SystemReady SR and ES Integration Guide

2.0

Non-Confidential

Issue 01

102677_0200_01_en

SystemReady SR and ES Integration Guide

Release information

Document history

Issue	Date	Confidentiality	Change
0100-01	1 January 2020	Non-Confidential	Initial release of SystemReady ES Integration Guide
0200-01	14 April 2023	Non-Confidential	Added SR requirements to create the SystemReady SR/ES Integration Guide

Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Feedback

Arm welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

This document includes language that can be offensive. We will replace this language in a future issue of this document.

To report offensive language in this document, email terms@arm.com.

Contents

1. Overview

This guide tells you how to integrate SystemReady SR/ES systems, how to develop and build the firmware, and how to run SystemReady SR/ES certification tests.

In this guide, you will learn:

- How to set up a Raspberry Pi 4 for SystemReady tests
- How to set up a Neoverse N2 reference design (RD-N2) FVP for SystemReady tests.
- How to use test suites
- About Advanced Configuration and Power Interface (ACPI) power management and System Management BIOS (SMBIOS) integration

---

Note

The only integration difference between SystemReady SR and ES is that SBSA only applies to SystemReady SR, which is why this guide covers both SystemReady SR and ES systems.

Before you begin

This guide assumes you are familiar with the following technologies and frameworks:

- UEFI
- EDK2 firmware development environment
- ACPI, ASL, and AML
- SMBIOS

This guide is aimed at the following audiences:

- IHVs and OEMs who develop SystemReady SR/ES complaint platforms
- UEFI developers who implement ACPI and SMBIOS support for SystemReady SR/ES compliant platforms
- Operating system developers who adapt their operating systems to run on SystemReady SR/ES compliant platforms

2. Set up the Raspberry Pi 4

In this section, we use a Raspberry Pi 4 to demonstrate how to build a SystemReady ES compliant platform.

To set up the Raspberry Pi, you will need the following hardware:

Power

A powered USB hub to avoid overloading the standard Raspberry Pi power supply.

Network controller (NIC)

UEFI supports the Raspberry Pi NIC such as for PXE booting, however the NIC driver is missing from many OS distributions. Use a USB NIC, such as a Realtek RTL8153 based device. For this guide, we tested the Raspberry Pi with RTL8153 NIC.

Storage

A micro SD card and a USB storage device. The micro SD holds the UEFI firmware and any FAT16 or FAT32 capable drive will work.

The USB Storage device is used as the main disk for the operating system. Connect it to the USB port of the Raspberry Pi. We recommend the USB 3.0 blue ports for better performance.

Check your OS for minimum install size, for example, 64 to 128GB as a starting point. Thumb drivers and drive enclosures can be used. We recommend a USAP enabled external drive. A second 8GB or larger thumb drive is recommended for the OS installer.

Interfacing

Use the Raspberry Pi video output with a keyboard and mouse or use a serial connection. Both types of connection can be set up at the same time.

Keyboard and mouse

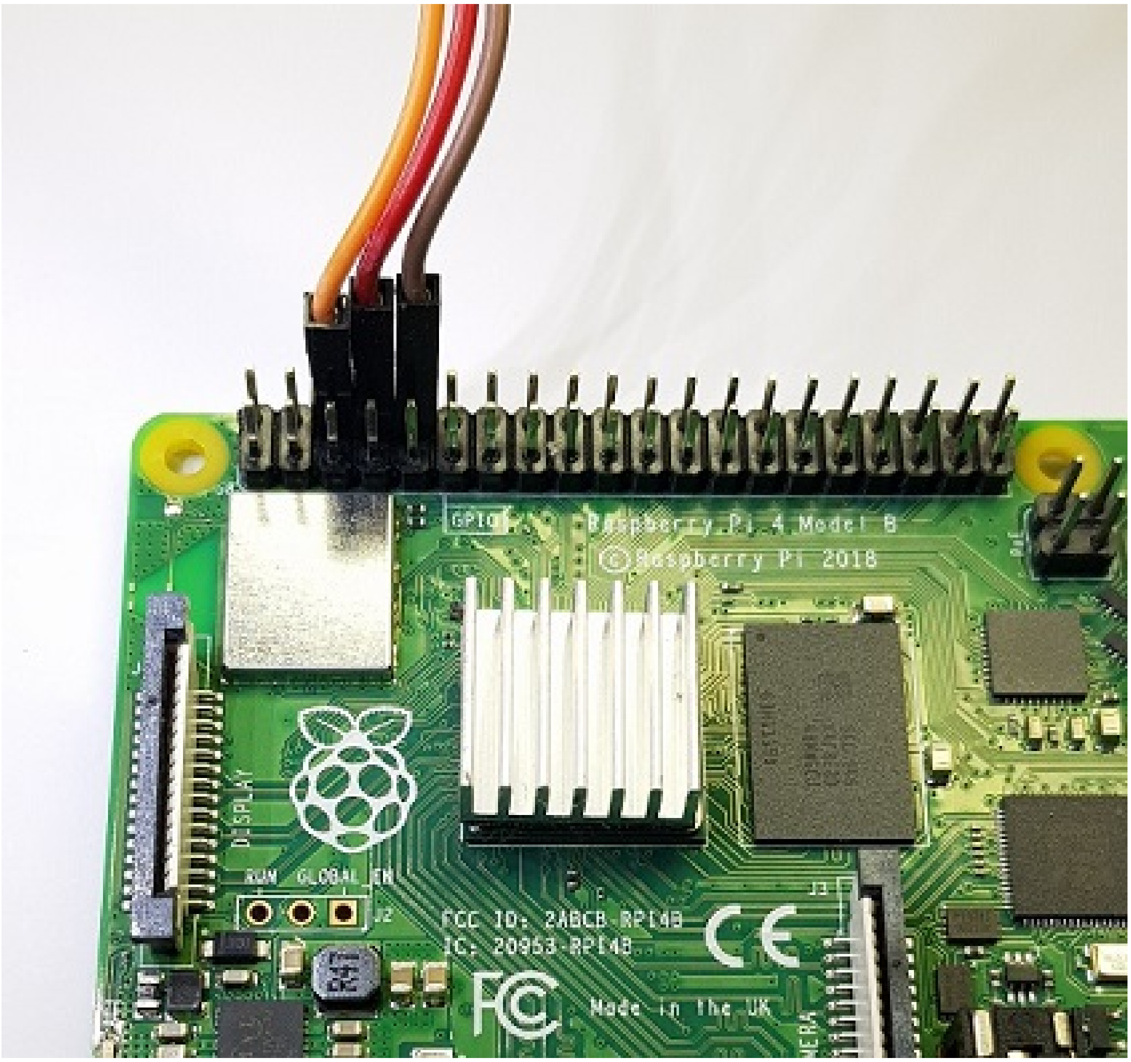
Use an HDMI micro to HDMI cable and an HDMI display to output the video. USB mice and keyboards with generic drivers will work.

Serial adapter

For this guide, use a generic TTL serial adapter that utilizes separate cables. You will need to use three of the wires.

The following image shows how to connect the serial adapter to your Raspberry Pi:

Figure 2-1: Raspberry Pi serial adapter connections



The wires are attached as shown in the table below:

Description	TX	RX	GRND
Color	Red	Brown	Orange
Header pin	8	10	6
GPIO	GPIO14	GPIO15	-

Finally, connect the serial cable USB connector to your PC.

2.1 Set up the terminal

If you are using Windows, you will need a terminal emulator such as PuTTY.

The following table shows you how to set up your connection:

Variable	Value
Baud rate	115200
Data bits	8
Parity	None
Stop bits	1

On the **Session** configuration panel in PuTTY, select **Serial** from the **Connection type** options. Use the **Serial line** and **Speed** options to specify which serial line to use and the Baud rate to use to transfer data. For more information on serial connection with PuTTY, see Connecting to a local serial line.

If you are using Linux or a Mac, use terminal emulators such as minicom or screen to connect to the TTL serial connection. If there are no serial devices connected to your computer, your serial connector will be `/dev/ttyUSB0`. If you have more than one serial device, use a tool such as `dmseg` to check `ttyUSB<num>`.

To connect using `screen`, enter the following command:

```
$ screen /dev/ttyUSB0 115200
```

To connect using minicom, enter the following command:

```
$ minicom -D /dev/ttyUSB0
```

For more information and troubleshooting, see Using a console cable with Raspberry Pi.

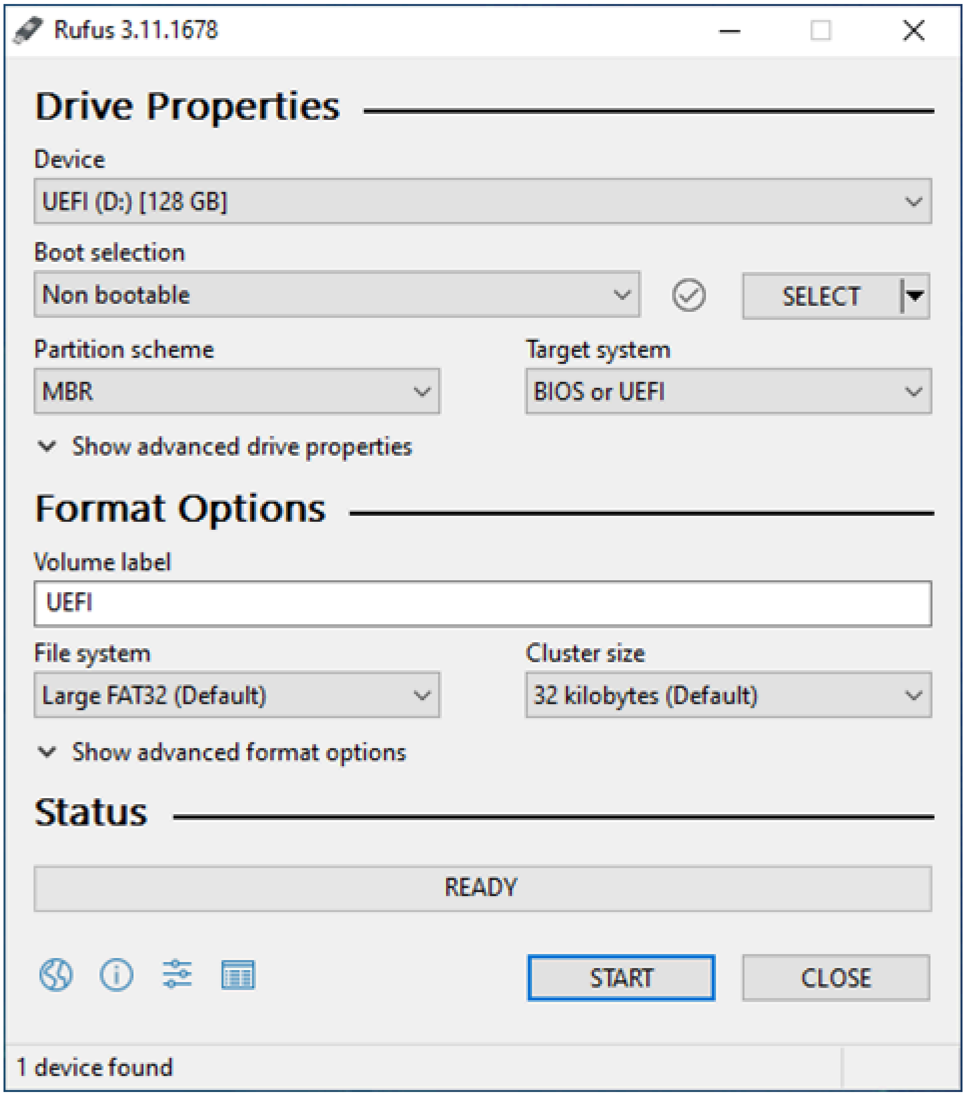
2.2 Format the SD drive

For Raspberry Pi 4, the SD drive can be formatted in FAT32 for updating the EEPROM and storing the UEFI firmware.

To format the SD drive on Windows, we use Rufus and the following procedure:

1. In Rufus, select your device then select **Non bootable** from the **Boot selection** menu. Ensure the file system type is Large FAT16 or Large FAT32, as shown in the following screenshot:

Figure 2-2: Rufus format options

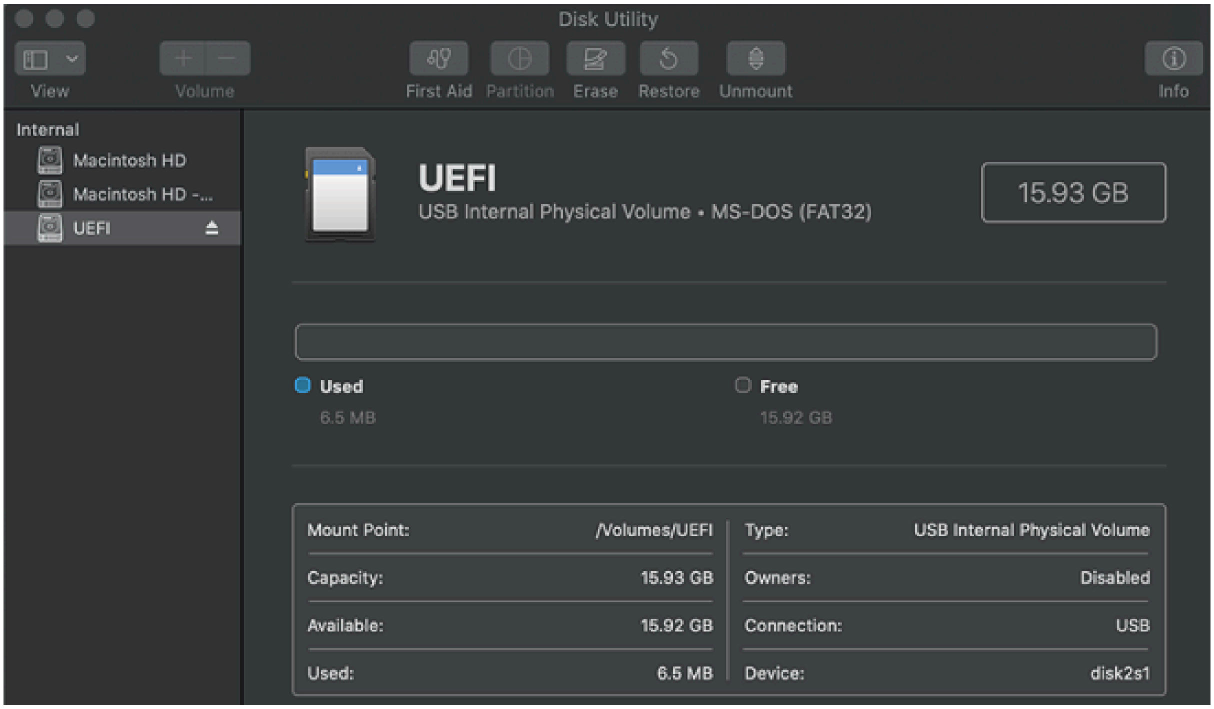


2. Click **Show advanced format options** and disable **Create extended label and icon files**. This option is not needed for this guide.
3. Click **START**.

