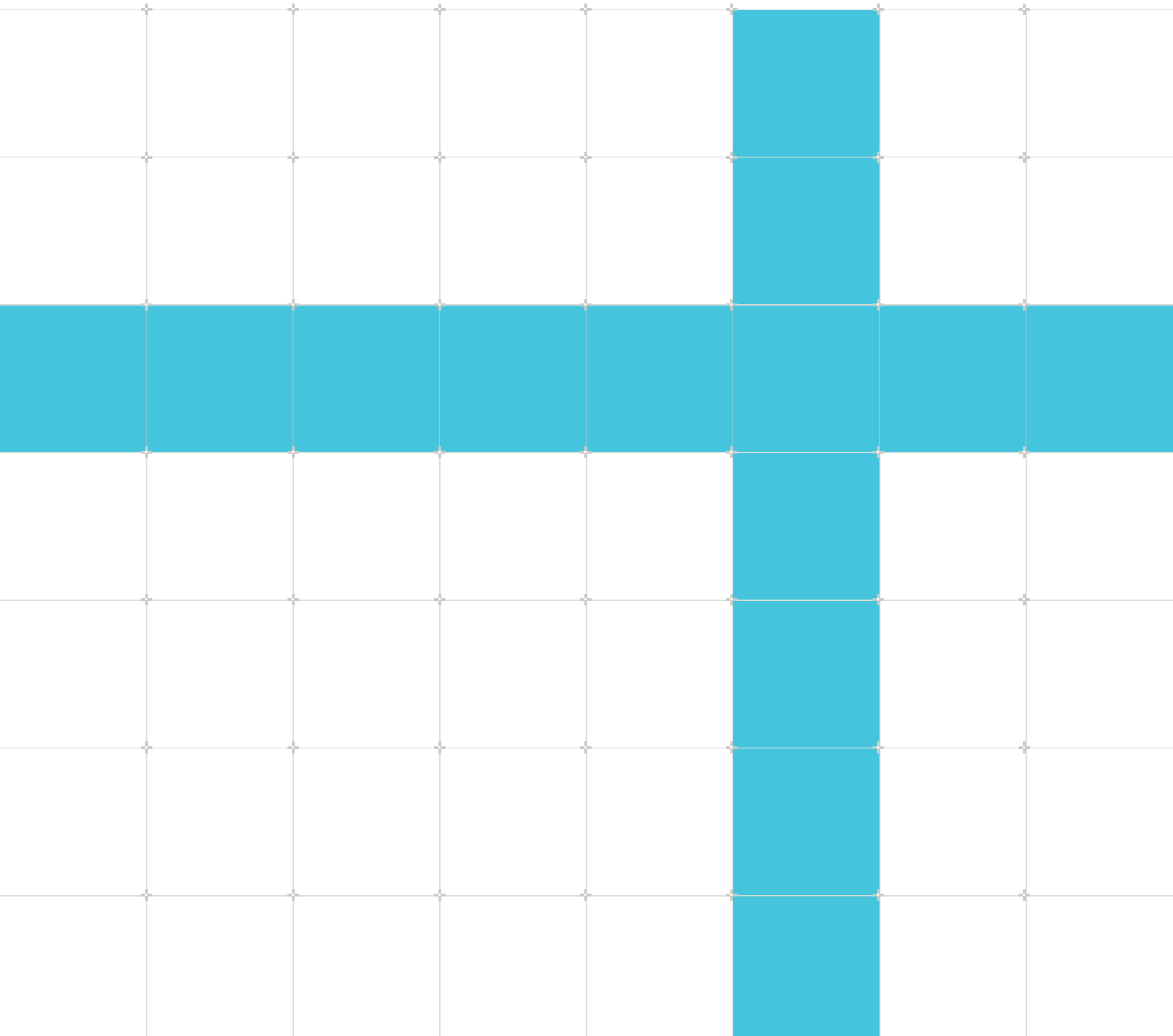
arm

SystemReady IR IoT Integration, Test, and Certification Guide

Version 1.1

Issue 02
DUI1101_1.1_02_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
0101-02	28 June 2022	Non-Confidential	Minor modifications to explicitly mention IR 1.1 certification, refer to the ACS-IR pre-built image v21.09_1.0, and refer to the ir1 branches in the components.

Proprietary Notice

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

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

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

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349)

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. Introduction

1.1 Conventions

The following subsections describe conventions used in Arm documents.

Glossary

The Arm Glossary is a list of terms used in Arm documentation, together with definitions for those terms. The Arm Glossary does not contain terms that are industry standard unless the Arm meaning differs from the generally accepted meaning.

See the Arm® Glossary for more information: developer.arm.com/glossary.

Typographic conventions

Arm documentation uses typographical conventions to convey specific meaning.

