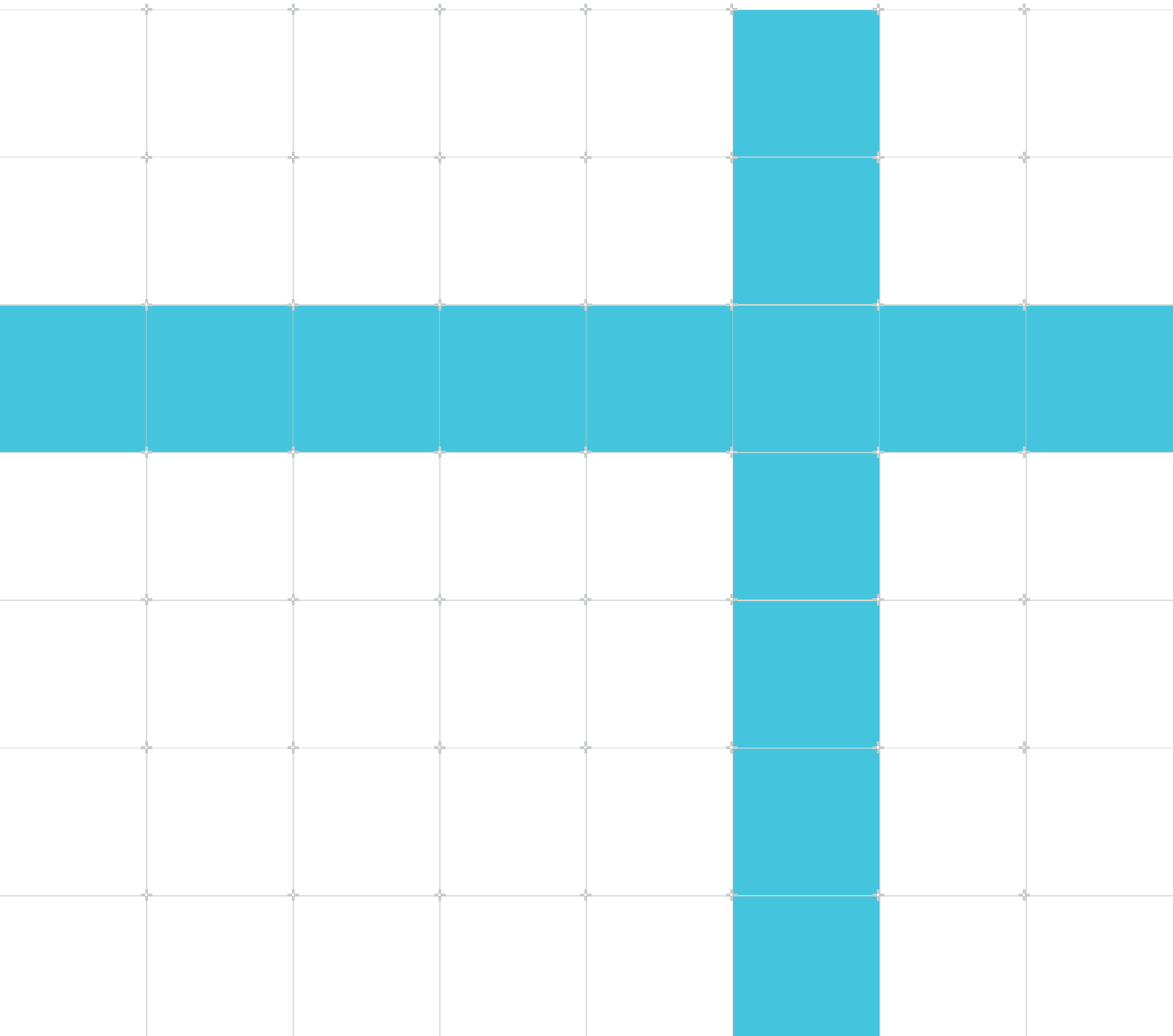
SystemReady Devicetree Band Integration and Testing Guide

Version 4.4

Issue 04
DUI1101_4.4_04_en



SystemReady Devicetree Band Integration and Testing Guide

Release information

Document history

Issue	Date	Confidentiality	Change
0100	17 August 2021	Non-Confidential	First release version 1.0
0101	7 April 2022	Non-Confidential	Second release version 1.1
0200	25 January 2023	Non-Confidential	Updates for SystemReady IR 2.0 - Beta Release
0200	15 May 2023	Non-Confidential	Updates for SystemReady IR 2.0 - EAC Release
0201	22 November 2023	Non-Confidential	Updates for SystemReady IR 2.1 to reflect SRS v2.2 new features and some automated tests in ACS image - EAC Release
0201-01	24 April 2024	Non-Confidential	Minor updates
0202	23 September 2024	Non-Confidential	Add “Deploying Yocto on SystemReady-compliant hardware” appendix
0300-01	24 December 2024	Non-Confidential	Updates of SystemReady Devicetree Band Integration and Testing Guide
0400-01	16 June 2025	Non-Confidential	Major update
0400-02	12 August 2025	Non-Confidential	Minor update
0400-03	23 October 2025	Non-Confidential	Moved the SystemReady Devicetree FAQs to the main SystemReady FAQs document

Issue	Date	Confidentiality	Change
0400-04	5 January 2026	Non-Confidential	Minor update

Proprietary Notice

PRE-1121-V1.0

Confidentiality Status

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. SystemReady Devicetree band overview

SystemReady is a compliance program. It is based on a set of hardware and firmware standards that enable interoperability with generic off-the-shelf operating systems and hypervisors. These standards include the following:

- The Base System Architecture (BSA)
- Base Boot Requirements (BBR)
- Base Boot Security Requirements (BBSR)
- Market-specific supplements

SystemReady replaces the successful ServerReady compliance program and extends it to a broader set of devices.

SystemReady compliance ensures that Arm-based servers, infrastructure edge devices, and embedded IoT systems are designed to specific requirements. This enables generic, off-the-shelf operating systems to work out of the box on Arm-based devices. The compliance program enables systems to meet the SystemReady standards.

1.1 Before you begin

Before you test for SystemReady devicetree compliance, review this guide objective and the SystemReady testing requirements in this section. This guide is specific for SystemReady Devicetree band.

This guide describes how to configure a U-Boot-based platform for SystemReady Devicetree band compliance, and how to run all the SystemReady Devicetree tests.

This guide assumes that your system has U-Boot firmware and the examples shown are captured on a U-Boot platform. However, you can achieve SystemReady Devicetree compliance with any UEFI-compliant firmware. Using U-Boot is not mandatory. You can also use EDK2 or another firmware implementation for compliance. If you are not using U-Boot, you can ignore the Configure U-Boot for SystemReady section.

Before running the given tests, the following tasks which are required for compliance should be performed:

- Enable Unified Extensible Firmware Interface (UEFI) features in U-Boot
- Run the SystemReady Devicetree live image of the Arm Architecture Compliance Suite (ACS) and analyze test results
- Enable the EFI System Resource Table (ESRT) feature in U-Boot and test it in ACS
- Run the ACS Devicetree validation test in ACS

- Test for availability and accessibility of the Ethernet ports from Linux
- Sign firmware images, and test the `UpdateCapsule()` interface to authenticate signatures and update the firmware (recommended)
- Enable secure boot in U-Boot and test it in ACS (recommended)
- Boot and install generic Linux distribution images

For more information about SystemReady compliance and testing requirements, see the Arm SystemReady Requirements Specification.

SystemReady Devicetree compliant platforms must provide a specific minimum set of hardware and firmware features to enable an operating system to be deployed. Compliant systems must conform to the following requirements:

- The Embedded Base Boot (EBBR) Requirements. The EBBR specification is aimed at Arm embedded device developers who want to use UEFI technology to separate firmware and OS development. For example, class-A embedded devices such as networking platforms can benefit from a standard interface that supports features such as secure boot and firmware updates. For more information, download the EBBR specification and reference source code from the EBBR GitHub repository.
- The EBBR recipe requirements described in the Arm Base Boot Requirements.
- Arm recommends that SystemReady Devicetree platforms comply with the Arm Base System Architecture (BSA) specification.
- Arm recommends that SystemReady Devicetree platforms comply with the Base Boot Security Requirements (BBSR). This compliance is currently recommended, not mandatory.

2. Configure U-Boot for SystemReady

This section explains how to enable the U-Boot configuration options required for SystemReady Devicetree compliance.

These options enable the following features:

- UEFI
- Device Firmware Upgrade, to enable `UpdateCapsule()` support
- Encryption, to enable verification of signed capsules with FMP format with `UpdateCapsule()`
- ESRT
- Secure boot (when certifying with the recommended BBSR option)

This section is only relevant if you are using U-Boot firmware. You can ignore this section if you are using EDK2 or other firmware.

2.1 Prerequisites

Build U-Boot and install it on your platform. U-Boot 2022.04 or later is required for SystemReady Devicetree v2.1 onwards. U-Boot releases and patches are in the U-Boot git repository. Instructions for porting and building U-Boot is beyond the scope of this document. See the U-Boot documentation for details on how to enable a new platform.

2.2 UEFI

The UEFI Application Binary Interface must be enabled and supported in U-Boot for SystemReady Devicetree.

To configure UEFI support in U-Boot:

1. In `<root_workspace>/u-boot/configs/<platform_name>_defconfig`, enable the configuration options as the following code shows:

```
// Core UEFI features
CONFIG_BOOTM_EFI=y
CONFIG_CMD_BOOTEFI=y
CONFIG_CMD_NVEDIT_EFI=y
CONFIG_HEXDUMP=y
CONFIG_CMD_GPT=y
CONFIG_EFI_PARTITION=y
CONFIG_EFI_LOADER=y
CONFIG_EFI_DEVICE_PATH_TO_TEXT=y
CONFIG_EFI_UNICODE_COLLATION_PROTOCOL2=y
CONFIG_EFI_UNICODE_CAPITALIZATION=y
CONFIG_EFI_HAVE_RUNTIME_RESET=y
CONFIG_EFI_VARIABLE_FILE_STORE=y
```

2. In `<root_workspace>/u-boot/configs/<platform_name>_defconfig`, add the following configuration options to enable Real Time Clock (RTC) support:

```
CONFIG_DM_RTC=y
CONFIG_EFI_GET_TIME=y
CONFIG_EFI_SET_TIME=y
CONFIG_RTC_EMULATION=y
```

   This configuration uses the RTC emulation feature that works on all platforms. If your platform has a real RTC, enable the `CONFIG_RTC_*` option for that device instead of `CONFIG_RTC_EMULATION`.

3. In `<root_workspace>/u-boot/configs/<platform_name>_defconfig`, add the following configuration options to enable the UEFI `UpdateCapsule()` interface to update firmware:

```
CONFIG_CMD_DFU=y
CONFIG_FLASH_CFI_MTD=y
CONFIG_EFI_CAPSULE_FIRMWARE_FIT=y
CONFIG_EFI_CAPSULE_FIRMWARE_MANAGEMENT=y
CONFIG_EFI_CAPSULE_FIRMWARE=y
CONFIG_EFI_CAPSULE_FIRMWARE_RAW=y
```

4. In `<root_workspace>/u-boot/configs/<platform_name>_defconfig`, add the following configuration options to enable partitions and filesystems support:

```
CONFIG_CMD_GPT=y
CONFIG_FAT_WRITE=y
CONFIG_FS_FAT=y
CONFIG_CMD_PART=y
CONFIG_PARTITIONS=y
CONFIG_DOS_PARTITION=y
CONFIG_ISO_PARTITION=y
CONFIG_EFI_PARTITION=y
CONFIG_PARTITION_UUIDS=y
```

With the UEFI ABI, U-Boot can find and execute UEFI binaries from a system partition on an eMMC, SD card, USB flash drive, or other device. You can test UEFI boot using either a Linux distribution ISO image or the ACS. Boot the platform with the image on a USB flash drive to boot either the GRUB Linux distribution or the EFI Shell.

While this is not strictly required for compliance, it can be useful to also add the following configuration options in `<root_workspace>/u-boot/configs/<platform_name>_defconfig` during firmware development:

```
CONFIG_CMD_EFIDEBUG=y
CONFIG_CMD_BOOTEFI_HELLO=y
CONFIG_CMD_BOOTEFI_HELLO_COMPILE=y
CONFIG_CMD_BOOTEFI_SELFTEST=y
```

2.3 Device Firmware Upgrade

In U-Boot, configure Device Firmware Upgrade (DFU) to enable `UpdateCapsule` support, if your system supports it. `UpdateCapsule` supports both signed capsule update and unsigned capsule update schemes, set by different U-Boot configuration options. Arm recommends using the signed capsule update scheme.

This section splits the configuration process into two parts:

- Common configuration describes steps that are common to both the signed and unsigned capsule update schemes.
- Generate capsule files describes steps that are specific to the signed capsule update scheme.

