once.

```
int SysEvent_UnregisterForEventNotifications(EventRegistration *reg);
```

## Parameters

- `reg` The *EventRegistration* struct to remove from the event loop.

## Returns

0 for success, or -1 for failure, in which case errno is set to the error value.

## Application manifest requirements

The [application manifest](#) must include the SystemEventNotifications capability.

# SysEvent\_Events Enum

10/9/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/sysevent.h>

Flags for system event types.

```
typedef uint32_t SysEvent_Events;
enum {
    SysEvent_Events_None = 0x00,
    SysEvent_Events_Update = 0x01,
    SysEvent_Events_Mask = SysEvent_Events_Update
};
```

## Members

### SysEvent\_Events\_None

No event given.

### SysEvent\_Events\_Update

An OS or application update.

### SysEvent\_Events\_Mask

A mask for all valid system events.

# SysEvent\_Status Enum

10/9/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/sysevent.h>

The status of a system event.

```
typedef uint32_t SysEvent_Status;
enum {
    SysEvent_Status_Invalid = 0,
    SysEvent_Status_Pending = 1,
    SysEvent_Status_Final = 2,
    SysEvent_Status_Rejected = 3,
    SysEvent_Status_Complete = 4
};
```

## Members

### SysEvent\_Status\_Invalid

The status was improperly initialized.

### SysEvent\_Status\_Pending

A 10-second warning that an event will occur, with the opportunity to defer the event.

### SysEvent\_Status\_Final

A 10-second warning that an event will occur, without the opportunity to defer the event.

### SysEvent\_Status\_Rejected

The previous pending event was deferred, and will occur at a later time.

### SysEvent\_Status\_Complete

The system event is complete.

For software update events, the complete status is only sent for application updates because OS updates require a device reboot to complete.

# SysEvent\_UpdateType Enum

10/9/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/sysevent.h>

The type of update to apply.

```
typedef uint32_t SysEvent_UpdateType;
enum {
    SysEvent_UpdateType_Invalid = 0,
    SysEvent_UpdateType_App = 1,
    SysEvent_UpdateType_System = 2
};
```

## Members

### **SysEvent\_UpdateType\_Invalid**

This enum was improperly initialized.

### **SysEvent\_UpdateType\_App**

An application update that restarts the updated application but doesn't reboot the device.

### **SysEvent\_UpdateType\_System**

An OS update that requires a device reboot.

# SysEvent\_Info Struct

10/9/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/sysevent.h>

An opaque struct that contains information about a system event.

The data in this struct can't be accessed directly. Instead it must be accessed by calling the system event function that is specific to the event type, such as [SysEvent\\_Info\\_GetUpdateData](#) for [SysEvent\\_Events\\_Update](#) events.

```
typedef struct SysEvent_Info SysEvent_Info;
```

# SysEvent\_Info\_UpdateData Struct

10/9/2019 • 2 minutes to read

## BETA feature

Header: #include <applibs/sysevent.h>

A struct that contains information about update events.

This struct is returned by [SysEvent\\_Info\\_GetUpdateData](#) after passing in a SysEvent\_Info struct that is returned by [SysEvent\\_EventsCallback](#).

This struct is only valid when the event type is [SysEvent\\_Events\\_Update](#).

```
typedef struct SysEvent_Info_UpdateData {
    unsigned int max_deferral_time_in_minutes;
    SysEvent_UpdateType update_type;
} SysEvent_Info_UpdateData;
```

## Members

### **max\_deferral\_time\_in\_minutes**

The maximum allowed deferral time, in minutes. The deferral time indicates how long an event listener can defer the event. This parameter is only defined if the status of the event is [SysEvent\\_Status\\_Pending](#).

### **update\_type**

The type of update event.

# Applibs uart.h

8/9/2019 • 2 minutes to read

**Header:** #include <applibs/uart.h>

The Applibs uart header contains functions and types that open and use a UART (Universal Asynchronous Receiver/Transmitter) on a device.

## NOTE

Define `UART_STRUCTS_VERSION` to the appropriate version when using this header.

## Application manifest requirements

To access individual UARTs, your application must identify them in the `Uart` field of the [application manifest](#).

## Concepts and samples

- [Use UART on Azure Sphere](#)
- [UART sample](#)
- [Sample: External MCU update - reference solution](#)

## Functions

FUNCTION	DESCRIPTION
<code>UART_InitConfig</code>	Initializes a <a href="#">UART config</a> struct with the default UART settings.
<code>UART_Open</code>	Opens and configures a UART, and returns a file descriptor to use for subsequent calls.

## Structs

STRUCT	DESCRIPTION
<code>UART_Config</code>	The configuration options for a UART. Call <code>UART_InitConfig</code> to initialize an instance.

## Enums

ENUM	DESCRIPTION
<code>UART_BlockingMode</code>	The valid values for UART blocking or non-blocking modes.
<code>UART_DataBits</code>	The valid values for UART data bits.
<code>UART_FlowControl</code>	The valid values for flow control settings.

ENUM	DESCRIPTION
UART_Parity	The valid values for UART parity.
UART_StopBits	The valid values for UART stop bits.

## Typedefs

TYPEDEF	DESCRIPTION
UART_BaudRate_Type	Specifies the type of the baudRate value for the <a href="#">UART_Config</a> struct.
UART_BlockingMode_Type	Specifies the type of the blockingMode value for the <a href="#">UART_Config</a> struct.
UART_DataBits_Type	Specifies the type of the dataBits value for the <a href="#">UART_Config</a> struct.
UART_FlowControl_Type	Specifies the type of the flowControl value for the <a href="#">UART_Config</a> struct.
UART_Id	A UART ID, which specifies a UART peripheral instance.
UART_Parity_Type	Specifies the type of the parity value for the <a href="#">UART_Config</a> struct.
UART_StopBits_Type	Specifies the type of the stopBits value for the <a href="#">UART_Config</a> struct.

# UART\_InitConfig Function

2/14/2019 • 2 minutes to read

**Header:** #include <applibs/uart.h>

Initializes a [UART config](#) struct with the default UART settings. The default UART settings are 8 for dataBits, 0 (none) for parity, and 1 for stopBits.

```
void UART_InitConfig(UART_Config * uartConfig);
```

## Parameters

- `uartConfig` A pointer to a [UART\\_Config](#) object that returns the default UART settings.

## Application manifest requirements

To access individual UARTs, your application must identify them in the Uart field of the [application manifest](#).

# UART\_Open Function

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/uart.h>

Opens and configures a UART, and returns a file descriptor to use for subsequent calls.

```
int UART_Open(UART_Id uartId, const UART_Config * uartConfig);
```

## Parameters

- `uartId` The ID of the UART to open.
- `uartConfig` A pointer to a `UART_Config` struct that defines the configuration of the UART. Call `UART_InitConfig` to get a `UART_Config` with default settings.

## Errors

If an error is encountered, returns -1 and sets `errno` to the error value.

- `EACCES`: access to `UART_Id` is not permitted as the `uartId` is not listed in the `Uart` field of the application manifest.
- `ENODEV`: the `uartId` is invalid.
- `EINVAL`: the `uartConfig` represents an invalid configuration.
- `EBUSY`: the `uartId` is already open.
- `EFAULT`: the `uartConfig` is NULL.

Any other `errno` may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

The file descriptor of the UART if it was opened successfully, or -1 for failure, in which case `errno` is set to the error value.

## Application manifest requirements

To access individual UARTs, your application must identify them in the `Uart` field of the [application manifest](#).

# UART\_Config Struct

1/7/2019 • 2 minutes to read

**Header:** #include <applibs/uart.h>

The configuration options for a UART. Call [UART\\_InitConfig](#) to initialize an instance.

## NOTE

this is an alias to a versioned structure. Define `UART_STRUCTS_VERSION` to use this alias.

```
struct UART_Config {  
    uint32_t z__magicAndVersion;  
    UART_BaudRate_Type baudRate;  
    UART_BlockingMode_Type blockingMode;  
    UART_DataBits_Type dataBits;  
    UART_Parity_Type parity;  
    UART_StopBits_Type stopBits;  
    UART_FlowControl_Type flowControl;  
};
```

## Members

### **uint32\_t z\_\_magicAndVersion**

A unique identifier of the struct type and version. Do not edit.

### **UART\_BaudRate\_Type baudRate**

The baud rate of the UART.

### **UART\_BlockingMode\_Type blockingMode**

The blocking mode setting for the UART.

### **UART\_DataBits\_Type dataBits**

The data bits setting for the UART.

### **UART\_Parity\_Type parity**

The parity setting for the UART.

### **UART\_StopBits\_Type stopBits**

The stop bits setting for the UART.

### **UART\_FlowControl\_Type flowControl**

The flow control setting for the UART.

# UART\_BlockingMode Enum

1/7/2019 • 2 minutes to read

**Header:** #include <applibs/uart.h>

The valid values for UART blocking or non-blocking modes.

```
typedef enum {
    UART_BlockingMode_NonBlocking = 0,
} UART_BlockingMode;
```

VALUES	DESCRIPTIONS
UART_BlockingMode_NonBlocking	Reads and writes to the file handle are non-blocking and return an error if the call blocks. Reads may return less data than requested.

# UART\_DataBits Enum

2/14/2019 • 2 minutes to read

**Header:** #include <applibs/uart.h>

The valid values for UART data bits.

```
typedef enum {
    UART_DataBits_Five = 5,
    UART_DataBits_Six = 6,
    UART_DataBits_Seven = 7,
    UART_DataBits_Eight = 8
} UART_DataBits;
```

VALUES	DESCRIPTIONS
UART_DataBits_Five	Five data bits.
UART_DataBits_Six	Six data bits.
UART_DataBits_Seven	Seven data bits.
UART_DataBits_Eight	Eight data bits.

# UART\_FlowControl Enum

1/7/2019 • 2 minutes to read

**Header:** #include <applibs/uart.h>

The valid values for flow control settings.

```
typedef enum {
    UART_FlowControl_None = 0,
    UART_FlowControl_RTSCTS = 1,
    UART_FlowControl_XONXOFF = 2
} UART_FlowControl;
```

VALUES	DESCRIPTIONS
UART_FlowControl_None	No flow control.
UART_FlowControl_RTSCTS	Enable RTS/CTS hardware flow control.
UART_FlowControl_XONXOFF	Enable XON/XOFF software flow control.

# UART\_Parity Enum

2/14/2019 • 2 minutes to read

**Header:** #include <applibs/uart.h>

The valid values for UART parity.

```
typedef enum {
    UART_Parity_None = 0,
    UART_Parity_Even = 1,
    UART_Parity_Odd = 2
} UART_Parity;
```

VALUES	DESCRIPTIONS
UART_Parity_None	No parity bit.
UART_Parity_Even	Even parity bit.
UART_Parity_Odd	Odd parity bit.

# UART\_StopBits Enum

2/14/2019 • 2 minutes to read

**Header:** #include <applibs/uart.h>

The valid values for UART stop bits.

```
typedef enum {
    UART_StopBits_One = 1,
    UART_StopBits_Two = 2
} UART_StopBits;
```

VALUES	DESCRIPTIONS
UART_StopBits_One	One stop bit.
UART_StopBits_Two	Two stop bits.

# UART\_BaudRate\_Type Typedef

1/7/2019 • 2 minutes to read

**Header:** #include <applibs/uart.h>

Specifies the type of the baudRate value for the [UART\\_Config](#) struct.

```
typedef uint32_t UART_BaudRate_Type;
```

# UART\_BlockingMode\_Type Typedef

1/7/2019 • 2 minutes to read

**Header:** #include <applibs/uart.h>

Specifies the type of the blockingMode value for the [UART\\_Config](#) struct.

```
typedef uint8_t UART_BlockingMode_Type;
```

# UART\_DataBits\_Type Typedef

1/7/2019 • 2 minutes to read

**Header:** #include <applibs/uart.h>

Specifies the type of the dataBits value for the [UART\\_Config](#) struct.

```
typedef uint8_t UART_DataBits_Type;
```

# UART\_FlowControl\_Type Typedef

1/7/2019 • 2 minutes to read

**Header:** #include <applibs/uart.h>

Specifies the type of the flowControl value for the [UART\\_Config](#) struct.

```
typedef uint8_t UART_FlowControl_Type;
```

# UART\_Id Typedef

1/7/2019 • 2 minutes to read

**Header:** #include <applibs/uart.h>

A UART ID, which specifies a UART peripheral instance.

```
typedef int UART_Id;
```

# UART\_Parity\_Type Typedef

1/7/2019 • 2 minutes to read

**Header:** #include <applibs/uart.h>

Specifies the type of the parity value for the [UART\\_Config](#) struct.

```
typedef uint8_t UART_Parity_Type;
```

# UART\_StopBits\_Type Typedef

1/7/2019 • 2 minutes to read

**Header:** #include <applibs/uart.h>

Specifies the type of the stopBits value for the [UART\\_Config](#) struct.

```
typedef uint8_t UART_StopBits_Type;
```

# Applibs wificonfig.h

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/wificonfig.h>

The Applibs wificonfig header contains functions and types that manage Wi-Fi network configurations on a device.

## NOTE

To use these functions, define WIFICONFIG\_STRUCTS\_VERSION with the structure version you're using. Currently, the only valid version is 1 (define WIFICONFIG\_STRUCTS\_VERSION 1). Thereafter, you can use the friendly names of the WifiConfig\_ structures, which start with WifiConfig\_.

## Application manifest requirements

You can only call these functions if your application has the WifiConfig capability in the application manifest.

## Thread safety

These functions are not thread safe.

## Concepts and samples

- [Sample: System Time](#)
- [BLE-based Wi-Fi setup and device control reference solution](#)

## Functions

FUNCTION	DESCRIPTION
<a href="#">WifiConfig_AddNetwork</a>	Adds a Wi-Fi network to the device and returns the ID of the network.
<a href="#">WifiConfig_ForgetAllNetworks</a>	Removes all stored Wi-Fi networks from the device. Disconnects the device from any connected network.
<a href="#">WifiConfig_ForgetNetwork</a>	<b>Obsolete:</b> This function is obsolete. Use <a href="#">WifiConfig_ForgetNetworkById</a> instead.
<a href="#">WifiConfig_ForgetNetworkById</a>	Removes a Wi-Fi network from the device.
<a href="#">WifiConfig_GetCurrentNetwork</a>	Gets a Wi-Fi network that is connected to the device.
<a href="#">WifiConfig_GetScannedNetworks</a>	Gets the Wi-Fi networks found by the last scan operation.
<a href="#">WifiConfig_GetStoredNetworkCount</a>	Gets the number of stored Wi-Fi networks on the device.
<a href="#">WifiConfig_GetStoredNetworks</a>	Retrieves all stored Wi-Fi networks on the device.

FUNCTION	DESCRIPTION
<a href="#">WifiConfig_PersistConfig</a>	Writes the current network configuration to nonvolatile storage so it persists over a device reboot.
<a href="#">WifiConfig_SetNetworkEnabled</a>	Enables or disables a Wi-Fi network configuration.
<a href="#">WifiConfig_SetPSK</a>	Sets the pre-shared key (PSK) for a Wi-Fi network.
<a href="#">WifiConfig_SetSecurityType</a>	Sets the security type for a Wi-Fi network.
<a href="#">WifiConfig_SetSsid</a>	Sets the SSID for a Wi-Fi network.
<a href="#">WifiConfig_SetTargetedScanEnabled</a>	Enables or disables targeted scanning for a network.
<a href="#">WifiConfig_StoreOpenNetwork</a>	<b>Obsolete:</b> This function is obsolete. Use <a href="#">WifiConfig_AddNetwork</a> instead.
<a href="#">WifiConfig_StoreWpa2Network</a>	<b>Obsolete:</b> This function is obsolete. Use <a href="#">WifiConfig_AddNetwork</a> instead.
<a href="#">WifiConfig_TriggerScanAndGetScannedNetworkCount</a>	Starts a scan to find all available Wi-Fi networks.

## Structs

STRUCT	DESCRIPTION
<a href="#">WifiConfig_ConnectedNetwork</a>	The properties of a connected Wi-Fi network, which represent a 802.11 Basic Service Set (BSS).
<a href="#">WifiConfig_ScannedNetwork</a>	The properties of a scanned Wi-Fi network, which represent a 802.11 BSS.
<a href="#">WifiConfig_StoredNetwork</a>	The properties of a stored Wi-Fi network, which represent a 802.11 BSS.

## Enums

ENUM	DESCRIPTION
<a href="#">WifiConfig_Security</a>	The security key setting for a Wi-Fi network.

## TypeDefs

TYPEDEF	DESCRIPTION
<a href="#">WifiConfig_Security_Type</a>	Specifies the type of the security settings values for the <a href="#">WifiConfig_Security</a> enum.

# WifiConfig\_AddNetwork Function

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/wificonfig.h>

Adds a Wi-Fi network to the device and returns the ID of the network.

```
int WifiConfig_AddNetwork(void);
```

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- EACCES: the application manifest doesn't include the WifiConfig capability.
- EAGAIN: the Wi-Fi device isn't ready.
- ENETDOWN: the Wi-Fi network interface is unavailable.

Any other errno may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

The ID of the new network, or -1 for failure, in which case errno is set to the error value. The network ID is a positive value.

## Remarks

The network ID is passed to *WifiConfig\_Set* functions, such as [WifiConfig\\_SetSSID](#).

The new network isn't configured and can be configured with the *WifiConfig\_Set* functions. Changes to the network configuration are effective immediately but are lost when the device reboots unless the [WifiConfig\\_PersistConfig](#) function is called to save the configuration to nonvolatile storage.

## Application manifest requirements

The [application manifest](#) must include the WifiConfig capability.

# WifiConfig\_ForgetAllNetworks Function

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/wificonfig.h>

Removes all stored Wi-Fi networks from the device. Disconnects the device from any connected network. This function is not thread safe.

The removal persists across device reboots.

```
int WifiConfig_ForgetAllNetworks(void);
```

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- EACCES: the application manifest does not include the WifiConfig capability.
- EAGAIN: the Wi-Fi device isn't ready yet.
- ENETDOWN: the Wi-Fi network interface is unavailable.

Any other errno may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

0 for success, or -1 for failure, in which case errno is set to the error value.

## Application manifest requirements

The [application manifest](#) must include the WifiConfig capability.

# WifiConfig\_ForgetNetwork Function

10/9/2019 • 2 minutes to read

Header: #include <applibs/wificonfig.h>

## IMPORTANT

This function is obsolete. Use [WifiConfig\\_ForgetNetworkById](#) instead.

Removes a Wi-Fi network from the device. Disconnects the device from the network if it's currently connected. This function is not thread safe.

```
int WifiConfig_ForgetNetwork(const WifiConfig_StoredNetwork * storedNetwork);
```

## Parameters

- `storedNetwork` Pointer to a [WifiConfig\\_StoredNetwork](#) struct that describes the stored Wi-Fi network to remove.

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- EACCES: the application manifest does not include the WifiConfig capability.
- EFAULT: the `ssid` parameter is NULL.
- ENOENT: the `storedNetwork` parameter doesn't match any of the stored networks.
- EINVAL: the `storedNetwork` parameter or its struct version is invalid.
- EAGAIN: the Wi-Fi device isn't ready yet.
- ENETDOWN: the Wi-Fi network interface is unavailable.
- ENOSPC: there are too many Wi-Fi networks for the configuration to persist; remove one and try again.

Any other errno may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

0 for success, or -1 for failure, in which case errno is set to the error value.

## Application manifest requirements

The [application manifest](#) must include the WifiConfig capability.

# WifiConfig\_ForgetNetworkById Function

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/wificonfig.h>

Removes a Wi-Fi network from the device. Disconnects the device from the network if it's currently connected.

The setting is effective immediately but won't persist across device reboots unless the [WifiConfig\\_PersistConfig](#) function is called after this function.

```
int WifiConfig_ForgetNetworkById(int networkId);
```

## Parameters

- `networkId` The ID of the network to remove.

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- EACCES: the application manifest doesn't include the WifiConfig capability.
- EINVAL: the `networkId` parameter is invalid.
- ENOENT: the `networkId` parameter doesn't match any of the IDs of the [stored networks](#).
- EAGAIN: the Wi-Fi device isn't ready.
- ENETDOWN: the Wi-Fi network interface is unavailable.

Any other errno may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

0 for success, or -1 for failure, in which case errno is set to the error value.

## Application manifest requirements

The [application manifest](#) must include the WifiConfig capability.

# WifiConfig\_GetCurrentNetwork Function

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/wificonfig.h>

Gets a Wi-Fi network that is connected to the device. This function is not thread safe.

```
int WifiConfig_GetCurrentNetwork(WifiConfig_ConnectedNetwork * connectedNetwork);
```

## Parameters

- `connectedNetwork` A pointer to a [WifiConfig\\_ConnectedNetwork](#) struct that returns the connected Wi-Fi network.

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- EACCES: the application manifest does not include the WifiConfig capability.
- EFAULT: the *ssid* is NULL.
- ENOTCONN: the device is not currently connected to any network.
- EINVAL: the *storedNetwork* or its struct version is invalid.
- EAGAIN: the Wi-Fi device isn't ready yet.
- ENETDOWN: the Wi-Fi network interface is unavailable.

Any other errno may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

0 for success, or -1 for failure, in which case errno is set to the error value.

## Application manifest requirements

The [application manifest](#) must include the WifiConfig capability.

# WifiConfig\_GetScannedNetworks Function

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/wificonfig.h>

Gets the Wi-Fi networks found by the last scan operation. This function is not thread safe.

- If `scannedNetworkArray` is too small to hold all the networks, this function fills all the elements and returns the number of array elements.
- If the WiFiConfig capability is not present, the function returns an empty array.

```
ssize_t WifiConfig_GetScannedNetworks(WifiConfig_ScannedNetwork * scannedNetworkArray, size_t  
scannedNetworkArrayCount);
```

## Parameters

- `scannedNetworkArray` A pointer to an array that returns the retrieved Wi-Fi networks.
- `scannedNetworkArrayCount` The number of elements `scannedNetworkArray` can hold. The array should have one element for each Wi-Fi network found by the last scan operation.

## Errors

If an error is encountered, returns -1 and sets `errno` to the error value.

- `EACCES`: the application manifest does not include the WiFiConfig capability.
- `EFAULT`: the `scannedNetworkArray` parameter is NULL.
- `ERANGE`: the `scannedNetworkArrayCount` parameter is 0.
- `EINVAL`: the `scannedNetworkArray` parameter or its struct version is invalid.
- `EAGAIN`: the Wi-Fi device isn't ready yet.

Any other `errno` may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

The number of [WifiConfig\\_ScannedNetwork](#) elements returned by `scannedNetworkArray`, or -1 for failure, in which case `errno` is set to the error value.

## Application manifest requirements

The [application manifest](#) must include the WiFiConfig capability.

# WifiConfig\_GetStoredNetworkCount Function

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/wificonfig.h>

Gets the number of stored Wi-Fi networks on the device. This function is not thread safe.

## NOTE

Before you call [WifiConfig\\_GetStoredNetworks](#), you must call `WifiConfig_GetStoredNetworkCount` and use the result as the array size for the [WifiConfig\\_StoredNetwork](#) array that is passed to `WifiConfig_GetStoredNetworks`.

```
ssize_t WifiConfig_GetStoredNetworkCount(void);
```

## Errors

If an error is encountered, returns -1 and sets `errno` to the error value.

- `EACCES`: the application manifest does not include the `WifiConfig` capability.
- `EAGAIN`: the Wi-Fi device isn't ready yet.
- `ENETDOWN`: the Wi-Fi network interface is unavailable.

Any other `errno` may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

The number of Wi-Fi networks stored on the device, or -1 for failure, in which case `errno` is set to the error value.

## Application manifest requirements

The [application manifest](#) must include the `WifiConfig` capability.

# WifiConfig\_GetStoredNetworks Function

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/wificonfig.h>

Retrieves all stored Wi-Fi networks on the device. This function is not thread safe.

## NOTE

Before you call WifiConfig\_GetStoredNetworks, you must call [WifiConfig\\_GetStoredNetworkCount](#) and use the result as the array size for the [WifiConfig\\_StoredNetwork](#) array that is passed in as the *storedNetworkArray* parameter.

- If `storedNetworkArray` is too small to hold all the stored Wi-Fi networks, this function fills the array and returns the number of array elements.
- If the WiFiConfig capability is not present, the function returns an empty array.

```
ssize_t WifiConfig_GetStoredNetworks(WifiConfig_StoredNetwork * storedNetworkArray, size_t  
                                     storedNetworkArrayCount);
```

## Parameters

- `storedNetworkArray` A pointer to an array that returns the stored Wi-Fi networks.
- `storedNetworkArrayCount` The number of elements `storedNetworkArray` can hold. The array should have one element for each stored Wi-Fi network.

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- EACCES: the application manifest does not include the WiFiConfig capability.
- EFAULT: the `storedNetworkArray` parameter is NULL.
- ERANGE: the `storedNetworkArrayCount` parameter is 0.
- EINVAL: the `storedNetworkArray` parameter or its struct version is invalid.
- EAGAIN: the Wi-Fi device isn't ready yet.

Any other errno may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

The number of elements in the [WifiConfig\\_StoredNetwork](#) array, or -1 for failure, in which case errno is set to the error value.

## Application manifest requirements

The [application manifest](#) must include the WiFiConfig capability.

# WifiConfig\_PersistConfig Function

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/wificonfig.h>

Writes the current network configuration to nonvolatile storage so it persists over a device reboot.

```
static int WifiConfig_PersistConfig(void);
```

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- EACCES: the application manifest doesn't include the WifiConfig capability.
- EAGAIN: the Wi-Fi device isn't ready.
- ENETDOWN: the Wi-Fi network interface is unavailable.
- ENOSPC: there are too many Wi-Fi networks for the configuration to persist; remove one and try again.

Any other errno may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

0 for success, or -1 for failure, in which case errno is set to the error value.

## Application manifest requirements

The [application manifest](#) must include the WifiConfig capability.

# WifiConfig\_SetNetworkEnabled Function

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/wificonfig.h>

Enables or disables a Wi-Fi network configuration.

The setting is effective immediately but won't persist across device reboots unless the [WifiConfig\\_PersistConfig](#) function is called after this function.

```
static int WifiConfig_SetNetworkEnabled(int networkId, bool enabled);
```

## Parameters

- `networkId` The ID of the network to configure. [WifiConfig\\_AddNetwork](#) returns the network ID.
- `enabled` *true* to enable the network; otherwise, *false*.

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- EACCES: the application manifest doesn't include the WifiConfig capability.
- EAGAIN: the Wi-Fi device isn't ready.
- ENETDOWN: the Wi-Fi network interface is unavailable.
- EINVAL: the *networkId* parameter is invalid.
- ENODEV: the *networkId* parameter doesn't match any of the IDs of the [stored networks](#).

Any other errno may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

0 for success, or -1 for failure, in which case errno is set to the error value.

## Application manifest requirements

The [application manifest](#) must include the WifiConfig capability.

# WifiConfig\_SetPSK Function

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/wificonfig.h>

Sets the pre-shared key (PSK) for a Wi-Fi network. The PSK is used for networks that are configured with the *WifiConfig\_Security\_Wpa2\_Psk* security type.

The setting is effective immediately but won't persist across device reboots unless the [WifiConfig\\_PersistConfig](#) function is called after this function.

```
static int WifiConfig_SetPSK(int networkId, const char *psk, size_t pskLength);
```

## Parameters

- `networkId` The ID of the network to configure. [WifiConfig\\_AddNetwork](#) returns the network ID.
- `psk` A pointer to the buffer that contains the PSK for the network.
- `pskLength` The length of the PSK for the network. This parameter must be less than or equals to *WIFI\_CONFIG\_WPA2\_KEY\_MAX\_BUFFER\_SIZE*.

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- `EACCES`: the application manifest doesn't include the `WifiConfig` capability.
- `EFAULT`: the `psk` parameter is NULL.
- `ERANGE`: the `pskLength` parameter is greater than *WIFI\_CONFIG\_WPA2\_KEY\_MAX\_BUFFER\_SIZE*.
- `EAGAIN`: the Wi-Fi device isn't ready.
- `ENETDOWN`: the Wi-Fi network interface is unavailable.
- `EINVAL`: the `networkId` parameter is invalid.
- `ENODEV`: the `networkId` parameter doesn't match any of the IDs of the [stored networks](#).

Any other errno may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

0 for success, or -1 for failure, in which case errno is set to the error value.

## Application manifest requirements

The [application manifest](#) must include the `WifiConfig` capability.

# WifiConfig\_SetSecurityType Function

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/wificonfig.h>

Sets the security type for a Wi-Fi network.

The setting is effective immediately but won't persist across device reboots unless the [WifiConfig\\_PersistConfig](#) function is called after this function.

```
static int WifiConfig_SetSecurityType(int networkId, WifiConfig_Security_Type securityType);
```

## Parameters

- `networkId` The ID of the network to configure. [WifiConfig\\_AddNetwork](#) returns the network ID.
- `securityType` The security type for the specified network.

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- EACCES: the application manifest doesn't include the WifiConfig capability.
- EINVAL: the `securityType` parameter is invalid.
- EAGAIN: the Wi-Fi device isn't ready.
- ENETDOWN: the Wi-Fi network interface is unavailable.
- EINVAL: the `networkId` parameter is invalid.
- ENOENT: the `networkId` parameter doesn't match any of the IDs of the [stored networks](#).

Any other errno may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

0 for success, or -1 for failure, in which case errno is set to the error value.

## Application manifest requirements

The [application manifest](#) must include the WifiConfig capability.

# WifiConfig\_SetSSID Function

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/wificonfig.h>

Sets the SSID for a Wi-Fi network.

