



SystemReady SR and ES Test & Integration Guide

3.0

Non-Confidential

Copyright © 2020, 2023 Arm Limited (or its affiliates).
All rights reserved.

Issue 01

102677_0300_01_en



SystemReady SR and ES Test & Integration Guide

Copyright © 2020, 2023 Arm Limited (or its affiliates). All rights reserved.

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
0300-01	25 October 2023	Non-Confidential	Integrated SR ACS requirements into ACS chapter

Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

THIS DOCUMENT IS PROVIDED “AS IS”. ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND

REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm’s customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its affiliates) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm’s trademark usage guidelines at <https://www.arm.com/company/policies/trademarks>.

Copyright © 2020, 2023 Arm Limited (or its affiliates). All rights reserved.

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. Terms and abbreviations.....	7
2. SR/ES Test and Integration Overview.....	8
3. Set up the Raspberry Pi 4.....	9
3.1 Set up the terminal.....	11
3.2 Format the SD drive.....	11
3.3 Update the EEPROM.....	15
3.4 Install UEFI.....	15
3.5 Configure UEFI.....	17
3.6 Troubleshooting UEFI.....	18
3.7 Set UEFI variables.....	20
3.8 Set the system table selection.....	21
3.9 Set the console preference.....	21
3.10 Limit RAM to 3GB.....	22
4. Set up the RD-N2 FVP.....	24
4.1 Set up the host machine and download the software stack.....	24
4.2 Download the RD-N2 FVP.....	24
4.3 Build the software stack and run the FVP.....	25
5. Preparation.....	26
5.1 Install and boot requirements.....	26
5.2 Prepare the OS installer media.....	26
5.3 Boot order.....	29
6. Install Windows PE.....	31
6.1 Download and run Windows ADK and WinPE.....	31
6.2 Create an ISO file.....	33
6.3 Install to a USB drive.....	34
6.4 Other Boot Configuration Data settings.....	34
6.5 Boot WinPE.....	35
7. ACS.....	37

7.1 ACS overview.....	37
7.2 BSA-ACS and SBSA-ACS.....	38
7.3 BBR-ACS.....	39
7.4 ACS prerequisites.....	39
7.5 Set up the test environment.....	39
7.6 Run the tests.....	40
7.7 Run tests in automated mode.....	42
7.8 Run tests in normal mode.....	45
7.9 Review the ACS test result logs.....	45
8. Debugging commands.....	47
9. Advanced Configuration and Power Interface.....	48
9.1 Example: Thermal zone.....	49
9.2 Example: Fan cooling device.....	50
9.3 Example: USB XHCI and PCIe.....	52
9.4 Example: UART.....	54
9.5 Example: Debug port.....	55
9.6 Example: Power button.....	56
9.7 Example: PCIe ECAM.....	57
9.8 ACPI integration recommendations.....	58
10. SMBIOS requirements.....	62
10.1 SMBIOS integration.....	62
10.2 Platform driver.....	63
10.3 System Management BIOS framework.....	64
11. UEFI requirements.....	66
12. Related information.....	67
13. Next steps.....	68
A. Running ACS tests manually.....	69

1. Terms and abbreviations

This document uses the following terms and abbreviations.

UEFI

Unified Extensible Firmware Interface

EDK2

EFI Development Kit 2

ACPI

Advanced Configuration and Power Interface

ASL

ACPI Source Language

AML

ACPI Machine Language

SMBIOS

System Management BIOS

PXE

Preboot Execution Environment

USAP

USB Attached SCSI Protocol

ACS

Architecture Compliance Suite

BSA

Base System Architecture

SBSA

Server Base System Architecture

BBR

Base Boot Requirement

2. SR/ES Test and Integration Overview

This guide describes 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 the SystemReady certification test suites
- About Advanced Configuration and Power Interface (ACPI) power management and System Management BIOS (SMBIOS) integration



This guide describes both SystemReady SR and ES systems. The only difference is that SBSA only applies to SystemReady SR.

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

3. Set up the Raspberry Pi 4

This section describes using a Raspberry Pi 4 to demonstrate how to build a SystemReady ES compliant platform.

To set up the Raspberry Pi, you 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 Preboot eXecution Environment (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 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. You can use thumb drives and drive enclosures. We recommend a UASP 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. You can setup both types of connection 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 need to use three of the wires.

Figure 3-1 shows how to connect the serial adapter to your Raspberry Pi:

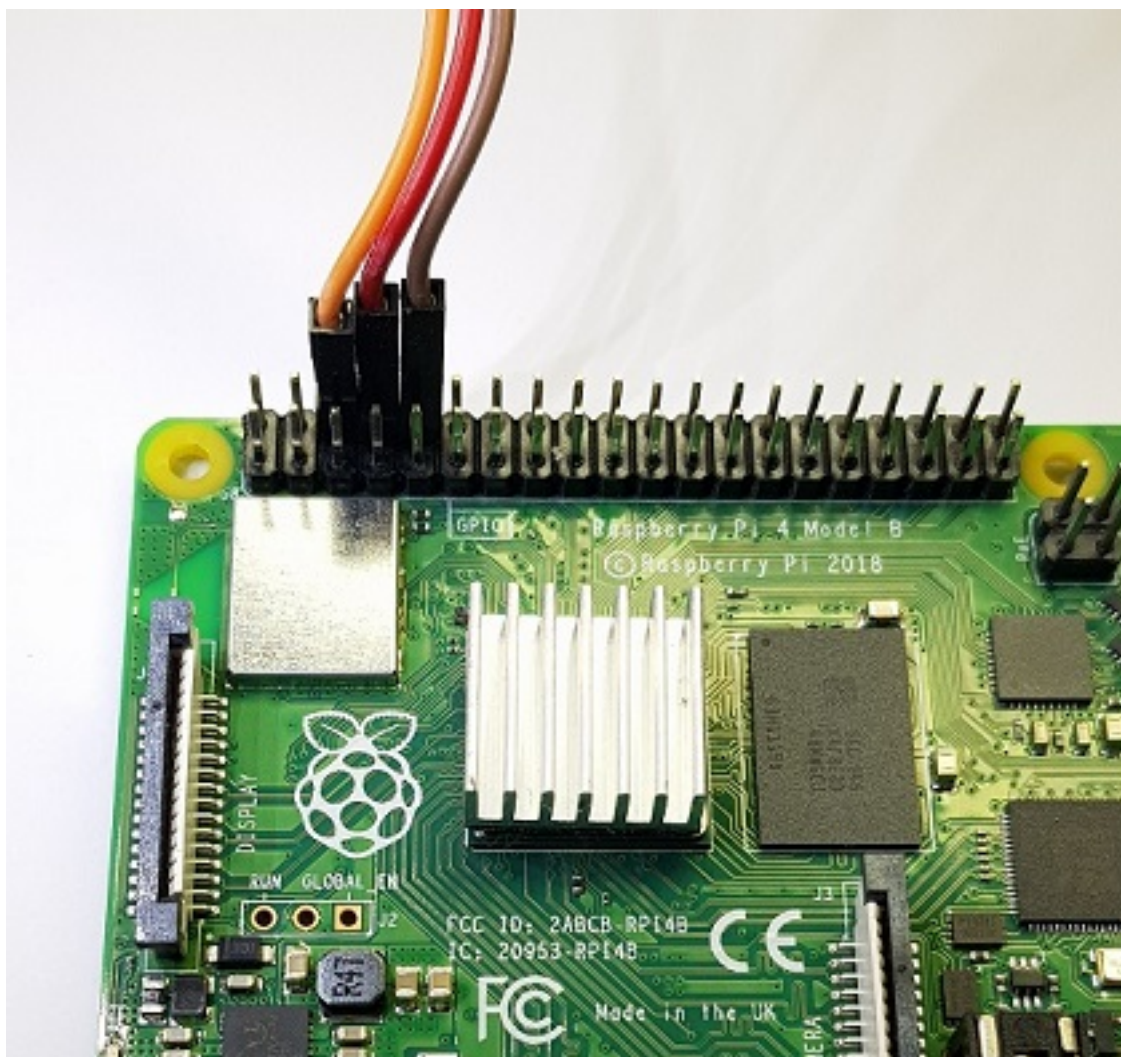
Figure 3-1: Raspberry Pi serial adapter connections

Table 3-1 shows the connection details:

Table 3-1: Connection Details

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.

3.1 Set up the terminal

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

Table 3-2 shows the configuration required, and the following text describes how to set up your connection with PuTTY:

Table 3-2: PuTTY Configuration

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

1. On the **Session** configuration panel in PuTTY, select **Serial** from the **Connection type** options.
2. Use the **Serial line** and **Speed** options to specify which serial line to use and the Baud rate to use to transfer data.
3. 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 is `/dev/ttyUSB0`. If you have more than one serial device, use a tool such as `dmseg` to check `tttyUSB<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
```

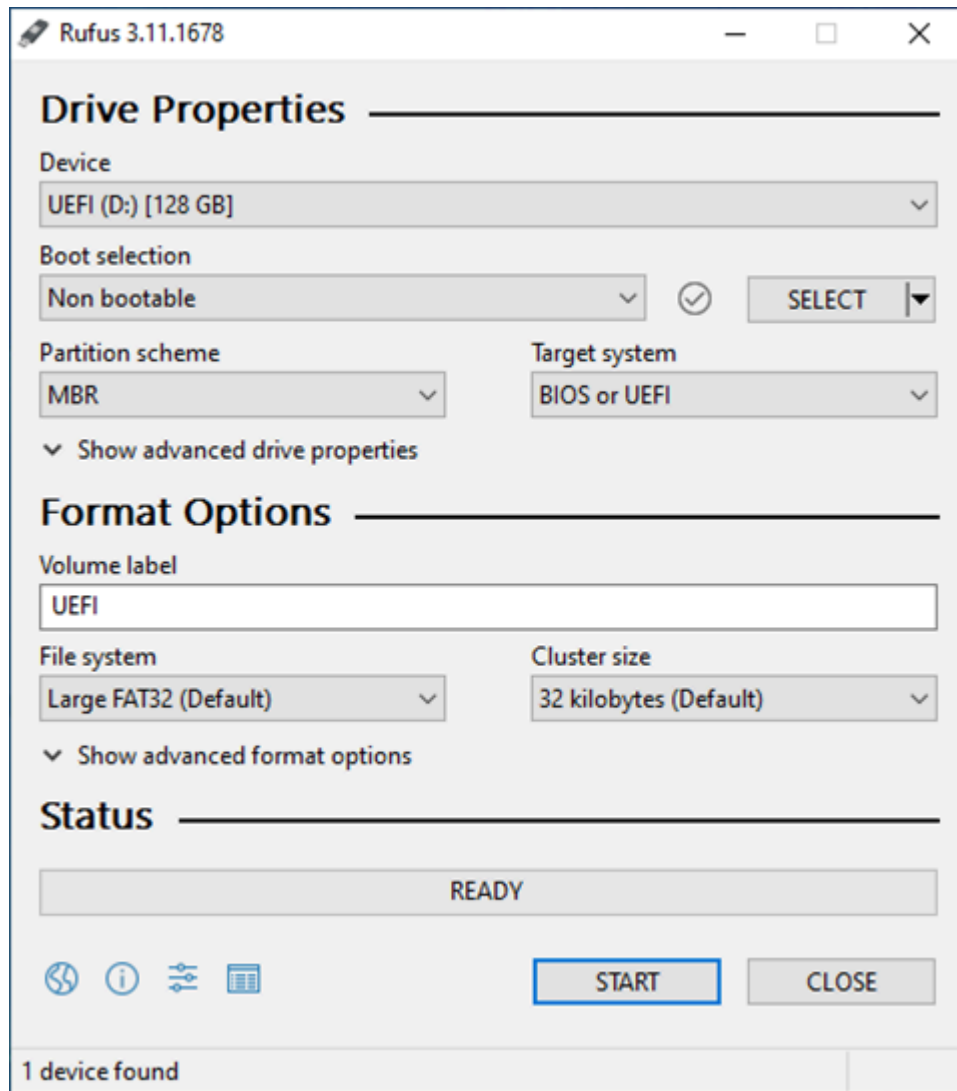
For more information and troubleshooting, see [Using a console cable with Raspberry Pi](#).

3.2 Format the SD drive

For Raspberry Pi 4, you can format the SD drive in Large FAT16 or Large FAT32 for updating the EEPROM and storing the UEFI firmware.

To format the SD drive on Windows, use [Rufus](#) and the following procedure:

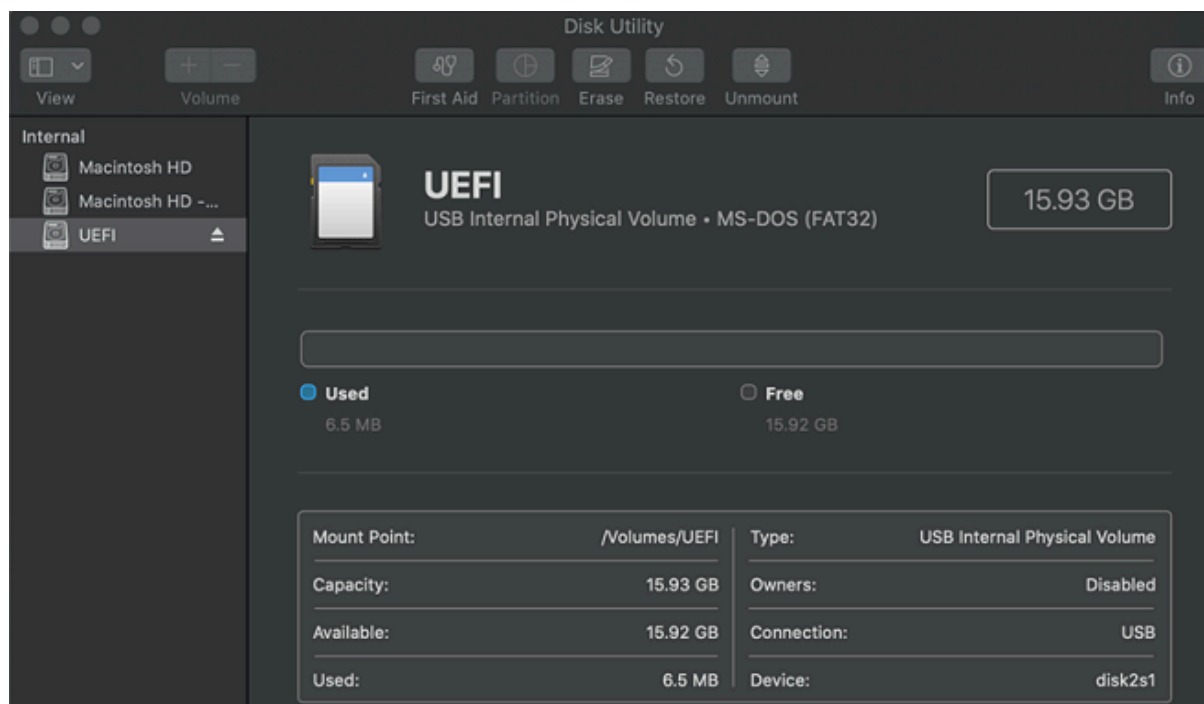
1. In Rufus, select your device then select **Non bootable** from the **Boot selection** menu. Ensure the file system type is Large FAT16 or Large FAT32. Figure 3-2 shows the file system type:

Figure 3-2: Rufus format options

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

To format the drive on Mac OS:

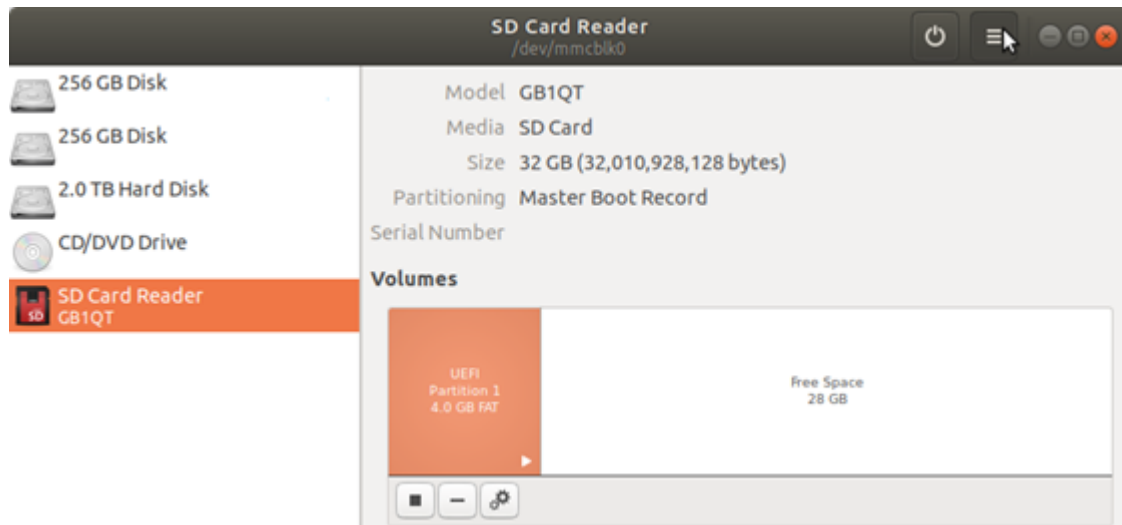
1. Open Disk Utility and select your SD card in the list of drives as shown in Figure 3-3:

Figure 3-3: Disk Utility window on Mac

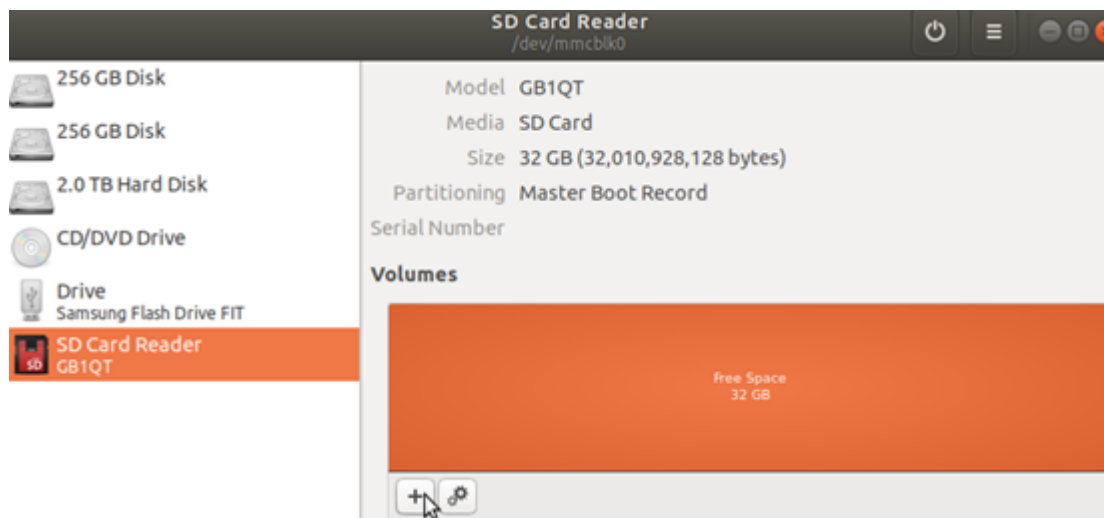
2. Click **Erase** to format the drive.
3. In the format list, select MS-DOS (FAT).

To format the drive on Linux:

1. Use either graphical or command-line instructions. When using graphical instructions, open Disks and select your SD card.
2. Figure 3-4 shows where to click on the bars at the top of the window.:

Figure 3-4: 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. Figure 3-5 shows where to click **+** to add a partition:

Figure 3-5: 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**.

3.3 Update the EEPROM

To update the EEPROM:

1. Ensure the Raspberry Pi 4 is running the latest firmware on the EEPROM.
2. Download the latest version of `rpi-eeeprom` from [RPI eeeprom github](#) and use this tool to update the boot EEPROM.
3. Unzip the contents of `rpi-boot-eeeprom-recovery` to a blank, FAT formatted SD-SDCARD.
4. Power off the Raspberry Pi 4.
5. Insert the SD card.
6. Power on the Raspberry Pi 4 and wait 10 seconds.

The green LED light blinks rapidly to show success. Otherwise, an error pattern is displayed.

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

3.4 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.



To boot from USB or ESP, you need the latest version of firmware on 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 downloaded 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 shows the embedded bootloader reading the data. The Raspberry Pi 4 logo appears 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` is found in the `RPi4/Build/<BUILD_TARGET>/FV` folder.

7. Follow the steps in the “Bootting the firmware section” in [Raspberry Pi 4 Platform](#) to prepare a bootable SD card.

3.5 Configure UEFI

Figure 3-6 shows the boot into the UEFI shell by pressing F1 during the boot process:

Figure 3-6: UEFI boot screen

