Application Note

Linux Board Porting on AM62Lx Devices

Judith Mendez

Abstract

The portfolio of AM62Lx System-on-Chips (SoCs) is designed for building a lite, low-power, and performance optimized Linux®-based, embedded systems. AM62Lx is suitable for a wide range of general-purpose applications such as human machine interfaces (HMI), medical patient monitoring, building automation, smart secure gateways, smart thermostats, EV charging stations, smart metering, solar energy and more. This SoC family offers advanced features to meet the needs of today's systems. Fully leveraging these features with the development of custom platforms using these devices can be complex for new users. This application note improves the development efficiency for custom boards based on AM62Lx SoCs by providing a clear starting point for Linux development. For more information about the AM62Lx products, see the device-specific datasheet, EVM user's guide and AM62Lx Processor SDK documentation.

Table of Contents

1 Introduction 3
2 Installing the SDK 3
3 Understanding the AM62Lx Bootflow 4
4 TF-A Build System Integration 5
5 U-Boot Build System Integration 5
6 Minimal Devicetree and Reaching the U-Boot Prompt 7
6.1 Introduction to Devicetrees 7
6.2 Minimal Platform Support 8
6.3 Adding Custom Board Devicetrees 8
6.4 Initial Devicetree Modifications 10
6.5 Building U-Boot 12
6.6 Reaching the U-Boot Prompt 13
7 Expanding the Devicetree 14
7.1 Adding New Devicetree Nodes 15
7.2 Revising Devicetree Nodes 16
7.3 Devicetree Development Tips 17
8 Booting the Linux Kernel 19
8.1 Boot Method Overview 19
8.2 Reaching the Linux Command Prompt 19
9 Tools and Debugging 22
9.1 Debug Traces 22
9.2 OpenOCD 22
10 Future Work 23
11 Summary 23
12 References 23
Revision History 24

List of Tables

Table 6-1. 8
Table 6-2. Environment Variables 12
Table 6-3. Generated Binaries Based on Device Security Type 13
Table 8-1. SDK Paths to Linux Kernel Components 19

Copyright © 2026 Texas Instruments Incorporated

Trademarks

myTI™ and Sitara™ are trademarks of Texas Instruments.
Linux® is a registered trademark of Linus Torvalds.
All trademarks are the property of their respective owners.

Copyright © 2026 Texas Instruments Incorporated

1 Introduction

TI provides a Software Development Kit (SDK) to accelerate development on products using AM62Lx SoC. This software package is tuned for Evaluation Modules (EVMs). The SDK and EVMs are designed for quick evaluation of the SoC's many capabilities for a variety of use cases. Many features and capabilities, such as a variety of boot modes, are enabled by default. While this is great for an evaluation platform, it can introduce unneeded complexity for board-porting exercises on custom boards that target specific use cases. For example, on some boards like BeagleBadge, only four boot modes are used in the end product, whilst AM62Lx EVM supports approximately 10 different boot modes.

Removing complexity from the finished AM62Lx EVM configurations may not be the best method for a board-porting process; it is inefficient and can extend bring-up time from days to weeks since the developer is arbitrarily disabling features to determine how to configure a simpler use case. Instead, the recommended approach is to start with a minimal configuration as a strong foundation and iterate toward a complete and optimized custom configuration. Incrementally adding one feature at a time quickly identifies which integrations are functional and which ones are not. Non-functional features can then be isolated and debugged.

This application note details a step-by-step process for adding custom board support using the Linux SDK. The SDK contains source code repositories for the TI Linux kernel, TI TF-A, and TI U-Boot bootloader, which must be ported to the custom board. Generally, follow these steps:

1. Install required packages for building TI U-Boot and TI Linux repositories and the Linux SDK
2. Download and install the Linux SDK
3. Integrate the custom board infrastructure into the TF-A build system
4. Integrate the custom board infrastructure into the U-Boot build system
5. Create the custom board devicetree and *Defconfig* profiles
6. Update the initial devicetree according to custom board hardware pin layout and SysConfig tool
7. Enable a simple peripheral capable of loading the Linux kernel and rootfs
8. Boot to the Linux kernel prompt
9. Iteratively enable peripherals supported on the hardware until the hardware description is complete

The same board devicetree must be used for both U-boot and Linux to complete the porting process.

To navigate any errors that arise during the bring-up process, this document also includes some guidance on debugging.

TI recommends going through the process detailed in this document using a TI EVM prior to the custom board. This makes for an efficient bring-up process and also validates the steps in this document on a board that is known to be functional.

All download links can also be found in Section 12.

2 Installing the SDK

The first step in the board bring-up process is to download the AM62Lx Linux Processor SDK. If it is not already installed on the host machine, download instructions are found in the AM62Lx Processor SDK: Installer guide.

This SDK contains source code repositories, pre-built binaries, filesystem images, and other useful development tools. The following instructions are based off of the 12.x version of the AM62Lx Processor Linux SDK. Exact paths can change across SDK releases, so it may be necessary to search through directories to determine where components are located. This document refers to the SDK directory path as *$TI_SDK*.

In the 12.x version of the SDK, source repositories are found at:

1. TF-A repository: *$TI_SDK/board-support/trusted-firmware-a-2.15+git*. This document refers to this path as *$TI_TFA*.
2. U-Boot repository: *$TI_SDK/board-support/ti-u-boot-2026.01+git*. This document refers to this path as *$TI_U_BOOT*.
3. Linux kernel repository: *$TI_SDK/board-support/ti-linux-kernel-6.18.xx+git-ti* (where xx represents the minor release number that changes). This document refers to this path as *$TI_LINUX*.

Copyright © 2026 Texas Instruments Incorporated

3 Understanding the AM62Lx Bootflow

Because the AM62Lx boot flow shifts early hardware initialization from the R5 core to the A53 core, custom board support must be added to both the TF-A and U-Boot repositories. The following summary quickly details this boot flow to explain why both repositories require porting.

AM62Lx bootflow:

1. First Phase: Security Initialization and Hardware Setup
   a. SMS release: At power-on, the Security Management Subsystem (SMS) MCU core releases first to secure the device, following standard K3 architecture protocol.
   b. A53 ROM execution: Because the AM62Lx lacks secondary MCU cores, the primary Boot ROM executes directly on one of the Cortex-A53 cores. The Boot ROM loads TI's Root-of-Trust firmware (TIFS) into the security subsystem, alongside BL1 into internal RAM.
   c. BL1 execution: The BL1 binary initializes the debug console, configures the DDR controller, and scales the Cortex-A53 core to the maximum clock speed.
   d. Phase 1 to Phase 2 handshake: Upon completion, BL1 reaches an idle state. The Secure ROM running on the SMS core proceeds to reset the Cortex-A53 core, transitioning it into the next boot stage.
2. Second Phase: Trustzone Establishment and OS Handoff
   a. Public ROM load: With the DDR memory initialization complete, the A53 Public ROM executes the second phase of the boot process to load the larger *tispl.bin* image from the boot media.
   b. Components in *tispl.bin*: This image serves as a container aggregating multiple essential system layers:
      i. TIFS Firmware: Finalizes security runtime services within the SMS core
      ii. TF-A BL31: The runtime Secure Monitor
      iii. OP-TEE OS: The Secure World operating system
      iv. U-Boot SPL: The Secondary Program Loader
   c. Environment isolation: Execution shifts from the Boot ROM to BL31. This shift configures ARM TrustZone, locks down hardware firewalls, and strictly separates the processing environments.
   d. Final handoff: Control passes to the U-Boot SPL. The SPL completes platform configuration, then loads, authenticates, and executes U-Boot proper (the second-stage bootloader contained in *u-boot.img)*, which ultimately loads and launches the Linux kernel.

For additional information on the AM62Lx bootflow, see the "*Boot Process*" section in the device TRM.

In this new bootflow, the BL1 payload is generated from TF-A build artifacts, the following sections walk through adding custom board support in TF-A repository to generate said BL1 payload.