The setting is effective immediately but won't persist across device reboots unless the [WifiConfig\\_PersistConfig](#) function is called after this function.

```
static int WifiConfig_SetSSID(int networkId, const uint8_t *ssid, size_t ssidLength);
```

## Parameters

- `networkId` The ID of the network to configure. [WifiConfig\\_AddNetwork](#) returns the network ID.
- `ssid` A pointer to a byte array that contains the new SSID. The character encoding is not specified.
- `ssidLength` The number of bytes in the `ssid` parameter. Must be less than or equal to `WIFICONFIG_SSID_MAX_LENGTH`.

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- `EACCES`: the application manifest doesn't include the `WifiConfig` capability.
- `EFAULT`: the `ssid` parameter is NULL.
- `ERANGE`: the `ssidLength` parameter is greater than `WIFICONFIG_SSID_MAX_LENGTH`.
- `EAGAIN`: the Wi-Fi device isn't ready.
- `ENETDOWN`: the Wi-Fi network interface is unavailable.
- `EINVAL`: the `networkId` parameter is invalid.
- `ENODEV`: the `networkId` parameter doesn't match any of the IDs of the [stored networks](#).

Any other errno may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

0 for success, or -1 for failure, in which case errno is set to the error value.

## Application manifest requirements

The [application manifest](#) must include the `WifiConfig` capability.

# WifiConfig\_SetTargetedScanEnabled Function

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/wificonfig.h>

Enables or disables targeted scanning for a network. Targeted scanning is disabled by default.

Targeted scanning is used to connect to access points that aren't broadcasting their SSID, or are in a noisy environment.

The setting is effective immediately but won't persist across device reboots unless the [WifiConfig\\_PersistConfig](#) function is called after this function.

## IMPORTANT

Targeted scanning causes the device to transmit probe requests that may reveal the SSID of the network to other devices. This should only be used in controlled environments, or on networks where this is an acceptable risk.

```
static int WifiConfig_SetTargetedScanEnabled(int networkId, bool enabled);
```

## Parameters

- `networkId` The ID of the network to configure. [WifiConfig\\_AddNetwork](#) returns the network ID.
- `enabled` `true` to enable targeted scanning on the network; otherwise, `false`.

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- EACCES: the application manifest doesn't include the WifiConfig capability.
- EAGAIN: the Wi-Fi device isn't ready.
- ENETDOWN: the Wi-Fi network interface is unavailable.
- EINVAL: the `networkId` parameter is invalid.
- ENOENT: the `networkId` parameter doesn't match any of the stored networks.

Any other errno may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

0 for success, or -1 for failure, in which case errno is set to the error value.

## Application manifest requirements

The [application manifest](#) must include the WifiConfig capability.

# WifiConfig\_StoreOpenNetwork Function

10/9/2019 • 2 minutes to read

Header: #include <applibs/wificonfig.h>

## IMPORTANT

This function is obsolete. Use [WifiConfig\\_AddNetwork](#) instead.

Stores an open Wi-Fi network without a key. This function is not thread safe.

This function will fail if an identical network is already stored on the device without a key. See the error section (EEXIST). However, if a stored network includes a key along with the same SSID, this function will succeed and store the network.

```
int WifiConfig_StoreOpenNetwork(const uint8_t * ssid, size_t ssidLength);
```

## Parameters

- `ssid` A pointer to an SSID byte array with unspecified character encoding that identifies the Wi-Fi network.
- `ssidLength` The number of bytes in the SSID of the Wi-Fi network.

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- EACCES: the application manifest does not include the WifiConfig capability.
- EEXIST: a stored Wi-Fi network that has the same SSID and no key already exists.
- EFAULT: the `ssid` is NULL.
- ERANGE: the `ssidLength` is 0 or greater than WIFICONFIG\_SSID\_MAX\_LENGTH.
- EAGAIN: the Wi-Fi device isn't ready yet.
- ENETDOWN: the Wi-Fi network interface is unavailable.
- ENOSPC: there are too many Wi-Fi networks for the configuration to persist; remove one and try again.

Any other errno may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

0 for success, or -1 for failure, in which case errno is set to the error value.

## Application manifest requirements

The [application manifest](#) must include the WifiConfig capability.

# WifiConfig\_StoreWpa2Network Function

10/9/2019 • 2 minutes to read

Header: #include <applibs/wificonfig.h>

## IMPORTANT

This function is obsolete. Use [WifiConfig\\_AddNetwork](#) instead.

Stores a WPA2 Wi-Fi network that uses a pre-shared key. This function is not thread safe.

## NOTE

This function will fail if a network with the same SSID and pre-shared key is already stored. See the error section (EEXIST).

```
int WifiConfig_StoreWpa2Network(const uint8_t * ssid, size_t ssidLength, const char * psk, size_t pskLength);
```

## Parameters

- `ssid` A pointer to an SSID byte array with unspecified character encoding that identifies the Wi-Fi network.
- `ssidLength` The number of bytes in the SSID of the Wi-Fi network.
- `psk` A pointer to a buffer that contains the pre-shared key for the Wi-Fi network.
- `pskLength` The length of the pre-shared key for the Wi-Fi network.

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- EACCES: the application manifest does not include the WifiConfig capability.
- EEXIST: a stored Wi-Fi network already exists that has the same SSID and uses WPA2.
- EFAULT: the `ssid` or `psk` is NULL.
- ERANGE: the `ssidLength` or `pskLength` parameter is 0 or greater than `WIFICONFIG_SSID_MAX_LENGTH` and `WIFICONFIG_WPA2_KEY_MAX_BUFFER_SIZE`.
- EAGAIN: the Wi-Fi device isn't ready yet.
- ENETDOWN: the Wi-Fi network interface is unavailable.
- ENOSPC: there are too many Wi-Fi networks for the configuration to persist; remove one and try again.

Any other errno may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

0 for success, or -1 for failure, in which case errno is set to the error value.

## Application manifest requirements

The [application manifest](#) must include the WifiConfig capability.

# WifiConfig\_TriggerScanAndGetScannedNetworkCount Function

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/wificonfig.h>

Starts a scan to find all available Wi-Fi networks.

- This function is not thread safe.
- This is a blocking call.

```
ssize_t WifiConfig_TriggerScanAndGetScannedNetworkCount(void);
```

## Errors

If an error is encountered, returns -1 and sets errno to the error value.

- EACCES: the application manifest does not include the WifiConfig capability.
- EAGAIN: the Wi-Fi device isn't ready yet.
- ENETDOWN: the Wi-Fi network interface is unavailable.

Any other errno may also be specified; such errors aren't deterministic and there's no guarantee that the same behavior will be retained through system updates.

## Returns

The number of networks found, or -1 for failure, in which case errno is set to the error value.

## Application manifest requirements

The [application manifest](#) must include the WifiConfig capability.

# WifiConfig\_ConnectedNetwork Struct

1/7/2019 • 2 minutes to read

**Header:** #include <applibs/wificonfig.h>

The properties of a connected Wi-Fi network, which represent a 802.11 Basic Service Set (BSS).

## NOTE

This is an alias to a versioned structure. Define WIFICONFIG\_STRUCTS\_VERSION to use this alias.

```
struct WifiConfig_ConnectedNetwork {  
    uint32_t z__magicAndVersion;  
    uint8_t ssid[WIFICONFIG_SSID_MAX_LENGTH];  
    uint8_t bssid[WIFICONFIG_BSSID_BUFFER_SIZE];  
    uint8_t ssidLength;  
    WifiConfig_Security_Type security;  
    uint32_t frequencyMHz;  
    int8_t signalRssi;  
};
```

## Members

### **uint32\_t z\_\_magicAndVersion**

A magic number that uniquely identifies the struct version.

### **uint8\_t ssid**

The fixed length buffer that contains the SSID.

### **uint8\_t bssid**

The fixed length buffer that contains the BSSID.

### **uint8\_t ssidLength**

The size of the SSID element in bytes.

### **WifiConfig\_Security\_Type security**

The [WifiConfig\\_Security](#) value that specifies the security key setting.

### **uint32\_t frequencyMHz**

The BSS center frequency in MHz.

### **int8\_t signalRssi**

The RSSI (Received Signal Strength Indicator) value.

# WifiConfig\_ScannedNetwork Struct

1/7/2019 • 2 minutes to read

**Header:** #include <applibs/wificonfig.h>

The properties of a scanned Wi-Fi network, which represent a 802.11 Basic Service Set (BSS).

## NOTE

This is an alias to a versioned structure. Define WIFICONFIG\_STRUCTS\_VERSION to use this alias.

```
struct WifiConfig_ScannedNetwork {
    uint32_t z__magicAndVersion;
    uint8_t ssid[WIFICONFIG_SSID_MAX_LENGTH];
    uint8_t bssid[WIFICONFIG_BSSID_BUFFER_SIZE];
    uint8_t ssidLength;
    WifiConfig_Security_Type security;
    uint32_t frequencyMHz;
    int8_t signalRssi;
};
```

## Members

### **uint32\_t z\_\_magicAndVersion**

A magic number that uniquely identifies the struct version.

### **uint8\_t ssid**

The fixed length buffer that contains the SSID.

### **uint8\_t bssid**

The fixed length buffer that contains the BSSID.

### **uint8\_t ssidLength**

The size of the SSID element in bytes.

### **WifiConfig\_Security\_Type security**

The WifiConfig\_Security value that specifies the security key setting.

### **uint32\_t frequencyMHz**

The BSS center frequency in MHz.

### **int8\_t signalRssi**

The RSSI (Received Signal Strength Indicator) value.

# WifiConfig\_StoredNetwork Struct

1/7/2019 • 2 minutes to read

**Header:** #include <applibs/wificonfig.h>

The properties of a stored Wi-Fi network, which represents a 802.11 Service Set.

## NOTE

This is an alias to a versioned structure. Define WIFICONFIG\_STRUCTS\_VERSION to use this alias.

```
struct WifiConfig_StoredNetwork {
    uint32_t z__magicAndVersion;
    uint8_t ssid[WIFICONFIG_SSID_MAX_LENGTH];
    uint8_t ssidLength;
    bool isEnabled;
    bool isConnected;
    WifiConfig_Security_Type security;
};
```

## Members

### **uint32\_t z\_\_magicAndVersion**

A magic number that uniquely identifies the struct version.

### **uint8\_t ssid**

The fixed length buffer that contains the SSID.

### **uint8\_t ssidLength**

The size of the SSID element in bytes.

### **bool isEnabled**

Indicates whether the network is enabled.

### **bool isConnected**

Indicates whether the network is connected.

### **WifiConfig\_Security\_Type security**

The [WifiConfig\\_Security](#) value that specifies the security key setting.

# WifiConfig\_Security Enum

10/9/2019 • 2 minutes to read

**Header:** #include <applibs/wificonfig.h>

The security key setting for a Wi-Fi network.

```
typedef enum {
    WifiConfig_Security_Unknown = 0,
    WifiConfig_Security_Open = 1,
    WifiConfig_Security_Wpa2_Psk = 2,
    WifiConfig_Security_Wpa2_EAP_TLS = 3,
} WifiConfig_Security;
```

VALUES	DESCRIPTIONS
WifiConfig_Security_Unknown	Unknown security setting.
WifiConfig_Security_Open	No key management.
WifiConfig_Security_Wpa2_Psk	A WPA2 pre-shared key.

# WifiConfig\_Security\_Type Typedef

1/7/2019 • 2 minutes to read

**Header:** #include <applibs/wificonfig.h>

Specifies the type of the security settings values for the [WifiConfig\\_Security](#) enum.

```
typedef uint8_t WifiConfig_Security_Type;
```

# Terminology

5/30/2019 • 9 minutes to read

## **Application**

An executable program that runs on an Azure Sphere device. An application is a type of component.

## **Application capability**

The permissions that an application requires to access resources. For example, applications require capabilities to use peripherals such as GPIOs and UARTs, connect to internet hosts, and change the Wi-Fi configuration.

## **Application containers**

The top (fourth) level of the multi-layer Azure Sphere OS architecture, which provides compartmentalization for agile, secure, and robust high-level applications.

## **Application libraries (applibs)**

The Microsoft-authored custom libraries that support high-level application development.

## **Application manifest**

A file that identifies the application capabilities that an application requires and includes application metadata.

Every application must have an application manifest named `app_manifest.json`.

## **Attestation**

The process by which a client proves its configuration to a remote server. In the Azure Sphere context, an Azure Sphere device attests to the Azure Sphere Security Service so that the service can determine the level of trust and integrity of the device.

## **Authentication**

The process by which an Azure Sphere device confirms its identity with a remote service.

## **Azure Sphere chip**

An MCU that is compatible with Azure Sphere.

## **Azure Sphere Core SDK Preview**

The tools, libraries, and header files that together enable application developers to build applications for the Azure Sphere device. The Core SDK includes all the tools that are required to build and manage applications and deployments without using Visual Studio.

## **Azure Sphere device**

Any device that incorporates an Azure Sphere chip, or the Azure Sphere chip itself.

## **Azure Sphere operating system (OS)**

Microsoft's custom, Linux-based microcontroller operating system that, as designed, runs on an Azure Sphere chip and connects to the Azure Sphere Security Service.

## **Azure Sphere reference development board (RDB)**

A compact development board that incorporates an Azure Sphere chip and conforms to the reference development board design specifications.

## **Azure Sphere SDK Preview for Visual Studio**

A package that includes the Azure Sphere Core SDK together with a Visual Studio extension that integrates build and deployment functionality for Azure Sphere devices within the Visual Studio development environment.

## **Azure Sphere Security Service**

Microsoft's cloud-based service that communicates with Azure Sphere chips to enable remote monitoring, maintenance, update, and control. Sometimes abbreviated AS3.

## Azure Sphere tenant

A special cloud-based entity that represents an organization for the Azure Sphere Security Service. The Azure Sphere tenant provides a secure way for an organization to manage its Azure Sphere devices in isolation from those of any other organization. Each device belongs to exactly one Azure Sphere tenant.

Note that the term "tenant" is sometimes used elsewhere to refer to an Azure Active Directory instance. In the context of Azure Sphere, however, we use "tenant" to refer exclusively to an Azure Sphere tenant.

## Beta API

An application programming interface (API) that is still in development and may change in or be removed from a later release.

## Certificate-based authentication

Authentication that is based on certificates, instead of passwords. A certificate is a statement of identity and authorization that is signed with a secret private key and validated with a known public key, and is thus more secure than a password. Azure Sphere uses certificates to prove identities for mutual authentication when communicating with other local devices and with servers in the cloud. One of the seven properties of highly secure devices.

## Chip SKU

A GUID that identifies a particular type of Azure Sphere-compatible MCU. Microsoft manages chip SKUs. See also [SKU](#), [product SKU](#), and [SKU sets](#).

## Claiming

The process by which an Azure Sphere tenant takes ownership of a device. Each Azure Sphere device must be "claimed" by an Azure Sphere tenant, so that the tenant knows about all its devices and can manage them as a group. A device cannot be claimed by multiple tenants.

## Compartmentalization

The use of protection boundaries within the hardware and software stack to prevent a flaw or breach in one component from propagating to other parts of the system. Azure Sphere incorporates hardware-enforced barriers between software components to provide compartmentalization. One of the seven properties of highly secure devices.

## Component

The updatable unit of software that a feed delivers. Each component has a unique component ID. The component ID for an application appears in the **ComponentId** field of the application's `app_manifest.json` file. See also [image](#).

## Connected device

A manufacturer's product that includes an embedded Azure Sphere chip that runs the Azure Sphere OS and connects to the Azure Sphere Security Service.

## Crossover MCU

A microcontroller unit (MCU) that combines real-time and application processors. The MT3620 is a crossover MCU.

## Defense in depth

A layered approach to security in which multiple mitigations are applied against each threat. One of the seven properties of highly secure devices.

## Deploy

To make a component available for over-the-air update. A deployment delivers software over the air to one or more Azure Sphere devices. See also [sideload](#).

## Device authentication and attestation service

The primary point of contact with the Azure Sphere Security Service for Azure Sphere devices to authenticate their identity, ensure the integrity and trust of the system software, and certify that they are running a trusted code base.

## Device capability

The permission to perform a device-specific activity. For example, the AppDevelopment capability enables debugging on an Azure Sphere device. Device capabilities are granted by the Azure Sphere Security Service and are stored in flash memory on the Azure Sphere chip. By default, Azure Sphere chips have no device capabilities.

#### **Device group**

A named collection of connected devices and the list of feeds that deliver software to those devices.

#### **Device ID**

The unique, immutable value generated by the silicon manufacturer to identify an individual Azure Sphere MCU.

#### **Failure reporting**

The automatic collection and timely distribution of information about a failure, so that problems can be quickly diagnosed and corrected. One of the seven properties of highly secure devices.

#### **Feed**

A named set of components along with a sequence of image sets that represent those components. A feed delivers its current image set to specific SKUs. Currently, the Azure Sphere Security Service supports—and the deployment tools recognize—two types of feeds: application software feeds and system software feeds.

#### **Hardware-based root of trust**

A security foundation that is generated in and protected by hardware. In the Azure Sphere chip, this is implemented as unforgeable cryptographic keys. Physical countermeasures resist side-channel attacks. One of the seven properties of highly secure devices.

#### **High-level application**

An application that runs on the high-level core on the Azure Sphere hardware. High-level applications run on the Azure Sphere OS and can use the application libraries and other OS features.

#### **Image**

A binary file that represents a single version of a specific component. The specific component is identified by its component ID.

#### **Image set**

A group of images that represent interdependent components and are deployed and updated as a unit.

#### **Image type**

An image attribute that identifies the type of component an image represents; synonymous with component type. Depending on the image type, the bits may be in different formats. For applications (which is one image type), images comprise a serialized file system that contains the executable for their code.

#### **Image package**

The combination of an image with its metadata that is produced by the build process. An image package can be sideloaded to an Azure Sphere device for testing and debugging. To deploy an image package, it must be added to an image set.

#### **Image set**

A group of images that are deployed and updated as an atomic unit.

#### **On-chip cloud services**

The third level of the multi-layer Azure Sphere OS architecture, which provides update, authentication, and connectivity.

#### **Over-the-air (OTA) loading**

The process by which the Azure Sphere Security Service communicates with an Azure Sphere device to perform an update. See also [sideload](#).

#### **Pluton security subsystem**

The Azure Sphere subsystem that creates a hardware root of trust, stores private keys, and executes complex cryptographic operations. It includes a Security Processor (SP) CPU, cryptographic engines, a hardware random number generator (RNG), a key store, and a cryptographic operation engine (COE).

## **Product manufacturer**

A company or individual who produces a connected device that incorporates an Azure Sphere MCU and has a custom application.

## **Product SKU**

A GUID that identifies a connected device product that incorporates an Azure Sphere MCU. A product manufacturer creates a product SKU for each model of connected device, such as a dishwasher or coffeemaker.

## **Real-time capable application (RTApp)**

An application that runs on one of the real-time cores on the Azure Sphere hardware. RTApps can run on bare-metal hardware or with a real-time operating system (RTOS).

## **Recovery**

The low-level process of replacing the Azure Sphere OS on the device, without using the OTA update process, but instead using a special recovery bootloader. See also [update](#).

## **Renewable security**

The ability to update to a more secure state automatically even after the device has been compromised. Renewal brings the device forward to a secure state and revokes compromised assets for known vulnerabilities or security breaches. One of the seven properties of highly secure devices.

## **Security monitor**

The lowest level of the Azure Sphere OS architecture, which is responsible for protecting security-sensitive hardware, such as memory, flash, and other shared MCU resources and for safely exposing limited access to these resources.

## **Sideload**

The process of loading software by a means that does not involve the Azure Sphere Security Service but instead is performed directly with the device, often under the control of a software developer, field engineer, or similar person. A developer can initiate sideloading by pressing F5 to deploy an application for debugging in Visual Studio, or by using the **azsphere** CLI with an attached device.

## **SKU set**

A list of SKUs that together identify a target for software updates.

## **Stock keeping unit (SKU)**

A value that identifies a model of physical device. Azure Sphere chips have [chip SKUs](#); connected devices have [product SKUs](#) and chip SKUs.

## **Sysroot**

A set of libraries, header files, and tools that are used to compile and link a high-level application that targets a particular set of APIs. Some sysroots support only production APIs, and other sysroots support both production APIs and Beta APIs. The Azure Sphere SDK includes multiple sysroots that target different API sets.

## **Trusted computing base (TCB)**

The software and hardware that are used to create a secure environment for an operation. The TCB should be kept as small as possible to minimize the surface that is exposed to attackers and to reduce the probability that a bug or feature can be used to circumvent security protections. A small TCB is one of the seven properties of highly secure devices.

## **Update**

The process of changing the Azure Sphere OS or application to comply with a deployment. An update can be sideloaded (such as during development and debugging) or can be delivered over the air by the Azure Sphere Security Service (in a normal end-user situation). Support for OTA update is an integral part of Azure Sphere. See also [recovery](#).

# Hardware and manufacturing overview

12/12/2018 • 2 minutes to read

This topic contains information for designers of boards and modules that incorporate an Azure Sphere chip as well as manufacturers of connected devices that incorporate a chip or module.

As you design your module or board or prepare for manufacturing, we suggest that you proceed in the following order:

1. Become familiar with [MT3620 status](#) and currently supported features.
2. Prototype your hardware by [using the MT3620 development board](#).
3. Design your hardware, borrowing from the [MT3620 reference board design](#) as appropriate.
4. Develop a [programming/debugging interface](#) that enables communication and control between a PC that runs the Azure Sphere tools and a product that incorporates an Azure Sphere chip.
5. If you are designing a module or custom board, evaluate and certify [radio frequency \(RF\) performance](#). The [RF test tools](#) topic describes how to program product-specific radio frequency (RF) settings in e-fuses, such as the antenna configuration and frequency, and how to tune individual devices for optimal performance.  
Alternatively, you can work with an RF test equipment vendor who supports Azure Sphere; Microsoft has currently partnered with [LitePoint](#) to offer RF testing support for the MT3620.
6. Set up the [volume manufacturing](#) and testing of an Azure Sphere connected device.

Supplementary tool packages are available upon request for hardware designers and manufacturers:

- The RF Tools package contains an interactive RF command-line tool, an RF ready-check tool, and an RF C library.
- The Factory Tools package contains the Azure Sphere Core SDK for use on factory-floor PCs, along with a device-ready check tool. The Core SDK includes the tools and utilities that are required to manage Azure Sphere chips.

Please contact your Microsoft representative if you need one of these packages.

# MT3620 Support Status

10/9/2019 • 12 minutes to read

This document describes the current status of Azure Sphere support for the Mediatek MT3620. You may also want to refer to the *MT3620 Product Brief*, which is available for download on the [Mediatek MT3620 web page](#). In addition, Mediatek produces the *MT3620 Hardware User Guide*, which is a detailed guide to integrating the MT3620 MCU into your own hardware. Please contact Mediatek if you require the hardware guide.

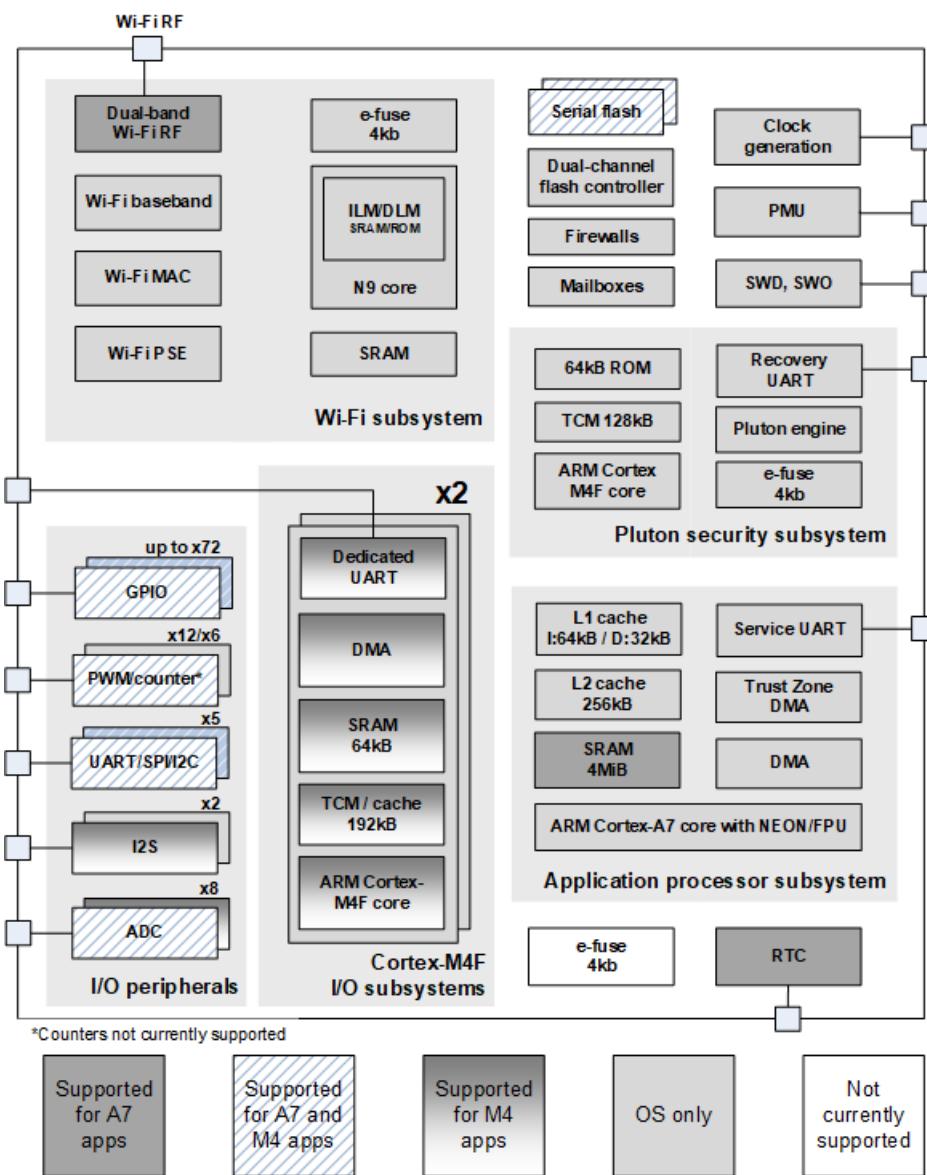
## IMPORTANT

In the context of this document, *not currently supported* means that customer use of the feature is restricted at the current time, and this restriction is likely to be removed in the future. Conversely, *not accessible* means that the feature cannot be used by customers, and this restriction is unlikely to change.

If you have feature requests or feedback, we welcome your comments on the [Azure Sphere forum](#).

## MT3620 Block Diagram

The block diagram shows the MT3620. Features that are not yet supported are shown in white and features that are partially supported are shown with gradient shading.



The sections that follow provide additional details about these features.

## I/O Peripherals

The MT3620 design includes a total of 76 programmable I/O pins, most of which can be multiplexed between general-purpose I/O (GPIO) and other functions.

### GPIO/PWM/counters

Some pins are multiplexed between GPIO, pulse width modification (PWM), and hardware counters. The counters are currently not supported.

GPIO functions currently supported are setting output high/low, reading input, and polling (software-based interrupts).

### Serial interface blocks

The MT3620 design includes five serial interface blocks, each of which contains five pins. (These blocks are also called ISU, for "I2C, SPI, UART.") These serial interface blocks can multiplex GPIO, universal asynchronous receiver-transmitter (UART), inter-integrated circuit (I2C) and serial peripheral interface (SPI).

UART is supported at 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200, 230400, 460800, 500000, 576000, 921600, 1000000, 1152000, 1500000, and 2000000 baud. There is a 32-byte hardware receive buffer. The following UART settings are supported, with 8N1 (8 data bits, 1 stop bit, and no parity) as the default setting:

- Data bit: 5, 6, 7, and 8.

- Stop bit: 1 and 2.
- Parity: odd, even, and none.
- Flow control mode: RTS/CTS, XON/XOFF, and no flow control.

SPI transactions are supported up to 40 MHz. You can connect up to two subordinate SPI devices to each ISU. When you use an ISU port as an SPI master interface, you can't use the same port as an I2C or UART interface. Simultaneous bidirectional read and write (full-duplex) SPI operations within a single bus transaction are not supported. The following SPI settings are supported:

- Communication mode (clock polarity, clock phase): SPI mode 0 (CPOL = 0, CPHA = 0), SPI mode 1 (CPOL = 0, CPHA = 1), SPI mode 2 (CPOL = 1, CPHA = 0), and SPI mode 3 (CPOL = 1, CPHA = 1).
- Bit order: least significant is sent first, and most significant is sent first.
- Chip select polarity: active-high, active-low. Active-low is the default setting.

7-bit subordinate device addresses are supported for I2C. When you use an ISU port as an I2C master interface, you can't use the same port as an SPI or UART interface. 0-byte I2C reads are not supported. The following I2C settings are supported:

- 100 KHz, 400 KHz, and 1 MHz bus speeds.
- Custom timeout for operations.

## I2S

Two blocks of five pins are multiplexed between GPIO and I2S. I2S is currently supported for M4 applications only.

GPIO functions currently supported are setting output high/low, reading input, and polling (software-based interrupts).

## ADC

A block of eight pins is multiplexed between GPIO and an analog-to-digital converter (ADC).

GPIO functions currently supported are setting output high/low, reading input, and polling (software-based interrupts).

## ARM Cortex-M4F subsystems

The MT3620 includes two general-purpose ARM Cortex-M4F subsystems, each of which has a dedicated GPIO/UART block. Customer use of these subsystems (and the GPIOs/UARTs) is currently in preview.

### NOTE

For details about programming the M4 cores on the MT3620, see the [MT3620 datasheet](#) published by Mediatek. If the datasheet does not contain all the information you need, please email Avnet (Azure.Sphere@avnet.com) to request the full datasheet.

