

Contents

Azure Sphere Documentation

Overview

[What is Azure Sphere](#)

[Usage scenario](#)

[Azure Sphere security](#)

[Architecture](#)

Install

[Overview](#)

[Set up Azure Sphere device](#)

[Update the OS](#)

[Set up an account](#)

[Claim your device](#)

[Configure Wi-Fi for development](#)

[Subscribe to notifications](#)

[Manage tenant and application access](#)

[Create an Azure Sphere tenant](#)

[Limit access to your tenant](#)

[Obtain admin approval](#)

Quickstarts

[Overview](#)

[Build an application](#)

[Deploy an application over Wi-Fi](#)

Networking

[Connect a device to Wi-Fi](#)

[Connect to private Ethernet](#)

[Networking requirements](#)

Application Development

[Overview](#)

[Concepts](#)

- [Application platform](#)
- [Lifecycle of an application](#)
- [Development environment](#)
- [Application manifest](#)

Tutorials

- [Use Azure IoT with Azure Sphere](#)
- [Set up IoT hub](#)
- [Set up IoT Central](#)
- [Run Azure IoT Hub sample](#)
- [Run UART sample](#)

How To

- [Use Beta APIs](#)
- [Sideload for testing](#)
- [Remove an application](#)
- [Determine application memory usage](#)
- [Develop without Visual Studio](#)
- [Connect to web services](#)
- [Manage time and use the real-time clock](#)
- [Use I2C](#)
- [Use SPI](#)
- [Use device storage](#)
- [Troubleshooting](#)

Best practices

- [Initialization and termination](#)
- [Pass arguments](#)
- [Periodic tasks](#)
- [Asynchronous events, threads, and concurrency](#)
- [Watchdog timers](#)
- [Handle and log errors](#)
- [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

- [Deploy an application](#)
- [Prepare the device for updates](#)
- [Link to feed](#)
- [Update a deployment](#)
- [Recover the system software](#)
- [Update an external MCU](#)

Reference

- [azsphere command-line utility](#)
- [component](#)
- [device](#)
- [device-group](#)
- [feed](#)
- [get-support-data](#)
- [image](#)
- [image-set](#)
- [login](#)
- [logout](#)
- [show-version](#)
- [sku](#)
- [tenant](#)

Application Libraries reference

- [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_GetInterfaceCount](#)

[Networking_GetInterfaces](#)

[Networking_GetNtpState](#)

[Networking_InitDhcpServerConfiguration](#)

[Networking_InitStaticIpConfiguration](#)

[Networking_IsNetworkingReady](#)

[Networking_SetInterfaceState](#)

[Networking_SetNtpState](#)

[Networking_SetStaticIp](#)

[Networking_StartDhcpServer](#)

[Networking_StartSntpServer](#)

Structs

[Networking_DhcpServerConfiguration](#)

[Networking_NetworkInterface](#)

[Networking_StaticIpConfiguration](#)

Enumerations

[Networking_InterfaceMedium](#)

[Networking_IpConfiguration](#)

Typedefs

[Networking_InterfaceMedium_Type](#)

[Networking_IpConfiguration_Type](#)

rtc.h

Functions

[clock_systohc](#)

spi.h

Enumerations

[SPI_BitOrder](#)

[SPI_ChipSelectPolarity](#)

[SPI_Mode](#)

[SPI_TransferFlags](#)

Functions

[SPI_SPIMaster_InitConfig](#)

[SPI_SPIMaster_InitTransfers](#)

[SPI_SPIMaster_Open](#)

[SPI_SPIMaster_SetBitOrder](#)

[SPI_SPIMaster_SetBusSpeed](#)

[SPI_SPIMaster_SetMode](#)

[SPI_SPIMaster_TransferSequential](#)

[SPI_SPIMaster_WriteThenRead](#)

Structs

[SPIMaster_Config](#)

[SPIMaster_Transfer](#)

Typedefs

[SPI_ChipSelectId](#)

[SPI_InterfacId](#)

storage.h

Functions

[Storage_DeleteMutableFile](#)

[Storage_GetAbsolutePathInImagePackage](#)

[Storage_OpenFileInImagePackage](#)

[Storage_OpenMutableFile](#)

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_ForgetAllNetworks](#)

[WifiConfig_ForgetNetwork](#)

[WifiConfig_GetCurrentNetwork](#)

[WifiConfig_GetScannedNetworks](#)

[WifiConfig_GetStoredNetworkCount](#)

[WifiConfig_GetStoredNetworks](#)

[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 support status](#)

[MT3620 development board user guide](#)

[MT3620 reference board design](#)

[MCU programming and debugging interface](#)

RF test tools

Manufacturing guide

[Factory floor tasks](#)

[Cloud configuration tasks](#)

Resources

[Samples and Reference Solutions](#)

[Release Notes](#)

[Release Notes 19.02](#)

[Release Notes 18.11](#)

[Release Notes TP 4.2.1](#)

[Support](#)

[Additional Resources](#)

Welcome to Azure Sphere

11/15/2018 • 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 Wi-Fi](#)
- [Subscribe to notifications](#)

Then follow these quickstarts to build and deploy an application:

- [Build your first application](#)
- [Deploy your application over Wi-Fi](#)

Concepts

- [Azure Sphere overview](#)
- [Application platform](#)
- [Deployment model](#)

Need a development kit?

[Order one!](#)

What is Azure Sphere?

8/13/2018 • 2 minutes to read

Azure Sphere is a secured, high-level application platform with built-in communication and security features for internet-connected devices.

Azure Sphere introduces a new class of secured, connected, crossover microcontroller unit (MCU), which 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 product manufacturers to create secured, internet-connected devices that can be updated, controlled, monitored, and maintained remotely. By embedding the MCU in a connected device, either alongside or in place of existing MCU(s), product manufacturers gain 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.

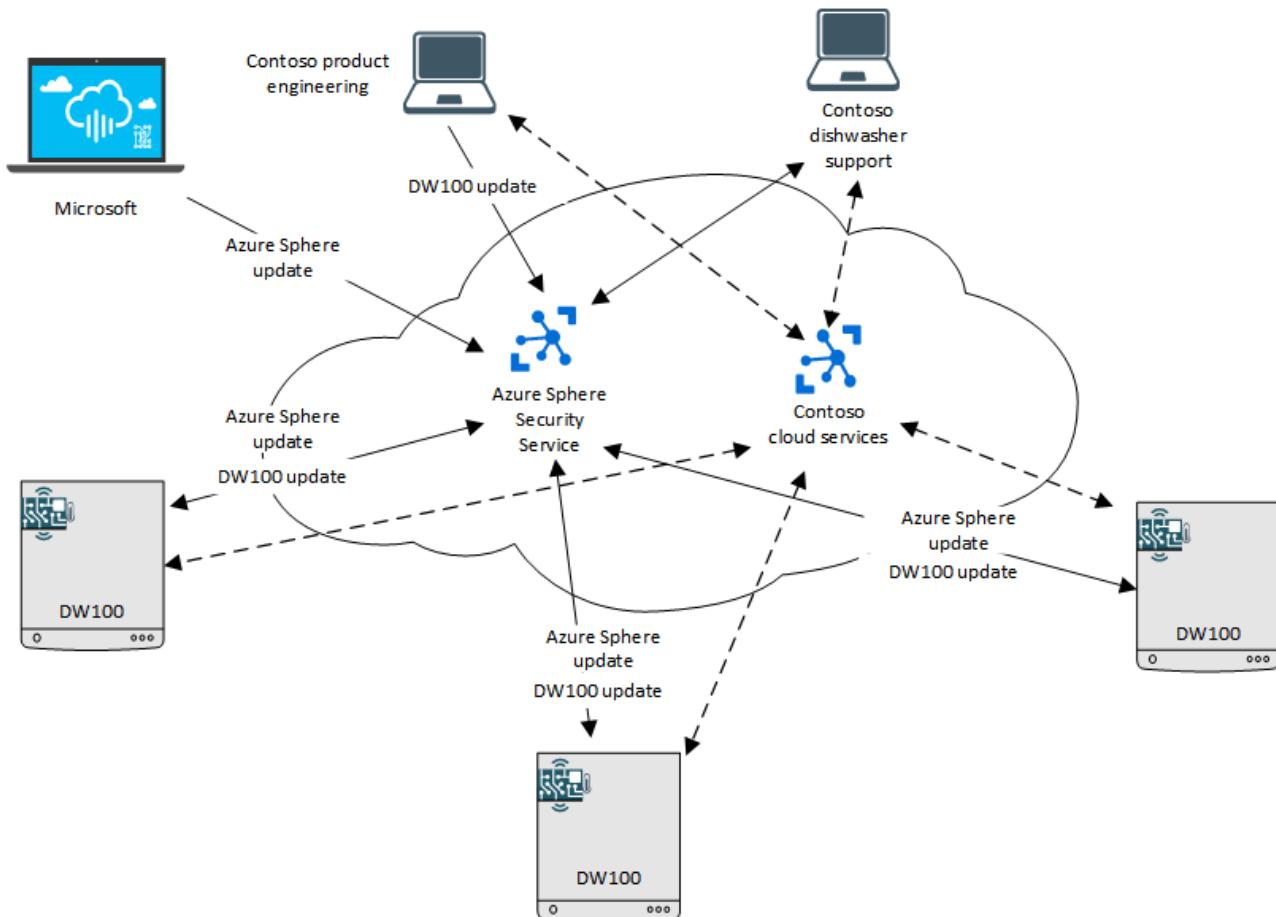
The Azure Sphere Security Service is an integral aspect of Azure Sphere. Using this service, Azure Sphere MCUs safely and securely connect to the cloud and web. The service ensures that the device boots only with an authorized version of genuine, approved software. In addition, it provides a secured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

Azure Sphere scenario

8/24/2018 • 2 minutes to read

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 application that runs on the Azure Sphere MCU. The onboard 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 device software 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 Microsoft device software 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 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

9/13/2018 • 2 minutes to read

The Azure Sphere platform aims to provide high-value security at a low cost, so that price-sensitive, microcontroller-powered devices can safely and reliably connect to the IoT. 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 Azure Sphere OS and OEM 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

11/15/2018 • 7 minutes to read

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

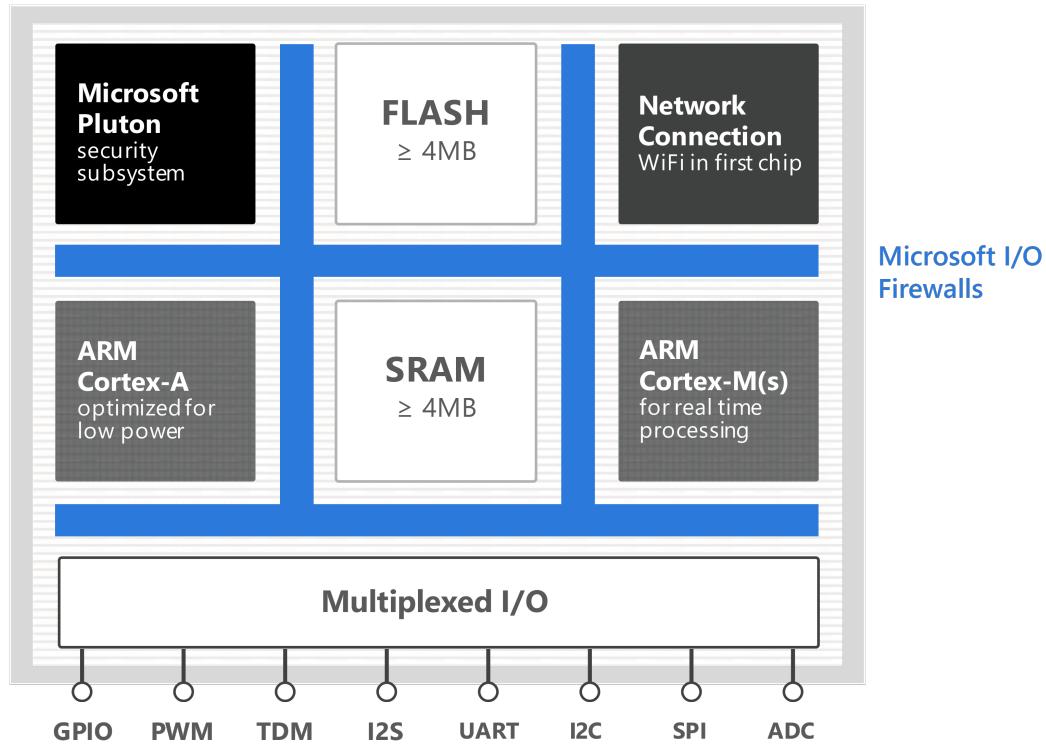
The hardware architecture provides a fundamentally secured computing base for devices on the IoT, allowing product manufacturers to focus on their expertise: product-specific features and enhancements.

The software architecture, with a secured custom OS kernel running atop the Microsoft-written Security Monitor, similarly enables manufacturers to concentrate their software efforts on value-added IoT features that benefit their customers and shareholders.

The Azure Sphere Security Service supports authentication, software update, and failure reporting over secured cloud-to-device and device-to-cloud channels, so that manufacturers can rely on a secured communications infrastructure and can be assured that their 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 (i.e., 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.

Application processor

The application processor 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, 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. A customer-developed application runs in NW user mode on the application processor.

Real-time processor(s)

The real-time processor(s) feature an ARM Cortex-M I/O subsystem that uses either bare-metal code or a real-time operating system (RTOS).

Connectivity and communications

All Azure Sphere MCUs include a wireless communications subsystem with which they can connect to internet services. The first Azure Sphere MCU provides an 802.11 b/g/n Wi-Fi radio operating at both 2.4GHz and 5GHz. Customer applications can configure, use, and query the wireless communications subsystem, but they cannot program it directly.

Multiplexed I/O

The Azure Sphere platform supports a variety of I/O capabilities, so that manufacturers can configure embedded devices to suit their market requirements. I/O peripherals can be mapped to either the ARM Cortex-A or to an ARM Cortex-M I/O core.

Microsoft firewalls

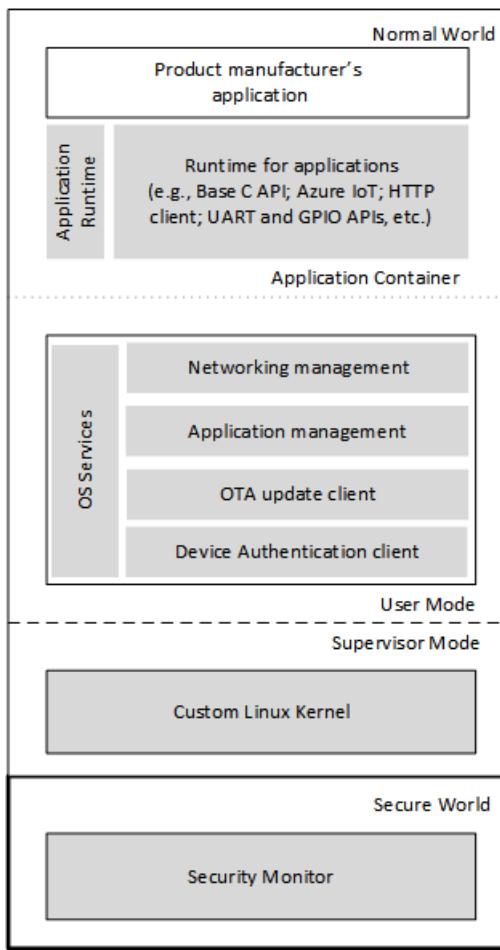
Hardware firewalls are silicon countermeasures that provide “sandbox” protection to ensure that I/O peripherals are accessible only to the core to which they are mapped. The firewalls impose compartmentalization, thus preventing a security threat that is localized in the A7 core from affecting the I/O M4 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.

Cortex-A software architecture

The ARM Cortex-A (A7) application platform is the first customer-programmable core to be available on the first Azure Sphere MCU. The following figure shows the elements of this platform. Microsoft-supplied elements are shown in gray, and product manufacturer-supplied elements are shown in white.



Cortex-A Software Platform

Microsoft provides and maintains all Cortex-A software other than the product manufacturer's application. All software that runs on the device, including the 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.

A7 application

The product manufacturer develops a custom A7 application that runs in NW user mode. Such an application can:

- Configure and interact with peripherals integrated into the Azure Sphere MCU, such as the general-purpose input/output (GPIO) pins, universal asynchronous receiver/transmitters (UARTs), and other interfaces
- Communicate with an Azure IoT hub or other cloud-based services
- Broker trust relationships with other devices and services via certificate-based authentication
- Communicate with line of business (LOB) services

The product manufacturer's application runs in an application container on the A7 core; it has access only to libraries and runtime services that Microsoft provides. Furthermore, as part of compartmentalized security, applications must specify the external services and interfaces that they require—for example, their I/O and network requirements—to prevent any unauthorized or unexpected use. To avoid corruption caused by outside software or malicious data, A7 applications cannot perform generic file I/O or interprocess communication (IPC). After authentication, however, they may interact with cloud services, or may use the http and https libraries that are supplied with the Azure Sphere software development kit (SDK) to interact with other internet services.

Only one A7 application runs on the device at a time. A7 applications are expected to run continuously and are automatically restarted if they stop or fail.

To prevent the installation of rogue software, applications can be loaded in only two ways:

- Sideload for software development and testing. Sideload requires direct access to the device and authentication with the Azure Sphere Security Service.
- Over-the-air update, which can be performed only by the Azure Sphere Security Service.

All deployments are signed to certify their authenticity to the device.

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 A7 application and Azure Sphere Security Service.

Application libraries do not support direct generic file I/O, IPC, 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 with customer applications.

OS services

OS services host the application container and are responsible for communicating with the Azure Sphere Security Service. They manage Wi-Fi authentication and the network firewall for all outbound traffic. During application 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, 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.

Software development experience

Azure Sphere provides a rich development experience based on Microsoft Visual Studio. The Visual Studio Extension for Azure Sphere supports IntelliSense and breakpoint-based debugging from a PC, along with an extension to simplify the use of an Azure IoT Hub. Sample applications and project templates are also included.

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 web services such as Microsoft Azure. 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 all Microsoft-supplied Azure Sphere OS and OEM software. The update service works with the cloud and device utilities in the Azure Sphere SDK to ensure continued operation and to enable manufacturers to update and service their application software remotely.
- **Failure reporting.** The planned failure reporting service provides simple crash reporting for deployed software. To obtain richer data, manufacturers can use the reporting and analysis features that are included with their Microsoft Azure subscription.

In general, Azure Sphere applications can use as much or as little of the Microsoft Azure functionality as their designers choose. Simple applications do not require any cloud-related code. A more typical application might monitor device operations and report data to an Azure IoT hub, to the manufacturer's cloud service application, or to the manufacturer's secured website.

Install Azure Sphere

2/14/2019 • 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 board
- [Update the OS](#)
- [Set up an account](#) to authenticate with Microsoft Azure
- [Claim your device](#)
- [Configure Wi-Fi](#)
- [Subscribe to Azure Sphere notifications](#) so that you receive the latest information about OS updates and other important news

Set up your board and install the Azure Sphere SDK

2/14/2019 • 2 minutes to read

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

Prerequisites:

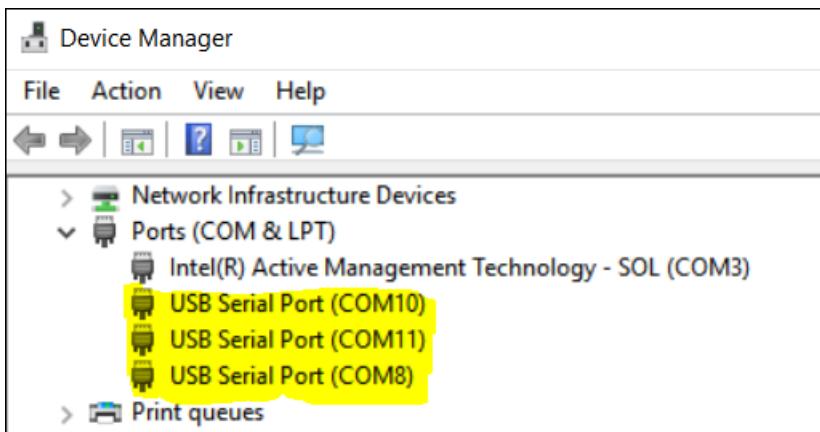
- An [Azure Sphere Development Kit](#)
- A PC running Windows 10 Anniversary Update or later
- An unused USB port on the PC
- Visual Studio 2017 Enterprise, Professional, or Community, version 15.7 or later. To [install Visual Studio](#), select the edition to install, and then run the installer. You can install any workloads, or none.

Connect the board

The development board connects to a PC through USB. When plugged in, the board exposes three COM ports.

The first time you plug in the board, 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, see [Troubleshooting](#) for help.

Install the Azure Sphere SDK Preview for Visual Studio

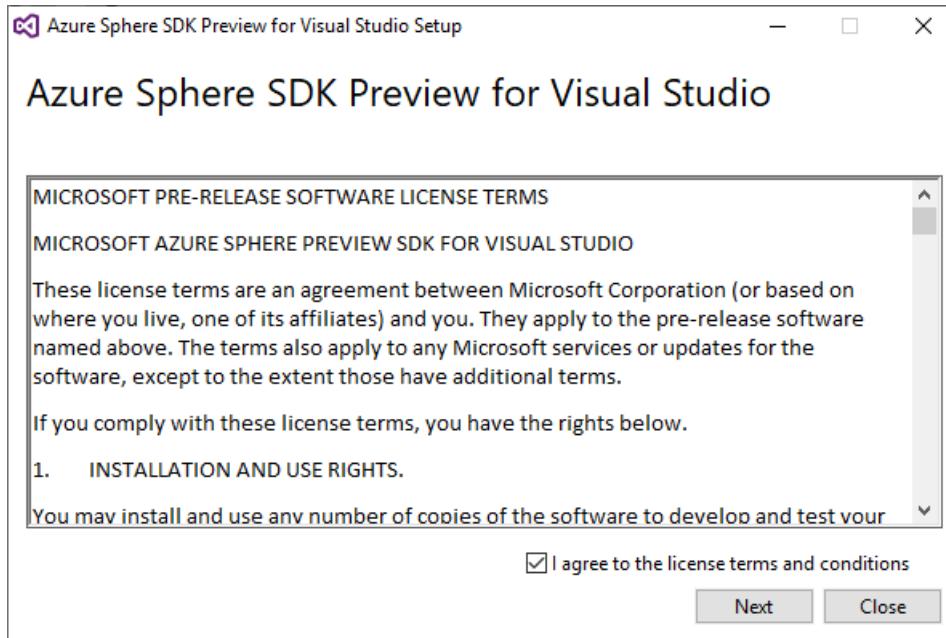
The Azure Sphere SDK Preview for Visual Studio includes:

- 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

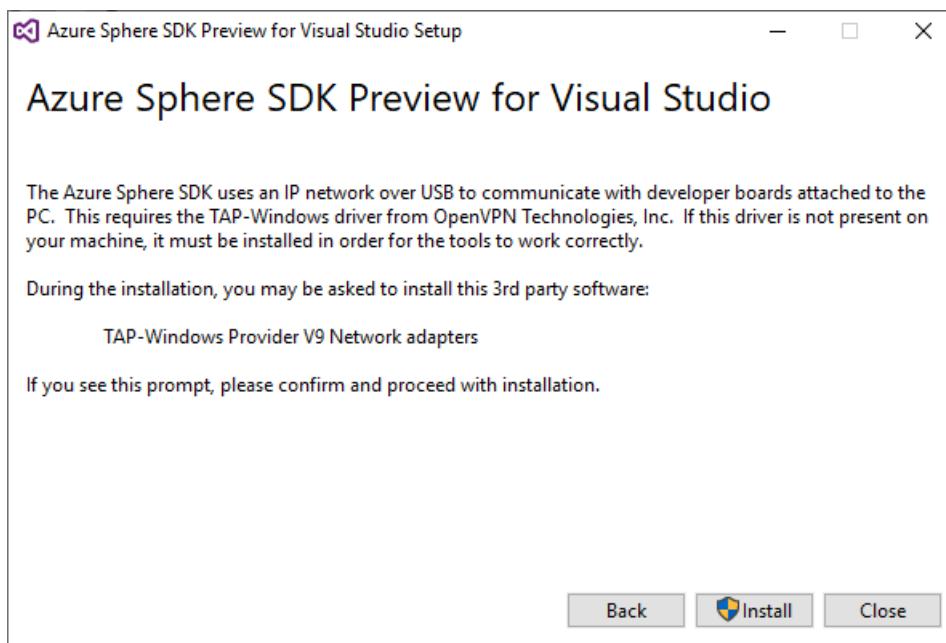
Azure_Sphere_SDK_Preview_for_Visual_Studio.exe installs the complete Azure Sphere software development kit (SDK).

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.
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.



If the message "This product requires Visual Studio 2017, Version 15.7 or newer" appears, ensure that Visual Studio 2017 version 15.7 or more recent is installed on your PC.

The message "No product to install SDK on" may appear if you do not have Visual Studio version 15.7 or newer installed, 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. The SDK requires Visual Studio version 15.7 or later.

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](#)

Update the OS

2/14/2019 • 2 minutes to read

All Azure Sphere devices are shipped from the manufacturer with the Azure Sphere OS installed. If your device is already in use and is running the 18.11 OS release (or more recent), it should receive the updated OS over the air if it is connected to the internet. You do not need to do anything to update the OS.

Currently, newly delivered MT3620 development boards require manual update. If you have an Azure Sphere device that has never been used, follow the steps in this section to update the Azure Sphere OS.