To format the drive on Mac OS:

1. Open Disk Utility and select your SD card in the list of drives. An example is shown in the following screenshot:

Figure 2-3: Disk Utility window

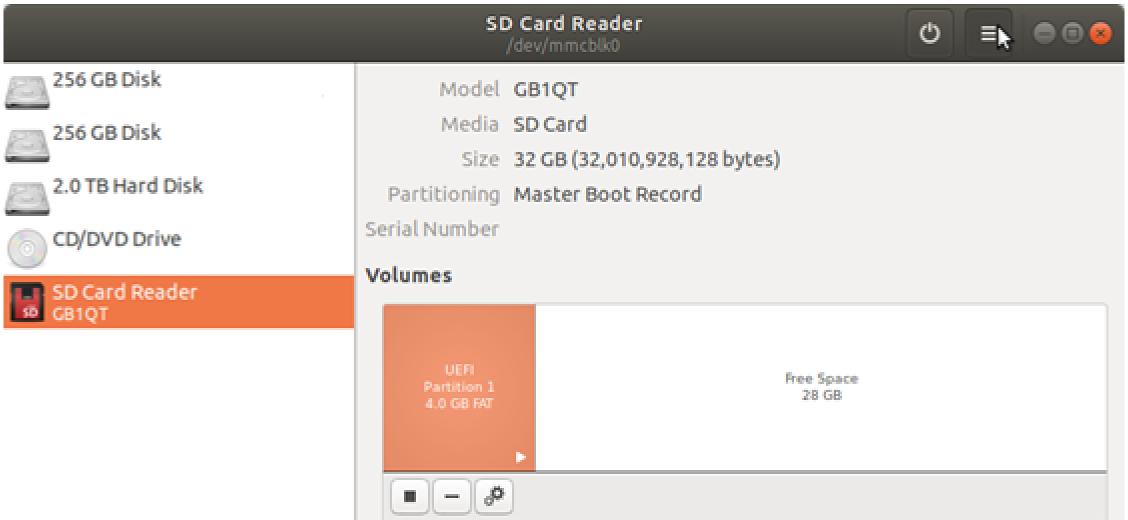


2. Click **Erase** to format the drive.
3. In the format list, select MS-DOS (FAT).

To format the drive on Linux:

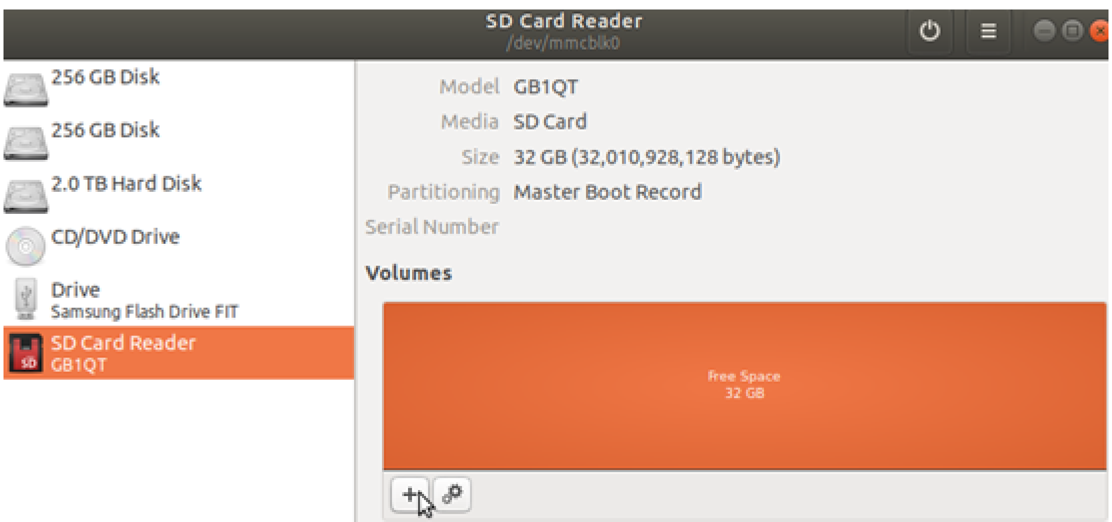
1. Use either graphical or command-line instructions. For graphical instructions, open Disks and select your SD card.
2. Click the bars at the top of the window, as shown in the following screenshot:

Figure 2-4: Disk format option



3. Select **Format Disk**, then select **Compatible with all systems and devices (MBR/DOS)**.
4. Click **Format**. A blank formatted disk is created.
5. Click **+** to add a partition, as shown in the following screenshot:

Figure 2-5: Add partition



6. Select a **Partition Size**. For this guide, the firmware image is under 10MB, so any partition size can be used. Click **Next**.
7. In **Type**, select **For use with all systems and devices (FAT)**. Click **Create**.

2.3 Update the EEPROM

Ensure the Raspberry Pi 4 is running the latest firmware on the EEPROM. Download the latest version of `rpi-eeprom` from RPi eeprom github and use this tool to update the boot EEPROM.

To update the EEPROM:

1. Unzip the contents of `rpi-boot-eeprom-recovery` to a blank FAT formatted SD-SDCARD.
2. Power off the Raspberry Pi 4.
3. Insert the SD card.
4. Power on the Raspberry Pi 4 and wait 10 seconds.

The green LED light will blink rapidly to indicate success, otherwise an error pattern is displayed.

If an HDMI display is attached to the Raspberry Pi 4, the screen will display green for success or red if failure a failure occurs.

2.4 Install UEFI

The latest UEFI binaries and installation guide are on PFTF Github.

To install UEFI:

1. Download the latest archive from Releases.
2. Create an SD card or a USB drive with at least one partition. This can be a regular partition or an ESP. Format the partition to FAT16 or FAT32.

> **Note**
> To boot from USB or ESP, you need the latest version of the EEPROM. If you are using the latest UEFI firmware and you cannot boot from USB or ESP, see Update the EEPROM.

3. Extract all the files from the archive to the partition you created. Do not change the names of the extracted files and directories.

To run UEFI:

1. Insert the SD card or connect the USB drive and power up your Raspberry Pi 4. A multicolored screen is displayed showing the embedded bootloader reading the data. The Raspberry Pi 4 logo is displayed when the UEFI firmware is ready.
2. Press Esc to enter the firmware setup, F1 to launch the UEFI Shell, or wait for the UEFI boot option to boot Raspberry Pi 4.

You can build UEFI firmware from source. The following steps are for Ubuntu Linux 18.04.1 on x86_64 host PC using cross compilation.

To build UEFI firmware:

1. Create a workspace directory with the following commands:

```
$ mkdir RPi4
$ export WORKSPACE=$(pwd)/RPi4
```

2. Clone the pftf/RPi4 repository:

```
$ git clone http://github.com/pftf/RPi4.git
$ git submodule update -init
```

3. Initialize submodules for both the `edk2` and `edk2-platform` repositories using the commands shown:

```
$ cd edk2
$ git submodule update -init
$ cd ../edk2-platforms
$ git submodule update -init
$ cd ..
```

4. Copy `0001-MdeModulePkg-UefiBootManagerLib-Signal-ReadyToBoot-o.patch` to the `edk2` folder and run the following command:

```
$ patch -p3 < 0001-MdeModulePkg-UefiBootManagerLib-Signal-ReadyToBoot-o.patch
```

5. Install a toolchain for cross compilation using the following command:

```
$ sudo apt-get install gcc-aarch64-linux-gnu
```

6. Follow the instructions on Building EDKII UEFI firmware for Arm Platforms to build a binary. An example of the build command for RPi4 platform follows:

```
$ GCC5_AARCH64_PREFIX=aarch64-linux-gnu-
$ build -n 8 -a AARCH64 -t GCC5 -p Platform/RaspberryPi/RPi4/RPi4.dsc
```

   The resulting binary `RPI_EFI.fd` can be found in the `RPi4/Build/<BUILD_TARGET>/FV` folder.

7. Follow the Booting the firmware section in Raspberry Pi 4 Platform to prepare a bootable SD card.

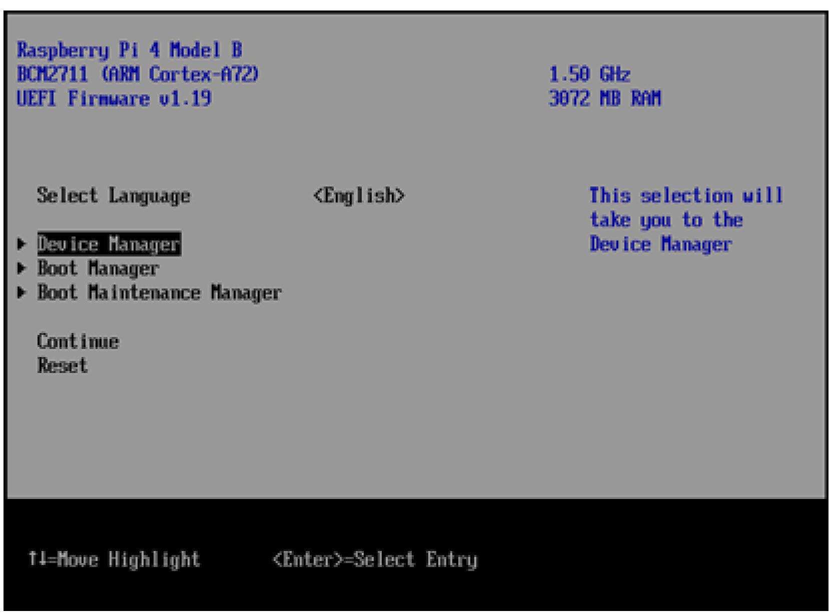
2.5 Configure UEFI

To boot into the UEFI shell, press F1 during the boot process, as shown in the following screenshot:

Figure 2-6: UEFI boot screen

To boot to the UEFI menu, press Esc during the boot process. The following UEFI menu is displayed:

Figure 2-7: UEFI menu



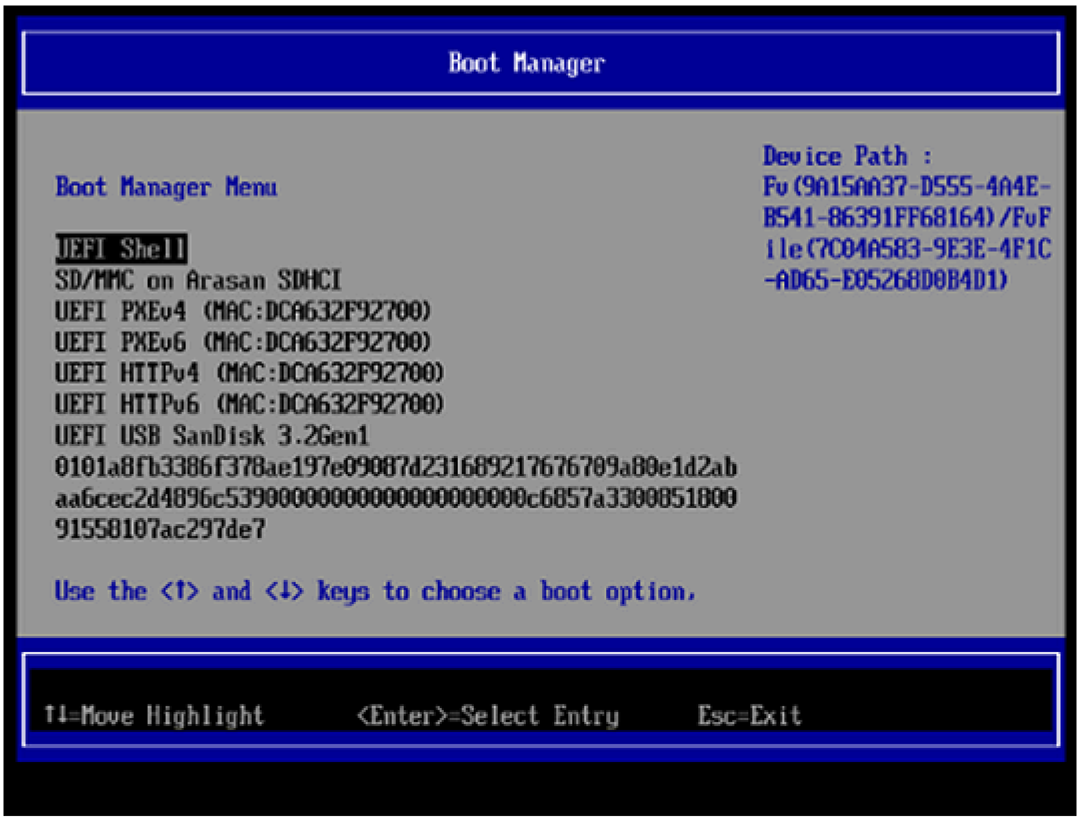
In this menu, you can change device settings and manually boot the device using Boot Manager.

2.6 Troubleshooting UEFI

To boot to the UEFI menu:

1. Press Esc to interrupt the boot process.
2. In the UEFI menu, navigate to the Boot Manager then select UEFI Shell. The Raspberry Pi 4 boots to the UEFI Shell. The UEFI Shell option is shown in the following screenshot:

Figure 2-8: UEFI Shell



3. Use the `map` command to see if a storage device is mounted. In the following example screenshot, an SD card is mounted as FS0:

Figure 2-9: SD example

```
Shell> map
Mapping table
      FS2: Alias(s):HD1b::BLK4:
          VenHw(100C2CFA-B586-4198-9B4C-1683D195B1DA)/HD(1,MBR,0x6F1D7A2C,0x800,
0xECD000)
      FS0: Alias(s):HD0c0b::BLK1:
          PcieRoot(0x0)/Pci(0x0,0x0)/Pci(0x0,0x0)/USB(0x2,0x0)/HD(1,GPT,162B535C
-7654-4FFB-BAA0-1D9E3C026035,0x800,0x40000)
      FS1: Alias(s):HD0c0c::BLK2:
          PcieRoot(0x0)/Pci(0x0,0x0)/Pci(0x0,0x0)/USB(0x2,0x0)/HD(2,GPT,54385270
-4B50-4D32-8B87-AF26785E21B7,0x40800,0x46800)
     BLK3: Alias(s):
          VenHw(100C2CFA-B586-4198-9B4C-1683D195B1DA)
     BLK0: Alias(s):
          PcieRoot(0x0)/Pci(0x0,0x0)/Pci(0x0,0x0)/USB(0x2,0x0)
Shell> _
```

4. Change the directory to FS0 by typing `FS0` at the command prompt.

The following UEFI Shell commands are also helpful for debugging:

Command	Description
`pci`	Show PCIe devices or PCIe function configuration space information
`drivers`	Show a list of drivers
`devices`	Show a list of devices managed by EFI drivers
`devtree`	Show a tree of devices
`dh -d -v > dh_d_v.txt`	Save a dump of all UEFI Driver Model-related handles to dh_d_v.txt
`memmap`	Save the memory map to memmap.txt
`smbiosview`	Show SMBIOS information
`acpiview -l`	Show a list of ACPI tables
`acpiview -r 2`	Validate that all ACPI tables required by SBBR 1.2 are installed.
`acpiview -s DSDT -d`	Generate a binary file of DSDT ACPI table.
`dmpstore -all > dmpstore.txt`	Dump all UEFI variables to dmpstore.txt

Refer to the UEFI Shell Specification for more details. The Shell commands section provides a list of shell commands, descriptions, and examples.

