



Arm SystemReady IR

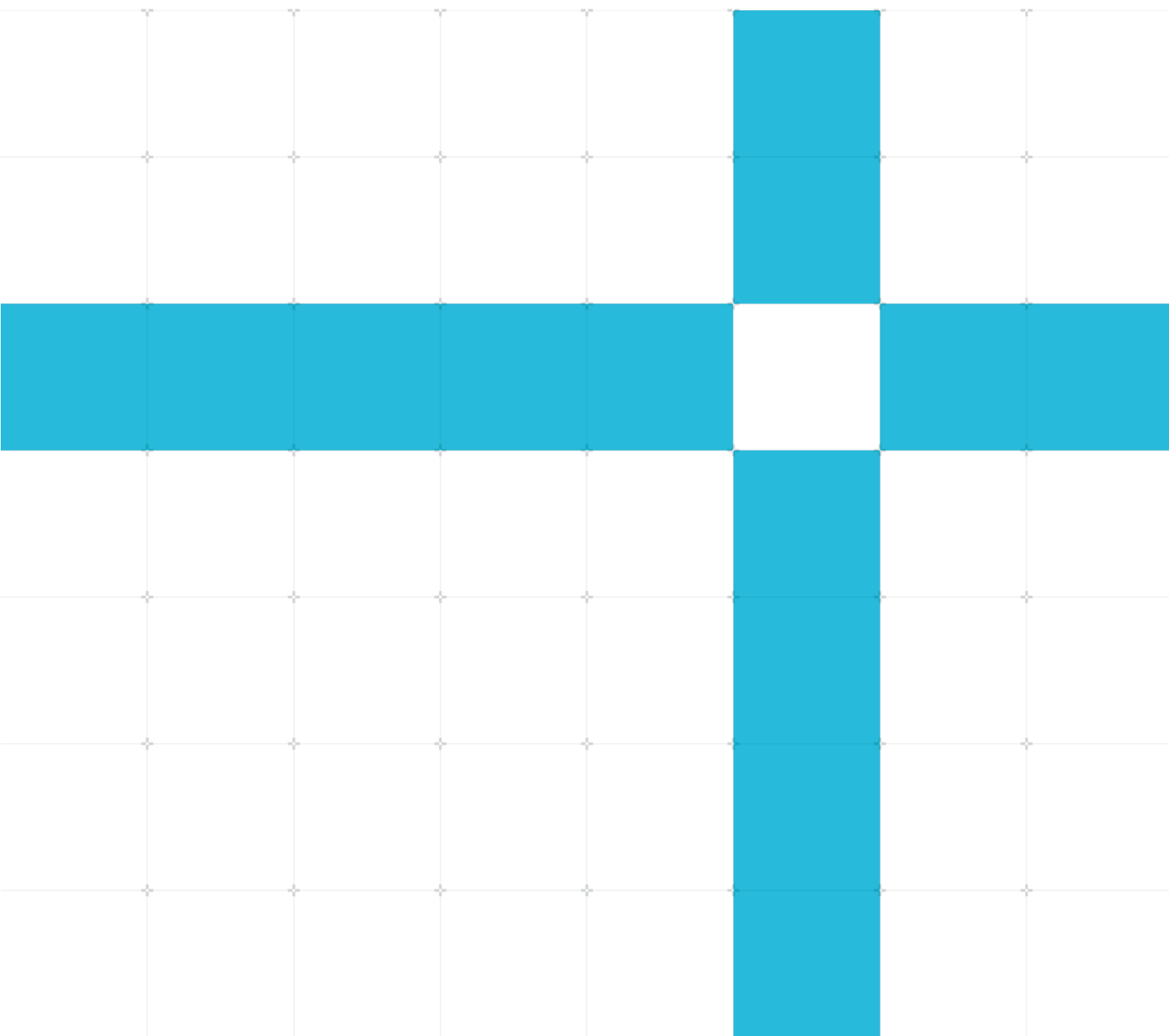
Deploying Yocto on SystemReady IR compliant hardware

Non-Confidential

Copyright © 2021 Arm Limited (or its affiliates).
All rights reserved.