Convention	Use
italic	Citations.
bold	Interface elements, such as menu names. Signal names. Terms in descriptive lists, where appropriate.
`monospace`	Text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace bold`	Language keywords when used outside example code.
`monospace underline`	A permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`<and>`	Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example: `MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>`
SMALL CAPITALS	Terms that have specific technical meanings as defined in the *Arm® Glossary*. For example, **IMPLEMENTATION DEFINED**, **IMPLEMENTATION SPECIFIC**, **UNKNOWN**, and **UNPREDICTABLE**.
Caution	Recommendations. Not following these recommendations might lead to system failure or damage.
Warning	Requirements for the system. Not following these requirements might result in system failure or damage.
Danger	Requirements for the system. Not following these requirements will result in system failure or damage.

Convention	Use
 Note	An important piece of information that needs your attention.
 Tip	A useful tip that might make it easier, better or faster to perform a task.
 Remember	A reminder of something important that relates to the information you are reading.

1.2 Other information

See the Arm website for other relevant information.

- Arm® Developer.
- Arm® Documentation.
- Technical Support.
- Arm® Glossary.

2. 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) and Base Boot Requirements (BBR) specifications, and market-specific supplements.

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

Figure 2-1: SystemReady IR Logo

The SystemReady certification ensures Arm-based servers, infrastructure edge, 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.

There are four bands of SystemReady:

- SR – Workstations and Enterprise Servers
- ES – General Purpose Embedded Servers for remote edge nodes
- IR – Embedded devices for IoT and Edge
- LS – Hyperscale Servers deployed with LinuxBoot

These bands are based on combinations or recipes from the BSA, supplements such as the Server Base System Architecture (SBSA), and the BBR specifications.

2.1 Before you begin

This guide tells you how to configure a U-Boot based platform for SystemReady IR 1.1 compliance, and how to run all the SystemReady IR tests before submitting the platform for certification in the Arm SystemReady Certification Program.

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

- Enable Unified Extensible Firmware Interface (UEFI) features in U-Boot

- Run the Arm Architecture Compliance Suite (ACS) and analyze test results
- Test the `UpdateCapsule()` interface to update firmware
- Boot and install generic Linux distribution images

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

This guide assumes U-Boot firmware and the examples shown are captured on a U-Boot platform. However, SystemReady IR 1.1 compliance can be achieved with any UEFI compliant firmware and U-Boot is not required. You can also use EDK2 or another firmware implementation for certification. If you are not using U-Boot, you can skip the Configure U-Boot for SystemReady section.

SystemReady certified platforms must use a specific set of hardware and firmware features to enable an operating system to deploy the operating system image. Compliant systems must conform to the following requirements:

- 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.
- EBBR recipe of the Arm Base Boot Requirements
- We recommended that SystemReady IR platforms comply with the Arm Base System Architecture (BSA) specification. SystemReady IR v1.1 certification does not require BSA compliance, but for certification the BSA compliance tests must still be run and the results submitted. BSA compliance will become a requirement in a future version of SystemReady IR.

Before you get started with this guide, build U-Boot and install it on your platform. U-Boot 2021.04 or later is required for SystemReady IR certification. U-Boot releases and patches can be found on the U-Boot git repository. Instructions for porting and building U-Boot is beyond the scope of this document. Please refer to the U-Boot documentation for details on how to enable a new platform.

3. Configure U-Boot for SystemReady

This section tells you how to enable the following configuration options required for SystemReady IR certification requirements:

- UEFI support with U-Boot
- Device Firmware Update to enable `UpdateCapsule()` support

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

Enable UEFI support with U-Boot

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. Enable the configuration options in `<root_workspace>/u-boot/configs/<platform_name>_defconfig` 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. Add the configuration options to enable Real Time Clock (RTC) support as shown:

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

3. 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`.
4. Add the configuration options to enable the UEFI `UpdateCapsule()` interface to update firmware as follows:

```
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
CONFIG_EFI_CAPSULE_FMP_HEADER=y
```

5. 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 UEFI ABI, U-Boot finds and executes UEFI binaries from a system partition on an eMMC, SD card, USB flash drive, or other devices. UEFI boot can be tested either with a Linux distribution ISO image or the ACS. Boot the platform with the image on a USB flash drive to boot either Grub Linux distribution or the EFI Shell.

Configure Device Firmware Upgrade

In U-Boot, configure Device Firmware Upgrade (DFU) to enable UpdateCapsule support, if it is supported for your system.