2.7 Set UEFI variables

The Raspberry Pi 4 UEFI configuration settings can be viewed and changed using the UI configuration menu and UEFI shell. To configure the Raspberry Pi 4 using the UEFI Shell, use `setvar` to read and write the UEFI variables for the GUID CD7CC258-31DB-22E6-9F22-63B0B8EED6B5.

To read a setting, use the following command:

```
setvar <NAME> -guid CD7CC258-31DB-22E6-9F22-63B0B8EED6B5
```

To write a setting, use the following command:

```
setvar <NAME> -guid CD7CC258-31DB-22E6-9F22-63B0B8EED6B5 -bs -rt -nv =<VALUE>
```

For string-type settings such as Asset Tag, use the following command:

```
setvar <NAME> -guid CD7CC258-31DB-22E6-9F22-63B0B8EED6B5 -bs -rt -nv =L"<VALUE>"
 =0x0000
```

The following commands are examples of reading and modifying UEFI variables:

Read the System Table Selection setting

```
Shell> setvar SystemTableMode -guid CD7CC258-31DB-22E6-9F22-63B0B8EED6B5
```

Change the System Table Selection setting to Devicetree

```
Shell> setvar SystemTableMode -guid CD7CC258-31DB-22E6-9F22-63B0B8EED6B5 -bs -rt
-nv =0x00000002
```

Read the Limit RAM to 3 GB setting:

```
Shell> setvar RamLimitTo3GB -guid CD7CC258-31DB-22E6-9F22-63B0B8EED6B5
```

Change the Limit RAM to 3 GB setting to Disabled:

```
Shell> setvar RamLimitTo3GB -guid CD7CC258-31DB-22E6-9F22-63B0B8EED6B5 -bs -rt -
nv =0x00000000
```

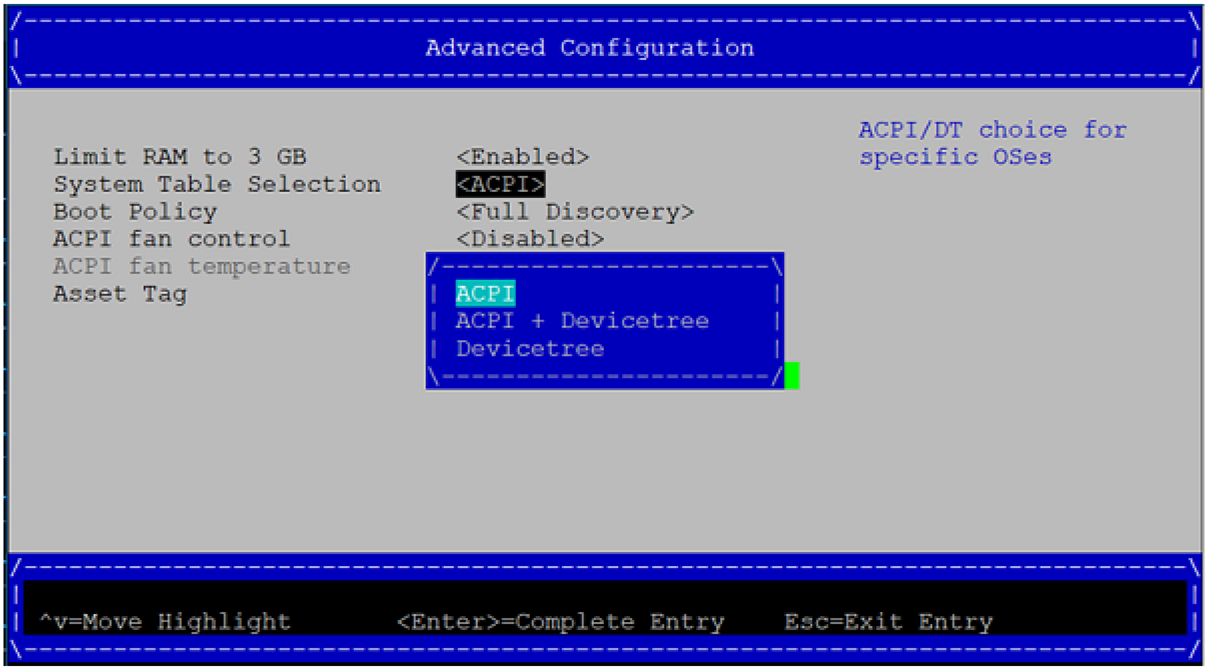
Change the Asset Tag to the string ASSET-TAG-123:

```
Shell> setvar AssetTag -guid CD7CC258-31DB-22E6-9F22-63B0B8EED6B5 -bs -rt -nv
=L"ASSET-TAG-123" =0x0000
```

2.8 Set the system table selection

In the **Advanced Configuration** menu, select ACPI as shown in the following screenshot:

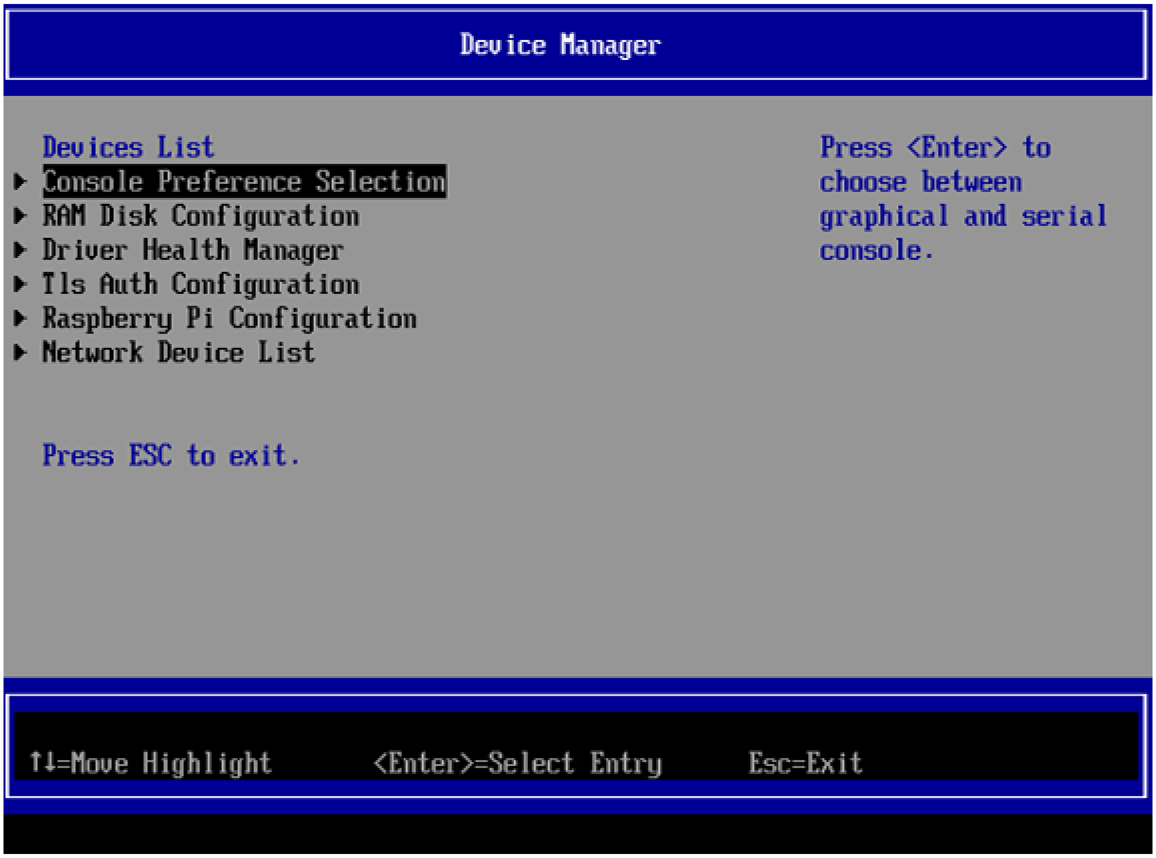
Figure 2-10: ACPI option



2.9 Set the console preference

Linux uses the `/chosen/stdout-path` DT property or the SPCR ACPI table to indicate that the primary console is the serial port, even if a graphical console is available. Therefore, for some Linux OSes, set the preference to **Graphical** to remove the SPCR table to make the graphical console work. To select the graphical console, open **Device Manager** in the UEFI menu and select **Console Preference Selection**. The **Console Preference Selection** option is shown in the following screenshot:

Figure 2-11: Console Preference Selection option



In the **Console Preference Selection** menu, select **Graphical** or **Serial**. To get serial console messages, set the preference to **Serial**.

Note

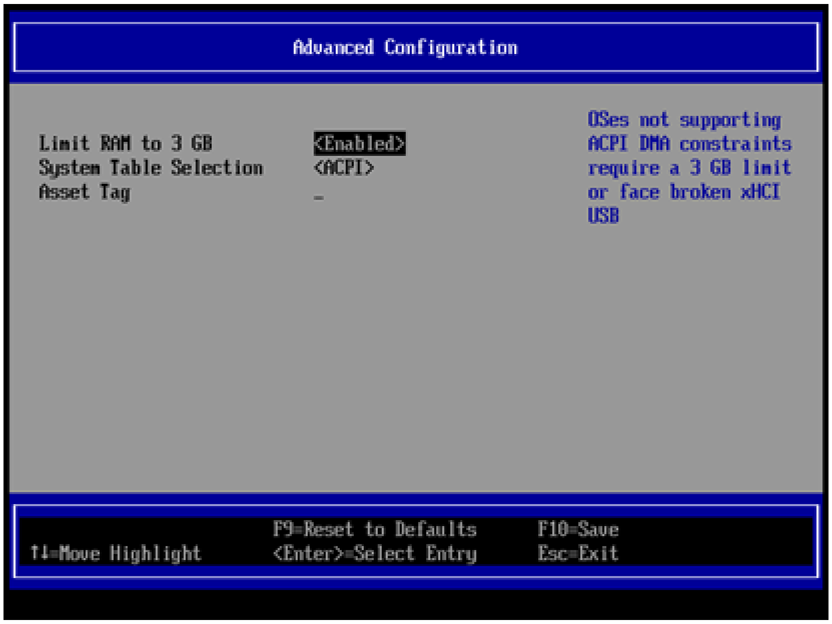
The graphical console removes the serial console on most OSes because the UEFI does not install the SPCR ACPI table. This setting must be **Serial** when running the ACS test suite because the SPCR ACPI table is mandatory for SystemReady ES and is used in parts of the ACS test.

2.10 Limit RAM to 3GB

Currently, many operating systems support 3GB of RAM on the Raspberry Pi 4. To set the limit to 3GB, from the UEFI menu go to **Device Manager** > **Raspberry Pi Configuration** > **Advanced**

Configuration and enable Limit RAM to 3GB. The RAM limit setting is shown in the following screenshot:

Figure 2-12: RAM limit enabled



The following operating systems do not require a 3GB RAM limit:

- OpenBSD 6
- NetBSD 9
- VMWare ESXi

3. Set up the RD-N2 FVP

This section of the guide describes how to set up the RD-N2 FVP.

3.1 Set up the host machine and download the software stack

A host machine with Ubuntu 18.04 or Ubuntu 20.04 with 64GB of free disk space and 32GB of RAM is the minimum requirement to sync and build the platform software stack. However, 48GB of RAM is recommended.

Follow the instructions in the setup-workspace.rst in the Arm-reference solutions GitLab to install the necessary tools and download the source code for the software stack.

If the host machine's memory is less than 32GB, follow the instructions for using the swap file to enable virtual memory.

A display manager is required to run the FVP. Using a text console to connect to the host machine does not work. For remote access to the host machine, you need a console application that supports display export. For example, you can follow these instructions to install the `lightdm` display manager and then install a remote desktop tool such as `xrdp`. An alternative is to use MobaXterm.

3.2 Download the RD-N2 FVP

The RD-N2 FVP installer is available from the Neoverse Infrastructure FVPs section on the arm-ecosystem-fvps site.

Run the following commands to download and install RD-N2 FVP:

```
$ wget https://developer.arm.com/-/media/Arm%20Developer%20Community/Downloads/OSS/
FVP/Neoverse-N2/Neoverse-N2-11-20-18-release/FVP_RD_N2_11.20_18_Linux64.tgz
$ tar -xvzf FVP_RD_N2_11.20_18_Linux64.tgz
$ ls
FVP_RD_N2_11.20_18_Linux64.tgz  FVP_RD_N2.sh  license_terms
$ ./FVP_RD_N2.sh
/FVP_RD_N2$ ls
bin  fmtplib          Iris           models   scripts
doc  install_history  license_terms  plugins  sw
```

For more information, see the rdn2/readme.rst in the Arm-reference solutions GitLab.

3.3 Build the software stack and run the FVP

Follow the instructions in the links below to build and run the FVP:

- ACS compliance test on Neoverse RD platforms
- WinPE boot on Neoverse RD platforms
- Install and boot an OS on Neoverse RD platforms

Note

You must run the FVP with the root account to access the console logs.

---

4. Preparation

This section of the guide describes the preparation that is required before running the SystemReady tests.

4.1 Install and boot requirements

SystemReady SR/ES operating systems must boot free of board-specific images and with generic installation instructions. For example, do not use Raspberry Pi versions of an OS and OS install guides. SystemReady SR/ES does not use special images and guides, and ensures your images are suitable for Arm64.

4.2 Prepare the OS installer media

Before you prepare the installer media, download the AARCH64 installer image for your OS. The following table provides links to install tested OSes for System Ready SR/ES. For more information, see OS-image-download-links.txt in the ES/SR template.