Copyright © 2026 Texas Instruments Incorporated

4 TF-A Build System Integration

This subsection provides a step-by-step guide to integrating custom board infrastructure into the TF-A build system. While these steps are not strictly required to begin early prototyping, it is best practice to establish dedicated board directories, header files, and Makefiles. Note that this section only configures the software build framework; board devicetree files are covered separately in the next section. Establishing these dedicated build files allows the user to:

1. Prevent board conflicts
2. Isolate board-specific configuration issues
3. Preserve the TI EVM files so they could be used as references

These instructions refer to the custom board's name as *<boardname>*.

1. Copy and paste the minimal support in *$TI_TFA/plat/ti/k3low/board/am62lx-minimal* and rename the *am62lx-minimal* folder name according to the custom board name.

```
$ cp -r $TI_TFA/plat/ti/k3low/board/am62lx-minimal $TI_TFA/plat/ti/k3low/board/<boardname>
```

2. If the custom board Main UART0 pin configuration (or WKUP UART0) is different than AM62Lx EVM, then *board_config.c* must be defined in the new directory as well:

```
$ cp $TI_TFA/plat/ti/k3low/board/am62lx/board_config.c $TI_TFA/plat/ti/k3low/board/<boardname>/
board_config.c
```

3. The UART pin configuration in *board_config.c board_init()* must be updated to match the pin layout of the custom board.
4. The *board.mk BL1_SOURCES* variable must be updated to use the custom board's *board_config.c*:

```
- ${PLAT_PATH}/board/am62lx/board_config.c   \
+ ${PLAT_PATH}/board/<boardname>/board_config.c   \
```

The infrastructure of the custom board is now integrated in the TF-A build system.

5 U-Boot Build System Integration

This subsection provides a step-by-step guide to integrating custom board infrastructure into the U-Boot build system. While these steps are not strictly required to begin early prototyping, it is best practice to establish dedicated board directories, Kconfig targets, and Makefiles. Note that this section only configures the software build framework; device tree files are covered separately in the next section. Establishing these dedicated build files allows the user to:

1. Prevent board conflicts
2. Isolate board-specific configuration issues
3. Preserve the TI EVM files so they can be used as references

These instructions refer to the custom board's name as *<boardname>* and the name of the company as *<company>*. The company's name is used when naming directories to help keep a company's boards grouped together.

1. Add the following line to the bottom of the *$TI_U_BOOT/arch/arm/mach-k3/am62lx/Kconfig* file, before *endif*.

```
+ source "board/<company>/<boardname>/Kconfig"
```

2. Make a new *<company>/<boardname>* board directory to store board files for the custom board.

```
$ mkdir -p $TI_U_BOOT/board/<company>/<boardname>/
```

3. Copy board files from am62lx board, as a starting point:

```
$ cp -r $TI_U_BOOT/board/ti/am62lx/* $TI_U_BOOT/board/<company>/<boardname>/
```


Copyright © 2026 Texas Instruments Incorporated

4. In the *$TI_U_BOOT/board/<company>/<boardname>* directory, edit the *Kconfig* file as seen below.

```
+ choice
+        prompt "<company> AM62L3 based <boardname>"
+        optional

+ config TARGET_<BOARDNAME>
+        bool "<COMPANY> <BOARDNAME>"
+        select ARM64
+        select BINMAN
+        select OF_UPSTREAM
+        select OF_SYSTEM_SETUP

+ endchoice

+ if TARGET_<BOARDNAME>
+ config SYS_BOARD
+        default "<boardname>"
+ config SYS_VENDOR
+        default "<company>"
+ config SYS_CONFIG_NAME
+        default "<boardname>"
+ source "board/ti/common/Kconfig"
+ endif
```

5. In the same directory, rename the file *evm.c* to *<boardname>.c* and the file *am62lx.env* to *<boardname>.env*.

```
$ mv $TI_U_BOOT/board/<company>/<boardname>/evm.c $TI_U_BOOT/board/<company>/<boardname>/
<boardname>.c
$ mv $TI_U_BOOT/board/<company>/<boardname>/am62lx.env $TI_U_BOOT/board/<company>/<boardname>/
<boardname>.env
```

6. In the same directory, edit the *Makefile* as seen below.

```
- obj-y    += evm.o
+ obj-y    += <boardname>.o
```

7. Copy the EVM board header file and rename it for the custom board.

```
$ cp $TI_U_BOOT/include/configs/am62l3_evm.h $TI_U_BOOT/include/configs/<boardname>.h
```

   Make the following modification inside the newly created header file.

```
- #ifndef __CONFIG_AM62L3_EVM_H
- #define __CONFIG_AM62L3_EVM_H
+ #ifndef __CONFIG_<BOARDNAME>_H
+ #define __CONFIG_<BOARDNAME>_H
```

8. Create the *Defconfig* files. Start by copying *am62lx_minimal_defconfig* as seen below.

```
$ cp $TI_U_BOOT/configs/am62lx_minimal_defconfig $TI_U_BOOT/configs/<boardname>_defconfig
```

   Add the CONFIG_TARGETs as seen below.

   <boardname>_defconfig

```
+ CONFIG_TARGET_<BOARDNAME>=y
+ # CONFIG_TARGET_AM62L3_EVM is not set
```

The infrastructure of the custom board is now integrated in the U-Boot build system.

Copyright © 2026 Texas Instruments Incorporated

6 Minimal Devicetree and Reaching the U-Boot Prompt

Porting U-Boot to a custom board largely involves creating a new devicetree file for the board. This file contains a tree-like structure that describes the hardware to the upper software levels, in this case U-Boot. For example, it includes board specific information to configure a UART port as a console output. A basic understanding of devicetrees will be helpful to complete this task and expedite the porting process.

6.1 Introduction to Devicetrees

A device tree contains nodes that represent peripherals. A node can contain properties that hold data used by U-Boot and Linux kernel drivers to initialize these peripherals. Without a device tree, the hardware configuration and parameters need to be hard-coded into drivers. Keeping the hardware configuration separate from the board initialization code simplifies development and maintenance.

Note

It is recommended to have a general understanding of device trees prior to beginning development. A good resource created by Bootlin can be found in Section 12.

It is useful to visualize the AM62Lx devicetree layout to understand what devicetree files must be created for a new AM62Lx-based board.

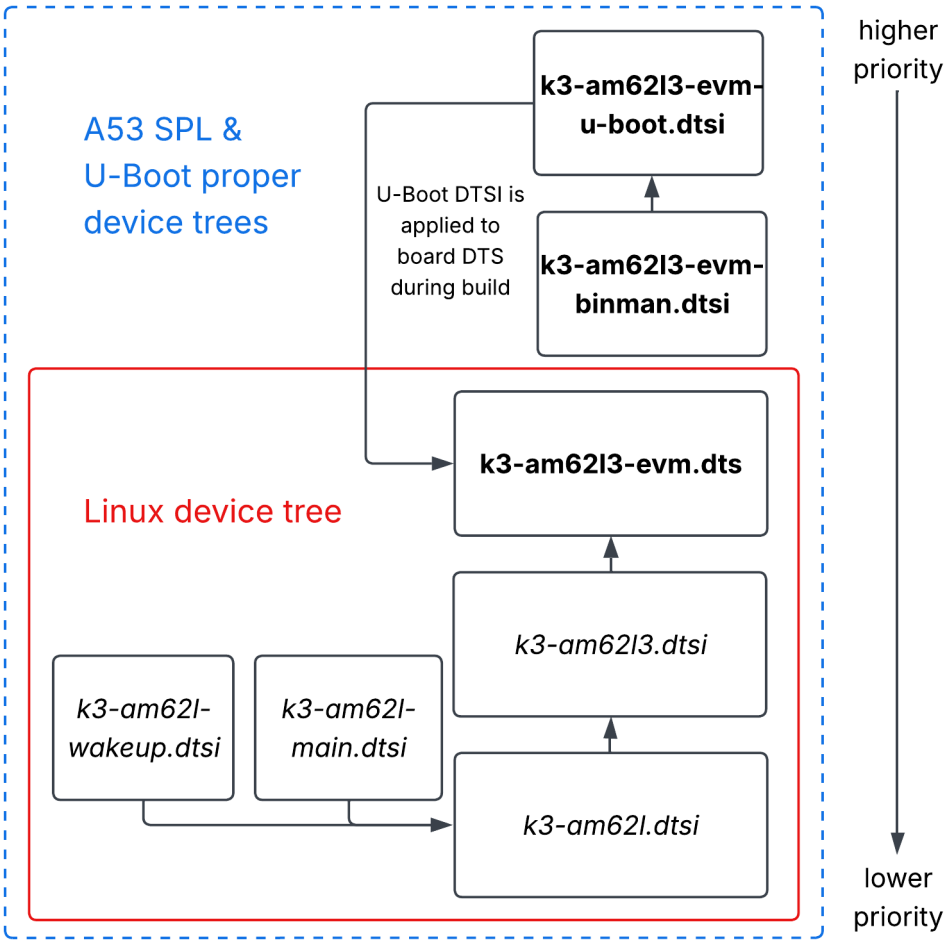


Figure 6-1. AM62Lx Devicetree Layout

Copyright © 2026 Texas Instruments Incorporated

The files emphasized in bold letters are the DTS or DTSI files that should be created for the custom board during a board-port process. The following list shows a summary of those files and the example final file name for the custom board:

Table 6-1.