## Application processor subsystem

The ARM Cortex-A7 subsystem runs a customer application along with the Microsoft-supplied Linux-based kernel, services, and libraries.

The service UART is dedicated to system functionality for the A7 subsystem. It is not available for customer application use.

The one-time programmable e-fuse block, for storing device specific information, cannot be used by customer applications.

# Wi-Fi Subsystem

The Wi-Fi subsystem is currently IEEE 802.11 b/g/n compliant at both 2.4 GHz and 5 GHz.

See [RF test tools](#) for information about radio-frequency testing and calibration.

## Power control

MT3620 includes a number of features to selectively control power consumption. Currently these are largely unsupported.

## Clocks and power sources

The main crystal can currently only be 26MHz. Crystal frequencies other than 26MHz are not currently supported in software.

Configuring clock generation with PWMs is not currently supported.

## Brownout detection

Brownout detection is not currently supported.

## Hardware watchdog timers

The hardware watchdog timers associated with the various cores in MT3620 are not currently supported.

## SWD, SWO

Serial-wire debug (SWD, pins 98-99) is supported for M4 applications only. Serial-wire output (SWO, pin 100) is not currently supported. Debugging an A7 application is supported by a Microsoft-supplied gdb-based mechanism.

## RAM and flash

The MT3620 includes approximately 5 MB RAM on-die, including 256 KiB in each I/O subsystem and 4 MB in the A7 application subsystem.

The MT3620 can be ordered with 16 MB of SPI flash memory.

For information about RAM and flash available for applications, see [Memory available for applications](#).

## Manufacturing test support

Documentation and utilities to support the integration of custom manufacturing test applications with factory processes are not yet available.

## Pinout

PIN#	PIN NAME	MAJOR FUNCTIONS	TYPE	DESCRIPTION	COMMENTS
1	GND		P	Ground	

PIN#	PIN NAME	MAJOR FUNCTIONS	TYPE	DESCRIPTION	COMMENTS
2	AVDD_3V3_WF_A_PA		PI	3.3V power rail for 5GHz Wi-Fi power amplifier	
3	AVDD_3V3_WF_A_PA		PI	3.3V power rail for 5GHz Wi-Fi power amplifier	
4	NC				
5	NC				
6	AVDD_1V6_WF_TRX		PI	1.6V power rail for Wi-Fi transmit/receive	
7	AVDD_1V6_WF_AFE		PI	1.6V power rail for Wi-Fi analog front end	
8	NC				
9	AVDD_1V6_XO		PI	1.6V power rail for main crystal oscillator	
10	MAIN_XIN		AI	Main crystal oscillator input	
11	WF_ANTSEL0		DO	Wi-Fi antenna select for external DPDT switch	
12	WF_ANTSEL1		DO	Wi-Fi antenna select for external DPDT switch	
13	GPIO0	GPIO0/PWM0	DIO	Interrupt-capable GPIO multiplexed with PWM output	
14	GPIO1	GPIO1/PWM1	DIO	Interrupt-capable GPIO multiplexed with PWM output	
15	GPIO2	GPIO2/PWM2	DIO	Interrupt-capable GPIO multiplexed with PWM output	

PIN#	PIN NAME	MAJOR FUNCTIONS	TYPE	DESCRIPTION	COMMENTS
16	GPIO3	GPIO3/PWM3	DIO	Interrupt-capable GPIO multiplexed with PWM output	
17	GPIO4	GPIO4/PWM4	DIO	Interrupt-capable GPIO multiplexed with PWM output	
18	GPIO5	GPIO5/PWM5	DIO	Interrupt-capable GPIO multiplexed with PWM output	
19	GPIO6	GPIO6/PWM6	DIO	Interrupt-capable GPIO multiplexed with PWM output	
20	GPIO7	GPIO7/PWM7	DIO	Interrupt-capable GPIO multiplexed with PWM output	
21	GPIO8	GPIO8/PWM8	DIO	Interrupt-capable GPIO multiplexed with PWM output	
22	GPIO9	GPIO9/PWM9	DIO	Interrupt-capable GPIO multiplexed with PWM output	
23	DVDD_1V15		PI	1.15V power rail	
24	DVDD_3V3		PI	3.3V power rail	
25	GPIO10	GPIO10/PWM10	DIO	Interrupt-capable GPIO multiplexed with PWM output	
26	GPIO11	GPIO11/PWM11	DIO	Interrupt-capable GPIO multiplexed with PWM output	
27	GPIO12		DIO	Interrupt-capable GPIO	Interrupts are not currently supported

PIN#	PIN NAME	MAJOR FUNCTIONS	TYPE	DESCRIPTION	COMMENTS
28	GPIO13		DIO	Interrupt-capable GPIO	Interrupts are not currently supported
29	GPIO14		DIO	Interrupt-capable GPIO	Interrupts are not currently supported
30	GPIO15		DIO	Interrupt-capable GPIO	Interrupts are not currently supported
31	GPIO16		DIO	Interrupt-capable GPIO	Interrupts are not currently supported
32	GPIO17		DIO	Interrupt-capable GPIO	Interrupts are not currently supported
33	GPIO18		DIO	Interrupt-capable GPIO	Interrupts are not currently supported
34	GPIO19		DIO	Interrupt-capable GPIO	Interrupts are not currently supported
35	GPIO20		DIO	Interrupt-capable GPIO	Interrupts are not currently supported
36	GPIO21		DIO	Interrupt-capable GPIO	Interrupts are not currently supported
37	GPIO22		DIO	Interrupt-capable GPIO	Interrupts are not currently supported
38	GPIO23		DIO	Interrupt-capable GPIO	Interrupts are not currently supported
39	GPIO26	GPIO26/ SCLK0/TXD0	DIO	GPIO multiplexed with ISU 0 functions	
40	GPIO27	GPIO27/ MOSI0/RTS0/SCL0	DIO	GPIO multiplexed with ISU 0 functions	
41	GND		P	Ground	

PIN#	PIN NAME	MAJOR FUNCTIONS	TYPE	DESCRIPTION	COMMENTS
42	GPIO28	GPIO28/ MISO0/RXD0/SD A0	DIO	GPIO multiplexed with ISU 0 functions	
43	GPIO29	GPIO29/CSA0/C TS0	DIO	GPIO multiplexed with ISU 0 functions	
44	DVDD_1V15		PI	1.15V power rail	
45	GPIO30	GPIO30/CSB0	DIO	GPIO multiplexed with ISU 0 functions	
46	GPIO31	GPIO31/ SCLK1/TXD1	DIO	GPIO multiplexed with ISU 1 functions	
47	GPIO32	GPIO32/ MOSI1/RTS1/SCL 1	DIO	GPIO multiplexed with ISU 1 functions	
48	GPIO33	GPIO33/ MISO1/RXD1/SD A1	DIO	GPIO multiplexed with ISU 1 functions	
49	GPIO34	GPIO34/CSA1/C TS1	DIO	GPIO multiplexed with ISU 1 functions	
50	GPIO35	GPIO35/CSB1	DIO	GPIO multiplexed with ISU 1 functions	
51	GPIO36	GPIO36/ SCLK2/TXD2	DIO	GPIO multiplexed with ISU 2 functions	
52	GPIO37	GPIO37/ MOSI2/RTS2/SCL 2	DIO	GPIO multiplexed with ISU 2 functions	
53	GPIO38	GPIO38/ MISO2/RXD2/SD A2	DIO	GPIO multiplexed with ISU 2 functions	
54	GPIO39	GPIO39/CSA2/C TS2	DIO	GPIO multiplexed with ISU 2 functions	
55	GPIO40	GPIO40/CSB2	DIO	GPIO multiplexed with ISU 2 functions	

PIN#	PIN NAME	MAJOR FUNCTIONS	TYPE	DESCRIPTION	COMMENTS
56	DVDD_3V3		PI	3.3V power rail	
57	DVDD_1V15		PI	1.15V power rail	
58	GPIO41	GPIO41/ADC0	DIO	GPIO multiplexed with ADC input	
59	GPIO42	GPIO42/ADC1	DIO	GPIO multiplexed with ADC input	
60	GPIO43	GPIO43/ADC2	DIO	GPIO multiplexed with ADC input	
61	GPIO44	GPIO44/ADC3	DIO	GPIO multiplexed with ADC input	
62	GPIO45	GPIO45/ADC4	DIO	GPIO multiplexed with ADC input	
63	GPIO46	GPIO46/ADC5	DIO	GPIO multiplexed with ADC input	
64	GPIO47	GPIO47/ADC6	DIO	GPIO multiplexed with ADC input	
65	GPIO48	GPIO48/ADC7	DIO	GPIO multiplexed with ADC input	
66	AVDD_2V5_ADC		PI	2.5V power rail for ADC	
67	VREF_ADC		AI	Reference voltage for ADC	
68	AVSS_2V5_ADC		P	Ground for ADC	
69	EXT_PMU_EN		DO	External power supply enable output	
70	WAKEUP		DI	External wakeup from deepest sleep mode	Not currently supported
71	AVDD_3V3_RTC		PI	3.3V power rail for real-time clock	
72	RTC_XIN		AI	Realtime clock crystal oscillator input	

PIN#	PIN NAME	MAJOR FUNCTIONS	TYPE	DESCRIPTION	COMMENTS
73	RTC_XOUT		AO	Realtime clock crystal oscillator output	
74	AVDD_3V3_XPPLL		PI	3.3V power rail for internal phase-locked loop	
75	I2S_MCLK0_ALT		AO	Analog alternative to MCLK0	I2S is currently supported for M4 applications only.
76	I2S_MCLK1_ALT		AO	Analog alternative to MCLK1	I2S is currently supported for M4 applications only.
77	DVDD_1V15		PI	1.15V power rail	
78	DVDD_1V15		PI	1.15V power rail	
79	VOUT_2V5		PO	Output from internal 2.5V LDO	
80	AVDD_3V3		PI	3.3V power rail	
81	PMU_EN		DI	Internal PMU override	
82	RESERVED				
83	GND		P	Ground	
84	SENSE_1V15		AI	Sense input to stabilise the 1.15V power supply	
85	VOUT_1V15		PO	Output from internal 1.15V LDO	
86	AVDD_1V6_CLDO		PI	1.6V power rail for the internal 1.15V core LDO	

PIN#	PIN NAME	MAJOR FUNCTIONS	TYPE	DESCRIPTION	COMMENTS
87	PMU_CAP		A	Connect a capacitor between this pin and AVDD_3V3_BUC K to maintain PMU stability	
88	AVDD_3V3_BUC K		PI	3.3V power rail for internal 1.6V buck DC-DC converter	
89	AVDD_3V3_BUC K		PI	3.3V power rail for internal 1.6V buck DC-DC converter	
90	VOUT_1V6		PO	Output from internal 1.6V buck converter	
91	VOUT_1V6		PO	Output from internal 1.6V buck converter	
92	AVSS_3V3_BUCK		P	Ground for internal 1.6V buck converter	
93	AVSS_3V3_BUCK		P	Ground for internal 1.6V buck converter	
94	DEBUG_RXD		DI	Reserved for Azure Sphere debug	
95	DEBUG_TXD		DO	Reserved for Azure Sphere debug	
96	DEBUG_RTS		DO	Reserved for Azure Sphere debug	
97	DEBUG_CTS		DI	Reserved for Azure Sphere debug	
98	SWD_DIO		DIO	ARM SWD for Cortex-M4F debug	

PIN#	PIN NAME	MAJOR FUNCTIONS	TYPE	DESCRIPTION	COMMENTS
99	SWD_CLK		DI	ARM SWD for Cortex-M4F debug	
100	SWO		DO	ARM SWO for Cortex-M4F debug	Not currently supported
101	GPIO56	GPIO56/TX0	DIO	GPIO multiplexed with I2S 0	I2S is currently supported for M4 applications only.
102	GPIO57	GPIO57 /MCLK0	DIO	GPIO multiplexed with I2S 0	I2S is currently supported for M4 applications only.
103	GPIO58	GPIO58/FS0	DIO	GPIO multiplexed with I2S 0	I2S is currently supported for M4 applications only.
104	GPIO59	GPIO59/RX0	DIO	GPIO multiplexed with I2S 0	I2S is currently supported for M4 applications only.
105	GPIO60	GPIO60/ BCLK0	DIO	GPIO multiplexed with I2S 0	I2S is currently supported for M4 applications only.
106	DVDD_1V15		PI	1.15V power rail	
107	DVDD_3V3		PI	3.3V power rail	
108	GPIO61	GPIO61/TX1	DIO	GPIO multiplexed with I2S 1	I2S is currently supported for M4 applications only.
109	GPIO62	GPIO62/ MCLK1	DIO	GPIO multiplexed with I2S 1	I2S is currently supported for M4 applications only.
110	GPIO63	GPIO63/FS1	DIO	GPIO multiplexed with I2S 1	I2S is currently supported for M4 applications only.

PIN#	PIN NAME	MAJOR FUNCTIONS	TYPE	DESCRIPTION	COMMENTS
111	GPIO64	GPIO64/RX1	DIO	GPIO multiplexed with I2S 1	I2S is currently supported for M4 applications only.
112	GPIO65	GPIO65/ BCLK1	DIO	GPIO multiplexed with I2S 1	I2S is currently supported for M4 applications only.
113	GPIO66	GPIO66/ SCLK3/TXD3	DIO	GPIO multiplexed with ISU 3 functions	
114	GPIO67	GPIO67/ MOSI3/RTS3/SCL3	DIO	GPIO multiplexed with ISU 3 functions	
115	GPIO68	GPIO68/ MISO3/RXD3/SDA3	DIO	GPIO multiplexed with ISU 3 functions	
116	GPIO69	GPIO69/CSA3/CTS3	DIO	GPIO multiplexed with ISU 3 functions	
117	GPIO70	GPIO70/CSB3	DIO	GPIO multiplexed with ISU 3 functions	Currently supports GPIO only
118	DVDD_3V3		PI	3.3V power rail	
119	GPIO71	GPIO71/ SCLK4/TXD4	DIO	GPIO multiplexed with ISU 4 functions	
120	GPIO72	GPIO72/ MOSI4/RTS4/SCL4	DIO	GPIO multiplexed with ISU 4 functions	
121	DVDD_1V15		PI	1.15V power rail	
122	GPIO73	GPIO73/ MISO4/RXD4/SDA4	DIO	GPIO multiplexed with ISU 4 functions	
123	GPIO74	GPIO74/CSA4/CTS4	DIO	GPIO multiplexed with ISU 4 functions	
124	GPIO75	GPIO75/CSB4	DIO	GPIO multiplexed with ISU 4 functions	

PIN#	PIN NAME	MAJOR FUNCTIONS	TYPE	DESCRIPTION	COMMENTS
125	SYSRST_N		DI	System reset, active low	
126	DVDD_1V15		PI	1.15V power rail	
127	SERVICE_TXD		DO	Azure Sphere service port	Not available for customer application use
128	SERVICE_RTS		DO	Azure Sphere service port	Not available for customer application use
129	SERVICE_RXD		DI	Azure Sphere service port	Not available for customer application use
130	SERVICE_CTS		DI	Azure Sphere service port	Not available for customer application use
131	RESERVED				
132	DVDD_1V15		PI	1.15V power rail	
133	DVDD_3V3		PI	3.3V power rail	
134	RECOVERY_RXD		DI	Azure Sphere recovery port	Not available for customer application use
135	RECOVERY_TXD		DO	Azure Sphere recovery port	Not available for customer application use
136	RECOVERY_RTS		DO	Azure Sphere recovery port	Not available for customer application use
137	RECOVERY_CTS		DI	Azure Sphere recovery port	Not available for customer application use
138	IO0_GPIO85	IO0_GPIO85/ IO0_RXD	DI	Dedicated GPIO multiplexed with UART for I/O CM4F 0	
139	IO0_GPIO86	IO0_GPIO86/ IO0_TXD	DO	Dedicated GPIO multiplexed with UART for I/O CM4F 0	

PIN#	PIN NAME	MAJOR FUNCTIONS	TYPE	DESCRIPTION	COMMENTS
140	IO0_GPIO87	IO0_GPIO87/ IO0_RTS	DO	Dedicated GPIO multiplexed with UART for I/O CM4F 0	
141	IO0_GPIO88	IO0_GPIO88/ IO0_CTS	DI	Dedicated GPIO multiplexed with UART for I/O CM4F 0	
142	IO1_GPIO89	IO1_GPIO89/ IO1_RXD	DI	Dedicated GPIO multiplexed with UART for I/O CM4F 1	
143	IO1_GPIO90	IO1_GPIO90/ IO1_TXD	DO	Dedicated GPIO multiplexed with UART for I/O CM4F 1	
144	DVDD_3V3		PI	3.3V power rail	
145	IO1_GPIO91	IO1_GPIO91/ IO1_RTS	DO	Dedicated GPIO multiplexed with UART for I/O CM4F 1	
146	IO1_GPIO92	IO1_GPIO92/ IO1_CTS	DI	Dedicated GPIO multiplexed with UART for I/O CM4F 1	
147	RESERVED				
148	TEST		DI	Must be pulled low for normal operation	
149	WF_G_RF_AUXIN		RF	2.4GHz Wi-Fi receive diversity port	
150	NC				
151	AVDD_3V3_WF_ G_PA		PI	3.3V power rail for 2.4GHz Wi-Fi power amplifier	
152	NC				
153	WF_G_RF_ION		RF	2.4GHz Wi-Fi antenna port (differential)	

PIN#	PIN NAME	MAJOR FUNCTIONS	TYPE	DESCRIPTION	COMMENTS
154	WF_G_RF_ION		RF	2.4GHz Wi-Fi antenna port (differential)	
155	WF_G_RF_IOP		RF	2.4GHz Wi-Fi antenna port (differential)	
156	WF_G_RF_IOP		RF	2.4GHz Wi-Fi antenna port (differential)	
157	NC				
158	AVDD_3V3_WF_G_TX		PI	3.3V power rail for 2.4GHz Wi-Fi power transmit	
159	WF_A_RF_AUXIN		RF	5GHz Wi-Fi receive diversity port	
160	AVDD_3V3_WF_A_TX		PI	3.3V power rail for 5GHz Wi-Fi power transmit	
161	NC				
162	WF_A_RFIO		RF	5GHz Wi-Fi antenna port (unbalanced)	
163	WF_A_RFIO		RF	5GHz Wi-Fi antenna port (unbalanced)	
164	GND		P	Ground	
165	EPAD		P	Ground	

# MT3620 Hardware Notes

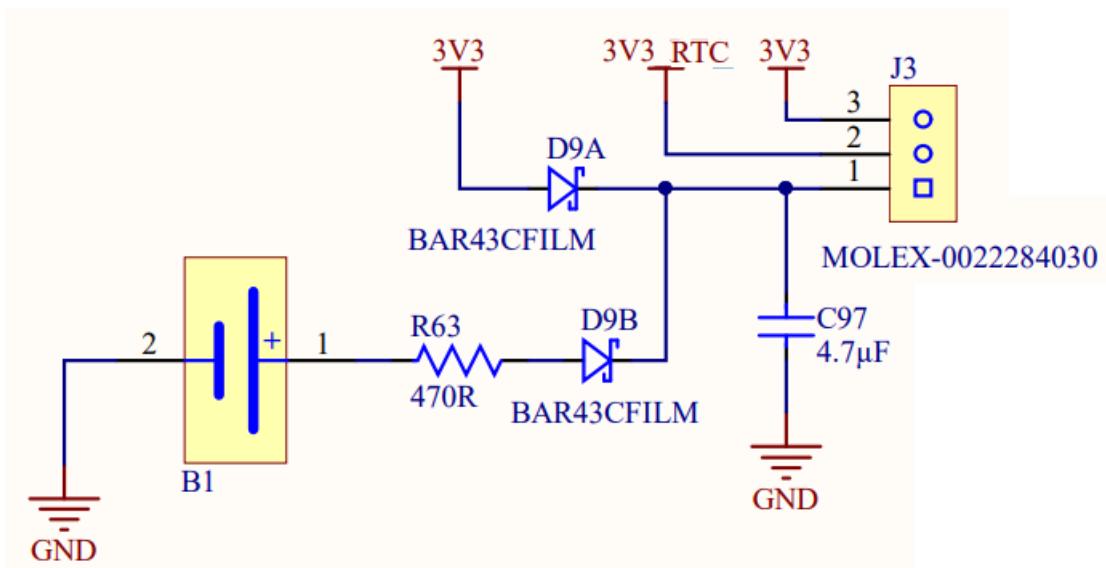
10/9/2019 • 2 minutes to read

The topics covered in this section reflect updated guidance from MediaTek in their MT3620 hardware design documents and data sheet. For further details about these topics please refer to the MediaTek MT3620 documentation.

## RTC power requirements

If the MT3620 is configured to use the on-board real-time clock (RTC) with a 32KHz crystal, you must ensure that the RTC will be powered at start-up or the system will hang. You can do this by simply connecting system power to the RTC power input (MT3620 pin 71). However, if your application requires a backup power source for the RTC, MediaTek recommends that you incorporate, in your design, a way to automatically switch between backup power and system power.

The following circuit appears in the MediaTek MT3620 Hardware Design Guide, and illustrates both ways of powering the RTC on the MT3620. The setting of J3 determines whether system power directly powers the RTC or whether a battery backup circuit powers the RTC. When a jumper connects pins 2 and 3 of J3, the 3V3\_RTC (RTC power input) power rail is directly connected to system power. If the jumper connects pins 1 and 2 of J3 then 3V3\_RTC is powered by the battery backup circuit; the battery backup circuit uses the battery when system power is unavailable or system power when the battery is depleted.



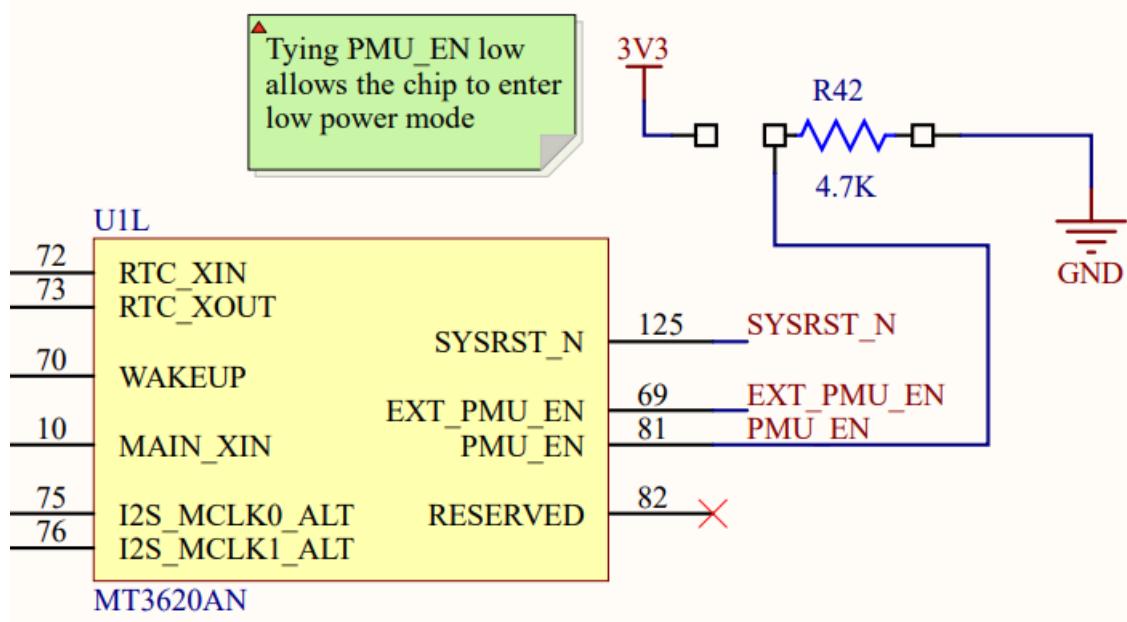
## ADC/GPIO voltage level requirements

The MT3620 ADC input pins can also be configured as GPIO pins. This is a potential source of confusion because when used as GPIO pins they can operate at 3.3 volts, whereas when used as ADC inputs the maximum input voltage cannot exceed 2.5V. Additionally, the voltage reference for the MT3620 (VREF\_ADC) has a maximum voltage of 2.5V so analog signals greater than 2.5V will exceed the full-scale range of the ADC. To handle analog signals at higher voltages, designers should use external filters or external ADC devices.

## PMU\_EN signal

The setting on the PMU\_EN pin controls whether the MT3620 can shut itself off and enter the low-power RTC mode. Although Azure Sphere OS does not yet support low-power RTC mode, this hardware setting affects the

OS's ability to put the MT3620 into a low power mode in the future. Microsoft's recommendation is to set this pin to logic 0 unless low-power functionality is not desired. For example, in the following circuit the MT3620 PMU\_EN pin is pulled low (set to logic zero) via pull-down resistor R42.



# MT3620 development board user guide

7/31/2019 • 7 minutes to read

This topic describes user features of the MT3620 development board:

- Programmable buttons and LEDs
- Four banks of interface headers for input and output
- Configurable power supply
- Configurable Wi-Fi antennas
- Ground test point

## Buttons and LEDs

The board supports two user buttons, a Reset button, four RGB user LEDs, an application status LED, a Wi-Fi status LED, a USB activity LED, and a power-on LED. The sample applications in the Azure Sphere SDK demonstrate the programming of the user buttons and the user LEDs. To ensure compatibility with the Microsoft samples, any development board should support these features.

The following sections provide details of how each of these buttons and LEDs connects to the MT3620 chip.

### User buttons

The two user buttons (A and B) are connected to the GPIO pins listed in the following table. Note that these GPIO inputs are pulled high via 4.7K resistors. Therefore, the default input state of these GPIOs is high; when a user presses a button, the GPIO input is low.

BUTTON	MT3620 GPIO	MT3620 PHYSICAL PIN
A	GPIO12	27
B	GPIO13	28

### Reset button

The development board includes a reset button. When pressed, this button resets the MT3620 chip. It does not reset any other parts of the board.

### User LEDs

The development board includes four RGB user LEDs, labelled 1-4. The LEDs connect to MT3620 GPIOs as listed in the table. The common anode of each RGB LED is tied high; therefore, driving the corresponding GPIO low illuminates the LED.

LED	COLOR CHANNEL	MT3620 GPIO	MT3620 PHYSICAL PIN
1	Red	GPIO8	21
1	Green	GPIO9	22
1	Blue	GPIO10	25
2	Red	GPIO15	30

LED	COLOR CHANNEL	MT3620 GPIO	MT3620 PHYSICAL PIN
2	Green	GPIO16	31
2	Blue	GPIO17	32
3	Red	GPIO18	33
3	Green	GPIO19	34
3	Blue	GPIO20	35
4	Red	GPIO21	36
4	Green	GPIO22	37
4	Blue	GPIO23	38

### Application status LED

The application status LED is intended to provide feedback to the user about the current state of the application that is running on the A7. This LED is not controlled by the Azure Sphere operating system (OS); the application is responsible for driving it.

LED	COLOR CHANNEL	MT3620 GPIO	MT3620 PHYSICAL PIN
Application status	Red	GPIO45	62
Application status	Green	GPIO46	63
Application status	Blue	GPIO47	64

### Wi-Fi status LED

The Wi-Fi status LED is intended to provide feedback to the user about the current state of the Wi-Fi connection. This LED is not controlled by the Azure Sphere OS; the application is responsible for driving it.

LED	COLOR CHANNEL	MT3620 GPIO	MT3620 PHYSICAL PIN
Wi-Fi Status	Red	GPIO48	65
Wi-Fi Status	Green	GPIO14	29
Wi-Fi Status	Blue	GPIO11	26

### USB activity LED

The green USB activity LED blinks whenever data is sent or received over the USB connection. The hardware is implemented so that data sent or received over any of the four Future Technology Devices International (FTDI) channels causes the LED to blink. The USB activity LED is driven by dedicated circuitry and therefore requires no additional software support.

### Power-on LED

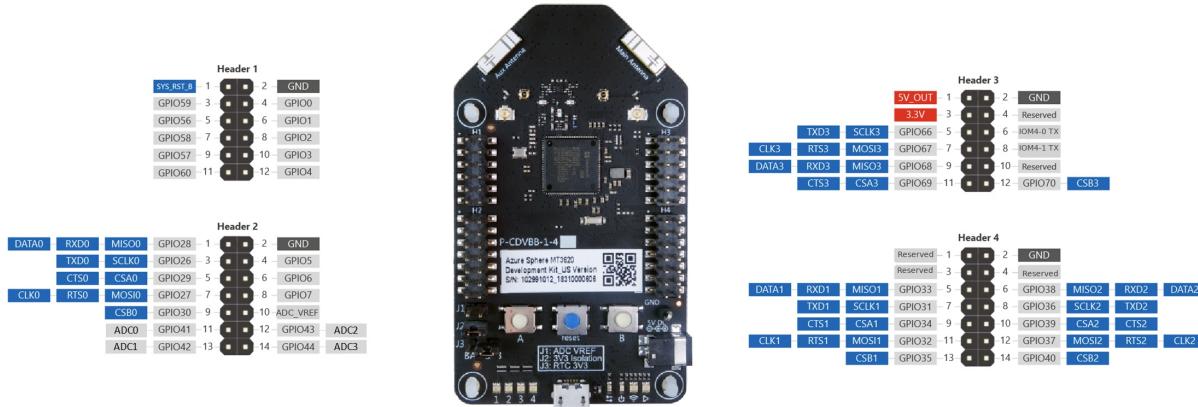
The board includes a red power-on LED that illuminates when the board is powered by USB, an external 5V supply, or an external 3.3V supply.

# Interface headers

The development board includes four banks of interface headers, labelled H1-H4, which provide access to a variety of interface signals. The diagram shows the pin functions that are currently supported.