1. Connect the board to the PC by USB.
2. Open an Azure Sphere Developer Command Prompt. It appears in the **Start** menu under **Azure Sphere**.
3. Issue the following command to 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.
```

Next Steps

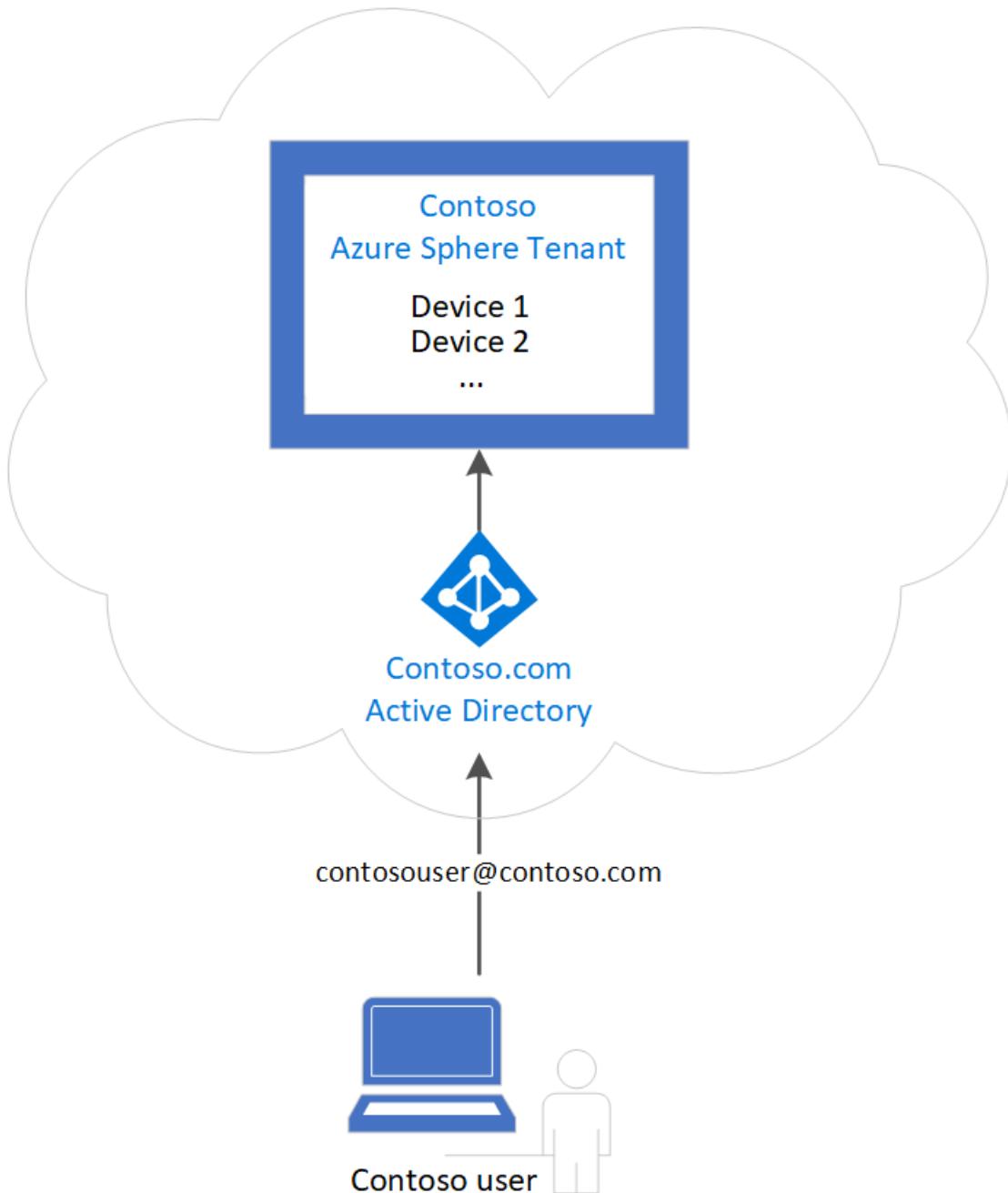
- [Set up an account](#) to use with Azure Sphere

Set up an account for Azure Sphere

2/14/2019 • 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 [Understanding Azure identity solutions](#).

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, 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.

Be aware, however, that 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](#).

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. An Azure subscription is not required to use Azure Sphere, but a subscription is required to use Azure IoT Hub or Azure IoT Central.

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. 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, the sidebar includes options like 'Create a resource', 'All services', and a 'FAVORITES' section with links to '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, followed by two sections: 'OVERVIEW' (with 'Overview' and 'Getting started' items) and 'MANAGE' (with 'Users' (highlighted with a red box), '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 and set the role for the user. If this user will manage access to your Azure Sphere applications and devices, select the Global Administrator role. 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

The screenshot shows the 'User' creation dialog in the Microsoft Azure portal. At the top, the navigation path is 'Home > default directory > Users - All users > User'. The title bar says 'User' and 'default directory'. The main area contains two required fields: 'Name' (with example 'Chris Green') and 'User name' (with example 'chris@contoso.com'). Below these are four configuration sections: 'Profile' (set to 'Not configured'), 'Properties' (set to 'Default'), 'Groups' (showing '0 groups selected'), and 'Directory role' (set to 'User'). At the bottom is a prominent 'Create' button.

Next steps

- [Claim your device](#)

Claim your device

2/14/2019 • 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.

If login succeeds, you should see a message informing you whether an Azure Sphere tenant exists for this directory. If no tenant exists, [create one](#).

4. 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.  
Claiming attached device ID  
'ABCDE082513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A558  
8B420851EE4F3F1A7DC51399ED' into tenant ID 'd343c263-4aa3-4558-adbb-d3fc34631800'.  
Successfully claimed device ID  
'ABCDE082513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A558  
8B420851EE4F3F1A7DC51399ED' into tenant ID 'd343c263-4aa3-4558-adbb-d3fc34631800'.  
Command completed successfully in 00:00:05.5459143.
```

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

Next steps

- [Configure Wi-Fi](#)

Configure Wi-Fi and update your device

2/14/2019 • 2 minutes to read

After you claim your Azure Sphere device, configure it for Wi-Fi 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 Wi-Fi, you must:

- [Install the SDK and set up the development board](#)
- [Update the OS](#)
- [Claim the device](#)

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:

```
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> --key <yourNetworkKey>
```

Replace <yourSSID> with the name of your network and <yourNetworkKey> with your WPA/WPA2 key. Azure Sphere devices do not support WEP. Network SSIDs are case-sensitive. For example:

```
azsphere device wifi add --ssid My5GNetwork --key secretnetworkkey
```

To add an open network, omit the --key flag.

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" --key "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 Wi-Fi, 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.

Update the device

When Wi-Fi 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

11/15/2018 • 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 new Azure Sphere product features, software updates, termination of support for older preview OS versions, and other useful information is posted on the [Azure Updates](#) 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, try building and deploying a simple application or explore the use of the IoT:

- [Build the Blink sample](#)
- [Deploy an application over the air](#)
- Explore the [Azure IoT sample](#)

Create an Azure Sphere tenant

2/14/2019 • 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 by using the following command in an Azure Sphere Developer Command Prompt:

```
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 can create only one tenant per device.

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.

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 applications. The first two rows are for 'Azure Sphere API' and 'Azure Sphere Utility', both of which have 'AS' icons next to them. The table columns include NAME, HOMEPAGE URL, OBJECT ID, and APPLICATION ID.

NAME	HOMEPAGE URL	OBJECT ID	APPLICATION ID
AS Azure Sphere API	https://www.microsoft.com/en-us/azure-sphere/	52e51a2f-59fc-46bf-86b2-9f870b2028f	7a663687-509b-4592-ab89-e20...
AS Azure Sphere Utility	https://www.microsoft.com/en-us/azure-sphere/	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 [?](#)

Object ID [?](#)

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

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

Manage

- [Properties](#) Selected
- [Owners](#)
- [Users and groups](#)
- [Provisioning](#)
- [Self-service](#)

Security

- [Conditional Access](#)
- [Permissions](#)

Activity

- [Sign-ins](#)
- [Audit logs](#)

Troubleshooting + Support

- [Virtual assistant \(Preview\)](#)
- [Troubleshoot](#)
- [New support request](#)

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, there's a warning message: 'Groups are not available for assignment due to your Active Directory plan level.' Below it, two sections are visible: 'Users' (None Selected) and 'Select Role' (Default Access). On the right, a list of users is shown with their initials in colored circles (PO, PE, TE) followed by grayed-out names.

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

Selected members:
No members selected

Assign **Select**

- 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 includes 'Overview', 'Getting started', 'Properties', 'Owners', 'Users and groups' (selected), and 'Provisioning'. The main area displays a table of users with columns for 'DISPLAY NAME', 'OBJECT TYPE', and 'ROLE ASSIGNED'. A note at the top says: 'The application will appear on the access panel for assigned users. Set 'visible to users?' to no in properties to prevent this.'

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

Obtain admin approval

2/14/2019 • 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. Click the following link and sign in with an AAD administrator account:

https://login.microsoftonline.com/common/adminconsent?client_id=0b1c8f7e-28d2-4378-97e2-7d7d63f7c87f&redirect_uri=https://docs.microsoft.com/azure-sphere/install/admin-consent-success

2. In the dialog box that appears, check the box and click **Accept**. If consent succeeds, you will be redirected to a [success page](#).

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](#).

Quickstarts for Azure Sphere

9/28/2018 • 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 application](#) by using the Azure Sphere SDK Preview for Visual Studio
- [Deploying the application](#) over Wi-Fi

Quickstart: Build the Blink sample application

2/14/2019 • 4 minutes to read

This quickstart shows how to enable application development on an Azure Sphere device and how to build and debug a sample application. It uses the Blink sample, which is part of the Azure Sphere SDK. The Blink sample shows how to access GPIOs and LEDs on the development board.

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](#)

Prepare your device for development and debugging

Before you can run the sample applications on your Azure Sphere device or develop new applications for it, you must enable development and debugging on the board. 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. Prepping 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
'ABCDE082513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A558
8B420851EE4F3F1A7DC51399ED'.
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
'ABCDE082513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A558
8B420851EE4F3F1A7DC51399ED'.
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
'ABCDE082513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A558
8B420851EE4F3F1A7DC51399ED' for application development, and disabled over-the-air updates.
Command completed successfully in 00:00:38.3299276.

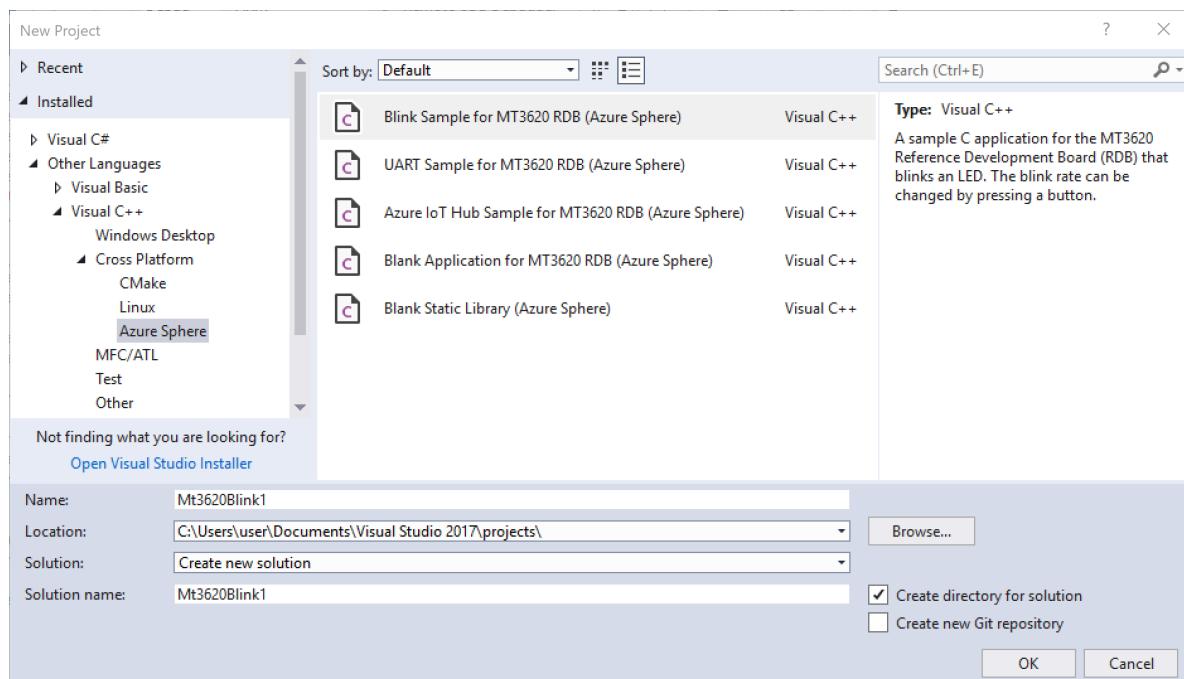
```

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.

You can also create your own device group that does not enable OTA application updates. See [azsphere device-group create](#) and [azsphere device update-device-group](#) for details.

Build and run the Blink sample

- Start Visual Studio 2017 and go to **File>New>Project**. The templates for Azure Sphere are available in **Visual C++>Cross Platform>Azure Sphere**. Select **Blink Sample for MT3620 RDB (Azure Sphere)**.



- Enter a name and location for the project and select **OK**.

The project opens with two windows in the editor: main.c and Mt3620Blink. The MT3620Blink window is a Visual Studio Overview page for the project, which links to information about your device and Azure

Sphere application development.

3. Navigate to the line that tests the value of newButtonState and press **F9** to set a breakpoint:

```
if (newButtonState == GPIO_Value_Low) {
```

4. Ensure that your board is connected to your PC by USB. Then select **Remote GDB Debugger** from the menu bar or press **F5**.



5. 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) by Wi-Fi.

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](#).

6. Press button A. Visual Studio stops at the breakpoint that you set. Open **Debug>Windows** and select **Autos** to display the variables that are used in the current and previous statements. Visual Studio populates the call stack and the auto variables after program execution stops at the breakpoint. In the following example, the value of the newButtonState variable is 0, equal to GPIO_Value_Low, which indicates a button press.

A screenshot of the Visual Studio 'Autos' window. It displays three variables in a table:

Name	Value	Type
GPIO_Value_Low	GPIO_Value_Low	enum {...}
buttonState	0x1	GPIO_Value_Type
newButtonState	0x0	GPIO_Value_Type

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. Select **Continue**. Execution pauses again at this breakpoint. Now the value of newButtonState variable is 1, equal to GPIO_Value_High, which indicates a button release. Press **F9** to remove the breakpoint. Now, each time you press and release Button A, the blink rate changes.
9. When you are done debugging, select the **Stop** icon in the menu bar or press **Shift+F5**.

Next steps

This quickstart showed how to prepare your Azure Sphere device for debugging and how to build and debug an application.

Next, learn how to [deploy the application over the air](#).

Quickstart: Deploy an application over the air

2/14/2019 • 5 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
- You have completed all the steps to [install Azure Sphere](#)
- You have completed [Quickstart: Build the Blink sample 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> --newsku <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 --newsku <unique-sku-name> 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 FieldGroupOTA that enables OTA application updates and assigns the attached device to it. It also creates a new product SKU named FieldProductSKU and assigns that SKU to the device.

```
azsphere device prep-field --newdevicegroupname "FieldGroupOTA" --newsku "FieldProductSKU"
Removing applications from device.
Successfully removed applications from device.
Locking device.
Downloading device capability configuration for device ID
'ABCDEF82513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A5588B4208
51EE4F3F1A7DC51399ED'.
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 'FieldGroupOTA'.
Setting device group ID '655d7b12-07ad-4e8a-b104-c0ec494b8489' for device with ID
'ABCDEF82513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A5588B4208
51EE4F3F1A7DC51399ED'.
Creating a new SKU with name 'FieldProductSKU'.
Setting product SKU to '946410a0-0057-4b11-af68-d56a684f6681' for device with ID
'ABCDEF82513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A5588B4208
51EE4F3F1A7DC51399ED'.
Successfully set up device
'ABCDEF82513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A5588B4208
51EE4F3F1A7DC51399ED' for OTA loading.
Command completed successfully in 00:00:15.0013734.
```

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 Blink application. 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 sample 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 sample application](#), the full path to the image file is displayed in the Visual Studio 2017 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\Test\Documents\Visual Studio 2017\Projects\Mt3620Blink2\Mt3620Blink2\bin\ARM\Debug\Mt3620Blink2.imagepackage" --newfeedname "BlinkFeed"
Getting the details for device with ID
'ABCDEF82513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A558
8B420851EE4F3F1A7DC51399ED'.
Uploading image from file 'C:\Users\Test\Documents\Visual Studio 2017\Projects\Mt3620Blink2\Mt3620Blink2\bin\ARM\Debug\Mt3620Blink2.imagepackage':
--> Image ID: '116c0bc5-be17-47f9-88af-8f3410fe7efa'
--> Component ID: '54acba7c-7719-461a-89db-49c807e0fa4d'
--> Component name: 'Mt3620Blink2'
Create a new feed with name 'BlinkFeed'.
Adding feed with ID 'ce680169-d893-49de-bb02-5f2c40c52932' to device group with ID '655d7b12-07ad-4e8a-b104-c0ec494b8489'.
Creating new image set with name 'ImageSet-Mt3620Blink2-2018.07.19-18.15.42+01:00' for image with ID
'116c0bc5-be17-47f9-88af-8f3410fe7efa'.
Adding image set with ID '6e9cdc9d-c9ca-4080-9f95-b77599b4095a' to feed with ID 'ce680169-d893-49de-bb02-5f2c40c52932'.
Successfully linked device
'ABCDEF82513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A558
8B420851EE4F3F1A7DC51399ED' to feed with ID 'ce680169-d893-49de-bb02-5f2c40c52932'.
Command completed successfully in 00:00:12.8912364.
```

This command creates a feed that is linked to the device group to which the attached device belongs—that is, the FieldGroupOTA created [earlier in this quickstart](#). It will deliver the Mt3620Blink2 application to all Azure Sphere devices in the group whose product SKU is FieldProductSKU.

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 a couple of minutes, and you should see LED 1 start to blink.

To verify that the application was installed on your device:

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

This command returns the following information about the installed images:

```
Installed images:  
--> gdbserver  
--> Image type: Application  
--> Component ID: 8548b129-b16f-4f84-8dbe-d2c847862e78  
--> Image ID: 43d2707f-0bc7-4956-92c1-4a3d0ad91a74  
--> Mt3620Blink2  
--> Image type: Application  
--> Component ID: 54acba7c-7719-461a-89db-49c807e0fa4d  
--> Image ID: 116c0bc5-be17-47f9-88af-8f3410fe7efa  
Command completed successfully in 00:00:01.5159298.
```

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 IoT Hub](#) and [UART](#) tutorials
- Understand Azure Sphere [deployment basics](#)
- [Update the deployment](#)

Connect Azure Sphere to Wi-Fi

2/14/2019 • 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.

Connect Azure Sphere to private Ethernet

2/14/2019 • 6 minutes to read

An Azure Sphere device can communicate with devices on a private, 10-Mbps Ethernet network via standard TCP or UDP networking in addition to communicating over the internet via Wi-Fi. Your Azure Sphere application will need to use the Azure Sphere networking API to configure the network. The Azure Sphere device does not act as a router in this scenario. It will never automatically pass packets received on the private Ethernet network to the public Wi-Fi network, or vice versa. Your application must implement all logic that sends and receives information on both networks.

Azure Sphere can act as a [dynamic host configuration protocol \(DHCP\) server](#) and a [simple network time protocol \(SNTP\) server](#) for use with private Ethernet. The DHCP server enables Azure Sphere applications to configure network parameters for an external device on the private Ethernet network. Using the SNTP server, the external device can synchronize its time with Azure Sphere.

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 or deploy 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 ENC286J60 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 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 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.

Network configuration

The Azure Sphere application that you use with the Ethernet board configuration image must set up the IP configuration for the private Ethernet network. There is currently no DHCP client support on private Ethernet networks to set this up automatically. To set up the IP configuration, your application must call the configuration functions in `applibs/networking.h`, and the [application manifest](#) must enable the **NetworkConfig** capability.

DHCP server

An external client device that is connected to Azure Sphere through a private Ethernet interface must be assigned an IP address and other network parameters so that it can communicate to 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 **NetworkConfig** capability in its [application manifest](#).

The Azure Sphere application calls **Networking_InitDhcpServerConfiguration()** 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_StartDhcpServer()** 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 **NetworkConfig** capability in its [application manifest](#).

To start the server, the Azure Sphere application calls **Networking_StartSntpServer()** 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 Wi-Fi in addition to the private 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. [Using the real-time clock with 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 private Ethernet network, and not on the public Wi-Fi network.

Ethernet setup sample

The [Private Ethernet sample](#) shows how to connect Azure Sphere to an [Olimex ENC28J60-H module](#). It includes a sample Azure Sphere application that demonstrates how to set up a static IP configuration for the private Ethernet, how to configure and start the DHCP and SNTP servers, and how to use listening sockets to respond to an incoming TCP connection.

Azure Sphere OS networking requirements

2/14/2019 • 2 minutes to read

The Azure Sphere OS and services communicate with devices, Azure IoT Hub, and other services using the endpoints, ports, and protocols listed here.

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, eastus-prod-azuresphere.azure-devices.net	Device provisioning, updates, and communication with IoT Hub
MQTT over TCP	443 (WebSocket)	global.azure-devices-provisioning.net, prodmsimg.blob.core.windows.net, prodpimg.blob.core.windows.net, prod.deviceauth.sphere.azure.net, prod.releases.sphere.azure.net, prod.core.sphere.azure.net	Device provisioning, updates, and communication with IoT Hub
HTTP over TCP	80	prod.update.sphere.azure.net	Check internet connection, download certificate files, and similar tasks
HTTPS over TCP	443	time.sphere.azure.net	Communication with web services, network time (NTP) server
UDP	53		Communication with domain name servers (DNS)
UDP	123		Communication with NTP

Customer 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

2/14/2019 • 2 minutes to read

An Azure Sphere application can:

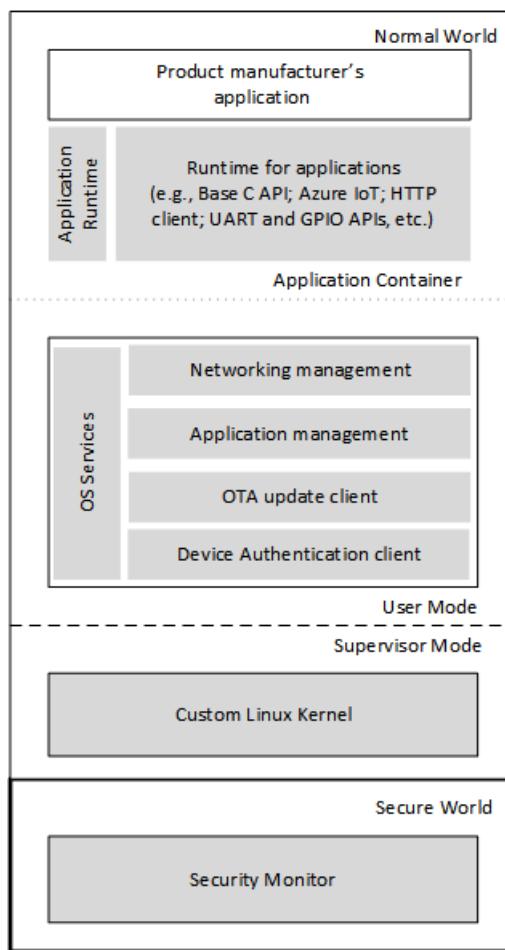
- Drive the general-purpose input/output (GPIO) peripherals
- Perform serial I/O by using the universal asynchronous receiver/transmitters (UART) on the device
- Communicate with an Azure IoT Hub
- Communicate with web services via HTTP/HTTPS by using the curl library
- Use inter-integrated circuit (I2C) or serial-peripheral interface (SPI) to access peripheral devices
- Manage time and use the real-time clock (RTC)
- Use device storage

Application platform

2/14/2019 • 4 minutes to read

Applications that run on a Azure Sphere device operate in a sandboxed container. This environment is designed to maintain the security of software and data and to prevent disruption of service.

The figure shows the software elements of the application platform. Microsoft-supplied elements are gray, and manufacturer-supplied elements are white.



Azure Sphere Application Platform

The application platform supports two operating environments: Normal World and Secure World. Azure Sphere applications run in an application container in Normal World user mode and have access to the Azure Sphere libraries and a limited set of OS services. The underlying custom Linux kernel, which includes Microsoft-supplied device drivers, runs in Normal World supervisor mode, and the Security Monitor runs in Secure World. Only Microsoft-supplied code can run in supervisor mode or in Secure World.

Applications run in containers that support a subset of the POSIX environment. System functions are accessible only through the Azure Sphere libraries and run-time services; the complete set of POSIX system functions is not available.

At the hardware level (not shown in the diagram), a security subsystem provides a hardware-based secure root of trust. At device boot, the security subsystem validates overall hardware security, then brings up other components only after validating that their firmware is correctly signed. Similarly, the firmware validates the security of the next layer of software before loading it, and each subsequent layer of software validates the layer above it.

Application security

The Azure Sphere application platform is designed to ensure the security of the device, the application, and its data. Application security features include:

- Limited access to external resources
- Application capabilities
- Signing requirements
- Device capabilities

Access to resources

To prevent malicious interference from outside software, Azure Sphere applications have access only to libraries and run-time services that Microsoft provides. The libraries are restricted to ensure that the platform remains secure and can be easily updated. They do not support direct file I/O, inter-process communication (IPC), or shell access, among other constraints.

Application capabilities

Application capabilities are the resources that an application requires. Application capabilities include the GPIO and UART peripherals that the application uses, the internet hosts to which it connects, and permission to change the Wi-Fi configuration. Every application must have an [application manifest](#), which identifies these resources.

Signing 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.

Device capabilities

A *device capability* allows a user to perform 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 a user 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) updates.

Memory available for applications

The Azure Sphere platform provides 256 KiB RAM on the A7 core for use by customer applications and 1 MiB read-only flash memory for customer-deployed image packages. Storage up to these limits is reserved entirely for your use.

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.

Applications can also use read/write (mutable) storage. For information about mutable storage, see [Using storage on Azure Sphere](#).

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.

Lifecycle of an application

8/13/2018 • 2 minutes to read

Only one customer application can run on an Azure Sphere device at a time. Applications should be written to run continuously; they should exit only when signaled to do so by the system software. If an application exits unexpectedly, system software automatically restarts it.

During development, you can sideload an application onto the device by using Visual Studio or the [azsphere device sideload deploy](#) command. The sideloaded application remains on the device until one of the following events occurs:

- You build and sideload a new application by using Visual Studio.
- You remove the application from the device by using Visual Studio or the [azsphere device sideload delete](#) command.

When development is complete, you can deploy the application so that the Azure Sphere Security Service will load it. For details about deployment procedures and requirements, see [Over-the-air application deployment](#).

Development environment

11/15/2018 • 3 minutes to read

Application development for the Azure Sphere development board requires the following:

- Windows 10 Anniversary Update or later
- Visual Studio Enterprise, Professional, or Community 2017 version 15.7 or later
- Azure Sphere SDK Preview for Visual Studio
- An Azure Sphere development board that is connected to your PC by USB

The Azure Sphere SDK Preview for Visual Studio includes the full Azure Sphere software development kit (SDK): build and debug tools, libraries, header files, project templates, and command-line tools. Currently, the Azure Sphere SDK supports application development only in C. The SDK is installed in C:\Program Files(x86)\Microsoft Azure Sphere SDK.

The [Quickstarts](#) walk you through building and deploying your first application.

The following sections briefly describe the libraries, templates, and tools.

Azure Sphere Application Runtime

The Azure Sphere Application Runtime includes several libraries for application development:

- A standard C library
- Azure Sphere-specific application libraries (applibs)
- Curl, a multi-protocol transfer library
- An Azure IoT library

C Library

The SDK includes a standard C library that is customized to provide extra security. It does not support file I/O, interprocess communication (IPC), or shell access. Applications built with the Azure Sphere SDK Preview for Visual Studio compile and link against this version of the library and use it for Intellisense.

The header files are installed in the Sysroots\API set\usr\include folders of the Azure Sphere SDK installation directory.

Azure Sphere application libraries

The following table lists the custom application libraries, which support device-specific APIs.

LIBRARY	DESCRIPTION
GPIO	Enables access to GPIOs. Applications that use GPIOs must list them in the Capabilities section of the application manifest .
Log	Writes debugging and other messages to a stream that an attached VS debugging session can read. Applications do not have stdout or stdin.

LIBRARY	DESCRIPTION
Networking	Determines whether network connectivity is available. Applications that connect to network hosts must list those hosts in the Capabilities section of the application manifest .
RTC	BETA feature Interacts with the real-time clock (RTC). These functions are only permitted if the application has the SystemTime capability in the Capabilities section of the application manifest .
Storage	BETA feature Supports the use of on-device storage.
UART	Reads and writes data using the UARTs. Applications that use UARTs must list them in the Capabilities section of the application manifest .
WiFiConfig	Adds, removes, and scans for Wi-Fi networks. Applications that use the WiFiConfig library must enable WifiConfig in the Capabilities section of the application manifest .

The header files are installed in the `Sysroots\API set\usr\include\applibs` folder of the Azure Sphere SDK installation directory.

The Azure Sphere project templates show how to use these libraries. See [Azure Sphere application libraries](#) for details about the functions in each API.

Curl library

In addition to the custom application libraries, the SDK includes a curl library through which applications can transfer data over HTTP/HTTPS. The header files are installed in the `Sysroots\API set\usr\include\curl` folders of the Azure Sphere SDK installation directory. [Connect to web services using curl](#) describes how to use curl in an Azure Sphere application.

Azure IoT library

The Azure Sphere Application Runtime also provides a subset of the Azure IoT client library for use in Azure Sphere applications. The Azure IoT header files are installed in the `Sysroots\API set\usr\include\azureiot` folders of the Azure Sphere SDK installation directory.

Templates

The Azure Sphere SDK includes several sample applications and templates for use in creating your own applications:

- Blink Sample blinks an LED on the board, using the GPIO API.
- Azure IoT Hub Sample communicates with an Azure IoT Hub to send and receive messages, update a device twin, and accept direct method calls. It flashes the LEDs on the board to signal its activities.
- Blank application is a skeleton application that uses the Log API to display messages and shows how to properly terminate an application.
- UART Sample shows how to send and receive data over serial UART. It uses the UART API.
- Blank static library template is the basis for creating a library of functions that use the Azure Sphere SDK. The template sets Visual Studio project properties that are tailored to Azure Sphere projects.

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

2/14/2019 • 5 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, sideloading the application fails.

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 templates automatically create a default application manifest when you create an application based on a template. You must edit the default manifest to list the capabilities that your application requires. When the 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).

The application manifest includes the following items:

NAME	DESCRIPTION
SchemaVersion	Version of the application manifest schema in use. Currently must be 1. Required.
Name	Name of the component. At project creation, this value is set to the name of the project. If you do not use Visual Studio, see How to manually build and load an application for information on adding a name. Required.
ComponentId	ID of the component. Visual Studio creates this ID when you build the application. If you do not use Visual Studio, see How to manually build and load an application for information on creating the ID. Required.
EntryPoint	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.
CmdArgs	Arguments to pass to the application at startup. Enclose each argument in double quotation marks and separate arguments with a comma. Optional.
TargetApplicationRuntimeVersion	Version of the Azure Sphere application runtime that the application requires. This field is automatically added during the build process. Optional. See Use beta features for details.

NAME	DESCRIPTION
TargetBetaApis	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 Use beta features for details.
ApplicationType	Type of application. Optional. Set to Debugger only if you are building a replacement for gdbserver.
Capabilities	List of key/value pairs that specify application resource requirements. Required.

The **Capabilities** section supports the following:

NAME	DESCRIPTION
AllowedConnections	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 wild cards in names and IP addresses are not accepted.
AllowedTcpServerPorts	BETA feature 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.
AllowedUdpServerPorts	BETA feature 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.
DeviceAuthentication	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.
Gpio	List of integers that identify which GPIOs the application uses.
I2cMaster	BETA feature List of I2C master interfaces that are used by the application. For the MT3620 development board, the possible values are "ISU0" through "ISU4".
MutableStorage	BETA feature Mutable storage settings that allow the application to use persistent storage.

NAME	DESCRIPTION
NetworkConfig	BETA feature Boolean that indicates whether the application has permission to configure networking. True if the application has the capability; otherwise, False.
SpiMaster	BETA feature List of SPI master interfaces that are used by the application. For the MT3620 development board, the possible values are "ISU0" through "ISU4".
SystemTime	BETA feature Boolean that indicates whether the application has permission to configure the system time. True if the application has the capability; otherwise, False.
Uart	List of UART peripherals that the application uses. For the MT3620 development board, the possible values are "ISU0" through "ISU4".
WifiConfig	Boolean that indicates whether the application has permission to use the WifiConfig API and consequently to change the Wi-Fi configuration. If this capability is omitted or is set to False, attempts to use the WifiConfig API fail.

The **MutableStorage** section supports the following:

NAME	DESCRIPTION
SizeKB	Integer that specifies the size of mutable storage in kilobytes. The maximum value is 64.

The following shows a sample app_manifest.json file:

```
{
  "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": [ 15, 16, 17, 12 ],
    "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 four GPIOs.
- Specifies the use of one UART peripheral.
- **BETA feature** Enables mutable storage with 64 kibibytes of storage space.
- Enables the app to use the WifiConfig API to change the Wi-Fi configuration.
- **BETA feature** Specifies the use of one SPI master interface.
- **BETA feature** Specifies the use of one I2C master interface.

Use Azure IoT with Azure Sphere

2/14/2019 • 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

2/14/2019 • 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**.

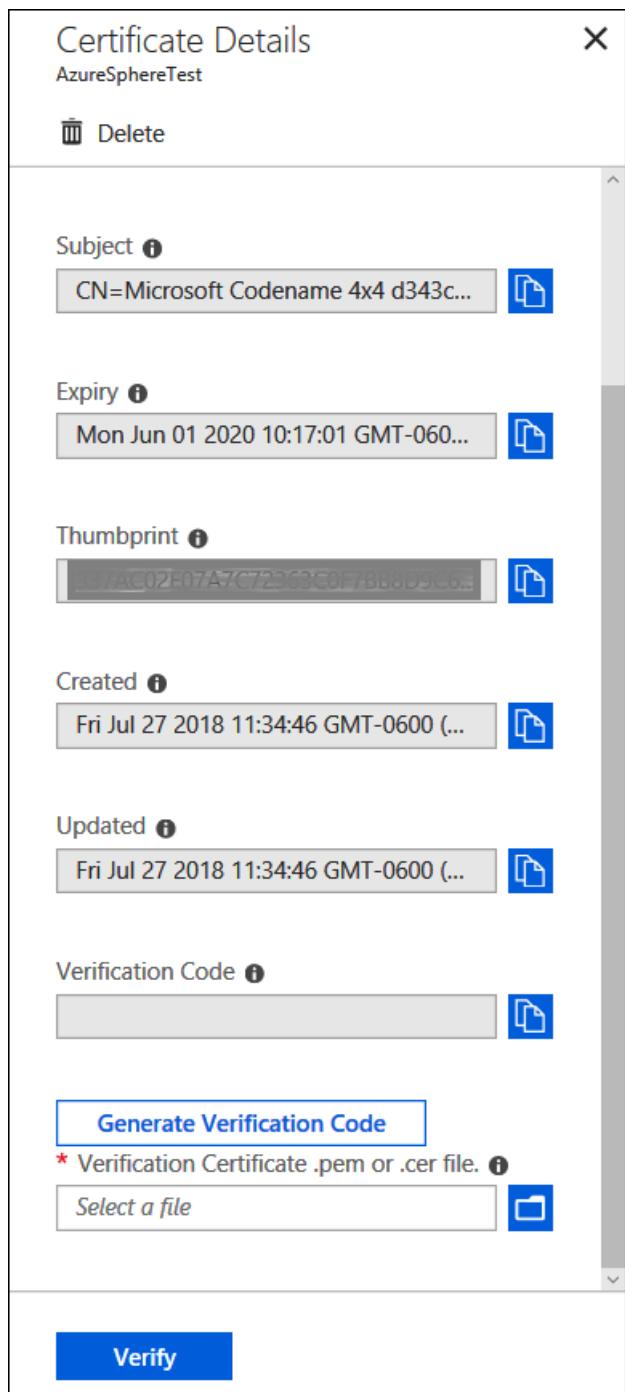
The screenshot shows the AzureSphereDocs-DPS - Certificates interface. On the left, there's a sidebar with options like Access control (IAM), Tags, Properties, Locks, Automation script, Quick Start, Shared access policies, Linked IoT hubs, and Certificates (which is selected). The main area has a search bar and buttons for Add, Columns, and Refresh. A modal window titled 'Add Certificate' is open, containing fields for 'Certificate Name' (set to 'AzureSphereTest') and 'Certificate .pem or .cer file' (set to '"CAcertificate.cer"'). Below these fields is a table with columns NAME, STATUS, and EXPIRY, showing one entry: 'AzureSphere...' with STATUS 'Verified' and EXPIRY 'Thu Jul 23 2020'. At the bottom right of the modal is a 'Save' button.

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 interface after saving the new certificate. The 'Certificates' table now lists two entries. The first entry, 'AzureSphereTena...', has a status of 'Verified'. The second entry, 'AzureSphereTest', has a status of 'Unverified' and is highlighted with a red border. The table includes columns for NAME, STATUS, EXPIRY, SUBJECT, THUMBPRINT, and CREATED.

NAME	STATUS	EXPIRY	SUBJECT	THUMBPRINT	CREATED
AzureSphereTena...	Verified	Thu Jul 23 2020 14:11:20	CN=Microsoft Azure Dev Center 61C85AAC8BAAD9	Mon Jul 23 2018 17:29:40	
AzureSphereTest	Unverified	Mon Jun 01 2020 10:10:10	CN=Microsoft Code 337AC02E07A7C72	Fri Jul 27 2018 11:34:20	

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.



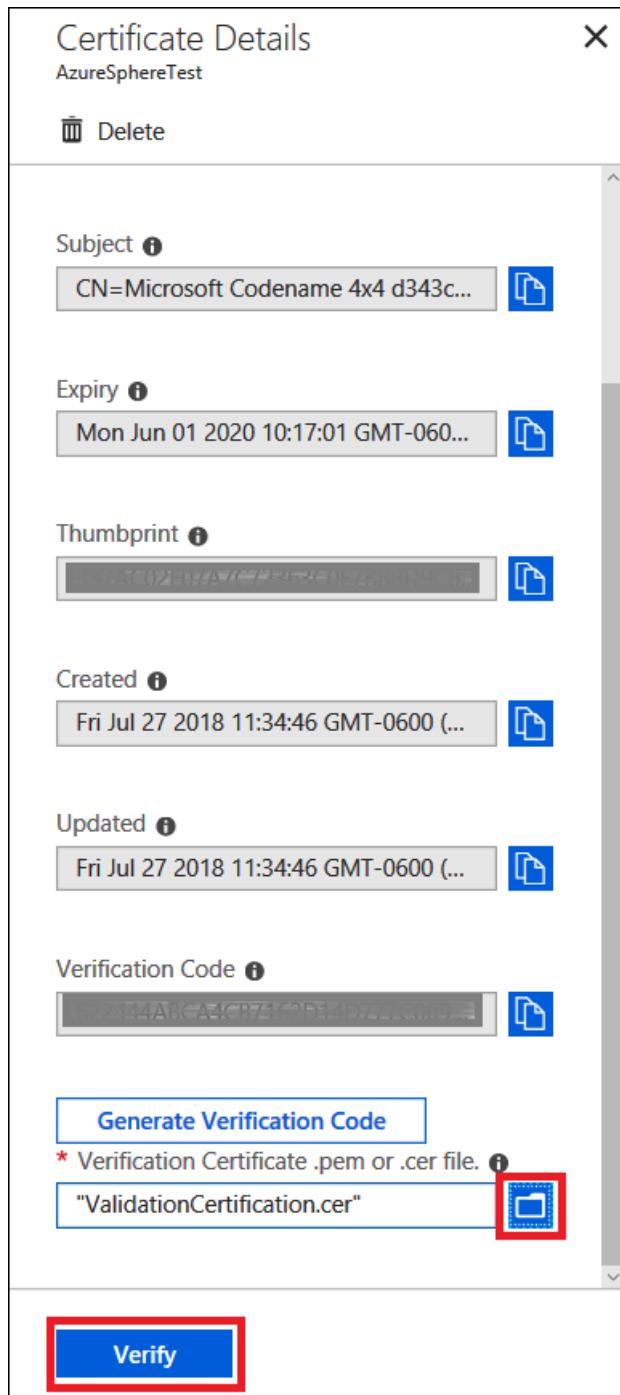
Step 4. Verify the tenant CA certificate

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 Type   

Primary Certificate  
AzureSphereTest

Secondary Certificate  
No certificate selected

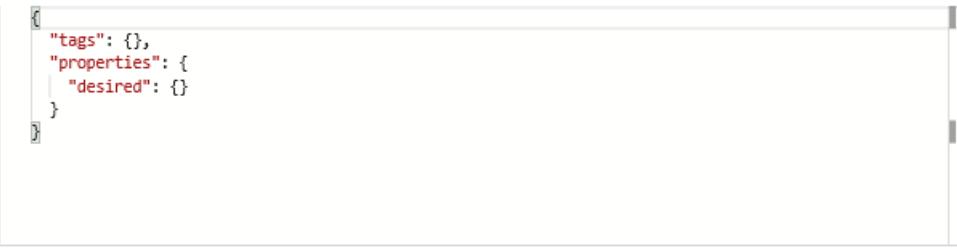
Select how you want to assign devices to hubs  
Evenly weighted distribution

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 Tutorial](#), the [Azure IoT sample](#), or build your own applications that use your IoT hub.

Set up Azure IoT Central to work with Azure Sphere

2/14/2019 • 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
```

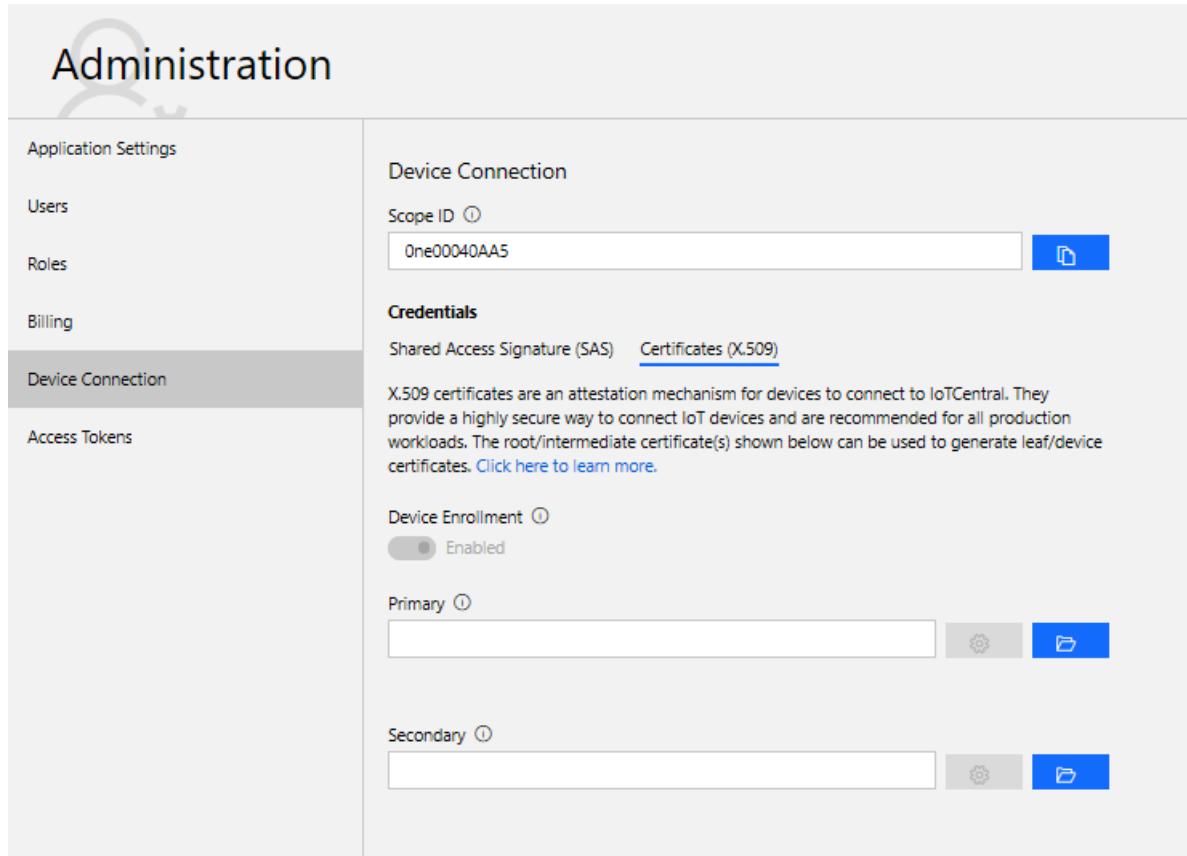
3. Download the Certificate Authority (CA) certificate for your Azure Sphere tenant:

```
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

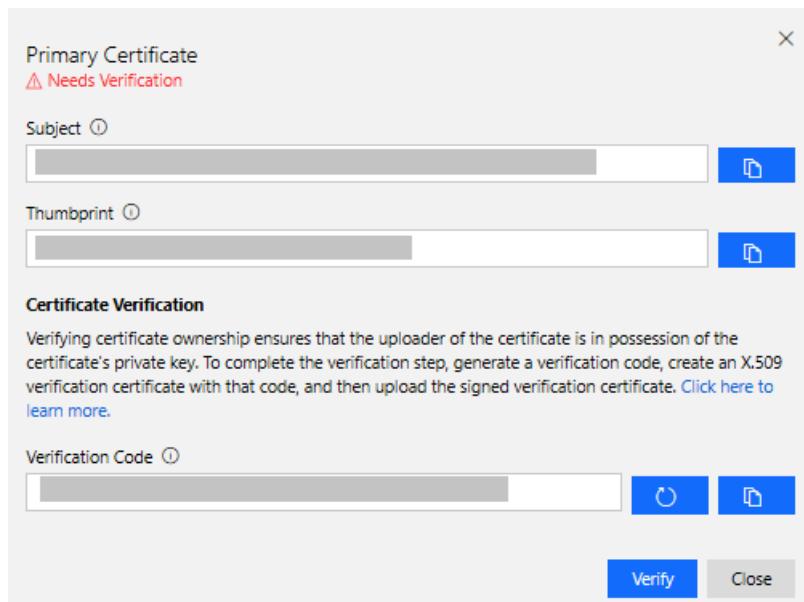
1. In [Azure IoT Central](#), go to **Administration > Device Connection > Certificates (X.509)**:



The screenshot shows the 'Administration' section of the Azure IoT Central portal. On the left, there's a sidebar with links like Application Settings, Users, Roles, Billing, Device Connection (which is 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 blue 'Save' button. Below this is a 'Certificates (X.509)' section with a link to 'Shared Access Signature (SAS)'. It explains that X.509 certificates are an attestation mechanism for devices to connect to IoTCentral. It also mentions that they provide a highly secure way to connect IoT devices and are recommended for all production workloads. There are two sections for 'Primary' and 'Secondary' certificates, each with a text input field, a gear icon for settings, and a folder icon for upload.

2. Click the folder icon next to the Primary box and navigate to the certificate you downloaded in Step 1. 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 a modal dialog box titled 'Primary Certificate'. It has a red warning message '⚠ Needs Verification'. It contains two input fields for 'Subject' and 'Thumbprint', each with a folder icon for upload. Below these is a 'Certificate Verification' section with a note about verifying certificate ownership. It also has a 'Verification Code' input field with a refresh icon and a folder icon. At the bottom are 'Verify' and 'Close' buttons.