AM62Lx EVM DTS or DTSI	Custom board DTS or DTSI
k3-am62l3-evm.dts	k3-am62lx-<boardname>.dts
k3-am62l3-evm-u-boot.dtsi	k3-am62lx-<boardname>-u-boot.dtsi
k3-am62l3-evm-binman.dtsi	k3-am62lx-<boardname>-binman.dtsi

The bottom of the Linux device tree (emphasized with italicized formatting) is formed by SoC Devicetree Include (DTSI) files. These are used to define hardware inside the silicon itself and are the same across all boards using AM62Lx chip. SoC DTSI files should always be included, directly or indirectly, by the Devicetree Source (DTS) files.

Other types of DTSI files can be created to group together common devicetree nodes shared between boards. These common files are standard practice for hardware revisions or boards that share a baseline design with minor component variations. Though a common DTSI will not be used in this guide, it is good to understand that a custom DTS file can include a common DTSI to prevent code duplication across multiple board designs.

DTS files describe board level hardware. During the board-port process, a developer should create custom DTS files within U-Boot and Linux sources to describe the on-board hardware. This process is detailed in Adding Custom Board Devicetrees.

When working on DTS/DTSI files, the following list shows a few useful concepts:

- If an inherited setting must be modified for a specific platform, the change must occur in the top-level file that includes the DTSI to avoid breaking other platforms that rely on the same shared file. This adjustment is achieved by referencing the specific node and overriding its properties with the correct target values.
- The device tree node defined last takes priority and overwrites previous states. For example, since *k3-am62l3-evm.dts* includes *k3-am62l3.dtsi* at the top of the file, the properties defined in the DTS file overwrites the properties in the DTSI file.

6.2 Minimal Platform Support

The minimal platform support consists of the minimal device tree and the *Defconfig* files that allow to build for a minimal platform in U-Boot and the Linux kernel. This minimal configuration can be used initially to test basic board functionality with a reduced set of peripherals before expanding the device trees with full feature support. For simplicity, the only peripherals initially supported by the minimal configuration are SD and UART. If a custom board relies on different peripherals to boot, or cannot use these peripherals to boot, new devicetree nodes will need to be created instead.

The minimal configuration enables the MMC controller in the slowest possible setting: *Legacy* mode. In this mode, the host controller communicates with an SD card at a 25MHz clock speed, Single Data Rate (SDR), using only 3.3V I/O signals. This setup functions on all SD cards and eliminates issues related to High Speed (HS) modes or Ultra-High Speed (UHS) voltage switching. Consequently, it prevents boot failures before reaching the Linux kernel prompt on boards that lack proper support for advanced SD card capabilities. Similarly, configuring DDR memory at a lower speed helps isolate intermittent high-speed issues common in new hardware board-porting processes, accelerating troubleshooting progress.

6.3 Adding Custom Board Devicetrees

In 12.x SDK, ARM64 platform devicetree files reside in *$TI_U_BOOT/dts/upstream/src/arm64*. Historically, U-Boot maintained its own separate copies of both binding documentation and devicetree files from the Linux kernel, which led to version divergence and redundant maintenance. To resolve this, U-Boot now mirrors the Linux kernel bindings and devicetree files via a source subtree that periodically syncs with upstream Linux. Because the 12.x SDK provides a fixed snapshot of the TI U-Boot source, this periodic sync does not occur automatically for the end user. Consequently, during compilation, U-Boot checks *CONFIG_DEFAULT_DEVICE_TREE* and searches *$TI_U_BOOT/dts/upstream/src/arm64/* for the target board

Copyright © 2026 Texas Instruments Incorporated

file (for example, *ti/k3-am62lx-<boardfile>.dts*). Developers must manually sync this exact DTS file with Linux later during the Linux board-port phase.

The following steps set up the custom board device trees and *Defconfig* files for development. It can be helpful to refer back to *AM62Lx Devicetree Layout* to understand how these device trees fit into the existing AM62Lx devicetree layout.

1. Create new devicetrees: Copy and rename the minimal device tree using the following commands. If the provided device trees are not present in the installed SDK, they can be downloaded in the References section.

```
$ cp $TI_U_BOOT/dts/upstream/src/arm64/ti/k3-am62l3-minimal.dts $TI_U_BOOT/dts/upstream/src/
arm64/ti/k3-am62lx-<boardname>.dts
$ cp $TI_U_BOOT/arch/arm/dts/k3-am62l3-minimal-u-boot.dtsi $TI_U_BOOT/arch/arm/dts/k3-am62lx-
<boardname>-u-boot.dtsi
```

2. Create Binman DTSI: Use the following command to duplicate the existing EVM binman devicetree file and rename it for a custom board.

```
$ cp $TI_U_BOOT/arch/arm/dts/k3-am62l3-evm-binman.dtsi $TI_U_BOOT/arch/arm/dts/k3-am62lx-
<boardname>-binman.dtsi
```

   Include this new file in *k3-am62lx-<boardname>-u-boot.dtsi.*

```
- #include "k3-am62l3-evm-binman.dtsi"
+ #include "k3-am62lx-<boardname>-binman.dtsi"
```

3. Generate DTSI for DDR Configuration: The DDR configuration used by TF-A for DDR initialization is contained in a DTSI file that is generated using TI's SysConfig tool. After launching the SysConfig tool and logging in to your myTI™ account, set the "*Software Product*" field to "*DDR Configuration for AM64x, AM625, AM623, AM62Ax, AM62Px*" and the "*Device*" field to "*AM62L*". Start the program to load the DDR configuration tool. Here, modify DDR settings to match the DDR hardware specification. Refer to the SysConfig DDR Configuration README for a complete guide to this tool. The generated file that needs to be included is named *k3-am62Lx-ddr-config.dtsi*. Download this file and run the below commands to integrate into the TF-A repository. The file name portion "*<#MTs>*" should be updated to reflect the final MTs value which is based-off the configured DDR frequency.

```
$ cp k3-am62Lx-ddr-config.dtsi $TI_TFA/ti/k3low/common/drivers/k3-ddrss/k3-am62lx-<boardname>-
lp4-<#MTs>.h
```

   Update the new header file by adding the following *#define* at the top:

```
+ #define DDRSS_PLL_FREQUENCY_0 25000000
```

   Update *$TI_TFA/plat/ti/k3low/board/<boardname>/board.mk*. The *DDR_CFG_HEADER* and *AM62L_DDR_RAM_SIZE* variables must be updated:

```
- DDR_CFG_HEADER          ?=      am62lx_skevm_lp4_50_800.h
+ DDR_CFG_HEADER          ?=      k3-am62lx-<boardname>-lp4-<#MTs>.h
- AM62L_DDR_RAM_SIZE      ?=      0x08000000ULL
+ AM62L_DDR_RAM_SIZE      ?=      <custom part DDR Size>
```

4. Setup the correct DTS in the defconfig files: Modify the defconfig files that re-use the existing AM62Lx EVM default configuration and change the device tree name. This defconfig file should exist in *$TI_U_BOOT/configs/* if the user followed the steps in Section 5. Update the line below in *<boardname>_defconfig*:

```
- CONFIG_DEFAULT_DEVICE_TREE="ti/k3-am62l3-minimal"
+ CONFIG_DEFAULT_DEVICE_TREE="ti/k3-am62lx-<boardname>"
```

The custom board configuration has now been set up within U-Boot and TF-A repositories.