To enable DFU mode:

1. Add the following configuration options as shown:

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

2. Add one or more of the DFU backend configuration options for the storage device containing the firmware as show in the following code:

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

3. Enable configuration options to ensure one or more of the DFU transport options are enabled for testing as shown:

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

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

```
/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 `test.itb` test image using `mkimage` using the code shown:

```
$ 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 test.itb
u-boot=> dhcp
u-boot=> dfu tftp ${kernel_addr_r}
```

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

```
$ mkeficapsule -fit tests.itb --index 1 capsule.bin
```

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

Alternatively, the `GenerateCapsule` tool from EDK2 could be used to create a UEFI Capsule binary.

Adapt the automated boot flow

Make sure the UEFI boot methods will be tried during the automated boot sequence. In U-Boot, the `bootcmd` environment variable holds the default boot command. This is usually a script that tries 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` are empty when booting with UEFI. The `bootargs` U-Boot environment variables 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`. 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` must be empty when booting with UEFI. If your system uses the generic distro configuration, the `bootargs` are handled appropriately.

Adapt the Devicetree

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

Also, ensure 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 snippet shows a simplified Devicetree example:

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

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

4. Test SystemReady IR

This section teaches you how to run the U-Boot and UEFI tests, and how to test the Linux installation for SystemReady IR certification. The following steps are also listed in the Test checklist appendix to help during testing.

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

- Provided by your platform vendor:
  - Platform under test with firmware already installed
  - Firmware capsule image in Firmware Management Protocol (FMP) format
- 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 if all required logs are provided and if the required tests have passed
  - Arm SystemReady IR ACS v21.09_1.0 image installed on a USB drive
- Provided by a third party: two generic Linux ISO distribution images on USB drives

To test SystemReady IR, you must run a test on the U-Boot console, the UEFI environment, the `UpdateCapsule()` interface, and install two Linux distributions.

Before running the tests, clone the SystemReady reporting template repo as shown and use it to capture the test results and logs:

```
$ git clone https://git.gitlab.arm.com/systemready/systemready-template.git -b ir1
```

Refer to the documentation in the template repository for the latest list of check commands.

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 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.

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, and run the following commands:

```
FS0:\Sct\> cd ..
FS0:\> pci
FS0:\> drivers
FS0:\> devices
FS0:\> devtree
FS0:\> dmpstore
FS0:\> dh -d -v
FS0:\> memmap
FS0:\> smbiosview
```

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

Test UpdateCapsule

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

To test `UpdateCapsule()`:

1. Copy the platform's capsule file into the `BOOT` partition of the ACS image on a USB drive.
2. Boot the ACS image on the platform, then select bsa/bbr tests from the Grub boot menu.
3. 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 running `CapsuleApp.efi`, the firmware reports a different version.
4. From the UEFI shell, use the following commands to test the `UpdateCapsule` interface. FS# references are different on each system. In this example, change `FS1:` to the `FS#:` for the ACS USB on your platform. With the current version U-Boot, several ASSERT errors are displayed on the console. These errors are due to U-Boot not implementing a protocol that is expected by `CapsuleApp.efi` and can be ignored:

```
 UEFI Interactive Shell v2.2
 EDK II
 UEFI v2.80 (Das U-Boot, 0x20210700)
 Mapping table
       FS0: Alias(s):HD0c:;BLK2:
           /VenHw(e61d73b9-a384-4acc-aeab-82e828f3628b)/eMMC(2)/eMMC(1)/
HD(2,GPT,09d411d0-9fce-43cc-bc40-b1104cd510d8,0x4800,0x80000)
       FS1: Alias(s):HD0b:;BLK5:
           /VenHw(e61d73b9-a384-4acc-aeab-82e828f3628b)/
UsbClass(0x0,0x0,0x9,0x0,0x1)/UsbClass(0x781,0x5581,0x0,0x0,0x0)/
HD(1,GPT,c2825b51-        9880-4738-ae28-a0792f1e4e3d,0x800,0xfffff)
```

```
       FS2: Alias(s):HD0c:;BLK6:
            /VenHw(e61d73b9-a384-4acc-aeab-82e828f3628b)/
UsbClass(0x0,0x0,0x9,0x0,0x1)/UsbClass(0x781,0x5581,0x0,0x0,0x0)/
HD(2,GPT,c2825b51-9880-4738-ae28-a0792f1e4e3d,0x100800,0x18fff)
 Press ESC in 4 seconds to skip startup.nsh or any other key to continue.
 Shell> fs1:
 FS1:\> ls
 Directory of: FS1:\
 00/00/0000  00:00           3,583,068  capsule.bin
 00/00/0000  00:00 <DIR>             0  EFI
 00/00/0000  00:00 <DIR>             0  grub
 00/00/0000  00:00                 298  grub.cfg
 00/00/0000  00:00          32,930,304  Image
 00/00/0000  00:00          92,389,888  ramdisk-busybox.img
   4 File(s)
   2 Dir(s)
 FS1:\> efi\boot\app\capsuleapp -E

 ASSERT_EFI_ERROR (Status = Not Found)
 ASSERT [CapsuleApp]
 /home/cherat01/ATEG/SystemReady/BBR/arm-systemready/IR/scripts/edk2/MdePkg/
Library/DxeServicesTableLib/DxeServicesTableLib.c(58):
 !EFI_ERROR (Status)
 ASSERT [CapsuleApp]
 /home/cherat01/ATEG/SystemReady/BBR/arm-systemready/IR/scripts/edk2/MdePkg/
Library/DxeServicesTableLib/DxeServicesTableLib.c(59):
 gDS != ((void *) 0)

