

Application Note

Linux Board Porting on AM62x Devices



Logan Bristol

ABSTRACT

The portfolio of AM62x System-on-Chips (SoCs) is designed for building cost optimized Linux based embedded systems. With a diverse set of peripherals and configurations, the AM62x is suitable for many market applications, both in automotive and industrial. This family offers many 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 AM62x SoCs by providing a clear starting point for Linux development. For more information about the AM62x products, see the device-specific data sheet, user guides and SDK documentation.

Table of Contents

1 Introduction	2
2 Installing the SDK	2
3 Configuring the SDK for a Custom Board	2
4 Starting U-Boot Board Port	4
4.1 Introduction to Devicetrees.....	4
4.2 Capabilities of the Minimal Configuration.....	5
4.3 Preparing Custom Board Files.....	6
4.4 Initial Devicetree Modifications.....	8
4.5 Building U-Boot Binaries.....	9
4.6 U-Boot Deployment Instructions.....	11
5 Expanding the Custom Board Devicetree	12
5.1 Devicetree Configuration.....	12
5.2 Describing Peripherals in Nodes.....	13
5.3 Revising the Devicetree Configuration.....	14
6 Booting the Linux Kernel	15
6.1 Kernel Boot Overview.....	15
6.2 Kernel Deployment Instructions.....	15
7 Tools and Debugging	17
7.1 Kernel Debug Traces.....	17
7.2 OpenOCD Debugging.....	18
8 Future Work	18
9 Summary	18
10 References	18
Revision History	19

List of Tables

Table 4-1. Generic Environment Variables.....	10
Table 4-2. Board-Specific Environment Variables.....	10
Table 4-3. Generated Binaries Based on Device Security Level.....	11
Table 6-1. SDK Paths to Linux Kernel Components.....	15

Trademarks

Arm® is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All trademarks are the property of their respective owners.

1 Introduction

TI provides a Software Development Kit (SDK) to accelerate development on the AM62x SoC. This software package is tuned for evaluation modules (EVMs), like Starter Kits (SKs). The SDK and SKs are designed for quick evaluation of the SoC's many capabilities for a variety of use cases. All features and capabilities, such as a variety of boot modes, are enabled and optimized. While this flexibility is great for an evaluation platform, this added complexity may not be needed on custom boards designed for specific use cases. In such cases, for example, only a couple of these boot modes would be used in an end product.

Removing complexity from the complete configurations that are provided in the SDK is not efficient and can extend custom board bring up time from days to weeks. This process can involve arbitrarily disabling features to determine how to configure a simpler use case. The interconnected nature of the system works against developers to discern where problems start. Instead of peeling back the layers of a complex system to find the problem, the recommended approach is to start with a minimal configuration as a strong foundation and iterate toward a complete and optimized configuration. Incrementally adding one feature at a time quickly identifies what aspects of the board are functional and which ones are not. Non-functional areas can be targeted for debugging.

This application note details adding custom board support using the SDK. The SDK contains source code repositories for the Linux kernel and U-Boot, which is used as the bootloader. To enable a new custom board, both Linux and U-Boot need to be ported to the custom board. First, the U-Boot environment is configured for the new board. Next, the U-Boot board port begins by creating the required custom board files and making initial modifications to the board's new devicetree, a software structure that describes the underlying hardware. Finally, individual features and peripherals are incrementally added to the devicetree until it is complete and all board features are functional as required by U-Boot to boot the board. The same devicetree is used to enable the Linux kernel on the new board to complete the porting process. To navigate any errors that arise during the bring-up process, this document also includes guidance on debugging.

It is recommended to go through the process detailed in this document using a TI EVM prior to receiving the custom board. This makes for an efficient bring up process once the custom board arrives, and also validates the steps in this document on a board that is known to be functional.

Software file download links can be found in [Section 10](#). While this guide and the provided files were developed for the AM62x family, it can be expanded to other TI SoCs as described further in [Section 8](#).

2 Installing the SDK

The first step of board bring up is to download the AM62x Linux Processor SDK if it is not already installed on the host machine. Download instructions are found in the [AM62x Processor SDK Guide](#).

This SDK contains source code repositories, pre-built binaries, file system images, and other useful development tools. The following instructions are based off of the 10.x version of the AM62x Processor Linux SDK. Exact paths can change across SDK releases, and it may be necessary to search for directories to determine where components are located. This document refers to the SDK directory path as `TI_SDK`.

In the 10.x version of the SDK, the U-Boot repository is found under `TI_SDK/board-support/ti-u-boot-2024.04+git`. This document refers to this path as `TI_U_BOOT`.