This guide does not describe the unsigned capsule update scheme, because it is not compliant.

2.3.1 Common configuration

To enable DFU mode:

1. In `<root_workspace>/u-boot/configs/<platform_name>_defconfig`, add the following configuration options:

```
CONFIG_FIT=y
CONFIG_OF_LIBFDT=y
CONFIG_DFU=y
CONFIG_CMD_DFU=y
```

2. In `<root_workspace>/u-boot/configs/<platform_name>_defconfig`, add one or more of the DFU backend configuration options for the storage device containing the firmware:

```
CONFIG_DFU_MMC=y
CONFIG_DFU_MTD=y
CONFIG_DFU_NAND=y
CONFIG_DFU_SF=y
```

3. In `<root_workspace>/u-boot/configs/<platform_name>_defconfig`, enable the following configuration options to ensure that one or more of the DFU transport options are enabled for testing:

```
CONFIG_DFU_OVER_TFTP=y
CONFIG_DFU_OVER_USB=y
```

4. Edit the `test.its` file to create a Flattened Image Tree (FIT) image used for testing:

```
/dts-v1/;

/ {
     description = "Automatic U-Boot update";
     \#address-cells = <1>;

     images {
             u-boot.bin {
                     description = "U-Boot binary";
```

```
                        data = /incbin/("u-boot.bin");
                        compression = "none";
                        type = "firmware";
                        arch = "arm64";
                        load = <0>;

                        hash-1 {
                                algo = "sha1";
                        };
                };
        };
};
```

5. Generate the binary `tests.itb` test image using the `mkimage` command:

```
$ mkimage -f test.its tests.itb
```

6. Use the `dfu` command to test that DFU is functioning correctly and reflash the device firmware. The following example code shows DFU over TFTP:

```
u-boot=> setenv updatefile tests.itb
u-boot=> dhcp
u-boot=> dfu tftp ${kernel_addr_r}
```

2.3.2 Generate capsule files

This section describes how to generate three different capsule files:

- A signed capsule supporting authentication.
- An unsigned capsule, which should fail authentication.
- A tampered capsule, which should also fail authentication.

To generate a signed capsule file, follow these steps:

1. To support signed capsule file authentication, you need to enable the asymmetric algorithm, HASH algorithm, secure boot, and X509 format certificate parser functions. These features correspond to the following configuration options in `<root_workspace>/u-boot/configs/<platform_name>_defconfig` for U-Boot:

```
CONFIG_EFI_CAPSULE_AUTHENTICATE=y
CONFIG_EFI_HAVE_CAPSULE_SUPPORT=y
CONFIG_EFI_RUNTIME_UPDATE_CAPSULE=y
CONFIG_EFI_CAPSULE_ON_DISK=y
CONFIG_EFI_SECURE_BOOT=y
CONFIG_EFI_SIGNATURE_SUPPORT=y
CONFIG_RSA=y
CONFIG_RSA_VERIFY=y
CONFIG_RSA_VERIFY_WITH_PKEY=y
CONFIG_IMAGE_SIGN_INFO=y
CONFIG_RSA_SOFTWARE_EXP=y
CONFIG_ASYMMETRIC_KEY_TYPE=y
CONFIG_ASYMMETRIC_PUBLIC_KEY_SUBTYPE=y
CONFIG_RSA_PUBLIC_KEY_PARSER=y
CONFIG_X509_CERTIFICATE_PARSER=y
CONFIG_PKCS7_MESSAGE_PARSER=y
CONFIG_PKCS7_VERIFY=y
CONFIG_HASH=y
```

```
CONFIG_SHA1=y
CONFIG_SHA256=y
CONFIG_SHA512=y
CONFIG_SHA384=y
CONFIG_MD5=y
CONFIG_CRC32=y
```

2. To authenticate the signed firmware, generate a private key-pair and use the private key to sign the firmware. This requires installing the following tools on your host:

   - `openssl`
   - `efitools`
   - `dtc version >=1.6`
   - `mkeficapsule`

   Create the keys and certificate files on your host:

```
$ openssl req -x509 -sha256 -newkey rsa:2048 -subj /CN=CRT/ \
        -keyout CRT.key -out CRT.crt -nodes -days 365
$ cert-to-efi-sig-list CRT.crt CRT.esl
```

3. Use the `mkeficapsule` command to package the U-Boot binary in the capsule format:

```
$ mkeficapsule --monotonic-count 1 \
 --private-key "CRT.key" \
 --certificate "CRT.crt" \
 --index 1 \
 --guid 058B7D83-50D5-4C47-A195-60D86AD341C4 \
 "tests.itb" \
 "signed_capsule.bin"
```

You can use the resulting `signed_capsule.bin` binary to update the firmware with UEFI capsule update, as described in Test SystemReady Devicetree band.

To generate an unsigned capsule file and a tampered capsule file from a signed capsule file, use `capsule-tool.py` from the SystemReady scripts as follows:

```
$ capsule-tool.py --de-authenticate --output unauth.bin signed_capsule.bin
$ capsule-tool.py --tamper --output tampered.bin signed_capsule.bin
```

The `mkimage` and `mkeficapsule` tools exist in the U-Boot repository `tools` directory. For information about how to build `mkimage` and `mkeficapsule`, see Building tools for Linux, in the U-Boot documentation. Alternatively, you can use the `GenerateCapsule` tool from EDK2 to create a UEFI Capsule binary.

GUID values are bound to particular systems. The GUID value `058B7D83-50D5-4C47-A195-60D86AD341C4` in the example above is for U-Boot using FIT format on the QEMU platform. Replace it with your system-specific GUID.

To authenticate the signed capsule firmware, insert the signing public key into a `dtb` file.

1. Create a `signature.dts` file:

```
/dts-v1/;
/plugin/;

&{/} {
     signature {
                capsule-key = /incbin/("CRT.esl");
     };
};
```

1. Compile the `signature.dts` file and overlay it on the original system dtb file:

```
$ dtc -@ -I dts -O dtb -o signature.dtbo signature.dts
$ fdtoverlay -i orig.dtb -o new.dtb -v signature.dtbo
```

The `orig.dtb` file is the original system dtb file. The `new.dtb` file is a new dtb file which includes the signing public key certificate. Test UpdateCapsule uses this `new.dtb` file.

2.4 EFI System Resource Table (ESRT)

The EFI System Resource Table (ESRT) is a standard table for providing firmware version and upgrade information to UEFI applications and the OS. Platforms with SystemReady Devicetree compliance can benefit from integrating with common firmware update infrastructure.

To support ESRT, the U-Boot configuration must enable the following option:

```
CONFIG_EFI_ESRT=y
```

2.5 Secure boot

Secure boot enables firmware authentication in the boot stage. This is required for BBSR compliance. To support this feature, enable the following configuration options in `<root_workspace>/u-boot/configs/<platform_name>_defconfig`, :

```
CONFIG_EFI_SECURE_BOOT=y
CONFIG_EFI_LOADER=y
CONFIG_FIT_SIGNATURE=y
CONFIG_EFI_SIGNATURE_SUPPORT=y
CONFIG_HASH=y
CONFIG_RSA=y
CONFIG_RSA_VERIFY=y
CONFIG_RSA_VERIFY_WITH_PKEY=y
CONFIG_IMAGE_SIGN_INFO=y
CONFIG_ASYMMETRIC_KEY_TYPE=y
CONFIG_ASYMMETRIC_PUBLIC_KEY_SUBTYPE=y
CONFIG_X509_CERTIFICATE_PARSER=y
CONFIG_PKCS7_MESSAGE_PARSER=y
CONFIG_PKCS7_VERIFY=y
```

These configuration options might already have been enabled if you configured them as part of Device Firmware Upgrade to support signed capsule file authentication.

2.6 Adapt the automated boot flow

Make sure that the automated boot sequence attempts the UEFI boot methods. In U-Boot, the `bootcmd` environment variable holds the default boot command. This is usually a script that attempts one or more boot methods in turn. This script tries to boot using the `bootefi bootmgr` and `bootefi` commands. If your system is using the generic distro configuration, the generated `scan_dev_for_efi` boot script automatically tries the UEFI boot methods.

Next, make sure the `bootargs` environment variable is empty when booting with UEFI. The `bootargs` U-Boot environment variable holds the arguments passed to the image being booted, which is traditionally the Linux kernel. When booting with the UEFI boot methods, the UEFI application binary receives the `bootargs` arguments. Commonly, operating systems boot with UEFI to run intermediate UEFI applications like GNU GRUB before booting the Linux kernel. To avoid interfering with UEFI applications, the `bootargs` environment variable must be empty when booting with UEFI. If your system uses the generic distro configuration, the `bootargs` are handled appropriately.

2.7 Adapt the Devicetree

Adapt the U-Boot built-in Devicetree to support OS boot. When booting with UEFI, the Devicetree is passed to UEFI applications, including the Linux kernel, as an EFI configuration table. With U-Boot, specify the Devicetree using an argument to the `bootefi` command. You can load this Devicetree by the boot scripts from the storage medium. However, if U-Boot is already using a built-in Devicetree in `$fdtcontroladdr`, the simplest option is to use this Devicetree. If necessary, you can adapt the U-Boot built-in Devicetree sources to support both U-Boot and Linux OS boot.

Also, ensure that the UEFI Devicetree displays the console UART. It is common with U-Boot to pass the console UART information to the Linux kernel as arguments using the `bootargs` variable. When booting with UEFI, the console UART must be specified as `stdout-path` in the `chosen` node of the Devicetree.

The following code shows a simplified Devicetree example:

```
/ {
        chosen {
                stdout-path = "/serial@f00:115200";
        };

        serial@f00 {
                compatible = "vendor,some-uart";
                reg = <0xf00 0x10>;
        };
};
```

3. Test SystemReady Devicetree band

This section explains how to run the U-Boot and UEFI tests, and how to test the Linux installation for SystemReady Devicetree.


Note

The Test checklist appendix includes the steps in this section to help during testing.

Before you start the SystemReady Devicetree testing, you need the following tools and images:

- Provided by your platform vendor:
  - The platform under test with firmware already installed.
  - Three capsule files in FMP format: signed, unsigned, and tampered. These files were generated in Generate capsule files.
- Provided by Arm:
  - This guide
  - Arm SystemReady Devicetree band ACS live image installed on a storage medium. For details about which version of the ACS image should be used, see the Arm SystemReady Requirements Specification.
- Provided by a third party:
  - Three actively supported versions of Linux distributions or BSD images on storage media.

The SystemReady Devicetree tests include the following:

- EFI System Resource Table (ESRT) dump and sanity check
- Devicetree validation
- Ethernet port availability and accessibility test
- UEFI Capsule Update tests
- EBBR tests
- UEFI BSA and Linux BSA tests
- Secure boot test, recommended
- Installation and boot of three different heritage of Linux/BSD distributions
- Boot sources stated by the vendor
- PSCI version check

