SystemReady IR IoT Integration, Test, and Certification Guide

Version 2.0

Non-Confidential

Issue

DUI1101_2.0_en

SystemReady IR IoT Integration, Test, and Certification 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

Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

Confidentiality Status

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

Unrestricted Access is an Arm internal classification.

Product Status

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

Feedback

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

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

Inclusive language commitment

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

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1. Overview

SystemReady is a compliance certification program 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 Base System Architecture (BSA), Base Boot Requirements (BBR), Security Interface Extension (SIE) specifications and market-specific supplements.

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

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

1.1 Before you begin

This guide is specific for SystemReay IR.

Figure 1-1: SystemReady IR Logo

This guide explains how to configure a U-Boot-based platform for SystemReady IR compliance, and how to run all the SystemReady IR tests before submitting the platform for certification in the Arm SystemReady Certification Program.

Note

This guide assumes U-Boot firmware and the examples shown are captured on a U-Boot platform. However, SystemReady IR compliance can be achieved with any UEFI-compliant firmware. The use of U-Boot is not mandatory. You can also use EDK2 or another firmware implementation for certification. If you are not using U-Boot, you can ignore the Configure U-Boot for SystemReady section.

By the end of this guide, you will be able to perform the following tasks which are required for certification:

- Enable Unified Extensible Firmware Interface (UEFI) features in U-Boot.
- Run 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.
- 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 certification and testing requirements, see the Arm SystemReady Requirements Specification.

SystemReady certified 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 like 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 IR platforms comply with the Arm Base System Architecture (BSA) specification. SystemReady IR v2.0 certification does not require BSA compliance, but for certification the BSA compliance tests must be run and the results submitted. BSA compliance will become a requirement in a future version of SystemReady IR.
- Arm recommends that SystemReady IR platforms comply with the Base Boot Security Requirements (BBSR). This compliance is currently recommended, not mandatory.

2. Configure U-Boot for SystemReady

This section of the guide explains how to enable the U-Boot configuration options required for SystemReady IR certification.

These required configuration 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 SIE option)

This section of the guide 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 IR v2.0 certification. U-Boot releases and patches can be found in the U-Boot git repository. Instructions for porting and building U-Boot is beyond the scope of this document. Refer to 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 IR certification.

To configure UEFI support in U-Boot:

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

```
// Core UEFI features
CONFIG_BOOTM_EFI=y
CONFIG_CMD_BOOTEFI=y
CONFIG_CMD_NVEDIT_EFI=y
CONFIG_CMD_EFIDEBUG=y
CONFIG_CMD_BOOTEFI_HELLO=y
CONFIG_CMD_BOOTEFI_HELLO_COMPILE=y
CONFIG_CMD_BOOTEFI_SELFTEST=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_CMD_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. UEFI boot can be tested 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.

2.3 Device Firmware Upgrade

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

This section of the guide splits the configuration process into two parts:

- Steps that are common to both the signed and unsigned capsule update schemes, described in Common configuration.

- Steps that are specific to the signed capsule update scheme, described in Generate capsule files.

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 explains how to generate three different capsule files:

1. A signed capsule supporting authentication.
2. An unsigned capsule, which should fail authentication.
3. A tampered capsule, which should also fail authentication.

To generate a signed capsule file, do the following:

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.

   Install the required 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"
```

The resulting `signed_capsule.bin` binary can be used to update the firmware with UEFI capsule update, as described in Test SystemReady IR.

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`, refer to Building tools for Linux, in the U-Boot documentation. Alternatively, you can use the `GenerateCapsule` tool from EDK2 to create a UEFI Capsule binary.

Note

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.

The signing public key is needed to authenticate the signed capsule firmware. Insert the signing public key into a `dtb` file.

First, create a `signature.dts` file:

```
/dts-v1/;
/plugin/;
&{/} {
     signature {
               capsule-key = /incbin/("CRT.esl");
```

```
        };
};
```

Next, 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. This `new.dtb` file is used in Test UpdateCapsule.

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 IR compliance certification can benefit from integrating with common FW 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 when certifying with the recommended SIE option. 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, the Devicetree is specified by an argument to the `bootefi` command. This Devicetree can be loaded 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 mentions 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 IR

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

Note

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

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

- Provided by your platform vendor:
    - The platform under test with firmware already installed.
    - Three capsule files (signed, unsigned, and tampered) in Firmware Management Protocol (FMP) format. These files were generated in Generate capsule files.
- Provided by Arm:
    - SystemReady IR Test and Certification Guide (this guide).
    - SystemReady reporting template. Use this directory structure to collect all test results.
    - SystemReady results parsing scripts. Use these scripts to check that all required logs are provided and the required tests have passed.
    - Arm SystemReady IR ACS installed on a storage medium. For details about which version of the ACS image should be used, please refer to the Arm SystemReady Requirements Specification.
- Provided by a third party:
    - Two generic Linux ISO distribution images on storage media.