Issue 01

DUI1102



Arm SystemReady IR

Deploying Yocto on SystemReady IR compliant hardware

Copyright © 2021 Arm Limited (or its affiliates). All rights reserved.

Release information

Document history

Issue	Date	Confidentiality	Change
01	11 October 2021	Non-Confidential	First version

Non-Confidential Proprietary Notice

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

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

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

This document may include technical inaccuracies or typographical errors.

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

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

If any of the provisions contained in these terms conflict with any of the provisions of any click through or signed written agreement covering this document with Arm, then the click through or signed written agreement prevails over and supersedes the conflicting provisions of these terms. This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its affiliates) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this

document may be the trademarks of their respective owners. Please follow Arm's trademark usage guidelines at <http://www.arm.com/company/policies/trademarks>.

Copyright © 2021 Arm Limited (or its affiliates). All rights reserved.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349)

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.

Web Address

www.arm.com

Contents

1 Overview	5
2 What is SystemReady IR?	6
2.1 Arm Base System Architecture.....	6
2.2 Embedded Base Boot Requirements	7
2.3 Devicetree specification	7
3 UEFI overview.....	9
3.1 UEFI boot process	10
3.2 Create a SystemReady IR bootable OS image.....	11
4 Yocto Project overview.....	12
4.1 SystemReady IR for Yocto	12
5 Example: deployment on an NXP board.....	14
5.1 Make the board SystemReady IR compatible.....	14
5.2 Build a generic SystemReady IR Yocto image.....	16
6 The meta-arm layer	18
7 Glossary	19
8 Related information	21
9 Next steps.....	22

1 Overview

SystemReady IR 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.

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

In this guide, learn how SystemReady IR compliant platforms make it easier to build, deploy, and maintain YP solutions. This guide focuses on the SystemReady IR band of the SystemReady program. This band addresses the needs of the embedded Linux ecosystem and is relevant for YP developers. All examples in this guide use SystemReady IR compliant platforms.

This guide teaches you about the following:

- The SystemReady IR boot flow
- A reference deployment example on an Arm-based NXP board
- Information about where to find the necessary components to get YP up and running on a SystemReady IR compliant platform

You can support SystemReady SR and ES certified platforms with the same YP configuration, but this configuration is not described in this guide.

2 What is SystemReady IR?

Cloud native software development is crucial to enable large scale IoT deployments. Working to cloud native principles, you can build systems in which the cloud and edge work together, with the benefits of each type of compute.

To help the growth of a cloud native edge ecosystem and support the continued growth and innovation in IoT, Arm launched the Project Cassini initiative. Project Cassini is an open, collaborative, standards-based initiative formed by three pillars: standardization, security, and ecosystem. SystemReady IR makes up the standardization pillar of Project Cassini and will be the focus of this guide. For more information about Project Cassini, see [Project Cassini](#).

IoT devices are generally Linux-based and are used in industries ranging from complex smart cameras to small industrial controllers. To enable these use cases, a flexible SoC ecosystem is necessary to implement unique feature sets to meet specific requirements. This flexibility can lead to fragmentation, where systems lack a set of standards for common features and interfaces. As a result, fragmentation affects developers and the broader IoT ecosystem.

IoT software typically needs to be developed for specific hardware targets. Each platform has different dependencies and requires a unique software stack. Debugging and ongoing maintenance can become costly, however SystemReady IR addresses fragmentation in IoT.

SystemReady is a compliance certification program with separate bands targeting different parts of the Arm ecosystem. For all bands, SystemReady compliant platforms use a standard firmware interface that simplifies the boot process and removes the need for platform-specific boot support. SystemReady IR provides standards for Linux and other operating systems that use the devicetree specification. These standards enable you to easily deploy a standard software stack across all SystemReady compliant platforms. Effort spent creating platform-specific software stacks can be redirected to more important tasks, such as building differentiating features.

SystemReady IR certified platforms use a minimum set of hardware and firmware features and must be compliant with the following requirements:

- [Arm Base System Architecture \(BSA\)](#) for 64-bit
- [Embedded Base Boot Requirements \(EBBR\)](#)
- [EBBR recipe of the Arm Base Boot Requirements](#)
- [Devicetree Specification](#)

Platforms are tested for compliance using the Architectural Compliance Suite (ACS) tool for the relevant SystemReady band. For more information about the certification process for SystemReady IR, see the [SystemReady IR - IoT integration, test, and certification](#).