In the 10.x version of the SDK, the Linux kernel repository is found under `TI_SDK/board-support/ti-linux-kernel-6.6.xx+git-ti` (where *xx* represents the minor release number that will change). This document refers to this path as `TI_LINUX`.

3 Configuring the SDK for a Custom Board

This section serves as a step-by-step guide on replicating the TI U-Boot baseline for a custom board. While these steps are not absolutely necessary in order to begin development, it is good practice for every board to use its own board-specific files to prevent any conflicts or board specific configuration issues. Creating new board-specific files also preserves the files provided with the SDK for TI EVMs that can be used as references.

These instructions refer to the custom board's name as `<boardname>` and the name of the organization as `<company>`. The organization name is used for naming directories to help keep the files for one's boards together.

- Copy and paste the existing EVM A53 and R5 Kconfig targets in `TI_U_BOOT/arch/arm/mach-k3/am62x/Kconfig` and rename the symbols for the custom board. Optionally, modify the corresponding board specific string. An example is below.

```
+ config TARGET_AM625_A53_<BOARDNAME>
+   bool "<COMPANY> K3 based AM625 <BOARDNAME> running on A53"
+   select ARM64
+   select BINMAN
+   select OF_SYSTEM_SETUP

+ config TARGET_AM625_R5_<BOARDNAME>
+   bool "<COMPANY> K3 based AM625 <BOARDNAME> running on R5"
+   select CPU_V7R
+   select SYS_THUMB_BUILD
+   select K3_LOAD_SYSPW
+   select RAM
+   select SPL_RAM
+   select K3_DDRSS
+   select BINMAN
+   imply SYS_K3_SPL_ATF
```

Add the following line to the bottom of the same Kconfig file.

```
+ source "board/<company>/<boardname>/am62x/kconfig"
```

- Make new `<company>` and `<boardname>` directories to store files that will be copied and modified.

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

- Copy and rename board files from the TI directory to the new directories.

```
$ cp TI_U_BOOT/board/ti/am62x/* TI_U_BOOT/board/<company>/<boardname>/
$ cp TI_U_BOOT/board/ti/common/* TI_U_BOOT/board/<company>/common/
```

- In the `TI_U_BOOT/board/<company>/<boardname>` directory, edit the Kconfig by modifying the default value of the configuration options as seen below.

```
if TARGET_AM625_A53_<BOARDNAME>
config SYS_BOARD
    default "<boardname>"
config SYS_VENDOR
    default "<company>"
config SYS_CONFIG_NAME
    default "<boardname>_evm"
source "board/<company>/common/kconfig"
endif

if TARGET_AM625_R5_<BOARDNAME>
config SYS_BOARD
    default "<boardname>"
config SYS_VENDOR
    default "<company>"
config SYS_CONFIG_NAME
    default "<boardname>_evm"
config SPL_LDSCRIPT
    default "arch/arm/mach-omap2/u-boot-spl.lds"
source "board/<company>/common/kconfig"
endif
```

- In the same directory, rename the file `evm.c` to `<boardname>.c` and the file `am62x.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>/am62x.env TI_U_BOOT/board/<company>/<boardname>/
<boardname>.env
```

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

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

- Copy the EVM board header file and rename the file for the custom board.

```
$ cp TI_U_BOOT/include/configs/am62x-evm.h TI_U_BOOT/include/configs/am62x-<boardname>.h
```

Make the following modification inside the newly created header file.

```
- #ifndef __CONFIG_AM625_EVM_H
- #define __CONFIG_AM625_EVM_H
+ #ifndef __CONFIG_AM625-<BOARDNAME>_H
+ #define __CONFIG_AM625-<BOARDNAME>_H
```

- Create the following configuration fragments in *TI_U_BOOT/configs/* and add the lines below to set the Kconfig targets.

am62x-<boardname>_r5.config

```
CONFIG_TARGET_AM625_R5-<BOARDNAME>=y
# CONFIG_TARGET_AM625_R5_EVM is not set
```

am62x-<boardname>_a53.config