Operating system	Download link
VMware ESXi-Arm Fling	ESXi Arm Edition
Red Hat Enterprise Linux (RHEL)	RHEL Server ISO - RHEL ARM 64
Fedora Server	Standard ISO image for aarch64
Fedora Workstation (Live ISO)	aarch64 Live ISO
SUSE Linux Enterprise Server (SLES)	Evaluation Copy of SUSE Linux Enterprise Server | SUSE
OpenSUSE Leap	OpenSUSE DVD iso
OpenSUSE Tumbleweed (Daily Build)	openSUSE Tumbleweed - Get openSUSE
Ubuntu Server	64-bit ARM (ARMv8/AArch64) server install image
Ubuntu Desktop Live (Daily Build)	64-bit ARM (ARMv8/AArch64) desktop image
Debian	arm64 DVD iso
NetBSD	NetBSD/evbarm
OpenBSD	OpenBSD FAQ: Installation Guide
FreeBSD	Download FreeBSD | The FreeBSD Project

Note

This list does not indicate that the OS is officially supported on the system. Please consult the system and OS vendors for official support.

You can use the disk tools below to set up a USB storage device with an OS installer. To set up the device, insert the USB drive then use a disk tool to restore a disk image to the drive.

- Rufus or balenaEtcher on Windows

> **Note**
> Use balenaEtcher for RHEL, Fedora, CentOS, and AlmaLinux because of an OS installer known issue that results in a “source can’t be found” error.

- `dd` command on Linux

  For example, if your USB drive is `/dev/sda` and you want to restore the Ubuntu install image, use the following command:

```
dd if=ubuntu-22.04.4-live-server-arm64.iso of=/dev/sda status=progress
```

If installation problems occur, for example a system hang in the OS bootloader, cleaning the media device may be needed as follows:

- `diskpart` on Windows:

```
C:\diskpart
DISKPART> list disk
DISKPART> select disk x (Where "x" it's the letter of the USB drive)
DISKPART> clean (if installation issue still exists, try "clean all" This may
 take hours.)
DISKPART> exit
```

- `dd` command on Linux:

  In the first instance, only clean the first megabyte. In most cases, this will fix the issue:

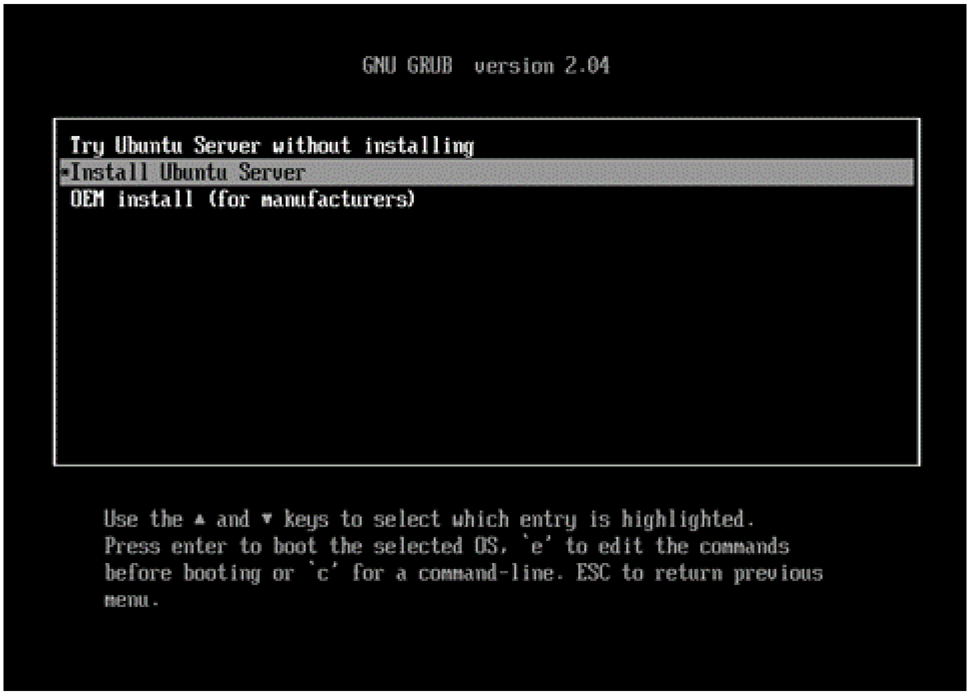
```
dd if=/dev/zero of=/dev/sdb bs=1M count=1 status=progress
```

If the installation issue persists, perform a deep clean. This may take several hours:

```
dd if=/dev/zero of=/dev/sdb bs=1M status=progress
```

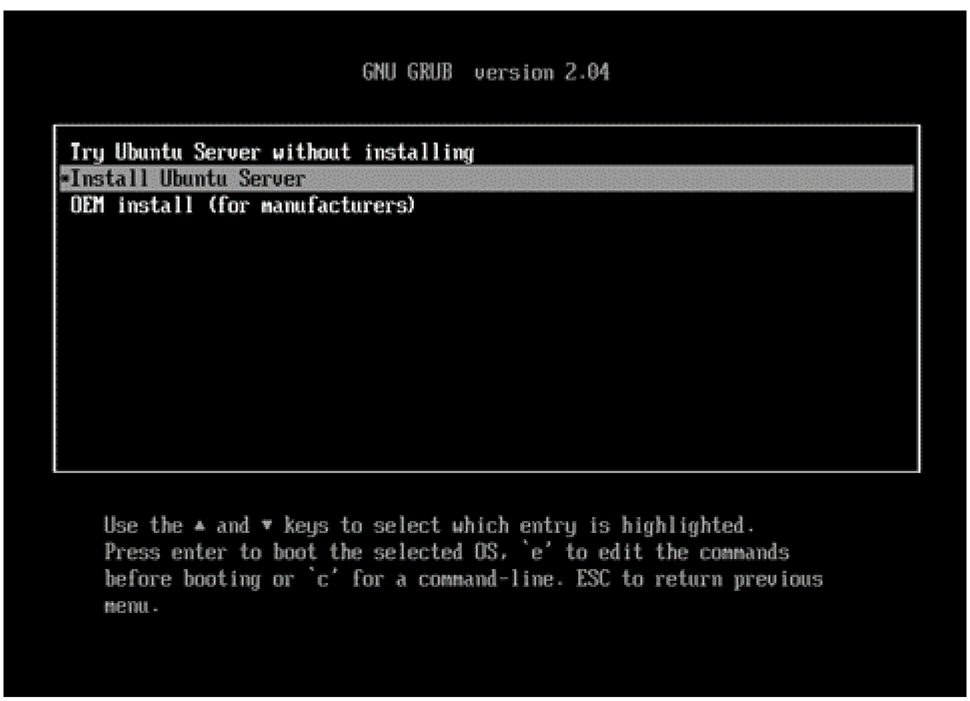
After you create the install media, insert the drive into the USB 3.0 (blue) USB ports on the system. If the USB drive is the first boot option, UEFI will discover and automatically boot into the installer media. The OS bootloader is shown in the following screenshot:

Figure 4-1: GRUB loader



If the first boot option is UEFI shell or PXE boot, press Esc to interrupt the boot process. In the UEFI menu, go to **Boot Manager** then choose the install media (USB drive). In the following screenshot, the USB key is called UEFI Verbatim STORE N GO:

Figure 4-2: USB key in Boot Manager



Press **Enter** to launch the OS bootloader. At this point, you can follow the installation instructions provided by your OS. For example, refer to Ubuntu or Fedora. Install the operating system to a storage device, not the installer media or the SD card that you used to store your firmware.

Note

Many operating systems have images and guides specific to a platform like Raspberry Pi 4. However, these guides are often designed without SystemReady SR/ES.

VMware offers ESXi-Arm Fling as a technical preview for evaluation. For more information, see ESXi Arm Edition.

4.3 Boot order

For the case where UEFI variables are not supported at runtime, the OS may not be able to create a boot entry, so the installed OS might not be automatically booted after installation and reboot.

In this case, you can modify the boot order to solve this problem:

1. After installation, power cycle the system an extra time or enter the UEFI configurator as described in Configure UEFI.
2. Open the Boot Maintenance Manager and change the boot order.
3. If the storage device with the installed OS is not at the top of the list, highlight the device and press + until it is at the top of the list.
4. Press **Enter**, then save and exit.

5. Install Windows PE

Windows PE (WinPE) is a small operating system used to deploy, troubleshoot, and repair Windows installations. Windows OS is required to build the USB key and ISO. This guide uses Windows ADK version 2004.

In this section, you will learn about the following steps:

- Build the ISO and USB key on a device running Windows 10
- Install ADK on Windows 10
- Build the WinPE image
- Set up QEMU to install WinPE

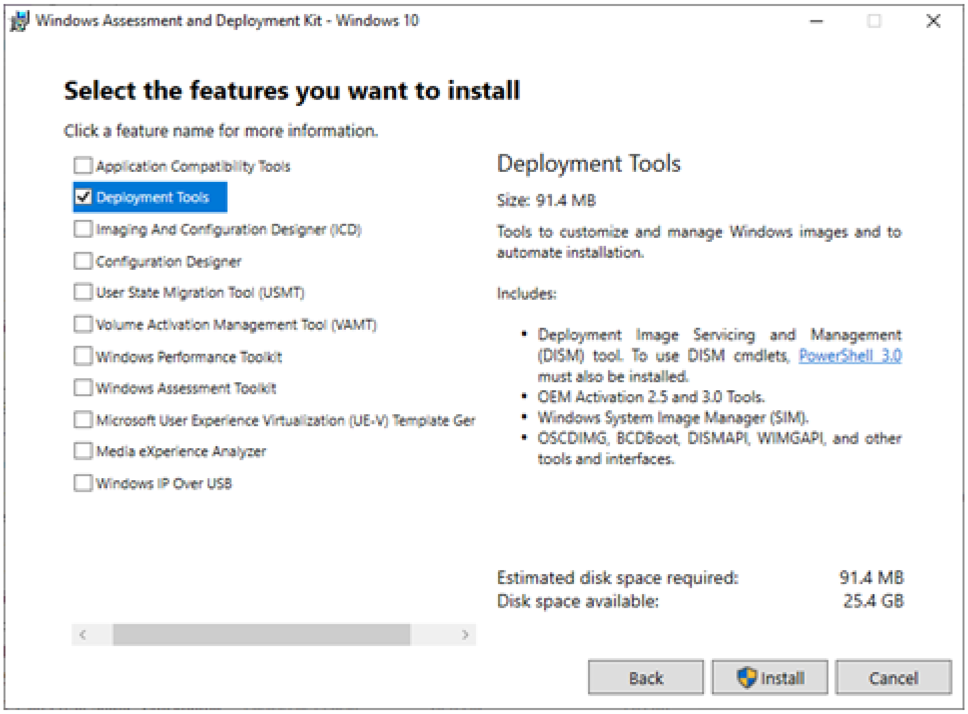
5.1 Download and run Windows ADK and WinPE

Microsoft does not provide an .iso file for WinPE. Instead, download the Windows ADK and Windows PE here to build one yourself.

To install and run Windows ADK and WinPE:

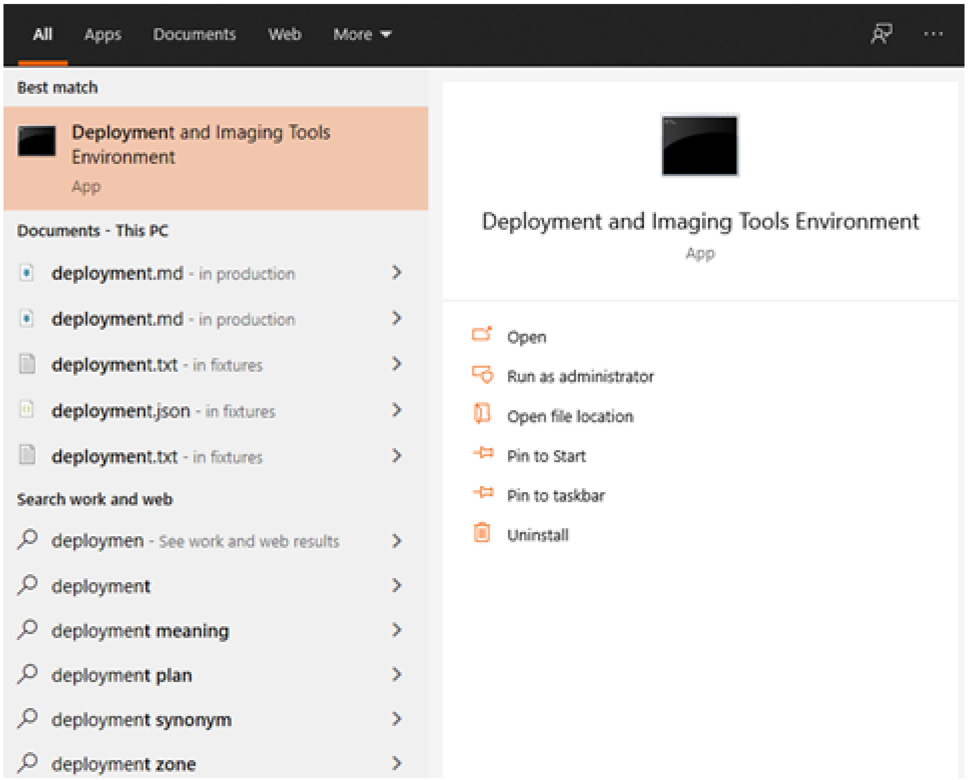
1. Run the `adksetup.exe` installer.
2. Select **Install the Windows Assessment and Deployment Kit – Windows 10 to this Computer** and follow the installer to feature selection.
3. Enable the **Deployment Tools** feature to build a WinPE image, as shown in the following screenshot:

Figure 5-1: Windows ADK features



4. Run the WinPE `adkwinpesetup.exe` installer and install the Windows Preinstallation Environment feature.
5. Create a bootable WinPE USB drive using the **Deployment and Imaging Tools Environment** as Administrator. The following screenshot shows how to start the Deployment and Imaging Tools Environment app window with administrator privileges:

Figure 5-2: Starting Deployment and Imaging Tools Environment



The Create bootable WinPE media guide uses amd64 architecture. Use Arm64 architecture to build an Arm64 USB.

6. If you are creating an ISO file, follow the instructions in Create an ISO file to change the boot parameters.
7. Run the following command to create a working copy of the Windows PE arm64 files:

```
> copype arm64 C:\WinPE_arm64
```

8. Create bootable media using MakeWinPEMedia. You can either create an ISO file or format a USB key directly.

5.2 Create an ISO file

To create an ISO file, change the boot parameters before creating the media. The files in the `\media` folder are copied to the USB key. This lets you change the boot parameters without having to mount the ISO.