```
UEFI Interactive Shell v2.2
EDK II
UEFI v2.70 (https://github.com/pftf/RPi4, 0x00010000)
Mapping table
FS0: Alias(s):HD0b::BLK1:
    VenHw(100C2CFA-B586-4198-9B4C-1683D195B1DA)/HD(1,MBR,0x36B2E766,0x800,
0x773800)
BLK0: Alias(s):
    VenHw(100C2CFA-B586-4198-9B4C-1683D195B1DA)
Press ESC in 5 seconds to skip startup.nsh or any other key to continue.
Shell> _
```

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

Figure 3-7: UEFI menu

```
Raspberry Pi 4 Model B
BCM2711 (ARM Cortex-A72)          1.50 GHz
UEFI Firmware v1.19              3072 MB RAM

Select Language      <English>      This selection will
                                     take you to the
                                     Device Manager
▶ Device Manager
▶ Boot Manager
▶ Boot Maintenance Manager

Continue
Reset

F1=Move Highlight      <Enter>=Select Entry
```

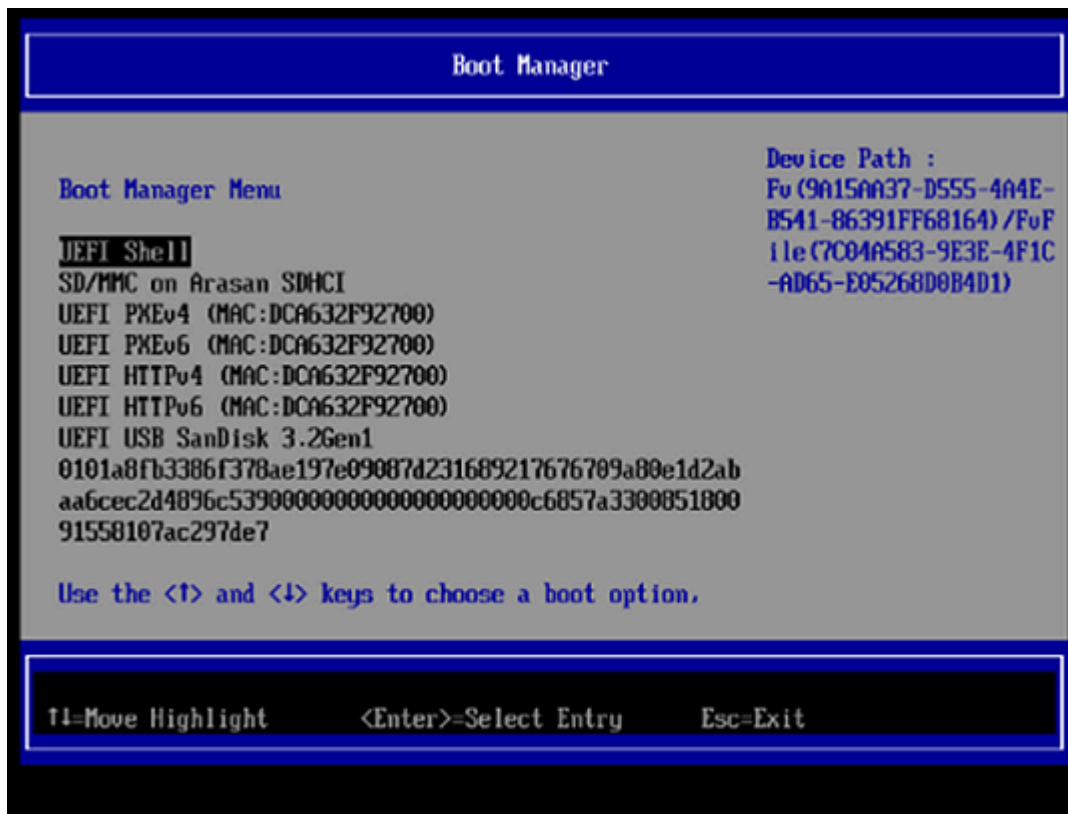
In this menu, you can change device settings and manually boot the device using Boot Manager.

3.6 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. Figure 3-8 shows the UEFI Shell option:

Figure 3-8: UEFI Shell



3. Use the `map` command to see if a storage device is mounted. Figure 3-9 shows an SD card is mounted as FSO:

Figure 3-9: 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:
    PciRoot(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:
    PciRoot(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) :
    PciRoot(0x0)/Pci(0x0,0x0)/Pci(0x0,0x0)/USB(0x2,0x0)
Shell> _

```

4. Change the directory to FS0 by typing `fs0` at the command prompt.

Table 3-3 shows UEFI Shell commands which are helpful for debugging:

Table 3-3: Uefi Shell Commands

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

See the [UEFI Shell Specification](#) for more details. The Shell commands section provides a list of shell commands, descriptions, and examples.

3.7 Set UEFI variables

You can view and change the Raspberry Pi 4 UEFI configuration settings 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
```

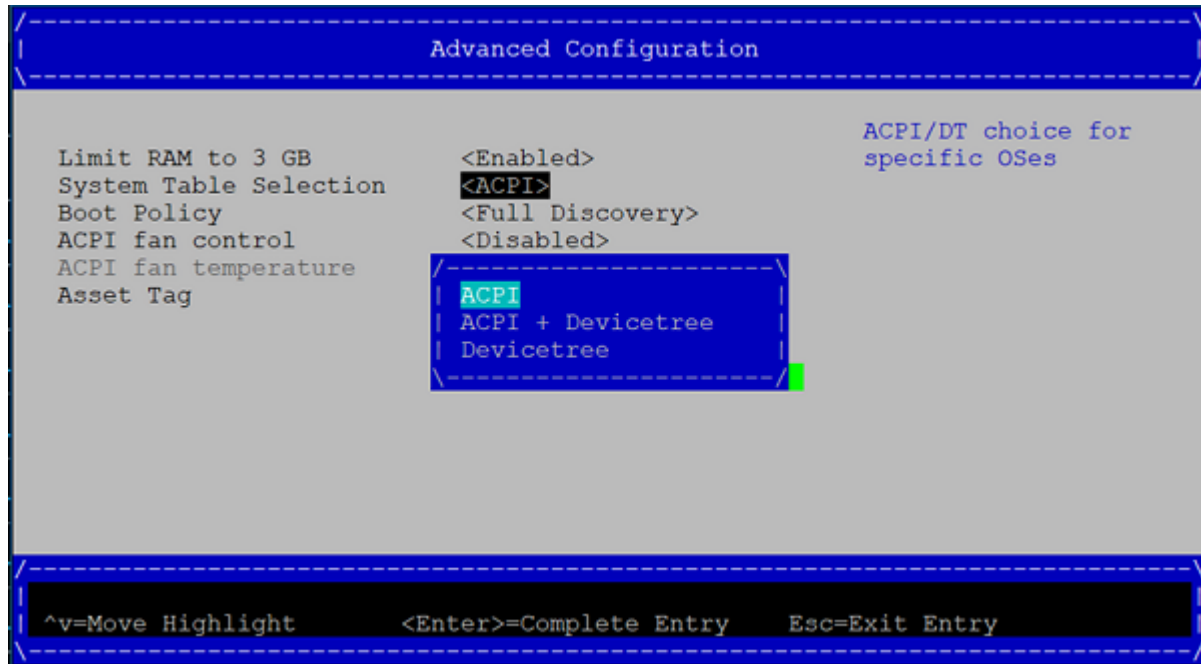
Change the Asset Tag to the string ASSET-TAG-123:

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

3.8 Set the system table selection

In the **Advanced Configuration** menu, select ACPI as shown in Figure 3-10:

Figure 3-10: ACPI option



3.9 Set the console preference

Linux uses the `/chosen/stdout-path` DT property or the SPCR ACPI table to show 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 which make the graphical console work.

To select the graphical console:

1. Open **Device Manager** in the UEFI menu
2. Select **Console Preference Selection**. Figure 3-11 shows the **Console Preference Selection** option:

Figure 3-11: Console Preference Selection option

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

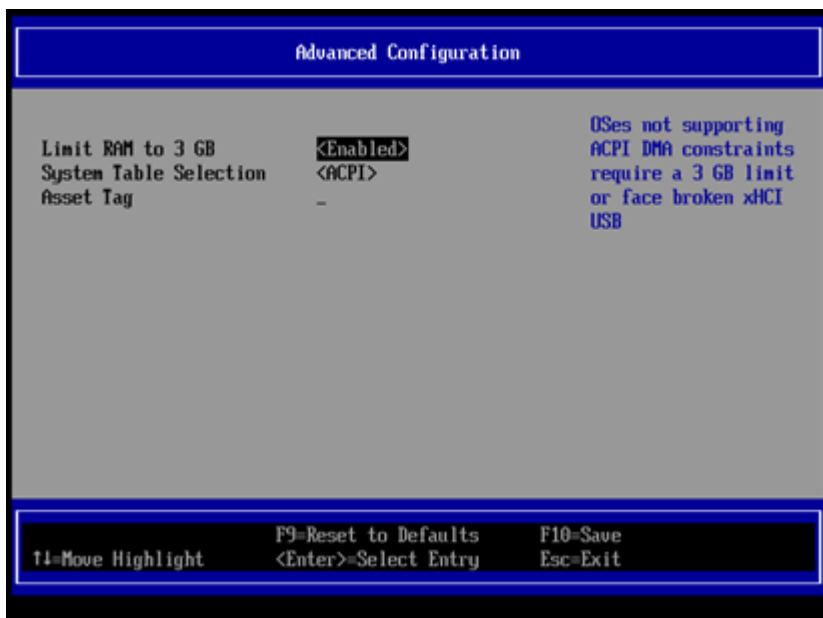


The serial console on most OSes may not work with the Graphical setting because the UEFI does not install the SPCR ACPI table. This setting must be **Serial** when running the ACS test suite because the SPCR ACPI table is mandatory for SystemReady SR/ES and is used in parts of the ACS test.

3.10 Limit RAM to 3GB

Currently, many operating systems support 3GB of RAM on the Raspberry Pi 4. To set the limit to 3GB:

1. From the UEFI menu go to **Device Manager > Raspberry Pi Configuration > Advanced Configuration**
2. Enable Limit RAM to 3GB. Figure 3-12 shows the RAM limit setting:

Figure 3-12: RAM limit enabled

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

- OpenBSD 6
- NetBSD 9
- VMWare ESXi

4. Set up the RD-N2 FVP

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

4.1 Set up the host machine and download the software stack

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

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 32 GB, follow the [instructions for using the swap file](#) to enable virtual memory.

You need a display manager 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: <https://itsfoss.com/install-gui-ubuntu-server/> to install the `lightdm` display manager. Then install a remote desktop tool such as `xrdp`. An alternative is to use MobaXterm.

4.2 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.

4.3 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](#)



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

5. Preparation

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

5.1 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 versions of an operating system or installation guides that are specifically designed for Raspberry Pi. SystemReady SR/ES does not use special images and guides, and ensures your images are suitable for AARCH64.

5.2 Prepare the OS installer media

Before you prepare the installer media, download the AARCH64 installer image for your OS. Table 5-1 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.

Table 5-1: Operating System Download Links

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



Entry in this list does not indicate that the OS is officially supported on the system. Consult the system and OS vendors for official support.

You can use the disk tools listed depending on your to set up a USB storage device with an OS installer. To set up the device:

1. Insert the USB drive then use a disk tool to restore a disk image to the drive.
 - [Rufus](#) or [balenaEtcher](#) on Windows
 - 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
```