```
CONFIG_TARGET_AM625_A53-<BOARDNAME>=y
# CONFIG_TARGET_AM625_A53_EVM is not set
```

You have now replicated TI's U-Boot baseline for the custom board.

4 Starting U-Boot Board Port

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 levels of software, in this case U-Boot. For example, this file contains the board specific information to enable a UART port to use as a console output. A basic understanding of the overall devicetree specification will be helpful to complete this file and expedite the porting process.

4.1 Introduction to Devicetrees

A devicetree contains nodes that represent peripherals. Each node contains properties that hold data that is used by U-Boot and Linux kernel device drivers to initialize peripherals. Without a devicetree, the hardware configuration and parameters would need to be hardcoded into the device drivers. Keeping the hardware configuration separate from the board initialization code allows for a simpler development and maintenance process.

Note

It is recommended to have a general understanding of devicetrees prior to beginning development. A good resource created by Bootlin is found in [Section 10](#).

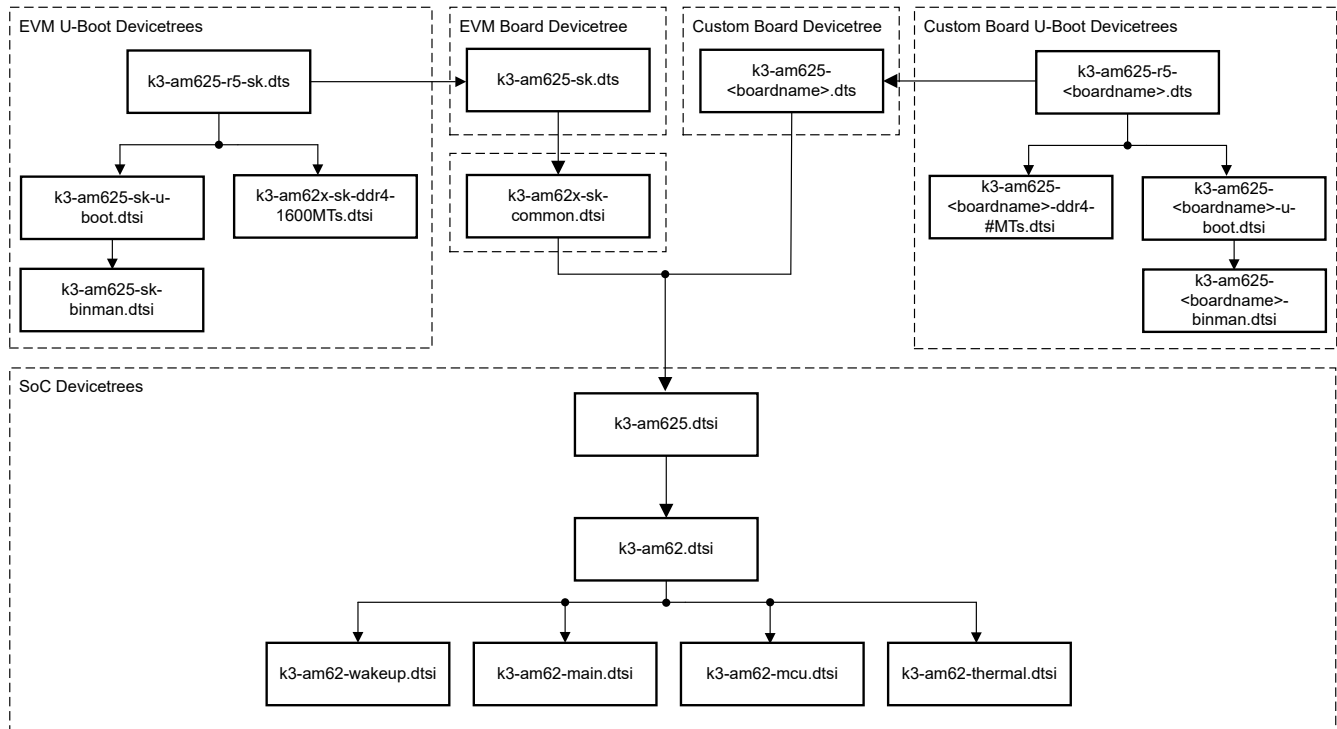


Figure 4-1. AM62x Devicetree Structure

It is important to understand the AM62x devicetree structure and the layers of the structure you will create. Files with names containing *<boardname>* are files that become the custom board's devicetrees.

The base of the tree is formed by *devicetree include* (DTSI) files. These are used to define SoC specific hardware that is generally the same across all board variations of the same SoC, including custom boards. Directly editing these files risks corrupting other devicetrees that include these files. For this reason, these should not be directly modified. Rather, devicetree nodes in SoC devicetree files are modified by referencing the node in the board devicetree.

The file *k3-am62x-sk-common.dtsi* contains hardware descriptions that are common between board variants of the AM62x SoC. This DTSI is included by *k3-am625-sk.dts* and *k3-am62-lp-sk.dts*. The files *k3-am625-sk.dts* and *k3-am62x-sk-common.dtsi* are applied on top of the SoC DTSI files. The *k3-am62x-sk-common.dtsi* is not used with a custom board devicetree.

At the top of the tree, there are *devicetree source* (DTS) files. These are used to describe board specific hardware and to modify DTSI nodes without altering them directly. This is what will be created for the custom board. Additionally, you will create other U-Boot specific devicetrees that are needed to build the bootloader binaries. This process is covered in [Section 4.3](#).

4.2 Capabilities of the Minimal Configuration

The minimal configuration consists of the minimal devicetrees and the configuration files that make these devicetrees accessible to U-Boot and the Linux kernel. This minimal configuration can be used to test initial board functionality with a reduced set of peripherals before beginning to expand the devicetrees to completely support a custom board. For simplicity, the only peripherals initially supported by the minimal configuration are the SD card reader 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.

The minimal configuration slows down and disables advanced features of the SD card reader. The SD card reader is forced to run at 25MHz Legacy SDR using only 3.3V signals to support the most common class of SD cards. This support reduces the number of issues related to high-speed or low voltage capabilities that can often result in a failure when attempting to boot to the Linux kernel if the board does not correctly support these advanced capabilities. DDR memory is also configured at a lower speed for similar reasons. It is common for new boards to have intermittent issues running at optimal speeds, so de-tuning these interfaces can help isolate problems and accelerate progress.

Using this minimal configuration, functionality of the custom board can be verified and incrementally expanded to include the full range of peripherals and optimized features. This guide was developed for the AM62x, however, it can be expanded to other TI SoCs as described in [Section 8](#).

4.3 Preparing Custom Board Files

The following steps set up the custom board devicetrees and configuration files for development. It may be helpful to refer back to [the AM62x Devicetree diagram](#) to understand how the devicetrees being created fit into the existing structure.

1. Duplicate and Rename Required Board Devicetrees.

Create renamed copies of the provided devicetrees using the following commands. If the provided devicetrees are not present in the installed SDK, they can be downloaded in the [references section](#).