3.1 Test ESRT

This test is run automatically as part of the ACS. To perform the test manually, run the `CapsuleApp.efi` application on UEFI Shell to dump the EFI System Resource Table (ESRT), as follows:

```
UEFI Interactive Shell v2.2
EDK II
UEFI v2.100 (Das U-Boot, 0x20230700)
Mapping table
      FS0: Alias(s):HD0b:;BLK1:
          /VenHw(e61d73b9-a384-4acc-aeab-82e828f3628b)/VenHw(63293792-adf5-9325-
b99f-4e0e455c1b1e,00)/HD(1,GPT,f5cc8412-cd9f-4c9e-a782-0e945461e89e,0x800,0x32000)
      FS1: Alias(s):HD0c:;BLK2:
          /VenHw(e61d73b9-a384-4acc-aeab-82e828f3628b)/VenHw(63293792-adf5-9325-
b99f-4e0e455c1b1e,00)/HD(2,GPT,ed59c37b-2a8d-4d58-a7ec-a2d7e42ab4a1,0x32800,0xba40e)
      FS2: Alias(s):HD0d:;BLK3:
          /VenHw(e61d73b9-a384-4acc-aeab-82e828f3628b)/VenHw(63293792-adf5-9325-
b99f-4e0e455c1b1e,00)/HD(3,GPT,3000afbb-d111-4bb9-ae70-5f2242f9c85f,0xed000,0x19000)
     BLK0: Alias(s):
          /VenHw(e61d73b9-a384-4acc-aeab-82e828f3628b)/VenHw(63293792-adf5-9325-
b99f-4e0e455c1b1e,00)
     BLK3: Alias(s):
          /VenHw(e61d73b9-a384-4acc-aeab-82e828f3628b)/VenHw(63293792-adf5-9325-
b99f-4e0e455c1b1e,01)
Press ESC in 5 seconds to skip startup.nsh or any other key to continue.
Shell> FS0:
FS0:\> ls
Directory of: FS0:\
08/05/2011  23:00 <DIR>             0  acs_results
08/05/2011  23:00 <DIR>             0  EFI
08/05/2011  23:00          33,683,456  Image
08/05/2011  23:00 <DIR>             0  security-interface-extension-keys
08/05/2011  23:00               3,288  startup.nsh
08/05/2011  23:00                   0  yocto_image.flag
          3 File(s)  33,686,744 bytes
          2 Dir(s)
FS0:\> \EFI\BOOT\app\CapsuleApp.efi -E

ASSERT_EFI_ERROR (Status = Not Found)
ASSERT [CapsuleApp] /home/edhcha01/RELEASE/arm-systemready/IR/Yocto/meta-woden/
build/tmp/work/generic_arm64-oe-linux/uefi-apps/1.0-r0/edk2/MdeModulePkg/Library/
UefiHiiServicesLib/UefiHiiServicesLib.c(94): !(((INTN)(RETURN_STATUS)(Status)) < 0)

ASSERT_EFI_ERROR (Status = Not Found)
ASSERT [CapsuleApp] /home/edhcha01/RELEASE/arm-systemready/IR/Yocto/meta-woden/
build/tmp/work/generic_arm64-oe-linux/uefi-apps/1.0-r0/edk2/MdePkg/Library/
DxeServicesTableLib/DxeServicesTableLib.c(58): !(((INTN)(RETURN_STATUS)(Status)) <
 0)
ASSERT [CapsuleApp] /home/edhcha01/RELEASE/arm-systemready/IR/Yocto/meta-woden/
build/tmp/work/generic_arm64-oe-linux/uefi-apps/1.0-r0/edk2/MdePkg/Library/
DxeServicesTableLib/DxeServicesTableLib.c(59): gDS != ((void *) 0)

ASSERT_EFI_ERROR (Status = Not Found)
ASSERT [CapsuleApp] /home/edhcha01/RELEASE/arm-systemready/IR/Yocto/meta-woden/
build/tmp/work/generic_arm64-oe-linux/uefi-apps/1.0-r0/build/Build/MdeModule/
RELEASE_GCC5/AARCH64/MdeModulePkg/Application/CapsuleApp/CapsuleApp/DEBUG/
AutoGen.c(415): !(((INTN)
ASSERT_EFI_ERROR (Status = Not Found)
ASSERT [CapsuleApp] /home/edhcha01/RELEASE/arm-systemready/IR/Yocto/meta-woden/
build/tmp/work/generic_arm64-oe-linux/uefi-apps/1.0-r0/edk2/MdePkg/Library/
DxeHobLib/HobLib.c(48): !(((INTN)(RETURN_STATUS)(Status)) < 0)
ASSERT [CapsuleApp] /home/edhcha01/RELEASE/arm-systemready/IR/Yocto/meta-woden/
build/tmp/work/generic_arm64-oe-linux/uefi-apps/1.0-r0/edk2/MdePkg/Library/
DxeHobLib/HobLib.c(49): mHobList != ((void *) 0)
##############
# ESRT TABLE #
```

```
##############
EFI_SYSTEM_RESOURCE_TABLE:
FwResourceCount    - 0x1
FwResourceCountMax - 0x1
FwResourceVersion  - 0x1
EFI_SYSTEM_RESOURCE_ENTRY (0):
  FwClass                 - 058B7D83-50D5-4C47-A195-60D86AD341C4
  FwType                  - 0x0 (Unknown)
  FwVersion               - 0x0
  LowestSupportedFwVersion - 0x0
  CapsuleFlags            - 0x0
  LastAttemptVersion      - 0x0
  LastAttemptStatus       - 0x0 (Success)
```

Perform the ESRT sanity check on SystemReady-devicetree band ACS Linux Shell as follows:

1. `exit` UEFI Shell, and enter into `Linux Boot`
2. Run the `fwts` command on Linux Shell, as follows:

```
root@generic-arm64:~# fwts --ebbr esrt
Test: Sanity check UEFI ESRT Table.
  Sanity check UEFI ESRT Table.                          1 passed
  Validity of fw_class in UEFI ESRT Table for EBBR.      1 passed
```

3.2 Test Devicetree

This section describes how to manually perform Devicetree validation and a Kernel self-test

3.2.1 Devicetree validation

Devicetree validation ensures that the Devicetree node data matches the schema constraints. The ACS performs this test automatically. This section describes how to perform the test manually.

1. Install Devicetree Schema Tools using the following command:

```
$ pip3 install -U dtschema
```

   See dt-schema for more information.

2. Install device specific bindings

   Download or clone the latest stable kernel. The Devicetree schemas are in the `Documentation/devicetree/bindings` directory.

3. Run the `bsa.efi` application on UEFI Shell to dump the Devicetree:

```
FS0:\EFI\BOOT\bsa\> Bsa.efi -dtb BsaDevTree.dtb


 BSA Architecture Compliance Suite
         Version 1.0.1

 Starting tests with print level :  3
```

```
Creating Platform Information Tables
PE_INFO: Number of PE detected       :    2
GIC_INFO: Number of GICD             :    1
GIC_INFO: Number of ITS              :    0
 MEM timer node offset not found
TIMER_INFO: Number of system timers  :    0
WATCHDOG_INFO: Number of Watchdogs   :    0
PCIE_INFO: Number of ECAM regions    :    1
PCIE_INFO: No entries in BDF Table
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
```

This command dumps the dtb content and stores it in the `BsaDevTree.dtb` file. The ACS test does this step automatically which is introduced in Run ACS in automated mode.

4. Mount the SystemReady-devicetree band ACS image on a Linux host machine and copy the `BsaDevTree.dtb` file to the host machine.
5. Use the `dtc` tool to de-compile the `BsaDevTree.dtb` file, as follows:

```
$ dtc -o /dev/null -O dts -I dtb -s acs_results/uefi/BsaDevTree.dtb &>log
```

6. Validate the Devicetree file as follows:

```
$ dt-validate -s <kernel path>/Documentation/devicetree/bindings -m acs_results/
uefi/BsaDevTree.dtb &>>log
```

7. Analyze the logs with the `dt-parser` script:

```
$ dt-parser.py log
```

There should be no errors reported. Read the warnings and act on any relevant information. For more information about how to download this script, see Verify the test results.

3.2.2 DT Kernel self-test

DT kernel self-test is integrated with ACS live image and is used for validating the devices described in the device tree have an associated drivers in OS.

The ACS performs this test automatically. This section describes how to perform the test manually.

1. Move to /usr/kernel-selftest directory `cd /usr/kernel-selftest`
2. Run the kselftest script `./run_kselftest.sh -t dt:test_unprobed_devices.sh &> log`

3.3 Ethernet port Test

SystemReady Devicetree compliance requires the ethernet port to be available and accessible in the OS. After the ACS Linux system boots up, the `init.sh` script automatically executes. It

executes `ethtool-test.py` script to detect and test the ethernet interfaces. The tests include bringing up the ethernet interface and pinging the router/gateway.

The log is automatically stored in `acs_results_template/acs_results/linux-tools/ethtool-test.log`. The expected logs look like:

```
*****************************************************************

                          Running ethtool

*****************************************************************
INFO: Detected following ethernet interfaces via ip command :
0: eth0

INFO: Bringing down all ethernet interfaces using ifconfig
ifconfig eth0 down

*****************************************************************

INFO: Bringing up ethernet interface: eth0
INFO: Running "ethtool eth0 " :
Settings for eth0:
.Supported ports: [  ]
.Supported link modes:   Not reported
.Supported pause frame use: No
.Supports auto-negotiation: No
.Supported FEC modes: Not reported
.Advertised link modes:  Not reported
.Advertised pause frame use: No
.Advertised auto-negotiation: No
.Advertised FEC modes: Not reported
.Speed: Unknown!
.Duplex: Unknown! (255)
.Auto-negotiation: off
.Port: Other
.PHYAD: 0
.Transceiver: internal
.Link detected: yes

INFO: Ethernet interface eth0 doesn't supports ethtool self test
INFO: Link detected on eth0
INFO: Running ip address show dev eth0 :
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel qlen 1000
    link/ether 52:54:00:12:34:56 brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.15/24 brd 10.0.2.255 scope global dynamic eth0
       valid_lft 86380sec preferred_lft 86380sec
    inet6 fec0::5054:ff:fe12:3456/64 scope site dynamic noprefixroute flags 100
       valid_lft 86383sec preferred_lft 14383sec
    inet6 fe80::5054:ff:fe12:3456/64 scope link
       valid_lft forever preferred_lft forever

INFO: eth0 support DHCP
INFO: Running ip route show dev eth0 :
default via 10.0.2.2 dev eth0  src 10.0.2.15  metric 10
10.0.2.0/24 dev eth0 scope link  src 10.0.2.15  metric 10
10.0.2.2 dev eth0 scope link  src 10.0.2.15  metric 10
10.0.2.3 dev eth0 scope link  src 10.0.2.15  metric 10

INFO: Router/Gateway IP for eth0 : 10.0.2.2
INFO: Running ifconfig eth0 up :
INFO: Running ping -w 10000 -c 3 -I eth0 10.0.2.2 :
PING 10.0.2.2 (10.0.2.2): 56 data bytes
64 bytes from 10.0.2.2: seq=0 ttl=255 time=3.320 ms

--- 10.0.2.2 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 1.774/2.562/3.320 ms
```

```
INFO: Ping to router/gateway[10.0.2.2] for eth0 is successful
INFO: Running ping -w 10000 -c 3 -I eth0 www.arm.com :
PING www.arm.com (96.17.150.99): 56 data bytes
64 bytes from 96.17.150.99: seq=0 ttl=255 time=121.453 ms

--- www.arm.com ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 121.453/121.755/122.213 ms

INFO: Ping to www.arm.com is successful

************************************************************
```

3.4 Test UpdateCapsule

UpdateCapsule is the standard interface to update firmware. The SystemReady devicetree ACS image provides automatic capsule update testing as part of ACS automation run.

Prerequisites:

1. Copy the platform's three capsule files:
   - `unauth.bin`
   - `tampered.bin`
   - `signed_capsule.bin`

These files were generated using Generate capsule files mentioned in the IR guide. They are in the `BOOT` partition `/acs_tests/app/` path of the ACS image on a storage drive.

1. Boot the ACS image on the platform with the `new.dtb` file which was generated in Generate capsule files.

The automatic capsule update flow is as follows.

1. Select `bbr/bsa` from the GRUB boot menu and press Enter. This is the default selection and runs automatically if you do not choose another option.
2. First, the UEFI-based tests, including SCT, BSA, and the Ethernet port test, runs. After these tests complete, ACS automatically boots to Linux.
3. In Linux, the FWTS, BSA, and MVP tests are executed. The image is automatically detected if you need a capsule update test and will reboot to perform the testing.
4. After rebooting to the UEFI shell, SCT and BSA tests are skipped, because they were already completed in the first run.
5. User input is requested to proceed with the capsule update, with a 10-second timeout.
6. If you press any key within 10 seconds, `capsule.efi` performs the on-disk capsule update test. If you do not press any key and ACS determines that capsule testing is needed, it proceeds with the test. However, a prompt is provided, allowing you to skip the capsule update test if wanted.
7. Before firmware update, script captures ESRT and SMBIOS tables for firmware update versions into `acs_results_template\acs_results\fw` path.

8. As part of firmware update, first the update with `unauth.bin` and `tampered.bin` is done, then update with `signed_capsule.bin` is performed.

```
FS0:\acs_tests\app\CapsuleApp.efi   FS0:\acs_tests\app\unauth.bin
FS0:\acs_tests\app\CapsuleApp.efi   FS0:\acs_tests\app\tampered.bin
FS0:\acs_tests\app\CapsuleApp.efi   FS0:\acs_tests\app\signed_capsule.bin -OD
```

9. If the update with `signed_capsule.bin` is successful, the system automatically resets for the firmware update. In case of failure, ACS boots to Linux with a failure acknowledgment.
10. On reboot after firmware update, the script captures ESRT and SMBIOS tables for firmware update versions into `acs_results_template\acs_results\fw` path and then boots to Linux with a success acknowledgement.
11. In Linux, a message is logged indicating whether the capsule update was successful or not, based on the uefi capsule update status.