The SystemReady IR tests include the following:

- EFI System Resource Table (ESRT) dump and sanity check
- Devicetree validation
- UEFI Capsule Update tests
- EBBR tests
- UEFI BSA and Linux BSA tests
- Secure boot test (recommended)
- Installation and boot of two different Linux distributions

Before running the tests, clone the SystemReady reporting template repository and use it to capture the test results and logs. Refer to the documentation in the template repository for the latest list of commands to run the tests.

3.1 Test the U-Boot shell

To perform the U-Boot tests:

1. Start a log of all the output from the serial console.
2. Reboot the platform and run the following commands from the U-Boot shell:

```
u-boot=> help
u-boot=> version
u-boot=> printenv
u-boot=> printenv -e
u-boot=> bdinfo
u-boot=> rtc list
u-boot=> sf probe
u-boot=> usb reset
u-boot=> usb info
u-boot=> mmc rescan
u-boot=> mmc list
u-boot=> mmc info
u-boot=> efidebug devices
u-boot=> efidebug drivers
u-boot=> efidebug dh
u-boot=> efidebug memmap
u-boot=> efidebug tables
u-boot=> efidebug query
u-boot=> efidebug boot dump
u-boot=> efidebug capsule esrt
u-boot=> bootefi hello ${fdtcontroladdr}
u-boot=> bootefi selftest ${fdtcontroladdr}
```

3. Save the log as `fw/u-boot-sniff.log` in the results directory.

Always refer to the `README.md` from the SystemReady IR template for the latest manual test instructions.

3.2 Test the UEFI shell

To capture the behavior of the UEFI shell:

1. Capture the output from the serial console and boot into the UEFI shell using the ACS utility, then run the following commands:

```
Shell> dmem
Shell> pci
Shell> drivers
Shell> devices
Shell> devtree
Shell> dmpstore
Shell> dh -d -v
Shell> memmap
Shell> smbiosview
```

2. Save the console log as `fw/uefi-sniff.log` in the results directory.

Note

`FS#` indicates separate partitions, but the numbering might vary on different system.

In this example:

- `FS0` is the `\` partition on the ACS-IR image
- `FS1` is the `boot` partition on the ACS-IR image
- `FS2` is the `results` partition on the ACS-IR image

For more details, see Prepare the ACS-IR live image.

Always refer to the `README.md` from the SystemReady IR template for the latest manual test instructions.

3.3 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 v2.90 (Das U-Boot, 0x20221000)
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)
No SimpleTextInputEx was found. CTRL-based features are not usable.
No SimpleTextInputEx was found. CTRL-based features are not usable.
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  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 ACS-IR 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
```

3.4 Test Devicetree

Devicetree validation ensures that the Devicetree node data matches the schema constraints. This test is performed automatically by the ACS. This section describes how to perform the test manually.

1. Install Devicetree Schema Tools using the following command:

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

   Refer to 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. This step is done automatically by the ACS test which is introduced in Run ACS in automated mode.

4. Mount the ACS-IR 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 follow:

```
$ 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. Warnings should be considered but are informative. For more information about how to download this script, see Verify the test results.

3.5 Test UpdateCapsule

UpdateCapsule is the standard interface for updating firmware. Use the `CapsuleApp.efi` application included in the ACS image to test that the `UpdateCapsule()` interface is working correctly.

To test `UpdateCapsule()`, do the following:

1. Copy the platform's three capsule files which were generated in Generate capsule files into the `BOOT` partition of the ACS image on a storage drive such as USB.
2. Boot the ACS image on the platform with the `new.dtb` file which was generated in Generate capsule files.
3. Select `bbr/bsa` from the GRUB boot menu.
4. Press Escape to stop running tests and open the UEFI shell. While the platform is booting, note the firmware version number reported on the console. After a successful firmware update with `CapsuleApp.efi`, the firmware should report a different version.
5. Use `CapsuleApp.efi` to attempt an update of the firmware with an unauthenticated capsule and with a tampered capsule. Both attempts should fail and the system should not reboot:

```
FS0:\EFI\BOOT\app\> CapsuleApp.efi -D unauth.bin
FS0:\EFI\BOOT\app\> CapsuleApp.efi unauth.bin
FS0:\EFI\BOOT\app\> CapsuleApp.efi -D tampered.bin
FS0:\EFI\BOOT\app\> CapsuleApp.efi tampered.bin
```

6. Use `CapsuleApp.efi` to authenticate and install the new signed firmware version and reboot the platform. If successful, the firmware reports the version of the firmware included in the capsule as follows:

```
FS0:\EFI\BOOT\app\> CapsuleApp.efi -D signed_capsule.bin
FS0:\EFI\BOOT\app\> CapsuleApp.efi signed_capsule.bin
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)
CapsuleApp: creating capsule descriptors at 0x9E273040
CapsuleApp: capsule data starts          at 0x9E13F040 with size 0xDE719
CapsuleApp: capsule block/size              0x9E13F040/0xDE719
Processing update 'u-boot.bin' :sha1+
dfu_alt_info set
using id 'nor0,0'
#######
resetting ...
```

7. Save the console log output from the previous commands as `fw/capsule-update.log` in the reporting directory structure.