Step 4. Verify the tenant CA certificate

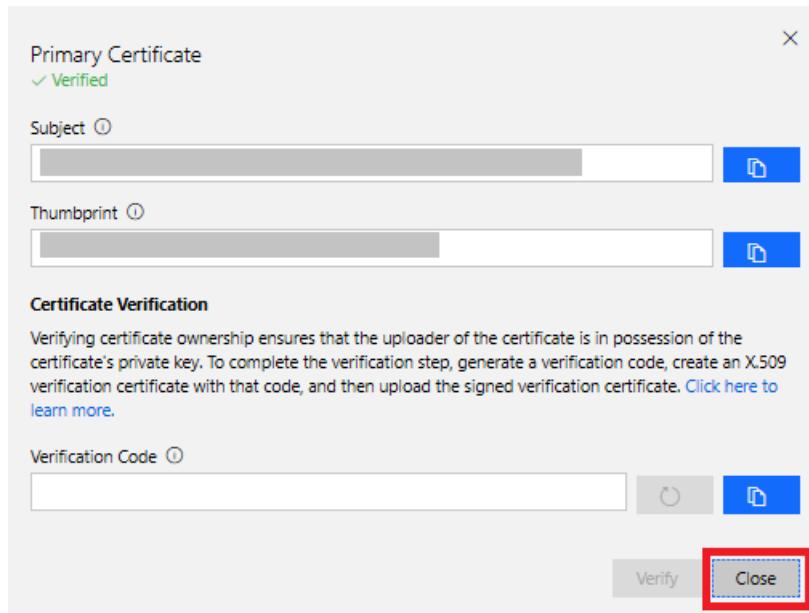
1. Return to the Azure Sphere Developer Command Prompt.
2. Download a validation certificate that proves that you own the tenant CA certificate. Replace *code* in the command with the verification code from the previous step.

```
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.

Azure IoT Hub sample application

2/14/2019 • 8 minutes to read

The Azure IoT Hub sample application shows how to connect to and communicate with an [Azure IoT Hub](#). It also demonstrates a feature of the Azure Sphere WiFiConfig API.

This sample uses the Connected Service for Azure Sphere, which is installed with the Azure Sphere SDK. This Visual Studio extension provides a template for connecting your device to an IoT Hub and communicating with the hub. It is similar to the Visual Studio Connected Service for Azure IoT Hub but provides Azure Sphere-specific features. Applications that use the Connected Service functionality can send messages to and receive messages from an IoT Hub, maintain a device twin, and respond to direct method calls from cloud service applications.

This sample does the following:

- Displays the currently connected Wi-Fi network.
- Blinks LED 1 constantly. Pressing button A changes the rate between three values. The rate is also stored in the device twin, where a cloud service program can change it.
- Sends a message to the IoT Hub when you press button B.
- Lights LED 3 green after start-up to indicate that the device has connected to the IoT Hub and the application has successfully authenticated with the hub.
- Changes the color of LED 1 in response to a direct method call.

Prerequisites

You must complete these steps before you run this sample:

- Connect your Azure Sphere device to your PC and to a [Wi-Fi network](#).
- Run **azsphere device prep-debug** to add the **appdevelopment** capability to your device.

Set up Microsoft Azure credentials

To run this sample, you must have a Microsoft Azure subscription. If your organization does not already have one, follow [these instructions](#) to set up a free trial subscription to Microsoft Azure.

After you set up the subscription, you can create a hub. Log into the [Azure Portal](#) and follow [these instructions](#) to set up your hub and DPS and to link them together. If you already have a hub that is linked to DPS, you can skip step 1 in those instructions but you must still perform [steps 2 through 5](#) to enable authentication and automatic enrollment.

Add your device to the IoT Hub

To add your Azure Sphere device to an IoT hub, you use the IoT Hub Device Provisioning Service. The Connected Service helps you do this by adding code to your Azure Sphere application:

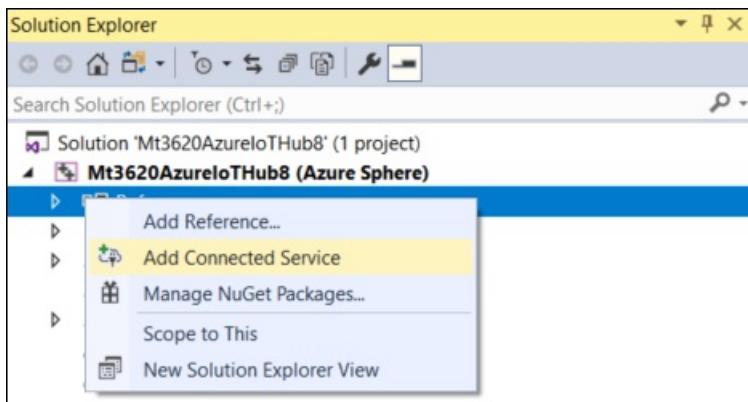
- Adds `azure_iot_utilites.c` and `azure_iot_utilities.h` to your application. The files contain sample code that demonstrates how to use the Azure IoT SDK. You can adapt it to fit your needs. It includes an important Azure Sphere SDK function call to `IoTHubDeviceClient_LL_CreateWithAzureSphereDeviceAuthProvisioning`. This call configures the IoT Hub SDK to connect to the Device Provisioning Service and your IoT Hub using the device certificate issued

during Azure Sphere device authentication and attestation.

- Gets the hostnames of the IoT Hubs that are linked to your Device Provisioning Service instance and adds them to the **AllowedConnections** field in app_manifest.json. This ensures that the Azure Sphere OS firewall allows outbound connections from the application to these URLs.
- Adds the global Device Provisioning Service hostname ("global.azure-devices-provisioning.net") to the **AllowedConnections** field in app_manifest.json. This ensures that the Azure Sphere OS firewall allows outbound connections from the application to this URL.
- Adds the scope ID of the Device Provisioning Service to the IoTHubDeviceClient_LL_CreateWithAzureSphereDeviceAuthProvisioning function call mentioned above. The scope ID uniquely identifies your Device Provisioning Service instance.
- Adds the Azure Sphere tenant ID to the Capabilities/DeviceAuthentication field in app_manifest.json.

To add your device to the IoT hub

1. Start Visual Studio 2017 and go to **File>New>Project**. The templates for Azure Sphere are available in **Visual C++>Cross Platform>Azure Sphere**. Select **Azure IoT Hub Sample for MT3620 RDB (Azure Sphere)**, specify a name and location, and select **OK**.
2. In Solution Explorer, right-click **References** and then select **Add Connected Service**.



3. Select **Device Connectivity with Azure IoT** from the list of connected services.



4. If prompted, log in to the account you're using with your Azure Sphere tenant.
5. In the Device Connectivity with Azure IoT menu, select your Azure subscription from the **Subscription** dropdown.



Subscription: Example Subscription

Connection type: Device Provisioning Service

Device provisioning service: Example Device Provisioning Service Instance

Browse to your subscription on the Azure Portal

Device Provisioning Service

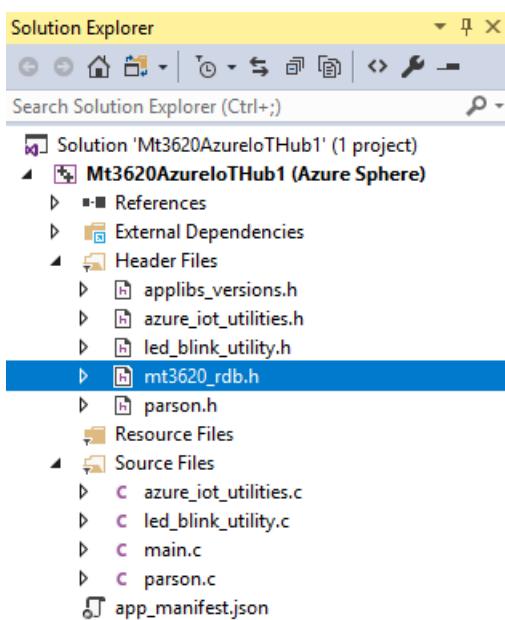
Adding a connection to Azure IoT using the Device Provisioning Service (DPS) will add the Azure IoT SDK to your project and some helper code to get started. The scope from your selected DPS instance will be used.

[See the quickstart for using the Device Provisioning Service](#)

[Review pricing](#)

Add

6. Select Device Provisioning Service from the **Connection Type** dropdown.
7. Select your Device Provisioning Service instance from the **Device Provisioning Service** dropdown.
8. Click Add.
9. In Solution Explorer, you should now see `azure_iot_utilities.h` and `azure_iot_utilities.c` in your solution.



In addition, the host name for the Azure IoT Hub has been added to the **Capabilities** section of the `app_manifest.json` file. An application can connect only to the internet hosts that are specified in the **AllowedConnections** field.

Prepare the sample

Now you can build the application and use the Azure IoT Hub. In this walkthrough, we will use existing IoT Hub tools to monitor and communicate with your device.

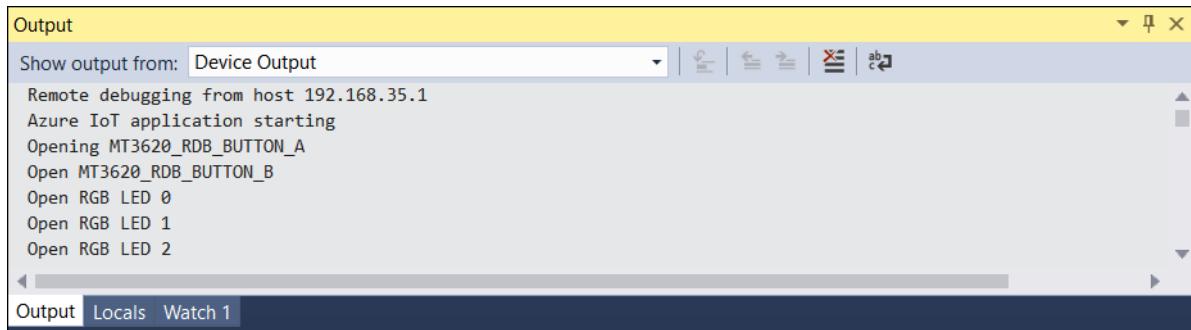
To prepare the sample code

1. Open `main.c` in the sample application.
2. Press F5 to build, load, and start the sample.

The sample opens handles to the buttons and RGB LEDs on the board, and displays the currently

connected Wi-Fi network.

It then starts the main loop. You should see LED 1 start to blink and see output like the following in the Device Output window:



Set up Device Explorer

The following sections describe how to see device-to-cloud messages, send cloud-to-device messages, request changes to the device twin, and make direct method calls. This walkthrough uses Device Explorer, a utility that is the easiest way to interact with your device from an IoT Hub. [Azure Portal](#) and [IoT Hub Explorer](#) provide many of the same capabilities.

NOTE

The direct method calls made from the Azure Portal do not work with this Azure Sphere sample because they use escaped strings, which aren't compatible with this sample.

To install Device Explorer

- On the [Azure IoT Hub SDK for C# Releases](#) repository, download and install [SetupDeviceExplorer.msi](#). If you installed an earlier version of Device Explorer, uninstall it through **Add and Remove Programs** on **Control Panel** before installing this version.

NOTE

This tool is called Device Explorer Twin in its window banner. Some versions of the tool may return an epoch error at random times during use. In most cases, you can close the error notification pop-up and continue using the tool. If the tool fails to operate properly after the error, quit the tool and restart it.

Send and receive messages

Device Explorer is an interactive application that lets you see messages from your device to the cloud and send messages from the cloud to your device.

To get your IoT Hub Connection String

Before you can configure Device Explorer, you need to get your IoT Hub Connection String:

- Go to the [Azure Portal](#).
- Click on your IoT Hub.
- Under **Settings**, click **Shared access policies**.
- Click **iothubowner**.
- Copy the text under **Connection string—primary key**.

To send and receive messages

1. Run Device Explorer. You can find it on the **Start** menu under **Azure IoT Hub**.
2. On the **Configuration** tab, paste the IoT Hub Connection String that you copied from above and click update.
3. On the **Data** tab, select your device from the Device ID dropdown menu and press **Monitor**.
4. Press button B on your Azure Sphere device to send a message to the cloud. If Device Explorer displays an error box describing an endpoint epoch error, restart Device Explorer.
5. In Device Explorer, check the **Data** tab to see the message from your device. In some situations, a delay may occur before the messages appear, and you'll sometimes see a batch of device-to-cloud messages at once.
6. In Device Explorer, open the **Messages to Device** tab and select your device from the Device ID dropdown menu. Type a message in the Message box, add a timestamp if you want, and press **Send**.
7. In Visual Studio, your message should appear in the **Output** window for Device Output.

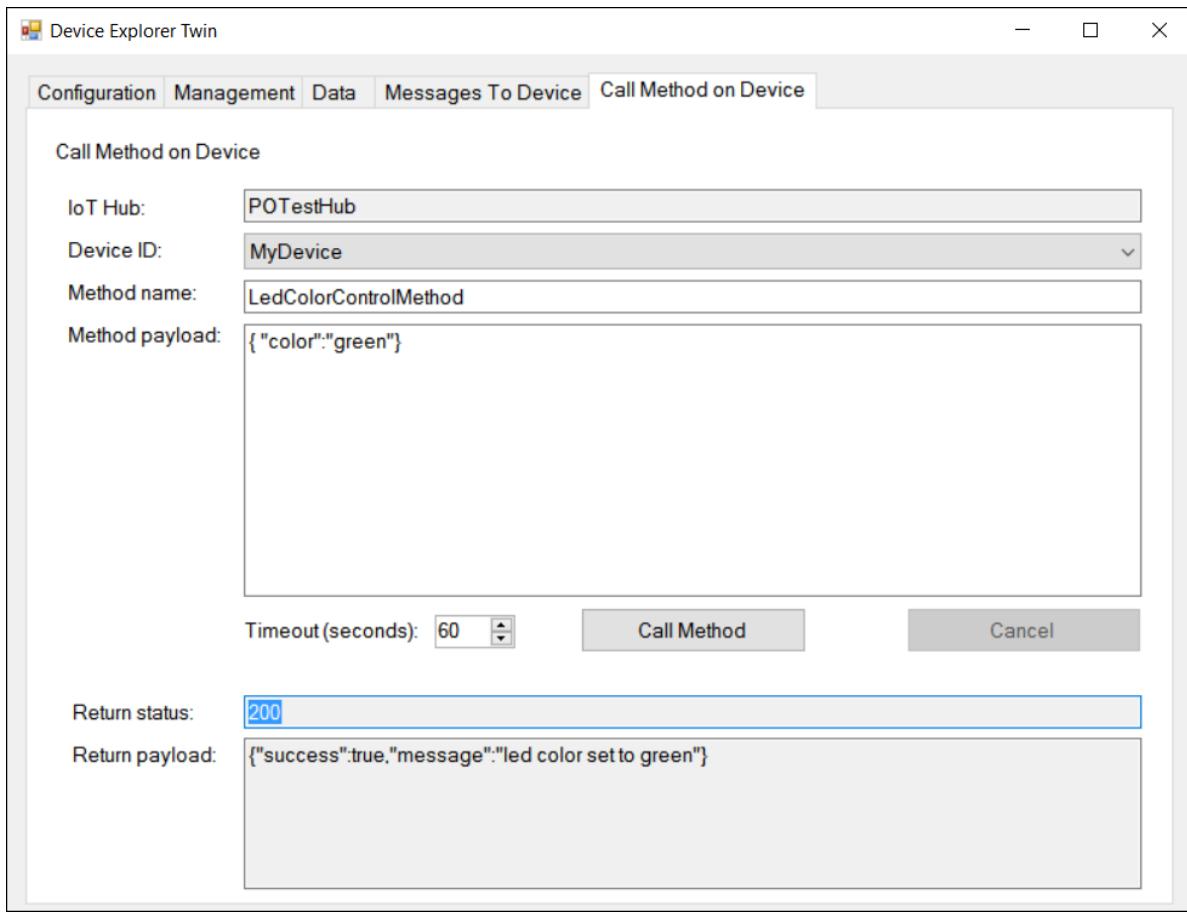
Call a direct method on the device

Applications that use an IoT Hub can respond to [direct method calls](#) from applications that run in the cloud. For example, a cloud-service application might collect data from the IoT Hub and then call the application on the device to reset a counter.

The example registers the DirectMethodCall() function to handle direct method calls. When a cloud service application calls a method on the device, this callback parses the method name and payload and responds appropriately.

To call a method on the device

1. Make sure that the application is running, and is not stopped at a breakpoint.
2. In Device Explorer, open the **Call Method on Device** tab. Ensure that the IoT Hub and Device ID are correct.
3. In the **Method name** box, type LedColorControlMethod, and in the Method payload box type `{"color": "green"}`. Then press **Call Method**. The sample is hardcoded to check for calls to LedColorControlMethod. When you call LedColorControlMethod with this payload, LED 1 should start blinking green.



4. In Visual Studio, check the Output window for Device Output. You should see:

```
[Azure IoT Hub client]:INFO: Trying to invoke method LedColorControlMethod  
INFO: color set to: 'green'.
```

5. In Device Explorer, check the Return status and Return payload boxes. The Return status should be 200 and the return payload should be:

```
{"success":true,"message":"led color set to green"}
```

Manage a device twin

A [device twin](#) is a JSON document in the cloud that stores information about a device. Use the device twin to synchronize status information and maintain device properties that should be accessible to cloud service applications as well as to applications that run on the device.

To view and change the device twin

1. While the sample program runs, open the **Management** tab in Device Explorer and then click **Twin Props**. Select your device from the drop-down menu in the bar and click **Refresh**. You should see the **Entire Twin** tab open on the left and an editable window on the right. Note that the twin for your device may have different properties and metadata than the one in the figure.

```

{
  "deviceId": "MyDevice",
  "etag": "AAAAAAA8=",
  "properties": {
    "desired": {
      "LedBlinkRateProperty": 1,
      "$metadata": {
        "$lastUpdated": "2018-06-29T17:00:00.1759513Z",
        "$lastUpdatedVersion": 15,
        "LedBlinkRateProperty": {
          "$lastUpdated": "2018-06-29T17:00:00.1759513Z",
          "$lastUpdatedVersion": 15
        }
      },
      "$version": 15
    },
    "reported": {
      "LedBlinkRateProperty": 1,
      "$metadata": {
        "$lastUpdated": "2018-06-29T17:00:00.2815462Z",
        "LedBlinkRateProperty": {
          "$lastUpdated": "2018-06-29T17:00:00.2815462Z"
        }
      },
      "$version": 113
    }
  }
}

```

2. When you press button A on the board, the sample changes the blink rate and reports the new rate to the device twin as `LedBlinkRateProperty` in the "reported" section. To see the new rate in Device Explorer, click **Refresh**. Note that the "reported" version number in the twin also changes.
3. You can also change the blink rate by setting the value of `LedBlinkRateProperty` in the device twin. Edit the JSON in the window on the right and then press **Send (use JSON format)**. You can set the value to 0, 1, or 2 for different blink rates.

```

{
  "properties": {
    "desired": {"LedBlinkRateProperty":0}
  }
}

```

When you change the value of `LedBlinkRateProperty`, the sample updates the LED 1 blink rate and reports the new value to the twin in the "reported" section. You should see the LED blink at a different rate.

4. In Visual Studio, check the Output window for Device Output. You should see:

```

Property LedBlinkRateProperty changed, new value is "0"
[Azure IoT Hub client]:INFO: Reported state set
[Azure IoT Hub client]:INFO: Reported state accepted by IoT Hub. Result is: 204

```

UART sample application

2/14/2019 • 2 minutes to read

The UART sample demonstrates the use of communication over UART. A simple way to test UART is to loop back a UART on the board. On header 2 (marked H2) on the lower left side of the board:

- Connect pins 1 and 3 (ISU0 RXD and ISU0 TXD) of H2 with a jumper header. These are the first two pins on the left side of the header, circled in red in the figure.



To run the UART Sample

1. Start Visual Studio 2017 and go to **File>New>Project**. The templates for Azure Sphere are available in **Installed>Visual C++>Cross Platform>Azure Sphere**. Select **UART Sample for MT3620 RDB (Azure Sphere)**, specify a name and location, and select **OK**.
2. Ensure that you have installed the jumper header in the correct location for your device. Connect the device to your PC by USB, then press **F5** or select **Remote GDB Debugger** on the menu bar. If you are prompted to build the project, select **Yes**.
3. Press button A on the board. This sends 13 bytes over the UART connection and displays the sent and received text in the Visual Studio Device Output window:

```
Sent 13 bytes over UART in 1 calls  
UART received 12 bytes: 'Hello world!'  
UART received 1 bytes: '  
'
```

All the received text might not appear at once, and it might not appear immediately.

LED 2 on the development board toggles on and off each time you press the button.

Beta API features

2/14/2019 • 2 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. Applications that use Beta APIs will generally require modifications after future Azure OS and SDK releases to continue to work correctly. 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 APIs, enabling prototyping and feedback before they are finalized.

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

Target API sets and sysroots

The target API set indicates which APIs the application uses: either production APIs only or production and Beta APIs. You set and edit this value with the Project Properties in Visual Studio. The Target API Set value is either an integer that represents the application runtime version or an integer plus a string that identifies the Beta API release. For example, the value "1" specifies only the production APIs in the current release, whereas "1+beta1811" specifies the production and Beta APIs at the 18.11 release.

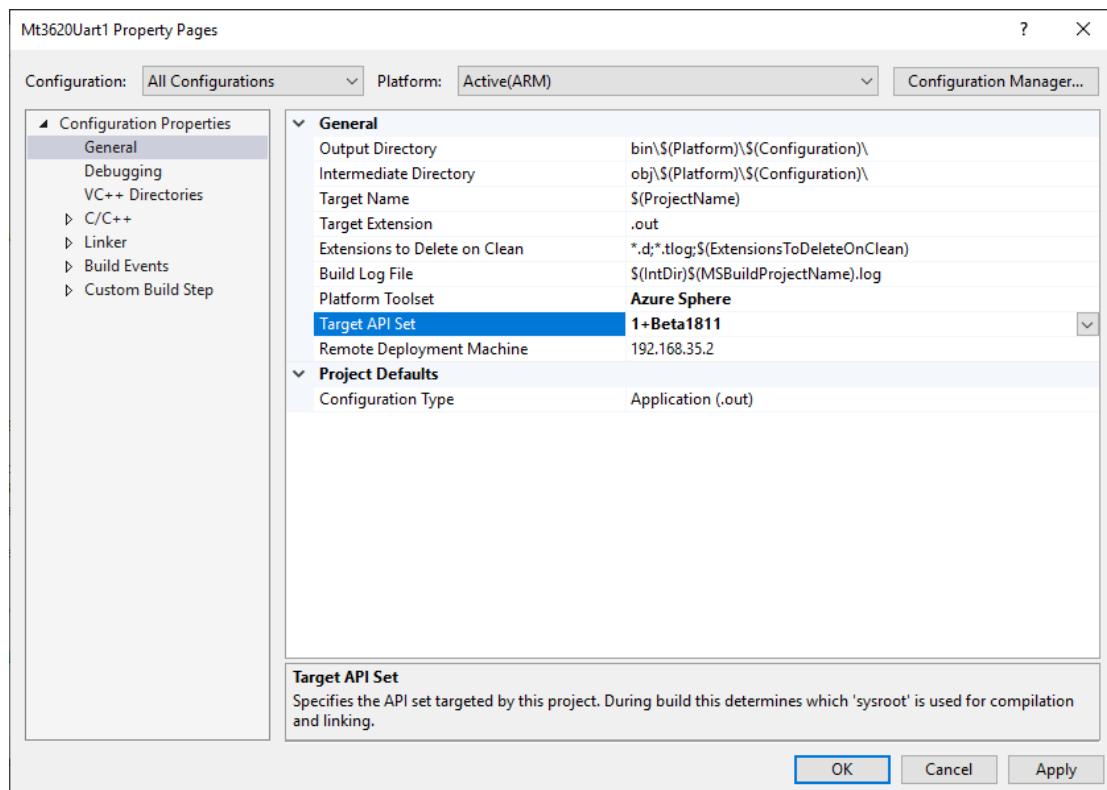
The Azure Sphere SDK implements multiple API sets by using *sysroots*. A sysroot specifies the libraries, header files, and tools that are used to compile and link an application that targets a particular set of APIs. The sysroots are installed in the Microsoft Azure Sphere SDK directory in the sysroots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

Develop applications with Beta APIs

If you start with an Azure Sphere sample that uses a Beta API, the Project Properties set the Target API Set to specify 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 set the project properties to use production and Beta APIs:

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 `1+Beta<release-date>` and then click OK.



Sideload and debug an application image package

11/15/2018 • 2 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>
```

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.

For example:

```
azsphere device prep-debug

Getting device capability configuration for application development.
Downloading device capability configuration for device ID
'ABCDE082513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A5588B420851
EE4F3F1A7DC51399ED'.
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
'ABCDE082513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A5588B420851
EE4F3F1A7DC51399ED'.
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
'ABCDE082513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A5588B420851
EE4F3F1A7DC51399ED' 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 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 existing applications, sideload the new image package, and start the application:

```
azsphere device sideload delete
azsphere device sideload deploy -p <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 -m -p imagepackagepath
azsphere device sideload start -d -i <ComponentID>
```

The command displays the output and debug ports:

```
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 -i option, as follows:

```
azsphere device sideload stop -i <ComponentID>
azsphere device sideload delete -i <ComponentID>
```

To stop and delete all applications on the device, omit the -i option.

Package, sideload, and debug a manually-built application

If you did not build the application with Visual Studio, follow the steps in [How to manually build and load an application](#).

Remove an application

8/13/2018 • 2 minutes to read

You can remove the application that is currently running 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**.
- From an Azure Sphere Developer Command Prompt, issue the following command to delete all apps from the device:

```
azsphere device sideload delete
```

After you delete the application, the device will not run any application until you sideload an application or trigger deployment. See [When do updates occur](#) for details.

Determine application memory usage

2/14/2019 • 2 minutes to read

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: 0000000200001000
SigCgt: 0000000000004000
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
```

How to manually build and load an application

11/15/2018 • 6 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 Blink sample as an example. By default, the source and header files required to build the Blink sample are in the Mt3620Blink 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 Blink 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 Blink sample, compile the main.c and epoll_timerfd_utilities.c source files. This sample uses only production APIs, so the commands specify sysroot "1".

```
gcc.exe -c -x c main.c -I "C:\Program Files (x86)\Microsoft Azure Sphere SDK\SysRoots\1\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\1\"
```

```
gcc.exe -c -x c epoll_timerfd_utilities.c -I "C:\Program Files (x86)\Microsoft Azure Sphere SDK\SysRoots\1\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\1\"
```

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
<code>-c</code>	Compiles but doesn't link the program.

OPTION	DESCRIPTION
-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=dir	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 1.

```
gcc.exe -o "Blink.out" --sysroot="C:\Program Files (x86)\Microsoft Azure Sphere SDK\SysRoots\1" -Wl,--no-undefined -nodefaultlibs -B "C:\Program Files (x86)\Microsoft Azure Sphere SDK\SysRoots\1\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
-o <i>filename</i>	Specifies the name of the output file.
--sysroot= <i>dir</i>	Uses <i>dir</i> as the logical root directory for headers and libraries.
-Wl,--no-undefined	Passes --no-undefined as a linker option, so that the linker reports any references that are unresolved after linking.
-nodefaultlibs	Does not use the standard system libraries when linking. Only the libraries you specify are passed to the linker.
-B <i>path</i>	Specifies the path to the gcc compiler.
-march=armv7ve	Specifies the armv7 architecture with virtualization exceptions.
-mcpu=cortex-a7	Specifies the name of the target ARM processor.
-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.
-lapplibs	Searches the applibs library during linking.
-lpthread	Searches the pthread library during linking.
-lgcc_s	Searches the gcc_s library during linking.
-lc	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" : "app",  
    "ComponentId" : "nnnnnnnn-nnnn-nnnn-nnnnnnnnnnnn",  
    "EntryPoint": "/bin/app",  
    "CmdArgs": [],  
    "Capabilities": {  
        "AllowedConnections": [],  
        "Gpio": [ 8, 9, 10, 12 ],  
        "Uart": [],  
        "WifiConfig": false  
    }  
}
```

- Run the **azsphere image package-application** command to package the image:

```
azsphere image package-application --input approot --output manual.imagepackage --sysroot 1 -v
```

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.

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.

- If your device is already running an application, delete the application:

```
azsphere device sideload delete
```

- Run the **azsphere device sideload deploy** command to sideload the application onto the device and start it:

```
azsphere device sideload deploy -p manual.imagepackage
```

You should see LED L1 start to blink.

To debug the application, stop it and then restart it with the -d option:

```
azsphere device sideload stop -i <ComponentId>
```

```
azsphere device sideload start -d -i <ComponentId>
```

You should see:

```
<ComponentId>  
App state  : debugging  
GDB port   : 2345  
Output port : 2342  
  
Command completed successfully in 00:00:00.9121174.
```

- 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**.

Connect to web services

2/14/2019 • 4 minutes to read

The Azure Sphere SDK includes the libcurl library, which applications can use to connect to and authenticate HTTP and HTTPS web services. The [CurlEasyHttps sample](#) uses a synchronous (blocking) API, and the [CurlMultiHttps sample](#) uses an asynchronous (non-blocking) API. The CurlEasyHttps sample demonstrates simple synchronous communication with web services. In general, however, Azure Sphere applications should use the more complex asynchronous technique shown in the CurlMultiHttps 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](#).

Server authentication is supported, so that applications can 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 common CAs are built into the Azure Sphere device. In addition, you can add one or more certificates to your application image package.

NOTE

If your applications require a TLS connection to run non-HTTPS protocols, please contact your Microsoft representative or provide feedback on the [Azure Sphere forum](#).

Requirements for applications that use curl

Applications that use the curl library and authenticate a server must include the appropriate header files, add a curl dependency to the linker, and specify the hosts to which they connect in the [application manifest](#).

Header files

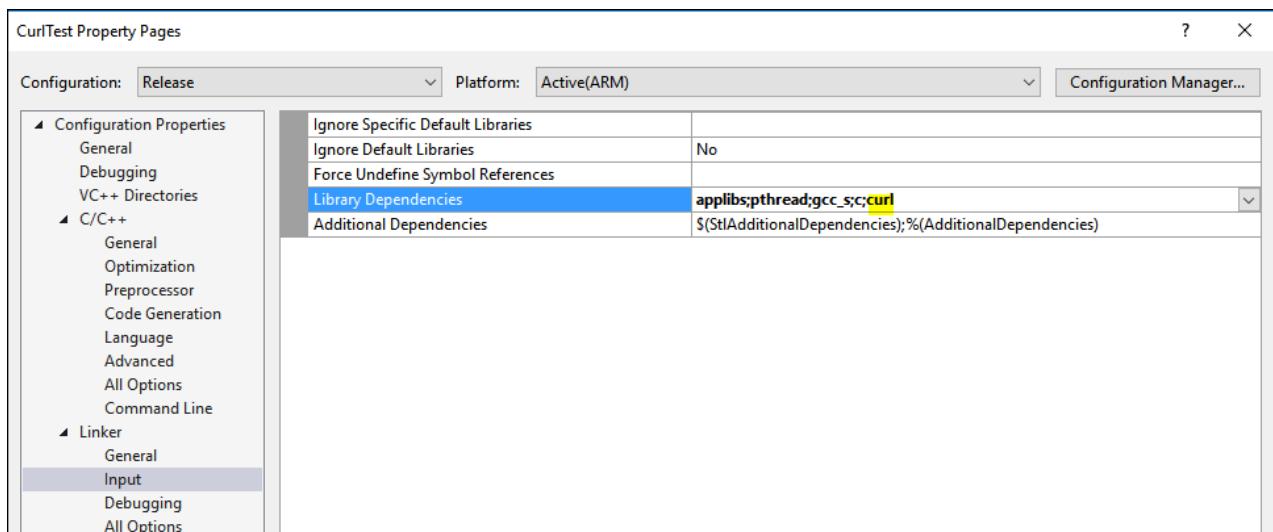
To use curl, include these header files in your application:

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

The storage.h header file is required only if you supply one or more certificates in the application image package. It is not required if you use only default certificates.

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

Every host to which the application tries to connect must be included in the **AllowedConnections** section of the app_manifest.json file. For example, the following specifies that the application connects only to www.example.com:

```
"Capabilities": {
    "AllowedConnections": [ "www.example.com" ],
    "Gpio": [],
    "Uart": [],
    "WifiConfig": false
}
```

The **AllowedConnections** section must also contain the name of each domain that the connection may encounter if redirected. For example, both microsoft.com and www.microsoft.com are required for an application that connects to the Microsoft home page.

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.

CA certificates

The server's certificate must be signed by a CA that the device trusts. 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.

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 file for your application.
2. In the certs folder, create a text file with the extension .pem, 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.

Server authentication

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 as described in the previous section 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);
```

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");
```

Using the real-time clock with Azure Sphere

1/7/2019 • 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 networking and RTC header in your project:

```
#include <applibs\rtc.h>
#include <applibs\networking.h>
```

Application manifest settings

To use the RTC 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
  }
}
```

NTP service

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.

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 currently 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 I2C with Azure Sphere

2/14/2019 • 2 minutes to read

BETA feature

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.

MT3620 support

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

- 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.

I2C Requirements

Applications that use I2C must enable beta APIs, include the appropriate header files, and add I2C 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 <applibs_versions.h>
#include <applibs/i2c.h>
```

Add `#define I2C_STRUCTS_VERSION 1` to `applibs_versions.h`. This specifies the struct version that is used by the application.

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. Here's an example of the I2c capability that is configured to use ISU1 and ISU3 as I2C master interfaces.

```
"I2cMaster": [ "ISU1", "ISU3" ],
```

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()`.

Using SPI with Azure Sphere

2/14/2019 • 3 minutes to read

BETA feature

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.

MT3620 support

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

- 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 I2C 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.

SPI requirements

Applications that use SPI must enable beta APIs, include the appropriate header files, and add SPI 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 <applibs_versions.h>
#include <applibs/spi.h>
```

Add `#define SPI_STRUCTS_VERSION 1` to `applibs_versions.h`. This specifies the struct version that is used by the application.