To enable EMS or serial console on the .iso image, use the following commands:

```
> cd C:\WinPE_arm64\media\EFI\Microsoft\Boot
```

```
C:\WinPE_arm64\media\EFI\Microsoft\Boot> bcdedit /store BCD /set {default} ems ON
```

Use the following command to create the ISO image.

```
> MakeWinPEMedia /ISO C:\WinPE_arm64 C:\WinPE_arm64\WinPE_arm64.iso
```

5.3 Install to a USB drive

To prepare the USB drive, use the following commands:

```
C:\diskpart
DISKPART> list disk
DISKPART> select disk x (Where "x" it's the number of the USB drive)
DISKPART> clean (if installation issue still exists, try "clean all" This may take
 hours.)
DISKPART> create partition primary
DISKPART> format fs=fat32 quick label="WINPE"
DISKPART> assign letter P
DISKPART> exit
```

To install directly to the USB drive and format the drive, use the following command:

```
> MakeWinPEMedia /UFD C:\WinPE_arm64 P:
```

To enable the EMS serial console on the WinPE media, enter the following commands:

```
> P:
P:\> cd P:\EFI\Microsoft\Boot\
P:\EFI\Microsoft\Boot> bcdedit /store BCD /set {default} ems ON
```

5.4 Other Boot Configuration Data settings

If the system has one UART, you cannot enable WinDBG and EMS at the same time.

To enable WinDBG serial debug, use the following commands:

```
> bcdedit /store BCD /dbgsettings SERIAL DEBUGPORT:1 BAUDRATE:115200
> bcdedit /store BCD /set {default} debug ON
```

Enter `bcdedit /store BCD /enum all` to list all Boot Configuration Data (BCD) settings.

5.5 Boot WinPE

To boot WinPE on an Arm64 system:

- Flash the WinPE ISO image to a media device, for example a USB drive.
- Install the media device on the system, for example by plugging the USB drive into a USB port
- Boot from the media device.

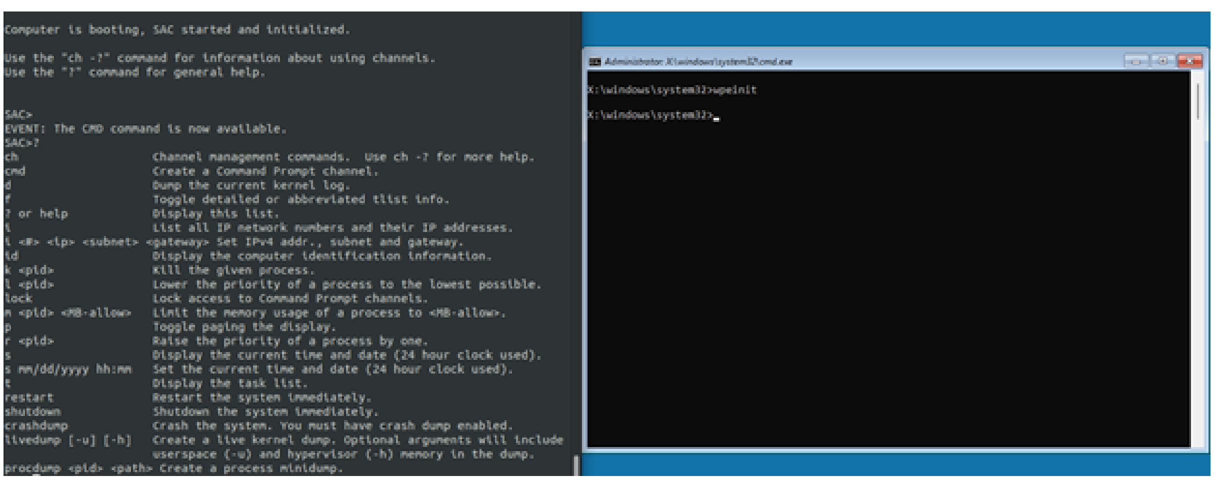
If you do not have an Arm64 system, use QEMU to emulate a system. To install QEMU and boot WinPE from an ISO file, follow the steps below:

1. Install QEMU from edk2-platforms Sbsa-Qemu and follow the instructions in this repository to build the UEFI firmware. You must use QEMU version 4.1.0 or later.
2. Run `git submodule update --init` in the `edk2` and `edk2-platforms` repositories after cloning them.
3. Compile QEMU with `gtk` enabled using `--enable-gtk` on the `../configure`
4. Start QEMU and provide an ISO file as a parameter for the `-cdrom`. In this step, `~/winpe-iso.iso` is the ISO file from the `.` directory. The following command shows how to start QEMU using `winpe-iso.iso` as a parameter:

```
$ qemu-system-aarch64 -m 1024 -M sbsa-ref -pflash SBSA_FLASH0.fd -pflash
 SBSA_FLASH1.fd -display gtk -cdrom ~/winpe-iso.iso -device qemu-xhci -device
 usb-mouse -device usb-kbd -serial stdio
```

5. Press any key to boot WinPE from CDROM. A `cmd` window is displayed and a SAC console in the UART terminal if you enabled EMS in the boot configuration. The following screenshot shows an example of the console and cmd window:

Figure 5-3: SAC console and cmd window



In QEMU, the keyboard and mouse do not work on the display, however the SAC terminal is fully functional.

6. ACS

SystemReady uses the Arm Architecture Compliance Suite (ACS), to help validate system compliance.

6.1 Install and run ACS

The ACS ensures architectural compliance across the architecture implementations. ACS includes examples of the behaviors in SystemReady SR/ES systems that can be verified for compliance.

Download the prebuilt images for each release from the ACS SR prebuilt_images and ACS ES prebuilt_images repositories. After flashing the prebuilt image to a media device, a FAT file system partition is created for test results, to store logs. Another FAT partition is created with bootable applications and test suites.

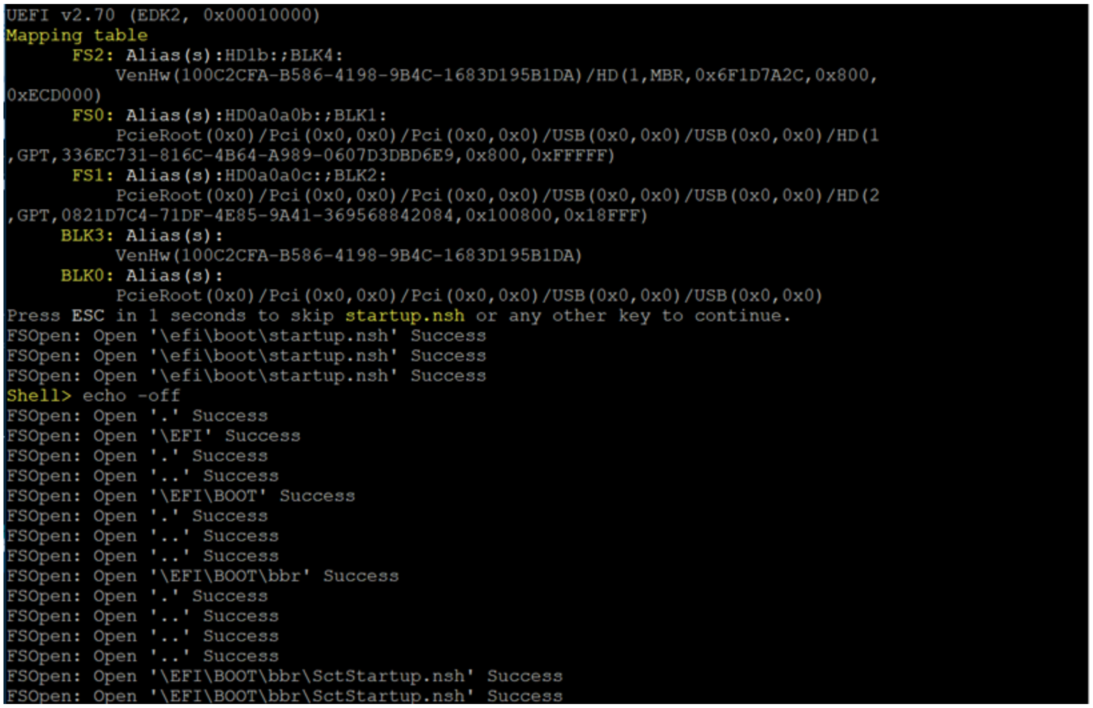
Flash the ACS prebuilt image to the USB drive. Insert the USB drive into the system and boot to the UEFI setup menu. Move the boot option for the ACS USB drive to the top of the boot order. Then boot from the USB drive.

In the GRUB menu, select **bbr/bsa**. This menu performs the following tests:

- UEFI Shell application for SBSA compliance
- SCT tests for SBBR compliance
- FWTS tests for SBBR compliance
- OS tests for SBSA compliance

After 10 seconds, the board boots to the UEFI shell and a sequence of tests start, as shown in the following screenshot:

Figure 6-1: SBBR and SBSA tests running in the UEFI shell

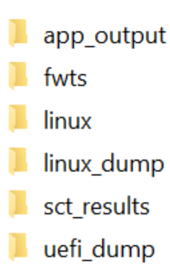


```
UEFI v2.70 (EDK2, 0x00010000)
Mapping table
      FS2: Alias(s):HD1b:;BLK4:
          VenHw(100C2CFA-B586-4198-9B4C-1683D195B1DA)/HD(1,MBR,0x6F1D7A2C,0x800,
0xECD000)
      FS0: Alias(s):HD0a0a0b:;BLK1:
          PcieRoot(0x0)/Pci(0x0,0x0)/Pci(0x0,0x0)/USB(0x0,0x0)/USB(0x0,0x0)/HD(1
,GPT,336EC731-816C-4B64-A989-0607D3DBD6E9,0x800,0xFFFFF)
      FS1: Alias(s):HD0a0a0c:;BLK2:
          PcieRoot(0x0)/Pci(0x0,0x0)/Pci(0x0,0x0)/USB(0x0,0x0)/USB(0x0,0x0)/HD(2
,GPT,0821D7C4-71DF-4E85-9A41-369568842084,0x100800,0x18FFF)
     BLK3: Alias(s):
          VenHw(100C2CFA-B586-4198-9B4C-1683D195B1DA)
     BLK0: Alias(s):
          PcieRoot(0x0)/Pci(0x0,0x0)/Pci(0x0,0x0)/USB(0x0,0x0)/USB(0x0,0x0)
Press ESC in 1 seconds to skip startup.nsh or any other key to continue.
FSOpen: Open '\efi\boot\startup.nsh' Success
FSOpen: Open '\efi\boot\startup.nsh' Success
FSOpen: Open '\efi\boot\startup.nsh' Success
Shell> echo -off
FSOpen: Open '.' Success
FSOpen: Open '\EFI' Success
FSOpen: Open '.' Success
FSOpen: Open '..' Success
FSOpen: Open '\EFI\BOOT' Success
FSOpen: Open '.' Success
FSOpen: Open '..' Success
FSOpen: Open '..' Success
FSOpen: Open '\EFI\BOOT\bbr' Success
FSOpen: Open '.' Success
FSOpen: Open '..' Success
FSOpen: Open '..' Success
FSOpen: Open '..' Success
FSOpen: Open '\EFI\BOOT\bbr\SctStartup.nsh' Success
FSOpen: Open '\EFI\BOOT\bbr\SctStartup.nsh' Success
```

After the tests finish running, the Raspberry Pi automatically boots to the OS to perform Firmware Test Suite (FWTS) tests.

Test results can be checked in the log files on the `RESULT` partition. Unplug the USB drive from the Raspberry Pi and connect the drive to a PC. In the `RESULT` partition on the USB drive, navigate to the `acs_results` folder. The following subfolders contain test results:

Figure 6-2: acs_results directory



By default, ACS executes tests automatically.

7. Debugging commands

The following Linux commands are helpful for debugging:

Command	Description
`hostnamectl`	Control the system hostname
`lspci`	Display information about PCI buses in the system and devices connected to them
`lspci -vvv`	Display everything that can be parsed
`lsusb`	Display information about USB buses in the system and the devices connected to them
`lsusb -v`	Display detailed information about the USB devices shown. This information includes configuration descriptors for the current speed of the device. Class descriptors are shown for USB device classes including hub, audio, HID, communications, and chipcard.
`df`	Report file system disk space usage
`cat /etc/os-release`	Show operating system identification data

8. Advanced Configuration and Power Interface

SystemReady SR/ES certified devices must be compliant with the following specifications:

- BSA
- SBBR recipe in BBR
- SBSA (only for SR)

The Advanced Configuration and Power Interface (ACPI) describes the hardware resources that are installed on SystemReady SR/ES compliant servers. ACPI also handles aspects of runtime system configuration, event notification, and power management.

For mandatory ACPI tables for SystemReady SR/ES compliant systems, see the Arm Base Boot Requirement (BBR) specification. For example, the Raspberry Pi 4 SystemReady ES compliant system, uses the following mandatory ACPI tables:

- Root System Description Pointer (RSDP)
- Extended system Description Table (XSDT)
- Fixed ACPI Description Table (FACP)
- Differentiated System Description Table (DSDT)
- Debug Port 2 Table (DBG2)
- Generic Timer Descriptor Table (GTDT)
- Multiple APIC Description Table (APIC)
- Processor Property Topology Table (PPTT)
- SPCR Serial Port Console Redirection Table. This table is not published by default. To publish this table, select Device Manager in the UEFI menu, then select Serial as the console device.
- Secondary System Description Table (SSDT)

The ACPI examples in this section demonstrate the following use cases:

- Thermal zones
- Fan cooling devices
- USB XHCI and PCIe
- UART
- Debug port
- Power buttons
- PCIe ECAM

8.1 Example: Thermal zone

Raspberry Pi 4 has hardware resources that allow the OS to perform thermal management. BCM2711 provides a register to read CPU temperature. You can enable platform-specific hardware resources by exposing memory map peripheral addresses with Devicetree or ACPI structures, and provide platform-specific OS drivers. For example, the bcm2711_thermal Linux driver consumes a register address provided through a Devicetree structure and produces an API to read CPU temperature. The OS requires an update for any hardware modifications because a new driver is installed to control this hardware. We recommend that you abstract these hardware resources using ACPI AML methods. In this example, you do not use a platform driver because the hardware resource is represented as an ACPI thermal model.

The ACPI DSDT table defines a simple thermal zone TZ00. TZ00 specifies the following methods:

ACPI Method Name	Description
_TMP	Returns the thermal zone's current temperature in tenths of degrees
_SCP	Sets the platform cooling policy, active or passive. A placeholder on the Raspberry Pi.
_CRT	Returns the critical trip point in tenth of degrees where OSPM must perform a critical shutdown
_HOT	Returns the critical trip point in tenths of degrees where OSPM can choose to transition the system into S4 sleeping state
_PSV	Return the passive cooling policy threshold value in tenths of degrees

The following objects are also presented:

Object	Description
_TZP	Thermal zone polling frequency in tenths of seconds
_PSL	List of processor device objects for clock throttling. Specifies all four cores on Raspberry Pi.