3.5 Run the BBR tests

For SystemReady Devicetree compliance, the BBR test suites only test a reduced subset of the UEFI and EBBR specifications. Some of the EBBR tests are contained in the UEFI Self-Certification Tests (SCT), which are automatically executed when you choose `bbr/bsa` in the ACS GRUB menu. Other EBBR tests based on FWTS are integrated into the `init.sh` script which is automatically executed when you choose `Linux Boot` in the ACS GRUB menu.

```
root@generic-arm64:/usr/bin# fwts --ebbr
Running 3 tests, results appended to results.log
Test: UEFI miscellaneous runtime service interface tests.
  Test for UEFI miscellaneous runtime service interfaces  6 skipped
  Stress test for UEFI miscellaneous runtime service i..  1 skipped
  Test GetNextHighMonotonicCount with invalid NULL par..  1 skipped
  Test UEFI miscellaneous runtime services unsupported..  1 passed
Test: UEFI Runtime service variable interface tests.
  Test UEFI RT service get variable interface.            1 skipped
  Test UEFI RT service get next variable name interface.  1 skipped
  Test UEFI RT service set variable interface.            1 skipped
  Test UEFI RT service query variable info interface.     1 skipped
  Test UEFI RT service variable interface stress test.    1 skipped
  Test UEFI RT service set variable interface stress t..  1 skipped
  Test UEFI RT service query variable info interface s..  1 skipped
  Test UEFI RT service get variable interface, invalid..  1 skipped
  Test UEFI RT variable services unsupported status.      2 passed, 2 skipped
Test: UEFI Runtime service time interface tests.
  Test UEFI RT service get time interface.                1 skipped
  Test UEFI RT service get time interface, NULL time p..  1 skipped
  Test UEFI RT service get time interface, NULL time a..  1 skipped
  Test UEFI RT service set time interface.                1 skipped
  Test UEFI RT service set time interface, invalid yea..  1 skipped
  Test UEFI RT service set time interface, invalid yea..  1 skipped
  Test UEFI RT service set time interface, invalid mon..  1 skipped
  Test UEFI RT service set time interface, invalid mon..  1 skipped
  Test UEFI RT service set time interface, invalid day 0  1 skipped
  Test UEFI RT service set time interface, invalid day..  1 skipped
  Test UEFI RT service set time interface, invalid hou..  1 skipped
  Test UEFI RT service set time interface, invalid min..  1 skipped
  Test UEFI RT service set time interface, invalid sec..  1 skipped
  Test UEFI RT service set time interface, invalid nan..  1 skipped
  Test UEFI RT service set time interface, invalid tim..  1 skipped
  Test UEFI RT service set time interface, invalid tim..  1 skipped
```

```
Test UEFI RT service get wakeup time interface.          1 skipped
Test UEFI RT service get wakeup time interface, NULL..   1 skipped
Test UEFI RT service get wakeup time interface, NULL..   1 skipped
Test UEFI RT service get wakeup time interface, NULL..   1 skipped
Test UEFI RT service get wakeup time interface, NULL..   1 skipped
Test UEFI RT service set wakeup time interface.          1 skipped
Test UEFI RT service set wakeup time interface, NULL..   1 skipped
Test UEFI RT service set wakeup time interface, inva..   1 skipped
Test UEFI RT service set wakeup time interface, inva..   1 skipped
Test UEFI RT service set wakeup time interface, inva..   1 skipped
Test UEFI RT service set wakeup time interface, inva..   1 skipped
Test UEFI RT service set wakeup time interface, inva..   1 skipped
Test UEFI RT service set wakeup time interface, inva..   1 skipped
Test UEFI RT service set wakeup time interface, inva..   1 skipped
Test UEFI RT service set wakeup time interface, inva..   1 skipped
Test UEFI RT service set wakeup time interface, inva..   1 skipped
Test UEFI RT service set wakeup time interface, inva..   1 skipped
Test UEFI RT service set wakeup time interface, inva..   1 skipped
Test UEFI RT time services unsupported status.           4 passed
```

3.6 Run Linux BSA

The SystemReady-devicetree band ACS automatically runs the Linux BSA test. The BSA test is integrated into the `init.sh` script which is executed automatically when SystemReady-devicetree band ACS Linux boots up.

To run the Linux BSA test, do the following:

1. In the SystemReady-devicetree band ACS Linux Shell, load the kernel module as follows:

```
root@generic-arm64:~# insmod /lib/modules/*/kernel/bsa_acs/bsa_acs.ko
[   78.227399] init BSA Driver
```

2. Run the BSA test under Linux, as shown in the following example:

```
root@generic-arm64:~# bsa

 ************ BSA Architecture Compliance Suite *********
                      Version 1.0.6

 Starting tests (Print level is  3)

 Gathering system information....
[  194.112536]  PE_INFO: Number of PE detected       :    2
[  194.113856]  PCIE_INFO: Number of ECAM regions    :    1
[  194.496935]  PCIE_INFO: Number of BDFs found      :    0
[  194.497460]  PCIE_INFO: No entries in BDF Table
[  194.498691]  Peripheral: Num of USB controllers   :    0
[  194.499009]  Peripheral: Num of SATA controllers  :    0
[  194.499331]  Peripheral: Num of UART controllers  :    0
[  194.881670]  DMA_INFO: Number of DMA CTRL in PCIe :    0
[  194.882575]  SMMU_INFO: Number of SMMU CTRL       :    0

      *** Starting Memory Map tests ***
[  194.886495]
[  194.886495]       *** Starting Memory Map tests ***
[  194.887066]
[  194.887066] Operating System View:
[  194.887711]  104 : Addressability
```

```
[  194.887711]        Checkpoint --  2                          : Result:
 SKIPPED
[  194.889531]
[  194.889531]       One or more Memory tests failed or were skipped.

      *** Starting Peripherals tests ***
[  194.893543]
[  194.893543]       *** Starting Peripheral tests ***
[  194.894110]
[  194.894110] Operating System View:
[  194.894471]  605 : Memory Attribute of DMA
[  194.894471]        No DMA controllers detected...
[  194.894471]        Checkpoint --  3                          : Result:
 SKIPPED
[  194.895844]
[  194.895844]       One or more Peripheral tests failed or were skipped.

      *** Starting PCIe tests ***
[  194.899800]
[  194.899800]       *** Starting PCIe tests ***
[  194.900558]
[  194.900558] Operating System View:
[  194.900884]  801 : Check ECAM Presence                       : Result:  PASS
[  194.901618]
[  194.901618]        No PCIe Devices Found, Skipping PCIe tests...
[  194.902074]
[  194.902074]      ------------------------------------------------------------
[  194.902074]       Total Tests Run =  3, Tests Passed =  1, Tests Failed =  0
[  194.902074]      ------------------------------------------------------------

                    *** BSA tests complete ***
```

3. The BSA tests log is stored in `/mnt/acs_results_template/acs_results/linux_acs/bsa_acs_app`.

3.7 Secure boot test

Secure boot enables cryptographic authentication of the software in the boot stage. It detects whether the images loaded are corrupted or have been tampered with.

To enable the secure boot feature, all the firmware should be signed, and the boot process must be configured to use the RSA public key algorithm. The secure boot feature is an important requirement in the Base Boot Security Requirements which is recommended by the Arm SystemReady Compliance Program.

For more details about the secure boot test, see the BBSR and Security Interface Testing FAQ.

3.8 Test installation of Linux distributions

SystemReady Devicetree must boot at least three unmodified generic UEFI distributors images, each installed from a distinct storage medium written from an ISO image.

The following Linux distributions produce suitable ISO images:

- Fedora IoT

- Red Hat Enterprise Linux
- Rocky Linux
- Debian Stable
- Ubuntu Server
- OpenSUSE Leap
- SUSE Linux Enterprise Server
- OpenWRT
- The Yocto Project

In testing, use the supported versions of the pre-built distributions from your chosen vendor, above. We recommend using stable versions.

The selection of Linux distributions should cover the diversity of heritages. Fox example, RHEL/Fedora/Rocky Linux belong to the same heritage, SLES/openSUSE belong to the same heritage, and Ubuntu/Debian belong to the same heritage.

To test the Linux distribution installation, do the following:

1. Write the ISO image to a USB drive or other storage medium.
2. From a bash shell, run the following command to write the downloaded ISO to a storage medium. This replaces `<usb-block-device>` with the path to the drive's block device on your Linux workstation:

```
$ dd if=/path/to/distro-image.iso of=/dev/<usb-block-device> ; sync
```

3. When testing the distribution installation, capture the log of the installation, the serial console output, from the very first power-on of the board and subsequent installation, boot, and steps 5, 6 and 7 as console.log.
4. After the ISO is written to the USB drive or equivalent, connect the USB drive to your board and turn the board on. U-Boot finds the image and boots from the image by default. A compliant system boots from the distribution ISO into the installer tool. Use this installer tool to complete the installation of Linux and then reboot into a working Linux environment installed on the eMMC or other local storage.
5. After Linux is installed, run the following sequence of Linux sniff tests as root using the serial console:

```
# dmesg
# lspci -vvv
# lscpu
# lsblk
# dmidecode
# uname -a
# cat /etc/os-release
# efibootmgr
# cp -r /sys/firmware ~/
# tar czf ~/sys-firmware.tar.gz ~/firmware
```

6. Download the script ethtool-test.sh and run on the serial console as below:

```
$ sudo ./ethtool-test.sh | tee ethtool_test.log
```

7. Demonstrate the read/write functionality of the expected boot sources (USB/SD Card/eMMC etc.). This can be done by writing and reading a file to each block device like so:

```
#create test file
touch ref
shred -v -n 1 -s 99M ref
md5sum ref
#find first block device to test, say /dev/sda
lsblk
#copy test file to block device
dd if=ref of=/dev/sda bs=1M
#Read it back and verify hash
dd if=/dev/sda bs=1M count=99 | md5sum
#verify the md5 hash matches the original
#repeat with remaining boot sources
```

   Ensure you save the output of your boot sources tests as boot_sources.log

8. Copy the previously created console.log, sys-firmware.tar.gz, ethtool_test.log and boot_sources.log onto the flashed ACS boot device's results partition (first partition of the ACS image) to the path `/os-logs/linux-<distroname>-<distroversion>/`

For more details, see ethtool-test.

Always read the README.md from the SystemReady Devicetree template for the latest manual test instructions.

3.9 Boot sources tests

Boot sources tests are limited to those sources stated by the vendor as such and block devices. Other boot sources as network boot are not covered.

Boot sources tests detect all the block devices on the test platform, and perform the first block reading on each block device. In automated SystemReady-devicetree band ACS test running, the storage medium tests are integrated into the `init.sh` script which is automatically executed when you choose `Linux Boot` in the SystemReady-devicetree band ACS GRUB menu. The logs are stored in `acs_results_template/acs_results/linux_tools/read_write_check_blk_devices` automatically as part of the ACS test.

This is an example of the Boot sources tests on open-source platform.

```
*************************************************************************

                            Read block devices tool

*************************************************************************
INFO: Detected following block devices with lsblk command :
0: ram0
1: ram1
```

```
2: ram2
3: ram3
4: ram4
5: ram5
6: ram6
7: ram7
8: ram8
9: ram9
10: ram10
11: ram11
12: ram12
13: ram13
14: ram14
15: ram15
16: mtdblock0
17: vda
18: vdb

********************************************************************************

INFO: Block device : /dev/ram0
INFO: Invalid partition table or not found for ram0
INFO: Block device : /dev/ram1
INFO: Invalid partition table or not found for ram1
INFO: Block device : /dev/ram2
INFO: Invalid partition table or not found for ram2
INFO: Block device : /dev/ram3
INFO: Invalid partition table or not found for ram3
INFO: Block device : /dev/ram4
INFO: Invalid partition table or not found for ram4
INFO: Block device : /dev/ram5
INFO: Invalid partition table or not found for ram5
INFO: Block device : /dev/ram6
INFO: Invalid partition table or not found for ram6
INFO: Block device : /dev/ram7
INFO: Invalid partition table or not found for ram7
INFO: Block device : /dev/ram8
INFO: Invalid partition table or not found for ram8
INFO: Block device : /dev/ram9
INFO: Invalid partition table or not found for ram9
INFO: Block device : /dev/ram10
INFO: Invalid partition table or not found for ram10
INFO: Block device : /dev/ram11
INFO: Invalid partition table or not found for ram11
INFO: Block device : /dev/ram12
INFO: Invalid partition table or not found for ram12
INFO: Block device : /dev/ram13
INFO: Invalid partition table or not found for ram13
INFO: Block device : /dev/ram14
INFO: Invalid partition table or not found for ram14
INFO: Block device : /dev/ram15
INFO: Invalid partition table or not found for ram15
INFO: Block device : /dev/mtdblock0
INFO: Invalid partition table or not found for mtdblock0
INFO: Block device : /dev/vda
INFO: Partition table type : GPT


INFO: Partition : /dev/vda1 Partition type GUID : EBD0A0A2-
B9E5-4433-87C0-68B6B72699C7 "Platform required bit" : 0
INFO: Performing block read on /dev/vda1 part_guid = EBD0A0A2-
B9E5-4433-87C0-68B6B72699C7
INFO: Block read on /dev/vda1 part_guid = EBD0A0A2-B9E5-4433-87C0-68B6B72699C7
 successful

INFO: Partition : /dev/vda2 Partition type GUID :
 0FC63DAF-8483-4772-8E79-3D69D8477DE4 "Platform required bit" : 0
INFO: Performing block read on /dev/vda2 part_guid =
 0FC63DAF-8483-4772-8E79-3D69D8477DE4
INFO: Block read on /dev/vda2 part_guid = 0FC63DAF-8483-4772-8E79-3D69D8477DE4
 successful
```

```
********************************************************************************

INFO: Block device : /dev/vdb
INFO: Partition table type : GPT


INFO: Partition : /dev/vdb1 Partition type GUID : C12A7328-F81F-11D2-
BA4B-00A0C93EC93B "Platform required bit" : 0
INFO: vdb1 partition is PRECIOUS.
      EFI System partition : C12A7328-F81F-11D2-BA4B-00A0C93EC93B
      Skipping block read...

INFO: Partition : /dev/vdb2 Partition type GUID : B921B045-1DF0-41C3-
AF44-4C6F280D3FAE "Platform required bit" : 0
INFO: Performing block read on /dev/vdb2 part_guid = B921B045-1DF0-41C3-
AF44-4C6F280D3FAE
INFO: Block read on /dev/vdb2 part_guid = B921B045-1DF0-41C3-AF44-4C6F280D3FAE
 successful

********************************************************************************
```

3.10 Run the ACS test suite

See Test with the ACS for instructions on running the ACS test suite. Save the full console log of the ACS test log as `acs-console.log` in the results directory. Also copy the entire contents of the ACS Results filesystem from the ACS drive into the results directory.