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. Here's an example of the `SpiMaster` capability that is configured to use ISU1 and ISU3 as SPI master controllers.

```
"SpiMaster": [ "ISU1", "ISU3" ],
```

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()`.

Using storage on Azure Sphere

2/14/2019 • 3 minutes to read

BETA feature

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

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](#)

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

BETA feature

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 enable beta APIs, include the appropriate header files, and add mutable storage 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 storage and unistd headers in your project:

```
#include <applibs/storage.h>
#include <unistd.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 kibibytes. 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](#).

Troubleshooting

1/7/2019 • 3 minutes to read

Here are some simple troubleshooting steps for common problems.

MT3620 board is unresponsive

If the MT3620 board appears "dead" or unresponsive, the problem could be a lack of power. For example, after recovery, **azsphere** reports the following message:

```
Failed to establish communication with device after recovery.
```

Solution

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

Visual Studio or **azsphere** cannot find your device

Visual Studio or the **azsphere** command returns the following message:

```
An error occurred. Please check your device is connected and your PC has been configured correctly, then retry.
```

Solution

This message can appear in several circumstances:

- The board is not connected by USB to the PC.
- The TAP driver is not installed.
- The IP address is not set correctly.
- The Azure Sphere Device Communication Service has not yet started.

Try the following solutions, in order:

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 issue 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 Azure Sphere device exists and is configured to use IP address 192.168.35.1. To access this control panel, press Start and type "View network connections" or "ncpa.cpl" and then select the control panel).

COM Ports

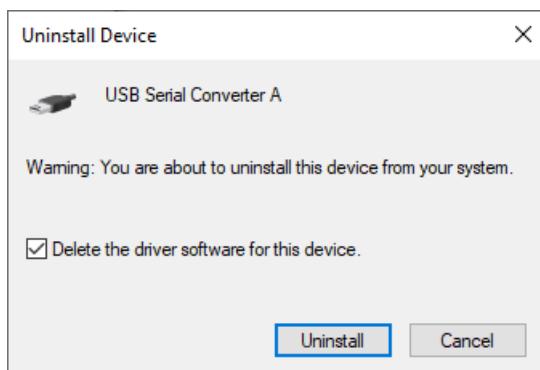
After setting up the MT3620 development board, you do not see three COM ports, but only one or two:



Typically, this situation indicates a problem with the FTDI driver.

Solution

1. Open **Device Manager** by clicking **Start** and typing "Device Manager."
2. 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.

3. 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 available.
4. 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.

USB errors

After you attach the MT3620 development board to your PC, the PC reports the following error:

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

Solution

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.

Debugger reports failure to connect to device

During debugging with Visual Studio, you see errors like the following:

```
Error: Time:Thu Jan  1 01:55:23 1970 File:/usr/src/debug/azure-c-shared-utility/2017-03-24.AUTOINC+6c921761a5-r0/git/c-utility/src/tlsio\_wolfssl.c Func:add\_certificate\_to\_store Line:426 wolfSSL\_CTX\_load\_verify\_buffer failed
```

```
Error: Time:Thu Jan  1 01:55:23 1970 File:/usr/src/debug/azure-c-shared-utility/2017-03-24.AUTOINC+6c921761a5-r0/git/c-utility/src/tlsio\_wolfssl.c Func:create\_wolfssl\_instance Line:480 Failed to add certificates to store
```

```
Error: Time:Thu Jan  1 01:55:23 1970 File:/usr/src/debug/azure-c-shared-utility/2017-03-24.AUTOINC+6c921761a5-r0/git/c-utility/src/tlsio\_wolfssl.c Func:tlsio\_wolfssl\_open Line:695 Cannot create wolfssl instance.
```

```
Error: Time:Thu Jan  1 01:55:23 1970 File:/usr/src/debug/umqtt/2017-03-24.AUTOINC+6c921761a5-r0/git/umqtt/src/mqtt\_client.c Func:mqtt\_client\_connect Line:895 Error: io\_open failed
```

```
Error: Time:Thu Jan  1 01:55:23 1970 File:/usr/src/debug/azure-iot-sdks/2017-03-24.AUTOINC+6c921761a5-r0/git/iothub\_client/src/iothubtransport\_mqtt\_common.c Func:SendMqttConnectMsg Line:doWork called...
```

```
1719 failure connecting to address v2Test.azure-devices.net:0.
```

Solution

Use the [azsphere device wifi show-status](#) command to make sure that your device is connected to Wi-Fi.

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.

Use Microsoft Azure IoT Hub

2/14/2019 • 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 Hub sample application](#) for a walkthrough of Azure IoT Hub and connected service 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. The [Azure IoT Hub sample application](#) includes a sample interface for performing basic operations with an Azure IoT Hub.

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.

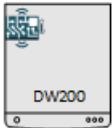
Deployment basics

2/14/2019 • 9 minutes to read

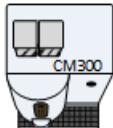
We will use three appliance models from the fictitious Contoso Corporation as an example throughout this discussion.



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.

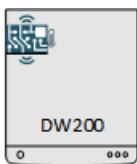
In addition, every Azure Sphere chip has a *chip SKU* that identifies a particular type of Azure Sphere-compatible MCU. The chip SKU is assigned by Microsoft and cannot be changed. Microsoft uses this SKU to deliver the correct system software updates to each Azure Sphere device.

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.

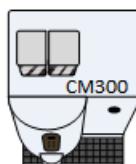
For example:



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.

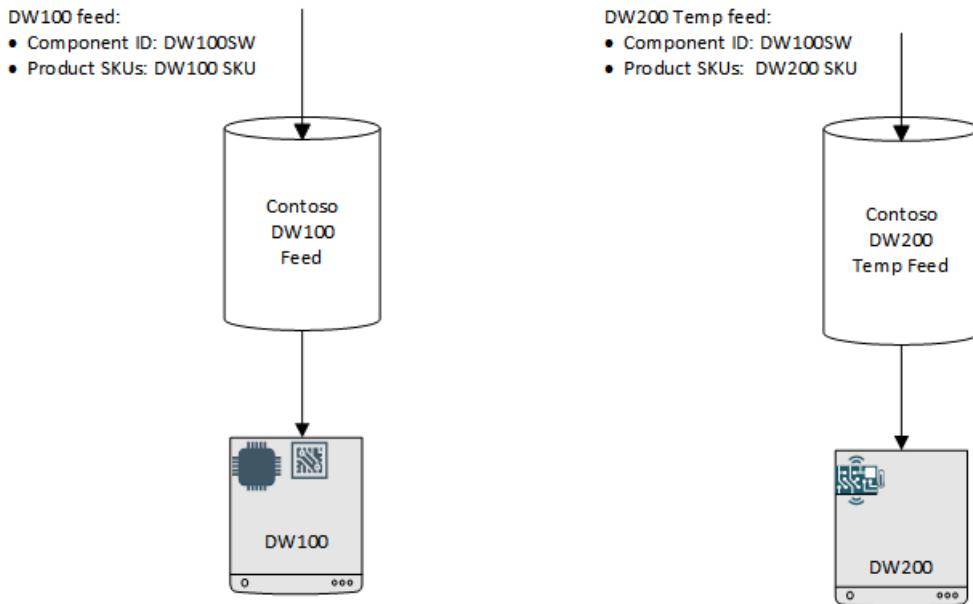
An application feed is analogous to a pipe that carries only a certain kind of material; that is, an application feed can only deliver image sets that represent a particular component. Therefore, Contoso would define one feed for the DW100SW application software and another for the CM300SW application 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:

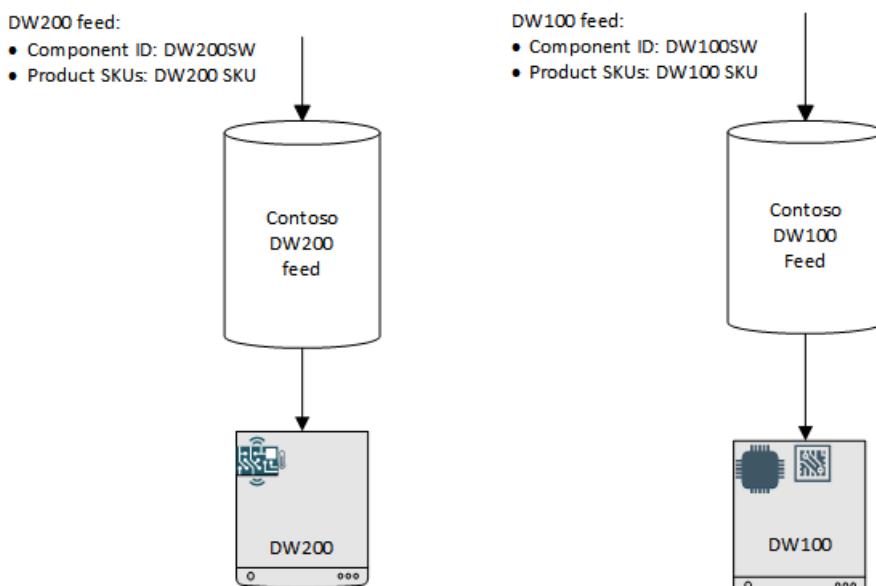
- Define a single feed that delivers the DW100SW application software to both the DW100 and DW200 product SKUs
- Define one feed that delivers the DW100SW application software to the DW100 product SKU and another feed that delivers the DW100SW application software to the DW200 product SKU

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

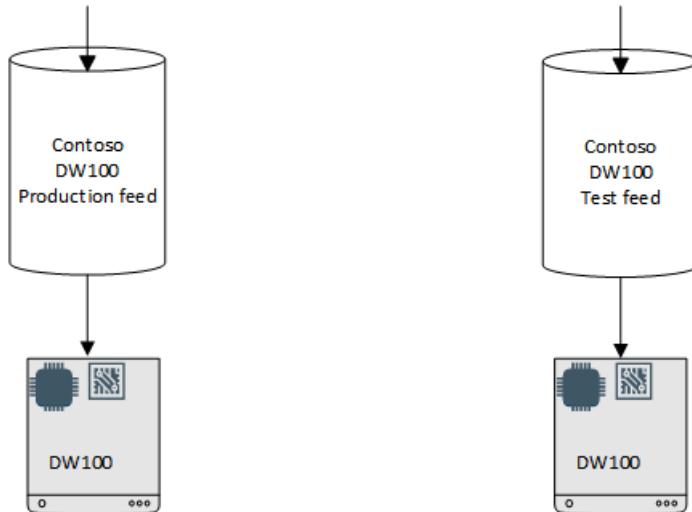
As this example implies, a product manufacturer would typically define multiple feeds. For example, Contoso might define two feeds for each of its applications: a Test feed and a Retail feed. The Test feed delivers application software that is still in development, and the Retail feed delivers application 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 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

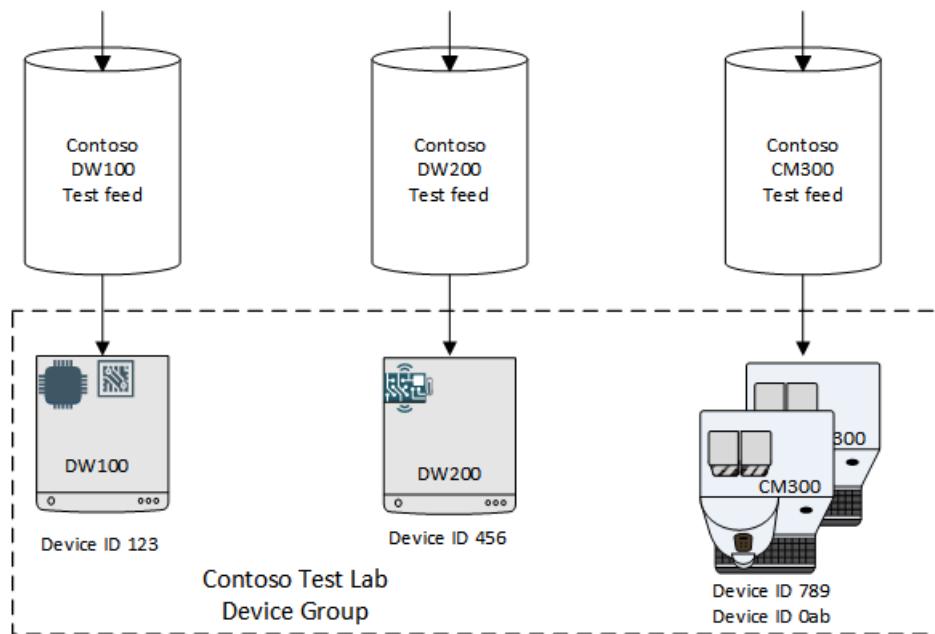
The next time device 456 in the Contoso Test Lab requests an update, it receives the DW200SW test software via the DW200 Test feed.

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?

8/13/2018 • 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

2/14/2019 • 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 retail use. The **Retail Evaluation Azure Sphere OS** feed provides OS software two weeks before its release to the Retail Azure Sphere OS feed, so that customers can test new features before broad deployment.

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 Azure Sphere OS feed.

NOTE

Currently, the Retail Evaluation Azure Sphere OS feed provides the same Azure Sphere OS as the Retail Azure Sphere feed, so that you can begin preparing for this additional feed. The feed will begin delivering evaluation software approximately two weeks before our next release, and we will [notify you](#) through the usual channels.

Deploy an application over the air

8/13/2018 • 2 minutes to read

Over-the-air (OTA) application deployment involves a series of tasks:

- Removing the current sideloaded application from the device
- Removing the **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

Although the **azsphere** command-line utility can perform each of these tasks individually, in most situations you can deploy an application in two steps by using **azsphere** commands that combine several commands:

- [Prepare the device for OTA updates](#)
- [Link the device to a 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.

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

11/15/2018 • 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 Wi-Fi credentials and reconnect the device to Wi-Fi by using the **azsphere device wifi** commands.

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 board 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 board for development, run **azsphere device prep-debug** to re-enable application sideload and debugging.
5. [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

2/14/2019 • 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](#)
- [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

12/12/2018 • 4 minutes to read

Manages components and images for the Azure Sphere Security Service.

OPERATION	DESCRIPTION
create	Creates a new component
image, img	Manages images for a component
list	Lists all components
publish	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
<code>-i, --componentid GUID</code>	Specifies the ID of the component. This value appears in the ComponentId field of the app_manifest.json file for the application.
<code>-n, --name string</code>	Specifies a name for the component. Component names must be unique within an Azure Sphere tenant.

Optional parameters

PARAMETER	DESCRIPTION
<code>-t, --imagetype ComponentImageType</code>	The image type for this component. If not provided, defaults to the Application image type. Values can be either Application or BoardConfiguration .

► 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
add	Uploads a new image and adds it to a component
download	Downloads an existing image
show	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.

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.
-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 -o "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 and IDs of all components in the Azure Sphere tenant.

► Global parameters

Example

```
azsphere component list

Listing all components.
Retrieved components:
--> [afb39a19-4198-4b87-a5c6-46fa40e87cf3] 'MyIoTHubApp':
--> [4275ecb3-5cf8-4147-9bfb-a7e8f3955e96] 'Mt3620BlinkEx':
--> [617b716c-714c-4008-a56e-bd654203273b] 'TestApp':
--> [34abd599-89ea-49af-bd80-21f6f63c5f8f] 'UartDoc':
--> [37c8baa4-33db-419d-b9b3-94e42964a3e6] 'Mt3620AzureIoTHub7':
--> [03ee58ce-d091-43a8-a2ac-0bb2920285db] 'Mt3620AzureIoTHub1':
--> [07a938d8-484f-4f3d-a0fa-e4f463acf1ff] 'flipdot_iot_central':
--> [36f987ed-dc26-42b1-bd1b-14a889fb83f4] 'blink test app':
--> [16995a70-377f-4bd2-b29d-1b0fffcbef287] 'Mt3620Blink3':
--> [8fb105a3-35d0-423a-9427-a21852623965] 'Mt3620Blink5':
--> [88d7fe6a-165a-443c-86f0-798531561f44] 'Mt3620Blink6':
--> [350ef47d-fa94-47fe-95bc-e2d393b589ab] 'Mt3620Blink1':
Command completed successfully in 00:00:04.7822167.
```

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
-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.
-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

2/14/2019 • 23 minutes to read

Manages Azure Sphere devices.

OPERATION	DESCRIPTION
capability, cap	Manages device capabilities
claim	Claims a previously unclaimed device to the Azure Sphere tenant
image, img	Manages images for a device
link-feed	Links a device to a feed
manufacturing-state, mfg	Manages the manufacturing state of the attached device
prep-debug	Sets up a device for local debugging
prep-field	Sets up a device to disable debugging and receive over-the-air (OTA) updates
recover	Uses special recovery mode to load new firmware onto the device
restart	Restarts the attached device
show-attached	Displays details about the attached device from the device itself
show-ota-config	Displays details about the OTA update configuration of the device
show-ota-status	Displays the status of the most recent OTA update
sideload, sl	Loads an application onto the attached device or changes the status of the application
update-device-group	Moves a device into a device group
update-sku	Sets the product SKU for a device
wifi	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
download	Downloads a device capability configuration from the Azure Sphere Security Service
show-attached	Displays the capability configuration for the attached device
update	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 appdevelopment or none . Use none to remove the appdevelopment 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>-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
'ABCDEF82513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A5588B42085
1EE4F3F1A7DC51399ED'.

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

► 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.

► 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, azsphere claims the attached device.

► Global parameters

Example

```
azsphere device claim

Claiming device.
Claiming attached device ID
'ABCDE082513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A5588B42085
1EE4F3F1A7DC51399ED' into tenant ID 'd343c263-4aa3-4558-adbb-d3fc34631800'.
Successfully claimed device ID
'ABCDE082513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A5588B42085
1EE4F3F1A7DC51399ED' 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
list-installed	Provides details about the images that are currently installed on the attached device
list-targeted	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.

► 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
-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.
-f, --full	Lists both customer and system software images that will be installed on the device. By default, lists only customer application images.

► Global parameters

Example

```
azsphere device image list-targeted

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.
```

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.
-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.
-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
'ABCDE082513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A5588B42085
1EE4F3F1A7DC51399ED'.
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
'ABCDE082513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A5588B42085
1EE4F3F1A7DC51399ED' 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 -n BlinkLink --dependentfeedid
3369f0e1-dedf-49ec-a602-2aa98669fd61

Getting the details for device with ID
'ABCDE082513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A5588B42085
1EE4F3F1A7DC51399ED'.
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
'ABCDE082513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A5588B42085
1EE4F3F1A7DC51399ED' to feed with ID '6daf66aa-5a3e-41f9-9659-6c30a2f7673f'.
Command completed successfully in 00:00:07.1315217.
```

manufacturing-state, mfg

Manages the manufacturing state of the attached device.

OPERATION	DESCRIPTION
show	Displays the manufacturing state of the attached device
update	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.

► 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
-s, --state <i>SettableManufacturingStates</i>	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.

► Global parameters

Example

```

azsphere device prep-debug

Getting device capability configuration for application development.
Downloading device capability configuration for device ID
'ABCDE082513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A5588B42085
1EE4F3F1A7DC51399ED'.
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
'ABCDE082513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A5588B42085
1EE4F3F1A7DC51399ED'.
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
'ABCDE082513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A5588B42085
1EE4F3F1A7DC51399ED' 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. See [Examples](#) for details.

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.
-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.
-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 --newskuusername 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
'ABCDE082513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A5588B42085
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 'AppUpdateGroup'.
Setting device group ID 'bbc91f02-1de4-43a3-bcf2-f6f0994ac723' for device with ID
'ABCDE082513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A5588B42085
1EE4F3F1A7DC51399ED'.
Creating a new SKU with name 'TestSKU'.
Setting product SKU to '5d88f658-be0c-4814-8319-473f21f4f88f' for device with ID
'ABCDE082513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A5588B42085
1EE4F3F1A7DC51399ED'.
Successfully set up device
'ABCDE082513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A5588B42085
1EE4F3F1A7DC51399ED' 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
'ABCDE082513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A5588B42085
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 'd80bf785-acae-497c-a62c-21a6ce65b81f' for device with ID
'ABCDE082513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A5588B42085
1EE4F3F1A7DC51399ED'.
Setting product SKU to '2bc8c605-6f8f-4802-ba69-c57d63e9c6dd' for device with ID
'ABCDE082513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A5588B42085
1EE4F3F1A7DC51399ED'.
Successfully set up device
'ABCDE082513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A5588B42085
1EE4F3F1A7DC51399ED' 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
'ABCDE082513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A5588B42085
1EE4F3F1A7DC51399ED'.
Getting the details of device group with ID ''.
Removing applications from device.
Component '54aca7c-7719-461a-89db-49c807e0fa4d' deleted.
Successfully removed applications from device.
Locking device.
Downloading device capability configuration for device ID
'ABCDE082513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A5588B42085
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 '655d7b12-07ad-4e8a-b104-c0ec494b8489' for device with ID
'ABCDE082513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A5588B42085
1EE4F3F1A7DC51399ED'.
Successfully set up device
'ABCDE082513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A5588B42085
1EE4F3F1A7DC51399ED' for OTA loading.
Command completed successfully in 00:00:11.1988981.
```

recover

Replaces the system software on the device.

Optional parameters

PARAMETER	DESCRIPTION
-b, --bootloader <i>filename</i>	Specifies the filename of the recovery bootloader to use. By default, the command uses recovery-loader.bin from the <i>folder</i> named with the --images flag.
-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.
-m, --manifest <i>filename</i>	Specifies the path to the recovery image manifest file. By default, the command uses the recovery.imagemanifest file from the <i>folder</i> specified with the --images flag.

► Global parameters

Example

```
azsphere device recover -i .\recovery`
```

restart

Restarts the attached device.

► 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.

► Global parameters

Example

```
azsphere device show-attached
Device ID:
ABCDE082513B529C45098884F882B2CA6D832587CAAЕ1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A5588B420851
EE4F3F1A7DC51399ED
Command completed successfully in 00:00:07.7393213.
```

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
-i, --deviceid <i>GUID</i>	Specifies the ID of the device. By default, shows information about the attached device.

► Global parameters

Example

```
azsphere device show-ota-config
Retrieved the over-the-air update configuration for device with ID
'ABCDEF864D649176A4C5F26FE01EAD92F01BA0C50A20E9F6E441F7C5B66DF193E775524D70C7176AF2592F94729C9936FE4ABDDA9B45B
8B76123682509ABCDEF':
--> 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.

► 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.

OPERATION	DESCRIPTION
delete	Deletes the current application from the device.
deploy	Loads an application onto the device.
show-quota	Displays the amount of storage used by the current application on the device.
show-status	Returns the status of the current application on the device.
start	Starts the application that is loaded on the device.

OPERATION	DESCRIPTION
stop	Stops the application that is running on the device.

sideload delete

Deletes applications from the device.

Optional parameters

PARAMETER	DESCRIPTION
-i, --componentid <i>GUID</i>	Specifies the ID of the component to delete. If omitted, deletes all applications.

► 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.

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 appdevelopment capability.

Optional parameters

PARAMETER	DESCRIPTION
-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
<code>-i, --componentid GUID</code>	Specifies the ID of the component for which to return storage information. If omitted, displays information for all applications.

► 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
<code>-i, --componentid GUID</code>	Specifies the ID of the component for which to display status. If omitted, shows status of all components.

► 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
<code>-i, --componentid GUID</code>	Specifies the ID of the component to start. If omitted, starts all applications.

PARAMETER	DESCRIPTION
-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.

► Global parameters

Example

```

azsphere device update-device-group --devicegroupid 63bbe6ea-14be-4d1a-a6e7-03591d882b42
Successfully moved device
'94B27082513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A5588B42085
1EE4F3F1A7DC51399ED' 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.

► Global parameters

Example

```

azsphere device update-sku --skuid 319725fd-1591-4a92-be9a-2f8cf90707f1
Successfully set SKU '319725fd-1591-4a92-be9a-2f8cf90707f1' for device ID
'94B27082513B529C45098884F882B2CA6D832587CAAE1A90B1CEC4A376EA2F22A96C4E7E1FC4D2AFF5633B68DB68FF4420A5588B42085
1EE4F3F1A7DC51399ED' 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
add	Adds the details of a wireless network to the device.
delete	Removes the details of a wireless network from the device.
disable	Disables a wireless network on the device.
enable	Enables a wireless network on the device.
list	Lists the current Wi-Fi configuration for the device.

OPERATION	DESCRIPTION
scan	Scans for available networks.
show-status	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.

Required parameters

PARAMETER	DESCRIPTION
-s, --ssid string	Specifies the SSID of the network. Network SSIDs are case-sensitive.

Optional parameters

PARAMETER	DESCRIPTION
-k, --key string	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.

► Global parameters

Example

```
azsphere device wifi add -s MyNetwork -k 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 -s "New SSID" -k "key \"value\" with quotes"
```

wifi delete

Deletes information about a wireless network from the device.

Required parameters

PARAMETER	DESCRIPTION
-i, --id integer	Specifies the ID of the network to delete.

► 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.

► 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.

► 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.

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. Displays up to 35 wireless networks.

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.

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

11/15/2018 • 2 minutes to read

Creates and manages device groups.

OPERATION	DESCRIPTION
create	Creates a new device group
feed	Manages the feeds that a device group targets
list	Lists all device groups
show	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
add	Adds a feed to the device group
list	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

2/14/2019 • 3 minutes to read

Manages feeds in an Azure Sphere tenant.

OPERATION	DESCRIPTION
create	Creates a new feed
image-set, ims	Manages image sets in a feed
list	Lists all feeds in the Azure Sphere tenant
list-device-groups	Lists all device groups that a feed targets
show	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 azsphere feed list 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
add	Adds an image set to a feed
list	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

2/14/2019 • 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
get-support-data	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

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

image, img

1/7/2019 • 3 minutes to read

Manages images.

OPERATION	DESCRIPTION
package-application	Creates an image package
package-board-config	Creates a board configuration image package
show	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.

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>-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 Use beta API for more information. Current valid values are 1 for production, and 1+beta1811 to use the preview beta API.
<code>-x, --executables executable1,executable2 ...</code>	Subpaths to one or more files to mark as executable in the image package. The EntryPoint listed in the <code>app_manifest</code> files is always marked as executable, so the <code>-x</code> flag is required only if other executables are present. By default, files are not executable when packaged into an image. The subpaths are relative to the <code>--input path</code> . 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 <code>-x</code> for each executable file, or use it only once and supply multiple paths separated by commas without intervening spaces.

► Global parameters

Example

```
azsphere image package-application --input bin --output myimage.imagepackage --sysroot 1
```

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
-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.
-o, --output <i>filename</i>	Specifies a filename for the output image package.
-p, --preset <i>string</i>	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".

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.
-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.

► 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

11/15/2018 • 2 minutes to read

Manages image sets in an Azure Sphere tenant.

OPERATION	DESCRIPTION
create	Creates a new image set
list	Lists all image sets in the Azure Sphere tenant
show	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

8/13/2018 • 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

11/15/2018 • 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

11/15/2018 • 2 minutes to read

Displays version of the current installed Azure Sphere SDK.