The following code shows a thermal zone (TZ00) implementation, which is listed in the ACPI DSDT table:

```
    Device (EC00)
    {
      Name (_HID, EISAID ("PNP0C06"))
      Name (_CCA, 0x0)
      // all temps in are tenths of K (aka 2732 is the min temps in Linux (aka 0C))
      ThermalZone (TZ00) {
        Method (_TMP, 0, Serialized) {
          OperationRegion (TEMS, SystemMemory, THERM_SENSOR, 0x8)
          Field (TEMS, DWordAcc, NoLock, Preserve) {
            TMPS, 32
          }
          return (((410040 - ((TMPS & 0x3ff) * 487)) / 100) + 2732);
        }
        Method (_SCP, 3) { }              // receive cooling policy from OS
        Method (_CRT) { Return (3632) }    // (90C) Critical temp point (immediate
power-off)
        Method (_HOT) { Return (3582) }    // (85C) HOT state where OS should
hibernate
        Method (_PSV) { Return (3532) }    // (80C) Passive cooling (CPU throttling)
trip point
        // SSDT inserts _AC0/_AL0 @60C here, if a FAN is configured
```

```
        Name (_TZP, 10)                    //The OSPM must poll this device every 1
seconds
        Name (_PSL, Package () { \_SB_.CPU0, \_SB_.CPU1, \_SB_.CPU2, \_SB_.CPU3 })
      }
    }
```

8.2 Example: Fan cooling device

Raspberry Pi 4 can be connected to extension hats with a variable speed fan, such as a POE hat. You can also connect a simple on/off fan. A POE hat uses the Raspberry Pi 4 proprietary mailbox for fan control and an on/off fan can be controlled with a single GPIO pin. As a result, each fan device uses specific drivers and can be presented to the OS in different ways.

To simplify OSPM and remove the platform driver, ACPI objects and methods can provide fan device information and control to the OS.

ACPI 1.0 defines a fan device, which is suitable for an on/off fan connected to GPIO. ACPI 4.0 defines additional fan device interface objects, enabling OSPM to perform more robust active cooling thermal control.

Currently, Raspberry Pi 4 supports the ACPI 1.0 fan device. The fan and other related objects and operators are specified in the SSDT ACPI table.

The following table lists PFAN fan power resource methods:

ACPI Method Name	Description
`_STA`	Returns the status of a fan device. This example returns the exact value of the GPIO pin which is used to connect a fan. The exact pin used is configured in the UEFI menu.
`_ON`	Puts the power resource into ON state by setting the GPIO pin, which is used to control a fan
`_OFF`	Puts the power resource into OFF state by clearing the GPIO pin, which is used to connect a fan

The following table lists methods and objects for the fan device:

Object	Description
`FAN0`	Fan device object
`_HID`	Plug and Play ID. This should be PNP0C0B
`_PR0`	Power Resource for the fan object (fully ON state)

The following table lists methods and objects for the Active Cooling point:

Object	Description
`_AC0`	Returns the temperature trip point at which OSPM must start or stop Active cooling
`_AL0`	Evaluates a list of Active cooling devices to be turned on when the corresponding _ACx temperature threshold is exceeded. _AL0 defines a single FAN0 device on RPi4

The following code shows the ACPI implementation of a fan cooling device and the device resources:

```
Scope (\_SB_.EC00)
  {
    // Define a NameOp we will modify during InstallTable
    Name (GIOP, 0x2) //08 47 49 4f 50 0a 02 (value must be >1)
    Name (FTMP, 0x2)
    // Describe a fan
    PowerResource (PFAN, 0, 0) {
      OperationRegion (GPIO, SystemMemory, GPIO_BASE_ADDRESS, 0x1000)
      Field (GPIO, DWordAcc, NoLock, Preserve) {
        Offset (0x1C),
        GPS0, 32,
        GPS1, 32,
        RES1, 32,
        GPC0, 32,
        GPC1, 32,
        RES2, 32,
        GPL1, 32,
        GPL2, 32
      }
      // We are hitting a GPIO pin to on/off a fan.
      // This assumes that UEFI has programmed the
      // direction as OUT. Given the current limitations
      // on the GPIO pins, its recommended to use
      // the GPIO to switch a larger voltage/current
      // for the fan rather than driving it directly.
      Method (_STA) {
        if (GPL1 & (1 << GIOP)) {
          Return (1)                // present and enabled
        }
        Return (0)
      }
      Method (_ON)  {               // turn fan on
        Store (1 << GIOP, GPS0)
      }
      Method (_OFF) {               // turn fan off
        Store (1 << GIOP, GPC0)
      }
    }
    Device (FAN0) {
      // Note, not currently an ACPIv4 fan
      // the latter adds speed control/detection
      // but in the case of linux needs FIF, FPS, FSL, and FST
      Name (_HID, EISAID ("PNP0C0B"))
      Name (_PR0, Package () { PFAN })
    }
  }
  // merge in an active cooling point.
  Scope (\_SB_.EC00.TZ00)
  {
    Method (_AC0) { Return ( (FTMP * 10) + 2732) } // (60C) active cooling trip
 point,
                                                 // if this is lower than PSV then
 we
                                                 // prefer active cooling
    Name (_AL0, Package () { \_SB_.EC00.FAN0 })    // the fan used for AC0 above
  }
```

With the ACPI 1.0 fan, you do not need a platform-specific GPIO driver and a temperature monitor. The ACPI fan driver consumes the PNP0C0B FAN0 device and uses an ACPI power subsystem to turn it on or off.

Use the following Hat 4 methods with ACPI 4.0 on a Raspberry Pi 4 for POE:

Object	Description
_FIF	Returns fan device information
_FPS	Returns a list of supported fan performance states
_FSL	Control method that sets the fan device's speed level (performance state). RPI_FIRMWARE_SET_POE_HAT_VAL would be used in ACPI AML on a Raspberry Pi 4.

In this example, instead of exposing a proprietary mailbox to the OS and using a platform driver, we allow the OS to use a standard ACP fan driver.

8.3 Example: USB XHCI and PCIe

If a PCIe controller is present and visible by the operating system, you must use an MCGF table.

The PCIe controller is present on the Raspberry Pi 4, but it is not SBSA compatible. To certify a Raspberry Pi 4 as SystemReady ES compliant, the PCIe is hidden and as a result MCFG is not used.

The USB XHCI controller is connected to the PCIe controller, and an ACPI node XHC0 is added to the DSDT table. Also, a _DMA object is defined to describe resources consumed by XCH0.

The following code shows the ACPI_DMA resource:

```
Name (_DMA, ResourceTemplate() {
        /*
         * XHC0 is limited to DMA to first 3GB. Note this
         * only applies to PCIe, not GENET or other devices
         * next to the A72.
         */
        QWordMemory (ResourceConsumer,+
            ,
            MinFixed,
            MaxFixed,
            NonCacheable,
            ReadWrite,
            0x0,
            0x0,        // MIN
            0xbfffffff, // MAX
            0x0,        // TRA
            0xc0000000, // LEN
            ,
            ,
            )
    })
```

_DMA is an optional object and returns a byte stream in the same format as a _CRS object. _DMA is defined under devices that represent buses, such as Device SCB0 for the Raspberry Pi 4. This object specifies the ranges the bus controller decodes on the child interface. This is analogous to the _CRS object, which describes the resources that the bus controller decodes on the parent interface. The ranges described in the resources of a _DMA object can be used by child devices for DMA or bus master transactions.

The `_DMA` object is only valid if a `_CRS` object is defined. The OSPM must reevaluate the `_DMA` object after an `_SRS` object has been executed because the `_DMA` ranges resources may change depending on how the bridge has been configured.

The following code shows the ACPI XCH0 USB 3.0 controller implementation:

```
Device (XHC0)
{
    Name (_HID, "PNP0D10")  // Hardware ID
    Name (_UID, 0x0)        // Unique ID
    Name (_CCA, 0x0)        // Cache Coherency Attribute
    Method (_CRS, 0, Serialized) {  // Current Resource Settings
        Name (RBUF, ResoureTemplate() {
            QWordMemory (ResourceConsumer,
                ,
                MinFixed,
                MaxFixed,
                NonCacheable,
                ReadWrite,
                0x0,
                SANITIZED_PCIE_CPU_MMIO_WINDOW, // MIN
                SANITIZED_PCIE_CPU_MMIO_WINDOW, // MAX
                0x0,
                0x1,                            // LEN
                ,
                ,
                MMIO
                )
            Interrupt (ResourceConsumer, Level, ActiveHigh, Exclusive, ,, ) {
                175
            }
        })
        CreateQwordField (RBUF, MMIO._MAX, MMBE)
        CreateQwoedField (RBUF, MMIO._LEN, MMLE)
        Add (MMBE, XHCI_REG_LENGTH - 1, MMBE)
        Add (MMLE, XHCI_REG_LENGTH - 1, MMLE)
        Return (RBUF)
    }
Method (_INI, 0, Serialized) {
    OperationRegion (PCFG, SystemMemory, SANITIZED_PCIE_REG_BASE +
 PCIE_EXT_FG_DATA, 0x10000)
    Field (PCFG, AnyAcc, NoLock, Preserve) {
        VNID, 16, // Vendor ID
        DVID, 16, // Device ID
        CMND, 16, // Command register
        STAT, 16, // Status register
    }
    Debug = "xHCI enable"
    Store (0x6, CMND)
}
}
```

8.4 Example: UART

Arm SBSA Generic UART and 16550 UART devices can be presented in the system. Serial Console Redirection (SPCR) can be used to describe these devices.

The Raspberry Pi 4 has a PL011 UART port described in `spcr.aslc` using C. The following code snippet shows the ACPI UART PL011 implementation:

```
STATIC EFI_ACPI_SERIAL_PORT_CONSOLE_REDIRECTION_TABLE Spcr = {
  ACPI_HEADER (
    EFI_ACPI_6_3_SERIAL_PORT_CONSOLE_REDIRECTION_TABLE_SIGNATURE,
    EFI_ACPI_SERIAL_PORT_CONSOLE_REDIRECTION_TABLE,
    EFI_ACPI_SERIAL_PORT_CONSOLE_REDIRECTION_TABLE_REVISION
  ),
  // UINT8                                   InterfaceType;
  RPI_UART_INTERFACE_TYPE,
  // UINT8                                   Reserved1[3];
  {
    EFI_ACPI_RESERVED_BYTE,
    EFI_ACPI_RESERVED_BYTE,
    EFI_ACPI_RESERVED_BYTE
  },
  // EFI_ACPI_6_3_GENERIC_ADDRESS_STRUCTURE  BaseAddress;
  ARM_GAS32 (RPI_UART_BASE_ADDRESS),
  // UINT8                                   InterruptType;
  EFI_ACPI_SERIAL_PORT_CONSOLE_REDIRECTION_TABLE_INTERRUPT_TYPE_GIC,
  // UINT8                                   Irq;
  0,                                         // Not used on ARM
  // UINT32                                  GlobalSystemInterrupt;
  RPI_UART_INTERRUPT,
  // UINT8                                   BaudRate;
#if (FixedPcdGet64 (PcdUartDefaultBaudRate) == 9600)
  EFI_ACPI_SERIAL_PORT_CONSOLE_REDIRECTION_TABLE_BAUD_RATE_9600,
#elif (FixedPcdGet64 (PcdUartDefaultBaudRate) == 19200)
  EFI_ACPI_SERIAL_PORT_CONSOLE_REDIRECTION_TABLE_BAUD_RATE_19200,
#elif (FixedPcdGet64 (PcdUartDefaultBaudRate) == 57600)
  EFI_ACPI_SERIAL_PORT_CONSOLE_REDIRECTION_TABLE_BAUD_RATE_57600,
#elif (FixedPcdGet64 (PcdUartDefaultBaudRate) == 115200)
  EFI_ACPI_SERIAL_PORT_CONSOLE_REDIRECTION_TABLE_BAUD_RATE_115200,
#else
#error Unsupported SPCR Baud Rate
#endif
  // UINT8                                   Parity;
  EFI_ACPI_SERIAL_PORT_CONSOLE_REDIRECTION_TABLE_PARITY_NO_PARITY,
  // UINT8                                   StopBits;
  EFI_ACPI_SERIAL_PORT_CONSOLE_REDIRECTION_TABLE_STOP_BITS_1,
  // UINT8                                   FlowControl;
  RPI_UART_FLOW_CONTROL_NONE,
  // UINT8                                   TerminalType;
  EFI_ACPI_SERIAL_PORT_CONSOLE_REDIRECTION_TABLE_TERMINAL_TYPE_VT_UTF8,
  // UINT8                                   Reserved2;
  EFI_ACPI_RESERVED_BYTE,
  // UINT16                                  PciDeviceId;
  0xFFFF,
  // UINT16                                  PciVendorId;
  0xFFFF,
  // UINT8                                   PciBusNumber;
  0x00,
  // UINT8                                   PciDeviceNumber;
  0x00,
  // UINT8                                   PciFunctionNumber;
  0x00,
  // UINT32                                  PciFlags;
  0x00000000,
  // UINT8                                   PciSegment;
```

```
      0x00,
      // UINT32                                  Reserved3;
      EFI_ACPI_RESERVED_DWORD
    };
```

8.5 Example: Debug port

For some OSes, the debug port is presented on the platform. To describe the debug ports available on the platform, Debug Port Table 2 is used. The table contains information about the configuration of the debug port.

The Raspberry Pi 4 has a PL011 UART port that can be described to the OS as a debug port. The following code shows the ACPI UART PL011 debug port implementation:

```
    #define RPI_DBG2_NUM_DEBUG_PORTS                        1
    #define RPI_DBG2_NUMBER_OF_GENERIC_ADDRESS_REGISTERS    1
    #define RPI_DBG2_NAMESPACESTRING_FIELD_SIZE             15
    #define RPI_UART_INTERFACE_TYPE
 EFI_ACPI_DBG2_PORT_SUBTYPE_SERIAL_ARM_PL011_UART
    #define RPI_UART_BASE_ADDRESS                BCM2836_PL011_UART_BASE_ADDRESS
    #define RPI_UART_LENGTH                      BCM2836_PL011_UART_LENGTH
    #define RPI_UART_STR                         { '\\', '_', 'S', 'B', '.', 'G',
 'D', 'V', '0', '.', 'U', 'R', 'T', '0', 0x00 }
    STATIC DBG2_TABLE Dbg2 = {
      {
        ACPI_HEADER (
          EFI_ACPI_6_3_DEBUG_PORT_2_TABLE_SIGNATURE,
          DBG2_TABLE,
          EFI_ACPI_DBG2_DEBUG_DEVICE_INFORMATION_STRUCT_REVISION
        ),
        OFFSET_OF (DBG2_TABLE, Dbg2DeviceInfo),
        RPI_DBG2_NUM_DEBUG_PORTS
      },
      {
        /*
         * Kernel Debug Port
         */
        DBG2_DEBUG_PORT_DDI (
          RPI_DBG2_NUMBER_OF_GENERIC_ADDRESS_REGISTERS,
          RPI_UART_INTERFACE_TYPE,
          RPI_UART_BASE_ADDRESS,
          RPI_UART_LENGTH,
          RPI_UART_STR
        ),
      }
    };
```

BBR requires platforms to keep a debug port on a separate UART port from the console port so there is no conflict in debug messages and OS console output. Because the Raspberry Pi has only one active UART, enable or disable DBG2 as needed for debugging.