 ASSERT_EFI_ERROR (Status = Not Found)
 ASSERT [CapsuleApp]
 /home/cherat01/ATEG/SystemReady/BBR/arm-systemready/IR/scripts/edk2/Build/
MdeModule/DEBUG_GCC5/AARCH64/MdeModulePkg/Application/CapsuleApp/CapsuleApp/
DEBUG/AutoGen.c(415):
 !EFI_ERROR (Status)

 ASSERT_EFI_ERROR (Status = Not Found)
 ASSERT [CapsuleApp]
 /home/cherat01/ATEG/SystemReady/BBR/arm-systemready/IR/scripts/edk2/MdePkg/
Library/DxeHobLib/HobLib.c(48):
 !EFI_ERROR (Status)
 ASSERT [CapsuleApp]
 /home/cherat01/ATEG/SystemReady/BBR/arm-systemready/IR/scripts/edk2/MdePkg/
Library/DxeHobLib/HobLib.c(49):
 mHobList != ((void *) 0)
 ###############
 # ESRT TABLE #
 ###############
 EFI_SYSTEM_RESOURCE_TABLE:
 FwResourceCount    - 0x0
 FwResourceCountMax - 0x0
 FwResourceVersion  - 0x1

 FS1:\> efi\boot\app\capsuleapp -P

 ASSERT_EFI_ERROR (Status = Not Found)
 ASSERT [CapsuleApp]
 /home/cherat01/ATEG/SystemReady/BBR/arm-systemready/IR/scripts/edk2/MdePkg/
Library/DxeServicesTableLib/DxeServicesTableLib.c(58):
 !EFI_ERROR (Status)
 ASSERT [CapsuleApp]
 /home/cherat01/ATEG/SystemReady/BBR/arm-systemready/IR/scripts/edk2/MdePkg/
Library/DxeServicesTableLib/DxeServicesTableLib.c(59):
 gDS != ((void *) 0)

 ASSERT_EFI_ERROR (Status = Not Found)
 ASSERT [CapsuleApp]
 /home/cherat01/ATEG/SystemReady/BBR/arm-systemready/IR/scripts/edk2/Build/
MdeModule/DEBUG_GCC5/AARCH64/MdeModulePkg/Application/CapsuleApp/CapsuleApp/
DEBUG/AutoGen.c(415):
 !EFI_ERROR (Status)
```

```
 ASSERT_EFI_ERROR (Status = Not Found)
 ASSERT [CapsuleApp]
 /home/cherat01/ATEG/SystemReady/BBR/arm-systemready/IR/scripts/edk2/MdePkg/
Library/DxeHobLib/HobLib.c(48):
 !EFI_ERROR (Status)
 ASSERT [CapsuleApp]
 /home/cherat01/ATEG/SystemReady/BBR/arm-systemready/IR/scripts/edk2/MdePkg/
Library/DxeHobLib/HobLib.c(49):
 mHobList != ((void *) 0)
 ############
 # FMP DATA #
 ############
 FMP protocol - Not Found
 FS1:\>
```

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

```
FS1:\> efi\boot\app\capsuleapp capsule.bin
```

6. Save the console log from the `UpdateCapsule()` procedure as `fw/capsule-update.log` in the reporting directory structure.

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 USB drive into the results directory.

Run Linux BSA

The Linux BSA test is not run automatically by the ACS-IR and must be run manually.

To run the Linux BSA test:

1. Boot the ACS-IR image and select Linux BusyBox.
2. Load the kernel module as follows:

```
 / # insmod /lib/modules/bsa_acs.ko
 [   78.227399] init BSA Driver
```

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

```
 / # /bin/bsa

    ************ BSA Architecture Compliance Suite *********
                 Version 1.0

 Starting tests (Print level is  3)

 Gathering system information....
 [  108.895524]  PE_INFO: Number of PE detected       :    1
 [  108.898411]  PCIE_INFO: Number of ECAM regions    :    1
 [  109.820411]  PCIE_INFO: No entries in BDF Table
 [  109.821970]  Peripheral: Num of USB controllers   :    0
 [  109.822530]  Peripheral: Num of SATA controllers  :    0
 [  109.823418]  Peripheral: Num of UART controllers  :    0
 [  109.826078]  DMA_INFO: Number of DMA CTRL in PCIe :    0
 [  109.827791]  SMMU_INFO: Number of SMMU CTRL       :    0
```

```
*** Starting Memory Map tests ***
[  109.835649]
[  109.835649] Operating System View:
[  109.836258]  104 : Addressability                              : Result:
PASS
[  109.838899]
[  109.838899]        All Memory tests have passed!!