```
$ cp TI_U_BOOT/arch/arm/dts/k3-am625-minimal.dts TI_U_BOOT/arch/arm/dts/k3-am625-<boardname>.dts
$ cp TI_U_BOOT/arch/arm/dts/k3-am625-r5-minimal.dts TI_U_BOOT/arch/arm/dts/k3-am625-r5-
<boardname>.dts
$ cp TI_U_BOOT/arch/arm/dts/k3-am625-minimal-u-boot.dtsi TI_U_BOOT/arch/arm/dts/k3-am625-
<boardname>-u-boot.dtsi
```

These naming changes need to be reflected in the `#include` preprocessor directives in `k3-am625-r5-<boardname>.dts`.

```
// in k3-am625-r5-<boardname>.dts
- #include "k3-am625-minimal.dts"
- #include "k3-am625-minimal-u-boot.dtsi"
+ #include "k3-am625-<boardname>.dts"
+ #include "k3-am625-<boardname>.dtsi"
```

2. Generate Devicetree for DDR Configuration.

The DDR configuration used by U-Boot for DDR initialization is contained in a devicetree 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 "AM62x". Start the program to load the DDR configuration tool. Here, you can modify DDR settings to match the hardware specification. To prevent signal integrity issues that can lead to boot failure, it is recommended to set the DDR frequency to 667MHz (1334 MTs). This is the slowest speed supported by DDR4 with DLL on. DDR frequency is set in the SysConfig DDR tool's "Memory Frequency (MHz)" field. After completing board bring-up, a new DDR configuration should be generated according to the DDR device specifications. 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-am62x-ddr-config.dtsi`. Download this file and use the commands below to rename this devicetree and place it in the SDK U-Boot repository.

```
$ cp k3-am62x-ddr-config.dtsi TI_U_BOOT/arch/arm/dts/k3-am62x-<boardname>-ddr4-<#MTs>.dtsi
```

Replace the following line in `k3-am625-r5-<boardname>.dts`.

```
- #include "k3-am62x-sk-ddr4-1600MTs.dtsi"
+ #include "k3-am62x-<boardname>-ddr4-<#MTs>.dtsi"
```

3. Setup Binman for Custom Board.

Binman is used to generate U-Boot binaries. 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-am625-sk-binman.dtsi TI_U_BOOT/arch/arm/dts/k3-am625-<boardname>-binman.dtsi
```

In this newly created file, make the following modification to use the custom devicetree.

```
- #define SPL_AM625_SK_DTB "spl/dts/k3-am625-sk.dtb"
+ #define SPL_AM625_SK_DTB "spl/dts/k3-am625-<boardname>.dtb"
```

Include this new file in *k3-am625-<boardname>-u-boot.dtsi*.

```
- #include "k3-am625-sk-binman.dtsi"
+ #include "k3-am625-<boardname>-binman.dtsi"
```