3.11 Verify the test results

SystemReady Devicetree results can be verified using an automated script, which detects common mistakes.

To verify the test results:

1. Clone the latest version of the scripts repositories:

```
$ git clone https://gitlab.arm.com/systemready/edk2-test-parser.git
$ git clone https://gitlab.arm.com/systemready/systemready-scripts.git
```

2. Run the script from the `systemready-ir-template` folder, which contains `acs-console.log` and `acs_results`:

```
$ export PATH="$PATH:/path/to/edk2-test-parser"
$ cd systemready-ir-template
$ /path/to/systemready-scripts/check-sr-results.py
WARNING check_file: `./acs_results/linux_dump/lspci.log' empty (allowed)
INFO <module>: 153 checks, 152 pass, 1 warning, 0 error
```

Make sure there are no errors reported, as shown in the example output.

For more information, see the documentation in the systemready-scripts and systemready-ir-template repositories.

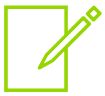
4. Test with the ACS

The ACS ensures architectural compliance across different implementations of the architecture. The ACS is delivered as a prebuilt release image and also with tests in source form within a build environment.

When verifying for SystemReady Devicetree compliance, choose ACS prebuilt image as recommended by Arm SystemReady Requirements Specification.

The image is a bootable live OS image containing a collection of test suites. This collection of Arm Compliance Suites (ACS) includes the BSA, BBR, BBR ACS. These test suites test compliance against the BSA, BBR, EBBR, and BBSR specifications for SystemReady Devicetree Compliance. Arm recommends using architectural implementations to sign off against the ACS to prove compliance with these specifications.

For the latest image, see SystemReady Devicetree band ACS Release details.


Note

For the latest changes to ACS, see https://github.com/ARM-software/arm-systemready/blob/main/changelog.txt

4.1 ACS overview

The SystemReady Devicetree ACS is delivered through a live OS image. This image provides a GRUB menu containing the following options:

- `Linux Boot`
- `bbr/bsa`
- `BBSR Compliance (Automation)`

The default option is `bbr/bsa`, which enables the basic automation to run the BSA and BBR tests. The OS image is a set of UEFI applications on UEFI Shell and Linux kernel with BusyBox integrated with the Firmware Test Suite (FWTS).

The BSA test suites check for compliance with the BSA specification. The tests are delivered through the following suites:

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

The BBR test suites check for compliance with the BBR specification. For compliance, the firmware is tested against the EBBR recipe which contains a reduced subset of UEFI, the BBR, and the EBBR specification. The tests are delivered through two bodies of code:

- EBBR tests contained in UEFI Self-Certification Tests (SCT). UEFI implementation requirements are tested by SCT.
- EBBR based on the FWTS. The FWTS is a package hosted by Canonical that provides tests for UEFI. The FWTS tests are a set of Linux-based firmware tests which are customized to run only UEFI tests applicable to EBBR.

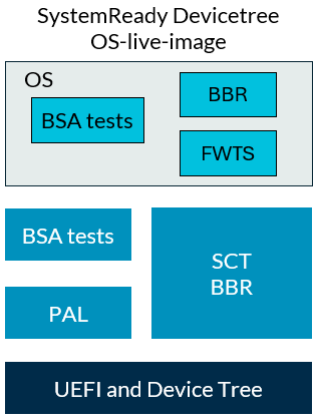
The BBSR test suites check for compliance with the Base Boot Security Requirements (BBSR) specification. These test suites are automatically executed when the `BBSR Compliance (Automation)` GRUB menu option is chosen. When this option is chosen, the ACS attempts to enroll the secure boot keys automatically before running the SCT test suite for SIE. For more details about how to run the SIE tests, see the BBSR and Security Interface Testing FAQ.

If automatic secure boot key enrollment fails, manual enrollment is required, and the BBSR tests must be restarted.

At the end of BBSR tests, the secure boot keys need to be manually cleared.

The following diagram shows the contents of the live OS image:

Figure 4-1: ACS components



4.2 Run the ACS tests

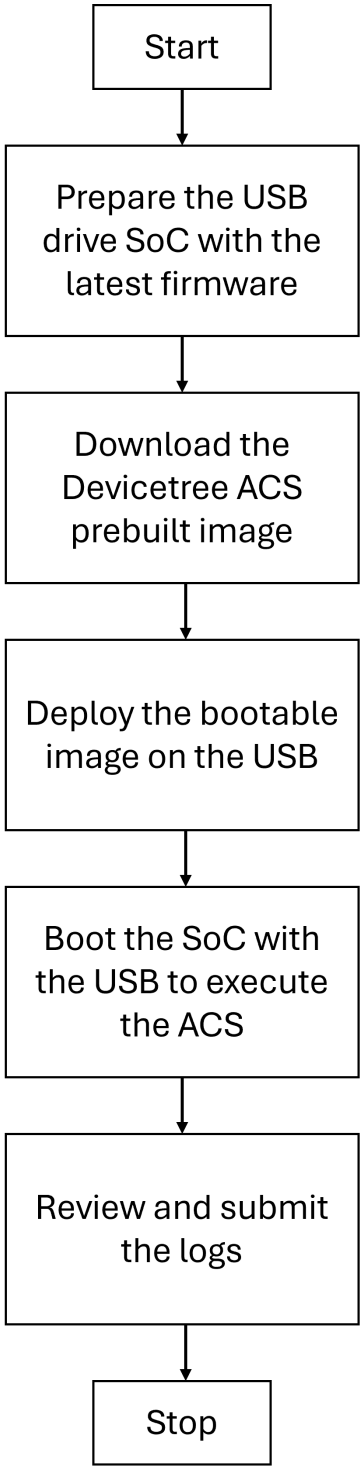
The prerequisites to run the ACS tests are as follows:

- Prepare a storage medium, such as a USB device, with a minimum of 1GB of storage. This storage medium is used to boot and run the ACS and to store the execution results.
- Prepare the System Under Test (SUT) machine with the latest firmware loaded.

- Prepare a host machine for console access to the SUT machine, and collecting the results.
- Configure and capture test waivers as described in: https://github.com/ARM-software/arm-systemready/blob/main/docs/waiver_guide.md

Figure 4-2 shows the ACS test process:

Figure 4-2: ACS test process



The ACS image must be set up on an independent medium or disk, such as a USB device. After the ACS image is written to the disk, it must not be edited again. The U-Boot firmware should be housed in a separate disk to that of the ACS. A storage device with ESP (EFI System Partition) must exist in the system, otherwise the related UEFI SCT tests can fail.

To set up the USB device:

1. Download the latest ACS prebuilt image from the Arm SystemReady Devicetree band prebuilt images repository to a local directory on Linux. For more information about the image releases, see the SystemReady Devicetree band ACS readme.
2. Deploy the ACS image on a USB device. Write the ACS bootable image to a USB storage device on the Linux host machine using the following commands:

```
$ lsblk
$ sudo dd if=/path/to/systemready-dt_acs_live_image.wic of=/dev/sdX
$ sync
```

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

To execute the SystemReady-devicetree Band ACS prebuilt image:

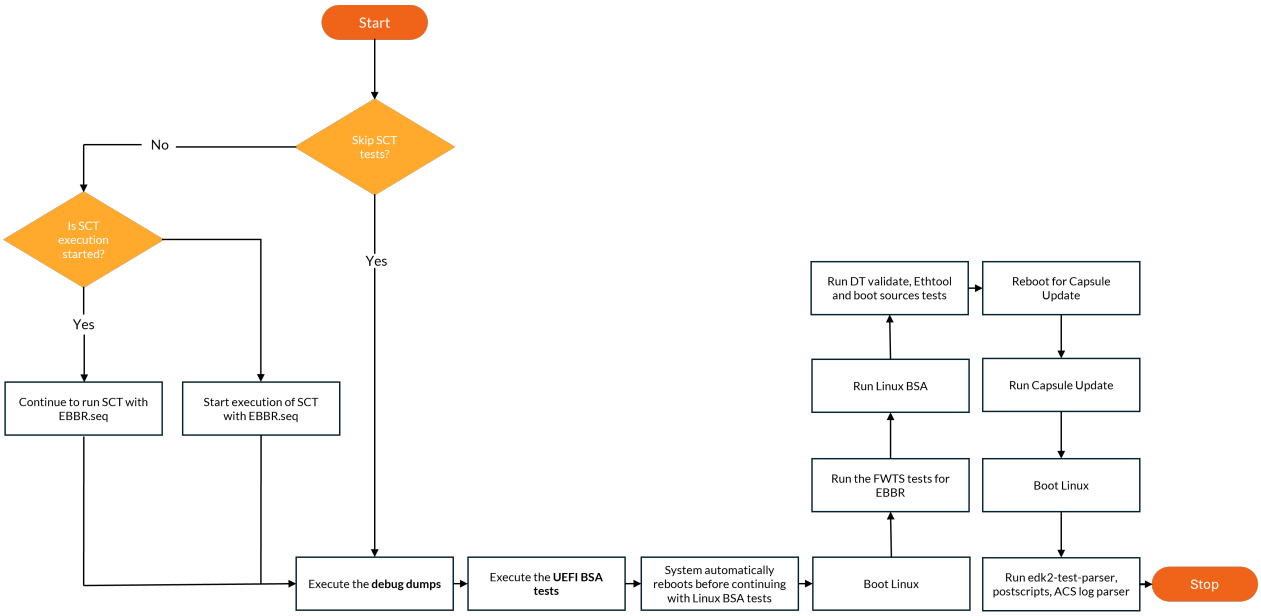
1. Start capturing a log of the serial console output. The log must start from the first power on of the board, and include the finished boot into Linux to run FWTS.
2. Select the option to boot from USB on the SoC.
3. Press any key to stop the boot process and change the `boot_targets` variable to specify the boot device. Use `setenv` to change the `boot_targets` value and `saveenv` to make it the default.
4. If the platform cannot boot from the USB device, use an alternative such as an SD card. If the platform cannot boot, the following message is displayed:

```
U-Boot 2023.07.02 (Oct 07 2023 - 06:23:19 +0000)

CPU:   [CPU Name] rev1.0 at 1200 MHz
Reset cause: POR
Model: [Board Name]
DRAM:  2 GiB
Core:  202 devices, 29 uclasses, devicetree: separate
WDT:   Not starting watchdog@30280000
Loading Environment from MMC... *** Warning - bad CRC, using default environment

In:    serial@30880000
Out:   serial@30880000
Err:   serial@30880000
Net:   eth0: ethernet@30be0000
Hit any key to stop autoboot:  2  0
u-boot=> print boot_targets
boot_targets=usb0 mmc2 mmc0 pxe dhcp
u-boot=> setenv boot_targets usb0 mmc2
u-boot=> saveenv
Saving Environment to MMC... Writing to MMC(2)... OK
u-boot=> boot
starting USB...
Bus usb@32e40000: USB EHCI 1.00
Bus usb@32e50000: USB EHCI 1.00
scanning bus usb@32e40000 for devices... 2 USB Device(s) found
scanning bus usb@32e50000 for devices... 5 USB Device(s) found
       scanning usb for storage devices... 2 Storage Device(s) found
...
```

5. Insert the USB device in one of the USB slots and start a power cycle. The live image boots to run the ACS.

Figure 4-3 shows the complete ACS execution process through the SystemReady-devicetree band ACS live image:

Figure 4-3: ACS execution process



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

New flashing method: using Rufus on Windows OS

If using a Windows host machine, you can write the ACS image to a USB storage device using the Rufus utility:

1. Download Rufus from the official website: https://rufus.ie
2. Insert a USB drive (minimum 1 GB)
3. Open Rufus, and select the ACS image file (*.wic)
4. Select the USB device from the device dropdown
5. Click START to flash the image
6. Safely eject the USB device once the process completes

4.3 Run ACS in automated mode

If you do not choose an option in the GRUB menu and do not skip any tests, the image runs the ACS in the following order:

1. SCT tests
2. Debug dumps
3. BSA ACS

4. Linux boot
5. Linux debug dump
6. FWTS tests
7. BSA tests
8. DT validate tool
9. DT kernel Kselftest
10. Ethtool
11. block device check script
12. System will automatically reboots for capsule update testing
13. Capsule Update test
14. Linux boot
15. edk2-test-parser
16. systemready-scripts
17. acs log parser

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

4.4 Run ACS in normal mode

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

- `Linux Boot` to execute FWTS and BSA
- `bbr/bsa` to execute the tests in the same sequence as fully automated mode
- `BBSR Compliance Automation` to execute BBSR tests, including authenticated variable tests, secure boot, TCG2 protocol test, FWTS, and TPM2 test.