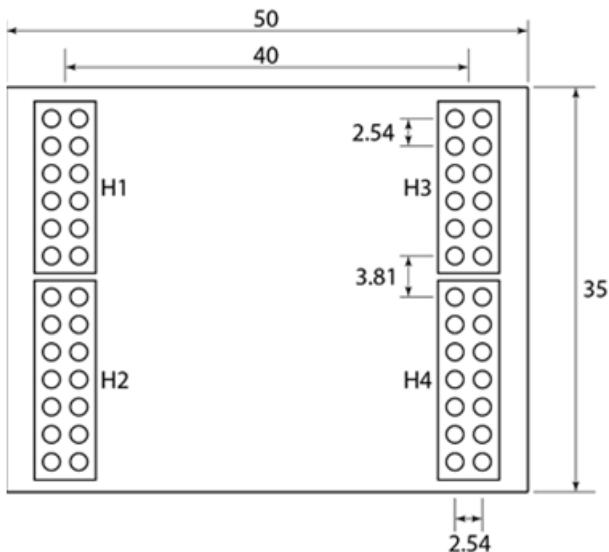
## NOTE

For I2C, DATA and CLK in the diagram correspond to SDA and SCL. Pull-up I2C SCL and I2C SDA with 10K ohm resistors.



# Daughter board

The headers are arranged to allow a daughter board (also referred to as a "shield" or "hat") to be attached to the board. The following diagram shows the dimensions of the daughter board that Microsoft has developed for internal use, along with the locations of the headers.



All dimensions in mm  
Not to scale

# Power supply

The MT3620 board can be powered by USB, by an external 5V supply, or by both. If both sources are simultaneously connected, circuitry prevents the external 5V supply from back-powering the USB.

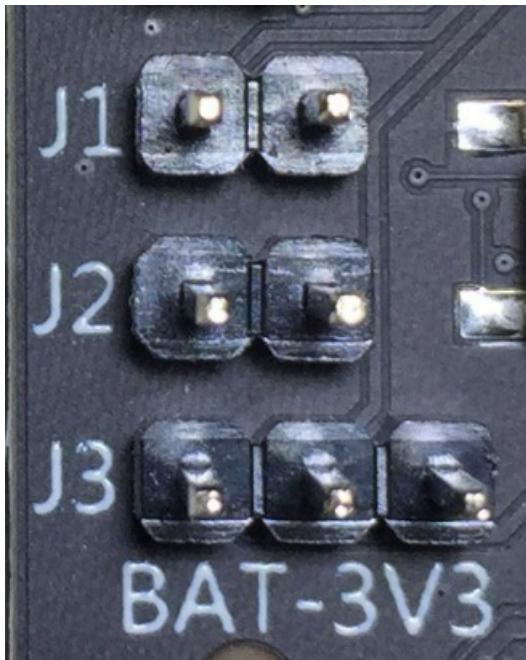
The on-board power supply is protected against reverse-voltage and over-current. If an over-current situation occurs, the protection circuit trips and isolates the incoming 5V supply from the rest of the on-board power supply rail. Note that in this case, the over-current circuit latches in the "tripped" state until the incoming 5V

power supply is turned off. That is, after the over-current circuit has tripped, it latches in the off state, regardless of whether the fault that caused the over-current situation is removed.

The power source must be capable of supplying 600mA even though this much current is not requested during USB enumeration. The board draws around 225mA while running, rising to around 475mA during Wi-Fi data transfer. During boot and while associating to a wireless access point, the board may require up to 600mA for a short time (approximately 2ms). If additional loads are wired to the development board header pins, a source capable of supplying more than 600mA will be required.

A CR2032 battery can be fitted to the board to power the internal real-time clock (RTC) of the MT3620 chip. Alternatively, an external battery can be connected.

Three jumpers (J1-J3) provide flexibility in configuring power for the board. The jumpers are located towards the lower-left of the board; in each case, pin 1 is on the left:



The board ships with headers on J2 and J3:

- A link on J2 indicates that the on-board power supply powers the board.
- A link on pins 2 and 3 of J3 sets the power source for the real-time clock (RTC) to the main 3V3 power supply. Alternatively, to power the board by a coin-cell battery, link pins 1 and 2 of J3 and fit a CR2032 battery into the slot on the back of the board.

**IMPORTANT**

The MT3620 fails to operate correctly if the RTC is not powered.

The following table provides additional detail about the jumpers.

JUMPER	FUNCTION	DESCRIPTION	JUMPER PINS
--------	----------	-------------	-------------

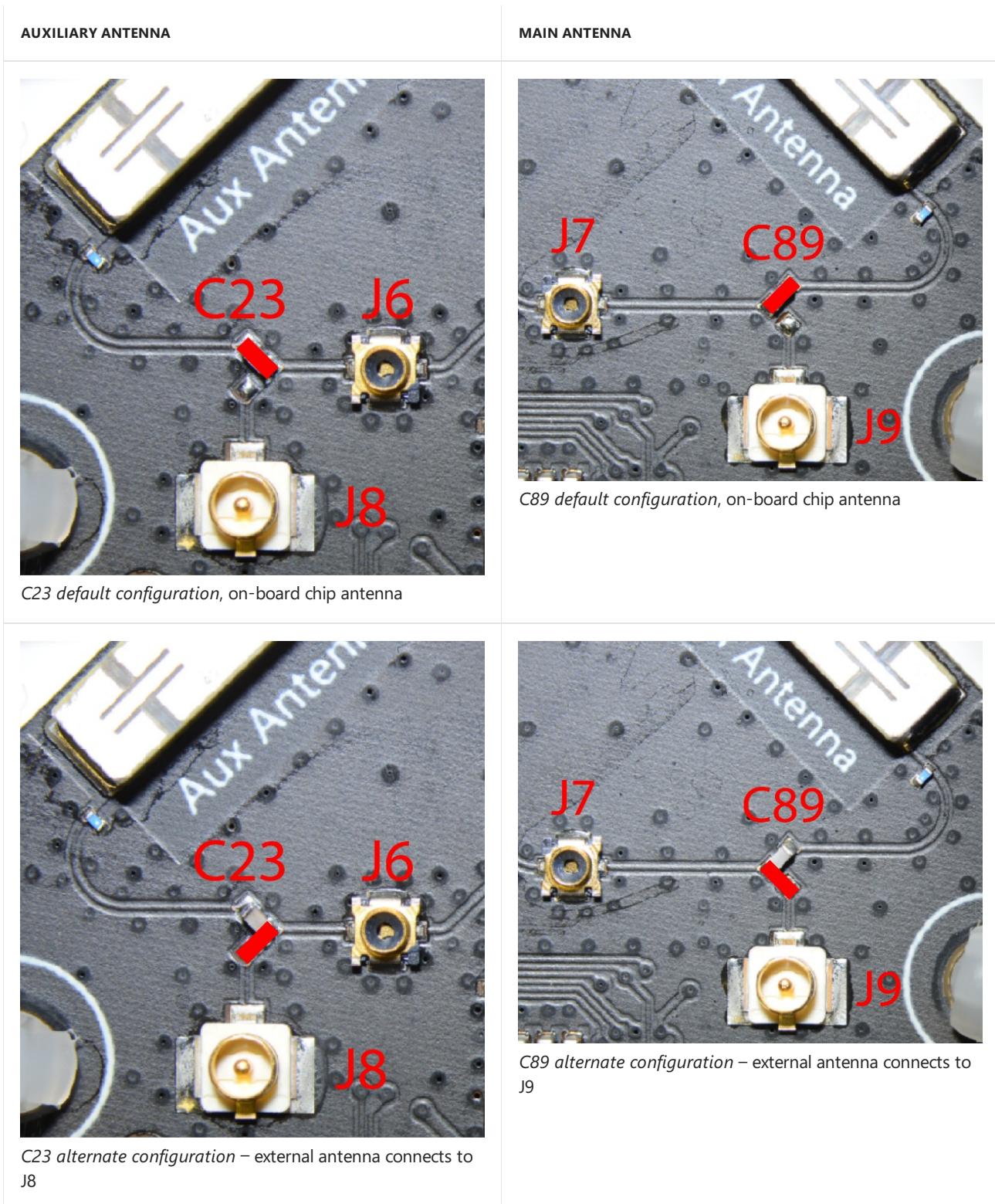
JUMPER	FUNCTION	DESCRIPTION	JUMPER PINS
J1	ADC VREF	<p>This jumper provides a way to set the ADC reference voltage. Place a link on J1 to connect the MT3620's 2.5V output to the ADC VREF pin, so that the ADC reference voltage is 2.5V. Alternatively, connect an external 1.8V reference voltage to pin 1 of the jumper.</p> <p><b>Note:</b> ADC is not currently supported in software.</p>	1, 2
J2	3V3 Isolation	<p>This jumper provides a way to isolate the on-board 3.3V power supply from the rest of the board. For normal use, place a link on J2, indicating that the on-board power supply powers the board. To use an external 3.3V supply to power the board, connect the external 3.3V supply to pin 2 of J2.</p>	1, 2
J3	RTC Supply	<p>This jumper sets the power source for the RTC.</p> <p>During development, it is often convenient to power the RTC directly from the main 3V3 supply, thus avoiding the need to fit a battery. To do so, place a link between pins 2 and 3 of J3. This is normal use.</p> <p>Alternatively, to power the RTC from the on-board coin cell battery, place a link between pins 1 and 2 of J3.</p> <p>Finally, it is possible to power the RTC from an external source by applying this to pin 2 of J3.</p>	

## Wi-Fi antennas

The MT3620 development board includes two dual-band chip antennas and two RF connectors for connecting external antennas or RF test equipment. One antenna is considered the 'main' antenna and the second is considered 'auxiliary'. By default, the development board is configured to use the on-board main antenna; the auxiliary antenna is not currently used.

To enable and use the RF connectors, you must reorient capacitors C23 and/or C89. The first row in the following table shows the default configuration where the on-board chip antennas are in use, with the associated capacitor positions highlighted in red. The images on the second row show the re-oriented capacitor positions in

green.



#### NOTE

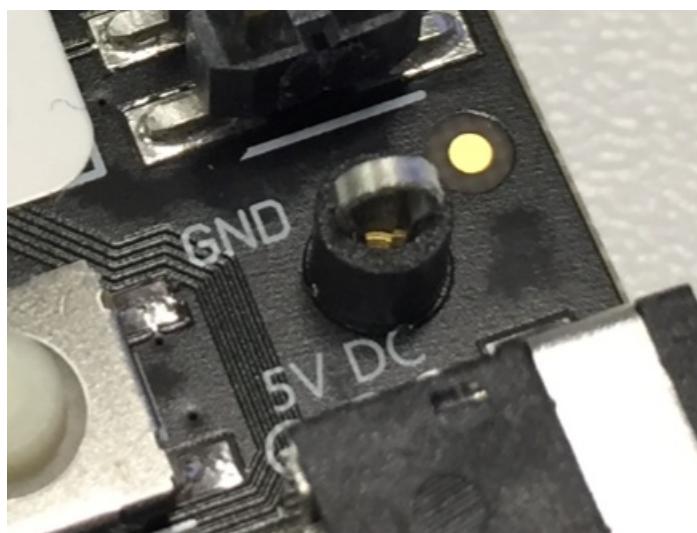
Connectors J6 and J7 are used for RF testing and calibration during manufacture and are not intended for permanent connection to test equipment or external antennas.

Any type of 2.4 or 5GHz external antenna with a U.FL or IPX connector can be used with the board, such as the Molex 1461530100 (pictured below). When fitting an external antenna, you are responsible for ensuring that all regulatory and certification requirements are met.



## Ground test point

The MT3620 development board provides a ground test point on the right-hand side, next to button B and immediately above the 3.5 mm barrel socket, as shown in the image. Use this during testing—for example, for attaching the ground lead of an oscilloscope probe.



# MT3620 reference board design

12/12/2018 • 2 minutes to read

During the development of Azure Sphere, Microsoft created a development board for the MT3620. This development board serves as a reference for others to build MT3620 development boards or to develop modules and devices based on MT3620, so we refer to it as the Microsoft MT3620 Reference Development Board (hereafter RDB). The RDB is compatible with the Azure Sphere templates and utilities.

This topic presents some of the considerations made during its design. It supplements the user information in the [MT3620 development board user guide](#).

As Azure Sphere development progresses, the Azure Sphere OS and tools evolve to support additional features of the MT3620. [MT3620 Support Status](#) describes the features that are currently supported. In addition, the [MT3620 Hardware User Guide](#) from Mediatek contains a detailed guide to integrating the MT3620 MCU into your own hardware. Please contact Mediatek if you require this document.

## RDB design files

The RDB design files—schematics, layout, and bill of materials—are available for reference in the [Azure Sphere Hardware Designs Git repository](#). The RDB was developed using Altium Designer. The design files therefore include Altium schematic files (extension: .SchDoc), an Altium layout file (extension: .PcbDoc), and an Altium project (extension: .PrjPcb). To assist those who do not use or have access to Altium Designer, PDFs of the design files and Gerber files are also included.

## Purpose of the board

The RDB was designed to facilitate MT3620 connectivity, debugging, and expansion.

- **Connectivity features.** The RDB includes the key elements needed to integrate MT3620 into an electronic device: the MT3620 itself, at least one Wi-Fi antenna, and essential external components including radio frequency (RF) matching, power supply, and signal conditioning. In addition, programmable buttons and LEDs aid customers in testing and debugging their applications. The [MT3620 development board user guide](#) describes the buttons and LEDs, the Wi-Fi antennas, and the power supply. To ensure compatibility with the Microsoft samples, any development board should support these features.
- **Debugging features.** The RDB exposes the MT3620's two 'management' UARTs and two control signals (reset and recovery) over USB in a way that enables the Azure Sphere PC software tools to recognize and interact with them. This USB interface thereby provides functionality for transferring an application to the board, loading a new operating system image, and debugging. The [MCU Programming and Debugging Interface](#) describes how the RDB implements these features and provides additional guidance for those who are designing boards that incorporate the MT3620.
- **Expansion features.** The RDB includes multiple headers to allow additional hardware to be connected, either with jumper wires or with a custom-made shield. In this way it is possible to interface with a bus or connect to sensors, displays, and so forth. The [MT3620 development board user guide](#) includes details about the headers and programmable I/O (PIO) features.

# MCU programming and debugging interface

10/9/2019 • 8 minutes to read

The MT3620 exposes two dedicated UARTs and two control signals (reset and recovery) for use during device provisioning and recovery. In addition, a further dedicated UART and an SWD interface are optionally available for debugging; these are reserved for future use.

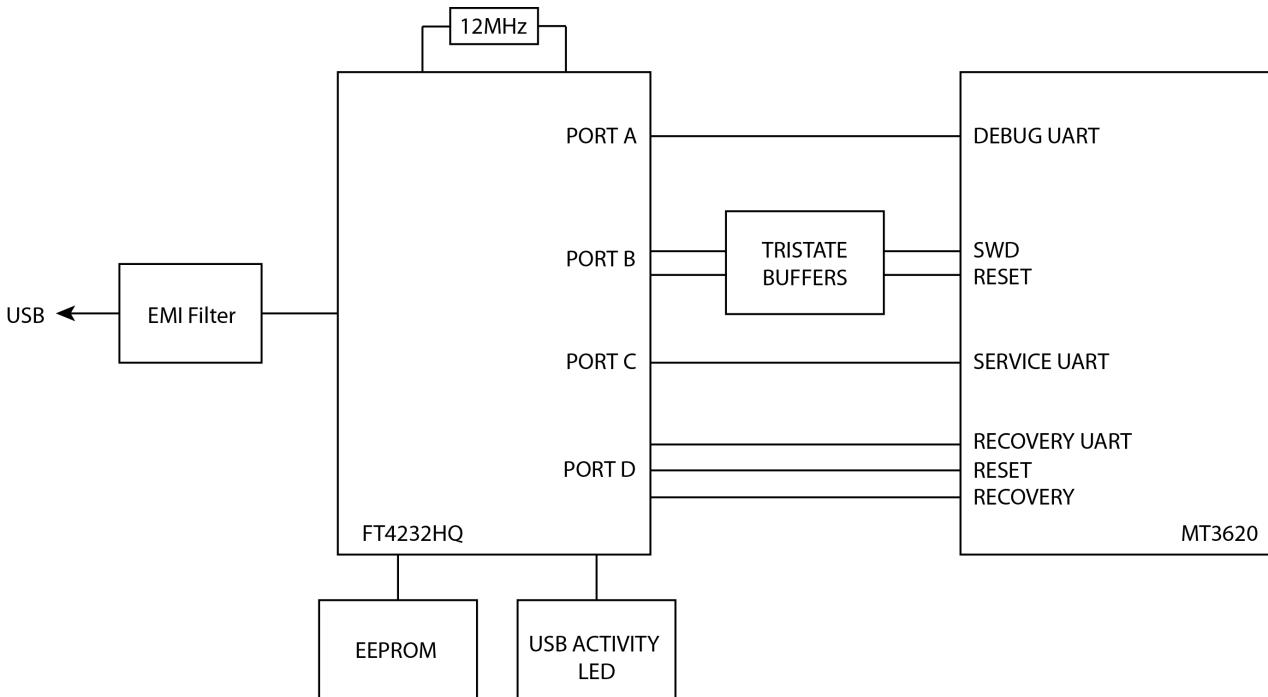
The MT3620 reference development board (RDB) incorporates a USB-to-UART interface chip that exposes these interfaces to a PC in a way that allows the Azure Sphere PC software tools to recognize and interact with them. The Azure Sphere tools include support for loading an application over USB by using the Service UART and for updating the Azure Sphere OS by using the Recovery UART.

If you are designing a custom board or module that uses the MT3620, you'll need to expose these interfaces over USB in the same way, so that you can use the PC tools with your board. Similarly, the manufacture of a connected device requires a solution that enables a UART interface on the chip to communicate with a factory-floor PC that runs the tools.

The PC tools require the use of the [Future Technology Devices International \(FTDI\)](#) FT4232HQ UART-to-USB interface chip to expose the interfaces. Currently, the tools do not support other FTDI chips or interface chips from different manufacturers. The [Azure Sphere hardware designs repository](#) on GitHub contains a design for a stand-alone FTDI-based programming and debugging interface board.

## Overview of components

The following diagram provides an overview of the main components of the 4-port FTDI interface and their interconnections with the MT3620:



You may choose to use the FTDI chip and circuitry as a part of the same board as the MT3620 (for example, if you're building a development board) or in a separate interface board that sits between your MT3620 device and the PC.

## Port assignments

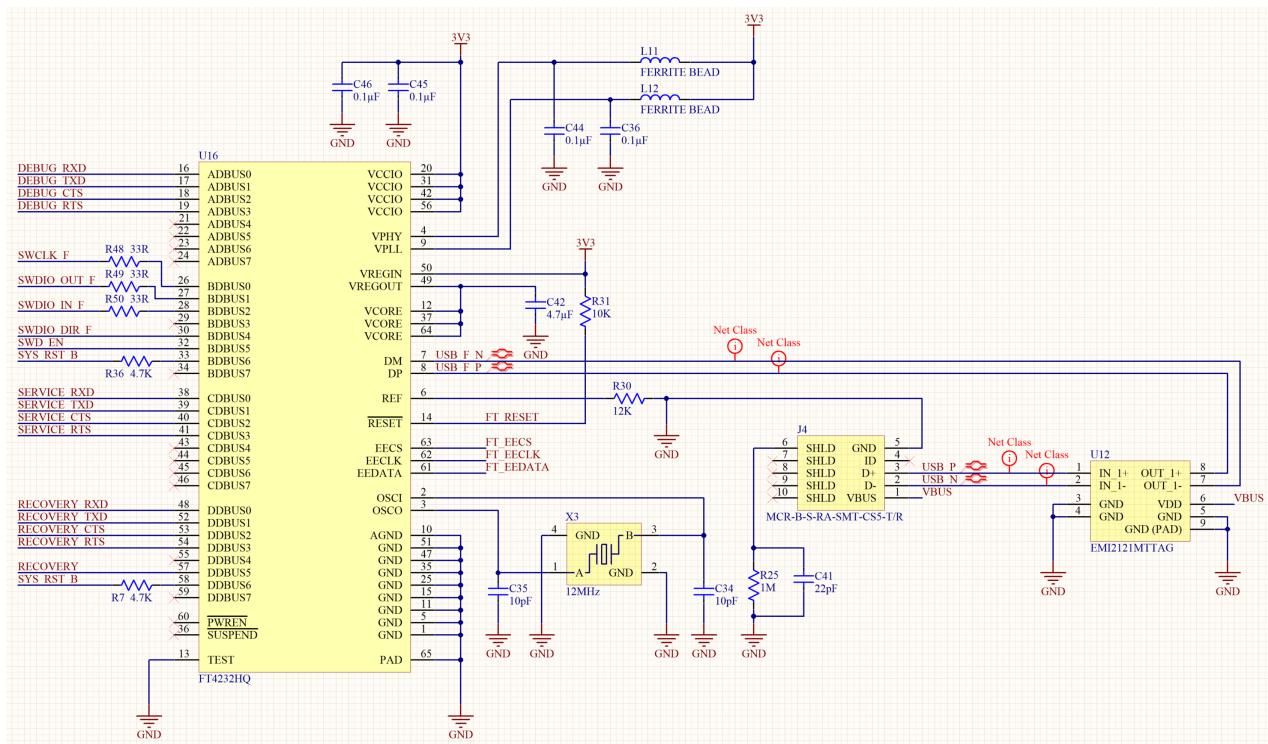
To ensure compatibility with the PC tools, each of the four FTDI ports must be connected to the MT3620 as described in the following table:

FUNCTION	FT4232HQ PIN FUNCTION (PIN NUMBER)	MT3620 PIN FUNCTION (PIN NUMBER)		
Recovery UART, reset and recovery strapping pin	Port-D	DDBUS0 (48)	RECOVERY_RXD (134)	
		DDBUS1 (52)	RECOVERY_TXD (135)	
		DDBUS2 (53)	RECOVERY_CTS (136)	
		DDBUS3 (54)	RECOVERY_RTS (137)	
		DDBUS5 (57)	DEBUG_RTS (96)*	
		DDBUS6 (58)	SYSRST_N (125)	
Service UART	Port-C	CDBUS0 (38)	SERVICE_RXD (129)	
		CDBUS1 (39)	SERVICE_TXD (127)	
		CDBUS2 (40)	SERVICE_CTS (130)	
		CDBUS3 (41)	SERVICE_RTS (128)	
SWD and reset  (Required for debugging real-time capable applications, otherwise optional)	Port-B	BDBUS0 (26)	SWCLK	See <a href="#">SWD interface</a> for details of SWD circuitry
		BDBUS1 (27)	SWDIO out	
		BDBUS2 (28)	SWDIO in	
		BDBUS4 (30)	SWDIO direction	
		BDBUS5 (32)	SWD enable	
		BDBUS6 (33)	SYSRST_N (125)	
Debug UART  (optional, for use as advised by Microsoft)	Port-A	ADBUS0 (16)	DEBUG_RXD (94)	
		ADBUS1 (17)	DEBUG_TXD (95)	
		ADBUS2 (18)	DEBUG_CTS (97)	
		ADBUS3 (19)	DEBUG_RTS (96)*	

\*DEBUG\_RTS is one of the MT3620's 'strapping pins' which, if pulled high during a chip reset, causes the chip to enter recovery mode. At all other times, this pin is the RTS pin of the Debug UART.

## Schematic

The following schematic shows the main components required to support the FT4232HQ chip:



See [MT3620 reference board design](#) for more information about the design files and schematics.

#### Notes:

- The 4.7K resistors in series with the reset line are included to avoid a short-circuit in the event a user presses the reset button (if included in the design).
- When laying out the PCB, ensure that the differential pair, USB\_P and USB\_N, are routed parallel to each other to give a characteristic differential impedance of 90Ω.
- The 33Ω resistors in series with the (optional) SWD lines are intended to reduce transients and should be placed close to the FT4232HQ.

## FTDI EEPROM

The FTDI interface chip provides a set of pins that must be connected to a small EEPROM that is used to store manufacturer's details and a serial number. After board assembly, this information is programmed into the EEPROM over USB using a software tool provided by FTDI, as described later in [FTDI FT\\_PROG Programming Tool](#).

The following EEPROM parts are compatible with the FTDI chip:

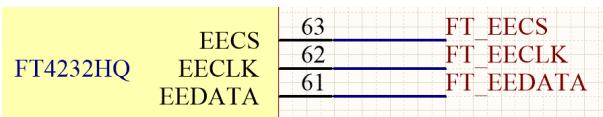
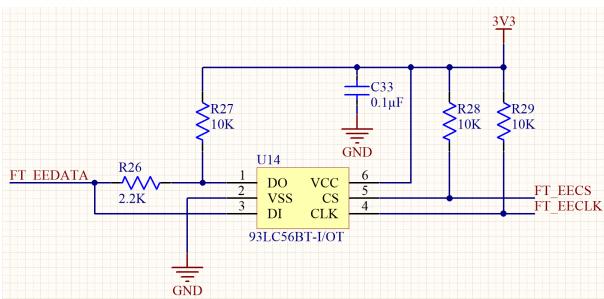
- 93LC46BT-I/OT
- 93LC56BT-I/OT
- 93LC66BT-I/OT

Note the use of the LC variant, which is compatible with a 3.3V supply. For internal development purposes, Microsoft has always used the 93LC56BT-I/OT part.

Connect the EEPROM to the FTDI chip as follows:

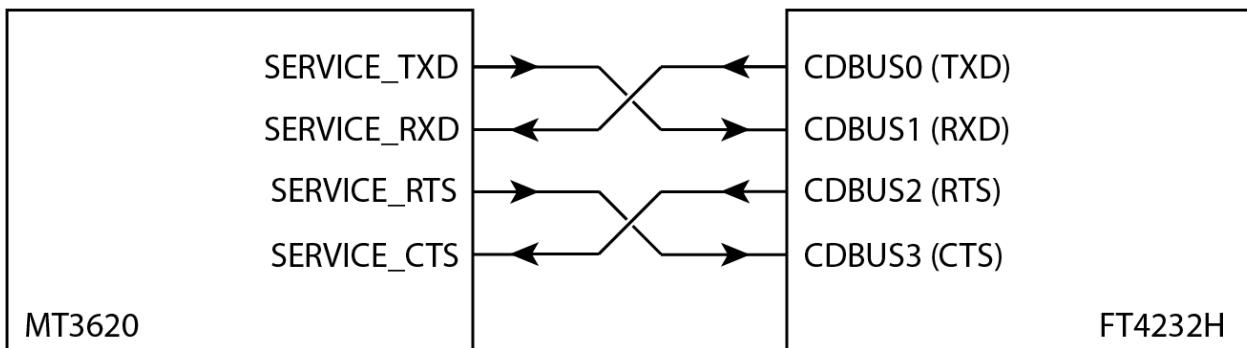
### EEPROM CIRCUIT

### CONNECTION TO FTDI CHIP



## Recovery and Service UART interfaces

The Recovery and Service UART connections between the MT3620 and the FTDI require no special circuitry. However, note the cross-over of TXD and RXD, and CTS and RTS. The FTDI documentation describes pin 0 of each port as TXD and pin 1 as RXD. These definitions are relative to the FTDI chip; that is, pin 0 is an output and pin 1 is an input. Consequently, it is necessary to cross over the RXD and TXD connections to the MT3620 (and similarly for CTS and RTS). The following diagram illustrates this for the Service UART; use the same scheme for the Recovery UART as well:



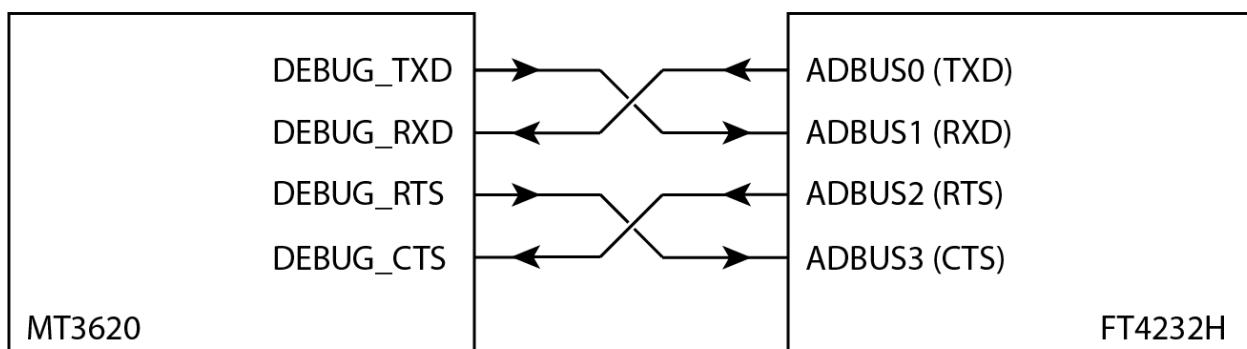
## Debug UART interface (optional)

### NOTE

Not all MT3620 modules expose this UART, and some expose only the output TXD signals.

The Debug UART provides additional debugging capabilities and is reserved for future use. If the UART is available, we recommend that you expose this interface, at least in development and lab environments, to enable easier debugging of any issues that arise.