4. Integrate Devicetrees to Build Process.

Newly created devicetrees are added to the *Makefile* by adding the following lines in *TI_U_BOOT/arch/arm/dts/Makefile*.

```
dtb-$(CONFIG_SOC_K3_AM625) += k3-am625-sk.dtb \
    k3-am625-r5-sk.dtb \
    k3-am625ip-r5-sk.dtb \
    k3-am625-beagleplay.dtb \
    k3-am625-r5-beagleplay.dtb \
    k3-am625-verdin-wifi-dev.dtb \
    k3-am625-verdin-r5.dtb \
    k3-am625-phyboard-lyra-rdk.dtb \
    k3-am625-r5-phycore-som-2gb.dtb \
    k3-am62-1p-sk.dtb \
    k3-am62-r5-1p-sk.dtb \
+    k3-am625-<boardname>.dtb \
+    k3-am625-r5-<boardname>.dtb
```

5. Setup Configuration Fragments.

Create or modify two configuration fragments that will be applied on top of the existing AM62x EVM default configuration files to change the devicetree being accessed by U-Boot. These should be created in *TI_U_BOOT/configs/*. If you followed the steps in [Section 3](#), these files have already been created. Add the lines below to the configuration fragments.

am62x_<boardname>_r5.config

```
CONFIG_DEFAULT_DEVICE_TREE="k3-am625-r5-<boardname>"
```

am62x_<boardname>_a53.config

```
CONFIG_DEFAULT_DEVICE_TREE="k3-am625-<boardname>"
CONFIG_SPL_OF_LIST="k3-am625-<boardname>"
CONFIG_OF_LIST="k3-am625-<boardname>"
```

The custom board configuration has now been set up within U-Boot.

4.4 Initial Devicetree Modifications

The custom board devicetrees provided in the U-Boot repository will need to be modified to function on a custom board. This section details changes that need to be made prior to attempting to boot the board.

The current pin configurations in the provided custom board devicetree are generated using [TI's SysConfig Pinmux Tool](#) for a TI EVM. This application generates pin configurations for TI SoCs, connecting internal peripheral pins to external balls on the SoC's package. As there are only so many options for these connections, there are only so many correct pin configurations. This tool understands these restrictions and will provide warnings and errors for invalid configurations. It should be used by the board designer to develop a valid configuration for the desired use case. Any changes to the configuration need to be made in the tool to make sure that a conflict is not created. Such a conflict would likely lead to a board issue that would need to be debugged and fixed, likely costing very valuable debug time. The tool creates files to import the configuration into software, so that the device can be configured properly at boot. These files need to be passed from the board designer to the software developers for efficient board bring-up.

CAUTION

The board design and the software pin configuration must be kept in sync. An invalid pinmux configuration for board design or software leads to board issues that can be hard to diagnose and debug. Any pin related board changes need to be validated with the SysConfig Pinmux Tool and incorporated into software.

The pin configuration provided in the example minimal configuration is for a TI board. If using a TI board to validate this process for the AM62x device family, the configurations should not need to be changed. To replace the pin configurations that are provided in the custom board devicetree with the custom pin configurations from the pinmux DTSI generated by the SysConfig Pinmux Tool for the custom board, remove the `&main_pmx` node in `k3-am625-<boardname>.dts`. Then, paste the contents of the pinmux DTSI into `k3-am625-<boardname>.dts`. An example of this is shown below.

```
/* Delete provided pin configuration nodes */
- &main_pmx {
- [... UART pin configuration ...]
- [... SD pin configuration ...]
- };

/* Paste new pin configuration nodes from pinmux DTSI */
+ &main_pmx {
+ [... New pin configurations ...]
+ };
```

The new pin configuration for a peripheral needs to be set in the peripheral's devicetree node as well. This is done by modifying the `pinctrl` property in the node in `k3-am625-<boardname>.dts`.

Suppose the new pin controller node contains the following configuration for UART0, labeled "uart0-custom_pins_default".

```
uart0-custom_pins_default: uart0-custom-default-pins {
    pinctrl-single,pins = <
        AM62X_IOPAD(0x01c8, PIN_INPUT, 0) /* (D14) UART0_RXD */
        AM62X_IOPAD(0x01cc, PIN_OUTPUT, 0) /* (E14) UART0_TXD */
    >;
};
```

In order for UART0 to access these pins, modify the `pinctrl` property in the UART0 node as seen below. Make sure to use the correct label following the "&" to avoid devicetree compilation problems.

```
&main_uart0 {
    bootph-all;
    status = "okay";
    pinctrl-names = "default";
-   pinctrl-0 = <&main_uart0_pins_default>;
+   pinctrl-0 = <&uart0-custom_pins_default>;
};
```


This modification needs to be made for all peripherals that are configured on a custom board.

The instance of the peripheral that is initially configured in the custom board devicetree may also need to be modified. The custom board devicetrees assume that UART0 is used for serial output and for UART boot. It also assumes that MMC1 is used as the SD card reader. If the instance of the peripheral on the custom board is different, the devicetree node needs to be modified.