2.1 Arm Base System Architecture

The Arm Base System Architecture (BSA) offers a standard hardware architecture for SoC designers. This architecture allows you to optimize the SoC for specific use cases and retain the minimum

feature set to enable software combability. The cost savings from removing additional hardware features for specific SoCs is offset by the complication of developing software across different hardware. Meeting minimum hardware specifications as defined by BSA simplifies this optimization process.

The BSA specifies a 64-bit hardware system based on the Arm architecture, with baseline requirements covering processing elements, memory subsystems, interrupts, Memory Management Units (MMU), PCIe, peripherals, and other power and security features. The BSA describes the minimum set of features required for an OS to boot and run. Further requirements can be added on top of these baseline requirements for specific market segments, for example, the BSA server supplement. The baseline BSA specification is sufficient for IoT.

On AArch64 platforms for SystemReady IR compliance, you must run the BSA tests included in the ACS test tool and report the results. However, the results are informational and the platform is not required to pass all BSA tests for certification. BSA does not specify 32-bit hardware requirements and there are no BSA test cases for AArch32 platforms.

2.2 Embedded Base Boot Requirements

The Embedded Base Boot Requirements (EBBR) addresses fragmentation in embedded boot sequences. Embedded platforms traditionally implement a bespoke boot sequence and as a result, require platform-specific modifications to the OS. The EBBR specifies boot features that ensure OSs can boot without modifications. As embedded deployments become larger and more diverse, standardized boot behavior becomes more important.

The EBBR specification defines features at the interface between the firmware and an OS or hypervisor. EBBR-compliant platforms present a consistent firmware interface to the OS, allowing compliant OSes to boot unmodified and to employ advanced features including Secure Boot and firmware updates. The EBBR uses the Unified Extensible Firmware Interface (UEFI) specification to define the boot and runtime services expected by an OS or hypervisor.

The [Arm Base Boot Requirements](#) (BBR) covers all SystemReady bands. This specification includes an EBBR recipe that specifies how BBR compliance is met when implementing the EBBR requirements.

2.3 Devicetree specification

The devicetree specification defines a data structure which describes the hardware components of a particular system. The bootloader loads both the kernel image and the Devicetree Blob (DTB). The OS's kernel then reads the devicetree information, allowing it to use and manage the hardware components. Devicetrees move the hardware description out of the kernel binary, helping to reduce the number of kernel forks needed to support specific hardware.

Traditionally in embedded Linux systems the devicetree has been included as part of the OS image, and the OS needs to include a devicetree for every supported platform. The difference with SystemReady IR is that firmware is required to provide a default devicetree that is suitable for booting the Linux kernel so that the OS is not required to provide its own copy. This makes it possible to deploy a single image on a much wider range of hardware.

The default devicetree requirement is a requirement on the firmware, not on the OS. The OS is not required to use the default devicetree and can install a replacement at boot time if needed.

3 UEFI overview

The UEFI standard defines the Application Binary Interface (ABI) exposed by the firmware that an OS uses. UEFI chooses which binary to load and execute, defines a set of abstract interfaces for accessing the console, network, storage, and other devices, and details how control is handed over from firmware to an OS. The most well-known open source implementation of UEFI compliant firmware is the Tianocore [EDK2 reference implementation](#). U-Boot also implements a subset of the UEFI ABI. Most IR compliant platforms use the U-Boot implementation.

UEFI firmware works by being able to find, load, and execute UEFI applications. This firmware provides basic access to devices until the OS is ready to take control of the platform and use its own device drivers. UEFI applications are executable code in PE/COFF format. The firmware doesn't need to know what the application does. When a UEFI application is run, it can either run to completion and return control back to firmware, or transfer control to an OS which assumes control of the hardware. The first type of application contains standalone utilities, boot menus, or other tools. The second type of UEFI application is referred to as an OS loader. A UEFI application can also load and execute another UEFI application. For example, a boot menu application like GRUB can load and execute an OS loader such as the UEFI stub embedded in a Linux kernel image.