To test delivery of capsules using a file on a mass storage device ("on disk"):

1. Re-flash the platform firmware and perform any necessary setup steps again to bring the system back to its original state. The firmware version reported should be the original one. Restart the console log output capture if necessary.
2. Use `CapsuleApp.efi` to update the firmware "on disk" using the `-OD` option. This copies the capsule file to the EFI System Partition (ESP) under the `EFI/UpdateCapsule` folder and the platform reboots. The firmware update is applied during system reboot, then there may be an additional system reboot, and ultimately the reported firmware version changes:

```
FS0:\EFI\BOOT\app\> CapsuleApp.efi -D signed_capsule.bin
FS0:\EFI\BOOT\app\> CapsuleApp.efi signed_capsule.bin -OD
```

3. Save the console log output from the previous commands as `fw/capsule-on-disk.log` in the reporting directory structure.

Always refer to the `README.md` from the SystemReady IR template for the latest manual test instructions.

3.6 Run the BBR tests

For SystemReady IR certification, 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-IR 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-IR 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 service set wakeup time interface, inva..  1 skipped
   Test UEFI RT time services unsupported status.          4 passed
```

3.7 Run Linux BSA

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

To run the Linux BSA test, do the following:

1. In the ACS-IR 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 1.0.1
Starting tests (Print level is  3)
Gathering system information....
  [   96.275278]  PE_INFO: Number of PE detected       :    2
  [   96.275700]  PCIE_INFO: Number of ECAM regions    :    1
  [   96.275984]  PCIE_INFO: No entries in BDF Table
  [   96.276340]  Peripheral: Num of USB controllers   :    0
  [   96.276679]  Peripheral: Num of SATA controllers  :    0
  [   96.277006]  Peripheral: Num of UART controllers  :    0
  [   96.277306]  DMA_INFO: Number of DMA CTRL in PCIe :    0
  [   96.282086]  SMMU_INFO: Number of SMMU CTRL       :    0
     *** Starting Memory Map tests ***
  [   96.286765]
  [   96.286765] Operating System View:
  [   96.287123]  104 : Addressability                              : Result:
 PASS
  [   96.287482]
  [   96.287482]        All Memory tests have passed!!
     *** Starting Peripherals tests ***
  [   96.294185]
  [   96.294185] Operating System View:
```

```
 [   96.294467]  605 : Memory Attribute of DMA
 [   96.294467]        No DMA controllers detected...
 [   96.294467]        Checkpoint --  3                           : Result:
SKIPPED
 [   96.295207]
 [   96.295207]       *** One or more tests have Failed/Skipped.***
    *** Starting PCIe tests ***
 [   96.297455]
 [   96.297455] Operating System View:
 [   96.301318]  801 : Check ECAM Presence                        : Result:
PASS
 [   96.301673]
 [   96.301673]        No PCIe Devices Found, Skipping PCIe tests...
 [   96.305981]
 [   96.305981]
------------------------------------------------------------
 [   96.305981]       Total Tests Run =  3, Tests Passed =  2, Tests Failed =  0
 [   96.305981]
------------------------------------------------------------
                  *** BSA tests complete ***
```

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

3.8 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 Certification Program.

For more details about the secure boot test, see the SystemReady Security Interface Extension User Guide.

3.9 Test installation of Linux distributions

SystemReady IR must boot at least two unmodified generic UEFI distribution images from an ISO image written to a storage medium.

The following Linux distributions produce suitable ISO images:

- Fedora IoT
- OpenSUSE Leap
- Debian Stable
- Ubuntu Server

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, replacing `<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 a log of the serial console output from the first power-on of the board.
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 will boot 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. Use the intermediate copy step to capture the `/sys/firmware` folder contents, then copy the resulting `console.log` file and the `sys-firmware.tar.gz` file into `os-logs/linux-<distroname>-<distroversion>/` in the results directory for reporting.

Always refer to the `README.md` from the SystemReady IR template for the latest manual test instructions.