Copyright © 2026 Texas Instruments Incorporated

6.4 Initial Devicetree Modifications

The device trees created in the U-Boot repository must be modified to function properly on the new custom hardware. This section walks through the necessary modifications required prior to attempting the first board boot.

The pin configurations in the custom board device tree were generated using TI's SysConfig PinMux Tool for a TI EVM. This application maps internal peripheral signals to physical external balls on the SoC package. Because of hardware multiplexing constraints, only specific pin combinations are valid. The tool identifies these restrictions and provides warnings and errors for invalid configurations. Finally, the tool exports the configuration files to integrate into the U-Boot source code. Board designers should use the SysConfig tool to develop valid pin configurations for their specific use case. The board designer must then pass these generated files to the software developer to ensure an efficient board bring-up process.

CAUTION

The board design and the software pin configuration must be kept strictly in sync. An invalid pin configuration in either the hardware design or the software can lead to system instability that is difficult to diagnose and debug. Any pin-related hardware change must be validated with the SysConfig PinMux Tool and immediately reflected in software.

Generate DTSI for pin configuration:

After launching the SysConfig tool and logging in to your myTI account, set the "*Device*" field to "*AM62L*". Start the program. Once within the PinMux tool, look at the left side column under "*PERIPHERALS*": Click the + sign next to the controller you want to route. Some controller instances are grouped together (for example, MCAN0, MCAN1, MCAN2), so the "*Use Peripheral*" dropdown can be used to further select a specific instance of a controller.

- Configuring Mux Modes: Under the individual signal dropdowns (for example., TXD or RXD), choose your target physical pin ball number
- Setting Pin Attributes: IO characteristics must be set explicitly:
  - Enable or disable the internal "*Pull Up/Down*" resistors
  - Set pin direction to "*PIN_INPUT*" or "*PIN_OUTPUT*" by toggling the "*Rx*" box

To review the final pin configurations, click on the *devicetree.dtsi* file listed in the rightmost column. This action launches an editor view with all configuration changes displayed in DTS language. See the SysConfig DDR Configuration README for a complete guide to this tool.

The default pin configurations in the minimal device tree are tailored for the TI EVM and do not require modification if the reference hardware is used. However, when deploying a custom board, replace these default settings with the custom pin configuration generated by TI's SysConfig tool. There are two methods to accomplish this:

- Method 1: Include the Generated DTSI File: Copy the SysConfig-generated DTSI file directly into the U-Boot source directory where the DTS file is located, and add an "*#include*" directive at the top of the *k3-am62lx-<boardname>.dts* file. This is the preferred industry practice because it separates automated, machine-generated code from manual board configurations and it keeps the main devicetree clean and simplifies future pinmux updates. With this method, remove the existing *pmx0* node in *k3-am62lx-<boardname>.dts* and rename the DTSI generated *main_pmx0* node name to *pmx0*.
- Method 2: Directly Paste the Pinmux Contents: Delete the contents of the *pmx0* node in *k3-am62lx-<boardname>.dts* and paste the raw text contents of the SysConfig-generated file in its place. This is the commonly used method on TI device trees, an example of this method is shown below.

```
&pmx0 {
- [... UART pin configuration ...]
- [... SD pin configuration ...]
};

&pmx0 {
+ [... New pin configurations ...]
};
```


Copyright © 2026 Texas Instruments Incorporated

After defining the pin settings, you must link the new configuration to its respective peripheral node. This is achieved by updating the *pinctrl-0* property within the peripheral node in *k3-am62lx-<boardname>.dts*.

Suppose the new pin configuration node, labeled "*uart0_custom_pins_default*", for UART0 looks like the following:

```
+ uart0_custom_pins_default: uart0-custom-default-pins {
+      pinctrl-single,pins = <
+           AM62LX_IOPAD(0x01b4, PIN_INPUT, 0) /* (D13) UART0_RXD */
+           AM62LX_IOPAD(0x01b8, PIN_OUTPUT, 0) /* (C13) UART0_TXD */
+      >;
+ };
```

To apply this configuration in software, modify the *pinctrl-0* property within the UART0 node as shown below. Ensure you use the exact reference label (e.g., *&uart0_custom_pins_default*) to prevent devicetree compilation errors.

```
&uart0 {
     bootph-all;
     status = "okay";
     pinctrl-names = "default";
-    pinctrl-0 = <&main_uart0_pins_default>;
+    pinctrl-0 = <&uart0_custom_pins_default>;
};
```

Repeat this process for all subsequent peripherals added to the new custom devicetree file.

The default peripheral instances configured in the reference devicetree template also need to be updated. The template assumes UART0 is designated for the serial console and UART boot, and that MMC1 serves as the SD card interface. If the custom hardware uses different peripheral instances, you must update the corresponding devicetree nodes to match the hardware.

In the example below, assume that the correct UART instance for the serial console is UART1. Modify the *k3-am62lx-<boardname>.dts* file as shown below to configure UART1 as the primary console output.

```
- &uart0 {
+ &uart1 {
     bootph-all;
     status = "okay";
     pinctrl-names = "default";
-    pinctrl-0 = <&uart0_custom_pins_default>;
+    pinctrl-0 = <&uart1_custom_pins_default>;
};
```

If a peripheral instance is changed (e.g., from *&uart0* to *&uart1)*, its pin configuration must also be updated. In the above example, the *pinctrl-0* property is updated to reference the pin configuration node for UART1.

If the UART instance for the serial console is changed, the chosen node must be updated accordingly. The example below demonstrates this modification:

```
chosen {
- stdout-path = &uart0;
+ stdout-path = &uart1;
};
```

This ensures the Linux kernel uses the correct UART instance for the console handoff, preventing a 'blind boot' where early kernel log output is lost before the main serial driver initializes.

If the custom board does not feature an SD card slot, remove the *&sdhci1* node as shown below. This prevents unnecessary MMC timeout and boot errors from occurring during initialization.

```
- &sdhci1 {
-      [... Properties ...]
- };
```


Copyright © 2026 Texas Instruments Incorporated

At the top of the device tree, the compatible and model property is set to "*minimal*" and "*ti*". Modify this string for the custom board as seen below.

```
- compatible = "ti,am62l3-minimal", "ti,am62l3";
- model = "Texas Instruments AM62L3 MINIMAL";
+ compatible = "<company>,am62lx-<boardname>", "ti,am62l3";
+ model = "<company> <boardname>";
```

6.5 Building U-Boot

This section explains how to compile the bootloader binaries using the custom board configuration and the AM62Lx Processor SDK.

For AM62Lx boards, three images generated by the build process are required to boot to U-Boot prompt:

- *tiboot3.bin*
- *tispl.bin*
- *u-boot.img*

The AM62Lx bootflow is changed considerably from other Sitara™ K3 SoC's bootflows, for more information refer back to Understanding the AM62Lx Bootflow.

Because the AM62Lx bootflow is carried out only on the 64-bit core, only a 64-bit cross-compiler is required for building U-Boot. The TI AM62Lx Processor SDK includes the relevant toolchain, but its respective path must be configured in the host environment before building.

Set up the toolchain path as well as the following environment variables:

Table 6-2. Environment Variables

Env Variable	Descriptions
CROSS_COMPILE_64	Cross compiler path for Armv8 (Arm 64bit). The recommended cross compiler for this project is gcc-aarch64-linux-gnu
LINUX_FW_PATH	Path to TI Linux firmware if repository is available, otherwise, pre-built binaries folder in PSDK
TFA_OUT_PATH	Path to Trusted Firmware ARM output build artifacts, (bl1.bin and bl31.bin)
OPTEE_PATH	Path to OPTEE output build artifacts, otherwise, pre-built binaries folder in PSDK: (bl32.bin)
OUTPUT_DIR	Directory path for all U-Boot build output

To build TI U-Boot, it is recommended to use the *aarch64-linux-gnu* cross-compile toolchain from Ubuntu package manager or the "*-non-linux-*" versions from ARM developer portal. After downloading said toolchains, the *$CROSS_COMPILE_64* environment variables must be properly setup accordingly. For more information, refer to the AM62Lx Processor SDK: toolchain documentation.