Examples of UEFI applications are as follows:

- GRUB, a boot system used by Linux distributions to display a menu of boot options
- Memtest86, an industry standard memory stress testing application
- UEFI shell
- Linux kernel. The Linux kernel itself is a UEFI application. The kernel contains a UEFI stub that calls `ExitBootServices()` and sets up the execution environment before jumping into the kernel.
- Tianocore EDK2 SCT, the standard UEFI Self Certification Test suite
- Doom

By design, UEFI is a simplified execution environment. Applications are single threaded using simple memory management. UEFI does not provide scheduling services. One UEFI application runs at a time. UEFI is designed to provide enough functionality for an OS to perform pre-boot actions, such as choosing a specific kernel version or loading additional files, before handing control to the OS. The defined ABIs, which are also called protocols, provide abstract access to network, storage, console, variable, and memory management services.

Also included in UEFI is the GUID Partition Table (GPT) format for partitioning block devices. GPT replaces the MBR (Master Boot Record) method of partitioning because GPT supports larger devices, can handle a greater number of partitions, and is robust against corruption.

3.1 UEFI boot process

This section explains how UEFI boot occurs when the device is powered on. The UEFI components and the OS are generic and independent of the system firmware. There are no platform-specific customizations. The information in this section applies to any SystemReady compliant platform.

When the platform is released from reset, the firmware performs an internal initialization before performing Boot Device Selection to locate and run a UEFI application. Boot Device Selection begins by using the `BootOrder` and `BootXXXX` variables to find a suitable UEFI application. Each of the `BootXXXX` variables contains the name of the boot option and a path to a UEFI executable in the form `/Device/path/in device`. Use the `efibootmgr` utility on Linux to see examples of `BootXXXX` entries, such as the following example code:

```
root@toybrick-debian:~# efibootmgr -v
BootCurrent: 0001
BootOrder: 0001
Boot0001* Debian      VenHw(e61d73b9-a384-4acc-aeab-82e828f3628b)/eMMC(0)/eMMC(1)/HD(3,GPT,5f6b0000-0000-403e-8000-0d7d00000b89,0x6000,0x100800)/File(EFI\debian\shimaa64.efi)
root@toybrick-debian:~#
```

`BootOrder` is a list of numbers in the form 0001, 0004, 0002. `BootOrder` corresponds to the `BootXXXX` variables that tell firmware where to find a UEFI application. In the code example, the firmware attempts to load and execute the UEFI applications pointed to by `Boot0001`, `Boot0004`, and `Boot0002`, in that order. Firmware will attempt to execute each `BootXXXX` entry in `BootOrder` order. If an application cannot be found or if it exits and returns to firmware, firmware continues to the next `BootOrder` entry until the list is complete.

Typically, the `BootXXXX` entries point to a file stored in the EFI System Partition (ESP). The ESP is a FAT formatted partition that firmware uses for storing boot applications and other utilities. When an OS is installed, the OS copies boot applications into the ESP so that firmware can find and read these applications.

If `BootOrder` is not defined or none of the `BootXXXX` targets can be run, firmware falls back to the default boot targets. These targets are either the removable device path of `/EFI/BOOT/BOOT<ARCH>.EFI` on any device with an ESP, or network boot. The order of the default boot targets is implementation defined and can usually be controlled by the user. Most EDK2 implementations provide a boot menu to the user. In U-Boot implementations, the default boot order is typically defined by the `boot_targets` variable, as shown in the following code:

```
U-Boot 2021.07-dirty (Aug 06 2021 - 21:43:32 +0100)

SoC: Rockchip rk3399
Reset cause: RST
Model: Rockchip RK3399 Toybrick ProD Board
DRAM:  2 GiB
```

```
PMIC: RK8090 (on=0x40, off=0x00)
MMC:  mmc@fe320000: 1, sdhci@fe330000: 0
Loading Environment from MMC... OK
In:    serial
Out:   serial
Err:   serial
Model: Rockchip RK3399 Toybrick ProD Board
Net:
Warning: ethernet@fe300000 (eth0) using random MAC address - ce:82:85:cf:61:0d
eth0: ethernet@fe300000
Hit any key to stop autoboot:  0
=> printenv boot_targets
boot_targets=mmc0 mmc1 usb0 pxe dhcp sf0
=>
```

3.2 Create a SystemReady IR bootable OS image

This section describes how an OS uses UEFI firmware to boot. Linux, BSD, and other operating systems contain support for the UEFI ABI. To build a bootable operating system image, the executable files must be in the right place.