2. If installation problems occur, for example a system hang in the OS bootloader, clean the media device 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:

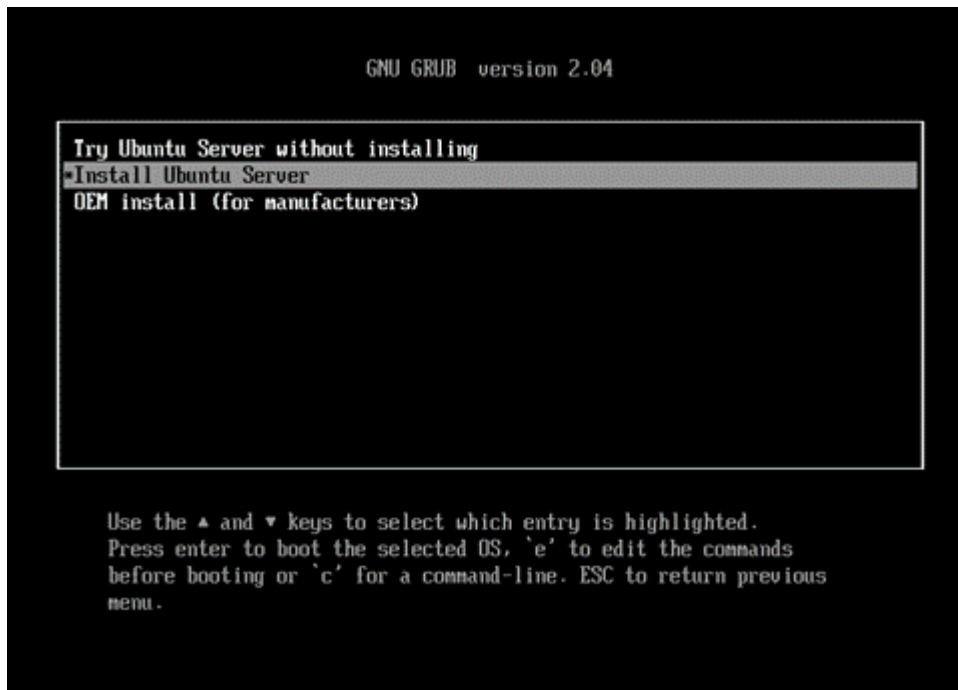
First, only clean the first megabyte. In most cases, this fixes the issue:

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

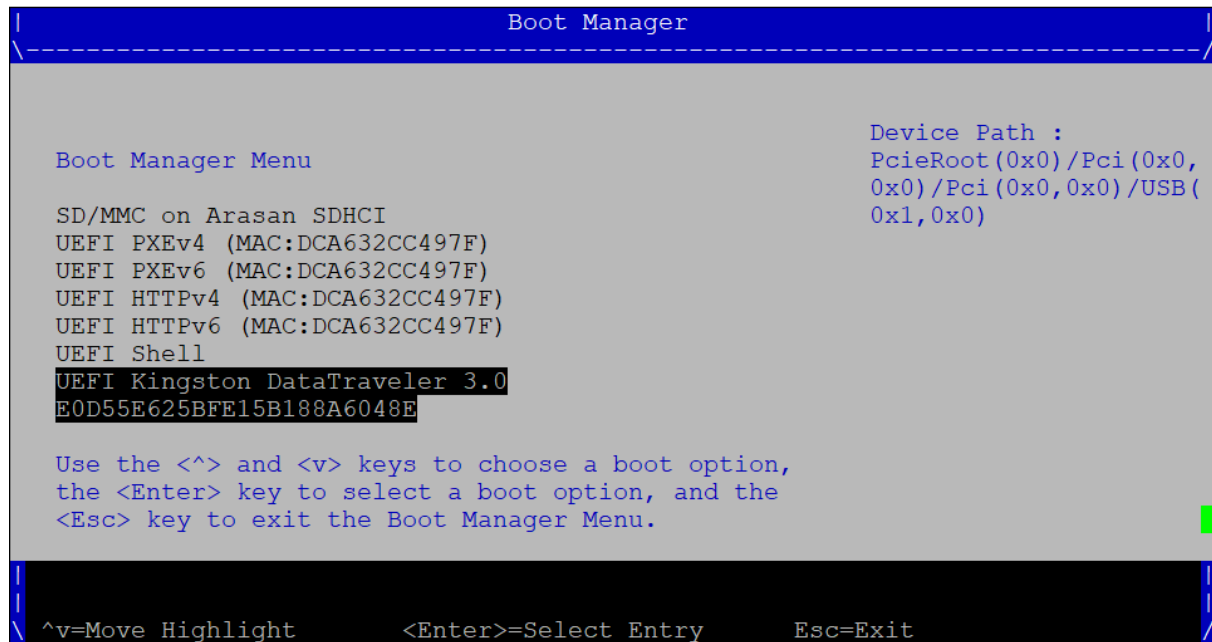
If the installation issue persists, perform a deep clean. This might take several hours:

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

3. After you create the install media, insert the drive into the USB 3.0 (blue) USB ports on the system.
4. If the USB drive is the first boot option, UEFI discovers and automatically boots into the installer media. Figure 6-1 shows the OS bootloader:

Figure 5-1: GRUB loader

5. If the first boot option is UEFI shell or PXE boot, press Esc to interrupt the boot process.
6. In the UEFI menu, go to **Boot Manager**
7. Choose the install media (USB drive).
8. Figure 6-2 shows the USB key which is called UEFI Kingston DataTraveler 3.0:

Figure 5-2: USB key in Boot Manager

9. Press **Enter** to launch the OS bootloader.
10. Now, you can follow the installation instructions provided by your OS. For example, see [Ubuntu](#) or [Fedora](#).
11. Install the operating system to a storage device, not the installer media or the SD card that you used to store your firmware.



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](#).

5.3 Boot order

When UEFI variables are not supported at runtime, the OS might not be able to create a boot entry. 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. The installed OS device must be at the top of the list. If it is not, highlight the device and press + until it is at the top of the list.
4. Press **Enter**, then save and exit.

6. 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 describes using Windows ADK version 2004.