The example below demonstrates how to setup the host environment to build TI U-Boot efficiently utilizing pre-built binaries provided in the PSDK. If the user is compiling using the source repositories, ensure these paths point to the repository output directories instead.

```
$ export LINUX_FW_PATH=$TI_SDK/board-support/prebuilt-images/am62lxx-evm
$ export OPTEE_PATH=$TI_SDK/board-support/prebuilt-images/am62lxx-evm
$ export TFA_OUT_PATH=$TI_TFA/build/k3low/<boardname>/release
$ mkdir -p $TI_SDK/board-support/out/ && export OUTPUT_DIR=$TI_SDK/board-support/out
```

Subsequent instructions refer to the build output directory path as *$OUTPUT_DIR.*

All Make commands must be executed from the root directory of the SDK TF-A or U-Boot repository.

Building *bl1.bin* and *bl31.bin*

```
$ cd $TI_TFA
$ make clean
$ make ARCH=aarch64 CROSS_COMPILE="$CROSS_COMPILE_64" PLAT=k3low SPD=opteed TARGET_BOARD=<boardname>
```


Copyright © 2026 Texas Instruments Incorporated

Note that *<boardname>* is to be substituted for custom board name created in Section 4.

Building *tiboot3.bin*, *tispl.bin*, and *u-boot.img*

```
$ cd $TI_U_BOOT
$ make clean O=$OUTPUT_DIR
$ make ARCH=arm CROSS_COMPILE="$CROSS_COMPILE_64" <boardname>_defconfig O=$OUTPUT_DIR
$ make ARCH=arm CROSS_COMPILE="$CROSS_COMPILE_64" O=$OUTPUT_DIR BL1=$TFA_OUT_PATH/bl1.bin
BL31=$TFA_OUT_PATH/bl31.bin TEE=$OPTEE_PATH/bl32.bin BINMAN_INDIRS=$LINUX_FW_PATH
```

Use the following binaries depending on the security level of the device on the custom board.

Table 6-3. Generated Binaries Based on Device Security Type

Security	Generated Binaries
HS-FS, HS-SE	• tiboot3.bin • tispl.bin • u-boot.img

Note

The device security type of the SoC on a new custom board is typically High Security - Field Securable (HS-FS). If the exact security type of the SoC is unknown, consult the *Device type* guide to identify it. The guide is for AM62x, but the content can be applied for AM62Lx as well.

The three generated binaries can be found in the *$OUTPUT_DIR* folder.

6.6 Reaching the U-Boot Prompt

To boot the custom board using the new U-Boot configuration, transfer the three generated binaries to the target hardware through UART. This method utilizes a common, accessible peripheral to establish a U-Boot command prompt. Note that your custom board must support a UART bootmode configuration to proceed with this example. While TI EVMs utilize onboard switches to toggle between various boot methods, production boards can accomplish this through jumpers, buttons, or similar hardware mechanisms. The following steps assume the board is already configured for UART boot.

The first step is to install a serial communication utility, such as *Minicom*, on the host PC and configure it to match the serial settings of the board. The AM62Lx Quick Start Guide contains detailed instructions for setting up this serial connection. To verify that the terminal setup and hardware connections are correctly established, check for the continuous stream of '*CCC'* characters output by the ROM code while it polls for a boot image. An example of this terminal output is shown below.

```
CCCCCCCC
```

If no terminal output appears, multiple hardware or configuration variables could be causing the failure. It is highly recommended to validate your host environment using a known-good TI EVM to isolate whether the issue resides within the host setup or the custom hardware. These communication issues must be completely resolved before proceeding with the board bring-up process.

- Hardware and Board Layout: Verify all power supply rails match the required tolerances, confirm the physical bootmode selection pins or switches are securely set for UART boot, and inspect the PCB trace continuity along the UART transmit (TX) and receive (RX) lines.
- Terminal Configuration: Ensure the host serial communication utility is configured for the correct baud rate (typically 115200), 8 data bits, no parity, and 1 stop bit (8N1), with both hardware and software flow control completely disabled.

Once the serial connection is established, transmit the boot binaries over UART in the following strict order: *tiboot3.bin*, *tispl.bin*, and then *u-boot.img*. If you are using minicom or a similar terminal utility, these files can be sent using the program's integrated file transfer menu. Alternatively, you can transmit them directly from the command line using the *lrzsz* package utilities (such as *sx* and *sb)*. For successful UART boot on a AM62Lx

Copyright © 2026 Texas Instruments Incorporated

board, the *XMODEM* protocol must be used to transfer *tiboot3.bin* and *tispl.bin*, while the *YMODEM* protocol is required to transfer *u-boot.img*.

The serial console outputs an initialization log block after each binary transfer completes. If *tispl.bin* is successfully transferred, its final log signature will match the example shown below.

```
Trying to boot from UART
CCCCC
```

After transferring *u-boot.img*, U-Boot provides a brief countdown to abort the autoboot process. Press any key during this window to stop the automatic boot sequence and enter the U-Boot interactive command prompt. An example boot log demonstrating a successful drop into the U-Boot prompt is shown below.

```
02000000011a0000616d36326c0000000000000048534653000101000001010002a6000000000000fe73d96bbf79f784d9ae
e9fc5698d1093936556f93abf79f3ec7ce923079c478f837b12296da2a81a2fa024cb9d2423a14511488622a9ca591e5ee05
7175b142ad0bc40b0000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000107882838a9bf29cfa7f8fb37c103f231a2702b0a0366a86fdaf86890d9ac120
CCCCCCCCCCov 11 2026
NOTICE:  BL1: dram_class: 11
NOTICE:  BL1: dram_size: 0x8000000
NOTICE:  DDR init done
NOTICE:  ENTERING WFI - end of bl1
02000000011a0000616d36326c0000000000000048534653000101000001010002a6000000000000ad0bc40b000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000ad0bc40b0000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000107882838a9bf29cfa7f8fb37c103f231a2702b0a0366a86fdaf86890d9ac120
CCCCCCNOTICE:  SYSFW ABI: 4.0 (firmware rev 0x000c '12.2.0--v12.02.00 (Clever Cat)')

U-Boot SPL 2026.01-00529-gc48e70588df8 (Nov 11 2026 - 15:29:56 -0600)
SPL initial stack usage: 1952 bytes
Trying to boot from UART
CCCLoaded 1416403 bytes


U-Boot 2026.01-00529-gc48e70588df8 (Nov 11 2026 - 15:29:56 -0600)

SoC:   AM62LX SR1.1 HS-FS
Model: Texas Instruments AM62L3 MINIMAL
DRAM:  128 MiB
Core:  47 devices, 24 uclasses, devicetree: separate
MMC:   mmc@fa00000: 1
Loading Environment from nowhere... OK
In:    serial@2800000
Out:   serial@2800000
Err:   serial@2800000
Net:   No ethernet found.
Hit any key to stop autoboot: 0
=>
```

Successfully reaching the U-Boot command prompt confirms that the custom board's UART devicetree configuration is functional. It also validates that the initial, stable DDR configuration are sufficient to execute U-Boot proper. The next milestone is to load and boot the Linux kernel to reach a login shell. This process requires a Linux kernel image, a compiled Devicetree Binary (DTB), and an initramfs. Because this volume of data cannot be efficiently transmitted over a slow UART connection, an alternative peripheral interface must be utilized to load these boot artifacts.

7 Expanding the Devicetree

At this point, basic board functionality and the UART configuration have been validated by successfully booting to the U-Boot prompt. To load the much larger Linux kernel image, a higher-speed interface or block storage device is required. Integrating this interface is the next milestone in our iterative bring-up process.

This section uses Ethernet to demonstrate how to introduce a new peripheral into the device tree. Because Ethernet is a complex subsystem that relies on multiple interconnected resources (such as clocks, power domains, and external PHY configurations), it serves as an excellent reference model for setting up other advanced nodes on the board.