For initial provisioning, a device will not have `BootOrder` or `BootXXXX` variables set, and the platform will boot from one of the default boot targets. The most common boot targets use the removable media boot path on a block device or boot over the network using DHCP/TFTP, iSCSI, or HTTPS. Currently, U-Boot implements TFTP boot and iPXE can be used for iSCSI boot. U-Boot does not have an implementation of HTTPS boot. In this guide, we boot from a block device like a USB drive.

For firmware to treat the block device as bootable, the firmware needs to be able to find an ESP partition containing a `/efi/boot/boot<architecture>.efi` file. For example, on AArch64 the boot file is `/efi/boot/bootaa64.efi`. For GPT formatted disks, the ESP must be FAT formatted and tagged with partition type 0xef00. ISO disk images do not need a separate ESP but must contain a `/efi/boot/bootaa64.efi` file.

To create a bootable image in GRUB, copy `grub.efi` to `/efi/boot/bootaa64.efi` and put the GRUB config file in the same directory. Other UEFI boot applications like systemd-boot are installed in a similar way. For more information, refer to the documentation for your boot application.

After an OS has been copied to local storage, it can replace the removable boot path with a `BootXXXX` variable. This variable allows the OS to specify the install path and a selection of UEFI boot targets. Linux distribution installers use the `SetVariable()` UEFI API to set `BootXXXX` after the boot loader is copied to the ESP.

4 Yocto Project overview

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

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

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

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

An example hierarchy can include the following layers:

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

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

4.1 SystemReady IR for Yocto

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

SystemReady IR, however Yocto builds can boot without modification across platforms compliant with SystemReady. For example, a Yocto build can be developed on a SystemReady SR compliant server then be seamlessly ported to production silicon that is SystemReady IR.

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

5 Example: deployment on an NXP board

Before you begin, you will need the following:

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

5.1 Make the board SystemReady IR compatible

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

	SW1	SW2	SW3	SW4	SW5	SW6	SW7	SW8	SW9	SW0
Top row	1	0	1	0	X	X	X	X	X	X
Bottom row	X	X	X	X	X	X	X	X	X	0
1=Switch Up, 0= Switch Down, X= Either										