As with the Recovery and Service UARTs, pay attention to the cross-over of TXD and RXD, and CTS and RTS:



## SWD interface (optional)

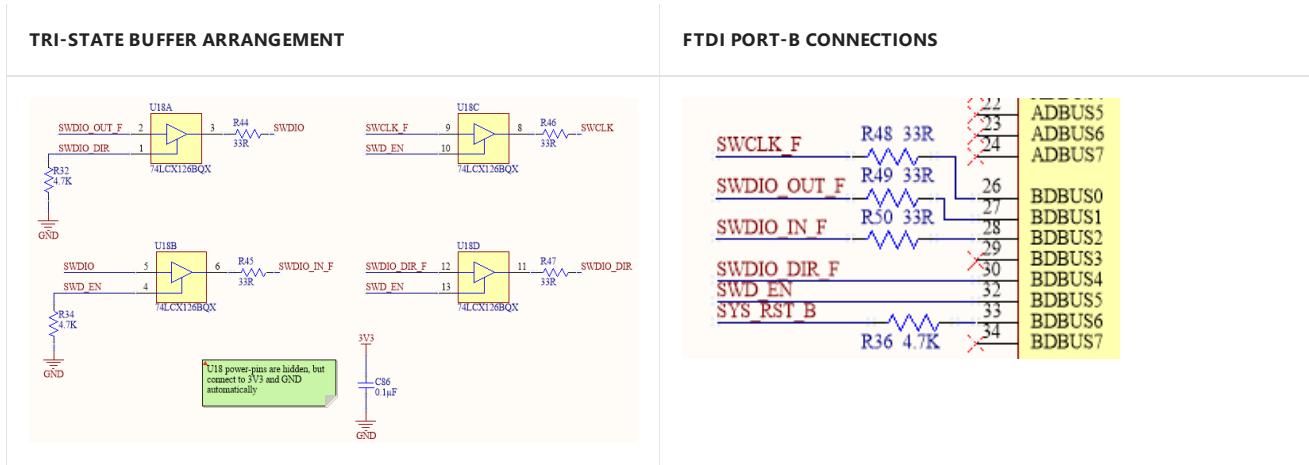
### NOTE

Not all MT3620-based modules expose the SWD interface.

The MT3620 SWD interface is used to debug real-time capable applications. In contrast, debugging high-level applications uses the Service UART rather than SWD. If the device you are creating is intended for general software development (for example, a development board) or if the device will run real-time capable applications, then making SWD available is highly recommended. If the programming interface is intended only for programming devices in the factory, or if the device will not run any real-time capable applications, then connections to the SWD interface are not necessary.

Although FTDI chips are typically used to provide a bridge between UARTs and USB, the Azure Sphere programming and debugging interface uses additional circuitry based on a quad tristate buffer to allow the FTDI part to operate as a high-speed SWD interface.

The following table illustrates the required circuit and connection to the FTDI chip. Note that the SWDIO signal connects to the MT3620 pin 98 and SWCLK connects to pin 99.

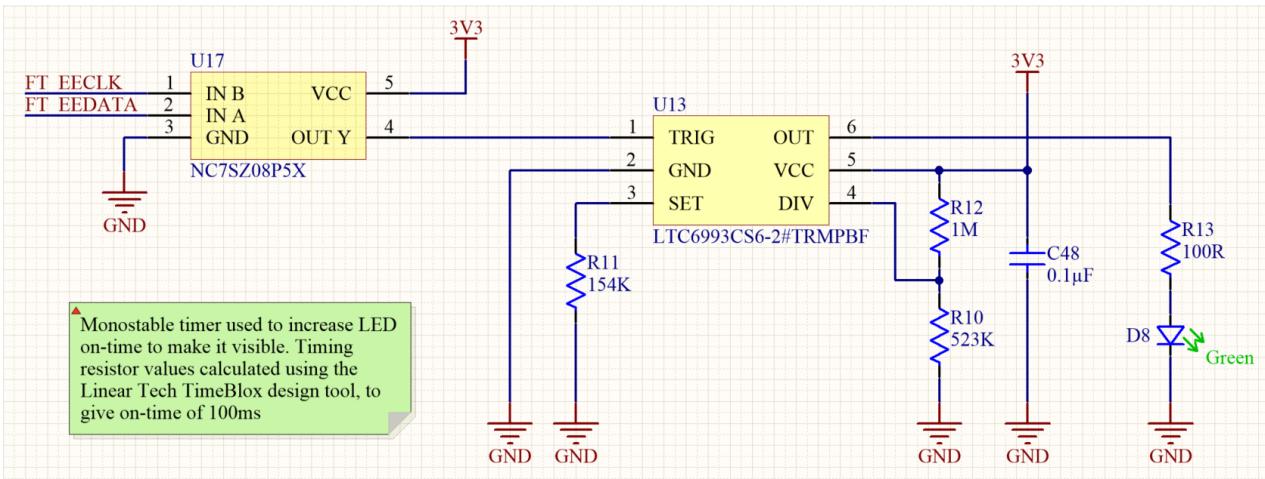


## USB activity LED (optional)

Although optional, a USB activity LED can be useful to indicate data transfer over the USB connection during normal operation. You can implement a USB activity LED in several ways. The following circuit is merely an example.

The circuit ANDs together the clock and data lines that connect the FT4232HQ to the EEPROM. Although not obvious, these two lines toggle when data is being sent and received over USB and therefore can be used to indicate USB activity. However, the on-time of the output from the AND gate is too short to illuminate an LED; therefore, this signal is used to drive a mono-stable circuit, which in turn drives the LED.

The on-time of the mono-stable circuit is set to 100ms, so that even short bursts of USB traffic will cause the LED to illuminate.



## ## FTDI FT\_PROG programming tool

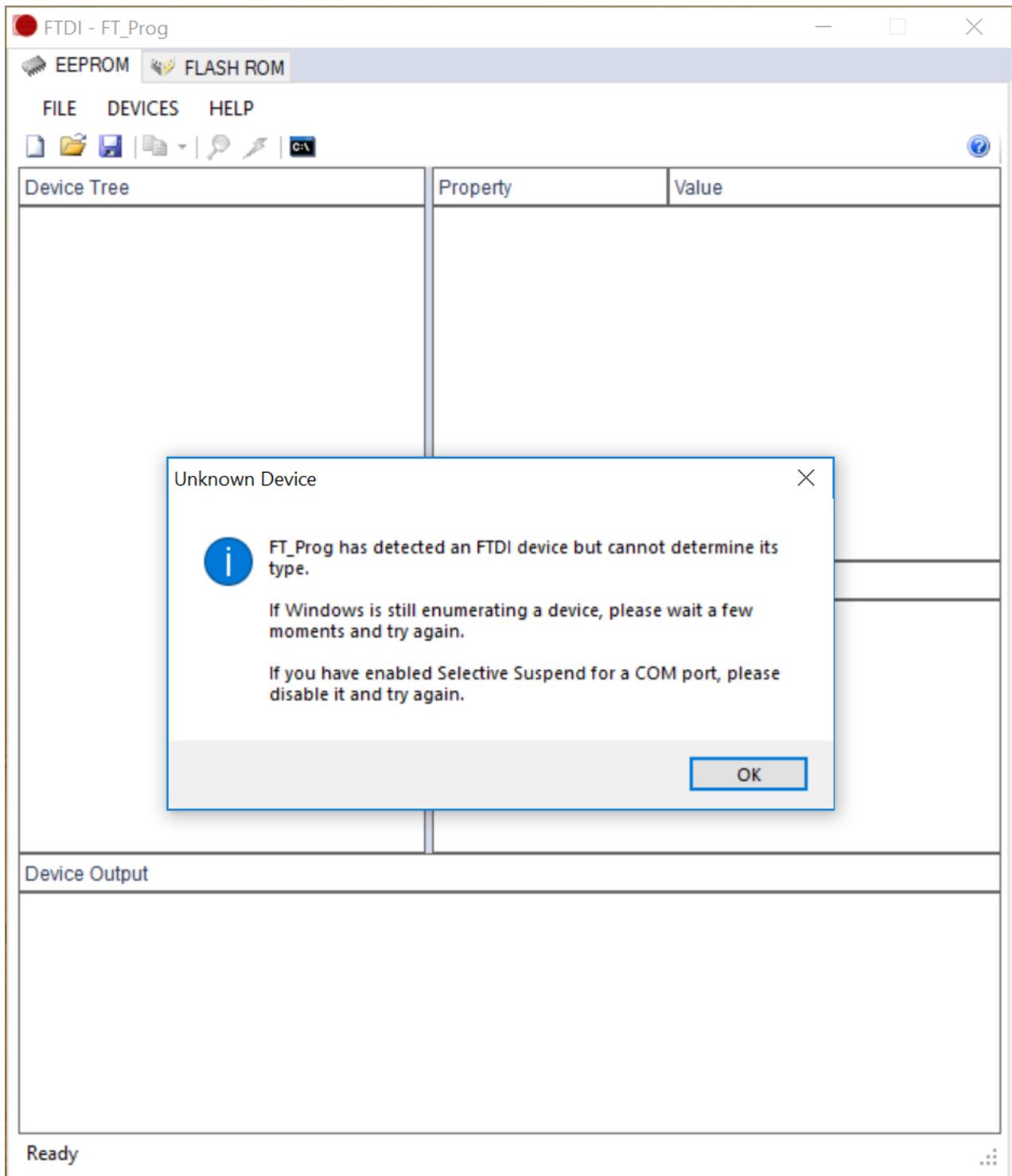
To help in programming the EEPROM, FTDI provides a free software tool called FT\_PROG. The tool is available as both a Windows GUI application and as a command-line tool; both options are installed at the same time from the same package. Download the tool from the [FTDI website](#) and install it in the default location.

### Using the FT\_PROG GUI application

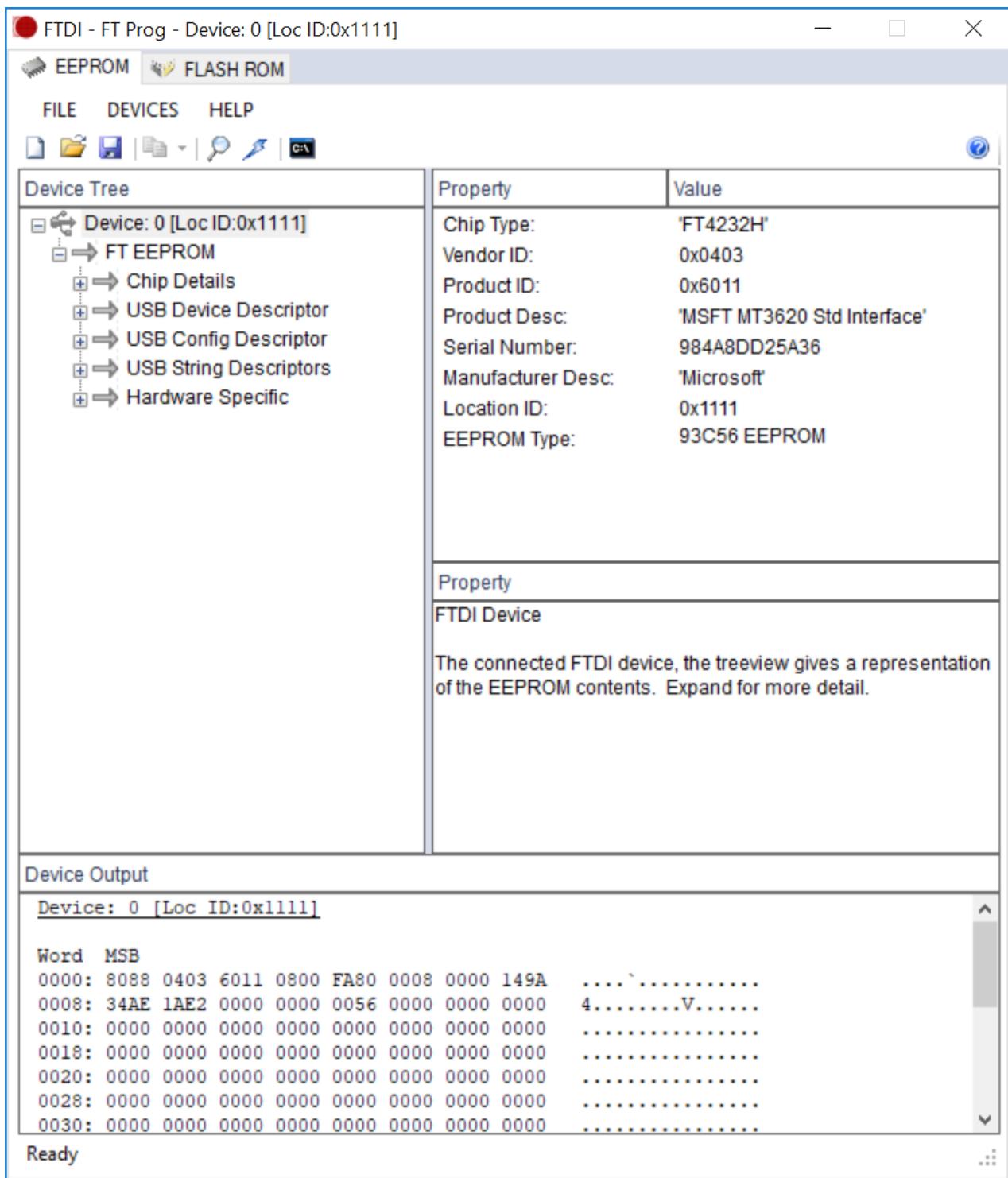
The Windows GUI version of the application is useful for reading and checking the state of the EEPROM information. You can also use it to change the information; however, we recommend the use of the command-line version of the tool to program the device, because you can program the complete configuration with a single command.

After you start the application, click the Scan button (with the magnifying glass icon) to read and display the current contents of the EEPROM.

If an Unknown Device dialog box appears, as in the following example, click **OK** until the application window displays the information correctly.



The following example shows the correct display:



See the FT\_PROG documentation for more information about using the software.

### Using the FT\_PROG command-line tool

The command-line version of FT\_PROG is the preferred method of programming the EEPROM, because it takes the name of a configuration file as a parameter and then programs the device with a single command.

The [EEPROM configuration file](#) is available in the azure-sphere-hardware-designs repo on GitHub. To program the EEPROM, you must use this file as is without modification, because the Azure Sphere PC tools look for the Product Description string and serial number and will fail if these values are changed. The serial number distinguishes an Azure Sphere programming interface from a stock FTDI device over USB.

### Step-by-step instructions for EEPROM programming

To use the command-line version of FT\_PROG to program the EEPROM for a four-port FTDI chip:

1. Install the FTDI tools in the default location: C:\Program Files(x86)\FTDI\FT\_Prog\

2. Ensure that the MT3620 board is the only development board attached to the host PC.
3. Open a command prompt (for example, cmd.exe or an Azure Sphere Developer Command Prompt) and change to the folder where you saved the configuration file.
4. Type the following command:

```
"c:\Program Files (x86)\FTDI\FT_Prog\FT_Prog-CmdLine.exe" scan prog 0 MT3620_Standard_Interface.xml cycl  
0
```

The tool should display:

```
Scanning for devices...  
Device 0: FT4232H, USB <-> Serial Converter  
Device 0 programmed successfully!  
Finished
```

5. To verify that programming was successful, enter:

```
"c:\Program Files (x86)\FTDI\FT_Prog\FT_Prog-CmdLine.exe" scan
```

The output should be:

```
Scanning for devices...  
Device 0: FT4232H, MT3620 Standard Interface, 984A8DD25A36
```

# RF test tools

6/20/2019 • 13 minutes to read

The Radio Frequency (RF) tools enable low-level control of the radio, as required during design verification and manufacturing of hardware based on Azure Sphere. The tools include interactive applications for control and display of the RF settings. Microsoft supplies the RF Tools package upon request.

If you are designing a board or module that incorporates an MT3620 chip, you must test and calibrate the radio before you ship the board or module. If you are manufacturing a connected device that includes a board or module from another supplier, the supplier should already have performed RF testing; check with your supplier if you have any questions.

[Manufacturing connected devices](#) includes information on how RF testing fits into the manufacturing workflow.

## Setup and installation

Before you can run the RF tools, you must set up your PC and your MT3620 device with the latest software and unzip the tools, as described in the following sections.

### PC setup

Set up your PC with the current Azure Sphere SDK.

If you plan to use this PC only for manufacturing and RF testing—and not for Azure Sphere application development or compilation—you can use the `Azure_Sphere_Core_SDK_Preview.exe` installer that is included with the Factory Tools package. It installs only the Azure Sphere command-line tools and SDK and does not require Visual Studio.

### MT3620 device setup

After you set up your PC, make sure that your MT3620 device is running the most recent Azure Sphere OS. Follow the instructions in the [Release Notes](#) for the current release.

### RF tool installation

Unzip the RF Tools Package into a directory on your PC. The resulting RF Testing Tools folder contains three subfolders:

- Configurations, which contains files to facilitate radio configuration settings
- Libraries, which contains the C libraries for performing RF testing
- RfToolCLI, which contains the interactive command-line RF tool and the read-only RfSettingsTool

## MT3620 RF configuration and calibration

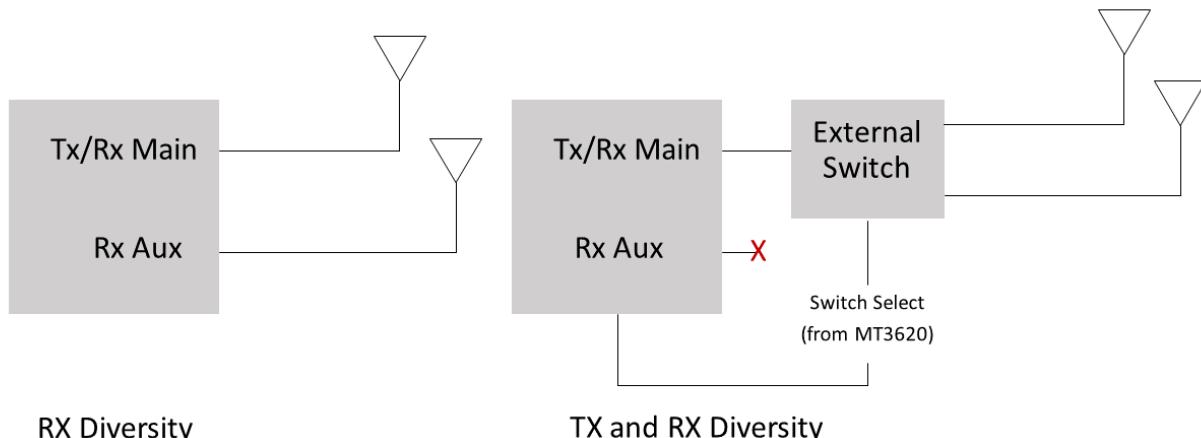
The MT3620 stores radio configuration and calibration data in e-fuses, which can be programmed a limited number of times. This data includes the radio bands (such as 2.4GHz or 5GHz) that the chip should support, adjustments to transmit power, and the antenna configuration on the device. For detailed information about e-fuse configuration, see the MT3620 N9 E-fuse content guidelines, which are available from MediaTek.

### Antenna diversity

Radio signals bounce off objects in the environment. As a result, a single radio signal takes multiple paths from transmitter to receiver. Because these radio signals travel different distances, they arrive at the receiver at different times. Occasionally, the arriving signals interfere destructively and the antenna does not see any signal. One way

to address this problem is through *antenna diversity*. To provide antenna diversity, a second antenna that has a different orientation is placed a short distance (at least a quarter wavelength) away from the first.

The MT3620 supports two antenna diversity configurations, which are configured using radio e-fuses. The figure shows the two configurations.



The configuration on the left shows *receive diversity* (RX diversity). In this configuration, a second antenna is attached to the auxiliary antenna port. If the received signal level on the main antenna port drops below a certain threshold, the MT3620 automatically switches to the second antenna when receiving data. In this configuration, transmissions must still use the primary antenna.

The configuration on the right shows *transmit and receive diversity* (TX and RX diversity), uses the secondary antenna to both transmit and receive. The MT3620 achieves this through the use of an external double-pole, double-throw (DPDT) switch, which allows the signal to be routed to either antenna. In the transmit and receive diversity configuration, the auxiliary antenna port is unused. The MT3620 has two dedicated antenna selection pins for controlling this external switch.

### Buffer bins

During RF testing, the MT3620 can use values in volatile memory instead of the permanent e-fuses, so that test operators and equipment can adjust these settings without permanently changing the e-fuses. The volatile memory used to store these settings is referred to as the "buffer bin." After the test operator or equipment is sure that the values in the buffer bin are correct, the state of the buffer bin can be permanently written to e-fuses.

When entering RF test mode, it is possible to set the contents of the buffer bin to known, pre-set values by loading a "default buffer bin" file. The test operator or equipment can then set additional configuration or calibration values as necessary.

The RF Tools package provides several default buffer bin files in the Configurations\MT3620 directory. These files can be used to initialize the device to a pre-configured state or to override any calibration settings that have previously been programmed into the permanent e-fuses on the device under test (DUT).

The following buffer bin files support transmission with the main antenna:

- MT3620\_eFuse\_N9\_V5\_20180321\_24G\_5G\_NoDpdt.bin sets the radio to support both 2.4GHz and 5GHz operation.
- MT3620\_eFuse\_N9\_V5\_20180321\_24G\_Only\_NoDpdt.bin sets the radio to support 2.4GHz operation only.

The following buffer bin files support transmitting with an auxiliary antenna:

- MT3620\_eFuse\_N9\_V5\_20180321\_24G\_5G\_Dpdt.bin supports 2.4GHz and 5GHz operation with the DPDT switch.

- MT3620\_eFuse\_N9\_V5\_20180321\_24G\_Only\_Dpdt.bin supports 2.4GHz operation with the DPDT switch.

Default buffer bin files can be further customized to your specific device application. Please contact Mediatek or Microsoft for other customization options.

## RfToolCli

RfToolCli is an interactive command-line tool that allows low-level control of the MT3620 radio for testing and diagnostic purposes. Before you run this tool, ensure that the device under test (DUT) is connected and is running the latest Azure Sphere OS.

To use the tool, open a Command prompt window, go to the directory that contains RfToolCli.exe, and run RfToolCli. The command has two start-up options:

```
rftoolcli [-BufferBin <filename>] [-Image <filename>]
```

The -BufferBin option passes the path to a custom default buffer-bin configuration file. By default, RfToolCli uses radio settings that are programmed onto the device. These settings include any transmit power adjustments, allowed frequency bands, and antenna configurations. To use an alternative settings file, supply the path to the file with the -BufferBin option.

The -Image option passes the path to the rftest-server.imagepackage file. This image package file must be loaded onto the DUT to put the device into RF test mode. The rftest-server is provided in the same folder as the RfToolCli executable and in most circumstances RfToolCli can locate this file. If you are running RfToolCli from a different location, you might need to use the -Image option to pass the path to this file.

At startup, RfToolCli prepares the device and then displays an interactive prompt:

```
C:\RF\RfToolCli> .\RfToolCli.exe
Preparing DUT...
>
```

**RFToolCli** provides the commands listed in the following table.

COMMAND (ABBREVIATION) OPTIONS	DESCRIPTION
<b>antenna</b> {aux   main}	Selects the auxiliary or main antenna.
<b>channel</b> <i>number</i>	Selects a channel.
<b>config read</b> {macaddress   data}	Gets device MAC address and buffer bin data.
<b>config write</b> {macaddress   data}	Sets device MAC address and buffer bin data.
<b>config save</b>	Saves changes to MAC address or buffer bin data to permanent e-fuses.
<b>exit</b>	Exits from the program.
<b>help</b> <i>command-name</i>	Displays help on a command.

COMMAND (ABBREVIATION) OPTIONS	DESCRIPTION
<b>receive (rx)</b> {start   stop   stats}	Starts or stops receiving, or displays statistics about received packets.
<b>settings</b>	Displays current radio settings.
<b>showchannel (sc)</b>	Lists the channels that the device supports.
<b>transmit (tx)</b> {frame   mode   power   rate   start}	<p>Configures and transmits packets.            The frame, mode, power, and rate options configure the packets; each has parameters that define the relevant configuration setting.            The start option starts transmission.</p>

You can get help for any command by typing help followed by the command name and, if applicable, an option. For example:

```
help transmit frame
Usage:
Transmit Frame [-BSS <Str>] [-Destination <Str>] [-Duration
<UInt16>]
[-FrameControl <UInt16>] [-Source <Str>]
Configure transmit frame header
Optional Parameters:
-BSS <Str> - BSS MAC address (in colon-delimited format)
-Destination <Str> - Destination MAC address (in colon-delimited
format)
-Duration <UInt16> - Frame duration [Alias: -D]
-FrameControl <UInt16> - Frame Control Number [Alias: -F]
-Source <Str> - Source MAC address (in colon-delimited format)
```

### Example: View start-up settings

At startup, RfToolCli sets several defaults including transmit modes, data rate and channel. To view these startup settings, use the **settings** command.

```
> settings
-----Radio-----
Mode: Normal
Power: 16.0
Channel: 1
Rate: Ofdm54M

---TX Frame Header---

Frame Control: 8000
Duration: 2000
BSS MAC: 62:45:8D:72:06:18
Source MAC: AC:AC:AC:AC:AC:AC
Destination MAC: 62:45:8D:72:06:18

---TX Frame Data---

Frame Size: 1000
Use Random Data: True
```

### Example: Set channel and get received packet statistics

This command sequence puts the radio into receive mode on the specified 802.11 channel and then gets statistics

about the packets received:

```
> channel 9
Setting channel to 9
> rx start
Starting receive
> rx stats
Total packets received: 2578
Data packets received: 4
Unicast packets received: 0
Other packets received: 4
>
```

#### **Example: Transmit packets on current channel**

This command causes the radio to transmit packets on the current channel:

```
> transmit start
Starting transmit
Press any key to stop transmission
```

#### **Example: Transmit packets in continuous mode on current channel**

This command causes the radio to transmit packets on the current channel in continuous mode until you stop transmission or set a different the mode:

```
> tx mode continuous
> tx start
Starting transmit
Press any key to stop transmission
```

When the device transmits in continuous mode, there is no gap between packets, which is useful for power measurements.

#### **Example: Transmit a continuous tone on the current channel**

This command sequence causes the radio to transmit a tone on the current channel until you press a key.

```
> tx continuouswave
> tx start
Starting transmit
Press any key to stop transmission
```

#### **Example: Get the device's currently configured MAC address**

This command reads the currently configured MAC address on the device.

```
> config read MacAddress
Device MAC address: 4E:FB:C4:1C:4F:0C
```

#### **Example: Set the device's MAC address**

This command writes a new MAC address to the device's buffer bin. If a MAC address is already set for the device, you will be asked to confirm the change.

```
> config write MacAddress 02:12:ab:cd:ef:11
Device already has MAC address 4E:FB:C4:1C:4F:0C
Are you sure you want to modify this? (y/N):y
```

#### NOTE

To make buffer bin or MAC addresses changes permanent, use the **config save** command.

### Example: Set one byte of configuration data

The config write data command can be used to set one byte of data at the specified buffer bin address.

```
> config write data 0x34 0xDD
```

### Example: Display device configuration data

The config read data command outputs the entire contents of the device buffer bin.

```
> config read data
Current configuration data:
0x0000: 20 36 04 00 B2 EE D2 16 E5 73 00 00 00 00 00 00 00 00
0x0010: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0030: 00 00 00 00 00 00 00 FF FF 20 00 60 00 CC 00
...
...
```

### Example: Save configuration data to e-fuses

The config save command permanently writes any changes to the buffer bin to the non-volatile e-fuses. The e-fuses can only be written a limited number of times, so we strongly recommend that you perform all buffer bin changes first and then write these changes to e-fuses in a single step.

```
> config save
About to commit data to non-volatile storage
Changes will be permanent. Continue? (y/N):y
Done
```

## RF settings tool

The RF Settings Tool displays MT3620 e-fuse settings so that you can validate that they have been set correctly. Unlike RfToolCli, the RF Settings tool is read-only. Therefore, it can be used to inspect device settings even after radio testing functionality has been disabled on a particular device.

To use the tool, open a Command prompt window, go to the RfToolCli folder, and run RfSettingsTool. The tool has two commands and has two start-up options:

```
rfsettingstool <command> [--image <filename>] [--usefile <filename>]
```

The following commands are supported:

COMMAND (ABBREVIATION)	DESCRIPTION
<b>check (c)</b>	Validates MT3620 device configuration data
<b>help (?)</b>	Shows help information
<b>show (s)</b>	Shows MT3620 configuration data.

### RfSettingsTool check command

The RfSettingsTool **check** command reads the configuration from the attached device and compares it against a buffer bin configuration file that contains the expected settings. The **check** command has the following format:

```
rfSettingsTool.exe check --expected <filename> [--image <filename>] [--nomacaddress] [--showconfig] [--usefile <filename>] [--verbose]
```

PARAMETERS (ABBREVIATION)	DESCRIPTION
--expected <i>filename</i> (-e)	Path to the buffer bin file that contains the expected e-fuse settings to check against. Required.
--image <i>filename</i> (-i)	Path to RF test image. If omitted, defaults to rftest-server.imagepackage. Optional.
--nomacaddress (-n)	Indicates that no MAC address should be set on the device. Optional.
--showconfig (-s)	Shows device configuration after check. Optional.
--usefile <i>filename</i> (-u)	Reads configuration data from the specified file instead of the attached device. Optional.
--verbose (-v)	Shows extra output information.

For example, the following command verifies that the radio setting match those in the specified buffer bin file:

```
> RfSettingsTool.exe check --expected ..\Configurations\MT3620\
MT3620_eFuse_N9_V5_20180321_24G_5G_DPDT.bin
```

In response to this command, RfSettingsTool checks the following items. All must be true for the command to succeed:

- Region code is identical to expected setting
- External antenna switch present identical to expected setting
- Antenna configuration identical to expected setting
- Target power identical to expected setting
- Operating bands identical to expected setting
- MAC address has been set

Radio power offsets, which are device-specific, are not checked.

### RfSettingsTool show command

The RfSettingsTool **show** command displays the radio settings that have been set on the MT3620 e-fuses in a human-readable way. The fields displayed are the user-configurable radio settings. The **check** command has the following format:

```
rfSettingsTool.exe show [--hexdump] [--image <filename>] [--usefile <filename>] [--verbose]
```

PARAMETERS (ABBREVIATION)	DESCRIPTION
--hexdump (-x)	Shows the raw hexadecimal contents of e-fuses. Optional.
--image <i>filename</i> (-i)	Path to RF test image. If omitted, defaults to rftest-server.imagepackage. Optional.
--usefile <i>filename</i> (-u)	Reads configuration data from the specified file instead of the attached device. Optional.
--verbose (-v)	Shows extra output information.