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

To verify the test results:

1. Clone the latest version of the script repository:

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

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

```
$ 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, refer to 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 IR Certification, please 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 test suites is known as the BSA, BBR, and SIE ACS. These test suites test compliance against the BSA, BBR, EBBR, and SIE specifications for SystemReady IR certification. Arm recommends using architectural implementations to sign off against the ACS to prove compliance with these specifications.

For the latest image, please refer to SystemReady IR ACS Release details.

4.1 ACS overview

The ACS for SystemReady IR certification is delivered through a live OS image, which provides a GRUB menu containing the following options:

- `Linux Boot`
- `bbr/bsa`
- `SCT for Security Interface Extension (optional)`
- `Linux Boot for Security Interface Extension (optional)`

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 certification, 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:

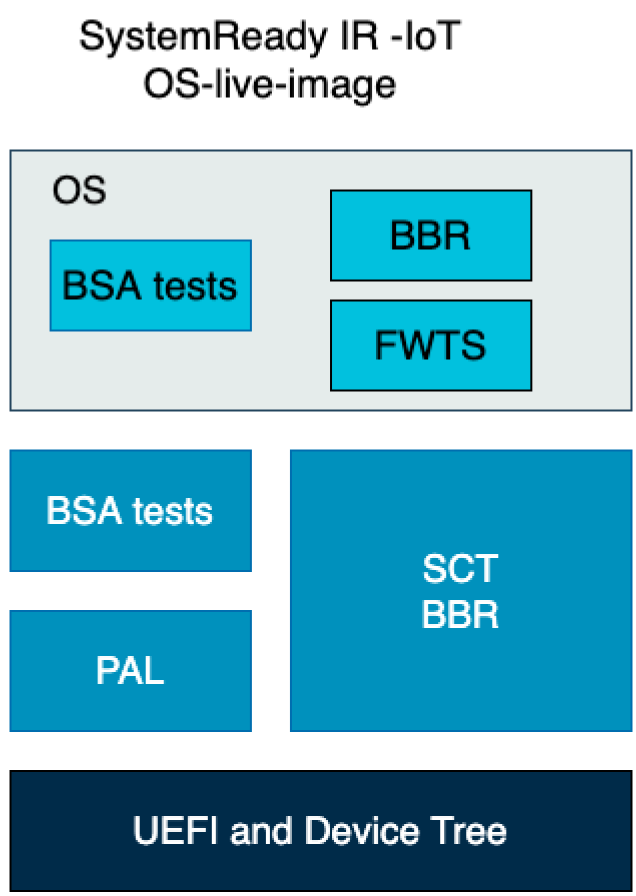
- EBBR tests contained in UEFI Self-Certification Tests (SCT). UEFI implementation requirements are tested by SCT.

- 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 SIE test suites check for compliance with the Base Boot Security Requirements (BBSR) specification. These test suites are automatically executed when the `SCT for Security Interface Extension (optional)` or `Linux Boot for Security Interface Extension (optional)` GRUB menu options are chosen. For more details about how to run the SIE tests, see the Security Interface Extension ACS Users Guide.

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

Figure 4-1: ACS components



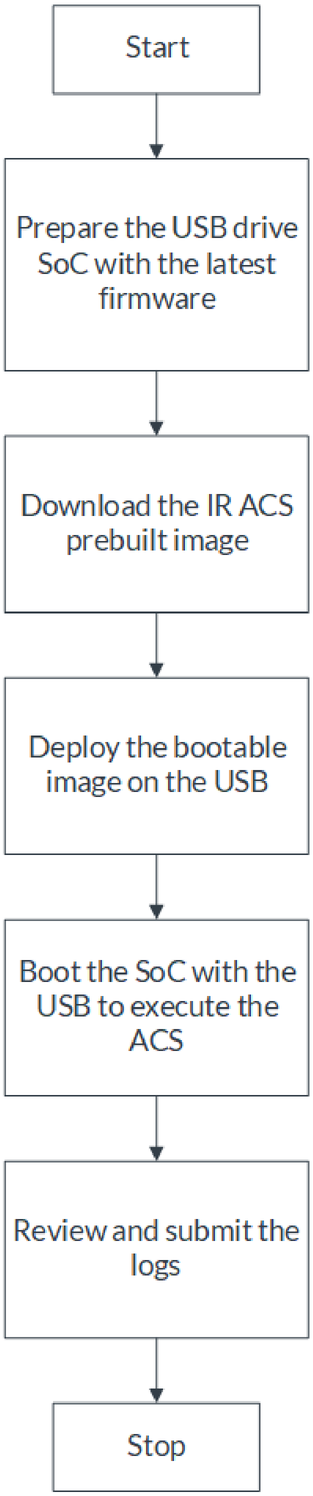
4.2 Run the ACS tests

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.

The following flow chart 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 prebuilt images repository to a local directory on Linux. For more information about the image releases, see the SystemReady IR ACS readme.
2. Deploy the ACS image on a USB device. Write the IR ACS bootable image to a USB storage device on the Linux host machine using the following commands:

```
$ lsblk
$ sudo dd if=/path/to/ir-acs-live-image-generic-arm64.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 ACS IR prebuilt image:

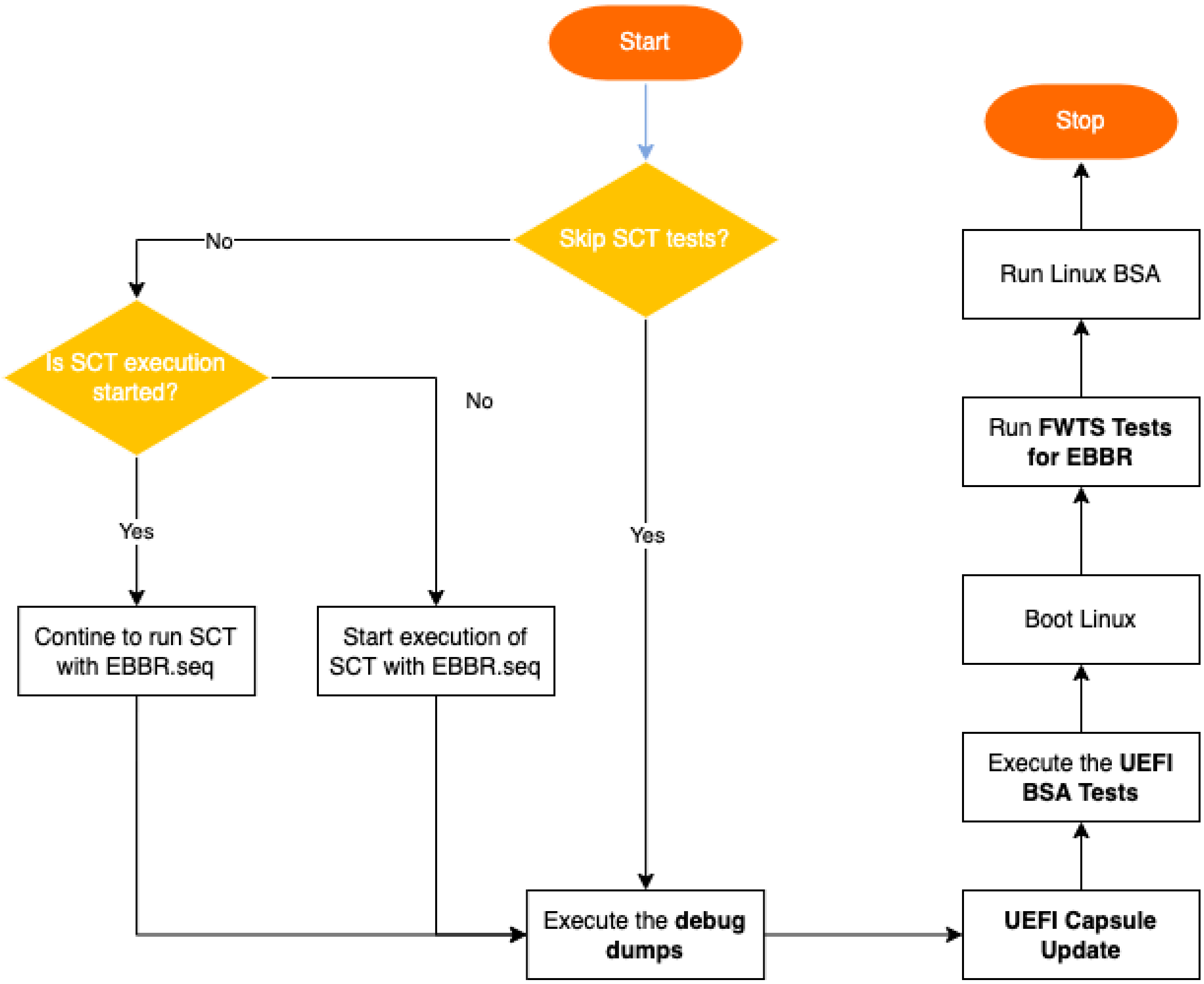
1. Start capturing a log of the serial console output. The log must start from the first power on of the board, and include the finished boot into Linux to run FWTS.
2. Select the option to boot from USB on the SoC.
3. Press any key to stop the boot process and change the `boot_targets` variable to specify 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 2022.10-rc2-g709b78e82b (Oct 26 2022 - 07:57:24 +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.

The following flowchart shows the complete ACS execution process through the IR 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.