4.4.1 ACS waiver application flow

Waivers are automatically applied as part of automation run, based on user input waiver file. The file should be kept at following location in the image: /mnt/acs_tests/config/acs_waiver.json.

On further details on waiver application process please check this guide: https://github.com/ARM-software/arm-systemready/blob/main/docs/waiver_guide.md

4.5 Review the ACS logs

The logs are stored in a separate partition in the image called `acs_results_template/acs_results`.

After the automated execution, the results partition `acs_results` is automatically mounted on `/mnt`. Navigate to `acs_results` to view the logs, as Figure 4-4 shows:

Figure 4-4: acs_results_template folder contents

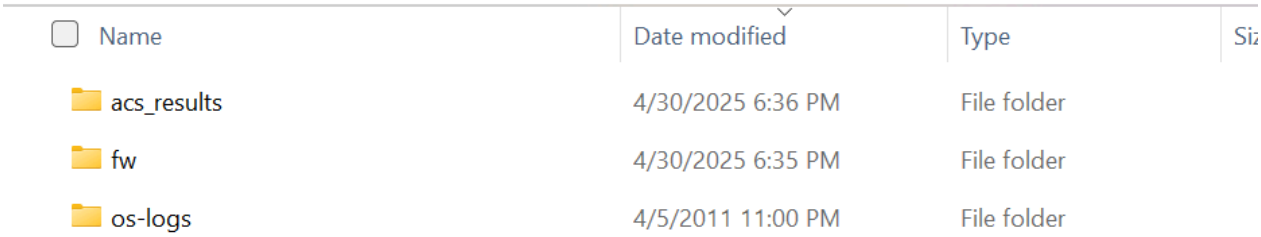
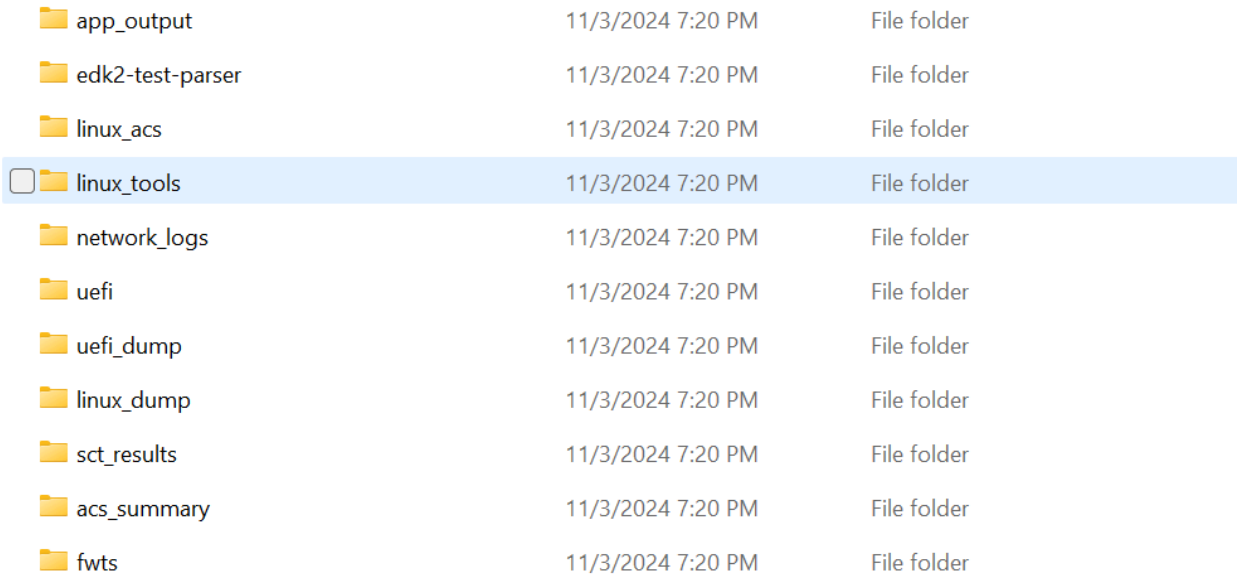


Figure 4-5 shows the contents of the acs_results folder::

Figure 4-5: acs_results folder contents



You can also extract the logs from the USB key on the host machine.

Check for the generation of the following logs after mounting the `acs_results` directory as Table 4-1 shows.

Table 4-1: Logs description

Number	ACS	Full log path	Running time	Description
1	BSA(UEFI)	`acs_results_template/acs_results/uefi/BsaResults.log` `acs_results_template/acs_results/uefi/BsaDevTree.dtb`	Less than two minutes	`BsaDevTree.dtb` is the dumped block of Devicetree
2	SCT	`acs_results_template/acs_results/sct_results/Summary.log`	Four to six hours	`Summary.log` contains the summary of all tests run. Logs of individual SCT test suites can be found in the same path.
3	SCT results parse	`acs_results_template/acs_results/edk2-test-parser/edk2-test-parser.log`	Less than two minutes	`edk2-test-parser.log` contains the edk2-test parser summary of SCT result.
4	FWTS	`acs_results_template/acs_results/fwts/FWTSResults.log`	Less than two minutes	`FWTSResults.log` contains a summary table and output of the Firmware Test Suite results.
5	Debug Dumps	`acs_results_template/acs_results/linux_dumps acs_results_template/acs_results/uefi_dumps`	Less than two minutes	Contains dumps of the `lspci` command, `drivers`, `devices`, `memmap`, and other files.
6	Linux tools	`acs_results_template/acs_results/linux_tools`	Less than two minutes	Contains the logs of `dt-validate`, `ethtool-test.py`, `device_driver_info.sh` `read_blk_devices.py`, and `device_tree.dts` file
7	ESRT	`acs_results_template/acs_results/app_output`	Less than two minutes	Contains the logs of ESRT and FMP tests.
8	SCT for BBSR	`acs_results_template/acs_results/BBSR/sct_results/Overall/Summary.log`	Less than four minutes	`Summary.log` contains the summary of all tests run.
9	FWTS for BBSR	`Summary.log` contains the summary of all tests run	Less than two minutes	FWTSResults.log contains a summary table and output of the Firmware Test Suite results

10	Capsule Update	`acs_results_template/acs_results/fw/capsule-on-disk`
		
		`acs_results_template/acs_results/fw/capsule-update`
		
		`acs_results_template/acs_results/fw/capsule_test_results`
			Less than five minutes	Results of Capsule Update testing

Number	ACS	Full log path	Running time	Description

11	PSCI	`acs_results_template/acs_results/linux_tools/psci/psci`
		
		`acs_results_template/acs_results/linux_tools/psci psci_kernel`
			Less than one minute	Result of PSCI checker and version

12	ACS JSON and HTML results	`/mnt/acs_results/BBSR/fwts/FWTSResults.log`	Less than ten minutes	`/mnt/acs_results/acs_summary/acs_jsons`
				
				The directory contains the ACS test tools results in JSON standard format.
				
				`/mnt/acs_results/acs_summary/html_detailed_summary`
				
				The directory contains the ACS test tools results in HTML format. It includes a ACS summary and detailed individual test suite report.

4.6 ACS logs

If any logs are missing, run the suite manually. To report the error, mount the `boot` partition to copy the `acs_results_template` folder to a local directory, then submit the logs.

Use an SSD in a USB enclosure to execute the SCT tests more quickly. The SCT tests result are parsed by `edk2-test-parser` script tool automatically in the SystemReady-devicetree band ACS Linux shell.

The ACS image includes log parser scripts that process raw logs from individual tests. These scripts generate detailed HTML pages for each test and an overall compliance report, summarized in the acs_summary.html page.

The results are available in the `/mnt/acs_results_template/acs_results/acs_summary`.

Please note that some results require manual interpretation, such as:

Test	Log Location	Check
SMBIOS	`/mnt/acs_results_template/acs_results/uefi_dump/smbiosview.log`	smbios table as per BBR specification
OS-Logs	`/mnt/acs_results_template/os-logs/linux-*/ethtool_test /mnt/acs_results_template/os-logs/linux-*/boot_source`	Manually check the logs for any failure Manually check the logs for any failure

4.6.1 ACS Configs

The following are the ACS configuration files:

- `acs_config_dt.txt`: The file specifies the ARM specification version that the ACS tool suite complies with, and this information is included in the System_Information table of the ACS_Summary.html report.
- `system_config.txt`: The file is used to collect below system information which is required for ACS_Summary.html report, this needs to be manually filled by user.
  - FW source code: Unknown
  - Flashing instructions: Unknown
  - product website: Unknown
  - Tested operated Systems: Unknown
  - Testlab assistance: Unknown

5. Related information

The following resources are related to material in this guide.

Specifications:

- Arm Base System Architecture (BSA) specification
- Arm Base Boot Requirements (BBR) specification
- Base Boot Security Requirements (BBSR)
- Arm SystemReady Requirements Specification

Repositories:

- Arm SystemReady ACS Repository
- Embedded Base Boot Requirements (EBBR) Repository

User Guides:

- Arm SystemReady Devicetree Band Compliance Policy Guidelines
- SystemReady FAQ

SystemReady Pages:

- SystemReady Devicetree Band Page
- Arm SystemReady Compliance Program
- Arm Community - SystemReady Forum

Other Resources:

- U-Boot Repository

6. Next steps

In this guide, you learned how to prepare for SystemReady Devicetree compliance and how to perform the tasks needed for the compliance program. This band is for devices in the IoT edge sector that are built around SoCs based on the Arm A-profile architecture. The SystemReady Devicetree band ensures interoperability with embedded Linux and other embedded operating systems.

After reading this guide, you can find more information about compliance registration at Arm SystemReady Program.

For support with the ACS, e-mail support-systemready-acs@arm.com.

Appendix A Build firmware for Compulab IOT-GATE-IMX8 platform

This section provides an example of how to build compliant firmware for an i.MX8M platform, specifically for the IOT-GATE-iMX8 from Compulab.

Use the following commands to fetch the relevant reference source code and build the reference firmware:

```
$ sudo apt install swig   # if the swig package is missing for Ubuntu
$ git clone https://git.linaro.org/people/paul.liu/systemready/build-scripts.git/
$ cd build-scripts
$ ./download_everything.sh
$ ./build_everything.sh
```

By default, the generated binary images are in the following directories:

- `/tmp/uboot-imx8/flash.bin`
- `/tmp/uboot-imx8/u-boot.itb`
- `/tmp/uboot-imx8/capsule1.bin`

For more information about how to test SCT on an i.MX8 board, see the following repositories:

- iot-gate-imx8
- Building and running iot-gate-imx8

Appendix B Run the SystemReady-devicetree band ACS image on simulator

Running the SystemReady-devicetree band ACS image on simulator involves the following steps:

1. Prepare the SystemReady-devicetree band ACS live image
2. Compile the U-Boot firmware
3. Execute the ACS on Simulator

It is possible to run either on [QEMU] or on [FVP].