The following example shows partial output from the **show** command:

```
> RfSettingsTool.exe show
Reading configuration data from device.
-----
MAC Address : C6:76:EC:79:1D:6B
-----
Region : GB
-----
External RF switch : Present
2.4GHz Diversity : MainOnly
5GHz Diversity : MainOnly
.
.
.
```

## RF test C library

The RF Tools package includes a C library that you can use to develop your own test programs. The C library is in the libraries\C directory. Header files for the C API are available in the libraries\C\Include folder, and binary files that are required to use the library are provided in the libraries\C\Bin folder. If you want to use the library, please contact Microsoft for documentation.

The RF testing server image (rftest-server.imagepackage) is also provided in the Bin folder. This image must be loaded on the device under test before the device can enter RF testing mode. The **mt3620rf\_load\_rf\_test\_server\_image()** function in the C library loads the image package programmatically.

If you redistribute an application that uses the C library, you must include the DLL files from libraries\C\Bin as well as the rftest-server.imagepackage file.

## Compatibility of RF Testing Tools across OS versions

There is no guarantee that RF Testing Tools for one OS release will be compatible across all OS versions. Generally, we recommend that you use the version of the RF Testing Tools (and associated C Library) that is issued with the manufacturing package for the OS running on the device under test. However, to determine if your current RF Testing Tools are compatible with the OS running on the device, refer to the table in this section.

OS VERSION	18.11 RF TEST TOOLS/API	19.02 RF TEST TOOLS/API	19.05 RF TEST TOOLS/API
18.11	Yes	Yes	Yes
19.02	Yes	Yes	Yes

OS VERSION	18.11 RF TEST TOOLS/API	19.02 RF TEST TOOLS/API	19.05 RF TEST TOOLS/API
19.05	Yes	No	Yes

## Errata

- The MT3620 Wi-Fi firmware has a minor bug:

If you switch to Continuous mode transmission (tx mode continuous) and start transmission (tx start) immediately after stopping a Normal mode transmission, there will be no signal output.

To work around this, stop the Continuous mode transmission and start it again for the transmission to commence. After this, Continuous mode transmission will work correctly.

The problem does not occur when switching from Continuous Mode to Normal mode.

- When switching from Continuous Wave transmission mode to Normal or Continuous transmission modes, transmit power will incorrectly increase by +6 dB. You must re-initialize the radio to return the power level to normal.
  - If using the RfToolCli interactive tool, reinitialize the radio by exiting and then and restarting the tool.
  - If using the C API, call the `mt3620_reinitialize_buffer_bin()` function. This will also reinitialize the radio and can be used to work around this issue.

# Guardian modules

10/9/2019 • 5 minutes to read

A *guardian module* is add-on hardware that incorporates an Azure Sphere chip and physically attaches to a port on a "brownfield" device—that is, an existing device that may already be in use. By using a guardian module, you can add secure IoT capabilities to equipment that either doesn't support internet connectivity or doesn't support it securely. In short, a guardian module provides a way to implement secure connectivity in existing devices without exposing those devices to the internet. Because it's an Azure Sphere device, all the Azure Sphere security and connectivity features are available: all data is encrypted, OS and application updates are delivered securely, and authentication ensures that the module communicates only with trusted hosts.

Here's how a guardian module works:

- The guardian module connects to a brownfield device through an existing peripheral on the device, and connects to the internet through Wi-Fi or Ethernet. The brownfield device itself is not connected to the network.
- The Azure Sphere OS runs on the guardian module along with a custom high-level application and any other [Azure Sphere applications](#) your scenario requires.
- The guardian module uses the Azure Sphere Security Service for certificate-based authentication, failure reporting, and over-the-air software updates.
- The brownfield device communicates with the guardian module, which can respond by taking a local action or by reporting to a cloud presence such as Azure IoT Central.

Every guardian module must have a [high-level Azure Sphere application](#) and may also have [real-time capable applications](#).

You can buy guardian modules from a vendor and further customize them for your usage scenario, or you can design your own guardian module, possibly working with a hardware partner. See the [Azure Sphere website](#) for information about hardware suppliers.

## Uses for a guardian module

A guardian module can do anything any other Azure Sphere device can do. Possible usage scenarios for a guardian module include:

- Cull data from the brownfield device, process it, and transmit it securely to a cloud endpoint
- Send data to multiple endpoints, provided that it can authenticate each endpoint
- Gather additional data that's not available from the brownfield device; for example, sensors on the guardian module could provide environmental data for use with operating data from the brownfield device
- Save data from the brownfield device in case connectivity is lost

## High-level applications for guardian modules

A guardian module requires at least one [high-level application](#). The high-level application communicates upstream with the internet (including the Azure Sphere Security Service and other cloud services) and downstream with the brownfield device. These connections, along with other device- and application-specific details, must be listed in the [application manifest](#). As a result, the high-level application must be custom-written or customized for each organization's brownfield devices. If your guardian module supplier provides an application, be sure that you receive the high-level application source code and libraries so that you can update the application as necessary.

A high-level application that runs on a guardian module is responsible for:

- Establishing and maintaining downstream connectivity with the brownfield equipment and upstream connectivity with the internet
- Handling data sent upstream from the brownfield device, unpacking and storing if necessary, and communicating with the internet hosts as appropriate
- Handling data sent from an internet host, unpacking and storing if necessary, and communicating with the brownfield equipment as appropriate

The application can [connect and authenticate to web servers](#) and use mutual authentication for such connections. Data sent upstream might include error reports, operating parameters, or overall telemetry. Azure Sphere ensures that all such data are encrypted.

Data sent downstream might include updated software or changes to settings or parameters for the brownfield device. To avoid potential security breaches, the app should validate incoming data before passing it downstream to the brownfield device.

## Connectivity

For upstream connections from the guardian module to the internet, Azure Sphere currently supports wired [Ethernet](#) and [Wi-Fi](#).

For downstream connections from the guardian module to the brownfield equipment, you can use any peripheral that the guardian module exposes. [Private Ethernet support](#) in the Azure Sphere OS allows a connection to the brownfield device over Ethernet without exposing it to the public internet.

## Peripheral device support

Like other Azure Sphere devices, guardian modules differ in the peripherals that they expose. Choose a guardian module that provides the connectivity and sensing capabilities your scenario requires. Depending on the hardware architecture of the guardian module—that is, how it exposes capabilities of the Azure Sphere chip—you can determine whether the software to access individual features must be implemented as a high-level or a real-time capable application.

## Storage considerations

Azure Sphere has [limited storage](#), so carefully consider how much data will be exchanged between the guardian module and the brownfield device, and between the guardian module and the cloud.

Azure Sphere applications can use up to 256 KiB of RAM. If you plan to send data downstream from the cloud to the brownfield device, ensure that the guardian module has enough space to hold the data. For example, if you want to deliver firmware updates for the brownfield device, make sure they will fit in the storage available on the guardian module. You might need to send such updates in "chunks"; the [HTTPS\\_Curl\\_Multi sample](#) in the Azure Sphere GitHub samples repo shows how to do this. Keep in mind that your downstream device will also need a similar mechanism.

For data that travels upstream (from the brownfield device to the guardian module), ensure that your application can handle upstream connectivity failures. If the brownfield device provides ongoing telemetry, you need to consider which data and how much of it to retain and later transmit when connectivity is restored.

## App development and deployment

Developing an application for a guardian module is no different from developing an application for any other Azure Sphere device. You'll need access to the service UART on the Azure Sphere chip so that you can read and write from the local device. If you design your own device, you will need to ensure that the service UART signals are exposed and that you support a way to [interface with the service UART](#) either on the guardian module itself or on a separate piece of hardware. If you purchase modules from a vendor, the vendor should provide a solution that enables this connection.

If your supplier or another third party will create the application, you might need to provide access to your Azure Sphere tenant so that the application developer can load and test the application and [create a deployment](#). If your module provides a UART service connection, you can sideload the production app before final deployment. You will need to [claim](#) the device afterwards.

# Manufacturing connected devices

12/12/2018 • 2 minutes to read

This topic contains information about hardware, testing, and manufacturing tools for use with Azure Sphere. It is intended for manufacturing engineers who are responsible for the volume manufacture and test of hardware based on Azure Sphere.

Manufacture of a connected device that incorporates Azure Sphere technology involves the following types of tasks:

- [Factory-floor tasks](#) to update, test, calibrate, and load software onto the Azure Sphere chip, and to prepare the connected device for shipping. Factory-floor tasks may require each individual chip to connect to a PC but do not usually require an internet connection.
- [Cloud configuration tasks](#) to configure the connected devices for over-the-air update. These tasks require internet connectivity, but do not require the individual Azure Sphere chips to connect to the internet or to a PC.

# Factory floor operations

8/9/2019 • 13 minutes to read

Manufacturing connected devices that incorporate Azure Sphere hardware involves several factory-floor operations:

- Connecting each Azure Sphere chip to a factory floor PC
- Recording device IDs
- Updating the Azure Sphere OS if necessary
- Loading software
- Running functional tests
- Performing radio frequency (RF) testing and calibration, if necessary
- Verifying RF configuration
- Finalizing the device

You must connect the chip to the PC first and finalize the device last, but you can perform other operations in any order that suits your manufacturing environment.

## Connect each Azure Sphere chip to a factory floor PC

During manufacturing, you must connect each Azure Sphere chip to a factory floor PC. The PC must be running Windows 10 v1607 (Anniversary update) or a more recent release.

The Azure Sphere tools run on the PC and interact with the chip over a chip-to-PC interface. You choose how to implement this interface:

- Design an interface board that connects to your PC during manufacturing.
- Build an interface into each connected device. For example, the MT3620 reference board design (RDB) includes such an interface.

The [MCU programming and debugging interface](#) provides details on the design and requirements for the chip-to-PC interface.



The Factory Tools package includes the Azure Sphere Core SDK Preview, which provides the utilities you'll need during manufacturing but does not require Visual Studio. To install the Core SDK, unzip the package, click `Azure_Sphere_Core_SDK_Preview.exe`, accept the license agreement, and click **Install**.

However, if you plan to use the same PC for both Azure Sphere application development and manufacturing/RF testing, you can instead use the `Azure_Sphere_SDK_Preview_for_Visual_Studio.exe` installer, which installs the Visual Studio tools in addition to the command-line tools and SDK. See [Install Azure Sphere](#) for instructions.

You can simultaneously connect as many Azure Sphere devices to your PC as the PC's USB subsystem will support. The Azure Sphere tools do not limit the number of devices that can be connected at one time.

#### NOTE

Support for multiple attached devices is provided only by the **azsphere** CLI. The Visual Studio Extension for Azure Sphere currently supports deployment and debugging of only a single device (the first device), which must have IP address 192.168.35.2.

The **azsphere** command supports several features that enable you to gather information about all the devices that are attached to a PC and to perform operations when multiple devices are connected.

#### Get information about attached devices

You can gather information about all attached Azure Sphere devices by using the following command:

```
azsphere device list-attached
```

This command returns the IP address for each Azure Sphere device and a value that identifies the device's USB connection. If the device is responsive, the command also returns the device ID. For example, the following shows output for two devices:

```
2 devices attached:  
--> Device ID: <deviceID>  
    --> Is responsive: yes  
    --> IP address: 192.168.35.2  
    --> Connection path: 11433  
  
--> Device ID: <deviceID>  
    --> Is responsive: yes  
    --> IP address: 192.168.35.3  
    --> Connection path: 123  
  
Command completed successfully in 00:00:01.7437655.
```

The IP address is assigned when an FTDI-based device interface is attached to the PC; it does not indicate that a responsive device is present. The IP address persists while the FTDI-based device interface is attached to the PC, even if a different Azure Sphere device is plugged into the interface. After a PC reboot, however, the IP address may change. The first device to be attached is assigned the address 192.168.35.2.

The connection path is an [FTDI location ID](#) that identifies the USB connection. The location persists while the FTDI-based device interface is attached to the same USB port on the same USB hub, and in turn to the same port on the PC. Thus, it persists over reboot. However, any changes in wiring between the PC and the device may result in changes to the connection path. Like the IP address, it doesn't change even if a different Azure Sphere device is plugged into the FTDI interface.

#### Perform operations on attached devices

The **azsphere device** command supports two additional global parameters to identify a device by USB location or IP address:

PARAMETER	DESCRIPTION
--deviceip, -ip <i>IP-address</i>	Identifies a device by its IP address, which is returned by <b>azsphere device list-attached</b>
--devicelocation, -l <i>USB-location</i>	Identifies a device by its connection path, which is returned by <b>azsphere device list-attached</b>

Use either of these parameters to identify the device to which an **azsphere device** command applies. You can specify only one device per command. If more than one device is attached to the PC and you don't specify an IP

address or USB location, the command fails.

Every device is assigned an IP address—even if it's unresponsive—so you can use the IP address to identify a device that requires recovery.

The following table lists the **azsphere** operations with which you can use these two parameters.

COMMAND	OPERATION
<b>device</b>	<b>capability, cap</b>
	<b>claim</b>
	<b>image, img</b>
	<b>list-attached</b>
	<b>link-feed</b>
	<b>manufacturing-state, mfg</b>
	<b>prep-debug</b>
	<b>prep-field</b>
	<b>recover</b>
	<b>restart</b>
	<b>show-attached</b>
	<b>show-ota-config</b>
	<b>show-ota-status</b>
	<b>sideload, sl</b>
	<b>update-device-group</b>
	<b>update-sku</b>
	<b>wifi</b>
<b>tenant</b>	<b>create</b>

## Update the Azure Sphere OS

Every Azure Sphere chip is loaded with the Azure Sphere OS when it is shipped from the silicon manufacturer. Depending on the version of the Azure Sphere OS on chips available from your supplier, and depending on the OS version requirements of your application, you might need to update the Azure Sphere OS during manufacture of the connected device.

If the Azure Sphere chip is not online on the factory floor, you can update it by using the **azsphere device recover** command over the programming and debugging interface described earlier. Use the `--images` parameter to install

specific recovery images.

The Factory Tools package includes a set of recovery files in the folder Standalone SDK\Images. You can obtain the most recently available Azure Sphere OS recovery files for MT3620-based hardware [here](#).

### Record device IDs

As part of the factory-floor process, you should record the device IDs of all Azure Sphere chips that your company incorporates into manufactured devices. To get the device IDs of all attached devices, use **azsphere device list-attached**.

You will need the device IDs during [cloud configuration](#) to set up device groups and deployments.

## Load software

All software that you load—regardless of whether it is a board configuration file, a testing application, or a production application intended for the end user—must be production-signed.



During manufacturing, Azure Sphere devices must not require any special capabilities installed, such as the [appdevelopment capability](#), which enables debugging. Acquiring capabilities for individual devices reduces device security and requires internet connectivity, which is typically undesirable on the factory floor.

### Get production-signed images

The Azure Sphere Security Service production-signs each image when you upload it. To avoid the need for internet connectivity on the line, create the production-signed images once, download them from the Azure Sphere Security Service, and then save them on a factory-floor PC for sideloading during production.

To get a production-signed image, upload it to the Azure Sphere Security Service by using the **azsphere component** command:

```
azsphere component image add --autocreatecomponent --filepath <imagepackage-file>
```

Replace <imagepackage-file> with the name of the image package that contains your software. The Security Service production-signs the image and retains it.

Applications that are intended for use only during factory testing must be explicitly identified as temporary images. This ensures that these applications can be removed at the end of the testing process. To mark an image as temporary, use the --temporary parameter when you upload the file for production signing:

```
azsphere component image add --autocreatecomponent --filepath <imagepackage-file> --temporary
```

Save the component ID that the command displays; you'll need it later to remove the temporary image from the device.

To download the production-signed image, use the following command:

```
azsphere component image download --imageid <image-id> --output <file-path>
```

Replace <image-id> with the ID of the image to download, and replace <file-path> with the filename and path in which to save the downloaded image. The image ID appears in the output of the **azsphere component image add** command.

After you save the production-signed image, no further internet connectivity is necessary.

### Deploy and delete images

To deploy a production-signed image onto a device in the factory, use the **azsphere device sideload** command:

```
azsphere device sideload deploy --imagepackage <file-path> [--deviceip <ip> | --devicelocation <USB location>]
```

Replace <file-path> with the name and path to the downloaded image file. If multiple devices are connected to the PC, include the **--devicelocation** or **--deviceip** parameter to identify the target device. See [Perform operations on attached devices](#) for details about these parameters.

If you load a temporary application for testing, use the following command to delete it after testing is complete:

```
azsphere device sideload delete --componentid <component id> [--deviceip <ip> | --devicelocation <USB location>]
```

## Run functional tests

Functional tests verify that the product operates correctly. The specific tests that you should run depend on your individual hardware.



You can organize your tests as a single OEM application or as a series of applications. The [application development documentation](#), the [Azure Sphere samples](#), and the templates in the Azure Sphere SDK provide information about application design. Whatever design you choose, this application needs to be production-signed and then deployed using the steps in the previous section.

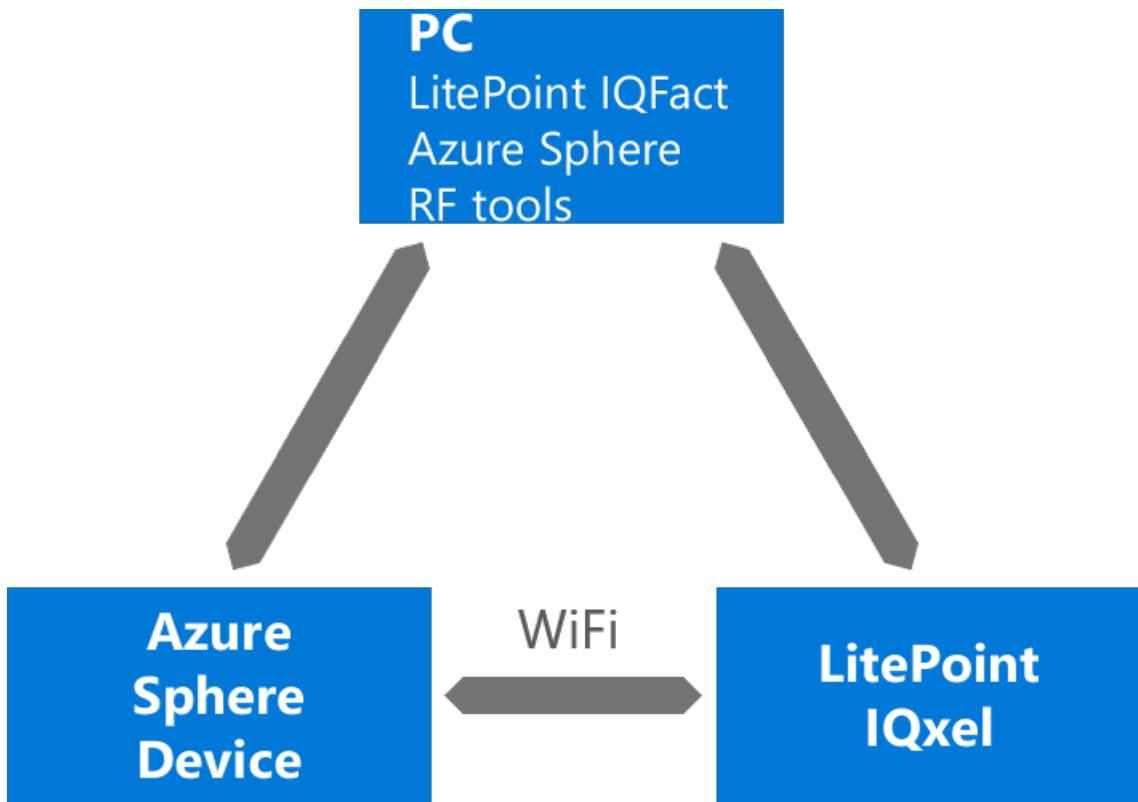
Some testing processes require communication with the chip that is being tested: to report errors, log data, or sequence tests. If your testing process requires communication, you can use the peripheral UARTs on the MT3620 (ISU0, ISU1, ISU2, or ISU3). Connect these UARTs to your factory PC or external test equipment using suitable circuitry of your design. If you created an interface board to support chip-to-PC communication, you might want to add this circuitry to that board.

## Perform RF testing and calibration

Azure Sphere chips require wireless connectivity to receive software updates and communicate with the internet. Testing and calibrating RF operation is therefore a critical part of the manufacturing process. If you are using a module, certain aspects of RF testing might not be required; consult the module supplier for details.

The RF Tools package, available upon request, includes utilities and a C API library for use during testing. Using the library, you can program product-specific RF settings in e-fuses, such as the antenna configuration and frequency, as well as tune individual devices for optimal performance. If your test house needs to use the tool to certify your device, please contact your Microsoft representative before sharing the software with them.

Integration between the library and test equipment is your responsibility. Currently, Microsoft has partnered with [LitePoint](#) to provide a turnkey solution that integrates the Azure Sphere RF test library with the LitePoint equipment. Solutions from other test equipment vendors may become available in the future.



The [RF testing tools](#) topic describes how to use the RF tools.

At the same time as RF and Wi-Fi calibration, consider also connecting to a Wi-Fi access point to verify that your end-user application will be able to communicate over Wi-Fi. Ensure that the Wi-Fi connection does not have internet access, because over-the-air update may occur if the chip connects to an internet-enabled access point. After Wi-Fi testing, you should remove any Wi-Fi access points used for testing from the chip so that it is not visible to customers. For details about Wi-Fi configuration, see [azsphere device wifi](#). Note that device recovery removes all Wi-Fi configuration data from the chip.

## Verify RF configuration

Use the `RfSettingsTool` to verify that the radio configuration options such as target transmit power, region code, and MAC address have been correctly set. The [RF settings tool](#) documentation provides more information about using this tool.

## Finalize the Azure Sphere device

Finalization ensures that the Azure Sphere device is in a secured state and is ready to be shipped to customers. You must finalize the device before you ship it. Finalization involves:

- Locking out RF configuration and calibration tools to prevent security breaches.
- Running ready-to-ship checks to ensure that the correct system software and production application are installed and RF tools are disabled.

### **Set the device manufacturing state to manufacturing complete**

Sensitive manufacturing operations such as placing the radio in test mode and setting Wi-Fi configuration e-fuses should not be accessible to end users of devices that contain an Azure Sphere chip. The *manufacturing state* of the Azure Sphere device restricts access to these sensitive operations.

When chips are shipped from the silicon factory, they are in the "Blank" manufacturing state. Chips in the "Blank" state can enter RF test mode and their e-fuses can be programmed.

When manufacturing is complete, use the following command to set the manufacturing to "AllComplete":

```
azsphere device manufacturing-state update --state AllComplete [--deviceip <ip> | --devicelocation <USB location>]
```

Currently, only the "Blank" and "AllComplete" states are supported. In the future, new states may be introduced to support additional stages in the manufacturing process.

#### IMPORTANT

Moving a chip to the "AllComplete" state is a **permanent** operation and cannot be undone. Once a chip is in the "AllComplete" state, it cannot enter RF test mode and its e-fuse settings cannot be adjusted. If you need to re-enable these capabilities on an individual chip, such as in a failure analysis scenario, please contact Microsoft.

### Run a ready-to-ship check

It is important to run a ready-to-ship check before you ship a product that includes an Azure Sphere device. A typical ready-to-ship check ensures the following:

- Azure Sphere OS is valid
- User-supplied images match the list of expected images
- Device manufacturing state is AllComplete
- No unexpected Wi-Fi networks are configured
- Device does not contain any special capability certificates

Some of the tests in the preceding list may differ if you are designing a module rather than a connected device. For example, as a module manufacturer you might choose to leave the chip in the "Blank" manufacturing state so that the customer of the module can perform additional radio testing and configuration.

The Factory Tools package includes a sample Python script called deviceready.py, which checks all the listed items. The deviceready.py script can be used as-is or modified to suit your needs. It also demonstrates how to run the Azure Sphere CLI tools to perform these device-ready checks programmatically as part of an automated test environment.

If your Azure Sphere SDK is installed in the default location, the deviceready.py script can be run from an Azure Sphere Developer Command Prompt without providing the path to the azsphere.exe executable. If you have installed the SDK in a different location, you will need to supply the path to the azsphere.exe executable with the --binpath command-line argument.

The deviceready.py script takes the following parameters:

- The --os parameter specifies the valid versions of the Azure Sphere OS for the check to succeed. If no OS versions are supplied, this check fails.
- The --images parameter specifies the image IDs that must be present for the check to succeed. If the list of installed image IDs differs from this list, the check fails. By checking image IDs (rather than component IDs) this check guarantees that a specific version of a component is present on the device.
- The --os\_components\_json\_file parameter specifies the path to the JSON file that lists the OS components that define each version of the OS. For MT3620-based devices, this file is named mt3620an.json and is installed with the Factory Tools package in the same folder as the deviceready.py script. You can also download the most recent version of this file from [here](#).
- The --binpath parameter specifies the path to the azsphere.exe utility. Use this parameter only if the Azure Sphere SDK is not installed in the default location.
- The --help parameter shows command-line help.
- The --verbose parameter provides additional output detail.

A sample invocation of the deviceready.py script when running from the same folder as the deviceready.py file looks like the following:

```
> python .\deviceready.py --os 19.02 --images e6ca6889-96d3-4675-bbe5-251e11d02de0
```

# Cloud configuration tasks

2/14/2019 • 4 minutes to read

After the product that contains the Azure Sphere device is finalized but before it is shipped, you must configure the device for over-the-air (OTA) use. Cloud configuration requires the following information:

- The [device ID](#) of each Azure Sphere chip
- The [product SKU](#) for each connected device
- The intended [device group](#) for each connected device

The PC that you use for cloud configuration must be connected to the internet, but it is not required to be connected to each chip.

The following steps are required for cloud configuration:

1. [Claim the chip](#)
2. [Configure over-the-air \(OTA\) software deployments](#)
3. [Verify the OTA configuration for the device](#)

These steps are critical to the continued operation of the device at the customer site.

## Claim the chip

Each Azure Sphere chip has a unique and immutable device identifier, called its *device ID*. The silicon manufacturer creates the device ID, stores it on the chip, and *registers* it with Microsoft. This device registration ensures that Microsoft is aware of all Azure Sphere chips and that only legitimate chips can be used in connected devices. As part of the factory-floor process, you should record the device IDs of all Azure Sphere chips that your company receives.

You must also *claim* the Azure Sphere chips in all your connected devices. *Claiming* involves moving the Azure Sphere chip to your organization's cloud tenant, so that both your organization and Microsoft can identify the chip's owner. Claiming ensures that all data associated with the chip resides in your tenant and is protected by your security policies. A chip must be claimed before it can communicate with Azure Sphere Security Service. Such communication, in turn, allows the chip to receive the software updates that you specify and to obtain certificates that are required for authentication to an Azure IoT Hub and other cloud-based services.

Internet connectivity is not required to obtain device IDs but is required for claiming. You can record the device IDs, store them on the factory floor, and then transfer the IDs to a different computer later for claiming. To claim one or more chips, use the following command in an Azure Sphere Developer Command Prompt:

```
azsphere device claim --deviceid <GUID>
```

Replace <GUID> with the device ID of the chip you want to claim.

### IMPORTANT

You *may* claim the Azure Sphere chip any time during the manufacturing process; the chip need not be incorporated into a connected device at the time of claiming. You *must* claim the Azure Sphere chip before you set up deployments, verify the OTA configuration, and ship the connected device.

## Configure over-the-air deployments

OTA deployments update the Azure Sphere device OS and your production application software.

To receive the correct over-the-air software updates, the Azure Sphere device must have a product SKU and belong to a device group that permits deployments. See [Over-the-air deployment](#) to learn about product SKUs, device groups, and deployments. Assign both the product SKU and the device group before shipping the connected device.

To assign a product SKU, use the following command in an Azure Sphere Developer Command Prompt:

```
azsphere device update-sku --deviceid <GUID> --skuid <SKU>
```

Replace <GUID> with the device ID of the chip and replace <SKU> with the ID of the product SKU for this model of connected device.

To assign a device group, use the following command:

```
azsphere device update-device-group --deviceid <GUID> --devicegroupid <DGID>
```

Replace <GUID> with the device ID of the chip and replace <DGID> with the ID of the device group to which to assign the device. The device group must support OTA application updates.

#### IMPORTANT

You must configure OTA application updates before you ship your product. If you sideload an application on the factory floor but do not configure OTA application update, the Azure Sphere Security Service will remotely erase the sideloaded application the first time the device connects to the internet. As a result, your customers will lose functionality. In addition, be sure to verify the configuration, as described in the next section.

## Verify the OTA configuration

As a final step before shipping, verify the OTA configuration for each device. This step checks that the Azure Sphere Security Service targets the images you expect for this device. To find out which images will be downloaded for a particular device, use the **azsphere device image list-targeted** command:

```
azsphere device image list-targeted --deviceid <GUID>
```

Replace <GUID> with the device ID for the device you're checking. The targeted images should be the same as the production-signed images that you sideloaded during manufacturing. The output shows the image set ID and name along with the IDs of the individual images in the image set. For example:

```
Successfully retrieved the current image set for device with ID  
'ABCDEF82513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A5588B42085  
1EE4F3F1A7DC5ABCDEF' from your Azure Sphere tenant:  
--> ID: [6e9cdc9d-c9ca-4080-9f95-b77599b4095a]  
--> Name: 'ImageSet-Mt3620Blink1-2018.07.19-18.15.42'  
Images to be installed:  
--> [ID: 116c0bc5-be17-47f9-88af-8f3410fe7efa]  
Command completed successfully in 00:00:04.2733444.
```

# Hardware abstraction files for samples

10/9/2019 • 6 minutes to read

The sample applications in the [Azure Sphere samples repository](#) on GitHub are [hardware independent]. They will run on any Azure Sphere-compatible hardware without changes to the code.