In the example below, assume that the correct UART instance for console output is UART1. Modify *k3-am625-<boardname>.dts* as seen below configure UART1 to function as the console output.

```
- &main_uart0 {
+ &main_uart1 {
    bootph-all;
    status = "okay";
    pinctrl-names = "default";
-   pinctrl-0 = <&uart0-custom_pins_default>;
+   pinctrl-0 = <&uart1-custom_pins_default>;
};
```

If the instance of a peripheral is modified, the pin configuration will also need to be changed to the correct pin node for that instance. In the example above, the pinctrl property is changed to the pin configuration for UART1.

If the UART instance for console output is changed, the *aliases* node in *k3-am625-<boardname>.dts* needs to be modified. The example below shows this modification if the UART instance for console output is UART1 instead of UART0.

```
aliases {
-   serial2 = &main_uart0;
+   serial2 = &main_uart1;
};
```

This ensures the Linux kernel uses the correct UART instance to output the kernel bootlog.

If the custom board does not have an SD card reader, remove or disable the corresponding node. This will prevent boot errors from occurring. The removal of this node is seen below.

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

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,am625-minimal", "ti,am625";
- model = "Texas Instruments AM625 MINIMAL";
+ compatible = "<company>,am625-<boardname>", "<company>,am625";
+ model = "<company> AM625 <boardname>";
```

4.5 Building U-Boot Binaries

This section explains how to create bootloader binaries using the custom board configuration and the AM62x Processor SDK.

For AM62x devices, three images, which are all created by the build process, are required to load U-Boot:

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

When the board is powered on, ROM code checks the bootmode pins and initializes the selected boot media required to load *tiboot3.bin* and execute it on the 32-bit R5 core. This image contains a secondary program loader (SPL) called the wakeup SPL. This SPL initializes DDR and basic board components that are used to load *tispl.bin*. This *tispl.bin* image contains the 64-bit main SPL running on an A53 core as well as firmware that is required to boot Linux. The main SPL initializes peripherals that are configured in *k3-am625-<boardname>.dts*, which are used to load the next artifacts in the bootflow. Finally, packaged in *u-boot.img*, U-Boot proper loads and additional peripherals may be initialized, preparing to boot Linux kernel components.

Since the boot process involves both 32-bit and 64-bit cores, a 32-bit and 64-bit cross compiler are required. The AM62x Processor SDK contains these cross-compilers, but the toolchain paths need to be set for each.

Set the following generic environment variables:

Table 4-1. Generic Environment Variables

Env Variable	Descriptions
CROSS_COMPILE_32	Cross compiler path for Armv7 (Arm32-bit), SDK provides arm-oe-eabi-
CROSS_COMPILE_64	Cross compiler path for Armv8 (Arm 64bit), SDK provides aarch64-oe-linux-
CC_64	Cross-compiler executable for Armv8 (Arm64bit) with default library and header paths specified, SDK provides aarch64-oe-linux-gcc
LNX_FW_PATH	Path to TI Linux firmware directory provided by SDK
TFA_PATH	Path to prebuilt Arm Trusted Firmware provided by SDK (bl31.bin)
OPTEE_PATH	Path to prebuilt OPTEE provided by SDK (bl32.bin)

To set the compiler toolchain paths, see the [AM62x Processor SDK Guide](#). The following is an example of how to set the firmware paths using the prebuilt binaries provided by the SDK.

```
$ export LNX_FW_PATH=TI_SDK/board-support/prebuilt-images/am62xx-evm/
$ export TFA_PATH=TI_SDK/board-support/prebuilt-images/am62xx-evm/bl31.bin
$ export OPTEE_PATH=TI_SDK/board-support/prebuilt-images/am62xx-evm/bl32.bin
```

Set the following board-specific environment variables:

Table 4-2. Board-Specific Environment Variables

Env Variable	Descriptions
UBOOT_CFG_CORTEXR	Default configuration file for Cortex-R
UBOOT_CFG_CORTEXA	Default configuration file for Cortex-A

Set the default configuration variable to apply the configuration fragments created earlier on top of the existing EVM default configuration. An example of setting these environment variables is below.