This section describes the following steps:

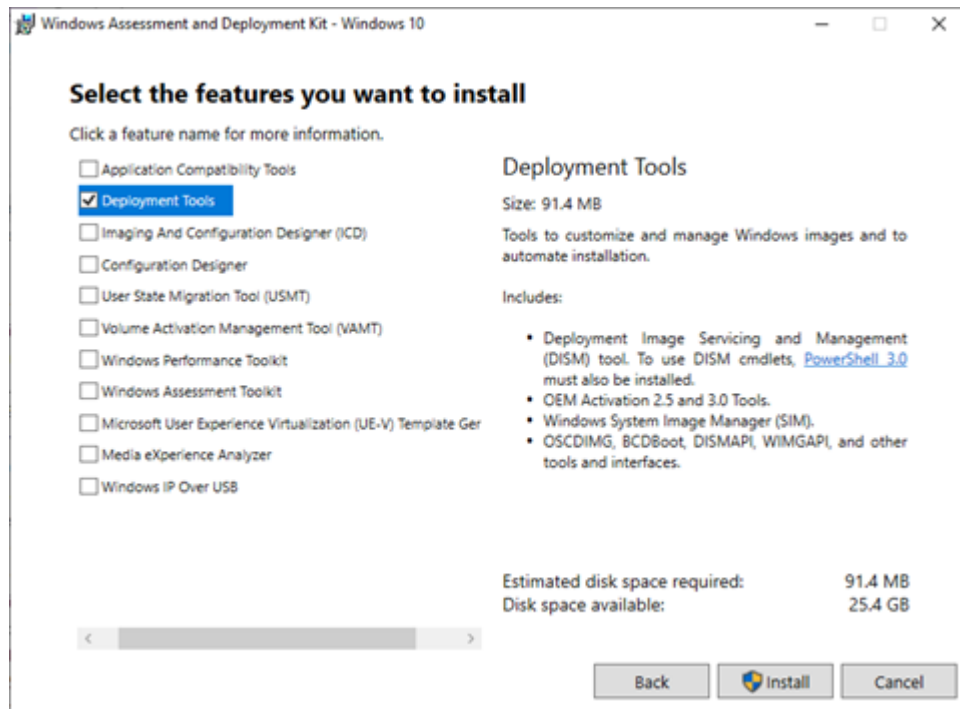
1. Build the ISO and USB key on a device running Windows 10
2. Install ADK on Windows 10
3. Build the WinPE image
4. Set up QEMU to install WinPE

6.1 Download and run Windows ADK and WinPE

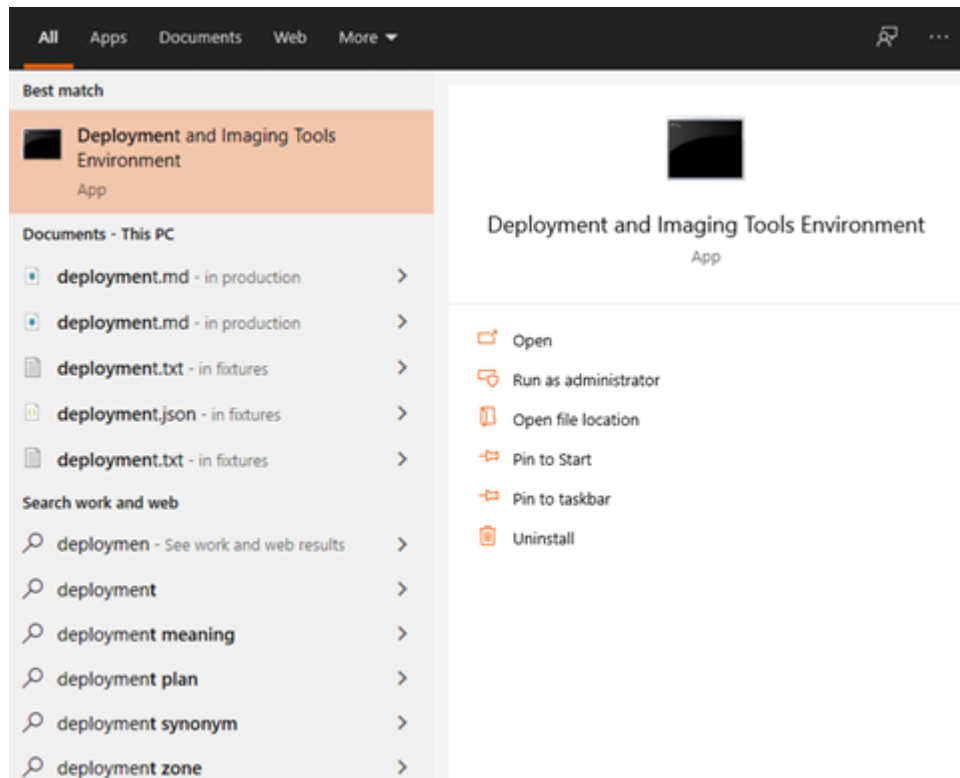
Microsoft does not provide an .iso file for WinPE. Instead, download the Windows ADK and Windows PE at <https://docs.microsoft.com/en-us/windows-hardware/get-started/adk-install> to build one yourself.

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. Figure 6-1 shows enabling the **Deployment Tools** feature to build a WinPE image:

Figure 6-1: 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. Figure 6-2 shows how to start the Deployment and Imaging Tools Environment app window with administrator privileges:

Figure 6-2: 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:

```
> copyype arm64 C:\WinPE_arm64
```

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

6.2 Create an ISO file

To create an ISO file:

1. Change the boot parameters before creating the media.
2. 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.

3. 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
```

4. Use the following command to create the ISO image.

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

6.3 Install to a USB drive

To prepare the USB drive, use the following commands:

1. Clean a selected USB drive, create a primary partition, format it to FAT32, assign it the letter P, and label it "WINPE".

```
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
```

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

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

3. 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
```

6.4 Other Boot Configuration Data settings

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

1. 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
```

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

6.5 Boot WinPE

To boot WinPE on an Arm64 system:

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

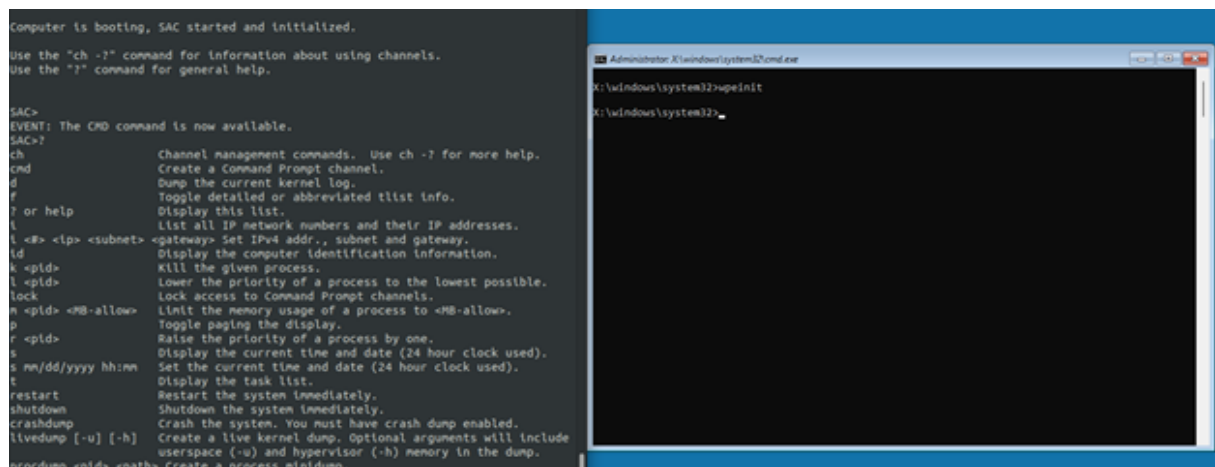
If you do not have an Arm64 system, use QEMU to emulate a system. To install QEMU and boot WinPE from an ISO file, follow these steps:

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. Figure 6-3 shows an example of the console and `cmd` window:

Figure 6-3: SAC console and cmd window





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

7. ACS

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

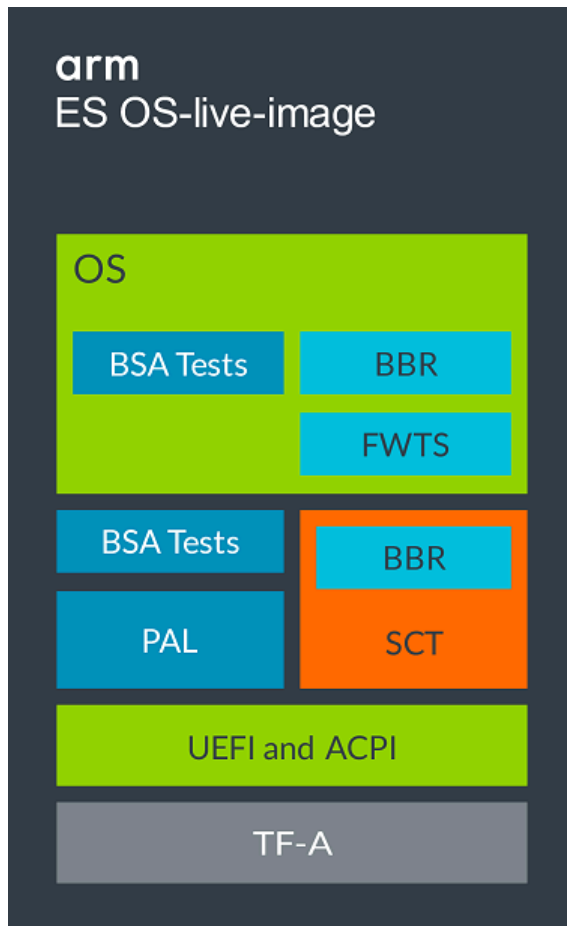
7.1 ACS overview

The Arm SystemReady Architecture Compliance Suite (ACS) is a collection of tests designed to ensure architectural compliance across various implementations and variants of the architecture. The product is delivered as source code with a build environment. It creates a bootable live OS image encompassing a series of test suites, collective referred to as the BSA, SBSA, and BBR ACS. These test suites assess compliance against the BSA, SBSA, and BBR specification for the SystemReady SR and ES. Arm recommends verifying architectural implementations against the ACS to demonstrate compliance with these specifications.

The ACS for SystemReady certification is facilitated through a live OS image, empowering basic automation for executing the BSA and BBR tests. The ACS image contains the following components for checking the requirements during boot time and runtime.

- UEFI applications that operate on a UEFI shell.
- Linux kernel that incorporates kernel modules and Firmware Test Suite (FWTS).

Figure 7-1 is a diagram illustrating the various ACS components:

Figure 7-1: ACS components

7.2 BSA-ACS and SBSA-ACS

The BSA test suites check for compliance against the [Arm Base System Architecture \(BSA\) specification](#), and the SBSA test suites check for compliance against the [Server Base System Architecture \(SBSA\) supplement specification](#). BSA-ACS test is required for both SystemReady SR and ES certifications. SBSA-ACS is only required for SystemReady SR certification. The tests are delivered through two parts:

- Tests on UEFI Shell. These tests consist of the UEFI shell command-line application `bsa.efi` and `sbsa.efi`. These tests are written on top of Validation Adaption Layers (VAL) and Platform Adaption Layers (PAL). The abstraction layers provide the tests with platform information and runtime environment to enable execution of the tests. In Arm deliveries, the VAL and PAL are written on top of UEFI.
- Tests on the Linux command line. These tests consist of the Linux command-line application `bsa` and `sbsa`, and the kernel module `bsa_acs.ko` and `sbsa_acs.ko`.

7.3 BBR-ACS

The BBR test suites check compliance against the [Arm Base Boot Requirements \(BBR\) specification](#). For SystemReady ES and SR certification, firmware is tested against the SBBR recipe of BBR.

These tests are delivered through two bodies of code:

- SBBR tests contained in UEFI Self Certification Tests (SCT) tests. UEFI implementation requirements which are tested by SCT.
- SBBR based on FWTS. The Firmware Test Suite (FWTS) is a package hosted by Canonical that provides tests for ACPI and UEFI. The FWTS tests are customized to run only UEFI tests.

7.4 ACS prerequisites

The prerequisites to run the ACS tests are as follows:

- Prepare a USB device with a minimum of 1GB of storage. This USB is used to boot and run the ACS and to store the execution results.



We recommend you use a USB disk enclosure with a fast SSD drive.

-
- Prepare the SUT (System Under Test) machine with the latest firmware loaded, a host machine for console access, then collect the results

7.5 Set up the test environment

To set up the USB device, use the following procedure:

1. Download the [prebuilt ACS image for SystemReady ES certification](#) or [prebuilt ACS image for SystemReady SR certification](#) to a local directory on Linux.

The pre-built ACS image for SystemReady ES/SR certification is available on GitHub at the following location:

ES:

```
https://github.com/ARM-software/arm-systemready/tree/main/ES/prebuilt_images/  
<release tag>/es_acs_live_image.img.xz
```

SR:

```
https://github.com/ARM-software/arm-systemready/tree/main/SR/prebuilt_images/  
<release tag>/sr_acs_live_image.img.xz
```

For information on the latest release and release tags, see the [SystemReady ES ACS README](#) or the [SystemReady SR ACS README](#).

2. Decompress and deploy the image to the USB disk.

Use a utility such as `xz` on Linux or 7-Zip on Windows to uncompress the `es_acs_live_image.img.xz` file.

3. On the Linux host machine, write the ES ACS bootable image to the USB disk using the following commands:

```
$ lsblk  
$ sudo dd if=/path/to/es_acs_live_image.img of=/dev/sdX  
$ sync
```

In this code, replace `/dev/sdX` with the name of your USB device. Use the `lsblk` command to display the USB device name.

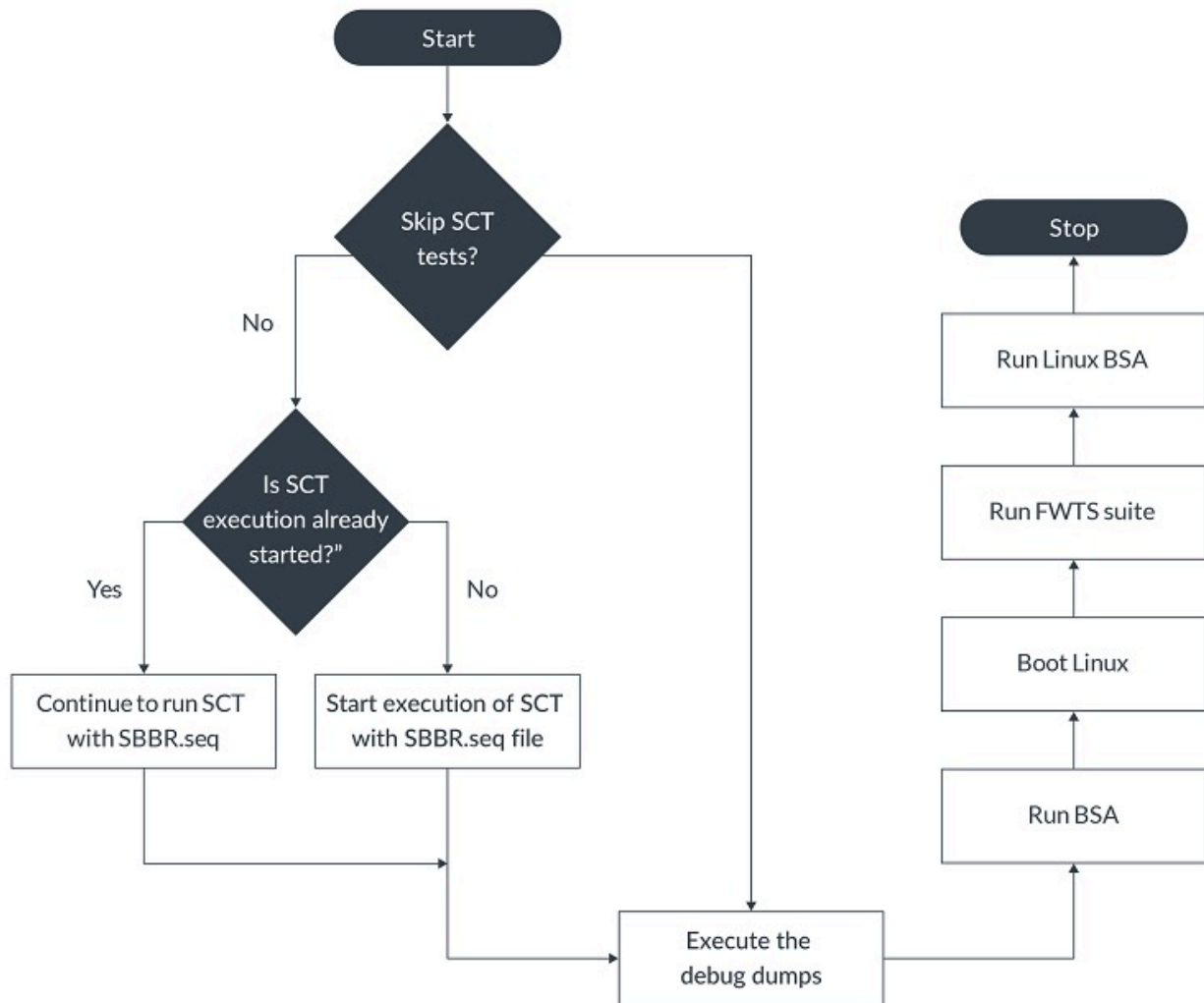
7.6 Run the tests

To execute the ACS SR/ES ACS prebuilt image, do the following:

1. Insert the ACS drive into the system.
2. Boot to the firmware setup menu.
3. Move the boot option of the ACS drive to the top of the boot order and then save the setting.
4. Reset the system.

The live image boots and runs ACS automatically.

Figure 7-2 shows the complete process of ACS execution through the ES/SR ACS live image:

Figure 7-2: Test execution process

To skip the debug and test steps shown in the diagram, press any key within five seconds.

As shown in the flowchart, there are two main modes of execution:

- Fully automated mode
- Normal mode

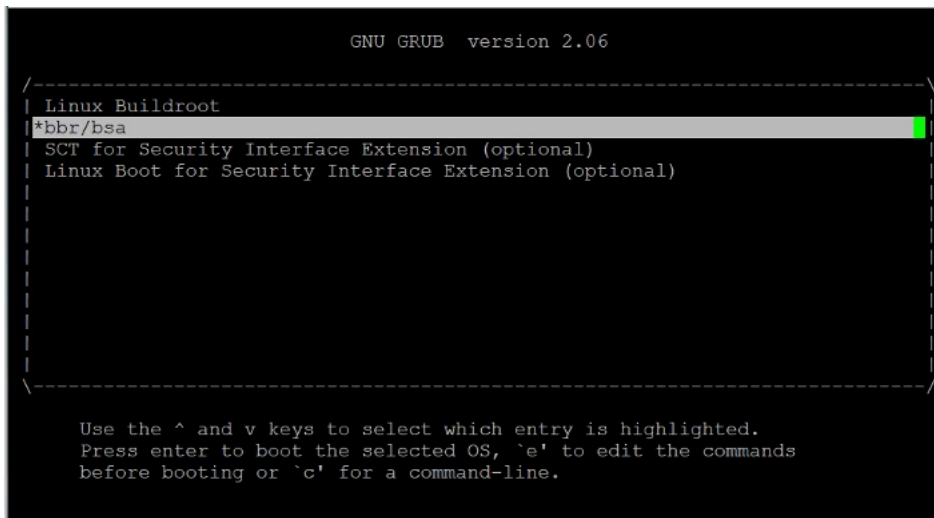
The following sections describe these modes in more detail.

7.7 Run tests in automated mode

If no option in GRUB is chosen and no tests are skipped, tests are run in fully automated mode.

Figure 7-3 shows the GRUB bootloader options screen:

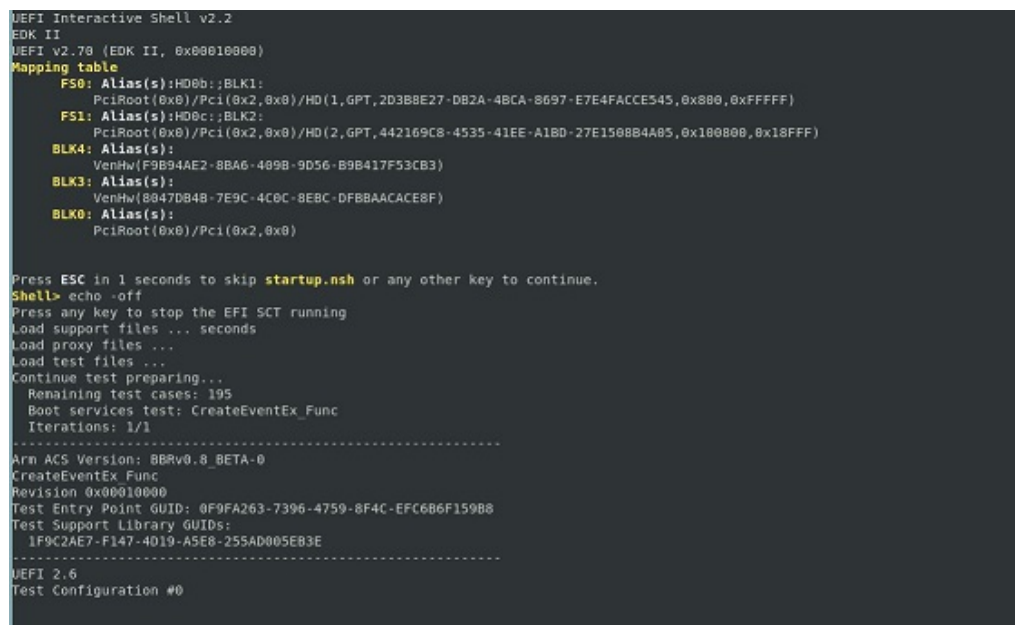
Figure 7-3: GRUB bootloader options



After a few seconds, the image executes the ACS process in the following order:

1. SCT tests:

Figure 7-4: SCT tests



2. UEFI debug dumps:

Figure 7-5: UEFI debug dumps