To run the sample applications on your board or module you will need to create the following files:

- **hardware definition file**

JSON file that maps the features exposed by your board or module to the features of the underlying MCU, or module.

- **sample.hardware.json file**

JSON file that maps the generic peripherals referenced in the sample code to the peripherals defined in the hardware definition file for your board or module.

- **Header files**

C header files that are generated from the hardware definition file and the *sample.hardware.json* file.

These files are placed in a folder created for your board or module. This folder is located in the [Hardware folder](#) of the Azure Sphere samples repository.

## Hardware definition files

A hardware definition file is a JSON file with the following format.

```
{  
    "Metadata":  
    {  
        "Type": "Azure Sphere Hardware Definition",  
        "Version": 1  
    },  
    "Description":  
    {  
        "Name": "<name of board or module>",  
        "MainCoreHeaderFileTopContent": [  
            "/* Copyright (c) <vendor name> All rights reserved.",  
            "  <vendor licensing information, if any> */",  
            "",  
            "/* This header contains the peripheral pinout definitions for the",  
            "/* <name of board or module>"  
        ]  
    },  
    "Imports" : [ {"Path": "<path to underlying hardware definition file>"} ],  
    "Peripherals": [  
        {"Name": "", "Type": " ", "Mapping": " ", "Comment": " "},  
    ]  
}
```

The **Metadata** section contains information about the file (file type, version, etc.).

The **Description** section contains information about the board or module. The "MainCoreHeaderFileTopContent" field contains information that will be placed at the beginning of the generated header file.

The **Imports** section specifies the pathname of the hardware definition file for the underlying hardware platform (MCU or module).

The **Peripherals** section lists the peripherals that this board exposes for use in applications. A peripheral description has the following format.

```
{"Name": "<name-in-code>", "Type": "<type>", "Mapping": "<name-in-imported-definition>", "Comment": "<helpful info>"}
```

- *Name* - The identifier used to reference the peripheral in your application code.
- *Type* - The type of peripheral (for example, Gpio, Uart, Adc). Refer to the hardware definition file listed in the **Imports** section to obtain type information.
- *Mapping* - Maps the identifier in the Name field to the identifier used for the peripheral in the imported hardware definition file.
- *Comments* - Provides helpful information to appear in the generated header file. For example, pin assignments for ISU\* peripherals, or mapping on-board LEDs to GPIO pins of the underlying MCU or Module.

The Azure Sphere samples repository on GitHub contains a [hardware definition directory](#) for each supported platform. The JSON files and the generated header files are located in these directories. To specify a target platform for an application, software developers set the properties for their project to use the appropriate hardware definition file. For more information, see [Set the hardware target for an application](#).

The following is a portion of the hardware definition file ([mt3620\\_rdb.json](#)) for the MT3620 Reference Development Board (RDB). It specifies the peripheral pinout definitions for the RDB. It imports resource definitions from the [physical hardware definition file \(mt3620.json\)](#) for the MT3620 MCU. The information in the **Peripherals** section results in a pin-to-pin mapping of the resources exposed by the RDB to the resources provided by the underlying MT3620 MCU.

```
{
  "Metadata": {
    "Type": "Azure Sphere Hardware Definition",
    "Version": 1
  },
  "Description": {
    "Name": "MT3620 Reference Development Board (RDB)",
    "MainCoreHeaderFileTopContent": [
      "/* Copyright (c) Microsoft Corporation. All rights reserved.",
      "   Licensed under the MIT License. */",
      "",
      "// This header contains the peripheral pinout definitions for the",
      "// MT3620 Reference Development Board (RDB)"
    ]
  },
  "Imports" : [ {"Path": "../mt3620/mt3620.json"} ],
  "Peripherals": [
    {"Name": "MT3620_RDB_LED1_RED", "Type": "Gpio", "Mapping": "MT3620_GPIO8", "Comment": "LED 1 Red channel uses GPIO8."},
    {"Name": "MT3620_RDB_LED1_GREEN", "Type": "Gpio", "Mapping": "MT3620_GPIO9", "Comment": "LED 1 Green channel uses GPIO9"},
    .
    .
    .
  ]
}
```

## sample.hardware.json files

To enable the sample applications in the Azure Sphere [samples repository on GitHub](#) to run on your Azure Sphere board or module, create a *sample.hardware.json* file. This file associates the peripherals and corresponding

identifiers referenced in the sample code with the physical peripherals and their identifiers specified in the hardware definition file for your board or module.

The following is a portion of the [sample.hardware.json](#) file for the MT3620 Reference Development Board (RDB). Note that the *sample.hardware.json* file has the same format as the hardware definition file.

```
{  
    "Metadata": {  
        "Type": "Azure Sphere Hardware Definition",  
        "Version": 1  
    },  
    "Description": {  
        "Name": "Sample hardware abstraction for MT3620 RDB",  
        "MainCoreHeaderFileTopContent": [  
            "/* Copyright (c) Microsoft Corporation. All rights reserved.",  
            " Licensed under the MIT License. */",  
            "",  
            "/* This file defines the mapping from the MT3620 reference development board (RDB) to the",  
            "/* 'sample hardware' abstraction used by the samples at https://github.com/Azure/azure-sphere-",  
            "samples.",  
            "/* Some peripherals are on-board on the RDB, while other peripherals must be attached externally  
            if needed.",  
            "/* See TODO(85663) for more information on how to use hardware abstractions",  
            "/* to enable apps to work across multiple hardware variants."  
        ]  
    },  
    "Imports" : [ {"Path": "mt3620_rdb.json"} ],  
    "Peripherals": [  
        {"Name": "SAMPLE_BUTTON_1", "Type": "Gpio", "Mapping": "MT3620_RDB_BUTTON_A", "Comment": "MT3620 RDB:  
        Button A"},  
        .  
        .  
        .  
    ]  
}
```

- The **imports** section contains the path to the hardware definition file for the physical board or module.

```
"Imports" : [ {"Path": "mt3620_rdb.json"} ],
```

- The **Peripherals** section maps generic peripherals to the corresponding peripherals on your board or module.

```
{"Name": "SAMPLE_BUTTON_1", "Type": "Gpio", "Mapping": "MT3620_RDB_BUTTON_A", "Comment": "MT3620 RDB:  
        Button A"},
```

- *Name* - contains the identifier (generic name) used for the peripheral in the sample code.
- *Mapping* - contains the identifier used for the peripheral in the hardware definition file for your board or module.

The following example illustrates how hardware abstraction, implemented through the use of hardware definition and *sample.hardware.json* files, enables the same application to run on two Azure Sphere-compatible boards, each supporting a different combination of hardware resources.

The Azure Sphere [high level GPIO sample application](#) uses a button to change the blink rate of an LED. This sample will run on the MT3620 Reference Development Board (RDB) and on the [Seeed Studios MT3620 Mini Dev board \(MDB\)](#).

The [sample.hardware.json](#) file for the RDB maps the generic button (called Sample\_Button\_1 in the application

code) to the built-in button, MT3620\_RDB\_BUTTON\_A, on the RDB.

```
{"Name": "SAMPLE_BUTTON_1", "Type": "Gpio", "Mapping": "MT3620_RDB_BUTTON_A", "Comment": "MT3620 RDB: Button A"},
```

The MDB does not have a built-in button. The [sample\\_hardware.json file for the MDB](#) maps Sample\_Button\_1 to a GPIO pin on the MDB (GPIO30, header J1, pin 9). An external button can be attached to the board via this pin.

```
{"Name": "SAMPLE_BUTTON_1", "Type": "Gpio", "Mapping": "SEEED_MT3620_MDB_J1_PIN9_GPIO30", "Comment": "MT3620 MDB: Connect external button using J1, pin 9."},
```

## Header files

A header file accompanies each hardware definition file and sample\_hardware.json file. Use the [azsphere hardware-definition generate-header](#) command to generate a header file.

To create the header file, navigate to the folder that contains the JSON file and enter the following line at the Azure Sphere command prompt. Replace with the name of your hardware definition file or sample\_hardware.json file.

```
azsphere hardware-definition generate-header --input <filename>
```

The header file filename.h will be created and placed in the folder *inc/hw*.

For example, enter the following line to generate a header file from the hardware definition file for the MT3620 Reference Development Board (RDB).

```
azsphere hardware-definition generate-header --input mt3620_rdb.json
```

A portion of the [mt3620\\_rdb.h header file](#) is shown below.

```
/* Copyright (c) Microsoft Corporation. All rights reserved.
   Licensed under the MIT License. */
// This header contains the peripheral pinout definitions for the
// MT3620 Reference Development Board (RDB)
// This file is autogenerated from ..\..\mt3620_rdb.json. Do not edit it directly.

#pragma once
#include "..\..\..\mt3620\inc\hw\mt3620.h"

// LED 1 Red channel uses GPIO8.
#define MT3620_RDB_LED1_RED MT3620_GPIO8

// LED 1 Green channel uses GPIO9
#define MT3620_RDB_LED1_GREEN MT3620_GPIO9

.
```

**Note** By default, the generated header file will be placed in *inc/hw*, which must be a subfolder of the folder that contains the input JSON file. If this subfolder doesn't exist it will be created.

# Samples and reference solutions

6/20/2019 • 2 minutes to read

You can find the following [Azure Sphere samples and reference solutions](#) on GitHub.

SAMPLE NAME	DESCRIPTION
ADC	Shows how to use analog-to-digital (ADC) functionality on Azure Sphere
AzureIoT	Shows how to use Azure Sphere with Azure IoT Central or an Azure IoT Hub
DNSServiceDiscovery	Shows how to perform domain name service (DNS) service discovery
ExternalMcuUpdate	Shows how you might deploy an update to an external MCU device using Azure Sphere
GPIO	Shows how to use GPIO on an Azure Sphere device
HelloWorld	Provides minimal code with which to verify installation and shows how to use CMake to build a high-level application
HTTPS	Shows how to use the cURL API with Azure Sphere
I2C	Shows how to use the Azure Sphere inter-integrated circuit (I2C) API to display data from an accelerometer connected via I2C
IntercoreComms	Shows how to communicate between high-level and real-time capable applications
MutableStorage	Shows how to use on-device storage in an Azure Sphere application
PrivateNetworkServices	Shows how you can connect Azure Sphere to a private network
SPI	Shows how to use the Azure Sphere serial peripheral interface (SPI) API to display data from an accelerometer connected via SPI
SystemTime	Shows how to manage the system time and to use the hardware real time clock (RTC)
UART	Shows how to use UART on an Azure Sphere device
WifiSetupAndDeviceControlViaBle	Shows how you might complete Wi-Fi setup and device control of an Azure Sphere-based device through BLE using a companion app on a mobile device

# Release Notes 19.09

10/9/2019 • 6 minutes to read

The Azure Sphere 19.09 preview release include the changes, new features, and known issues described in this article.

## To update to the latest release

- If your Azure Sphere device is already running the 18.11 OS release (or later), is connected to the internet, and is a member of a device group that depends on the Retail OS feed, it should be updated OTA within 24 hours after the new release is available on the Retail feed.
- If you have an Azure Sphere device that has never been used, complete all the procedures in [Install Azure Sphere](#). When you complete these steps, your device will be running the OS that the Retail feed delivers and will be ready for application development.

### How to verify installation

You can verify the installed OS version by issuing the following command in an Azure Sphere Developer Command Prompt:

```
azsphere device show-ota-status
```

To check which version of the SDK is installed, use the following command:

```
azsphere show-version
```

## New features and changes in the 19.09 release

The 19.09 release includes substantial investments in the Azure Sphere OS support for the MT3620 hardware and in tooling support for additional application development scenarios. The following sections describe new and changed features.

### Time server name change

The name of the time server has changed to prod.time.sphere.azure.net instead of time.sphere.azure.net. See [Azure Sphere OS networking requirements](#) for details.

#### IMPORTANT

Ensure that your network firewall settings enable access to prod.time.sphere.azure.net so that your device can receive over-the-air updates and new certificates.

### ADC for high-level applications

The MT3620 supports analog-to-digital (ADC) converters. This release provides support for using [ADC on the high-level \(A7\) core](#). The [ADC folder](#) in the Azure Sphere samples repository contains the **ADC\_HighLevelApp** sample, which shows how to use ADC in a high-level application.

### PWM support

The MT3620 supports pulse-width modulation (PWM). This release provides support for [using PWM](#) on the high-level (A7) and real-time capable (M4) cores. The [PWM folder](#) in the Azure Sphere samples repository contains the **PWM\_HighLevelApp** sample, which shows how to use PWM in a high-level application.

## Wi-Fi capabilities

Wi-Fi capabilities have been expanded at this release to enable connection to hidden Wi-Fi networks and to better support network discovery in congested Wi-Fi environments. These features are supported both through the **azsphere device wifi** command and in the [wificonfig API](#). Note that on the command line, the name of the `--key` parameter has been changed to `--psk` and the abbreviation has changed to `-p`.

## Device update deferral

High-level applications can now request notification of pending OS and application updates, so that they can defer updates that would interrupt critical operations. For details, see [Defer device updates](#).

## Mutual authentication for web services

This release adds support for [mutual authentication](#) in high-level apps that connect to web services. Mutual authentication verifies that both the web server and the client device are legitimate. It is a Beta feature at this release.

## API changes

The 19.09 Azure Sphere OS release includes several changes to the API set:

- New Beta APIs for event notification[[..../reference/applibs-reference/applibs-sysevent/sysevent-overview.md](#)] and [event loops](#). High-level apps can use these APIs to request notification and deferral of updates.
- Changes to the Beta API for networking. Several previous Beta features have been replaced by new functionality. See the [networking API reference](#) for details.
- Changes to the wificonfig API. See [Applib's wificonfig.h](#) for details.
- Promotion of the SPI API to production/long-term support (LTS) from Beta.
- Promotion of the I2C API to production/LTS from Beta.

## Application runtime version

The application runtime version (ARV) has been incremented from 2 to 3 at this release. The Azure Sphere SDK now includes four sysroots: 1, 2, 3, and 3+Beta1909. Support in **azsphere** commands enables detection of conflicting application and OS dependencies. See [Target API sets, ARV, and sysroots](#) for details about ARVs.

## Samples

This release provides new sample applications:

- [ADC\\_HighLevelApp](#)
- [PWM\\_HighLevelApp](#)

## Compatibility with the previous release

The 19.09 release supports Target API sets 3, 3+Beta1909, 2, and 1.

For details about compatibility with the previous release, see [Beta APIs and OS compatibility](#).

### IMPORTANT

The Networking API, which was a Beta feature in the previous release, remains a Beta feature in this release. However, the API has changed in ways that might break existing code at future releases. See the [networking API reference](#) for details.

## Known issues in 19.09

This section lists known issues in the current release.

### Number of Wi-Fi networks reported

Azure Sphere currently supports a maximum of 37 Wi-Fi networks. Attempts to store more than 37 networks may

result in undefined behavior.

## CMake path lengths

If the length of your starting directory path is long, CMake returns an error when you open the folder. To avoid this problem, use short paths when developing with CMake.

## Inaccurate clock on real-time cores

The GPT0 and GPT1 timers that are built into the real-time cores should use a 32 KHz clock source, but currently they do not run at the correct frequency. Consequently, applications that use these timers will get inaccurate timeouts. For example, a request for a 1-second delay could actually delay for 1.5 seconds. GPT3 has higher resolution, but if your scenario requires more than one timer, you may need to use GPT0 or GPT1. You can work around this issue by implementing a timer queue and running all the timers from GPT3.

## Determine where an RTApp runs

By default, the RTApp is deployed to the first available real-time core on the device. To find out which core the application is running on, use the **azsphere device sideload start** command to start the application. The **azsphere device sideload show-status** command does not currently display this information.

You cannot choose which of the two real-time cores an RTApp runs on.

## CMake startup item

When you're developing RTApps using CMake, the Select Startup Item menu may reset, resulting in an error when you try to start the application. This tends to occur when you regenerate the CMake cache. Before starting the application, ensure that the menu specifies GDB Debugger (RTCore):



## Installation of both Visual Studio 2017 and Visual Studio 2019 Preview

If you have installed the 18.11 Azure Sphere SDK Preview for Visual Studio with both Visual Studio 2017 and Visual Studio 2019 Preview, the Azure Sphere SDK Preview for Visual Studio installer may fail to install or uninstall on either or both versions with the message "unexpected error". To recover from this problem:

1. Start Visual Studio 2019 Preview and go to **Extensions > Extensions and Updates**. Search for "azure Sphere" and uninstall the Visual Studio Extension for Azure Sphere Preview.
2. Close Visual Studio 2019.
3. Run the Azure Sphere SDK Preview for Visual Studio installer again.

## Wi-Fi commands return device error 13.1

The **azsphere device wifi show-status** command may return `error: device error 13.1` if the most recent **azsphere device wifi add** command supplied an incorrect --key value. If this occurs, use the **azsphere device wifi delete** command to delete the incorrect Wi-Fi network configuration and then add the network again with the correct key.

## Non-ASCII characters in paths

The Azure Sphere tools do not support non-ASCII characters in paths.

## Build errors with C++

The Visual Studio Integrated Development Environment (IDE) does not generate an error if you add a C++ source file to an Azure Sphere project. However, C++ development is not supported with Azure Sphere, and the resulting project will not build correctly.

# Release Notes 19.05, 19.06, and 19.07

7/31/2019 • 10 minutes to read

The Azure Sphere 19.05, 19.06, and 19.07 preview releases include the changes, new features, and known issues described in this article.

## IMPORTANT

**19.07 is an important update for the development and manufacturing scenarios described below. For these scenarios, you need to update your devices to the 19.07 release by August 5.**

If your devices are connected to the internet, they should receive the update over the air (OTA); no action is required on your part. To find out which release a device is running, use the **azsphere device show-ota-status** command.

If it's not possible to put the device online, you will need to [recover the device](#).

After August 5, attempts to add or remove the AppDevelopment capability (**azsphere device prep-debug** and **azsphere device prep-field**) will fail on development devices that are running an older release of the Azure Sphere OS, with the error below. This is due to the normal yearly update to the production keys that are used to sign capabilities and images.

```
error: The device did not accept the device capability configuration. Please check the Azure Sphere OS on your device is up to date using 'azsphere device show-ota-status'.
```

In addition, if you create a new [production-signed application image for factory installation](#) after August 5, you must also update your factory process to install the 19.07 OS release. However, you can continue to run your existing factory process with your existing application and OS images, until you need to create a new version of the application.

## To update to the latest release

- If your Azure Sphere device is already running the 18.11 OS release (or later), is connected to the internet, and is a member of a device group that depends on the Retail OS feed, it should be updated OTA within 24 hours after the new release is available on the Retail feed.
- If you have an Azure Sphere device that has never been used, complete all the procedures in [Install Azure Sphere](#). When you complete these steps, your device will be running the OS that the Retail feed delivers and will be ready for application development.

## How to verify installation

You can verify the installed OS version by issuing the following command in an Azure Sphere Developer Command Prompt:

```
azsphere device show-ota-status
```

To check which version of the SDK is installed, use the following command:

```
azsphere show-version
```

## New features and changes in the 19.07 release

The 19.07 quality release includes an updated Azure Sphere OS only; it does not include an updated SDK. Continue to use the [Azure Sphere SDK Preview from the Visual Studio Marketplace](#). This release supports the same APIs as the 19.05 release.

## Fixed bugs

This release fixes a bug that prevented the deletion of temporary images. Starting at the 19.05 release, if you sideloaded a production-signed image with the `--temporary` flag (as in a factory-floor scenario), the image could not be deleted unless the device had the AppDevelopment capability. This bug was a regression from the previous release. It has now been fixed.

## Common vulnerabilities

The periodic Azure Sphere quality releases incorporate current patches to the custom Linux-based OS. When potentially significant common vulnerabilities and exposures (CVEs) have been found and fixed, we list those issues in the Release Notes. The 19.07 quality release includes changes to mitigate against [CVE-2019-5436](#).

Although Microsoft does not believe that Azure Sphere is exploitable as a result of these vulnerabilities, we have nevertheless prioritized mitigation.

## New features and changes in the 19.06 release

The 19.06 quality release includes an updated Azure Sphere OS only; it does not include an updated SDK.

Continue to use the [Azure Sphere SDK Preview from the Visual Studio Marketplace](#). This release supports the same APIs as the 19.05 release. It does not contain any user-visible changes.

The periodic Azure Sphere quality releases incorporate current patches to the custom Linux-based OS. When potentially significant common vulnerabilities and exposures (CVEs) have been found and fixed, we list those issues in the Release Notes.

The 19.06 quality release includes important changes to the Azure Sphere OS to mitigate against the following Linux vulnerabilities:

- [CVE-2019-11477](#)
- [CVE-2019-11478](#)
- [CVE-2019-11479](#)

Although Microsoft does not believe that Azure Sphere is exploitable as a result of these vulnerabilities, we have nevertheless prioritized mitigation.

## New features and changes in the 19.05 release

The 19.05 release includes substantial investments in the Azure Sphere OS support for the MT3620 hardware and in tooling support for additional application development scenarios. The following sections describe new and changed features.

### Real-time core preview

This release includes Beta support for developing, deploying, and debugging real-time capable applications (RTApps) on the MT3620's two M4 cores. You can find information about RTApp development in the [Develop real-time capable applications](#) topic in the Azure Sphere online documentation.

The [Azure Sphere samples repository](#) on GitHub includes samples that show how to use GPIO and UART peripherals from an RTApp and how to communicate between an RTApp and a high-level (A7) application.

### Support for additional hardware platforms

Several hardware ecosystem partners have recently announced new Azure Sphere-enabled products:

- SEEED MT3620 Mini Development Board: Development board with single-band Wi-Fi for size-constrained prototypes. This board uses the AI-Link module for a quick path from prototype to commercialization.
- AI-Link WF-M620-RSA1 Wi-Fi Module: Single-band Wi-Fi module designed for cost-sensitive applications.
- USI Azure Sphere Combo Module: Dual-band Wi-Fi and Bluetooth module. The on-board Bluetooth chipset supports BLE and Bluetooth 5 Mesh. The chipset can also work as an NFC tag to support non-contact

Bluetooth pairing and device provisioning scenarios.

- Avnet Guardian module: Module designed for securely connecting existing equipment to the internet. The module attaches to the equipment through Ethernet and connects to the cloud via dual-band Wi-Fi.
- Avnet MT3620 Starter Kit: Development board with dual-band Wi-Fi connectivity featuring modular connectors that support a range of MikroE Click and Grove modules.
- Avnet Wi-Fi Module: Dual-band Wi-Fi module. Stamp hole (castellated) pin design allows for easy assembly and simpler quality assurance.

To target a particular hardware product in an application, you set the application's

**TargetHardwareDefinitionDirectory** property. Both Visual Studio and CMake build procedures support this property, and it applies to the sample applications as well as your own custom applications. For more information, see [Manage target hardware dependencies](#) in the online documentation and the [Hardware/README.md file](#) in the Azure Sphere Samples repository on GitHub.

### Application runtime version

The application runtime version (ARV) has been incremented from 1 to 2 at this release. The Azure Sphere SDK now includes three sysroots: 1, 2, and 2+Beta1905. Support in **azsphere** commands enables detection of conflicting application and OS dependencies. See [Target API sets, ARV, and sysroots](#) for details about ARVs.

### ADC preview

The MT3620 supports analog-to-digital (ADC) converters. This release provides preview support for using ADC on the real-time capable (M4) cores. The [ADC folder](#) in the Azure Sphere samples repository contains the **ADC\_RTApp\_MT3620\_BareMetal** sample, which shows how to use ADC in a real-time capable application.

### API changes

The 19.05 Azure Sphere OS release includes several changes to the API set:

- New Application API for communication with real-time capable applications.
- Changes to the Beta API for networking. Several previous Beta features have been replaced by new functionality. See the [networking API reference](#) for details.
- Promotion of the mutable storage API to production from Beta.
- POSIX API support for random number generation from Pluto's random number generator.

### Visual Studio support

This release requires any edition of Visual Studio 2019, version 16.04 or later, or Visual Studio 2017, version 15.9 or later.

Visual Studio now supports F5 deployment and debugging for applications that use CMake.

The UART, GPIO, and Azure IoT templates have been moved to the [Azure Sphere samples repository](#) on GitHub. Visual Studio now provides a simple Blink template and a static library template for Azure Sphere.

### Ethernet

Ethernet is now supported as an alternative to Wi-Fi for communicating with the Azure Sphere Security Service and your own services. The HTTPS and AzureIoT samples can now be run over Ethernet in addition to Wi-Fi. The Azure Sphere Samples repository on GitHub includes documentation on how to wire the supported MicroChip part, bring up the Ethernet interface, and connect to Azure IoT or your own web services.

### Local device discovery

The Azure Sphere OS offers new network firewall and multicast capabilities that enable apps to run mDNS and DNS-SD for device discovery on local networks.

### Azure Sphere samples on GitHub

All Azure Sphere samples in GitHub require the 19.05 SDK. We recommend that you update your local version of the [GitHub samples repo](#) upon installation of the 19.05 release.

To build the sample applications, you must clone the entire repository.

#### Hardware-specific information

The Hardware folder in the samples repository contains a folder for each current Azure Sphere hardware platform. Within each folder are hardware-specific JSON and header (.h) files that you can use to target the samples—or your own applications—at any Azure Sphere hardware.

The [README.md file](#) describes how to change the hardware target for the samples.

#### Network requirements for samples

All Azure Sphere samples now support the use of either Wi-Fi or Ethernet for networking.

#### Security changes

Strong stack protection and address space layout randomization (ASLR) are now enabled by default when building high-level applications. To take advantage of these features in an existing application, you need to rebuild the application.

## Compatibility with the previous release

The 19.05 and 19.06 releases support three Target API sets: 2, 2+Beta1905, 1

For details about compatibility with the previous release, see [Beta APIs and OS compatibility](#).

#### IMPORTANT

The Networking API, which was a Beta feature in the 19.02 release, remains a Beta feature in the 19.05 release. However, the API has changed in ways that might break existing code at future releases. See the [networking API reference](#) for details.

## Known issues in 19.05

This section lists known issues in the current release.

#### Inaccurate clock on real-time cores

The GPT0 and GPT1 timers that are built into the real-time cores should use a 32 KHz clock source, but currently they do not run at the correct frequency. Consequently, applications that use these timers will get inaccurate timeouts. For example, a request for a 1-second delay could actually delay for 1.5 seconds. GPT3 has higher resolution, but if your scenario requires more than one timer, you may need to use GPT0 or GPT1. You can work around this issue by implementing a timer queue and running all the timers from GPT3.

#### Determine where an RTApp runs

By default, the RTApp is deployed to the first available real-time core on the device. To find out which core the application is running on, use the **azsphere device sideload start** command to start the application. The **azsphere device sideload show-status** command does not currently display this information.

You cannot choose which of the two real-time cores an RTApp runs on.

#### CMake startup item

When you're developing RTApps using CMake, the Select Startup Item menu may reset, resulting in an error when you try to start the application. This tends to occur when you regenerate the CMake cache. Before starting the application, ensure that the menu specifies GDB Debugger (RTCore):



#### Installation of both Visual Studio 2017 and Visual Studio 2019 Preview

If you have installed the 18.11 Azure Sphere SDK Preview for Visual Studio with both Visual Studio 2017 and

Visual Studio 2019 Preview, the Azure Sphere SDK Preview for Visual Studio installer may fail to install or uninstall on either or both versions with the message "unexpected error". To recover from this problem:

1. Start Visual Studio 2019 Preview and go to **Extensions > Extensions and Updates**. Search for "azure Sphere" and uninstall the Visual Studio Extension for Azure Sphere Preview.
2. Close Visual Studio 2019.
3. Run the Azure Sphere SDK Preview for Visual Studio installer again.

### Wi-Fi commands return device error 13.1

The **azsphere device wifi show-status** command may return `error: device error 13.1` if the most recent **azsphere device wifi add** command supplied an incorrect --key value. If this occurs, use the **azsphere device wifi delete** command to delete the incorrect Wi-Fi network configuration and then add the network again with the correct key.

### Non-ASCII characters in paths

The Azure Sphere tools do not support non-ASCII characters in paths.

### Build errors with C++

The Visual Studio Integrated Development Environment (IDE) does not generate an error if you add a C++ source file to an Azure Sphere project. However, C++ development is not supported with Azure Sphere, and the resulting project will not build correctly.

# Release Notes 19.02, 19.03, and 19.04

5/30/2019 • 11 minutes to read

The Azure Sphere 19.02 preview feature release and 19.03 and 19.04 quality releases include the changes, new features, and known issues described in this article.

## NOTE

Starting with the 19.02 feature release, we plan one major feature release and two minor quality releases per quarter. A feature release contains new functionality, whereas a quality release contains bug fixes. The Azure Sphere 19.03 release is the first quality release.

## About the 19.04 quality release

The 19.04 quality release includes an updated Azure Sphere OS only; it does not include an updated SDK.

Continue to use the [Azure Sphere SDK Preview from the Visual Studio Marketplace](#). This release supports the same APIs as the 19.02 release. It does not contain any user-visible changes.

As noted in [OS feeds](#), we now support both a Retail Evaluation feed and a Retail feed for the Azure Sphere OS. The 19.04 update was released on the Retail Evaluation feed on April 10. Starting April 24, it is available on the Retail feed. During the evaluation period, you can verify that your OTA-deployed software continues to work as you expect.

If your devices are connected to the internet, they will be updated OTA within 24 hours after the 19.04 version is released to the feed that their device groups depend on.

## About the 19.03 quality release

The 19.03 quality release includes an updated Azure Sphere OS only; it does not include an updated SDK.