*** Starting Peripherals tests ***
[  109.846249]
[  109.846249] Operating System View:
[  109.847315]  605 : Memory Attribute of DMA
[  109.847315]       No DMA controllers detected...
[  109.847315]       Checkpoint --  3                             : Result:
SKIPPED
[  109.852298]
[  109.852298]       *** One or more tests have Failed/Skipped.***

*** Starting PCIe tests ***
[  109.856345]
[  109.856345] Operating System View:
[  109.858919]  801 : Check ECAM Presence                         : Result:
PASS
[  109.861826]
[  109.861826]       *** No Valid Devices Found, Skipping PCIE tests ***
[  109.862750]
[  109.862750]      -------------------------------------------------------------
[  109.862750]       Total Tests Run =  3, Tests Passed =  2, Tests Failed =  0
[  109.862750]      -------------------------------------------------------------

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

4. Save the console log from the Linux BSA procedure as `manual-results/bsa-linux/console.log` in the reporting directory structure.

Test Linux Distributions installation

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

The following Linux distributions produce suitable ISO images:

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

To test the Linux distribution installation, write the ISO image to a USB drive. Use the following command from a bash shell to write the downloaded ISO to a USB drive. Replace `<usb-block-device>` with the path to the USB drive's block device on your Linux workstation:

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

When testing the distribution installation, capture a log of the serial console output from the first power on the board. To capture this log, use the installation USB attached to the final installed Linux distro after running the Linux sniff tests.

Once the ISO is written to the USB drive, connect the USB drive to your board and turn the drive on. U-Boot finds the image and boots from the image by default. A compliant system will boot from the distro ISO into the installer tool. Use the tool to complete the installation of Linux and then reboot into a working Linux environment installed on the eMMC or other local storage.

After Linux is installed, run the following sequence of Linux sniff tests as root using the serial console:

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

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.

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 scripts:

   ```
   $ git clone https://gitlab.arm.com/systemready/systemready-scripts -b ir1
   ```