```
$ export UBOOT_CFG_CORTEXR="am62x_evmm_r5_defconfig am62x-<boardname>_r5.config"
$ export UBOOT_CFG_CORTEXA="am62x_evmm_a53_defconfig am62x-<boardname>_a53.config"
```

Since both 32-bit and 64-bit binaries are built, they need to be output into separate folders for each architecture. Create this output directory on the host machine. These instructions refer to the output directory path as `OUTPUT_DIR`. The 32-bit and 64-bit binaries are built into subdirectories of `OUTPUT_DIR`. In the following instructions, these subdirectories are named `r5` and `a53` for the 32-bit and 64-bit builds, respectively. All build commands are run from the root of the SDK U-Boot repository.

Building *tiboot3.bin*

```
$ make clean O=OUTPUT_DIR/r5
$ make ARCH=arm CROSS_COMPILE="$CROSS_COMPILE_32" $UBOOT_CFG_CORTEXR O=OUTPUT_DIR/r5
$ make ARCH=arm CROSS_COMPILE="$CROSS_COMPILE_32" O=OUTPUT_DIR/r5 BINMAN_INDIRS=$LNX_FW_PATH
```

Building tisp1.bin and u-boot.img

```
$ make clean O=OUTPUT_DIR/a53
$ make ARCH=arm CROSS_COMPILE="$CROSS_COMPILE_64" $UBOOT_CFG_CORTEXA O=OUTPUT_DIR/a53
$ make ARCH=arm CROSS_COMPILE="$CROSS_COMPILE_64" CC="$CC_64" O=OUTPUT_DIR/a53 BL31=$TFA_PATH
TEE=$OPTEE_PATH BINMAN_INDIRS=$LINUX_FW_PATH
```

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

Table 4-3. Generated Binaries Based on Device Security Level

Security	Generated Binaries
GP	tiboot3-am62x-gp-evm.bin tisp1.bin_unsigned u-boot.img_unsigned
HS-FS	tiboot3-am62x-hs-fs-evm.bin tisp1.bin u-boot.img
HS-SE	tiboot3-am62x-hs-evm.bin tisp1.bin u-boot.img

Note

The device security level of a new board is likely to be HS-FS.

The three generated binaries are found in the root directory of the r5 and a53 subdirectories of OUTPUT_DIR. These binaries must be renamed to exactly *tiboot3.bin*, *tisp1.bin* and *u-boot.img*. This is done with the following commands.

```
// in the r5 subdirectory of OUTPUT_DIR
$ mv tiboot3-am62x-{gp/hs-fs/hs}.bin tiboot3.bin
// in the a53 subdirectory of OUTPUT_DIR
$ mv tisp1.bin{unsigned} tisp1.bin
$ mv u-boot.img{unsigned} u-boot.img
```

4.6 U-Boot Deployment Instructions

To boot the device using the custom board configuration, the three boot binaries generated in the previous step are loaded to the board via UART. This method uses a simple, common peripheral to get to a U-Boot command prompt. The custom board needs to be able to configure the bootmode pins on the device for a valid UART boot setting. On TI EVMs, switches are provided to change the bootmode settings to support a variety of boot sources. On production boards, this change is likely accomplished with something as simple as a jumper or other setting. The following steps assume the board has been appropriately configured for UART boot.

The first step is to install a serial communication program, such as minicom, on the host PC. Configure this program to properly communicate with the board. The [AM62x Quick Start Guide](#) contains directions to setting up this serial connection. Verify the terminal setup and hardware connection is established by checking for the "C" characters that are continuously output from ROM while searching for the boot image. Example output shown below.

```
CCCCCCC
```

If no terminal output is seen, there are a variety of issues that could be causing this problem. It can be very useful to validate this step with a known good TI board to help isolate problems between the board and host setup. These issues must be resolved before continuing the bring-up process.

- Board issues (power supply, bootmode pins settings, trace continuity, and so forth)
- UART instance
- UART pin configuration
- Terminal setup

After establishing a serial connection from the host PC to the board, load the boot binaries over UART. The correct order to send the binaries is *tiboot3.bin*, *tispl.bin*, *u-boot.img*. If using minicom or a similar serial communication program, binaries are sent using a built-in application. If not, UART binaries can be sent to the board using the command prompt and the *lrzsz* package. For a complete description on using UART to load U-Boot binaries, see the [AM62x Processor SDK Guide on UART boot](#). As the document describes, the XMODEM protocol is used to send *tiboot3.bin* and the YMODEM protocol is used to send *tispl.bin* and *u-boot.img*.

The console outputs a block of text after each binary is transferred. If *tiboot3.bin* and *tispl.bin* were successfully transferred, the final line of each output is seen below.

```
Trying to boot from UART
CCCC
```

After transferring *u-boot.img*, the device prepares to boot Linux. At this point, U-Boot gives an opportunity to exit autoboot by pressing any key. Use this feature to stop the boot process at the U-Boot prompt. An example bootlog showing successfully reaching the U-Boot command shell is shown below.