► Global parameters

Example

```
azsphere show-version
18.11
Command completed successfully in 00:00:00.6230420.
```

sku

8/13/2018 • 2 minutes to read

Manages SKUs in an Azure Sphere tenant.

OPERATION	DESCRIPTION
create	Creates a new SKU
list	Lists all SKUs in the Azure Sphere tenant
show	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 -n DW100SKU -d "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

11/15/2018 • 2 minutes to read

Manages an Azure Sphere tenant.

OPERATION	DESCRIPTION
create	Creates a new tenant
download-ca-certificate	Downloads the CA certificate for the current tenant
download-validation-certificate	Downloads the validation certificate for the current tenant, based on the provided verification code
list	Lists the available Azure Sphere tenants
select	Selects the default Azure Sphere tenant to use on this PC
show-selected	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
-f, --force	Forces creation of a new Azure Sphere tenant in the current user's Azure Active Directory (AAD).

► 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
<code>-o, --output <i>filepath</i></code>	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
<code>-c, --verificationcode <i>string</i></code>	Provides the verification code required to get a validation certificate.
<code>-o, --output <i>filepath</i></code>	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

2/14/2019 • 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
applibs/gpio.h	Interacts with GPIOs (general-purpose input/output).
applibs/i2c.h	Interacts with I2C (Inter-Integrated Circuit) interfaces.
applibs/log.h	Logs debug messages that are displayed when you debug an application through the Azure Sphere SDK.
applibs/networking.h	Interacts with the networking subsystem to query the network state, and to get and set the network service configuration.
applibs/rtc.h	Interacts with the real-time clock (RTC).
applibs/spi.h	Interacts with SPI (Serial Peripheral Interface) devices.
applibs/storage.h	Interacts with on-device storage, which includes read-only storage and mutable storage.
applibs/uart.h	Opens and interacts with a UART (Universal Asynchronous Receiver/Transmitter) on a device.
applibs/wificonfig.h	Manages Wi-Fi network configurations on a device.

Applibs gpio.h

2/14/2019 • 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](#)
- [UART sample application](#)
- [Sample: CurlMultiHttps](#)
- [Sample: System Time](#)
- [Sample: External MCU update - reference solution](#)

Functions

FUNCTION	DESCRIPTION
GPIO_GetValue	Gets the current value of a GPIO.
GPIO_OpenAsInput	Opens a GPIO (General Purpose Input/Output) as an input.
GPIO_OpenAsOutput	Opens a GPIO (General Purpose Input/Output) as an output.
GPIO_SetValue	Sets the output value for an output GPIO.

Enums

ENUM	DESCRIPTION
GPIO_OutputMode	The options for the output mode of a GPIO.
GPIO_Value	The possible read/write values for a GPIO.

Typedefs

TYPEDEF	DESCRIPTION
GPIO_Id	Specifies the type of a GPIO ID, which is used to specify a GPIO peripheral instance.
GPIO_OutputMode_Type	Specifies the type of the GPIO output mode .
GPIO_Value_Type	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

2/14/2019 • 2 minutes to read

BETA feature

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

2/14/2019 • 2 minutes to read

BETA feature

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

2/14/2019 • 2 minutes to read

BETA feature

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

2/14/2019 • 2 minutes to read

BETA feature

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

2/14/2019 • 2 minutes to read

BETA feature

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

2/14/2019 • 2 minutes to read

BETA feature

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

2/14/2019 • 2 minutes to read

BETA feature

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

2/14/2019 • 2 minutes to read

BETA feature

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

2/14/2019 • 2 minutes to read

BETA feature

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
Log_Debug	Logs and formats a debug message with printf formatting.
Log_DebugVarArgs	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

2/14/2019 • 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.

Application manifest requirements

These functions are only permitted if the application has the NetworkConfig capability in the [application manifest](#).

Concepts and samples

- [Sample: Private Ethernet](#)
- [Sample: CurlEasyHttps](#)

Functions

FUNCTION	DESCRIPTION
Networking_GetInterfaceCount	Gets the number of network interfaces in an Azure Sphere device.
Networking_GetInterfaces	Gets the list of network interfaces in an Azure Sphere device.
Networking_GetNtpState	Indicates whether the NTP time-sync service is enabled.
Networking_InitDhcpServerConfiguration	Initializes a Networking_DhcpServerConfiguration struct with the default DHCP server parameters.
Networking_InitStaticIpConfiguration	Initializes a Networking_StaticIpConfiguration struct with the default static IP parameters.
Networking_IsNetworkingReady	Verifies whether internet connectivity is available and time is synced.
Networking_SetInterfaceState	Enables or disables a network interface.
Networking_SetNtpState	Enables or disables the NTP time-sync service.
Networking_SetStaticIp	Sets the static IP configuration for a network interface.
Networking_StartDhcpServer	Registers and starts the DHCP server for a network interface.
Networking_StartSntpServer	Registers and starts the SNTP server for a network interface.

Structs

STRUCT	DESCRIPTION
Networking_DhcpServerConfiguration	The DHCP server configuration for a network interface.
Networking_NetworkInterface	The properties of a network interface.
Networking_StaticIpConfiguration	The static IP address configuration for a network interface.

Enums

ENUM	DESCRIPTION
Networking_InterfaceMedium	The valid network technologies used by the network interface.
Networking_IpConfiguration	The possible IP configuration options for a network interface.

Typedefs

TYPedef	DESCRIPTION
Networking_InterfaceMedium_Type	Specifies the type of Networking_InterfaceMedium enum values.
Networking_IpConfiguration_Type	Specifies the type of Networking_IpConfiguration enum values.

Networking_GetInterfaceCount Function

2/14/2019 • 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 Ethernet](#)

Networking_GetInterfaces Function

2/14/2019 • 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* parameter 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 Ethernet](#)

Networking_GetNtpState Function

2/14/2019 • 2 minutes to read

BETA feature

Header: #include <applibs/networking.h>

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

2/14/2019 • 2 minutes to read

BETA feature

Header: #include <applibs/networking.h>

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

1/7/2019 • 2 minutes to read

BETA feature

Header: #include <applibs/networking.h>

Initializes a [Networking_StaticIpConfiguration](#) struct with the default static IP parameters.

```
static inline void Networking_InitStaticIpConfiguration(Networking_StaticIpConfiguration
*staticIpConfiguration);
```

Parameters

- `staticIpConfiguration` A pointer to a [Networking_StaticIpConfiguration](#) struct that returns the default static IP parameters.

Application manifest requirements

The [application manifest](#) must include the NetworkConfig capability.

Networking_IsNetworkingReady Function

2/14/2019 • 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

1/7/2019 • 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.

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.

Networking_SetNtpState Function

1/7/2019 • 2 minutes to read

BETA feature

Header: #include <applibs/networking.h>

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.

Application manifest requirements

The [application manifest](#) must include the NetworkConfig capability.

Networking_SetStaticIp Function

1/7/2019 • 2 minutes to read

BETA feature

Header: #include <applibs/networking.h>

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.

Application manifest requirements

The [application manifest](#) must include the NetworkConfig capability.

Networking_StartDhcpServer Function

2/14/2019 • 2 minutes to read

BETA feature

Header: #include <applibs/networking.h>

Registers and starts the DHCP server for a network interface.

The DHCP server configuration passed with 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 [Networking_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.

Application manifest requirements

The [application manifest](#) must include the NetworkConfig capability.

Networking_StartSntpServer Function

2/14/2019 • 2 minutes to read

BETA feature

Header: #include <applibs/networking.h>

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.

Application manifest requirements

The [application manifest](#) must include the NetworkConfig capability.

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_NetworkInterface Struct

2/14/2019 • 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_IpConfiguration_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_IpConfiguration_Type ipConfigurationType

The [Networking_IpConfiguration](#) enum that contains the IP configuration types for the interface.

Networking_InterfaceMedium_Type InterfaceMediumType

The [Networking_InterfaceMedium](#) enum that contains the network types for the interface.

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_InterfaceMedium Enum

1/7/2019 • 2 minutes to read

BETA feature

Header: #include <applibs/networking.h>

The valid network technologies used by the network interface.

```
typedef enum {
    GPIO_Value_Low = 0,
    GPIO_Value_High = 1
} GPIO_Value;
```

VALUES	DESCRIPTIONS
Networking_InterfaceMedium_Wifi = 0	Wi-Fi
Networking_InterfaceMedium_Ethernet = 1	Ethernet

Concepts and samples

- [Sample: Private Ethernet](#)

Networking_IpConfiguration Enum

1/7/2019 • 2 minutes to read

BETA feature

Header: #include <applibs/networking.h>

The possible IP configuration options for a network interface.

```
typedef enum {
    GPIO_Value_Low = 0,
    GPIO_Value_High = 1
} GPIO_Value;
```

VALUES	DESCRIPTIONS
Networking_Networking_IpConfiguration_DhcpNone	The interface does not have a DHCP service attached. Networking_SetStaticIp must be used to configure a static IP for the interface.
Networking_IpConfiguration_DhcpClient	The interface has a DHCP client service attached.

Concepts and samples

- [Sample: Private Ethernet](#)

Networking_InterfaceMedium_Type Typedef

1/7/2019 • 2 minutes to read

Header: #include <applibs/networking.h>

Specifies the type of [Networking_InterfaceMedium](#) enum values.

```
typedef uint8_t Networking_InterfaceMedium_Type;
```

Networking_IpConfiguration_Type Typedef

1/7/2019 • 2 minutes to read

Header: #include <applibs/networking.h>

Specifies the type of [Networking_IpConfiguration](#) enum values.

```
typedef uint8_t Networking_IpConfiguration_Type;
```

Applibs rtc.h

1/7/2019 • 2 minutes to read

Header: #include <applibs/rtc.h>

The Applibs rtc header contains functions that interacts 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
clock_systohc	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

2/14/2019 • 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
SPI_SPIMaster_InitConfig	Initializes a SpiMaster_Config struct with the default SPI master interface settings.
SPI_SPIMaster_InitTransfers	Initializes an array of SpiMaster_Transfer structs with the default SPI master transfer settings.
SPI_SPIMaster_Open	Opens and configures an SPI master interface for exclusive use, and returns a file descriptor to use for subsequent calls.
SPI_SPIMaster_SetBitOrder	Configures the order for transferring data bits on an SPI master interface.
SPI_SPIMaster_SetBusSpeed	Sets the SPI bus speed for operations on an SPI master interface.
SPI_SPIMaster_SetMode	Sets the communication mode for an SPI master interface.
SPI_SPIMaster_TransferSequential	Performs a sequence of half-duplex read or write transfers using the SPI master interface.
SPI_SPIMaster_WriteThenRead	Performs a sequence of a half-duplex writes immediately followed by a half-duplex read using the SPI master interface.

Structs

STRUCT	DESCRIPTION
SPIMaster_Config	The configuration options for opening an SPI master interface.
SPIMaster_Transfer	The description of an SPI master transfer operation.

Enums

ENUM	DESCRIPTION
SPI_BitOrder	The possible SPI bit order values.
SPI_ChipSelectPolarity	The possible chip select polarity values for an SPI interface.
SPI_Mode	The possible communication mode values for an SPI interface.
SPI_Mode	The possible <i>flags</i> values for a SPIMaster_Transfer struct.

Typdefs

TYPEDEF	DESCRIPTION
SPI_ChipSelectId	An SPI chip select ID.
SPI_Interfaceld	The ID for an SPI interface instance.

SPI_BitOrder Enum

2/14/2019 • 2 minutes to read

BETA feature

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

2/14/2019 • 2 minutes to read

BETA feature

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

2/14/2019 • 2 minutes to read

BETA feature

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

2/14/2019 • 2 minutes to read

BETA feature

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

2/14/2019 • 2 minutes to read

BETA feature

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

2/14/2019 • 2 minutes to read

BETA feature

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

2/14/2019 • 2 minutes to read

BETA feature

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.
- ENOENT: the specified IDs are invalid.

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

2/14/2019 • 2 minutes to read

BETA feature

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

2/14/2019 • 2 minutes to read

BETA feature

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

2/14/2019 • 2 minutes to read

BETA feature

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

2/14/2019 • 2 minutes to read

BETA feature

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.

Application manifest requirements

To access individual SPI interfaces, your application must identify them in the `SpiMaster` field of the [application manifest](#).

SPIMaster_WriteThenRead Function

2/14/2019 • 2 minutes to read

BETA feature

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

2/14/2019 • 2 minutes to read

BETA feature

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

2/14/2019 • 2 minutes to read

BETA feature

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

2/14/2019 • 2 minutes to read

BETA feature

Header: #include <applibs/spi.h>

A SPI chip select ID.

```
typedef int SPI_ChipSelectId;
```

SPI_InterfaceId Typedef

2/14/2019 • 2 minutes to read

BETA feature

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
Storage_DeleteMutableFile	Deletes any existing data previously written to the mutable file when Storage_OpenMutableFile is called.
Storage_GetAbsolutePathInImagePackage	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.
Storage_OpenFileInImagePackage	Takes a relative path inside the image package and returns an opened read-only file descriptor.
Storage_OpenMutableFile	Provides a file descriptor to a file placed in a location on disk where data will be persisted over device reboot.

Storage_DeleteMutableFile Function

1/7/2019 • 2 minutes to read

BETA feature

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.

- EPERM: The application does not have the application capability (MutableStorage) required in order to use this API.
- 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

1/7/2019 • 2 minutes to read

BETA feature

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.

- EPERM: The application does not have the application capability (MutableStorage) required in order to use this API.
- 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 uart.h

2/14/2019 • 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

- [Quickstart: Build the Blink sample application](#)
- [Sample: External MCU update - reference solution](#)

Functions

FUNCTION	DESCRIPTION
<code>UART_InitConfig</code>	Initializes a UART config 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>Networking_NetworkInterface</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 UART_Config struct.
UART_BlockingMode_Type	Specifies the type of the blockingMode value for the UART_Config struct.
UART_DataBits_Type	Specifies the type of the dataBits value for the UART_Config struct.
UART_FlowControl_Type	Specifies the type of the flowControl value for the UART_Config struct.
UART_Id	A UART ID, which specifies a UART peripheral instance.
UART_Parity_Type	Specifies the type of the parity value for the UART_Config struct.
UART_StopBits_Type	Specifies the type of the stopBits value for the UART_Config 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

1/7/2019 • 2 minutes to read

Header: #include <applibs/uart.h>

Opens and configures a UART, and returns a file descriptor to use for subsequent calls.

Opens the UART for exclusive access.

```
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 networking.h

2/14/2019 • 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
WifiConfig_ForgetAllNetworks	Removes all stored Wi-Fi networks from the device. Disconnects the device from any connected network.
WifiConfig_ForgetNetwork	Removes a Wi-Fi network from the device. Disconnects the device from the network if it's currently connected.
WifiConfig_GetCurrentNetwork	Gets the Wi-Fi network that is connected to the device.
WifiConfig_GetScannedNetworks	Gets the Wi-Fi networks found by the last scan operation.
WifiConfig_GetStoredNetworkCount	Gets the number of stored Wi-Fi networks on the device.
WifiConfig_GetStoredNetworks	Retrieves all stored Wi-Fi networks on the device.
WifiConfig_StoreOpenNetwork	Stores an open Wi-Fi network without a key.
WifiConfig_StoreWpa2Network	Stores a WPA2 Wi-Fi network that uses a pre-shared key.

FUNCTION	DESCRIPTION
WifiConfig_TriggerScanAndGetScannedNetworkCount	Starts a scan to find all available Wi-Fi networks.

Structs

|[WifiConfig_ConnectedNetwork](#) | The properties of a connected Wi-Fi network, which represent a 802.11 Basic Service Set (BSS). |[WifiConfig_ScannedNetwork](#) | The properties of a scanned Wi-Fi network, which represent a 802.11 Basic Service Set (BSS). |[WifiConfig_StoredNetwork](#) | The properties of a stored Wi-Fi network, which represents a 802.11 Service Set.

Enums

ENUM	DESCRIPTION
WifiConfig_Security	The security key setting for a Wi-Fi network.

Typedefs

TYPedef	DESCRIPTION
WifiConfig_Security_Type	Specifies the type of the security settings values for the WifiConfig_Security enum.

WifiConfig_ForgetAllNetworks Function

1/7/2019 • 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.

```
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.

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

1/7/2019 • 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. 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` is NULL.
- ENOENT: the `storedNetwork` does not match any of the stored networks.
- EINVAL: the `storedNetwork` 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

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

1/7/2019 • 2 minutes to read

Header: #include <applibs/wificonfig.h>

Gets the 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.

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

1/7/2019 • 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.

- ERANGE: the `ssidLength` is 0 or greater than `WIFICONFIG_SSID_MAX_LENGTH`.
- EINVAL: the `storedNetwork` 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

2/14/2019 • 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.

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

2/14/2019 • 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.

- EFAULT: the *ssid* is NULL.
- ERANGE: the *ssidLength* is 0 or greater than WIFICONFIG_SSID_MAX_LENGTH.
- EINVAL: the *storedNetwork* 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_StoreOpenNetwork Function

1/7/2019 • 2 minutes to read

Header: #include <applibs/wificonfig.h>

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.

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

1/7/2019 • 2 minutes to read

Header: #include <applibs/wificonfig.h>

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 a SSID byte array with unspecified character encoding that identifies the Wi-Fi network.
- `ssidLength` The number of bytes in the *sid* 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* is 0 or greater than WIFICONFIG_WPA2_KEY_MAX_BUFFER_SIZE and WIFICONFIG_SSID_MAX_LENGTH.
- 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

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

1/7/2019 • 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.