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

   ```
   $ cd systemready-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 is no error reported, as shown in the example output.

For more information, refer to the documentation in the systemready-scripts and systemready-template repositories.

5. Test with the ACS

The ACS ensures architectural compliance across different implementations of the architecture. The ACS is delivered with tests in source form with a build environment. The build output is a bootable live OS image containing a collection of test suites. This collection of test suites is known as the BSA and BBR ACS. These test suites test compliance against BSA, BBR, and EBBR specifications for SystemReady IR certification. We recommend using architectural implementations to sign off against the ACS to prove compliance with these specifications.

ACS overview

The ACS for SystemReady IR certification is delivered through a live OS image, 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 FWTS is a package hosted by Canonical which provides tests for Device tree and UEFI. The FWTS test is customized to run only UEFI tests.

The BSA test suites are checked for compliance against the BSA specification. The tests are delivered through the following suites:

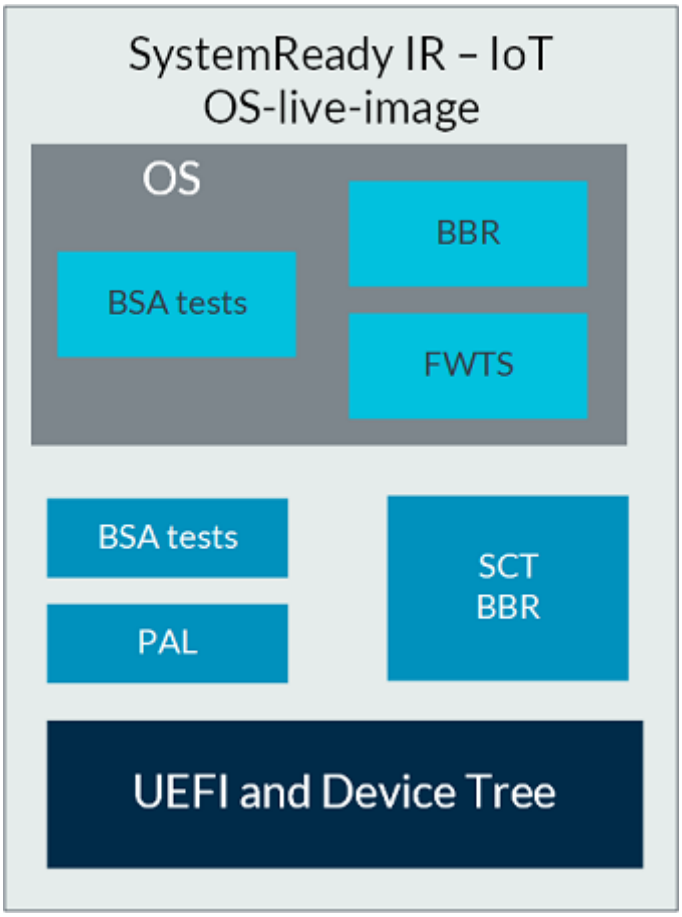
- BSA tests on UEFI Shell. These tests are written on top of Validation Adaption Layers (VAL) and Platform Adaptation Layers (PAL). The abstraction layers provide the tests with platform information and runtime environment to enable execution of the tests. In Arm deliveries, the VAL and PAL are written 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 are checked for compliance against the BBR specification. For certification, the firmware is tested against the EBBR recipe which contains a reduced set 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 customized to run only UEFI tests applicable to EBBR.

The contents of the live OS image are shown in the following diagram:

Figure 5-1: ACS components



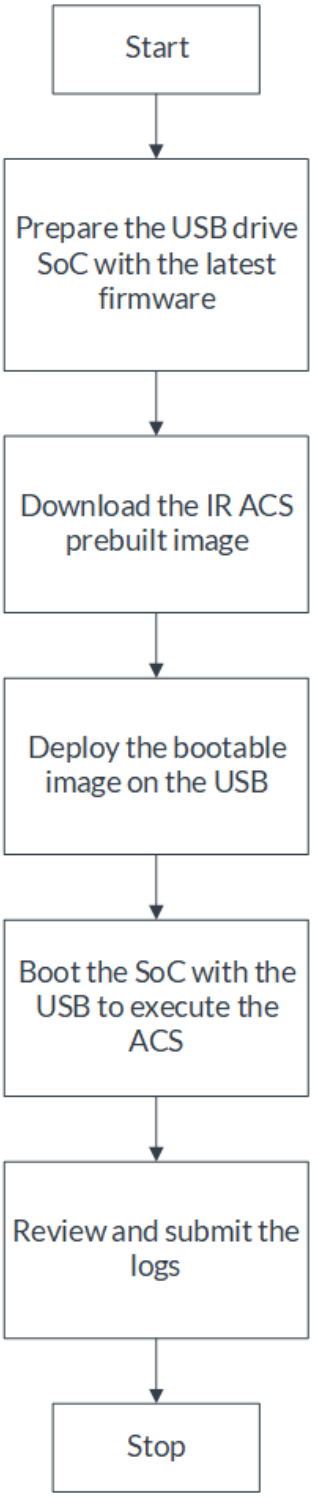
Run the ACS test

The prerequisites to run the ACS tests are as follows:

- Prepare a USB device with a minimum of 1GB of storage. This USB is used to boot and run the ACS and to store the execution results.
- Prepare the System Under Test (SUT) machine with the latest firmware loaded, a host machine for console access, then collect the results

The ACS test process is shown in the following flow chart:

Figure 5-2: ACS test process



The ACS image must be set up on an independent medium or disk, like a USB. 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 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 ACS v21.09_1.0 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 USB. Write the IR ACS bootable image to a USB stick on the Linux host machine using the following commands:

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

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

To execute the ACS IR prebuilt image:

1. Start capturing a log of the serial console output. The log must cover from the first power on of the board to 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 like an SD card. If the platform cannot boot, the following message is displayed:

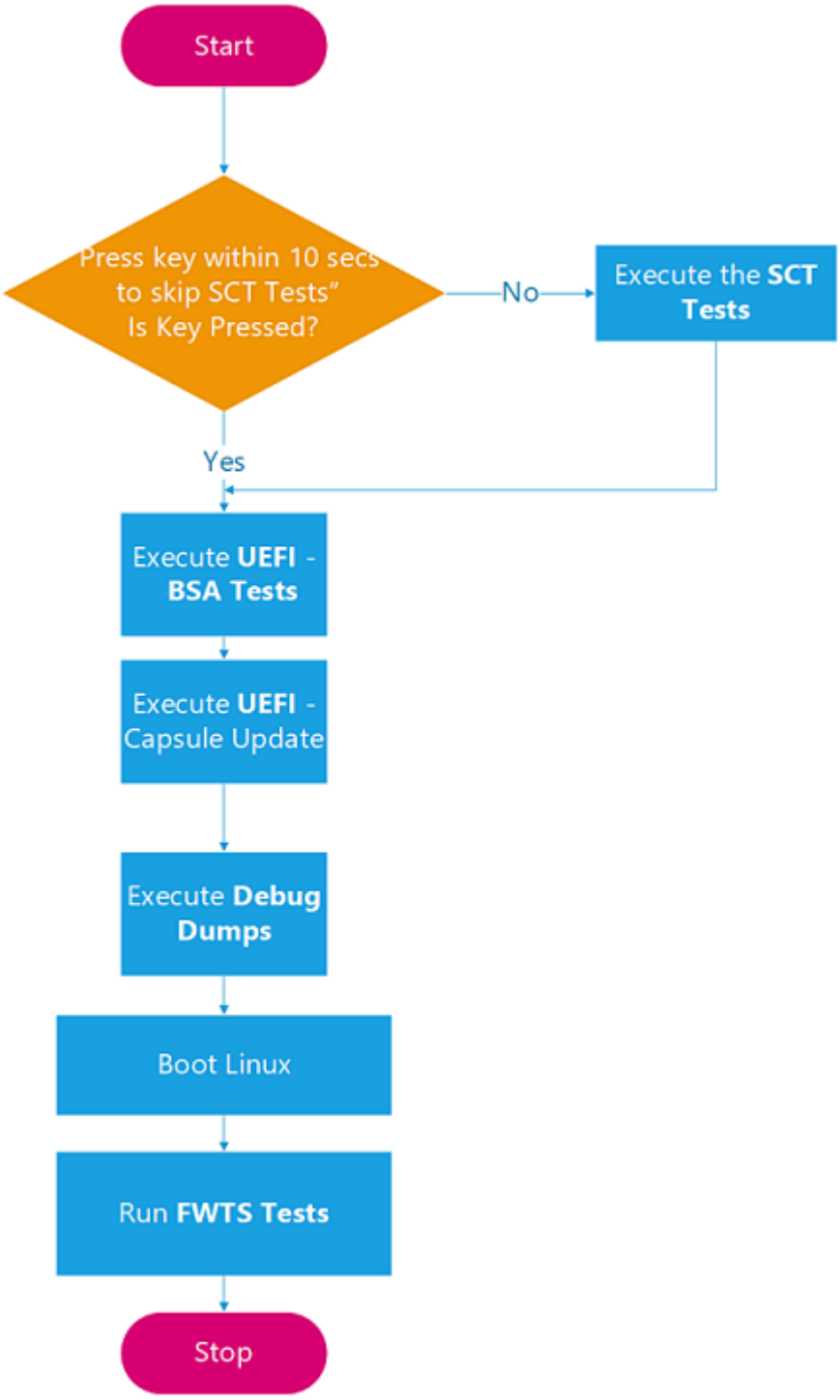
```
U-Boot 2021.01-ge4477e7954 (Apr 23 2021 - 14:54:15 +0100)