```

UEFI Interactive Shell v2.2
EDK II
UEFI v2.70 (EDK II, 0x00010000)
Mapping table
  FS0: Alias(s):HD0B::BLK1:
        PciRoot(0x0)/Pci(0x2,0x0)/HD(1,GPT,2D3B8E27-DB2A-4BCA-8697-E7E4FACCE545,0x800,0xFFFFF)
  FS1: Alias(s):HD0C::BLK2:
        PciRoot(0x0)/Pci(0x2,0x0)/HD(2,GPT,442169C8-4535-41EE-A1BD-27E1508B4A05,0x100800,0x10FFF)
  BLK4: Alias(s):
        VenHw(F9B94AE2-8BA6-409B-9D56-B9B417F53CB3)
  BLK3: Alias(s):
        VenHw(8047DB4B-7E9C-4C0C-8EBC-DFBBAACACE8F)
  BLK0: Alias(s):
        PciRoot(0x0)/Pci(0x2,0x0)

Press ESC in 1 seconds to skip startup.nsh or any other key to continue.
Shell> echo -off
Press any key to stop the EFI SCT running
- [ok]any key within 4 seconds
- [ok]
- [ok]
Starting UEFI Debug dump

```

3. BSA ACS:

Figure 7-6: BSA ACS

```

BSA Architecture Compliance Suite
Version 0.8

Starting tests with Print level is 3

Creating Platform Information Tables
PE_INFO: Number of PE detected      : 1
GIC_INFO: Number of GICD           : 1
GIC_INFO: Number of ITS             : 0
TIMER_INFO: Number of system timers : 0
WATCHDOG_INFO: Number of Watchdogs : 0
PCIE_INFO: Number of ECAM regions   : 1
PCIE_INFO: BDF Table : No Devices Found
SMMU_INFO: Number of SMMU CTRL      : 0
Peripheral: Num of USB controllers  : 0
Peripheral: Num of SATA controllers  : 0
Peripheral: Num of UART controllers  : 1

*** Starting PE tests ***
Operating System:
 1 : 0 PE 01: Check Arch symmetry across PE
      Skipping as num of PE is 1      : Result: -SKIPPED- 1
 2 : 0 PE 02: Check for number of PE   : Result: PASS
 3 : 0 PE 03: Check for AdvSIMD and FP support : Result: PASS
 4 : 0 PE 04: Check PE 4KB Granule Support : Result: PASS
 5 : 0 PE 05: Check HW Coherence support   : Result: PASS
 6 : 0 PE 06: Check Cryptographic extensions : Result: PASS
 7 : 0 PE 07: Check Little Endian support   : Result: PASS
 8 : 0 PE 08: Check EL1 and EL0 implementation : Result: PASS
 9 : 0 PE 09: Check for PMU and PMU counters
      Failed on PE - 0                 : Result: --FAIL-- 1
10 : 0 PE 10: Check PMU Overflow signal
      Error : Received Sync Exception
      Failed on PE - 0                 : Result: --FAIL-- 1
11 : 0 PE 11: Check num of Breakpoints & type : Result: PASS
12 : 0 PE 12: Check Synchronous Watchpoints : Result: PASS
13 : 0 PE 13: Check CRC32 instruction support : Result: PASS
14 : 0 PE 15: Check PAUTH if implementation : Result: -SKIPPED- 1
15 : 0 PE 17: Check SPE if implemented       : Result: -SKIPPED- 1
16 : 0 SEC 01: Check Speculation Restriction

```

4. FWTS tests:

Figure 7-7: FWTS tests

```

Test: UEFI secure boot test.
Test aborted.
Test: Authenticated variable tests.
  Create authenticated variable test. 1 passed
  Authenticated variable test with the same authentica.. 1 passed
  Authenticated variable test with another valid authe.. 1 passed
  Append authenticated variable test. 1 passed
  Update authenticated variable test. 1 passed
  Authenticated variable test with old authenticated v.. 1 passed
  Delete authenticated variable test. 1 passed
  Authenticated variable test with invalid modified data 1 passed
  Authenticated variable test with invalid modified ti.. 1 passed
  Authenticated variable test with different guid. 1 passed
  Set and delete authenticated variable created by dif.. 2 passed
Test: UEFI miscellaneous runtime service interface tests.
  Test for UEFI miscellaneous runtime service interfaces 1 passed, 5 skipped
  Stress test for UEFI miscellaneous runtime service i.. 1 passed, 5 skipped
  Test GetNextHighMonotonicCount with invalid NULL par.. 1 passed
  Test UEFI miscellaneous runtime services supported s.. 1 skipped
Test: UEFI variable info query.
  UEFI variable info query.
Test: Sanity check UEFI ESRT Table.
Test aborted.
Test: Sanity check for UEFI Boot Path Boot###.
  Test UEFI Boot Path Boot###. 1 skipped
Test: UEFI miscellaneous runtime service interface tests.
  Test for UEFI miscellaneous runtime service interfaces 1 passed, 5 skipped
  Stress test for UEFI miscellaneous runtime service i.. 1 passed, 5 skipped
  Test GetNextHighMonotonicCount with invalid NULL par.. 1 passed
  Test UEFI miscellaneous runtime services supported s.. 1 skipped
Test: UEFI Runtime service time interface tests.
  Test UEFI RT service get time interface. 1 passed
  Test UEFI RT service get time interface, NULL time p.. 1 passed
  Test UEFI RT service get time interface, NULL time a.. 1 passed
  Test UEFI RT service set time interface. 1 passed
  Test UEFI RT service set time interface, invalid yea.. 1 passed
  Test UEFI RT service set time interface, invalid yea.. 1 passed
  Test UEFI RT service set time interface, invalid mon.. 1 passed
  Test UEFI RT service set time interface, invalid mon.. 1 passed

```

5. Linux BSA test:

Figure 7-8: Linux BSA test

```

Test UEFI RT variable services supported status. 1 skipped
Running Linux BSA tests
[ 21.492908] init BSA Driver
[ 21.504154] PE_INFO: Number of PE detected : 4
[ 21.509659] PCIE_INFO: Number of ECAM regions : 0
[ 21.515133] Peripheral: Num of USB controllers : 0
[ 21.520577] Peripheral: Num of SATA controllers : 0
[ 21.525996] Peripheral: Num of UART controllers : 0
[ 21.531416] DMA_INFO: Number of DMA CTRL in PCIe : 0
[ 21.536965] SMMU_INFO: Number of SMMU CTRL : 0
[ 21.542545]
[ 21.542545] Operating System:
[ 21.547007] 605 : B_PER_09,B_PER_10: Memory Attribute of DMA
[ 21.547007] No DMA controllers detected... : Result: -SKIPPED- 3
[ 21.559993]
[ 21.559993] *** One or more tests have Failed/Skipped.***
[ 21.567587]
[ 21.567587] Operating System:
[ 21.572046] 104 : B_MEM_03,B_MEM_04,B_MEM_06: Addressability
[ 21.572046] Skip as No peripherals detected : Result: -SKIPPED- 1
[ 21.585030]
[ 21.585030] *** One or more tests have Failed/Skipped.***
[ 21.592622]
[ 21.592622] *** No ECAM region found, Skipping PCIE tests ***
[ 21.600384]
[ 21.600384] -----
[ 21.600384] Total Tests Run = 2, Tests Passed = 0, Tests Failed = 0
[ 21.600384] -----

```

After these tests are executed, the control returns to a Linux prompt.

7.8 Run tests in normal mode

When the image boots, choose one of the following GRUB options to specify the test automation:

- Linux BusyBox to boot Linux and execute FWTS and Linux BSA
- BBR or BSA to execute the tests in the same sequence as fully automated mode



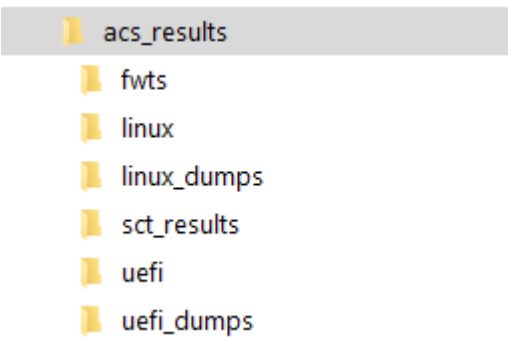
You can also skip individual test stages by pressing a key at the appropriate point.

7.9 Review the ACS test result logs

The logs are stored in a separate partition within the image called `acs-results`.

Figure 7-9 shows the logs directory structure:

Figure 7-9: ACS results directory



After the tests finish, the `acs_results` partition is mounted on `/mnt`.

1. Navigate to `acs_results` to view the logs.
2. Extract the logs from the USB disk to view the logs on the host machine later.
3. After mounting the `acs_results` directory, check that the logs shown in Table 7-1 were generated:

Table 7-1: Log Details

Log number	ACS	Path	Running time	Notes
1	BSA (UEFI)	acs_results/uefi/BsaResults.log	Less than 2 minutes	—

Log number	ACS	Path	Running time	Notes
2	BSA (Linux)	acs_results/uefi/BsaResults.log acs_results/linux/BsaResultsKernel.log	Less than 2 minutes	—
3	SCT	acs_results/sct_results/Summary.log	1-6 hours Note: To execute SCT tests faster, use an SSD in a USB enclosure	Summary.log contains the summary of all tests run. Individual SCT test suite logs are in the same path.
4	FWTS	acs_results/fwts/results.log	Less than 2 minutes	—
5	Debug Dumps	acs_results/linux_dumps acs_results/uefi_dumps	Less than 2 minutes	Contains output from commands including: acpiview, smbiosview, lspci, drivers, devices, and memmap.

Post testing, mount the `acs_result` partition to view or extract the logs. Ensure all necessary logs are generated.

8. Debugging commands

Linux commands are helpful for debugging:

Table 8-1: Command Details

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

9. 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

9.1 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.

Table 9-1 defines a simple thermal zone TZ00. TZ00 specifies the following methods:

Table 9-1: ACPI Methods

Method	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:

Table 9-2: ACPI Objects

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 Table 9-1 and Table 9-2:

```
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 })
    }
}

```

9.2 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 Table 9-4.

Tables 9-3 lists PFAN fan power resource methods:

Table 9-3: PFAN Fan Power Resource methods

Method	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

Table 9-4 lists methods and objects for the fan device:

Table 9-4: Fan Device Methods and Objects

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)

Table 9-5 lists methods and objects for the Active Cooling point:

Table 9-5: Active Cooling Point Methods and Objects

Object	Description
_AC0	Returns the temperature trip point at which OSPM must start or stop Active cooling

Object	Description
_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:

Table 9-6: Hat 4 Methods

Method	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.

9.3 Example: USB XHCI and PCIe

If a PCIe controller is present and visible by the operating system, you must use an MCFG 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, ResourceTemplate() {
            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)
        CreateQwordField (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)
    }
}
```

9.4 Example: UART

The system can present Arm SBSA Generic UART and 16550 UART devices. You can describe the devices with Serial Console Redirection (SPCR).

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
    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
EFI_ACPI_RESERVED_DWORD
};
Reserved3;
```

9.5 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.

9.6 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.

Table 9-7 shows the Generic Event Device objects:

Table 9-7: Generic Event Device Objects

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

Table 9-8 lists the Generic Event Device methods:

Table 9-8: 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

Table 9-9 shows the power button objects:

Table 9-9: Power Button Objects

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

Table 9-10 lists the power button methods:

Table 9-10: 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, set up GPIO pin 5 to trigger an interrupt when a rising edge is detected. Then, in the _EVT method, 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)
    }
}
```

9.7 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, take a different approach.

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. You can use this interface as an 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, see 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 SMCCCv1.1 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 is in [Arm Trusted Firmware](#). Arm Trusted Firmware 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, 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, 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 is in the NetBSD. NetBSD implements [pci_smccc_call\(\)](#), which uses a Secure Monitor Call to request a PCI Configure access service to a platform firmware running in EL3. With PCI_SMCCC enabled, the 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.

You can build and run NetBSD, Arm Trusted Firmware and EDK2 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 might implement this interface.

9.8 ACPI integration recommendations

You can implement ACPI tables 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, .asl 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, you can 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, see 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.



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.

10. 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

10.1 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;
```

10.2 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](#).

10.3 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>.

11. 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.

12. 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](#)

13. 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.

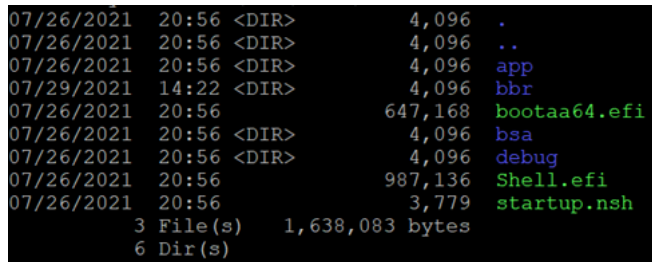
After reading this guide, you can go to the [Arm SystemReady Certification Program](#) site for more information about certification registration.

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.

Figure A-1 shows the folder contents:

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
```

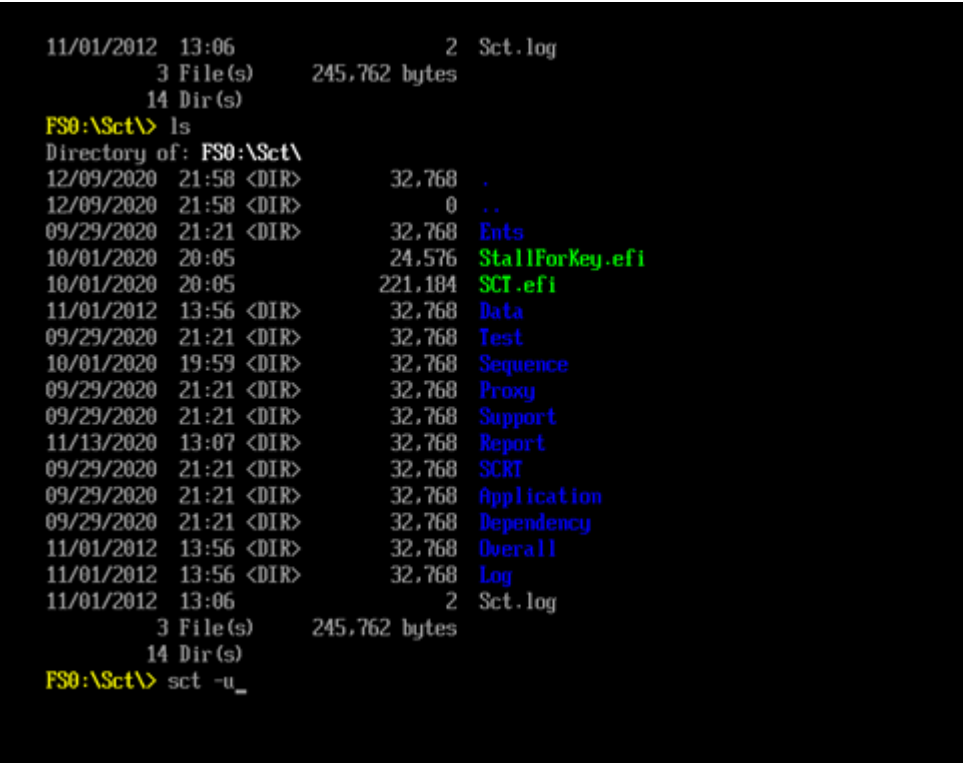
For a list of application parameters, see the [Arm BSA Compliance User Guide](#) and [Arm SBSA Compliance User Guide](#).

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
```

Figure A-2 shows how to run specific SCT test cases, you can start SCT with a GUI by passing `-u` as a parameter:

Figure A-2: Start SCT with a GUI



1. Press F5 to select tests manually. Press Enter.
2. Figure A-3 shows the view, add, or remove tests in the Test Case Management menu:

Figure A-3: Test Case Management menu



3. Press F9 to run SCT, as shown in Figure A-4:

Figure A-4: SCT screen

```
UEFI Compliant - Validating a boot image received through a network device must
be implemented -- FAILURE
98551AE7-5020-4DDD-861A-CFFF84D60382
/home/jefboo01/edk2-test-build/SctPkg/TestCase/UEFI/EFI/Generic/EfiCompliant/BlackBoxTest/EfiCompliantBBTestPlatform_uefi.c:1635:SetupMode equal zero - No

UEFI Compliant-UEFI General Network Application required -- PASS
76A6A1B0-8C53-407D-8486-9A6E6332D3CE
/home/jefboo01/edk2-test-build/SctPkg/TestCase/UEFI/EFI/Generic/EfiCompliant/BlackBoxTest/EfiCompliantBBTestPlatform_uefi.c:1848:MnpSB-Y,ArpSB-Y,Ip4SB-Y,Dhcp4SB-Y,Tcp4SB-Y,Udp4SB-Y,Ip4Config2-Y,Mnp-Y,Arp-Y,Ip4-Y,Dhcp4-Y,Tcp4-Y,Udp4-Y

UEFI Compliant-UEFI U6 General Network Application required -- PASS
4C82EB2D-C785-410C-95D1-AE27122144C8
/home/jefboo01/edk2-test-build/SctPkg/TestCase/UEFI/EFI/Generic/EfiCompliant/BlackBoxTest/EfiCompliantBBTestPlatform_uefi.c:2058:Dhcp6SB-Y,Tcp6SB-Y,Ip6SB-Y,Udp6SB-Y,Ip6Config-Y,Dhcp6-Y,Tcp6-Y,Ip6-Y,Udp6-Y

UEFI Compliant - VLAN protocols must be implemented -- PASS
329027CE-406E-48C8-8AC1-A02C1A6E3983
/home/jefboo01/edk2-test-build/SctPkg/TestCase/UEFI/EFI/Generic/EfiCompliant/BlackBoxTest/EfiCompliantBBTestPlatform_uefi.c:2119:VLAN - Yes
```



Note

Sometimes SCT can hang in the process of self-reset. In this case, power off the system then power it on. The tests are not reset. During the next boot if the same USB drive with SCT has been selected as the boot device, the test continues. 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.