B.1 Prerequisite

To test SystemReady Devicetree band on simulator, use steps:

1. Use a PC running [Ubuntu 24.04 LTS]
2. Install the following packages:

```
$ sudo apt install bash bc binutils build-essential bzip2 cpio diffutils \
           expect file findutils g++ gcc git gzip make patch perl rsync sed \
           tar telnet unzip wget xterm xz-utils
```

B.1.1 Install the FVP simulator

To run on FVP, download the FVP simulator from the Arm website and extract the archive:

```
$ wget https://developer.arm.com/-/cdn-downloads/permalink/Fixed-Virtual-Platforms/
FM-11.27/FVP_Base_RevC-2xAEMvA_11.27_19_Linux64.tgz
$ tar xf FVP_Base_RevC-2xAEMvA_11.27_19_Linux64.tgz
```

The FVP model executable is `Base_RevC_AEMvA_pkg/models/Linux64_GCC-9.3/FVP_Base_RevC-2xAEMvA`.

B.2 Prepare the SystemReady-devicetree band ACS live image

Arm provides the Devicetree ACS live image, prepare it as the following steps:

1. Download the prebuilt Devicetree ACS image from arm-systemready github. Arm recommends that you select the latest prebuilt Devicetree ACS image to download.
2. Uncompress the image with the following command:

```
$ xz -d systemready-dt_acs_live_image.wic.xz
```

This image comprises two file system partitions recognized by UEFI:

/
: Stores rootfs of Linux and test-suites to run in Linux environment.

boot
: Contains bootable applications and test suites. The `bbr` and `bsa` test applications are stored in this partition under the `EFI/BOOT` directory. Contains a 'acs_results' directory which stores logs of the automated execution of ACS. Approximate size: 150 MB.

B.3 Compile the U-Boot firmware

You can build the U-Boot firmware with [Buildroot].

More information on Buildroot is available in [The Buildroot user manual].

To download Buildroot, do the following:

```
$ git clone https://gitlab.com/buildroot.org/buildroot.git -b 2025.08.x
$ cd buildroot
```

B.3.1 Compile the firmware to run on Qemu

To build the firmware code to run on Qemu, do the following:

```
$ make qemu_aarch64_ebbr_defconfig
$ make
```

When the build completes, it generates the firmware file `output/images/flash.bin`, comprising TF-A, OP-TEE and the U-Boot bootloader. A QEMU executable is also generated at `output/host/bin/qemu-system-aarch64`.

Specific information for this Buildroot configuration is available in the file `board/qemu/aarch64-ebbr/readme.txt`.

B.3.2 Compile the firmware to run on FVP

To build the firmware code to run on FVP, do the following:

```
$ make arm_fvp_ebbr_defconfig
$ make
```

When the build completes, it generates the firmware file `output/images/bl1.bin` and `output/images/fip.bin`, comprising TF-A, OP-TEE and the U-Boot bootloader.

Specific information for this Buildroot configuration is available in the file `board/arm/fvp-ebbr/readme.txt`.

B.4 Execute the ACS on simulator

This section shows how to execute the ACS on simulator

B.4.1 Execute on QEMU

Launch Qemu using the following command:

```
$ ./output/host/bin/qemu-system-aarch64 \
    -M virt,secure=on,acpi=off \
    -bios output/images/flash.bin \
    -cpu cortex-a53 \
    -device virtio-blk-device,drive=hd1 \
    -device virtio-blk-device,drive=hd0 \
    -device virtio-net-device,netdev=eth0 \
    -device virtio-rng-device,rng=rng0 \
    -drive file=<path-to/ir-acs-live-image-generic-
arm64.wic>,if=none,format=raw,id=hd0 \
    -drive file=output/images/disk.img,if=none,id=hd1 \
    -m 2048 \
    -netdev user,id=eth0 \
    -nographic \
    -object rng-random,filename=/dev/urandom,id=rng0 \
    -rtc base=utc,clock=host \
    -smp 2
```

The SystemReady-devicetree band ACS starts, as Figure B-1 shows:

Figure B-1: SystemReady-devicetree band ACS image starting on QEMU

The EFI System Partition (ESP) in use is the one created by Buildroot in the OS image file `disk.img`.

B.4.1.1 Troubleshooting QEMU advice

If the ACS halts at the following BSA test, restart `qemu-system-aarch64` to finish running the ACS.

```
 502 : Wake from System Timer Int
       Checkpoint --  1                          : Result:  SKIPPED
 503 : Wake from EL0 PHY Timer Int
```

B.4.2 Execute on FVP

Launch the FVP using the following command:

```
<path-to/FVP_Base_RevC-2xAEMvA> \
    --config-file board/arm/fvp-ebbr/fvp-config.txt \
    -C bp.secureflashloader.fname="output/images/bl1.bin" \
    -C bp.flashloader0.fname="output/images/fip.bin" \
    -C bp.virtioblockdevice.image_path=<path-to/ir-acs-live-image-generic-arm64.wic>
```

There is no EFI System Partition (ESP) on FVP at this point.

- FVP: https://developer.arm.com/Tools%20and%20Software/Fixed%20Virtual%20Platforms
- QEMU: https://www.qemu.org
- Ubuntu 24.04 LTS: https://releases.ubuntu.com/
- Buildroot: https://buildroot.org
- The Buildroot user manual: https://buildroot.org/downloads/manual/manual.html

Appendix C Rebuild the SystemReady-devicetree band ACS image

SystemReady ACS GitHub contains the prebuilt image. For details of the latest version for download, see the release details. For debug purposes, if you want to rebuild the SystemReady devicetree ACS image, see the steps from the ACS build steps in the SystemReady documentation.

C.1 Prerequisites

Before starting the ACS build, ensure that the following requirements are met:

- Ubuntu 22.04 LTS with a minimum of 32GB free disk space
- Bash shell
- Sudo privilege to install tools required for build
- Git is installed

If Git is not installed, install Git using `sudo apt install git`. Also, run the `git config --global user.name "Your Name"` and `git config --global user.email "Your Email"` commands to configure your Git installation.

C.2 Build the SystemReady-devicetree band ACS live image

To build the live image:

1. Clone the `arm-systemready` repository using the following code with the latest release tag, for example `v24.11_SR_REL3.0.0-BETA0_SR-DT_REL3.0.0-BETA0`:

```
git clone https://github.com/ARM-software/arm-systemready.git \
        --branch <release_tag>
```

2. Navigate to the `SystemReady-devicetree band/Yocto directory` directory:

```
cd arm-systemready/SystemReady-devicetree-band/Yocto
```

3. Run `get_source.sh` to download the sources and tools for the build. Provide the sudo password if prompted:

```
./build-scripts/get_source.sh
```

4. To start building the SystemReady-devicetree band ACS live image, use the following command:

```
./build-scripts/build-systemready-dt-band-live-image.sh
```

If this procedure is successful, the bootable image is available at `/path-to-arm-systemready/SystemReady-devicetree-band/Yocto/meta-woden/build/tmp/deploy/images/generic-arm64/systemready-dt_acs_live_image.wic.xz`.


Note

The image is generated in a compressed (`.xz`) format. You must uncompress the image before using it. You can use the following command to uncompress the image:

```
xz xz -d systemready-dt_acs_live_image.wic.xz
```

C.3 Troubleshooting advice

When building SystemReady-devicetree band ACS image in step 4, you might see a kernel download error:

```
Resolving cdn.kernel.org (cdn.kernel.org)... failed: Name or service not known.
wget: unable to resolve host address 'cdn.kernel.org'
```

If you see this error, clone the latest `arm-systemready` repository code with the following command:

```
git clone https://github.com/ARM-software/arm-systemready.git
```

Then continue from step 2.

Appendix D Test checklist

The following checklist summarizes the steps you must take to test your system for SystemReady Devicetree compliance:

1. Perform U-Boot sanity tests manually as described in the [Test the U-Boot Shell] section in Test SystemReady Devicetree band.
2. Perform UEFI sanity tests manually as described in the [Test the UEFI Shell] section in Test SystemReady Devicetree band.
3. Perform capsule update manually as described in the Test UpdateCapsule section in Test SystemReady Devicetree band.
4. Run the automated ACS-IR as described in the Run the ACS test suite section in Test SystemReady Devicetree band.
5. Optionally perform SCT test for BBSR compliance as described in BBSR and Security Interface Testing FAQ.
6. Optionally perform Linux BBSR FWTS and Secure firmware update tests as described in SystemReady Security Interface Extension User Guide.
7. Install two Linux distributions and perform OS tests manually as described in the Test installation of Linux distributions section in Test SystemReady Devicetree band.
8. Verify your test results using the scripts as described in the Verify the test results section in the Test SystemReady Devicetree band and the Review the ACS logs section in Test with the ACS.

Appendix E Steps to run edk2-test Parser manually

This section describes how to run `edk2-test parser` script tool to parse the SCT test results.

To run the edk2-test Parser tool:

1. Clone the latest version of the parser:

```
$ git clone https://git.gitlab.arm.com/systemready/edk2-test-parser.git
```

2. Run the parser from the `acs_results` folder:

```
$ cd acs_results
$ /path/to/edk2-test-parser/parser.py sct_results/Overall/Summary.ekl \
 sct_results/Sequence/EBBR.seq
INFO ident_seq: Identified `sct_results/Sequence/EBBR.seq' as
 "ACS-IR v23.03_IR_2.0.0 EBBR.seq".
INFO apply_rules: Updated 55 tests out of 10657 after applying 144 rules
INFO print_summary: 0 dropped, 0 failure, 51 ignored, 1 known acs limitation,
 3 known u-boot limitations, 10602 pass, 0 warning
```

3. Make sure the sequence file is recognized correctly, and that there are no dropped, skipped, failures, or warnings reported.

Run the edk2-test Parser tool to parse the logs further, based on YAML configurations.

For more information, see the document in the EDK2 SCT Results Parser repository.

Appendix F Deploying Yocto on SystemReady-compliant hardware

The Yocto Project (YP) is an industry standard development tool to build Linux-based software stacks for embedded devices. YP provides the flexibility to create custom solutions, however most YP builds require custom engineering to run on a specific hardware platform. As a result, it is difficult to support many targets with a single configuration. On SystemReady compliant platforms, YP builds rely on consistent boot behavior, a firmware-provided system description, and mainline Linux support to eliminate per-platform enablement. SystemReady reduces the effort for maintenance and can support many platforms with a single image.

This section provides further detail and guidance on the following:

- The Yocto Project (YP) and SystemReady
- A reference deployment example on an Arm-based NXP board
- Information about where to find the necessary components to get YP up and running on SystemReady compliant platforms

F.1 Yocto Project overview

The Yocto Project helps developers build custom embedded Linux distributions. It is popular due to its modularity, which allows you to optimize speed, footprint, and memory utilization. The Yocto Project contains the following key elements:

- Tools for Linux development
- Poky, a reference embedded distribution
- OpenEmbedded build system

Poky is Yocto's stable reference OS, which demonstrates a basic level of functionality for embedded systems. Poky combines core components from the Yocto Project, is tested and supported, and receives frequent updates. Developers can use Poky as a foundation and adapt it to meet their requirements. Underpinning Yocto's modularity is the Layer Model, which allows for functionality to be logically organized into layers. Layers group related recipes and tell the build system what to make. Recipes are a form of metadata, which contain instructions on where to find the source code and information on dependencies and compilation options.

The OpenEmbedded layer index provides an easy way to find layers, such as Board Support Packages (BSP), GUIs, middleware, and the Poky layer. The Yocto Project Compatible Layer Index includes a curation of layers validated to work with Yocto. Combining pre-built layers with custom built layers, developers can build an entire distribution. The layer system creates a logical hierarchy which enables collaboration and reuse of code, whilst simplifying the overall view of the software.

An example hierarchy can include the following layers:

- Developer Specific Layer, a custom functionality for the specific product requirements
- Poky, a reference OS to act as a foundation
- Hardware Specific BSP provided by the silicon vender or ODM
- Yocto Specific Layer, which are recipes specific to Yocto builds
- OpenEmbedded-core, a small set of foundational recipes consistent across OpenEmbedded derived builds

The OpenEmbedded Build System uses the BitBake tool. BitBake parses recipes to compile a final image either through native or cross compilation. For more detailed information about Yocto, see the Yocto documentation.

F.2 SystemReady for Yocto

Yocto offers an easy way to build custom OS images. Embedded developers often support many platforms with different hardware and firmware, creating bespoke Yocto images for each configuration. SystemReady is designed to address this complexity. SystemReady compliant platforms expose a standardized set of interfaces to the OS so that Yocto builds and other off-the-shelf Linux distributions can boot without modifications. Embedded platforms can target SystemReady, however Yocto builds can boot without modification across platforms compliant with SystemReady.

SystemReady provides an effective solution to the issue of fragmentation for embedded developers, offering a software experience that works while retaining the customizability that Yocto is known for. With this combination, you can support large, diverse deployments with substantially reduced effort.

F.3 Build a generic SystemReady Yocto image

This is a simple guide how to build a base generic Yocto image that will boot on any SystemReady compatible platform.