8.6 Example: Power button

If you remove the power cable from the device without shutting down the OS, the file system can be corrupted and other unrecoverable errors can occur. A power button is a useful addition to the embedded platform, which allows an OS to implement shutdown safely.

If we connect a button to one of the Raspberry Pi 4 GPIO pins, we can define an ACPI power button. The GPIO interrupt functionality in the BCM2711 is used with a Generic Event Device to generate the Notify command to tell OSPM that the button has been pressed. The OS then initiates sleep or soft shutdown based on user settings.

The Generic Event Device has the following objects:

Object	Description
GED1	Generic Event Device Object
_HID	Plug and Play ID: ACPI0013 for GED
_CRS	List of interrupts

The following table lists the Generic Event Device methods:

Method	Description
_EVT	Interrupt handler. This has arg0, which contains the Generic System Interrupt Vector of the interrupt.
_INI	Platform Specific Initialization

The power button has the following objects:

Object	Description
PWRB	Power Button object
_HID	Plug and Play ID: PNP0C0C for power button

The following table lists the power button methods:

Method	Description
_STA	Status of the device. We return `0xF`, which means the device is present, enabled, should be shown in UI and is functioning properly.

Using the `_INI` method, we set up GPIO pin 5 to trigger an interrupt when a rising edge is detected. Then, in the `_EVT` method, we check the status of the pins to check that the interrupt was GPIO0, and that pin 5 triggered the interrupt. If the interrupt is triggered, the status is reset and the power button notified.

The following code shows an ACPI power button implementation:

```
// Generic Event Device
Device (GED1) {
  Name (_HID, "ACPI0013")
  Name (_UID, 0)
  Name (_CRS, ResourceTemplate () {
```

```
    Interrupt(ResourceConsumer, Edge, ActiveHigh, ExclusiveAndWake) {
    BCM2386_GPIO_INTERRUPT0 }
  })
  OperationRegion (PH0, SystemMemory, GPIO_BASE_ADDRESS, 0x1000)
  Field (PH0, DWordAcc, NoLock, Preserve) {
    GPF0, 32, /* GPFSEL0 - GPIO Function Select 0 */
    offset(0x40),
    GPE0, 32, /* GPEDS0 - GPIO Pin Event Detect Status 0 */
    GPE1, 32, /* GPEDS1 - GPIO Pin Event Detect Status 1 */
    GRE0, 32, /* GPREN0 - GPIO Pin Rising Edge Detect Enable 0 */
    GRE1, 32, /* GPREN1 - GPIO Pin Rising Edge Detect Enable 1 */
    offset(0xe4),
    GUD0, 32, /* GPIO_PUP_PDN_CNTRL_REG0 - GPIO Pull-up / Pull-down
 Register 0 */
  }
  Method (_INI, 0, NotSerialized) {
    /* 0x00000020 = GPIO pin 5 */
    /* Enable rising edge detect */
    Store(0x00000020, GRE0)
    /* Enable Pull down resistor for pin 5 */
    Store(0x00000800, GUD0)
  }
  Method (_EVT, 1) {
    If (ToInteger(Arg0) == BCM2386_GPIO_INTERRUPT0)
      Name()
      Store(0x00000020, GPE0) // Clear the status
      Notify (\_SB.PWRB, 0x80) // Sleep/Off Request
    }
  }
}
Device (PWRB) {
  Name (_HID, "PNP0C0C")
  Name (_UID, Zero)
  Method (_STA, 0x0, NotSerialized) {
    Return(0xF)
  }
}
```

8.7 Example: PCIe ECAM

If a platform supports PCIe, the platform reports PCIe Configuration Space using the MCFG ACPI table. If the PCIe Root complex is not SBSA compatible, a different approach can be taken.

The Raspberry Pi 4 hides PCIe Configuration space and the MCFG table is not published on this platform. Only the USB XHCI is exposed in the DSDT table.

Alternatively, you can use the Arm PCI Configuration Space Access Firmware Interface. This interface can be used as alternative to the Enhanced Configuration Access Mechanism (ECAM) hardware mechanism .

The interface enables a caller to:

- Access PCI configuration space reads and writes
- Discover the implemented PCI segment groups and bus ranges for each segment

For the list of supported calls, refer to the Arm PCI Configuration Space Access Firmware Interface.

Arm PCI Configuration Space Access Firmware Interface implementation requires the following:

- On the platform with EL3 presented, Platform Firmware SMCCCv1.1 compliant implementation
- If EL3 is not present but EL2 is present, HVC conduit must be implemented in hypervisor
- Operating System SMCCv1.1 compliant SMC or HVC conduit implementation

Enabling Arm PCI Configuration Space Access Firmware Interface requires patches for a platform firmware, UEFI, and an OS.

An example of the SMCCC implementation supporting Arm PCI Configuration Space Access Firmware Interface can be found in Arm Trusted Firmware. Arm Trusted Firmware already allows platforms to handle PCI configuration access requests through standard SMCCC. To enable these access requests, the SMC_PCI_SUPPORT build flag is provided.

To use PCIe SMCCC, we need to describe PCIe Root Complex in the SSDT ACPI table. Refer to this patch [PATCH v2 3/6] Platform/RaspberryPi: Add PCIe SSDT. With this patch, instead of hiding the PCIe root complex, we expose PCIe to the OS. The OS ACPI PCI driver controls the PCIe root complex but because the MCFG table is absent, the driver uses the OS SMC conduit to get access to the PCIe ECAM.

An example of the OS SMC conduit implementation can be found in the NetBSD. NetBSD implements pci_smccc_call(), which uses Secure Monitor Call to request a PCI Configurate access service to a platform firmware running in EL3. With PCI_SMCCC enabled, NetBSD PCIe subsystem uses the PCI_VERSION SMC call to check if the SMCCC supports PCI configuration access. If the SMCCC version is 1.1 or later, the PCI SMCCC is supported.

NetBSD, Arm Trusted Firmware, and EDK2 can be built and run on the Raspberry Pi 4 with PCI SMCCC enabled. As a result, the PCIe is exposed through SMCCC driving the XHCI controller.

In the future, other operation systems or hypervisors such as VMWare ESXi may implement this interface.

8.8 ACPI integration recommendations

ACPI tables can be implemented using a platform driver or dynamic ACPI framework.

For platform drivers, you manually create ACPI tables using ACPI Source Language (ASL). Create a set of `.asl` files and an edk2 module information file `AcpiTable.inf`. You can also create an ACPI table using C language. In this case, `.aslc` files must be used.

These files are compiled at build time and stored in a firmware volume. At boot time, a platform driver uses ArmLib methods, shown in the following code:

```
EFI_STATUS LocateAndInstallAcpiFromFvConditional(
 IN CONST EFI_GUID* AcpiFile,
 IN EFI_LOCATE_ACPI_CHECK CheckAcpiTableFunction
)
or
```

```
EFI_STATUS LocateAndInstallAcpiFromFv(
 IN CONST EFI_GUID* AcpiFile
)
```

These methods locate and install ACPI tables in a firmware volume. The following code snippet locates ACPI tables implemented for the platform and installs it in a firmware volume:

```
Status = LocateAndInstallAcpiFromFv(&mAcpiTableFile);
```

In this example, `mAcpiTableFile` is a GUID of the ACPI storage file in a firmware volume and matches `FILE_GUID` in the `AcpiTable.inf`.

Although ACPI tables are compiled at build time and stored in a firmware volume, it is still possible to modify these tables at boot time. The second parameter `CheckAcpiTableFunction` in `LocateAndInstallAcpiFromFvConditional()` is a pointer to a function. This parameter is an algorithm `LocateAndInstallAcpiFromFvConditional()` used to locate and install ACPI, and performs the following steps:

1. Use `EFI_FIRMWARE_VOLUME2_PROTOCOL` and `mAcpiTableFile` GUID to find an ACPI table in a firmware volume.
2. Prior to the installation of the table, call `CheckAcpiTableFunction()` with a pointer to a newly found ACPI table as a parameter.
3. Provided `CheckAcpiTableFunction()` indicates that the table should be installed, use EFI_ACPI_TABLE_PROTOCOL to install the table.
4. Repeat until all ACPI tables are found and installed.

`CheckAcpiTableFunction()` has a pointer to a newly discovered ACPI table and can modify the table before being installed. For an example, refer to the `HandleDynamicNamespace()` function of the Raspberry Pi 4 ACPI platform driver and see how it is used to modify DSDT and SSDT ACPI tables with values taken from PCD values.

For a Raspberry Pi 4 ACPI table implementation, see AcpiTables.

To learn how ACPI tables are installed on the Raspberry Pi 4, see ConfigDxe.

For another example of the ACPI platform driver, see PlatformDxe. The dynamic ACPI table generators that are implemented as libraries. These generators query a platform-specific Configuration Manager to collate the information required for generating the tables at runtime. See Arm at master for a list of the generators supported.

To implement Configuration Manager, include a platform-specific DXE driver called ConfigurationManagerDxe. Configuration Manager produces EDKII_CONFIGURATION_MANAGER_PROTOCOL and implements its API. The declaration of the API for the EDKII_CONFIGURATION_MANAGER_PROTOCOL is in ConfigurationManagerProtocol.h.

The following code shows the GUID of the Configuration Manager Protocol:

```
#define EDKII_CONFIGURATION_MANAGER_PROTOCOL_GUID       \
  { 0xd85a4835, 0x5a82, 0x4894,                         \
    { 0xac, 0x2, 0x70, 0x6f, 0x43, 0xd5, 0x97, 0x8e }   \
  };
```

The following code shows a software interface of the Configuration Manager Protocol:

```
typedef struct ConfigurationManagerProtocol {
  UINT32                                  Revision;
  EDKII_CONFIGURATION_MANAGER_GET_OBJECT  GetObject;
  EDKII_CONFIGURATION_MANAGER_SET_OBJECT  SetObject;
  EDKII_PLATFORM_REPOSITORY_INFO        * PlatRepoInfo;
} EDKII_CONFIGURATION_MANAGER_PROTOCOL;
```

The API consists of the following functions:

- `GetObject()`. The GetObject() function defines the interface implemented by the Configuration Manager Protocol used to return the Configuration Manager Objects
- `SetObject()`. The SetObject() function defines the interface implemented by the Configuration Manager Protocol to update the Configuration Manager Objects

Configuration Manager Objects are objects that represent platform configuration and are stored in the EDKII_PLATFORM_REPOSITORY_INFO repository, maintained by Configuration Manager.

Configuration Manager maintains a list of ACPI tables to be installed. Based on this list, the corresponding ACPI table generators are invoked by the Dynamic ACPI framework.

For example, the IORT ACPI table generator handles the following ACPI objects:

- EArmObjItsGroup
- EArmObjNamedComponent
- EArmObjRootComplex
- EArmObjSmmuV1SmmuV2
- EArmObjSmmuV3
- EArmObjPmcg
- EArmObjGicItsIdentifierArray
- EArmObjIdMappingArray
- EArmObjGicItsIdentifierArray

If the OEM platform has an SMMUv3 hardware block, include an object with ID equal to EArmObjSmmuV3 in the Configuration Manager repository. For more information, refer to the list of Arm object IDs and data structures in ArmNameSpaceObjects.h.

The IORT ACPI table generator requests the EArmObjSmmuV3 object using the EDKII_CONFIGURATION_MANAGER_GET_OBJECT function and adds the SMMUv3 node to the IORT ACPI table. The same mechanism is used by other ACPI table generators.

For an implementation example, see ConfigurationManager for EDKII_CONFIGURATION_MANAGER_PROTOCOL.

Note

Currently, the capability to generate ASL tables (DSDT and SSDT) is limited to generating ASL Serial Port Information corresponding to DBG2 and SPCR because it is platform-specific.

---

9. SMBIOS requirements

The SMBIOS table version 3.0.0 or later is required to conform to the SMBIOS specification. Earlier SMBIOS table and format versions are not supported.

For information about the required SMBIOS records for SystemReady SR/ES compliant systems, see the Arm Base Boot Requirement (BBR) specification. For example, the Raspberry Pi 4 SystemReady ES compliant system uses the following SMBIOS records:

- Type 00: BIOS information
- Type 01: system information
- Type 02: base board information (optional)
- Type 03: chassis information
- Type 04: processor information
- Type 07: cache information
- Type 09: system slot information
- Type 11: OEM string (optional)
- Type 16: physical memory array
- Type 17: memory device
- Type 19: memory array mapped address
- Type 32: boot status

9.1 SMBIOS integration

SMBIOS data structures are built on top of the platform-independent driver SmbiosDxe, which uses the EFI_SMBIOS_PROTOCOL API. EFI_SMBIOS_PROTOCOL allows consumers to log SMBIOS data records and enables the producer (SmbiosDxe) to create the SMBIOS tables for a platform. SmbiosDxe is responsible for installing the pointer to the tables in the EFI System Configuration Table.

The following code shows a GUID of SMBIOS Protocol:

```
    #define EFI_SMBIOS_PROTOCOL_GUID \
    { 0x3583ff6, 0xcb36, 0x4940, { 0x94, 0x7e, 0xb9, 0xb3, 0x9f,\
    0x4a, 0xfa, 0xf7 } }
```

The following code shows an SMBIOS Protocol data structure:

```
typedef struct _EFI_SMBIOS_PROTOCOL {
 EFI_SMBIOS_ADD Add;
 EFI_SMBIOS_UPDATE_STRING UpdateString;
 EFI_SMBIOS_REMOVE Remove;
 EFI_SMBIOS_GET_NEXT GetNext;
 UINT8 MajorVersion;
```

```
 UINT8 MinorVersion;
} EFI_SMBIOS_PROTOCOL;
```

9.2 Platform driver

The SMBIOS driver is a platform-specific DXE driver that uses SMBIOS data records provided by the OEM. The driver consumes EFI_SMBIOS_PROTOCOL, which is produced by SmbiosDxe and uses its interface to add SMBIOS records.