The UpdateCapsule tests must be tested manually, then the logs must be recorded and submitted.

4.3 Run ACS in automated mode

If no option in the GRUB menu is chosen and no tests are skipped, the image runs the ACS in the following order:

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

4. Linux boot
5. FWTS tests
6. BSA tests

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:

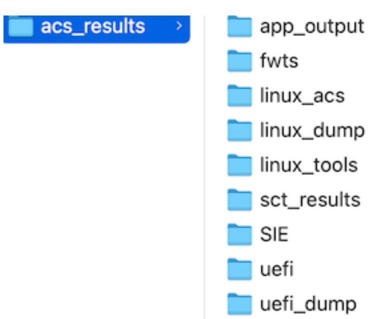
- `Linux Boot` to execute FWTS and BSA
- `bbr/bsa` to execute the tests in the same sequence as fully automated mode
- `SCT for Security Interface Extension` to execute the BBSR tests, include authenticated variable tests, secure boot, and TCG2 protocol tests
- `Linux Boot for Security Interface Extension` to execute authenticated variable tests and Devicetree base tests of FWTS and the Trusted Platform Module 2 test

4.5 Review the ACS logs

The logs are stored in a separate partition in the image called `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 shown in the following screenshot:

Figure 4-4: acs_results file location



The logs can also be extracted from the USB key on the host machine.

Check for the generation of the following logs after mounting the `acs_results` directory as shown in the table:

Number	ACS	Full log path	Running time	Description
1	BSA (UEFI)	`acs_results/uefi/BsaResults.log` `acs_results/uefi/BsaDevTree.dtb`	Less than two minutes	`BsaDevTree.dtb` is the dumped block of Devicetree
2	SCT	`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	FWTS	`acs_results/fwts/FWTSResults.log`	Less than two minutes	FWTSResults.log contains a summary table and output of the Firmware Test Suite results.
4	Debug Dumps	`acs_results/linux_dumps` `acs_results/uefi_dumps`	Less than two minutes	Contains dumps of the `lspci` command, `drivers`, `devices`, `memmap`, and other files
5	ESRT	`acs_results/app_output`	Less than two minutes	Contains the logs of ESRT and FMP tests
6	SCT for SIE	`acs_results/SIE/sct_results/Overall/Summary.log`	Less than four minutes	`Summary.log` contains the summary of all tests run

4.6 ACS logs

If any logs are missing, run the suite manually and report the error to your Arm Certification Partner. To report the error, mount the `acs_results` partition to copy the logs to a local directory, then submit the logs in the `acs_results` partition. Use the systemready-ir-template directory structure for recording the logs.

Use an SSD in a USB enclosure to execute the SCT tests more quickly.

Note

Run the SCT Parser tool to parse the logs further, based on YAML configurations.

To run the SCT 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
```

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

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

5. Related information

Here are some resources that are related to the material in this guide:

- Arm Base Boot Requirements
- Arm Base System Architecture (BSA) specification
- Arm Community
- Arm SystemReady Certification Program
- Arm SystemReady GitHub repository
- Arm SystemReady Requirements Specification
- Embedded Base Boot (EBBR) Requirements
- Base Boot Security Requirements (BBSR)
- Introduction to SystemReady
- SystemReady IR
- U-Boot git repository

6. Next steps

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

After reading this guide, you can find more information about certification registration at Arm SystemReady Certification 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 of the guide 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 ACS-IR image on QEMU

Running the ACS-IR image on QEMU involves the following operations:

- Prepare the ACS-IR live image
- Compile the U-Boot firmware and QEMU
- Execute the ACS on QEMU

B.1 Prerequisite

To test SystemReady IR for QEMU, use a PC running Ubuntu 22.04 LTS and install the following packages:

```
$ sudo apt install bc build-essential cpio file git rsync unzip wget xz-utils
```

B.2 Prepare the ACS-IR live image

Download the prebuilt IR ACS image from arm-systemready github.

Arm recommends that you select the latest prebuilt IR ACS image to download.

Uncompress the image with the following command:

```
$ xz -d ir-acs-live-image-generic-arm64.wic.xz
```

This image comprises three file system partitions recognized by UEFI:

results
: Stores the logs of the automated execution of ACS. The `acs_results` directory is stored in this partition. Approximate size: 50 MB.

/
: Stores rootfs.

boot
: Contains bootable applications and test suites. The `bbr` and `bsa` test applications are stored in this partition under the `EFI/BOOT` directory. Approximate size: 100 MB.

B.3 Compile the U-Boot firmware and QEMU

The U-Boot firmware and QEMU can be built with Buildroot.

To download and build the firmware code, do the following:

```
$ git clone https://git.buildroot.net/buildroot -b 2022.11.x
$ cd buildroot
$ 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`.