Copyright © 2026 Texas Instruments Incorporated

Note that if your custom board features an SD card, eMMC, or USB interface, starting with those simpler peripherals is recommended. In fact, a reference SD card implementation is already included in the minimal device tree template and can function immediately if the hardware supports it.

7.1 Adding New Devicetree Nodes

The first step to implementing a peripheral is to inspect its baseline definition within the SoC DTSI file. The nodes defined here contain core register and interrupt mappings but are kept disabled by default because they lack board-specific routing and they are incomplete. To enable the node, it must be overridden in the board devicetree file (*k3-am62lx-<boardname>.dts*) and updated to match the hardware layout. See the *AM62Lx Devicetree Layout* to visualize how board DTS files inherit from and extend SoC DTSI files. When enabling a node, make sure that any referenced dependencies (such as clock, power domain, pin control, and interrupt controller nodes) are also enabled. The following Ethernet implementation demonstrates this workflow in detail.

Cpsw3g, an Ethernet MAC and switch, operates within the main domain. Its foundational SoC devicetree node is defined in *k3-am62l-main.dtsi*. This node references other foundational nodes, such as *&scmi_clk*, *&main_pktdma*, and *&phy_gmii_sel*. It is necessary to ensure these dependent nodes are enabled within the board devicetree. A node is enabled by setting or overriding its *status* property, as shown below.

```
+ &cpsw3g {
+    status = "okay";
+ };
```

In addition to being enabled, every newly introduced node should have a *pinctrl-names* and a *pinctrl-0* property pointed to the corresponding pin node. After completing these steps, the *cpsw3g pinctrl-0* property should look as follows:

```
&cpsw3g {
     status = "okay";
+      pinctrl-names = "default";
+      pinctrl-0 = <&rgmii2_custom_pins_default>;
};
```

In *k3-am62l3-evm.dts*, the Ethernet PHYs are defined as child nodes of the MDIO bus node. This configuration should be added to the device tree node as shown below. The section assumes the pin configuration and the status properties have already been set for the MDIO bus node.

```
&cpsw3g_mdio {
    status = "okay";
    pinctrl-names = "default";
    pinctrl-0 = <&mdio_custom_pins_default>;
+     cpsw3g_phy1: ethernet-phy@1 {
+     };
};
```

In *k3-am62l3-evm.dts*, several properties are defined within the Ethernet PHY nodes. A grep search for any of these properties points to *$TI_U_BOOT/dts/upstream/Bindings/net/ti,dp83867.yaml*, which defines the properties for these PHY nodes. These match the properties specified in the reference EVM board device trees. The binding documentation also references a specific header file *$TI_U_BOOT/dts/upstream/include/dt-bindings/net/ti-dp83867.h* containing pre-defined macros for these properties, which correspond directly to the control registers of the Ethernet PHY. You can configure these settings by copying the EVM devicetree parameters or by referencing your custom Ethernet port design documentation for precise device specifications. Note that both these values and the method of integration in the devicetree varies if a different physical transceiver is used. A sample implementation is demonstrated below:

```
+ #include <dt-bindings/net/ti-dp83867.h>
...
&cpsw3g_mdio {
     status = "okay";
     pinctrl-names = "default";
     pinctrl-0 = <&mdio_custom_pins_default>;
          cpsw3g_phy1: ethernet-phy@1 {
+              reg = <1>;
```


Copyright © 2026 Texas Instruments Incorporated

```
+               ti,rx-internal-delay = <DP83867_RGMIIDCTL_2_00_NS>;
+               ti,fifo-depth = <DP83867_PHYCR_FIFO_DEPTH_4_B_NIB>;
+               ti,min-output-impedance;
+       };
};
```

The EVM device tree defines a *phy-handle* property inside the *&cpsw_port2* node, establishing a reference from the MAC port to the physical PHY node. This property can be configured as shown below after enabling the port node.

```
+ &cpsw_port2 {
+     status = "okay";
+     phy-handle = <&cpsw3g_phy1>;
+ };
+
+ &cpsw_port1 {
+     status = "disabled";
+ };
```

Note that *cpsw_port1* or *cpsw_port2* nodes may not be disabled by default and might already be available when *cpsw3g* is enabled. Setting status = "okay" is still good practice, as it makes the intent explicit. If a port is not used on your board, set status = "disabled" on it so the driver does not try to bring it up.

7.2 Revising Devicetree Nodes

Suppose U-Boot has been built with no errors, and the resulting binaries are boot-tested after updating the Ethernet configuration. After transferring *u-boot.img* over UART, the U-Boot console output displays as shown below when any key is pressed to abort autoboot:

```
02000000011a0000616d36326c0000000000000048534653000101000001010002a6000000000000fe73d96bbf79f784d9ae
e9fc5698d1093936556f93abf79f3ec7ce923079c478f837b12296da2a81a2fa024cb9d2423a14511488622a9ca591e5ee05
7175b142ad0bc40b0000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000107882838a9bf29cfa7f8fb37c103f231a2702b0a0366a86fdaf86890d9ac120
Cov 12 2026
NOTICE:  BL1: dram_class: 11
NOTICE:  BL1: dram_size: 0x8000000
NOTICE:  DDR init done
NOTICE:  ENTERING WFI - end of bl1
02000000011a0000616d36326c0000000000000048534653000101000001010002a6000000000000ad0bc40b000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000ad0bc40b0000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000107882838a9bf29cfa7f8fb37c103f231a2702b0a0366a86fdaf86890d9ac120
CCCCNOTICE:  SYSFW ABI: 4.0 (firmware rev 0x000c '12.2.0--v12.02.00 (Clever Cat)')

U-Boot SPL 2026.01-00530-g70f7282f6c91 (Nov 12 2026 - 11:16:15 -0600)
SPL initial stack usage: 1952 bytes
Trying to boot from UART
CCCCCLoaded 1417227 bytes


U-Boot 2026.01-00530-g70f7282f6c91 (Nov 12 2026 - 11:16:15 -0600)

SoC:   AM62LX SR1.1 HS-FS
Model: Texas Instruments AM62L3 MINIMAL
DRAM:  128 MiB
Core:  53 devices, 27 uclasses, devicetree: separate
MMC:   mmc@fa00000: 1
Loading Environment from nowhere... OK
In:    serial@2800000
Out:   serial@2800000
Err:   serial@2800000
Net:   am65_cpsw_nuss_port ethernet@8000000port@2: Invalid PHY mode, port 2
No ethernet found.

Hit any key to stop autoboot: 0
=>
```

The *Net* field indicates the runtime status of the Ethernet configuration. In this scenario, the error log reveals that the Ethernet Port 2 PHY is operating in an invalid mode. This behavior typically points to a devicetree configuration issue, indicating that the PHY is configured in the incorrect or unsupported mode.

Copyright © 2026 Texas Instruments Incorporated

To resolve this error, inspect both the PHY and Ethernet MAC port nodes as potential root causes. In the reference EVM devicetree, the port node contains a phy-mode property set to "*rgmii-rxid*". Searching for this value points us to *$TI_U_BOOT/dts/upstream/Bindings/net/ethernet-controller.yaml* file, which specifies that phy-mode defines the interface between the PHY and Ethernet port. The binding states that "*rgmii-rxid*" must be used if the PHY provides an internal RX delay. Since an RX delay was already configured as required by the PHY bindings in an earlier step, this property applies and must be added to the port node.

```
&cpsw_port2 {
      phy-handle = <&cpsw3g_phy1>;
+     phy-mode = "rgmii-rxid";
 };
```

This troubleshooting loop highlights the iterative workflow required to develop a fully functional board devicetree. Once the fix is applied, rebuilding U-Boot and booting over UART should confirm that the Ethernet port is active and available. This high speed interface can now be utilized to efficiently transfer the larger images and boot the Linux kernel.

7.3 Devicetree Development Tips

This section provides diagnostic strategies for troubleshooting common integration issues, specifically when newly added devicetree nodes fail to enumerate, or when a peripheral successfully enumerates but fails to function within Linux userspace.

Pin configuration:

As a best practice, explicitly define all pin configurations in the device tree, even if a pin's default "power-on-reset" state satisfies the peripheral's requirements. This approach prevents from accidentally reusing the same pin for a different peripheral later, avoiding hard-to-trace pin contention conflicts.