CPU:   [CPU Name] rev1.0 at 1200 MHz
Reset cause: POR
Model: [Board Name]
DRAM:  2 GiB
WDT:   Started with servicing (60s timeout)
2
Loading Environment from MMC... OK
In:    serial
Out:   serial
Err:   serial
Net:
Warning: ethernet@30be0000 (eth0) using random MAC address - ea:d7:72:f7:a2:30
eth0: ethernet@30be0000
Hit any key to stop autoboot:  0
u-boot=> print boot_targets
boot_targets=mmc2 mmc0 usb0 pxe dhcp
u-boot=> setenv boot_targets usb0 mmc2
u-boot=> saveenv
u-boot=> boot
starting USB...
Bus usb@32e40000: USB EHCI 1.00
Bus usb@32e50000: USB EHCI 1.00
[...]
```

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 complete ACS execution process through IR ACS live image is shown in the following flowchart:

Figure 5-3: 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.

The prebuilt ACS live-images are released with a specific version Linux kernel which are progressively upgraded with every release of the ACS. If the boot on your SoC demands the updates for a newer version of the kernel, then you may build the ACS image manually. The steps to build a customized ACS live-image with a newer kernel are detailed in [Appendix: Rebuild the ACS-IR image]. The test results obtained from manually built ACS live-image is for reference only. For the purpose of SystemReady IR certifications, these tests have to be rerun after the ACS image with the new kernel is released.

Run ACS in automated mode

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

1. SCT tests are run.
2. Debug dumps are executed.
3. BSA ACS is run.
4. Linux BusyBox Boot.
5. FWTS tests are run.

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

Run ACS in normal mode

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

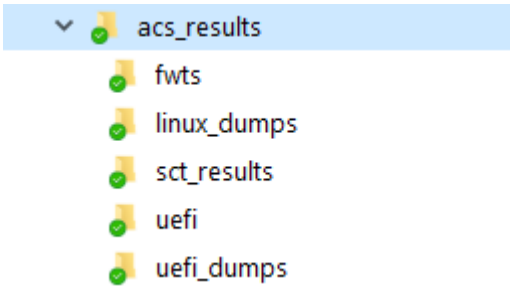
- Linux BusyBox to boot Linux and execute FWTS
- BBR or BSA to execute the tests in the same sequence as fully automated mode

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 5-4: 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`	Less than two minutes	
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/results.log`	Less than two minutes	
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

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. Clone the systemready-template directory structure as shown and use it for recording the logs:

```
$ git clone https://git.gitlab.arm.com/systemready/systemready-template.git -b ir1
```

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

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 -b ir1
```