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

1/7/2019 • 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
} GPIO_OutputMode;
```

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

1/22/2019 • 8 minutes to read

Application

An executable program that runs on an Azure Sphere device. Customers write applications to perform tasks that are specific to the customer's connected 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 applications.

Application libraries (applibs)

The Microsoft-authored custom libraries that support 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. The Azure Sphere templates and sample applications are designed to work with an RDB.

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 operating system (HLOS)

The second level of the multi-layer Azure Sphere OS architecture. The HLOS is a custom Linux kernel.

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.

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 an 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

2/14/2019 • 13 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.

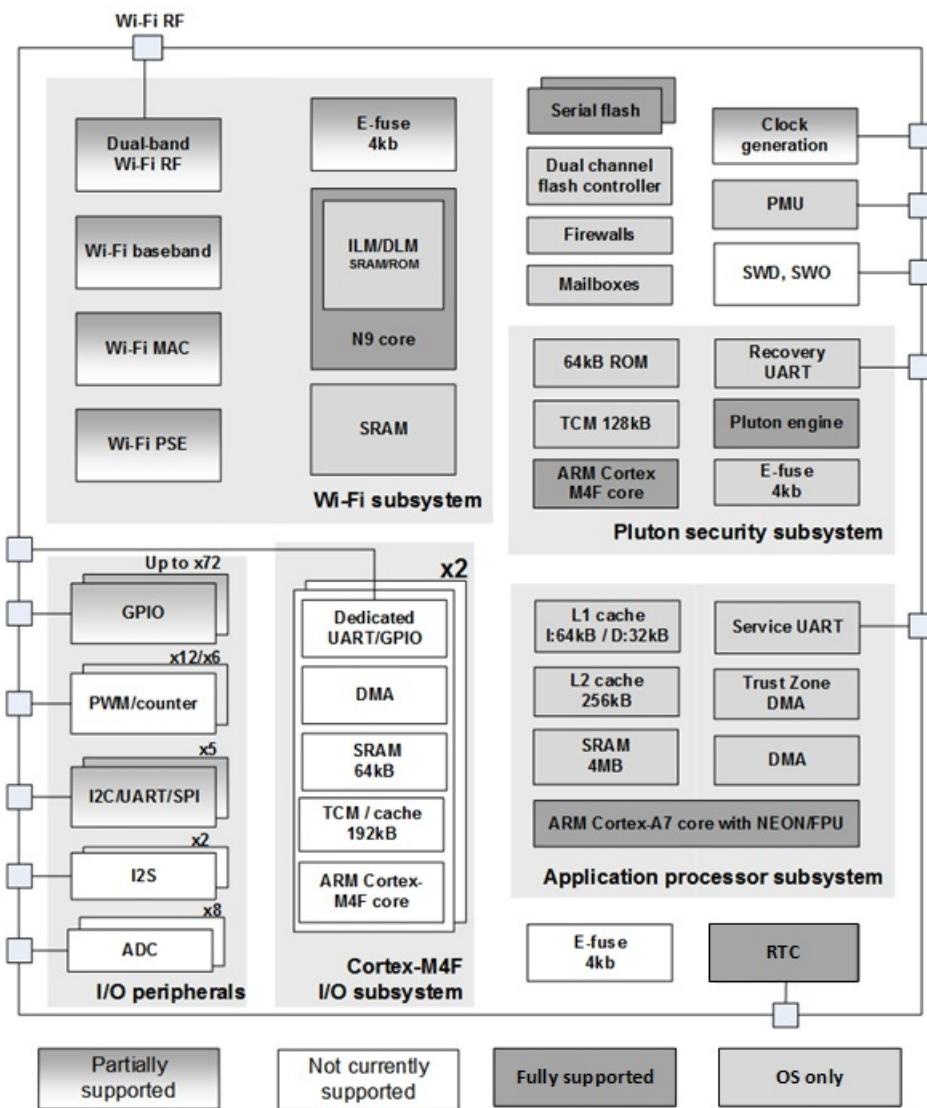
IMPORTANT

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 questions about the timeframe or details of any Azure Sphere features, please contact your Microsoft representative.

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. Currently, I/O peripherals are mapped only to the ARM Cortex-A7 core.

GPIO/PWM/counters

Some pins are multiplexed between GPIO, pulse width modification (PWM), and hardware counters. Neither PWM nor the counters are currently 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 not currently supported.

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). ADC is not currently supported.

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 not yet supported.

Customers who require M4F I/O subsystem support are encouraged to contact Microsoft to learn more about availability.

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) and serial-wire output (SWO, pin 100) are not currently supported.

Debugging the customer 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 customer 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	
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	

PIN#	PIN NAME	MAJOR FUNCTIONS	TYPE	DESCRIPTION	COMMENTS
4	NC				
5	NC				
6	AVDD_1V6_WF_RX		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	PWM is not currently supported
14	GPIO1	GPIO1/PWM1	DIO	Interrupt-capable GPIO multiplexed with PWM output	PWM is not currently supported
15	GPIO2	GPIO2/PWM2	DIO	Interrupt-capable GPIO multiplexed with PWM output	PWM is not currently supported
16	GPIO3	GPIO3/PWM3	DIO	Interrupt-capable GPIO multiplexed with PWM output	PWM is not currently supported
17	GPIO4	GPIO4/PWM4	DIO	Interrupt-capable GPIO multiplexed with PWM output	PWM is not currently supported

PIN#	PIN NAME	MAJOR FUNCTIONS	TYPE	DESCRIPTION	COMMENTS
18	GPIO5	GPIO5/PWM5	DIO	Interrupt-capable GPIO multiplexed with PWM output	PWM is not currently supported
19	GPIO6	GPIO6/PWM6	DIO	Interrupt-capable GPIO multiplexed with PWM output	PWM is not currently supported
20	GPIO7	GPIO7/PWM7	DIO	Interrupt-capable GPIO multiplexed with PWM output	PWM is not currently supported
21	GPIO8	GPIO8/PWM8	DIO	Interrupt-capable GPIO multiplexed with PWM output	PWM is not currently supported
22	GPIO9	GPIO9/PWM9	DIO	Interrupt-capable GPIO multiplexed with PWM output	PWM is not currently supported
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	PWM is not currently supported
26	GPIO11	GPIO11/PWM11	DIO	Interrupt-capable GPIO multiplexed with PWM output	PWM is not currently supported
27	GPIO12		DIO	Interrupt-capable GPIO	Interrupts are not currently supported
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

PIN#	PIN NAME	MAJOR FUNCTIONS	TYPE	DESCRIPTION	COMMENTS
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	
42	GPIO28	GPIO28/ MISO0/RXD0/SDA0	DIO	GPIO multiplexed with ISU 0 functions	
43	GPIO29	GPIO29/CSA0/CTS0	DIO	GPIO multiplexed with ISU 0 functions	
44	DVDD_1V15		PI	1.15V power rail	

PIN#	PIN NAME	MAJOR FUNCTIONS	TYPE	DESCRIPTION	COMMENTS
45	GPIO30	GPIO30/CSB0	DIO	GPIO multiplexed with ISU 0 functions	Currently supports GPIO only
46	GPIO31	GPIO31/SCLK1/TXD1	DIO	GPIO multiplexed with ISU 1 functions	
47	GPIO32	GPIO32/MOSI1/RTS1/SCL1	DIO	GPIO multiplexed with ISU 1 functions	
48	GPIO33	GPIO33/MISO1/RXD1/SDA1	DIO	GPIO multiplexed with ISU 1 functions	
49	GPIO34	GPIO34/CSA1/CTS1	DIO	GPIO multiplexed with ISU 1 functions	
50	GPIO35	GPIO35/CSB1	DIO	GPIO multiplexed with ISU 1 functions	Currently supports GPIO only
51	GPIO36	GPIO36/SCLK2/TXD2	DIO	GPIO multiplexed with ISU 2 functions	
52	GPIO37	GPIO37/MOSI2/RTS2/SCL2	DIO	GPIO multiplexed with ISU 2 functions	
53	GPIO38	GPIO38/MISO2/RXD2/SDA2	DIO	GPIO multiplexed with ISU 2 functions	
54	GPIO39	GPIO39/CSA2/CTS2	DIO	GPIO multiplexed with ISU 2 functions	
55	GPIO40	GPIO40/CSB2	DIO	GPIO multiplexed with ISU 2 functions	Currently supports GPIO only
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	ADC input is not currently supported

PIN#	PIN NAME	MAJOR FUNCTIONS	TYPE	DESCRIPTION	COMMENTS
59	GPIO42	GPIO42/ADC1	DIO	GPIO multiplexed with ADC input	ADC input is not currently supported
60	GPIO43	GPIO43/ADC2	DIO	GPIO multiplexed with ADC input	ADC input is not currently supported
61	GPIO44	GPIO44/ADC3	DIO	GPIO multiplexed with ADC input	ADC input is not currently supported.
62	GPIO45	GPIO45/ADC4	DIO	GPIO multiplexed with ADC input	ADC input is not currently supported
63	GPIO46	GPIO46/ADC5	DIO	GPIO multiplexed with ADC input	ADC input is not currently supported
64	GPIO47	GPIO47/ADC6	DIO	GPIO multiplexed with ADC input	ADC input is not currently supported
65	GPIO48	GPIO48/ADC7	DIO	GPIO multiplexed with ADC input	ADC input is not currently supported
66	AVDD_2V5_ADC		PI	2.5V power rail for ADC	ADC input is not currently supported
67	VREF_ADC		AI	Reference voltage for ADC	ADC input is not currently supported
68	AVSS_2V5_ADC		P	Ground for ADC	ADC input is not currently supported
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 not currently supported
76	I2S_MCLK1_ALT		AO	Analog alternative to MCLK1	I2S is not currently supported
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_CLD_O		PI	1.6V power rail for the internal 1.15V core LDO	
87	PMU_CAP		A	Connect a capacitor between this pin and AVDD_3V3_BUC_K to maintain PMU stability	

PIN#	PIN NAME	MAJOR FUNCTIONS	TYPE	DESCRIPTION	COMMENTS
88	AVDD_3V3_BUCK		PI	3.3V power rail for internal 1.6V buck DC-DC converter	
89	AVDD_3V3_BUCK		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	Cortex-M4F is not currently supported
99	SWD_CLK		DI	ARM SWD for Cortex-M4F debug	Cortex-M4F is not currently supported
100	SWO		DO	ARM SWO for Cortex-M4F debug	Cortex-M4F is not currently supported

PIN#	PIN NAME	MAJOR FUNCTIONS	TYPE	DESCRIPTION	COMMENTS
101	GPIO56	GPIO56/TX0	DIO	GPIO multiplexed with I2S 0	I2S is not currently supported
102	GPIO57	GPIO57 /MCLK0	DIO	GPIO multiplexed with I2S 0	I2S is not currently supported
103	GPIO58	GPIO58/FS0	DIO	GPIO multiplexed with I2S 0	I2S is not currently supported
104	GPIO59	GPIO59/RX0	DIO	GPIO multiplexed with I2S 0	I2S is not currently supported
105	GPIO60	GPIO60/ BCLK0	DIO	GPIO multiplexed with I2S 0	I2S is not currently supported
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 not currently supported
109	GPIO62	GPIO62/ MCLK1	DIO	GPIO multiplexed with I2S 1	I2S is not currently supported
110	GPIO63	GPIO63/FS1	DIO	GPIO multiplexed with I2S 1	I2S is not currently supported
111	GPIO64	GPIO64/RX1	DIO	GPIO multiplexed with I2S 1	I2S is not currently supported
112	GPIO65	GPIO65/ BCLK1	DIO	GPIO multiplexed with I2S 1	I2S is not currently supported
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	

PIN#	PIN NAME	MAJOR FUNCTIONS	TYPE	DESCRIPTION	COMMENTS
115	GPIO68	GPIO68/ MISO3/RXD3/SD A3	DIO	GPIO multiplexed with ISU 3 functions	
116	GPIO69	GPIO69/CSA3/CT S3	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/SCL 4	DIO	GPIO multiplexed with ISU 4 functions	
121	DVDD_1V15		PI	1.15V power rail	
122	GPIO73	GPIO73/ MISO4/RXD4/SD A4	DIO	GPIO multiplexed with ISU 4 functions	
123	GPIO74	GPIO74/CSA4/CT S4	DIO	GPIO multiplexed with ISU 4 functions	
124	GPIO75	GPIO75/CSB4	DIO	GPIO multiplexed with ISU 4 functions	
125	SYSRST_N		DI	System reset, active low	
126	DVDD_1V15		PI	1.15V power rail	
127	SERVICE_RXD		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_TXD		DI	Azure Sphere service port	Not available for customer application use

PIN#	PIN NAME	MAJOR FUNCTIONS	TYPE	DESCRIPTION	COMMENTS
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	Cortex-M4F is not currently supported
139	IO0_GPIO86	IO0_GPIO86/ IO0_TXD	DO	Dedicated GPIO multiplexed with UART for I/O CM4F 0	Cortex-M4F is not currently supported
140	IO0_GPIO87	IO0_GPIO87/ IO0_RTS	DO	Dedicated GPIO multiplexed with UART for I/O CM4F 0	Cortex-M4F is not currently supported
141	IO0_GPIO88	IO0_GPIO88/ IO0_CTS	DI	Dedicated GPIO multiplexed with UART for I/O CM4F 0	Cortex-M4F is not currently supported
142	IO1_GPIO89	IO1_GPIO89/ IO1_RXD	DI	Dedicated GPIO multiplexed with UART for I/O CM4F 1	Cortex-M4F is not currently supported
143	IO1_GPIO90	IO1_GPIO90/ IO1_TXD	DO	Dedicated GPIO multiplexed with UART for I/O CM4F 1	Cortex-M4F is not currently supported

PIN#	PIN NAME	MAJOR FUNCTIONS	TYPE	DESCRIPTION	COMMENTS
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	Cortex-M4F is not currently supported
146	IO1_GPIO92	IO1_GPIO92/ IO1_CTS	DI	Dedicated GPIO multiplexed with UART for I/O CM4F 1	Cortex-M4F is not currently supported
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)	
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	

PIN#	PIN NAME	MAJOR FUNCTIONS	TYPE	DESCRIPTION	COMMENTS
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 development board user guide

2/14/2019 • 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

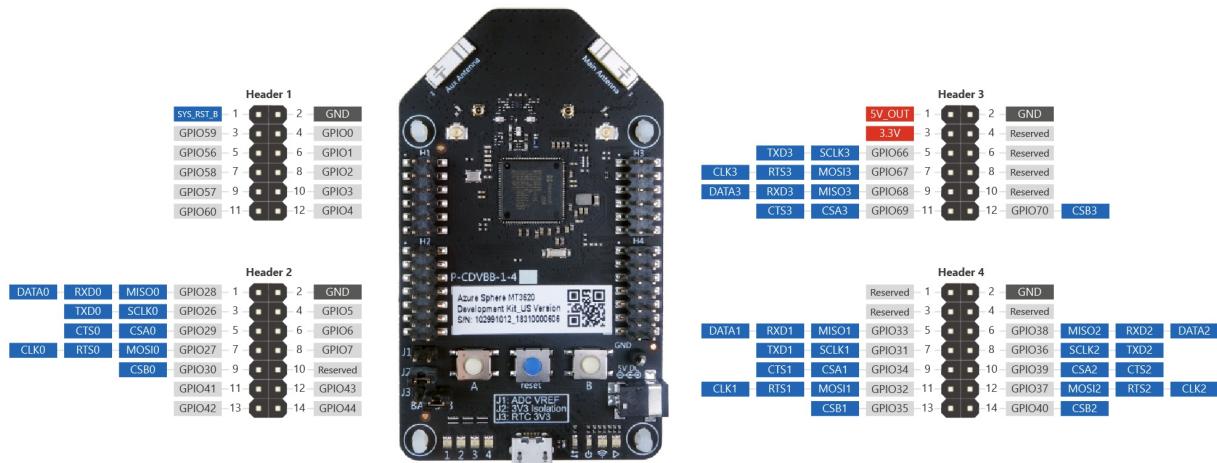
The board includes a red power-on LED that illuminates when the board is powered by USB, an external 5V supply, or an external 3.3V supply.

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.

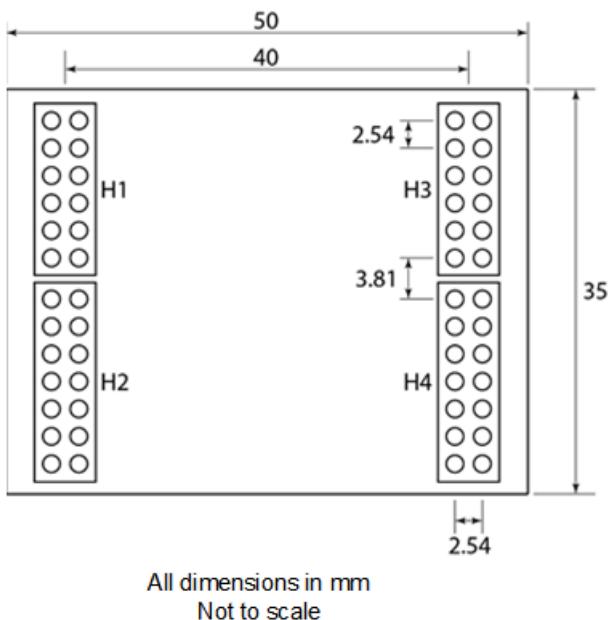
NOTE

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.



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.

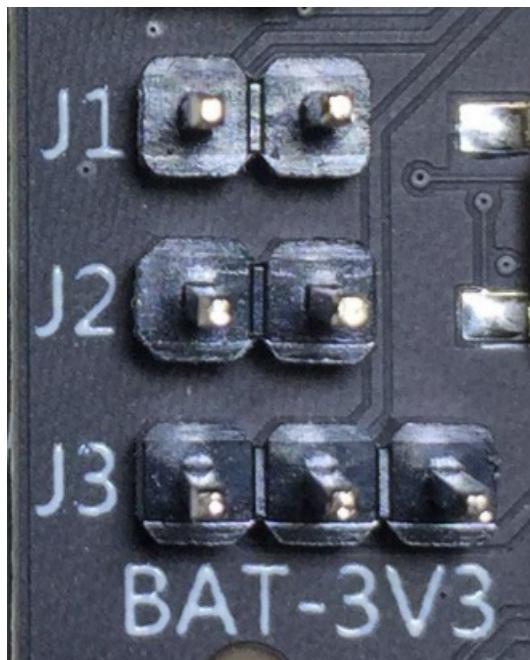
The on-board power supply is protected against reverse-voltage and over-current. If an over-current situation occurs, the protection circuit trips and isolates the incoming 5V supply from the rest of the on-board power supply

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
--------	----------	-------------	-------------

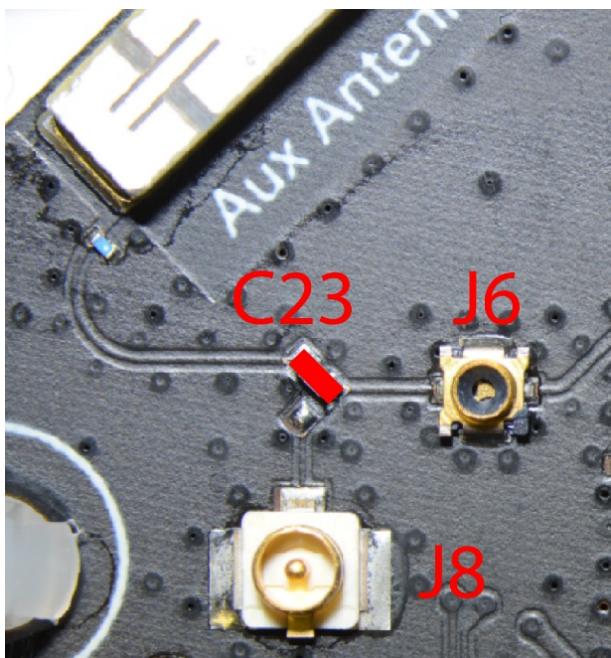
JUMPER	FUNCTION	DESCRIPTION	JUMPER PINS
J1	ADC VREF	<p>This jumper provides a way to set the ADC reference voltage. Place a link on J1 to connect the MT3620's 2.5V output to the ADC VREF pin, so that the ADC reference voltage is 2.5V. Alternatively, connect an external 1.8V reference voltage to pin 1 of the jumper.</p> <p>Note: 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.

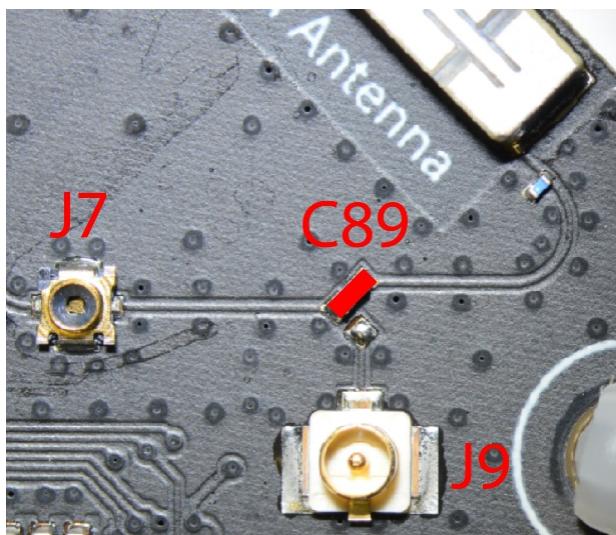
To enable and use the RF connectors, you must reorient capacitors C23 and/or C89. The first row in the following table shows the default configuration where the on-board chip antennas are in use, with the associated capacitor positions highlighted in red. The images on the second row show the re-oriented capacitor positions in green.

AUXILIARY ANTENNA

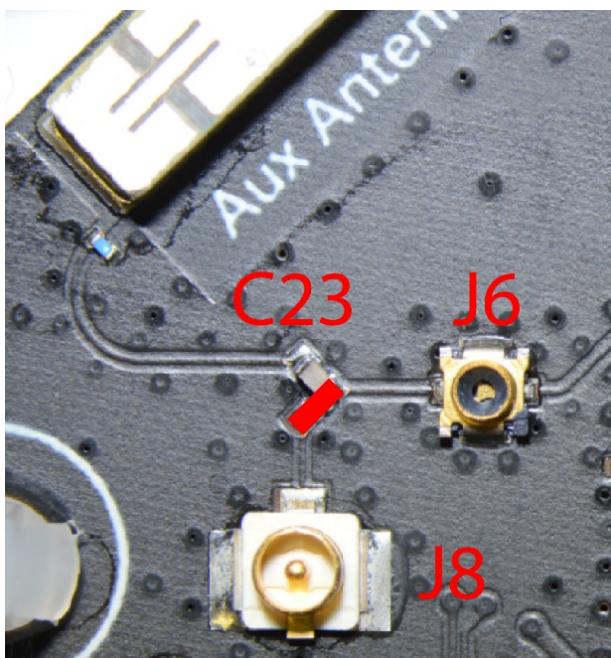


C23 default configuration, on-board chip antenna

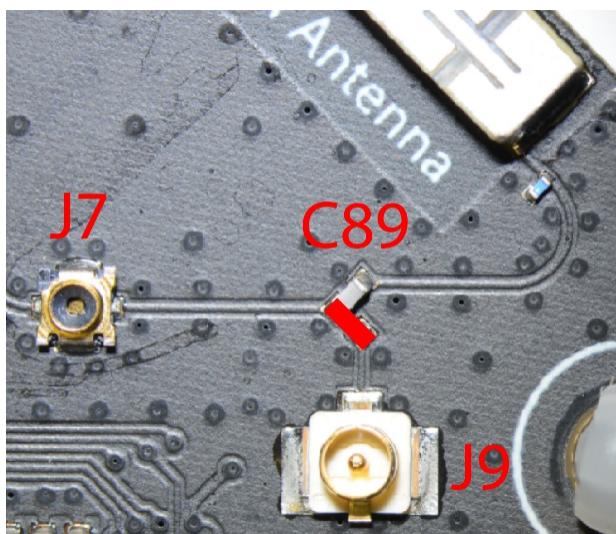
MAIN ANTENNA



C89 default configuration, on-board chip antenna



C23 alternate configuration – external antenna connects to J8



C89 alternate configuration – external antenna connects to J9

NOTE

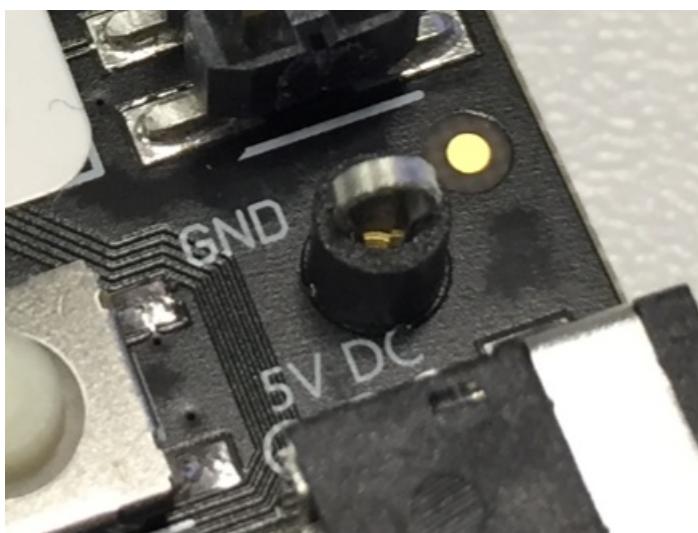
Connectors J6 and J7 are used for RF testing and calibration during manufacture and are not intended for permanent connection to test equipment or external antennas.

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

12/12/2018 • 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.

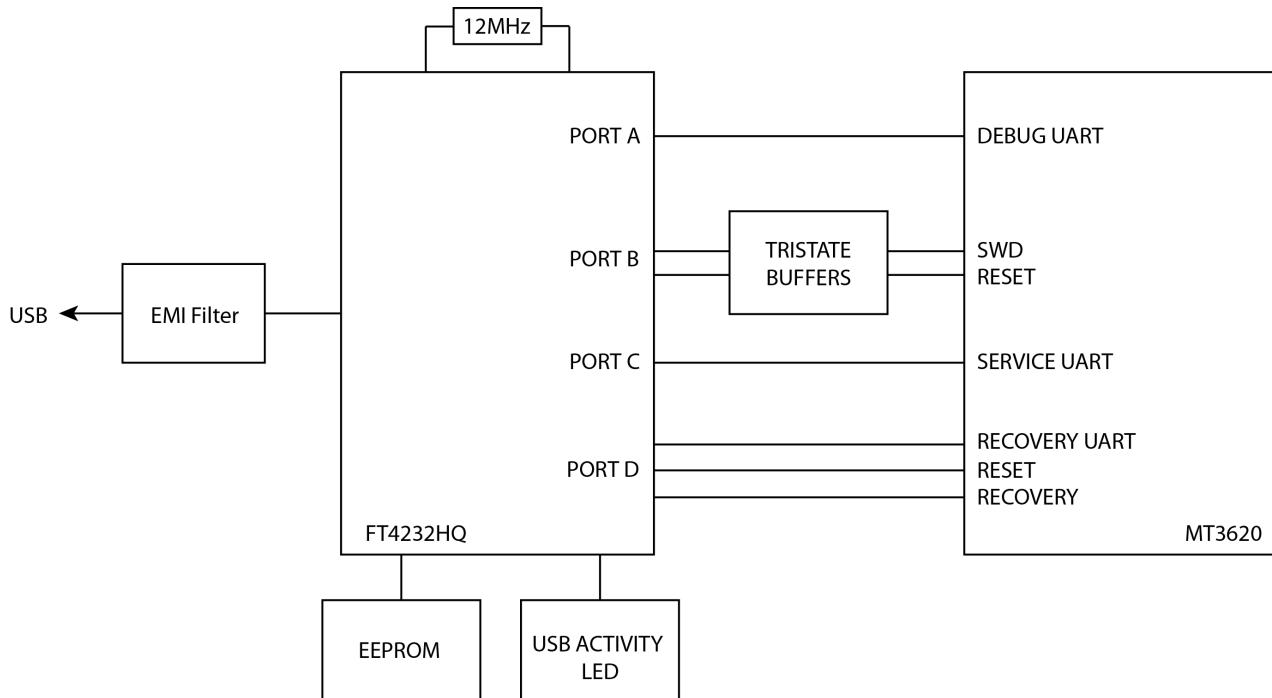
The MT3620 reference development board (RDB) incorporates a USB-to-UART interface chip that exposes these interfaces to a PC in a way that allows the Azure Sphere PC software tools to recognize and interact with them. The Azure Sphere tools include support for loading an application over USB by using 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, too. 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.

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

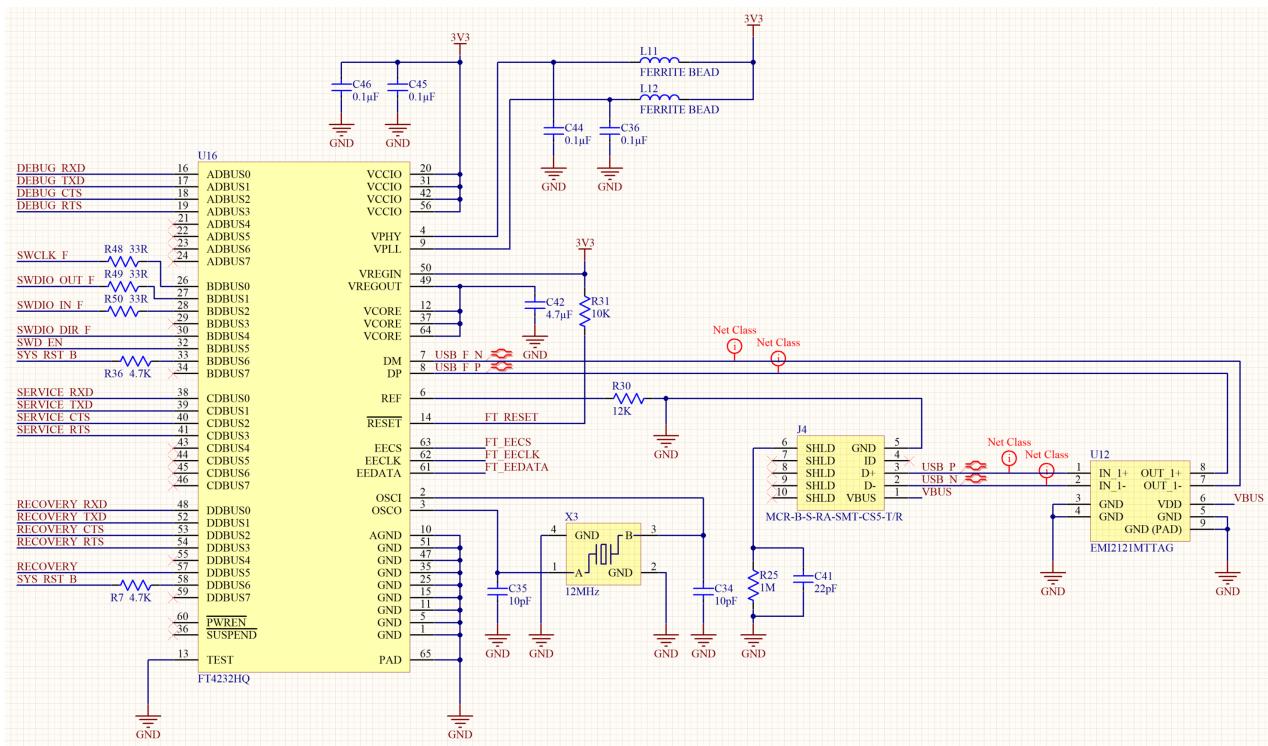
To ensure compatibility with the PC tools, each of the four FTDI ports must be connected to the MT3620 as described in the following table:

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 (optional, for use as advised by Microsoft)	Port-B	BDBUS0 (26)	SWCLK	See SWD interface 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:

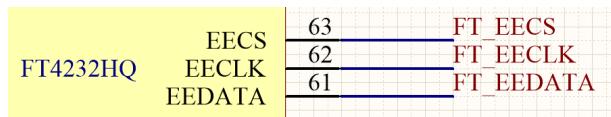
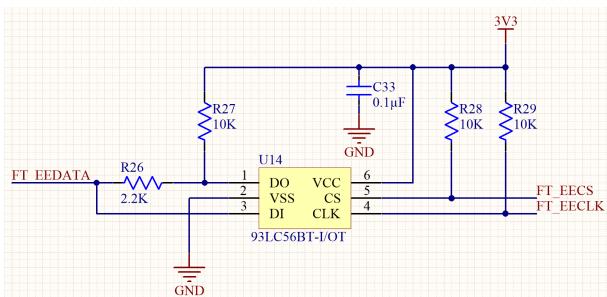
- 93LC46BT-I/OT
- 93LC56BT-I/OT
- 93LC66BT-I/OT

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:

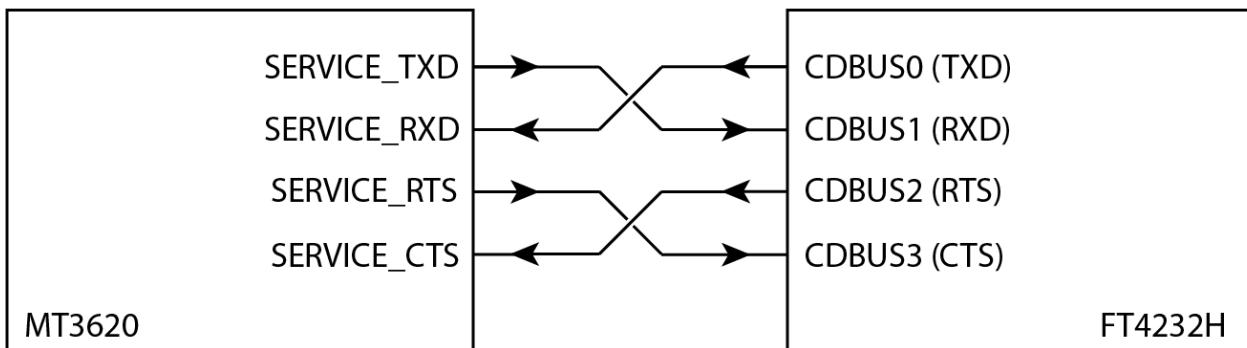
EEPROM CIRCUIT

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:



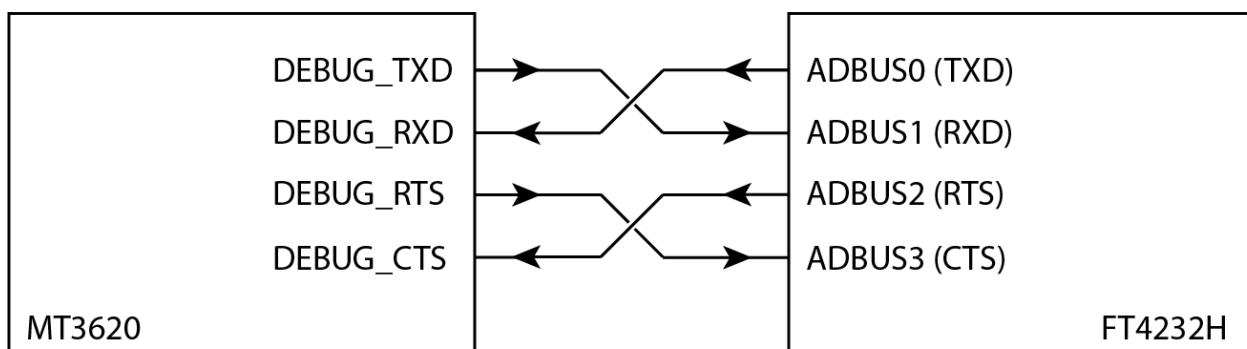
Debug UART interface (optional)

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:



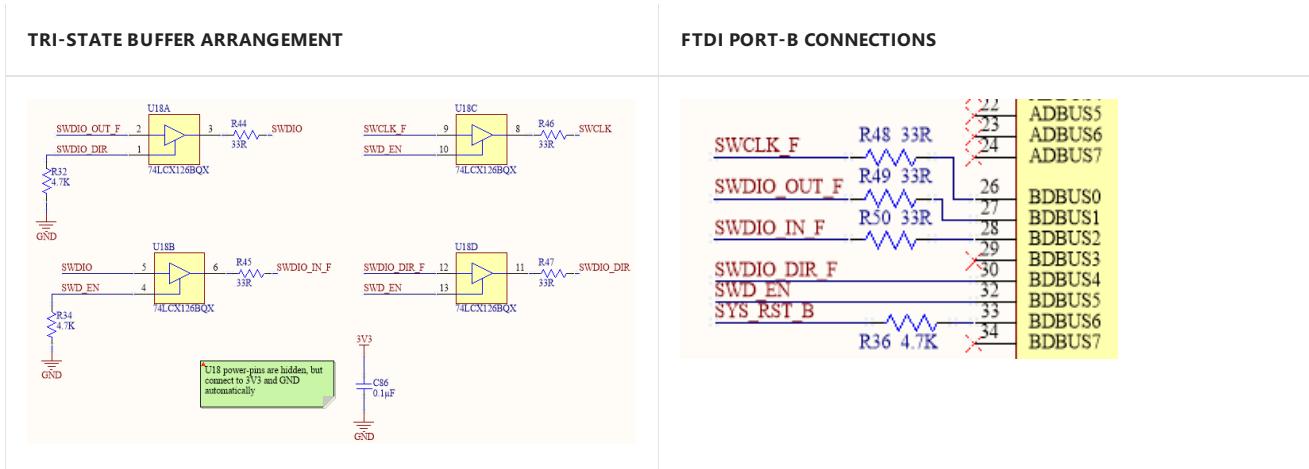
SWD interface (optional)

NOTE

Not all MT3620-based modules expose the SWD interface.

The SWD interface provides additional debugging capabilities and is reserved for future use. If the interface is available, we recommend that you expose it, at least in development and lab environments, to enable easier debugging of any issues that arise. Note that the Azure Sphere SDK does not use SWD for debugging; it operates over the Service UART described earlier.

Although the FTDI chip is designed to provide a bridge between UARTs and USB, it is also possible to add additional circuitry based on a quad tristate buffer to enable 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.

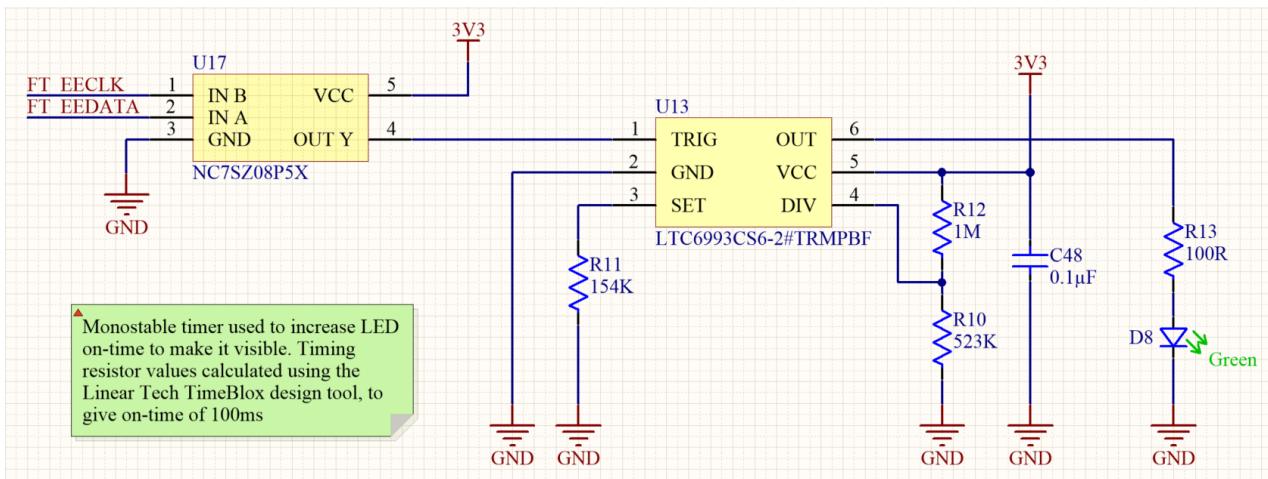


USB activity LED (optional)

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

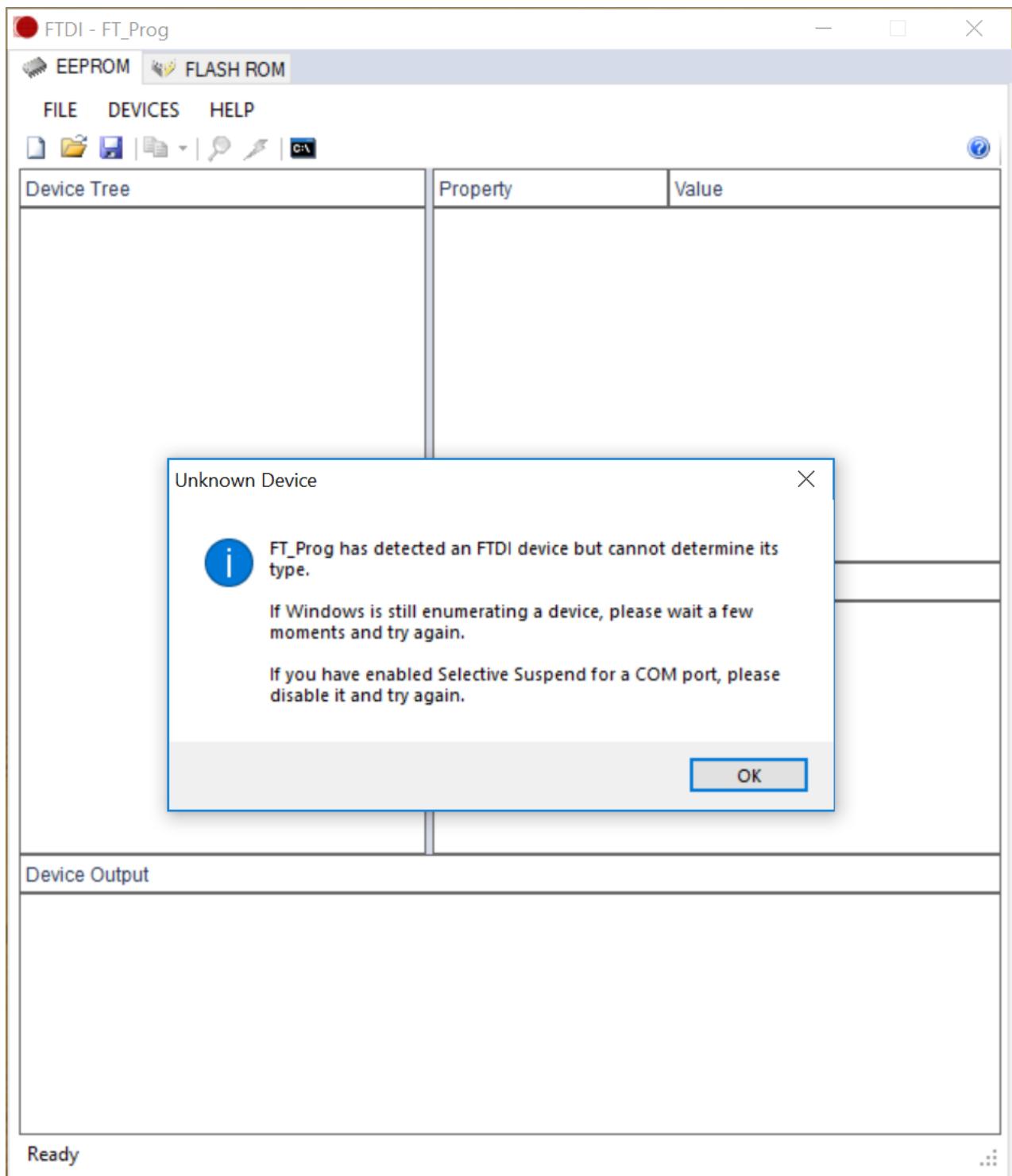
To help in programming the EEPROM, FTDI provides a free software tool called FT_PROG. The tool is available as both a Windows GUI application and as a command-line tool; both options are installed at the same time from the same package. Download the tool from the [FTDI website](#) and install it in the default location.

Using the FT_PROG GUI application

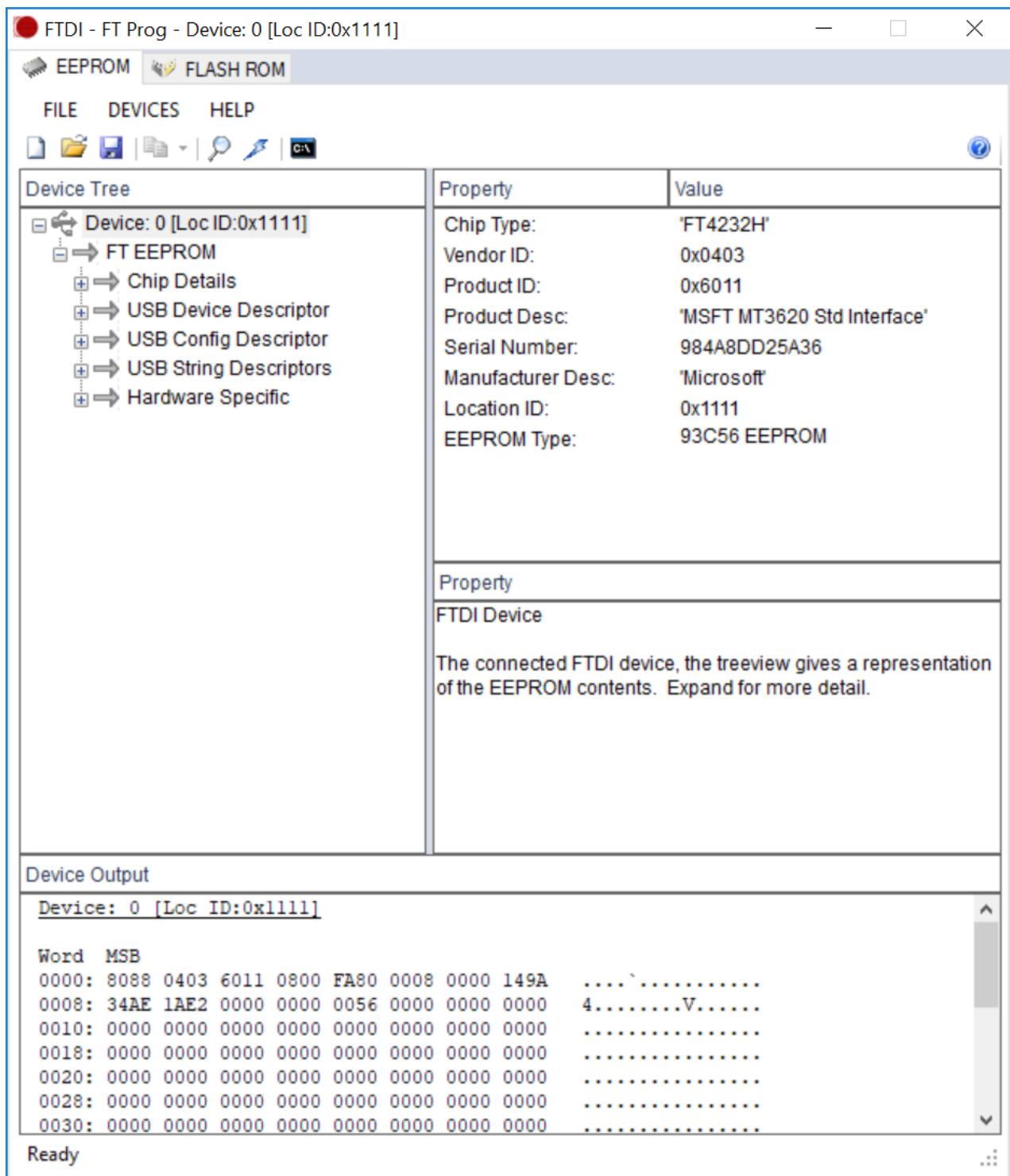
The Windows GUI version of the application is useful for reading and checking the state of the EEPROM information. You can also use it to change the information; however, we recommend the use of the command-line version of the tool to program the device, because you can program the complete configuration 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. [Standard configuration file](#), later in this document, shows the standard configuration file. 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.

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_FTDI_Interface_4.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
```