The pinctrl-names property defines the available runtime states for a peripheral's pins, such as "*default*" or "*sleep*". Corresponding properties like *pinctrl-0* and *pinctrl-1* map the actual pin configuration handles to each respective state index. For example, if "*default*" is specified at index 0 within *pinctrl-names*, the kernel applies the configurations listed in *pinctrl-0* during device initialization. For a complete reference on pin configuration syntax, consult the schema file at *$TI_U_BOOT/dts/upstream/Bindings/pinctrl/pinctrl-bindings.txt*.

For further guidance on example pin configurations to include for a specific peripheral, reference the pinmux nodes defined within the EVM files, such as *k3-am62l3-evm.dts*.

Decompiling:

A highly effective diagnostic technique is to decompile the DTB generated during the build process. This operation produces a single, human-readable file that consolidates all properties and nodes across every DTSI file included during compilation, making it easy to audit the final peripheral configurations. For the 64-bit architecture, the binary is located at *$OUTPUT_DIR/dts/upstream/src/arm64/ti/k3-am62lx-<boardname>.dtb*. To decompile this file back into source format, execute the command below:

```
$ dtc -I dtb -O dts k3-am62lx-<boardname>.dtb -o k3-am62lx-<boardname>-reversed.dts
```

The decompiled file is output as *k3-am62lx-<boardname>-reversed.dts*. This flat source file can be audited alongside the kernel binding schemas and working reference design files to verify complete and correct hardware definitions.

Another effective strategy is to directly compare the custom board's decompiled output against the decompiled output of a known-working reference board, such as the *k3-am62l3-evm.dtb*. In general, the EVM's device tree nodes serve as an excellent reference configuration. If the custom hardware utilizes a peripheral in a similar manner to the EVM, only minor modifications, if any, may be required.

References:

A highly effective step during custom board porting is to search across the reference EVM devicetree files (*k3-am62l3-evm.dts*) for all properties applied on top of the baseline SoC definition of a node. Because these

Copyright © 2026 Texas Instruments Incorporated

properties make up the essential board-level definitions for the peripheral to work on AM62Lx EVM, they typically must be configured within the custom devicetree file as well.

Devicetree bindings define exactly how hardware must be described in devicetree files. Each binding specifies a mandatory compatible string, required properties, optional properties, and valid value ranges. U-Boot developers should construct custom board DTS files following the Linux kernel binding specifications, which are duplicated and stored at *$TI_U_BOOT/dts/upstream/Bindings/*. For practical guidance during construction, consult the example nodes included within the binding files, or review existing devicetree files that share the same compatible strings. You can locate relevant binding schemas by running a grep search for specific compatible strings or unique property names. For a comprehensive, structured course on this topic, see the Bootlin Devicetree Training materials cited in Section 12.

Copyright © 2026 Texas Instruments Incorporated

8 Booting the Linux Kernel

8.1 Boot Method Overview

The Linux kernel boot sequence outlined in the next section utilizes prebuilt components provided in the SDK to streamline the bring-up workflow. When testing a new hardware configuration, the devicetree binary is the only component that needs to be rebuilt; the base kernel image and rootfs can be reused completely. This decoupling highlights the fundamental value of using a device tree for hardware description, as the exact same Linux kernel Image and rootfs can be deployed across completely different boards without requiring driver changes. This holds true as long as all onboard components are already supported by default in Linux kernel. Adding custom driver support or enabling a custom driver for an unsupported on-board chip is outside the scope of this guide.

The U-Boot board device tree configured in the previous section must be reused as the Linux kernel device tree. Maintaining absolute synchronization between the U-Boot and Linux kernel device trees is critical throughout the board porting process. Instructions in the subsequent section will reinforce this requirement and outline the integration steps.

The initramfs (initial RAM filesystem) is utilized in this board bring-up process in lieu of the rootfs provided in TI default images. Since it runs from RAM, the initramfs prevents rootfs complexity from masking hardware and driver failures. This allows isolated verification of hardware interfaces, kernel modules, and storage drivers. Once verified, a full production filesystem can be deployed to non-volatile storage like eMMC or NFS.

8.2 Reaching the Linux Command Prompt

This section covers loading the Linux kernel Image, device tree, and filesystem once a high speed peripheral capable of loading these components into RAM has been configured. The method outlined below utilizes the Linux kernel Image and initramfs components provided by the TI AM62Lx Processor SDK. The table below outlines exactly where these prebuilt images are located within the SDK installation directory.

Table 8-1. SDK Paths to Linux Kernel Components

Component	Path
Kernel Image	$TI_SDK/board-support/prebuilt-images/am62lxx-evm/Image
Initramfs	$TI_SDK/filesystem/am62lxx-evm/arago-tiny-initramfs-am62lxx-evm.cpio

The Linux kernel device tree must be compiled before proceeding. Although this device tree is identical to the one used for U-Boot up to this point, it must be compiled independently within the Linux kernel source repository.

Follow the commands below to copy the latest board devicetree file from the U-Boot repository to the Linux kernel repository and compile the DTS.

```
$ cp $TI_U_BOOT/dts/upstream/src/arm64/ti/k3-am62lx-<boardname>.dts $TI_LINUX/arch/arm64/
boot/dts/ti/

$ cd $TI_LINUX
$ make ARCH=arm64 CROSS_COMPILE="$CROSS_COMPILE_64" distclean
$ make ARCH=arm64 CROSS_COMPILE="$CROSS_COMPILE_64" defconfig ti_arm64_prune.config
$ make DTC_FLAGS=-@ ARCH=arm64 CROSS_COMPILE="$CROSS_COMPILE_64" arch/arm64/boot/dts/ti/k3-am62lx-
<boardname>.dtb
```

The compiled DTB can be found in *$TI_LINUX/arch/arm64/boot/dts/ti/k3-am62lx-<boardname>.dtb.*

The next major milestone is to reach the Linux command prompt using the kernel image, compiled devicetree binary, and initramfs. This can be accomplished either via SD card boot, which should already be functional assuming your pin configurations were verified and updated, or over the network using TFTP and an NFS mount. Because configuring an SD card interface is generally simpler, the subsequent section provides step-by-step instructions for reaching the Linux command prompt using SD card boot.

Copyright © 2026 Texas Instruments Incorporated

To use an SD card for loading kernel components, prepare a FAT32-formatted SD card partition (or raw device) using the *Disks* application or by running the following command in a Linux terminal:

```
$ sudo mkfs.vfat -F 32 -n "boot" /dev/<sd_card_device_or_partition>
```

Copy the following files to this disk partition:

- Compiled Linux kernel DTB located at *$TI_LINUX/arch/arm64/boot/dts/ti/k3-am62lx-<boardname>.dtb*
- Kernel Image located at *$TI_SDK/board-support/prebuilt-images/am62lxx-evm/Image*
- Initramfs located at *$TI_SDK/filesystem/am62lxx-evm/arago-tiny-initramfs-am62lxx-evm.cpio*

Boot the board using the UART bootmode to reach the U-Boot prompt and halt autoboot. Next, use the appropriate U-Boot load command to transfer the kernel components from the storage media into RAM. Refer to the official U-Boot Documentation to determine the exact command required for your specific peripheral interface. The example below demonstrates how to load these kernel components from the SD card into RAM. For additional information or examples on how to load binaries from the SD card, refer to the AM62Lx Processor SDK: U-Boot MMC documentation.

```
/* using MMC device 1 (SD) */
=> fatload mmc 1 $loadaddr Image
=> fatload mmc 1 $fdtaddr k3-am62lx-<boardname>.dtb
=> fatload mmc 1 $rdaddr arago-tiny-initramfs-am62lxx-evm.cpio
```

Note

If the *loadaddr*, *fdt_addr_r*, and *ramdisk_addr_r* (or *rdaddr*), U-Boot environment variables are not set at this stage, U-Boot was not properly configured for the custom hardware. Return to the U-Boot Build System Integration section to correct the environment settings for the custom board.

If the image fails to load, the issue likely stems from the MMC controller or the SD card configuration. Verify the pin configuration and, if necessary, enable MMC core debugging in U-Boot to diagnose SD card enumeration failures.