2. Run the parser from the SCT results folder:

```
$ cd acs_results/sct_results
$ /path/to/edk2-test-parser/parser.py Overall/Summary.ekl Sequence/EBBR.seq
INFO ident_seq: Identified `Sequence/EBBR.seq' as "ACS-IR v21.09_1.0 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, failure or warnings reported.

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

6. 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
- Introduction to SystemReady
- SystemReady IR
- U-Boot git repository

7. Next steps

In this guide, you learned how to prepare for SystemReady IR certification and perform 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. It ensures interoperability with embedded Linux and other embedded operating systems.

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

For support with the ACS, please send an email to support-systemready-acs@arm.com.

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

This is an example of how to build compliant firmware for an iMX8 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 iMX8 board, see the following repositories:

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

Appendix B Run the ACS-IR image on QEMU

To test SystemReady IR for QEMU, use a PC running Ubuntu 21.10 (Impish Indri) and install the packages, as shown in the following example:

```
$ sudo apt install git curl build-essential autoconf pkg-config libconfuse-dev \
    flex bison crossbuild-essential-arm64 libssl-dev bc uuid-dev \
    python3-distutils zip dosfstools mtools qemu-system-arm python-is-python3 \
    wget
```

Then download and run an automated script as follows:

```
$ git clone https://git.gitlab.arm.com/systemready/systemready-scripts.git -b ir1
$ ./systemready-scripts/ir-guide/acs-on-qemu
```

The `acs-on-qemu` script will perform the following intermediate operations:

- Download repo
- Compile a U-Boot firmware
- Prepare an EFI System Partition (ESP) image
- Download and uncompress the SystemReady ACS-IR pre-built image v21.09_1.0

Finally, the `acs-on-qemu` script will launch QEMU. The ACS-IR will start, as shown in the following screenshot:

Figure B-1: ACS-IR image starting on qemu

B.1 Advises

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
```

Just restart the `acs-on-qemu` script to finish running the ACS.

If the ACS halts during SCT with the following error:

```
System will cold reset after 1 second. please run this test again...resetting...
ERROR:   QEMU System Reset: with GPIO.
```

It is likely that the QEMU version is too old and does not support reset.

Appendix C Rebuild the ACS-IR image

These steps are from ACS build steps in the SystemReady documentation.

Prerequisites

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

- Ubuntu 18.04 or 20.04 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, `git config --global user.name Your Name` and `git config --global user.email Your Email` must be configured.

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: `v21.07_0.9_BETA`:

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

2. Update the `common_config.cfg` in the path `arm-systemready/common/config/common_config.cfg` to change the kernel version.
3. Change the parameter `LINUX_KERNEL_VERSION=5.13` to `LINUX_KERNEL_VERSION=<New Version>`.
4. Make the following changes to disable linux bsa:

```
diff --git a/common/ramdisk/files.txt b/common/ramdisk/files.txt
 file /bin/fwts                ./fwts_output/bin/fwts 755 0 0
-file /bin/bsa                 ./linux-bsa/bsa 755 0 0
-file /lib/modules/bsa_acs.ko  ./linux-bsa/bsa_acs.ko 755 0 0
 file /lib/libbsd.so.0         ./fwts_build_dep/libbsd.so.0 755 0 0
 file /lib/libfdt.so.1         ./fwts_build_dep/libfdt.so.1 755 0 0
 file /lib/libgio-2.0.so.0     ./fwts_build_dep/libgio-2.0.so.0 755 0 0

diff --git a/common/scripts/build-all.sh b/common/scripts/build-all.sh
 source ./build-scripts/build-sct.sh $@
 source ./build-scripts/build-uefi-apps.sh $@
 source ./build-scripts/build-linux.sh
-source ./build-scripts/build-linux-bsa.sh
 source ./build-scripts/build-grub.sh
 source ./build-scripts/build-fwts.sh $@
 source ./build-scripts/build-busybox.sh

diff --git a/common/scripts/get_source.sh b/common/scripts/get_source.sh
 get_linux_src
 get_cross_compiler
 get_fwts_src
-get_linux-acs_src
```

5. Navigate to the `IR/scripts` directory as shown:

```
cd /path-to/arm-systemready/IR/scripts
```

6. Run `get_source.sh` to download the sources and tools for the build. Provide the sudo permission as follows:

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

7. 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/scripts/output/ir_acs_live_image.img.xz`

The image is generated in a compressed (`.xz`) format. The image must be uncompressed before it is used.

Appendix D Test checklist

The following summarizes the steps 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. Run Linux BSA test manually as described in the Run Linux BSA section in Test SystemReady IR.
6. Install two Linux distributions and perform OS tests manually as described in the Test Linux Distributions installation section in Test SystemReady IR.
7. 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.

General

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 two important 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 config options are enabled.

Can I certify using a custom kernel?

No. Certification requires evidence that mainline Linux or BSD works on the platform. Unmodified third party distros 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.

Certification testing

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 can be found on Arm Gitlab.

How do I fix Variable services test failures?

Firmware must have the ability to set UEFI variable and persist the value over reboots. If variable values do not persist over reboot, you will get VariableServices 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 will use the partition to store variables and the failure will stop.

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.