Table 1: Boot mode switch settings

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

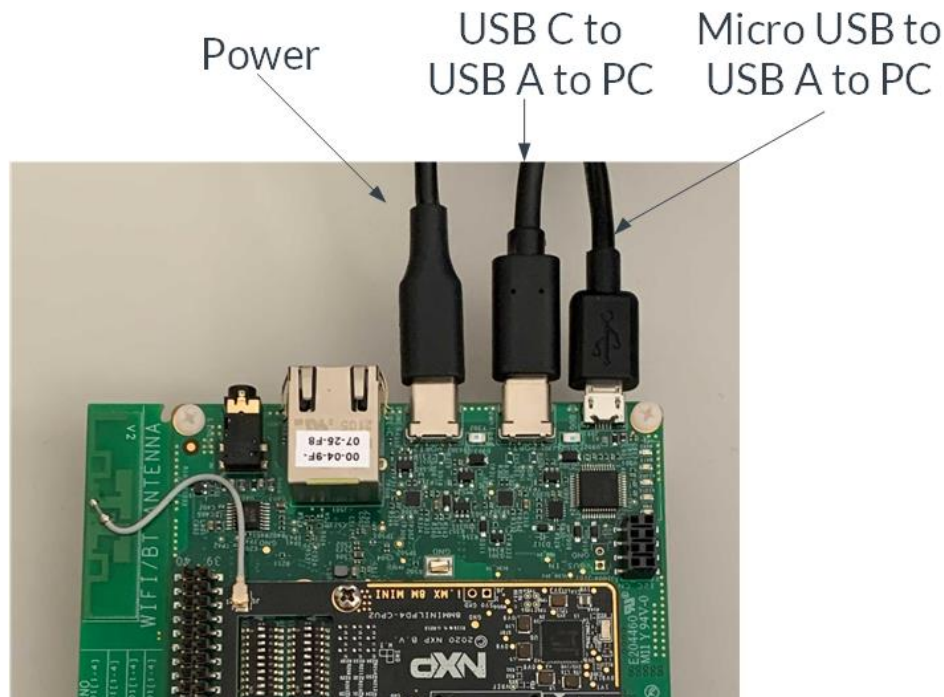


Figure 1: Cable connection

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

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

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

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

	SW1	SW2	SW3	SW4	SW5	SW6	SW7	SW8	SW9	SW0
Top Row	0	1	1	0	1	1	0	0	0	1
Bottom Row	0	0	0	1	0	1	0	1	0	0
1=Switch Up, 0= Switch Down										

Table 2: eMMC mode switch settings

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

5.2 Build a generic SystemReady IR Yocto image

Now that the board is SystemReady IR compatible, the next step is to build a base generic Yocto image that will boot on any SystemReady compatible platform.

1. Make sure all the Yocto build prerequisites are met, as described in [Yocto Project Quick Build](#).
2. Set up Poky and the meta-arm layers using the following code:

```
$ git clone git://git.yoctoproject.org/poky
$ git clone git://git.yoctoproject.org/meta-arm
$ cd poky
$ source oe-init-build-env
$ bitbake-layers add-layer ../../meta-arm/meta-arm-toolchain
$ bitbake-layers add-layer ../../meta-arm/meta-arm
```

3. Change the MACHINE config value in the conf/local.conf config file to generic-arm64, as shown in the following snippet:

```
$ sed -i 's/qemux86-64/generic-arm64/' conf/local.conf
```

Generic-arm 64 is a generic SystemReady IR aarch64 machine.

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

```
$ echo 'DISTRO_FEATURES += " systemd"
VIRTUAL-RUNTIME_init_manager = "systemd"
DISTRO_FEATURES_BACKFILL_CONSIDERED += "sysvinit"' >> conf/local.conf
```

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

5. Build the Yocto image using the following code:

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

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

```
sudo dd if=tmp/deploy/images/generic-arm64/core-image-base-generic-arm64.wic
of=/dev/sdX bs=4M
```


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

```
[ OK ] Reached target Basic System.
[ OK ] Started Kernel Logging Service.
[ OK ] Started System Logging Service.
[ OK ] Started D-Bus System Message Bus.
[ OK ] Started Getty on tty1.
[ OK ] Started Serial Getty on ttymxc1.
[ OK ] Reached target Login Prompts.
      Starting User Login Management...
[ OK ] Started User Login Management.
[ OK ] Reached target Multi-User System.
      Starting Update UTMP about System Runlevel Changes...
[ OK ] Finished Update UTMP about System Runlevel Changes.
[  8.655503] random: crng init done
[  8.658932] random: 7 urandom warning(s) missed due to ratelimiting
[ OK ] Finished Load/Save Random Seed.

Poky (Yocto Project Reference Distro) 3.3+snapshot-5dce2f3da20a14c0eb5229696561b0c5f6fce54c generic-arm64 ttymxc1
generic-arm64 login: █
```

Figure 2: Boot display

The Yocto image is now installed on the NXP board.

6 The meta-arm layer

meta-arm is a layer with recipes specific for Arm platforms, and contains generic-arm64 and qemuarm64-sbsa QEMU machines.