Now that the required kernel components are loaded into RAM, the next step is to establish the kernel boot configuration. To accomplish this, configure the *bootargs* environment variable, which U-Boot uses to pass critical command-line parameters to the Linux kernel during initialization. The following U-Boot command sets this configuration:

```
=> setenv bootargs rdinit=/bin/sh
```

The board is now prepared to boot the Linux kernel using the binaries loaded into RAM. The following command executes the boot process using these memory-cached components:

```
=> booti $loadaddr $rdaddr:0x$filesize $fdtaddr
```

If successful, the kernel will stream initialization logs through the serial console, and the device drops into a Linux command shell.

To have access to the system and hardware information, mount the following pseudo-filesystems using the following commands:

```
# mount -t proc none /proc
# mount -t sysfs none /sys
# mount -t devtmpfs none /dev
```

If you did not reach the Linux command prompt, there are multiple methods to approach this issue. For information on how to attempt to reconfigure a peripheral correctly to successfully boot the kernel, see Section 7.1. For help in debugging, see Section 9.

Reaching the Linux command prompt provides access to essential runtime diagnostic utilities that are invaluable for developing the device tree and debugging hardware issues. It is critical to maintain this same iterative

Copyright © 2026 Texas Instruments Incorporated

approach, enabling peripherals one at a time and committing verified changes to the repository once a working configuration is established. The configuration methodologies outlined in Section 7.1, combined with the runtime utilities detailed in the next section, will guide you through initializing the remaining peripherals for the custom board.

Common next steps to finalize the board bring-up process include migrating to a complete production filesystem, enabling high-speed data interfaces, and optimizing the kernel Image. For more details, consult the AM62Lx Processor SDK documentation.

Copyright © 2026 Texas Instruments Incorporated

9 Tools and Debugging

During custom board bring-up, engineers frequently encounter complex, difficult-to-debug hardware and software issues. This section outlines additional diagnostic strategies to help isolate and resolve these issues.

9.1 Debug Traces

Early in the board bring-up process, diagnosing a catastrophic failure that occurs before the Linux kernel initializes the serial console is highly challenging. Furthermore, raw kernel boot logs can be difficult to interpret without context. Compiling debugging features into the kernel image through the menuconfig tool can significantly help isolate the root cause of early boot failures.

Enabling additional debug features requires building a custom kernel Image. These steps come from the *AM62Lx Processor SDK: Linux Users Guide* and the commands must be executed from the root of the Linux kernel repository.

1. Reset the kernel source tree to a clean state.

```
$ make ARCH=arm64 CROSS_COMPILE="$CROSS_COMPILE_64" distclean
```

2. Open the interactive menuconfig tool.

   The next command loads the default ARM64 kernel configuration, applies the *ti_arm64_prune.config* fragment for cleanup, and opens the interactive menuconfig tool to manually configure additional kernel options.

```
$ make ARCH=arm64 CROSS_COMPILE="$CROSS_COMPILE_64" defconfig ti_arm64_prune.config menuconfig
```

3. Enable debug features.

   Navigate to the "*Kernel hacking*" menu to access the available debugging options. More information of these debug configuration options can be found at *$TI_LINUX/lib/Kconfig.debug*. After enabling the debug features, exit the interactive menuconfig tool, save the configuration, and build the kernel image using the following commands:

```
$ make ARCH=arm64 CROSS_COMPILE="$CROSS_COMPILE_64" savedefconfig
$ make ARCH=arm64 CROSS_COMPILE="$CROSS_COMPILE_64"
```

The resulting kernel binary is located at *$TI_LINUX/arch/arm64/boot/Image*. Deploy this debug-enabled kernel image during the board porting process to help diagnose and resolve issues with newly enabled peripherals.

9.2 OpenOCD

It is recommended to debug the most complex issues using JTAG debugger. OpenOCD is an open source debugger that interacts with a variety of gdb-based debug IDEs. A guide to using OpenOCD is available in the U-Boot documentation under K3 Generation.

Copyright © 2026 Texas Instruments Incorporated

10 Future Work

This guide, built around the minimal platform support framework in U-Boot and Linux, was created to introduce a structured process that aids in custom board-porting efforts. Looking forward, several critical advanced sections can be appended to this documentation, including but not limited to:

1. Cross-Platform Porting Guides: Expanding this methodology to cover board bring-up on alternative TI K3 processors, such as the AM62A, AM64x, AM62P, and AM67A SoCs.
2. Advanced Troubleshooting Diagnostics: Adding a wider variety of specialized debugging utilities, trace scripts, and hardware inspection procedures to the "*Tools and Debugging*" section.
3. Practical OpenOCD Integration: Expanding the "*OpenOCD*" chapter to include a step-by-step example of using a JTAG hardware debugger resolve low-level hardware integration issues.

11 Summary

The minimal platform support framework in U-Boot and Linux accelerates TI AM62Lx board bring-up from weeks to hours, with this application note providing a structured, step-by-step roadmap for custom board-porting efforts. By utilizing this modular approach (which covers iterative testing, debugging, and documentation) developers can efficiently validate hardware and apply these scalable methodologies to other TI K3 SoCs and ARM-based processors.

12 References

1. Texas Instruments, AM62Lx Technical Reference Manual, technical reference manual
2. Texas Instruments, AM62Lx Sitara™ Processors, datasheet
3. Texas Instruments, AM62L Evaluation Module, evaluation module user's guide
4. Texas Instruments, SysConfig tool, webpage
5. Texas Instruments, TI Linux kernel, repository
6. TI Linux, k3-am62l3-minimal.dts, DT file
7. Texas Instruments, TI U-Boot, repository
8. TI U-Boot, k3-am62l3-minimal-u-boot.dtsi, DT file
9. Texas Instruments, Processor SDK Linux 12_01_00_05_03 for AM62Lx
10. Texas Instruments, Device Types Guide, webpage
11. Texas Instruments, AM62Lx EVM Quick Start Guide
12. Texas Instruments, U-boot build dependencies
13. Texas Instruments, Linux build dependencies
14. Texas Instruments, Linux SDK build dependencies
15. OpenOCD, OpenOCD User's Guide
16. U-Boot, Documentation, webpage
17. U-Boot Documentation, K3 Generation, webpage
18. Bootlin, Device Tree for Dummies, PDF

Copyright © 2026 Texas Instruments Incorporated

Revision History

Changes from Revision * (September 2026) to Revision A (November 2026)	Page
• Updated *Introduction* section	3
• Updated *U-Boot Build System Integration* section	5
• Updated Figure 6-1 in *Introduction to Devicetrees* section	7
• Updated column header in Table 6-1 from "AM62x SK" to "AM62Lx EVM" in *Introduction to Devicetrees* section	7
• Updated *Adding Custom Board Devicetrees* section	8
• Fixed typos: "AM62X_IOPAD" to "AM62LX_IOPAD" in *Initial Devicetree Modifications* section	10
• Updated *Adding Custom Board Devicetrees* section to make pinmux reference label names consistent	10
• Fixed typos: "TI AM62x Processor SDK" to "TI AM62Lx Processor SDK" in *Building U-Boot* section	12
• Updated *Reaching the U-Boot Prompt* section	13
• Added note in *Expanding the Devicetree* section	14
• Fixed pinmux node label names in *Adding New Devicetree Nodes* section	15
• Updated AM62x example log to AM62Lx example log in *Revising Devicetree Nodes* section	16
• Fixed typo: "k3-am62x-<boardname>-reversed.dts" to "k3-am62lx-<boardname>-reversed.dts" in *Devicetree Development Tips* section	17
• Fixed initramfs name in *Reaching the Linux Command Prompt* section	19
• Updated *Debug Traces* section	22
• Fixed spelling of "Reference" in *References* section	23
• Added U-boot build dependencies, Linux build dependencies, and Linus SDK build dependencies to *References* section	23

Copyright © 2026 Texas Instruments Incorporated

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you fully indemnify TI and its representatives against any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale, TI’s General Quality Guidelines, or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products. Unless TI explicitly designates a product as custom or customer-specified, TI products are standard, catalog, general purpose devices.

TI objects to and rejects any additional or different terms you may propose.

Copyright © 2026, Texas Instruments Incorporated

Last updated 10/2025