Standard configuration file

MT3620_FTDI_Interface_4.xml is the standard configuration file for the EEPROM. To program the EEPROM, you must use this file as it appears here, without modification.

```
<?xml version="1.0" encoding="utf-16"?>
<FT_EEPROM>
  <Chip_Details>
    <Type>FT4232H</Type>
  </Chip_Details>
  <USB_Device_Descriptor>
    <VID_PID>0</VID_PID>
    <idVendor>0403</idVendor>
    <idProduct>6011</idProduct>
    <bcdUSB>USB 2.0</bcdUSB>
  </USB_Device_Descriptor>
  <USB_Config_Descriptor>
    <bmAttributes>
      <RemoteWakeupEnabled>false</RemoteWakeupEnabled>
      <SelfPowered>false</SelfPowered>
      <BusPowered>true</BusPowered>
    </bmAttributes>
    <IOpullDown>false</IOpullDown>
    <MaxPower>500</MaxPower>
  </USB_Config_Descriptor>
  <USB_String_Descriptors>
    <Manufacturer>Microsoft</Manufacturer>
    <Product_Description>MT3620 Standard Interface</Product_Description>
    <SerialNumber_Enabled>true</SerialNumber_Enabled>
    <SerialNumber>984A8DD25A36</SerialNumber>
    <SerialNumberPrefix />
    <SerialNumber_AutoGenerate>false</SerialNumber_AutoGenerate>
  </USB_String_Descriptors>
  <Hardware_Specific>
    <TPRDRV>0</TPRDRV>
    <Port_A>
      <VCP>true</VCP>
      <D2XX>false</D2XX>
      <RI_RS485>false</RI_RS485>
    </Port_A>
    <Port_B>
```

```
<Port_A>
  <VCP>false</VCP>
  <D2XX>true</D2XX>
  <RI_RS485>false</RI_RS485>
</Port_A>
<Port_B>
  <VCP>true</VCP>
  <D2XX>false</D2XX>
  <RI_RS485>false</RI_RS485>
</Port_B>
<Port_C>
  <VCP>true</VCP>
  <D2XX>false</D2XX>
  <RI_RS485>false</RI_RS485>
</Port_C>
<Port_D>
  <VCP>true</VCP>
  <D2XX>false</D2XX>
  <RI_RS485>false</RI_RS485>
</Port_D>
<IO_Pins>
  <Group_A>
    <SlowSlew>false</SlowSlew>
    <Schmitt>false</Schmitt>
    <Drive>4mA</Drive>
  </Group_A>
  <Group_B>
    <SlowSlew>false</SlowSlew>
    <Schmitt>false</Schmitt>
    <Drive>4mA</Drive>
  </Group_B>
  <Group_C>
    <SlowSlew>false</SlowSlew>
    <Schmitt>false</Schmitt>
    <Drive>4mA</Drive>
  </Group_C>
  <Group_D>
    <SlowSlew>false</SlowSlew>
    <Schmitt>false</Schmitt>
    <Drive>4mA</Drive>
  </Group_D>
</IO_Pins>
</Hardware_Specific>
</FT_EEPROM>
```

RF test tools

12/12/2018 • 12 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

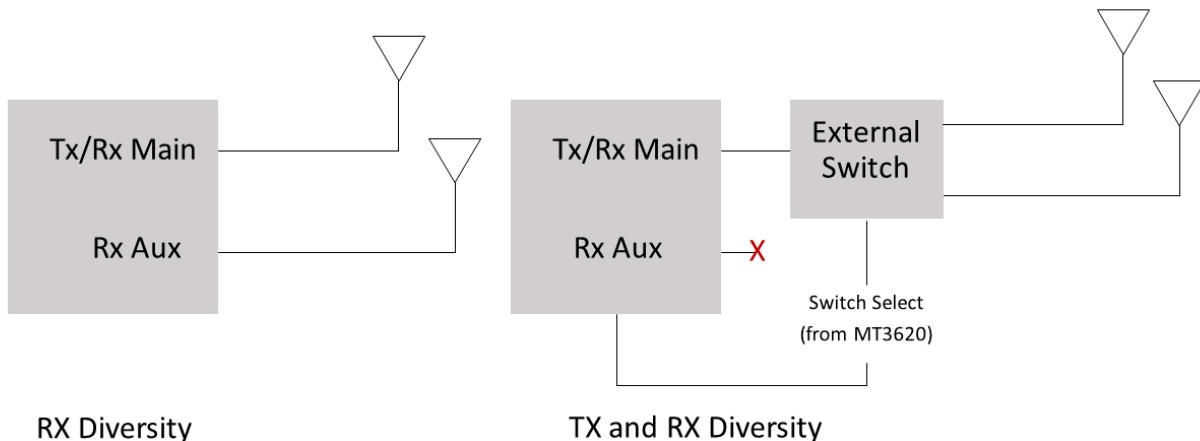
The MT3620 stores radio configuration and calibration data in e-fuses, which can be programmed a limited number of times. This data includes the radio bands (such as 2.4GHz or 5GHz) that the chip should support, adjustments to transmit power, and the antenna configuration on the device. For detailed 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
antenna {aux main}	Selects the auxiliary or main antenna.
channel <i>number</i>	Selects a channel.
config read {macaddress data}	Gets device MAC address and buffer bin data.
config write {macaddress data}	Sets device MAC address and buffer bin data.
config save	Saves changes to MAC address or buffer bin data to permanent e-fuses.
exit	Exits from the program.
help <i>command-name</i>	Displays help on a command.

COMMAND (ABBREVIATION) OPTIONS	DESCRIPTION
receive (rx) {start stop stats}	Starts or stops receiving, or displays statistics about received packets.
settings	Displays current radio settings.
showchannel (sc)	Lists the channels that the device supports.
transmit (tx) {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
check (c)	Validates MT3620 device configuration data
help (?)	Shows help information
show (s)	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.

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.

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

2/14/2019 • 9 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.

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. On the factory floor, you can read the device ID from a chip that is attached to a PC by using the following **azsphere** command:

```
azsphere device show-attached
```

You will need the device IDs during [cloud configuration](#) to set up device groups and deployments.

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 [device recovery](#) over the programming and debugging interface described earlier.

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, MT3620 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 image add** 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
```

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>
```

Replace <file-path> with the name and path to the downloaded image file.

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>
```

NOTE

The command to remove an application from a device uses the component id, rather than the image id. You can find the component id in the output of the **azsphere component image add** command or by using the **azsphere image show** command.

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.

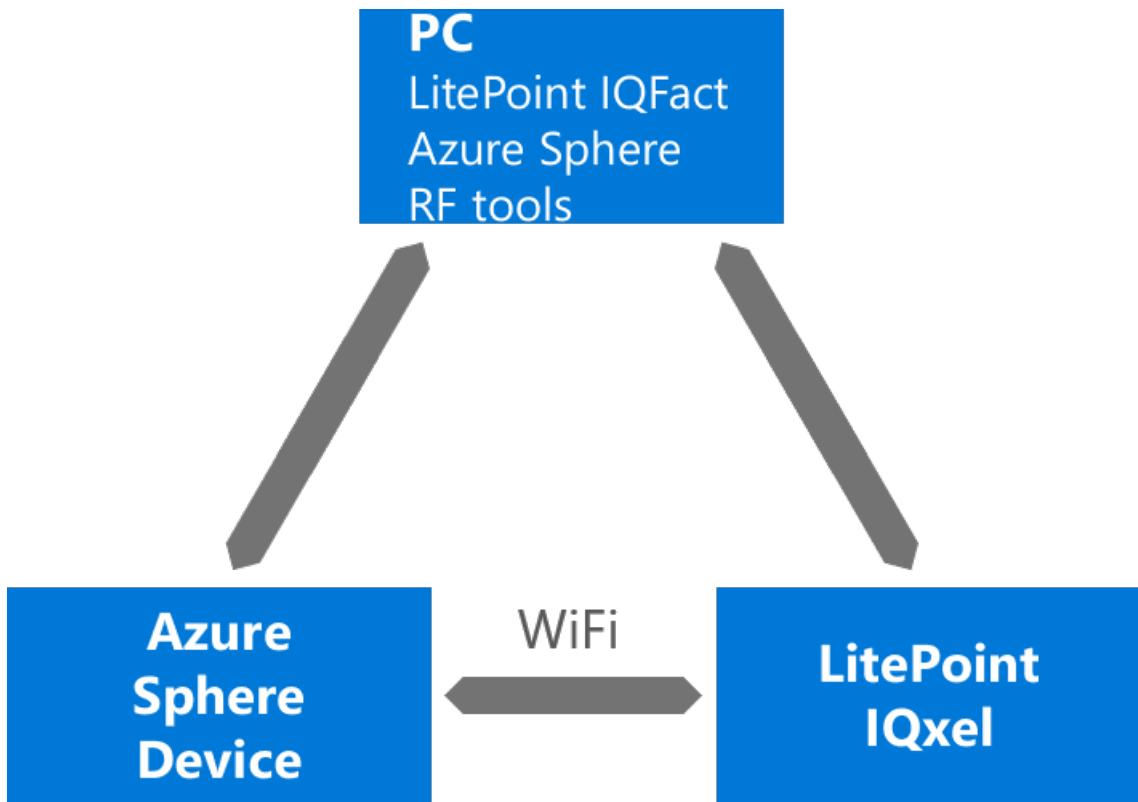
Some testing processes require communication with the MT3620 chip that is being tested: to report errors, log data, or sequence tests. If your testing process requires communication, you can use the peripheral UARTs on the MT3620 (ISU0, ISU1, ISU2, or ISU3). Connect these UARTs to your factory PC or 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
```

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 an MT3620 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. This file is installed with the Factory Tools package in the same folder as the deviceready.py script. The default value is mt3620an.json.
- 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.
```

Samples and reference solutions

2/14/2019 • 2 minutes to read

You can find the following [Azure Sphere samples and reference solutions](#) on GitHub.

SAMPLE NAME	DESCRIPTION
AzureIoT	Shows how to use Azure Sphere with Azure IoT Central or an Azure IoT Hub
CMakeSample	Shows how to use CMake to build Azure Sphere applications
CurlEasyHttps	Shows how to use the cURL "easy" API with Azure Sphere over a secure HTTPS connection
CurlMultiHttps	Shows how to use the cURL "multi" API with Azure Sphere over a secure HTTPS connection
ExternalMcuUpdateNrf52	Shows how you might deploy an update to an external MCU device using Azure Sphere
LSM6DS3_I2C	Shows how to use the Azure Sphere inter-integrated circuit (*I2C) API to display data from an accelerometer connected via I2C
LSM6DS3_SPI	Shows how to use the Azure Sphere serial peripheral interface (SPI) API to display data from an accelerometer connected via SPI
MutableStorage	Shows how to use on-device storage in an Azure Sphere application
PrivateEthernet	Shows how you can connect Azure Sphere to a private Ethernet network
SystemTime	Shows how to manage the system time and to use the hardware real time clock(RTC)
WifiConfigViaBle	Shows how you might complete Wi-Fi configuration of an Azure Sphere-based device through Bluetooth Low Energy (BLE) using a companion app on a mobile device
WifiSetupAndDeviceControlViaBle	Shows how you might complete Wi-Fi 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 19.02

2/14/2019 • 8 minutes to read

The Azure Sphere 19.02 preview release includes the changes, new features, and known issues described in this article.

How to update to this release

- If your Azure Sphere device is already running the 18.11 OS release (or later) and is connected to the internet, it should receive the 19.02 release by over-the-air (OTA) update.
- 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.
- 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 19.02 release.

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 this 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](#)
- [Determine application memory usage](#)

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 ID 82bacf85-990d-4023-91c5-c6694a9fa5b4) will start delivering an evaluation version of the Azure Sphere OS approximately two weeks before the next release. Currently, it delivers the same OS version as the Retail Azure Sphere OS 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

1/9/2019 • 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.

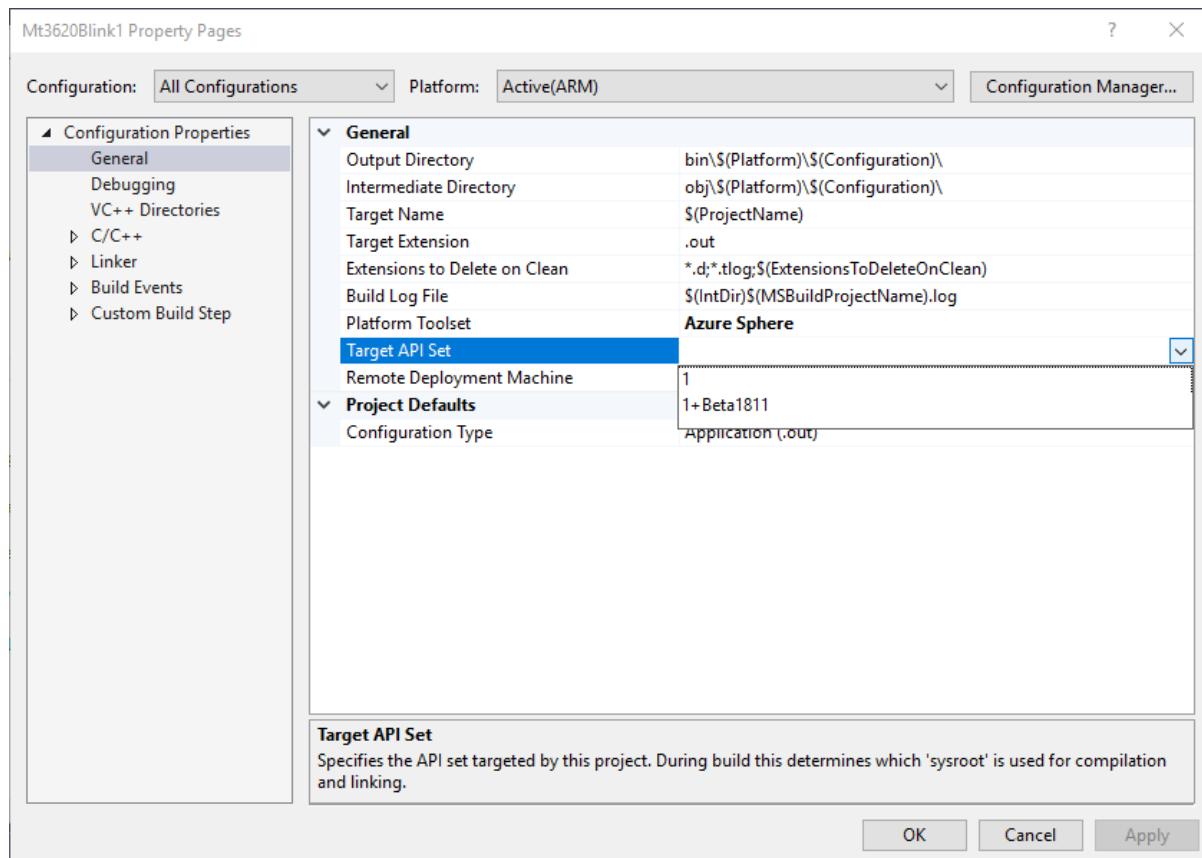
To enable you to develop applications that use either the production APIs or the production and Beta APIs, this Azure Sphere release adds the **Target API Set** property to the **Project Properties** in Visual Studio. The Target API Set value 1 selects only production APIs; the value 1+Beta1811 selects 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](#) 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

9/28/2018 • 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: Operational notifications and updates from the Azure Sphere Engineering team. [View Notifications](#)

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](#)