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

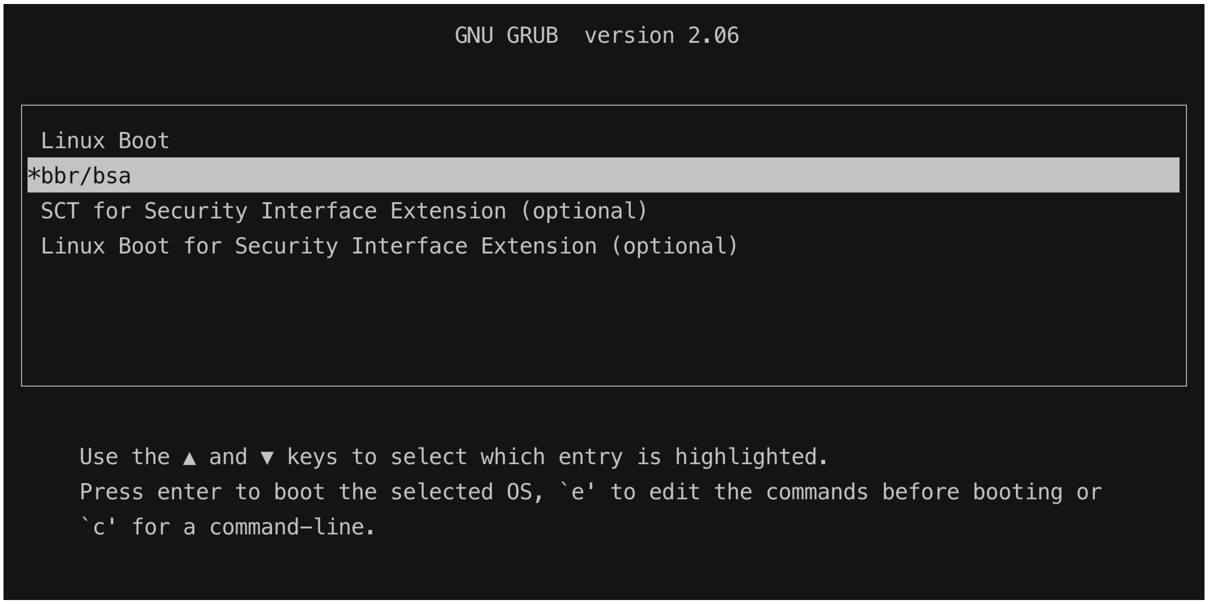
B.4 Execute the ACS on QEMU

Launch the model using the following command:

```
$ ./output/host/bin/qemu-system-aarch64 \
    -bios output/images/flash.bin \
    -cpu cortex-a53 \
    -d unimp \
    -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 1024 \
    -machine virt,secure=on \
    -monitor null \
    -netdev user,id=eth0 \
    -no-acpi \
    -nodefaults \
    -nographic \
    -object rng-random,filename=/dev/urandom,id=rng0 \
    -rtc base=utc,clock=host \
    -serial stdio \
    -smp 2
```

The ACS-IR starts, as shown in the following screenshot:

Figure B-1: ACS-IR 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.5 Troubleshooting advice

If the ACS halts at the following BSA test:

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

Restart `qemu-system-aarch64` to finish running the ACS.

Appendix C Rebuild the ACS-IR image

SystemReady ACS GitHub contains the prebuilt image. For details of the latest version for download you can refer to the release details. For debug purposes, if you want to rebuild the ACS-IR image, you can refer to these 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`. Additionally, 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 IR 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 `v23.03_IR_2.0.0`:

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

2. Navigate to the `IR/Yocto` directory:

```
cd /path-to/arm-systemready/IR/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 IR ACS live image, use the following command:

```
./build-scripts/build-ir-live-image.sh
```

If this procedure is successful, the bootable image will be available at `/path-to-arm-systemready/IR/Yocto/meta-woden/build/tmp/deploy/images/generic-arm64/ir-acs-live-image-generic-arm64.wic.xz`.

Note

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