The driver creates SMBIOS records defined in SmBios.h. These records are standard SMBIOS data structures, defined according to the latest SMBIOS specification.

For example, the following code shows the definition for a TYPE 1 System information SMBIOS table, which is defined by the PlatformSmbiosDxe Raspberry Pi 4 platform driver:

```
SMBIOS_TABLE_TYPE1 mSysInfoType1 = {
  { EFI_SMBIOS_TYPE_SYSTEM_INFORMATION, sizeof (SMBIOS_TABLE_TYPE1), 0 },
  1,    // Manufacturer String
  2,    // ProductName String
  3,    // Version String
  4,    // SerialNumber String
  { 0x25EF0280, 0xEC82, 0x42B0,
 { 0x8F, 0xB6, 0x10, 0xAD, 0xCC, 0xC6, 0x7C, 0x02 } },
  SystemWakeupTypePowerSwitch,
  5,    // SKUNumber String
  6,    // Family String
};
```

PlatformSmbiosDxe uses EFI_SMBIOS_PROTOCOL method `Add()` to add mSysInfoType1 record:

```
Status = gBS->LocateProtocol (&gEfiSmbiosProtocolGuid, NULL, (VOID**)&Smbios);
Status = Smbios->Add (
                     Smbios,
                     gImageHandle,
                     &c,
                     Record // mSysInfoType1
                   );
```

The platform driver is responsible for ensuring that the SMBIOS record is formatted to match the version of the SMBIOS specification as defined in the MajorVersion and MinorVersion fields of the EFI_SMBIOS_PROTOCOL.

Add both a platform driver and SmbiosDxe driver to your platform and flash description files. Use the RPi4.dsc and the RPi4.fdf files as a reference.

For more information about how the platform driver is implemented on the Raspberry Pi 4, see the PlatformSmbiosDxe implementation.

9.3 System Management BIOS framework

The platform driver requires the OEM to define SMB records using C and check that these records are formatted according to the version of the SMBIOS specification as defined in the MajorVersion and MinorVersion fields of the EFI_SMBIOS_PROTOCOL.

The generic Arm System Management BIOS (SMBIOS) framework allows you to generate SMBIOS tables without writing C code. This framework uses platform configuration PCD database entries and strings from a Human Interface Infrastructure (HII).

For example, the OEM can provide the following PCD entries in its platform description file:

- `gEfiMdeModulePkgTokenSpaceGuid.PcdFirmwareVendor`
- `gEfiMdeModulePkgTokenSpaceGuid.PcdFirmwareVersionString`
- `gArmTokenSpaceGuid.PcdSystemBiosRelease`
- `gArmTokenSpaceGuid.PcdEmbeddedControllerFirmwareRelease`

These entries are taken by the SMBIOS framework and added to the SMBIOS table type 00 BIOS information automatically.

The OEM must provide an OemMiscLib library with the following platform-specific definitions:

Processor Information

The SMBIOS framework creates processor and cache information tables and requires the following functions:

- `OemGetCpuFreq()`
- `OemGetProcessorInformation()`
- `OemGetCacheInformation()`
- `OemGetMaxProcessors()`

The SMBIOS framework calls these functions to get processor and cache information and uses the EFI_SMBIOS_PROTOCOL `Add()` function to add SMBIOS type 04 and type 07 tables.

OemUpdateSmbiosInfo() function

The SMBIOS framework uses hardcoded PCD entries to create SMBIOS tables, but platform-specific information is needed in runtime. For example, a baseboard serial number or chassis serial number must not be hardcoded in the UEFI binary the OEM uses to flash the board. The OEM can write `OemUpdateSmbiosInfo()` so that these two strings are read in runtime from a baseboard management controller. The SMBIOS framework calls `OemUpdateSmbiosInfo()` to retrieve these two strings and update default information in the SMBIOS type 02 and type 03 tables.

For more details about the OemMiscLib implementation, see `tianocore/edk2-platforms/Platform/Qemu/SbsaQemu/OemMiscLib`.

For more information about the SMBIOS framework, see https://github.com/tianocore/edk2/tree/master/ArmPkg/Universal/Smbios/SmbiosMiscDxe.

10. UEFI requirements

The boot and system firmware for 64-bit Arm embedded servers is based on the UEFI specification version 2.8 or later and incorporates the AArch64 bindings.

UEFI compliant systems must follow the requirements in section 2.6 of the specification. However, to ensure a common boot architecture for server-class AArch64, systems compliant with this specification must provide the UEFI services and protocol from the provided list.

UEFI compliance is tested using UEFI Self-Certification Tests (SCT) and FWTS. For more information about using SCT and FWTS, see ACS.

For a list of required UEFI runtime and boot services, see the Arm Base Boot Requirements specification.

11. Related information

The following documents are related to material in this guide:

- Advanced Configuration and Power Interface (ACPI) Specification
- Arm Base Boot Requirements specification
- Arm PCI Configuration Space Access Firmware Interface
- Project Cassini
- Arm SystemReady SR
- Arm SystemReady ES
- UEFI Self-Certification Test

12. Next steps

In this guide, you learned how to integrate SystemReady SR/ES systems, how to develop and build the firmware, and how to test SystemReady SR/ES using a RD-N2 FVP and Raspberry Pi 4.

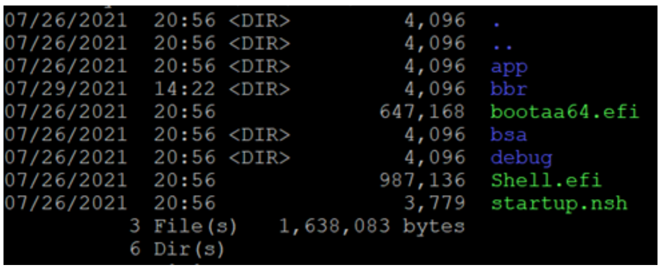
After reading this guide, you can go to the Arm SystemReady Certification Program site for more information about certification registration.

Appendix A Running ACS tests manually

To run ACS tests manually, press Esc after the UEFI shell loads. Then, navigate to the `EFI/BOOT` folder on the ACS USB drive partition.

The folder contents are shown in the following screenshot:

Figure A-1: EFI/BOOT folder contents



```
07/26/2021  20:56 <DIR>          4,096  .
07/26/2021  20:56 <DIR>          4,096  ..
07/26/2021  20:56 <DIR>          4,096  app
07/29/2021  14:22 <DIR>          4,096  bbr
07/26/2021  20:56              647,168  bootaa64.efi
07/26/2021  20:56 <DIR>          4,096  bsa
07/26/2021  20:56 <DIR>          4,096  debug
07/26/2021  20:56              987,136  Shell.efi
07/26/2021  20:56                3,779  startup.nsh
          3 File(s)   1,638,083 bytes
          6 Dir(s)
```

In this directory, the `bbr` folder contains the UEFU Self-Certification Test and the bsa folder has a UEFI shell application for BSA and SBSA compliance. For more information, see bsa-acs and sbsa-acs.

To run the bsa and sbsa test, go to the folder and start the application using the following command:

```
FS0:\efi\boot\bsa\> bsa.efi
```

or

```
FS0:\efi\boot\bsa\sbsa\> sbsa.efi
```

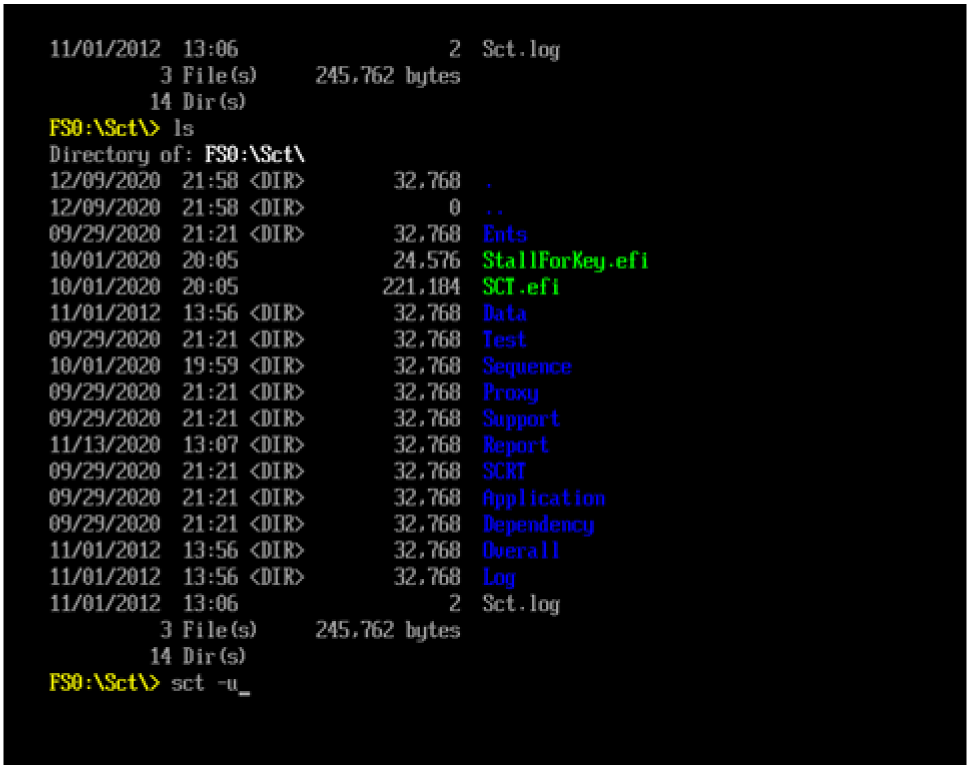
For a list of application parameters, refer to the Arm BSA Compliance User Guide and Arm SBSA Compliance User Guide.

To run the same SCT command as the automated process, that is to run all the SCT SBBR tests, go to the folder and start the application using the following command:

```
FS0:\efi\boot\bbr\sct\> SCT.efi -s SBBR.seq
```

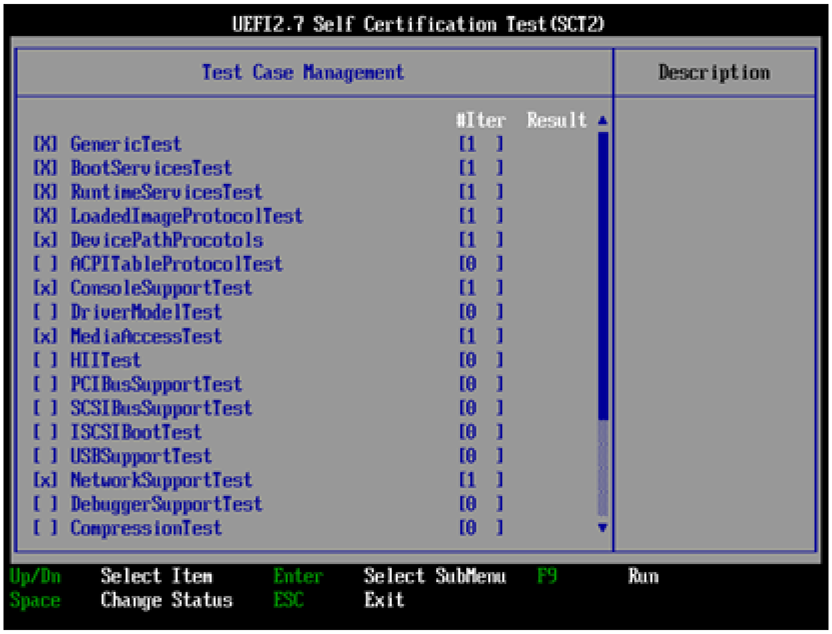
To run specific SCT test cases, you can start SCT with a GUI by passing `-u` as a parameter, as shown in the following screenshot:

Figure A-2: Start SCT with a GUI



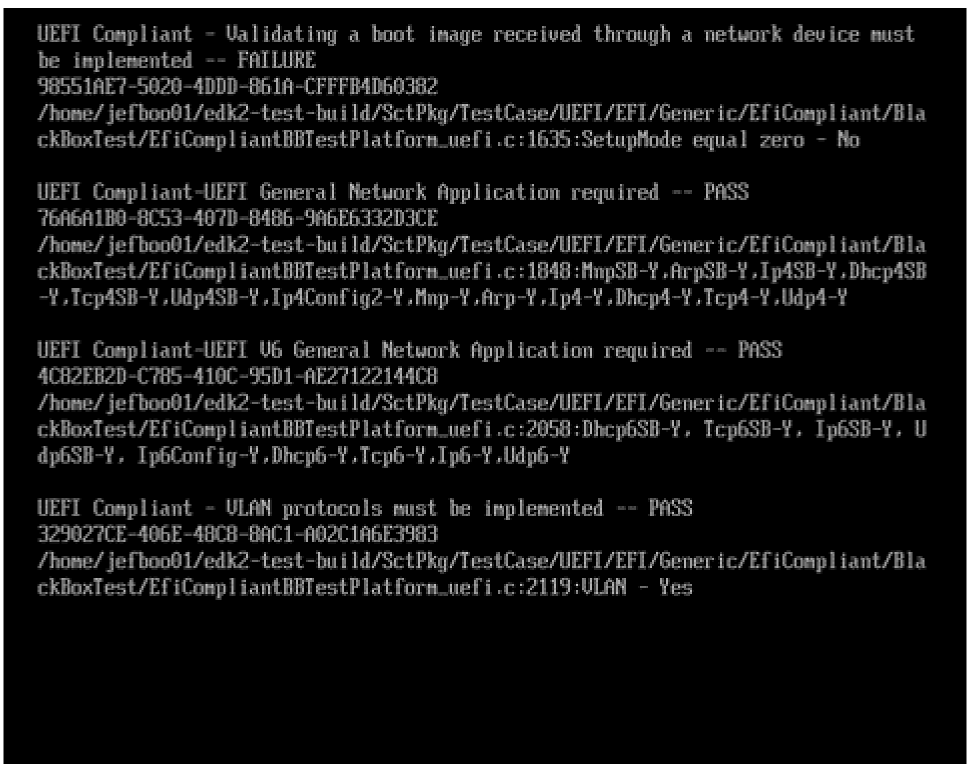
1. Press F5 to select tests manually. Press Enter.
2. View, add, or remove tests in the Test Case Management menu, as shown in the following screenshot:

Figure A-3: Test Case Management menu



3. Press F9 to run SCT, as shown in the following screenshot:

Figure A-4: SCT screen



> **Note**
>
> Sometimes SCT can hang in the process of self-reset. In this case, power off the system then power it on. The tests will not be reset. During the next boot if the same USB drive with SCT has been selected as the boot device, the test will continue. Follow the steps outlined in Boot order to ensure the USB drive is the first boot option.

If you need to build your own image, ACS tests are open source and can be downloaded from SystemReady ACS. Read the documentation in this repository to learn how to build and construct test images.