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

Recipe	Description
Android Common Kernel	Downstream of kernel.org kernels, including selected patches that have not been merged into the mainline or Long Term Supported (LTS) kernel
Arm FVP – Architecture Envelope Model	Support for Fixed Virtual Platform (FVP) Architecture Envelope Models (AEM)
Arm FVP – Library Ecosystem Reference Design	Support for Arm FVP library, which includes all CPU FVPs
DS-5 Streamline Gator daemon	Daemon for gathering data for Arm Streamline Performance Analyser (part of Arm Development Studio)
Hafnium	A reference Secure Partition Manager (SPM) for systems that implement the Armv8.4-A Secure-EL2 extension, enabling multiple, isolated Secure Partitions (SPs) to run at Secure-EL1
OpenCSD	API for decoding trace streams from Arm CoreSight trace hardware
OP-TEE	Trusted Execution Environment (TEE) designed as companion to a non-secure Linux kernel running on Cortex-A cores using TrustZone
SCP Firmware	System Control Processor (SCP) and Manageability Control Processor (MCP) firmware reference implementation
Tianocore EDK2	Open-source implementation of UEFI
Trusted Firmware-A	Reference implementation of secure world software for Cortex-A
Trusted Firmware for Cortex-M	Reference implementation of secure world software for Cortex-M

Table 3: meta-arm layer recipes

7 Glossary

Abbreviations and terms used in this document are defined in the following table:

Term	Abbreviation	Definition
AArch32/64	-	32/64-bit version of the Arm architecture.
Application Binary Interface	ABI	Describes the low-level interface between two binaries, one of which is often an OS.
Advanced Configuration and Power Interface	ACPI	An open standard that OSs can use to discover and configure hardware components.
Architecture Compliance Suite	ACS	Test suite from Arm to evaluate platforms against EBBR and BSA specifications.
Base Boot Requirements	BBR	Set of requirements for boot and runtime services that system software can rely on. BBR is a superset of EBBR.
Base Standard Architecture	BSA	Specification for a hardware system architecture based on AArch64. See Arm Base System Architecture .
Basic Input/Output System	BIOS	Firmware that performs hardware initialization during system boot, largely superseded by UEFI.
Board Support Package	BSP	Software containing hardware specific drivers.
Cloud native	-	Method of building and running applications in a way that leverages cloud computing.
Devicetree	-	A data structure which describes the hardware components of a particular system. See Devicetree specification .
Devicetree blob	DTB	A format that is a flat binary encoding of Devicetree data.
Embedded Base Boot Requirements	EBBR	Subset of the BBR targeted at embedded devices. See Embedded Base Boot Requirements .
Globally Unique Identifiers	GUID	128-bit reference numbers that are unlikely to repeat and are considered unique.
GUID Partition Table	GPT	A standard layout for partition tables for storage devices using GUIDs.
GNU GR and Unified Bootloader	GRUB	Bootloader package from the GNU Project which displays a boot menu.
OpenEmbedded	-	Build automation and cross-compile environment used to create Linux distributions for embedded environments.
Poky	-	Reference embedded OS for the Yocto project. See Yocto Project overview .
Project Cassini	-	Standards-based initiative from Arm, with a focus on standardization, security, and ecosystem.
UEFI Secure Boot	-	Protocol to secure the boot process by verifying loaded UEFI driver or OS boot loader signatures against known keys.
System Management BIOS	SMBIOS	Specification that defines data structures that can be used to read information produced by the BIOS.

Term	Abbreviation	Definition
SystemReady	-	Compliance certification program from Arm based on a set of hardware and firmware standards. See What is SystemReady IR?
Tianocore EDK2	-	Open-source UEFI implementation.
U-boot	-	Open-source primary bootloader (first and second stage) often used in embedded devices.
Unified Extensible Firmware Interface	UEFI	A standard that defines an ABI that is exposed by the firmware for an OS to use. See UEFI overview .
Yocto layer	-	A collection of related recipes organized as a modular block. See Yocto Project overview .

8 Related information

The following are resources related to material in this guide:

- [Arm Base System Architecture \(BSA\) specification](#)
- [Embedded Base Boot Requirements \(EBBR\) specification](#)
- [EBBR recipe of the Arm Base Boot Requirements \(BBR\)](#)
- [SystemReady IR](#)
- [SystemReady IR ACS GitHub repository](#)
- [SystemReady IR - IoT integration, test, and certification](#)
- [Yocto Project Documentation](#)

9 Next steps

In this guide, you learned about SystemReady, the Yocto Project, and how the two work together. As a next step, you can follow the example in [Example: deployment on an NXP board](#) to deploy a Yocto build on SystemReady compliant hardware.