```
xz -d ir-acs-live-image-generic-arm64.wic.xz
```

C.3 Troubleshooting advice

When building the IR ACS live image in step 4, you may encounter 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 encounter 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 before submitting your results for SystemReady IR certification:

1. Perform U-Boot sanity tests manually as described in the Test the U-Boot Shell section in Test SystemReady IR.
2. Perform UEFI sanity tests manually as described in the Test the UEFI Shell section in Test SystemReady IR.
3. Perform capsule update manually as described in the Test UpdateCapsule section in Test SystemReady IR.
4. Run the automated ACS-IR as described in the Run the ACS test suite section in Test SystemReady IR.
5. Optionally perform SCT test for Security Interface Extension as described in SystemReady Security Interface Extension User Guide.
6. Optionally perform Linux SIE 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 IR.
8. Verify your test results using the scripts as described in the Verify the test results section in the Test SystemReady IR and the Review the ACS logs section in Test with the ACS.

Appendix E Frequently Asked Questions

This section answers some common questions related to SystemReady IR.

E.1 General

This section answers general questions related to SystemReady IR.

What operating systems can run on a SystemReady IR platform?

While SystemReady IR is intended to make it easier to build embedded Linux and BSD systems, it defines a base platform architecture that can be used by any operating system. Operating systems that use the UEFI firmware ABI and the Devicetree system description can boot on a SystemReady IR platform.

How does SystemReady IR differ from SystemReady ES and SystemReady SR?

SystemReady IR differs from SystemReady ES and SystemReady SR in the following ways:

- SystemReady IR requires only a subset of the UEFI ABI required by SystemReady ES and SystemReady SR. In particular, SystemReady IR does not require most Runtime Services after `ExitBootServices()` has been called, and SystemReady IR does not require Option ROM loadable driver support. The lack of Runtime Services means changes to firmware variables, like `BootXXXX`, must be done in the UEFI environment before the OS boots. The lack of Option ROM support means that booting from PCIe devices may not be supported if the firmware does not have native drivers for the device.
- SystemReady IR uses the Devicetree system description instead of ACPI and SMBIOS. Devicetree is used by Embedded Linux products and many embedded SoCs do not currently have working ACPI descriptions. Linux supports both ACPI and Devicetree system descriptions, so SystemReady IR, SystemReady ES, and SystemReady SR platforms can all be supported with a single kernel image if the appropriate configuration options are enabled.
- SystemReady IR has a different user experience to SystemReady ES and SystemReady SR. SystemReady ES and SystemReady SR provide forward and backward compatibility with generic off-the-shelf OS images. SystemReady IR only supports limited compatibility and has the dependency that the board support package (BSP) is upstreamed and downported to the distros.

Can I certify using a custom kernel?

No. Certification requires evidence that mainline Linux or BSD works on the platform. Unmodified third-party distributions are the best way to provide that evidence. Using a custom kernel can hide firmware or hardware problems that prevent mainline from running on the hardware.

E.2 Certification testing

This section answers questions related to certification testing in SystemReady IR.

I get X errors from the ACS SCT results. How many errors are acceptable?

If you are using the latest copy of the SCT_Parser script you should not see any errors. If you have errors, it is likely a problem with your firmware configuration. The latest copy of the SCT parser script is available from Arm GitLab.

How do I fix Variable Services test failures?

Firmware must have the ability to set UEFI variables, and these values must persist over reboots. If variable values do not persist over reboot, you will see Variable Services test failures and the following failure:

```
   |BS.ExitBootServices - ConsistencyTestCheckpoint1
   |FAILURE
   |BootServicesTest|ImageServicesTest|ExitBootServices_Conf
   |303ABFAB-C865-4255-86E3-6EEF175E30DD
   |0
   |19-08-2021|08:15:00
   |0x00010001
   |Image Services Test
   |No device path
   |A5BB81FA-1063-4358-97AF-AD57D42BF055
   |/home/charles/work/acs_images/arm-systemready/IR/scripts/edk2-test/
uefi-sct/SctPkg/TestCase/UEFI/EFI/BootServices/ImageServices/BlackBoxTest/
ImageBBTestConformance.c 917  GetVariable service routine failed - Not Found
   |Check logs for messages such as "No EFI system partition" or "Failed to persist
 EFI variables" and check that system has an EFI System Partition
   |Add comments to failure due to missing ESP|
```

By default, U-Boot stores UEFI variables as a file in the EFI System Partition (ESP). The most common cause of this failure is not having an ESP on the primary storage device, like eMMC or SD. To fix the problem, make sure the eMMC or SD has a GPT partition table and create a small FAT formatted 100MB partition with type `0xEF00`. U-Boot uses this partition to store variables, and the failure stops.

How do I work around Debian's Failed to install GRUB error?

Debian currently requires UEFI `SetVariable()` to work after `ExitBootServices()` while the operating system is running, but SystemReady IR does not require `SetVariable()` to be supported after `ExitBootServices()`. Fedora IoT and OpenSUSE both have workaround code to handle installing GRUB in a failsafe way, but Debian does not.

The workaround for Debian is to finalize the GRUB install manually before exiting the installer. After the Debian installer displays the No Bootloader Installed error message, select **Execute a shell** and enter the following commands:

```
 ~ # in-target grub-install --no-nvram --force-extra-removable
 ~ # in-target update-grub
```

Exit the `chroot` and the shell to return to the installer and select **Continue without boot loader** to finish installation.