Continue to use the [Azure Sphere SDK Preview from the Visual Studio Marketplace](#). This release supports the same APIs as the 19.02 release.

As noted in [OS feeds](#), we now support both a Retail Evaluation feed and a Retail feed for the Azure Sphere OS. The 19.03 release was initially available only through the Retail Evaluation feed. Effective March 28, it is available on the Retail feed. During the evaluation period, you can verify that your OTA-deployed software continues to work as you expect.

### Changes in the 19.03 release

The 19.03 release resolves an issue in which Azure Sphere did not accept a DHCP offer if the packet sent by the DHCP server had a zero UDP checksum.

## How to update to the 19.03 release

You can update to the 19.03 release immediately from the Retail Evaluation feed, or you can update from the Retail feed after the evaluation period ends.

### To update during the evaluation period

To obtain the release during the evaluation period, you will need to set up a new device group and feed that depend on the Retail Evaluation feed. [Set up a device group for OS evaluation](#) explains how. Currently, update to the Retail Evaluation feed is supported for device groups that enable over-the-air application updates.

The procedure assumes that your device is running the 18.11 release or later. If your device is running an earlier release or has never been used, first follow the instructions in the next section to update to the version on the Retail feed and then set up a device group that depends on the Retail Evaluation feed and assign your device to it. Currently, the Retail Evaluation feed downloads the OS only to devices that are configured for over-the-air (OTA) application deployment.

The **azsphere device recover** command always recovers to the Retail OS version. If you update to the 19.03 release during the evaluation period and then recover your device, the device will receive the 19.02 release.

### To update after the evaluation period

- If your Azure Sphere device is already running the 18.11 OS release (or later), is connected to the internet, and is a member of a device group that depends on the Retail OS feed, it should be updated OTA within 24 hours after the 19.03 release is available on the Retail feed.
- If you have an Azure Sphere device that has never been used, complete all the procedures in [Install Azure Sphere](#). When you complete these steps, your device will be running the OS that the Retail feed delivers and will be ready for application development.
- If your Azure Sphere device has been claimed but is running TP 4.2.1, follow the instructions in the [18.11 Release Notes](#) to update the Azure Sphere OS on a claimed device and move the device into a device group that does not deliver OTA application updates. This procedure will update the device to the current release on the Retail feed.

## How to verify installation

You can verify the installed OS version by issuing the following command in an Azure Sphere Developer Command Prompt:

```
azsphere device show-ota-status
```

To check which version of the SDK is installed, use the following command:

```
azsphere show-version
```

## Deprecation of TP 4.2.1

We no longer support the TP 4.2.1 release. Devices that are running TP 4.2.1 will no longer be issued certificates by the Azure Sphere Security Service, and consequently cannot authenticate to the device provisioning service (DPS). Attempts to authenticate will return the error `AZURE_SPHERE_PROV_RESULT_DEVICEAUTH_NOT_READY`. You must [update these devices to the current release](#).

## New features and changes in the 19.02 release

This release includes substantial investments in the Azure Sphere OS support for the MT3620 hardware. The following sections describe new and changed features.

### SPI

This release adds support for use of the [Serial Peripheral Interface \(SPI\)](#) on the MT3620. A [Beta API for SPI](#) is available for use in application development. The [LSM6DS3\\_SPI sample](#) shows how you can develop applications that use SPI.

### I2C

This release adds support for the [Inter-Integrated Circuit \(I2C\)](#) interface on the MT3620. A [Beta API for I2C](#) is available for use in application development. The [LSM6DS3\\_I2C sample](#) shows how you can develop applications that use I2C.

## DHCP and SNTP servers

In this release, the Azure Sphere OS includes DHCP and SNTP server support for private LAN configurations. The [Private Ethernet sample](#) shows how to use them.

## Application size and storage

One MiB of read-only flash memory is now dedicated for customer-deployed image packages at runtime. During development, the 1 MiB limit includes the gdbserver debugger, which requires approximately 280KiB in the current release. Keep in mind, however, that the debugger size may change in future releases. For additional details, see:

- [Memory available for applications](#)
- [Using storage on Azure Sphere](#)

The [azsphere device sideload show-quota](#) command has been added to display the amount of mutable storage allocated and used by an application.

## Azure IoT support

The Azure Sphere OS has updated its Azure IoT SDK to the LTS Oct 2018 version. In addition, a new [Azure IoT reference solution](#) shows how to use Azure Sphere with either Azure IoT Central or an Azure IoT Hub.

## OS update protection

The Azure Sphere OS now detects additional update scenarios that might cause the device to fail to boot. When one of these problems occurs, the OS automatically rolls back the device to its last known good configuration. Rollback may take longer than successful update because the device reboots but is otherwise not visible to customers.

## CMake preview

A [CMake Sample](#) provides an early preview of CMake as an alternative for building Azure Sphere applications. This limited preview lets customers begin testing the use of existing assets in Azure Sphere development.

## Wi-Fi setup and device control via BLE

The [Wi-Fi Setup and Device Control Via BLE reference solution](#) extends the Bluetooth Wi-Fi pairing solution that was released in 18.11 to demonstrate how to control a device locally via BLE when, for example, internet access is not available. The updated reference solution also uses a passkey to protect the BLE bonding process. Only bonded devices may provide Wi-Fi credentials or perform local control via encrypted BLE connection.

## OS feeds

The 'Preview MT3620' feed has been renamed to 'Retail Azure Sphere OS.' It retains the same feed ID (3369f0e1-dedf-49ec-a602-2aa98669fd61) as in the 18.11 release. Because the feed ID has not changed, you do not need to change your existing application deployments.

The new 'Retail Evaluation Azure Sphere OS' feed (feed ID 82bacf85-990d-4023-91c5-c6694a9fa5b4) delivers an evaluation version of the Azure Sphere OS approximately two weeks before that version is released to the Retail feed.

See [Azure Sphere OS feeds](#) for details about the feeds.

## Compatibility with the previous release

The 19.02 Azure Sphere OS release includes several changes to the API set:

- New APIs for SPI and I2C along with additions to the networking API for SNTP and DHCP
- Enhancements to production APIs, including additional options for UART and additional error codes for networking

The 19.02 release supports two Target API sets: 1 and 1+Beta1902.

## 18.11 applications and the 19.02 release

Existing application images that were built with the 18.11 SDK should run successfully on the 19.02 OS. For the 19.02 release, this is true for images that use Beta APIs as well images that use only production APIs. Thus, if you've created an OTA deployment for an 18.11 image package, it should continue to work on 19.02.

If you use the 19.02 SDK to rebuild an 18.11 application that uses Beta APIs, however, you must update the **Target API Set** property to **1+Beta1902** in the Visual Studio **Project > Properties** page, as described in [Beta API features](#).

If you do not update the **Target API Set** property, the build fails with the following message:

```
Target API Set %s is not supported by this SDK. Please update your SDK at http://aka.ms/AzureSphereSDKDownload, or open the Project Properties and select a 'Target API Set' that this SDK supports. Available targets are: [ ..., ...]. Ensure that the Configuration selected on that page includes the active build configuration (e.g. Debug, Release, All Configurations).
```

## 19.02 applications and the 18.11 release

If you install the 19.02 SDK and try to build an application before your device has received the OTA update to 19.02, you may encounter errors upon sideloading the application onto your device. If in doubt, [verify the OS and SDK versions](#). The sections that follow summarize the expected behavior.

### Building 19.02 applications against the 18.11 SDK

If you try to build an application that uses Target API Set 1 against the 18.11 SDK, compilation will fail if the application uses any new symbols that were introduced at 19.02, such as new UART enums or networking errors codes. Otherwise, the build will succeed but sideloading might fail, as described in the following section.

If you try to build an application that uses Target API Set 1+Beta1902 against the 18.11 SDK, the build will fail with one of the following messages:

- The specified task executable location "C:\Program Files (x86)\Microsoft Azure Sphere SDK\\$\SysRoot\tools\gcc\arm-poky-linux-musleabi-gcc.exe" is invalid.

OR:

- 1>A task was canceled., followed by multiple errors like  
mt3620\_rdb.h:9:10: fatal error: soc/mt3620\_i2cs.h: No such file or directory

### Sideload 19.02 applications on the 18.11 OS

Both Visual Studio and the **azsphere** command sideload applications onto Azure Sphere devices. If you sideload an application that was built against the 19.02 SDK onto a device that is running the 18.11 OS, expect the following results:

- If your application was built for Target API Set **1**, it fails at run time if it uses new UART options or networking errors.
- If your application was built for Target API Set **1+Beta1902**, it fails at run time if it uses new Beta APIs, UART options, or networking error codes. For example, if your 19.02 application uses the new I2C API, you might see an error like the following:

```
Error relocating /mnt/apps/c5b532c2-8379-49b9-8771-7228b03c23f3/bin/app: I2CMaster_Open: symbol not found
```

### OTA deployment of 19.02 applications on the 18.11 OS

OTA deployment of 19.02 applications to the 18.11 OS is not possible because all application feeds depend on the Retail Azure Sphere OS feed, which provides the 19.02 OS.

### Sample applications

All Azure Sphere samples in GitHub require the 19.02 SDK. We recommend that you update your local version of the [GitHub samples repo](<https://github.com/Azure/azure-sphere-samples>) upon installation of the 19.02 release.

If you try to build older (18.11) samples against the 19.02 SDK, you may see the following error:

```
Could not add sysroot details to application manifest from '*filename*'. The specified sysroot '1+Beta1902' contains TargetBetaApis 'Beta1902', but the application manifest only contains the TargetApplicationRuntimeVersion field. Either the TargetApplicationRuntimeVersion field must be removed or the TargetBetaApis field must be added.
```

To correct this error, remove **TargetApplicationRuntimeVersion** field from the application manifest or update the samples.

## Known issues in 19.02

This section lists known issues in the current release.

### Installation of both Visual Studio 2017 and Visual Studio 2019 Preview

If you have installed the 18.11 Azure Sphere SDK Preview for Visual Studio with both Visual Studio 2017 and Visual Studio 2019 Preview, the Azure Sphere SDK Preview for Visual Studio installer may fail to install or uninstall on either or both versions with the message "unexpected error". To recover from this problem:

1. Start Visual Studio 2019 Preview and go to **Extensions > Extensions and Updates**. Search for "azure Sphere" and uninstall the Visual Studio Extension for Azure Sphere Preview.
2. Close Visual Studio 2019.
3. Run the Azure Sphere SDK Preview for Visual Studio installer again.

### Wi-Fi commands return device error 13.1

The **azsphere device wifi show-status** command may return `error: device error 13.1` if the most recent **azsphere device wifi add** command supplied an incorrect --key value. If this occurs, use the **azsphere device wifi delete** command to delete the incorrect Wi-Fi network configuration and then add the network again with the correct key.

### Non-ASCII characters in paths

The Azure Sphere tools do not support non-ASCII characters in paths.

### Build errors with C++

The Visual Studio Integrated Development Environment (IDE) does not generate an error if you add a C++ source file to an Azure Sphere project. However, C++ development is not supported with Azure Sphere, and the resulting project will not build correctly.

# Release Notes 18.11, 18.11.1, and 18.11.2

5/30/2019 • 11 minutes to read

The Azure Sphere 18.11 preview release includes the changes, new features, and known issues described in this article.

## NOTE

The numbering pattern for our SDK releases has changed. Although the previous release was 4.2.1, this release and subsequent releases are numbered according to the year and month of release, with an additional number to indicate an update. Thus, the November 2018 release is numbered 18.11, and minor updates are numbered 18.11.1 and so forth.

The 18.11 release features substantial changes to the Azure Sphere Security Service and cloud infrastructure. These security improvements will enable devices that have been offline for an extended period to reconnect. As a result of this change, updating the OS to the 18.11 release will involve connecting the device to a PC and manually installing the OS instead of receiving the software over the air (OTA). We expect to update by OTA for future releases.

## About the 18.11.1 SDK release

The Azure Sphere 18.11.1 SDK release supplements the 18.11 release by adding support for [Private Ethernet](#). If you have already updated your device and development environment to 18.11, you do not need to update the SDK to 18.11.1 unless you plan to use the private Ethernet support. To update to the 18.11.1 SDK, [download it](#) from Visual Studio Marketplace and run Azure\_Sphere\_SDK\_Preview\_for\_Visual\_Studio.exe from the download to install. The 18.11.1 SDK works with the 18.11 OS release; there is not an 18.11.1 OS release.

## About the 18.11.2 OS release

The 18.11.2 update resolves an issue that prevented Azure Sphere devices from connecting to Wi-Fi in some cases. If you have already updated your device and development environment to 18.11, you will receive the 18.11.2 release OTA. It includes only the Azure Sphere OS; it does not include the SDK. If you have not updated to 18.11, follow the instructions in [Install the 18.11 release](#); you will receive the 18.11.2 update OTA after 18.11 installation is complete.

To ensure that your Azure Sphere device receives the 18.11.2 update, connect the device to the internet. Delivery of the update should occur within 24 hours. To download the update immediately, press the Reset button on the development board. Installation of the updated OS should complete within 10 minutes.

To verify installation, issue the following command to list all images on the device:

```
azsphere device image list-installed --full
```

Scroll through the image list and verify that the image ID for NW Kernel matches the following:

```
Installed images:  
...  
--> NW Kernel  
  --> Image type: System software image type 7  
  --> Component ID: ec96028b-080b-4ff5-9ef1-b40264a8c652  
  --> Image ID: 44ed692e-ce49-4425-9aa6-952b84ce7b32  
...  
Command completed successfully in 00:00:02.0395495.
```

## Install the 18.11 release

All Azure Sphere devices are shipped from the manufacturer with the Azure Sphere OS installed. Normally, connecting the device to Wi-Fi triggers over-the-air (OTA) OS update if a more recent version is available. For the 18.11 release, you must instead manually update each device because of substantial changes to the Azure Sphere Security Service and cloud infrastructure. Manual update requires that you connect the Azure Sphere device to a PC and use the *device recovery* procedure to replace the system software.

If you have an Azure Sphere device that has never been used, complete all the procedures in [Install Azure Sphere](#). When you complete these steps, your device will be ready for application development. You can ignore the procedures that are described in this section.

If your Azure Sphere device has already been claimed, follow the instructions in these release notes to recover your device and reconnect to Wi-Fi. If you previously configured OTA application deployment, you will also need to create new feeds and device groups. Existing TP4.2.1 application will continue to run on this release, but require changes if you rebuild, as described in [Target API set and Beta APIs](#) later in this document.

### IMPORTANT

Upgrade to release 18.11 as soon as possible. Devices that run the TP 4.2.1 release cannot receive any OTA updates for either device or application software.

TP 4.2.1 will continue to be supported until January 15, 2019, or later. We will post a notification on the Azure Updates website at least two weeks before TP 4.2.1 support will end. Thereafter, devices that are running the TP 4.2.1 OS will not be able to authenticate to an Azure IoT Hub. We strongly encourage you to [subscribe to Azure Update notifications](#) so that you receive timely information about TP 4.2.1 support and other Azure Sphere news.

## Update the Azure Sphere OS on a claimed device

If this Azure Sphere device has already been claimed, follow these steps to update the Azure Sphere OS:

1. Connect your Azure Sphere device to your PC over USB.
2. Open an Azure Sphere Developer Command Prompt and issue the **azsphere** command to determine which version of the Azure Sphere SDK Preview for Visual Studio is installed. If the utility reports version 18.11.n.n, you have the current version and can proceed to the next step.

If the utility reports version 2.0.n.n, you need to update the SDK before you can update the OS. To update the SDK, [download it](#) from Visual Studio Marketplace and run Azure\_Sphere\_SDK\_Preview\_for\_Visual\_Studio.exe from the download to install the SDK.

3. Assign your device to a device group that does not deliver over-the-air application updates. The simplest way is to list the available device groups and select the "System Software Only" group:

```
azsphere device-group list
```

```
Listing all device groups.  
--> [ID: cd037ae5-27ca-4a13-9e3b-2a9d87f9d7bd] 'System Software Only'  
Command completed successfully in 00:00:02.0129466.
```

Move your device to the System Software Only group:

```
azsphere device update-device-group -d cd037ae5-27ca-4a13-9e3b-2a9d87f9d7bd
```

4. Issue the following command to manually update the Azure Sphere OS:

```
azsphere device recover
```

You should see output similar to this:

```
Starting device recovery. Please note that this may take up to 10 minutes.  
Board found. Sending recovery bootloader.  
Erasing flash.  
Sending images.  
Sending image 1 of 16.  
Sending image 2 of 16.  
...  
Sending image 16 of 16.  
Finished writing images; rebooting board.  
Device ID: <GUID>  
Device recovered successfully.  
Command completed successfully in 00:02:37.3011134.
```

If update is successful, the **azsphere device show-ota-status** command should return output similar to this:

```
azsphere device show-ota-status -v  
Your device is running Azure Sphere OS version 18.11.  
The Azure Sphere Security Service is targeting this device with Azure Sphere OS version 18.11.  
Your device has the expected version of the Azure Sphere OS: 18.11.  
Command completed successfully in 00:00:03.2653689.
```

If **azsphere device show-ota-status** fails with the following message, the device is in a device group that performs OTA application updates and is therefore receiving the older OS feed on which the OTA application depends.

```
warn: Your device running Azure Sphere OS version 18.11 is being targeted with a deprecated version TP4.2.1. Go to aka.ms/AzureSphereUpgradeGuidance for further advice and support.
```

To solve this problem, assign the device to the System Software Only device group as described in Step 3 and then run the **azsphere device recover** command again.

After your device is successfully updated, set it up for either [application development](#) or for [field use](#).

### To enable application development

Devices that are set up for application development receive system software OTA but do not receive application software OTA.

1. To enable local application development and debugging, use the [prep-debug](#) command as follows:

```
azsphere device prep-debug
```

This command enables application development on the device and moves it to a System Software device group that uses the new 18.11 preview OS feed.

2. [Configure Wi-Fi](#) on the device. The device recovery procedure deletes the existing Wi-Fi configuration.

### To enable field use

Devices that are set up for field use can receive both system software and application software OTA, if they are linked to a feed that delivers applications. If your device previously received application updates OTA, set it up for field use to continue receiving them.

1. To set up the device for field use, issue the [prep-field](#) command in the following form:

```
azsphere device prep-field --newdevicegroupname <new-group-name>
```

Replace <new-group-name> with a unique name for a device group. A new device group is required because of changes to the dependent OS feed at this release.

2. Rebuild the application you want to deploy. Be sure to update the target API set as described [here](#).

3. [Link your device to a new application software feed](#), as follows:

```
azsphere device link-feed --newfeedname <name-for-new-feed> --dependentfeedid <**replace with new feed id**> --imagepath <image-path>
```

Supply a unique name for the new feed and replace <image-path> with the path to the newly rebuilt application image that you want the feed to deliver. A new application feed is required because of the changes to the OTA mechanism.

#### IMPORTANT

Specify "Preview MT3620" with feed ID 3369f0e1-dedf-49ec-a602-2aa98669fd61 as the dependent feed. Do not use the obsolete "Preview MT3620 Feed" (feed ID edd33e66-9b68-44c8-9d23-eb60f2e5018b), and do not link the device to an existing application feed that depends on this feed.

4. [Configure Wi-Fi](#) on the device. The device recovery procedure deletes the existing Wi-Fi configuration.

### Behavior of TP 4.2.1 devices and SDK after 18.11 release

If you continue to use both the TP 4.2.1 OS and the TP 4.2.1 SDK, expect the following behavior:

- Existing TP 4.2.1 devices will continue to operate. Existing sideloaded or OTA applications can continue to communicate with an existing IoT Hub until support for TP 4.2.1 terminates (currently scheduled for January 15, 2019).
- TP 4.2.1 devices will no longer receive OTA application updates or Azure Sphere OS updates. Attempts to deploy or update applications OTA have no effect.

If you install the 18.11 SDK but continue to use a device that has the TP 4.2.1 OS, you may encounter the following behavior:

- The 18.11 SDK displays a reminder message about updating the attached Azure Sphere device whenever it performs an operation on device that is running the TP 4.2.1 release.

## New features and changes in this release

This release includes substantial investments in our security infrastructure, and it incorporates some of your feedback. The following sections describe new and changed features.

### Target API set and Beta APIs

This release includes [Beta APIs](#) for testing and development. Beta APIs are still in development and may change in or be removed from a later release.

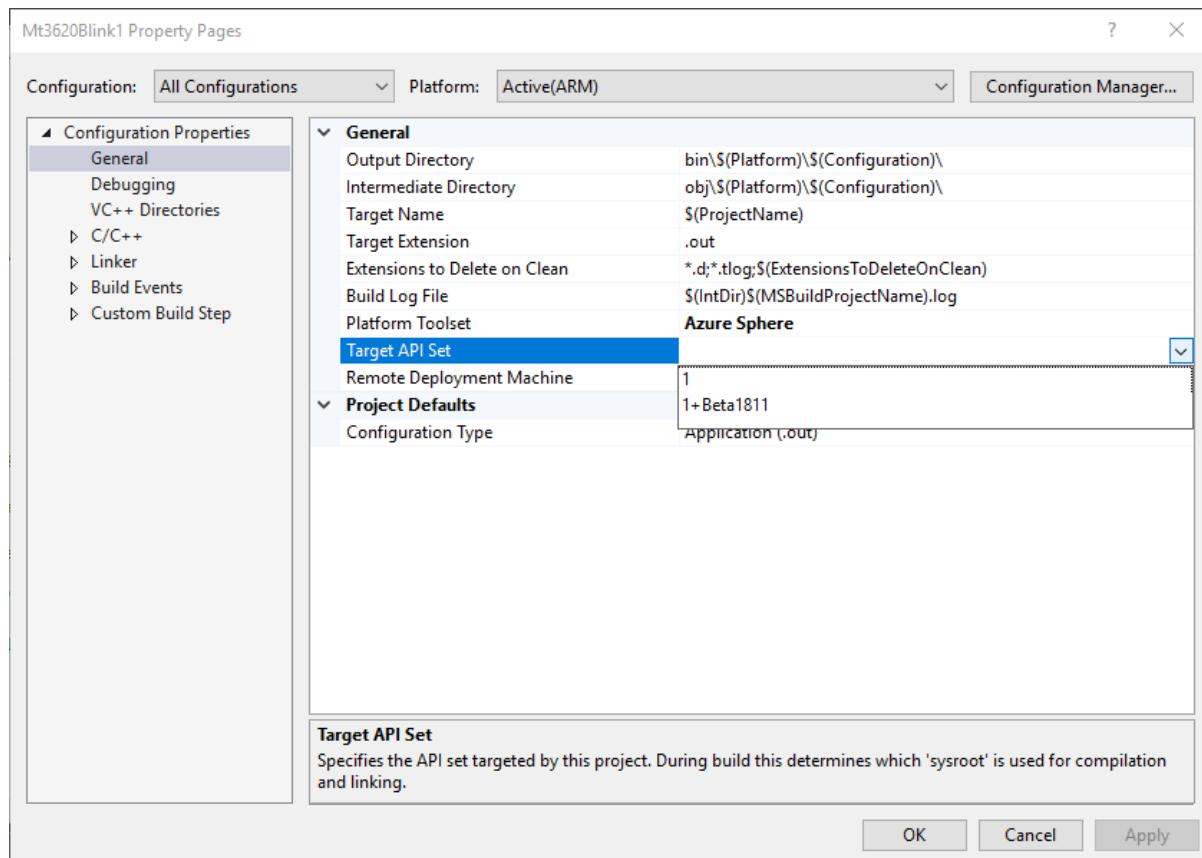
To enable you to develop applications that use either the production APIs or the production and Beta APIs, this Azure Sphere release adds the **Target API Set** property to the **Project Properties** in Visual Studio. The Target API Set value 1 selects only production APIs; the value 1+Beta1811 selects both production and Beta APIs for the current release.

Attempts to rebuild existing TP 4.2.1 applications with the 18.11 Azure Sphere SDK Preview for Visual Studio will initially fail with the following error:

'Target API Set' project property is not set. Open the Project Properties and set this field using the drop-down menu. Ensure that the Configuration selected on that page includes the active build configuration (e.g. Debug, Release, All Configurations).

To correct this error:

1. Open the Azure Sphere project in Visual Studio.
2. On the **Project** menu, select *project-name* **Properties**.
3. Ensure that the Configuration is set to either All Configurations or Active (*configuration name*).
4. Select **Target API Set**. On the drop-down menu, choose 1 to use production APIs or 1+Beta1811 to use production and Beta APIs.



5. Click **OK**.

You can then rebuild the project.

For additional information about Beta APIs and the target API set, see [Beta features](#).

### Strict prototype checking

The Visual Studio compiler settings now include strict prototype checking by default. As a result, the **gcc** compiler returns a warning for a function that uses empty parentheses () instead of (**void**) to indicate that it has no arguments.

To eliminate the warning in existing Azure Sphere applications, edit the function signature to add **void**. For

example, change `int foo()` to `int foo(void)`.

## Wi-Fi setup using Bluetooth low-energy (BLE)

This release includes a reference solution that shows how you can [use a mobile device to configure Wi-Fi](#) on an Azure Sphere-based device. The BLE solution requires the Windows 10 Fall Creators Edition, which provides additional required Bluetooth support.

## Real-time clock

A Beta API enables applications to set and [use the internal clock](#) and leverages support for using a coin-cell battery to ensure the RTC continues to keep time when power is lost.

### IMPORTANT

The RTC must have power for the MT3620 development board to operate. This requires a jumper header on pins 2 and 3 of J3, or a coin cell battery and a jumper on pins 1 and 2 of J3. See [Power Supply](#) for details.

## Mutable storage

A Beta API provides access to a maximum of 64k for storage of [persistent read/write data](#).

## Private Ethernet

You can connect Azure Sphere to a private Ethernet network by using a Microchip Ethernet part over a serial peripheral interface (SPI). This functionality enables an application that runs on the A7 chip to communicate with devices on a private, 10-Mbps network via standard TCP or UDP networking in addition to communicating over the internet via Wi-Fi.

## External MCU update

A reference solution shows how your application can [update the firmware of additional connected MCUs](#).

## Software update improvements

The Azure Sphere security service now seamlessly handles expired root certificates to ensure that devices that are intermittently connected or are disconnected for long periods of time can always connect to Azure Sphere and securely update.

## Ping command

Although the **ping** command previously worked to contact the Azure Sphere device over a USB connection, this unsupported functionality has been removed. Use the **azsphere device show-attached** command to verify device connectivity.

## Known issues

This section lists known issues in the current release.

### Feed list command fails

After the 18.11 release, the **azsphere feed list** command from the TP 4.2.1 SDK fails with the following message:

```
azsphere feed list
Listing all feeds.
error: Invalid argument provided.
error: Command failed in 00:00:01.2002890.
```

The reason for this failure is the addition of a new feed type at the 18.11 release. To avoid this error, update your Azure Sphere device OS and SDK to the 18.11 release.

## Non-ASCII characters in paths

The Azure Sphere tools do not support non-ASCII characters in paths.

## **Development language**

The Azure Sphere SDK supports application development only in C.

Currently, the Visual Studio Integrated Development Environment (IDE) does not generate an error if you add a C++ source file to an Azure Sphere project. However, C++ development is not supported, and the resulting project will not build correctly.

# TP 4.2.1 Release Notes

9/13/2018 • 2 minutes to read

Technical Preview 4.2.1 (August, 2018) includes the changes and known issues described in this article.

## New features and changes

In addition to various usability enhancements, this Technical Preview release includes the changes described below.

### Curl support

Curl support has been modified in the following ways:

- Functions that the underlying Azure Sphere OS does not support have been removed, such as those that require writable files (cookies) or operate on UNIX sockets.
- Additional functions, such as the **mprintf()** family, have also been removed because they will not be supported in future libcurl releases.
- Server authentication is now supported, so that applications can now verify that they are communicating with the expected server. The server's certificate must be signed by a Certificate Authority (CA) that the device trusts. Several CAs are built into the Azure Sphere device. In addition, you can add a certificate to your application image package.

Clients should use existing curl mechanisms such as curl\_version\_info or checking the return code from curl\_easy\_setopt to determine whether a particular feature is supported.

### Device update status

The azsphere.exe command-line utility now includes the **azsphere device show-ota-status** command, which returns information about which version of the Azure Sphere OS your device is running and whether an OS update is available or being downloaded.

## Known issues

The following are known issues in this Technical Preview release.

### Development language

The Azure Sphere SDK supports application development only in C.

Currently, the Visual Studio Integrated Development Environment (IDE) does not generate an error if you add a C++ source file to an Azure Sphere project. However, C++ development is not supported, and the resulting project will not build correctly.

### Status LEDs

The application status and Wi-Fi status LEDs on the MT3620 Reference Board are provided for customer application use. The system software does not control these LEDs automatically.

# Support

6/20/2019 • 2 minutes to read

Comprehensive support options are available to meet your needs, whether you are getting started or already deploying Azure Sphere.

- Community Support: Engage with Microsoft engineers and Azure Sphere community experts.
  - For product-related questions: [MSDN Forum](#)
  - For development questions: [StackOverflow](#)
- Feedback: Submit [product feedback or new feature requests](#)
- Azure Sphere Assisted Support: One-on-one technical support for Azure Sphere is available for customers who have an Azure Subscription that is associated with an Azure Support Plan.
  - Already have an Azure Subscription with Azure Support plan? [Sign in](#) to submit a support request.
  - See [Azure subscription options](#)
  - Select an [Azure support plan](#)
- Service Notifications: [Service notifications and updates](#) from the Azure Sphere Engineering team.

# Additional Resources

2/14/2019 • 2 minutes to read

- [Azure Sphere website](#)
- [The Seven Properties of Highly Secure Devices](#)
- [Azure Sphere Device Authentication and Attestation Service](#)