1. Make sure all the Yocto build prerequisites are met, as described in Yocto Project Quick Build.
2. Set up Poky using the following code:

```
$ git clone git://git.yoctoproject.org/poky
$ cd poky
$ source oe-init-build-env
```

3. Set the `MACHINE` config value in the `conf/local.conf` config file to `genericarm64`, as shown in the following snippet:

```
$ echo 'MACHINE ?= "genericarm64"' >> conf/local.conf
```

genericarm64 is a generic SystemReady aarch64 machine.

4. Change the `init` system from `sysvinit` to `systemd`, as shown in the following code:

```
$ echo 'INIT_MANAGER = "systemd"' >> conf/local.conf
```

   This system can properly detect the console tty from the kernel.

5. Build the Yocto image using the following code:

```
$ bitbake core-image-base
```

   After the build successfully finishes the Yocto image can be found as `<path to yocto repo>/build/tmp/deploy/images/genericarm64/core-image-base-genericarm64.rootfs.wic`

   Although Yocto is often used to build firmware images, genericarm64 machine focuses on OS image only. The expectation is that real hardware comes with the firmware pre-loaded.

6. Genericarm64 images for specific projects

The genericarm64 machine definition can be easily used to build Yocto images for specific projects. For example, to build a Yocto image including Parsec and AWS Greengrass, in addition to the steps described above you also need:

- Clone and add OpenEmbedded, Security, clang and AWS Yocto layers using the following code:

```
git clone git://git.openembedded.org/meta-openembedded
git clone git://git.yoctoproject.org/git/meta-security
git clone https://github.com/aws4embeddedlinux/meta-aws.git
git clone https://github.com/kraj/meta-clang.git
cd poky
source oe-init-build-env
bitbake-layers add-layer ../../meta-openembedded/meta-oe
bitbake-layers add-layer ../../meta-openembedded/meta-python
bitbake-layers add-layer ../../meta-openembedded/meta-networking
bitbake-layers add-layer ../../meta-openembedded/meta-multimedia
bitbake-layers add-layer ../../meta-clang
bitbake-layers add-layer ../../meta-aws
bitbake-layers add-layer ../../meta-security/meta-tpm
bitbake-layers add-layer ../../meta-security/meta-parsec
```

- Include Parsec and AWS Greengrass into the Yocto image:

```
echo '
# TPM
DISTRO_FEATURES:append = " tpm2"
IMAGE_INSTALL:append = " tpm2-tools"

# PARSEC
IMAGE_INSTALL:append = " parsec-service parsec-tool"

# AWS greengrass and command line utility
IMAGE_INSTALL:append = " aws-cli greengrass-bin"' >> conf/local.conf
```

The result image can be used to boot any SystemReady compliant platform and provision it as an AWS IoT thing.

1. Pre-built Generic arm64 Yocto images.

Yocto project publishes pre-built Yocto images which can be used for initial testings. Genericarm64 images with systemd init system can be found in Yocto 5.0.5 genericarm64-alt images or later Yocto releases.

F.4 Example: deployment on an NXP board

Before you begin, you will need the following:

- An NXP i.MX 8M Mini EVK board
- A micro SD card that is 2GB or larger
- A computer running a Linux environment

Make the board SystemReady Devicetree compatible

To make the board SystemReady Devicetree compatible, do the following:

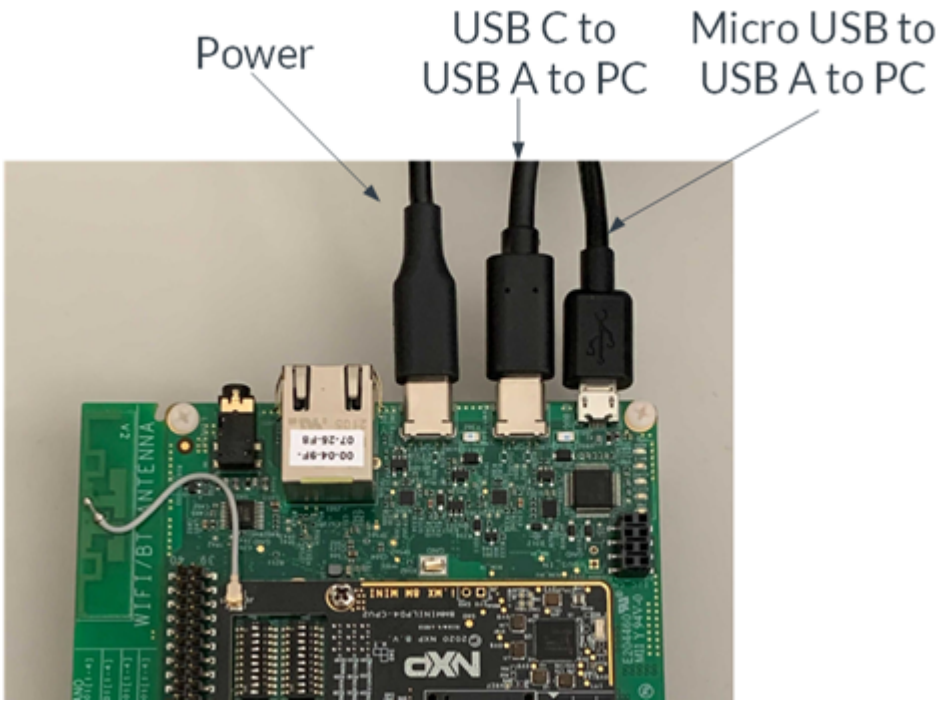
1. Ensure you have an NXP account.
2. Download and extract the i.MX 8M Mini EVK boot image (SystemReady certified) from Embedded Linux for i.MX Applications Processors.
3. Download the uuu tool from the mfgtools GitHub repository. This tool is used to program the onboard eMMC.
4. On the NXP board, slide the power switch to the off position and set the boot mode switches to Download mode. The following table shows the switch settings:

	SW1	SW2	SW3	SW4	SW5	SW6	SW7	SW8	SW9	SW10
Top row	1	0	1	0	X	X	X	X	X	X
Bottom row	X	X	X	X	X	X	X	X	X	0

1=Switch Up, 0= Switch Down, X= Either

5. Connect the USB-C power cable to the power supply, the USB-C USB cable to the PC, and the USB Micro cable to the PC (serial). The following diagram shows how the cables should be connected:

Figure F-1: An image showing how to connect the USB cables



6. Slide the power switch to the on position and flash the boot firmware as shown in the following code snippet:

```
$ sudo uuu -b emmc imx-boot-imx8mmevk-sd.bin-flash_evk
```

   The SystemReady Devicetree compatible version of U-Boot is installed to the onboard eMMC.

7. Slide the power switch to the off position and set the boot mode switches to eMMC mode, as shown in the following table:

	SW1	SW2	SW3	SW4	SW5	SW6	SW7	SW8	SW9	SW10
Top row	0	1	1	0	1	1	0	0	0	1
Bottom row	0	0	0	1	0	1	0	1	0	0

   1=Switch Up, 0= Switch Down

The board is now ready to boot SystemReady compatible operating systems from an SD Card or USB.

Boot a generic SystemReady Yocto image

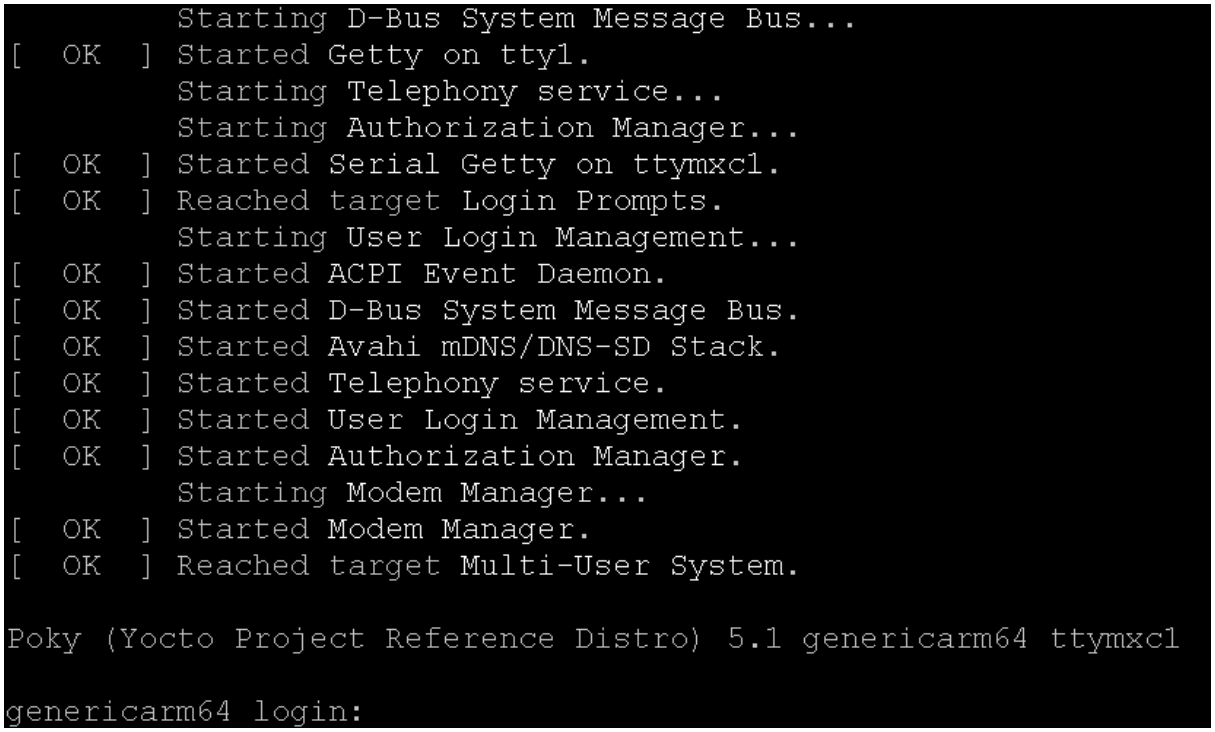
Now that the board is SystemReady Devicetree compatible, the next step is to boot the base generic Yocto image built following the Build a generic SystemReady Yocto image guide.

1. Copy the image to a micro SD card using the following code:

```
sudo dd if=<path to yocto repo>/build/tmp/deploy/images/genericarm64/core-image-base-genericarm64.rootfs.wic of=/dev/sdX bs=4M
```

2. Insert the microSD card into the development board and power it on. As shown in the following screen shot, the board shows U-boot, the systemd-boot menu, and a login shell for Yocto:

Figure F-2: A screenshot showing the Yocto login shell



The Yocto image is now installed on the NXP board.

F.5 The meta-arm layer

The meta-arm is a layer with recipes specific for Arm platforms, and contains qemuarm64-secureboot and qemuarm64-sbsa QEMU machines.

The following table describes the recipes in the meta-arm layer:

Recipe	Description
Android Common Kernel	Downstream of kernel.org kernels, including selected patches that have not been merged into the mainline or Long Term Supported (LTS) kernel
Arm FVP - Architecture Envelope Model	Support for Fixed Virtual Platform (FVP) Architecture Envelope Models (AEM)

Recipe	Description
Arm FVP - Library Ecosystem Reference Design	Support for Arm FVP library, which includes all CPU FVPs
DS-5 Streamline Gator daemon	Daemon for gathering data for Arm Streamline Performance Analyser (part of Arm Development Studio)
Hafnium	A reference Secure Partition Manager (SPM) for systems that implement the Armv8.4-A Secure-EL2 extension, enabling multiple, isolated Secure Partitions (SPs) to run at Secure-EL1
OpenCSD	API for decoding trace streams from Arm CoreSight trace hardware
OP-TEE	Trusted Execution Environment (TEE) designed as companion to a non-secure Linux kernel running on Cortex-A cores using TrustZone
SCP Firmware	System Control Processor (SCP) and Manageability Control Processor (MCP) firmware reference implementation
Tianocore EDK2	Open-source implementation of UEFI
Trusted Firmware-A	Reference implementation of secure world software for Cortex-A
Trusted Firmware for Cortex-M	Reference implementation of secure world software for Cortex-M

Appendix G Document Revisions

This section contains changes between current version and previous versions.

Table G-1: Changes between v2.0 EAC and v2.1 EAC

Changes	Topics affected
Added key enrollment automation test	• ACS overview
Added how to test boot sources	• Boot sources tests
Added how to test ethernet port	• Ethernet port Test

Table G-2: Changes between ACS v2.1.0 and ACS v2.0.0

Changes	Topics affected
Added the automated Secure Boot key enrollment	• ACS overview
Added the automated running of edk2-test parser and saving of results in acs_results	• ACS logs • Steps to run edk2-test Parser manually
Added the automated running of ethtool.py to test the ethernet ports	• Ethernet port Test
Added the automated discovery and read of block devices and identifying “precious” partitions	• Boot sources tests
Added the saving of /sys folder and device to driver mapping logs	• Test installation of Linux distributions