```
U-Boot 2024.04-00006-g49c04dedb6d-dirty (Jul 19 2024 - 14:01:17 -0500)

SoC:   AM62X SR1.0 HS-FS
Model: Texas Instruments AM625 MINIMAL
EEPROM not available at 0x50, trying to read at 0x51
Reading on-board EEPROM at 0x51 failed -19
DRAM:  2 GiB
Core:  37 devices, 21 uclasses, devicetree: separate
MMC:   mmc@fa00000: 0
Loading Environment from nowhere... OK
In:    serial
Out:   serial
Err:   serial
Failed to probe am65_cpsw_nuss driver
EEPROM not available at 0x50, trying to read at 0x51
Reading on-board EEPROM at 0x51 failed -19
Net:   No ethernet found.
Hit any key to stop autoboot:  0
=>
```

Reaching the U-Boot command shell means that the custom board's devicetree configuration for UART is functional. It also shows that the de-tuned DDR settings work well enough to get U-Boot up and running, which is a very good sign. The next step is to boot Linux and reach the kernel command shell. This requires a Linux kernel image, kernel devicetree, and *initramfs* filesystem. Since this amount of data can not be efficiently transmitted via UART, another peripheral needs to be used to load these images.

5 Expanding the Custom Board Devicetree

At this point basic board functionality and the UART configuration have been validated by successfully loading U-Boot. In order to load Linux, which is much larger, a faster peripheral or block memory device is needed. Configuring this peripheral is the next step in the iterative bring-up process. Using Ethernet as an example, this section demonstrates how to add a new peripheral. Ethernet is a complex peripheral to configure, and it serves as a good example to understand the devicetree resources that may be required to configure other devicetree peripheral nodes. It will be simpler to start with a peripheral like SD, eMMC, or USB if those are available on the custom board. A sample SD configuration is included in the minimal configuration and may already be functional if the custom board has SD capability.

5.1 Devicetree Configuration

Devicetree bindings serve as a guide on how to create a node for a given peripheral. They are standardized node formats that are specific to the hardware that they represent. A devicetree binding specifies how to name a node and properties that a peripheral's node requires. They are required when drivers are submitted to upstream maintainers to be included in projects like U-Boot and Linux. These files also often have example nodes. Other devicetrees that use the same drivers will also be great examples for how to properly create a new node for a given peripheral.

Devicetree bindings are found in the Linux kernel documentation in *TI_LINUX/Documentation/devicetree/bindings/*. U-Boot provides bindings for some nodes as well. These are found in *TI_U_BOOT/include/dt-bindings/*. Since the board devicetrees in U-Boot and the Linux kernel are the same, it is helpful to consult both sources to construct peripheral nodes. Relevant bindings files are located by using a grep search on the compatible string associated with the node, property names, and property settings. More guidance on using the devicetree bindings is in the Bootlin devicetree training in [Section 10](#).

The first step to configuring a node is to check the SoC devicetree definition of the node. Nodes in SoC devicetrees contain most of the configuration settings required for the peripheral, but they are disabled since they are incomplete. The nodes are modified to match a board's hardware configuration and enabled in the board devicetree file, in this case *k3-am625-<boardname>.dts*. It may be helpful to refer back to the [AM62x Devicetree diagram](#) to understand how the board DTS files and the SoC DTSI files interact. Nodes often reference other nodes located throughout included devicetree files. To enable a node, ensure its referenced dependencies are also enabled. Working through the following Ethernet example should help make this more clear.

Cpsw3g, an Ethernet MAC, operates in the main domain. This indicates the SoC devicetree node definition is in *k3-am62-main.dtsi*. This node references other nodes, such as *&k3_clks*, *&main_pktdma*, and *&phy_gmii_sel*. It is necessary to ensure these nodes are enabled by the board devicetree. A node is enabled by setting the "status" property, as seen below.

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

In addition to being enabled, every peripheral that is configured needs to have its *pinctrl* property set to the correct pin configuration that was pasted into *k3-am625-<boardname>.dts*. Refer to [Section 4.4](#) to make this modification. After following these steps, the pin configuration properties are set below.

```
&cpsw3g {
    status = "okay";
    pinctrl-names = "default";
    pinctrl-0 = <rgmii1-custom_pins_default>;
};
```

For guidance on which pin configurations to include for a given peripheral, refer to the peripheral's configuration in *k3-am625-sk.dts* and *k3-am62x-sk-common.dtsi*.

The property "pinctrl-names" is added and set to "default" to specify the added pin configuration to always be used. Some configurations may require multiple pin configurations. For a complete guide to configuring pin settings, see *TI_LINUX/Documentation/devicetree/bindings/pinctrl/pinctrl-bindings.txt*.

5.2 Describing Peripherals in Nodes

A useful tool to help understand devicetree structure is reverse compiling the devicetree binary (DTB) that results from the build process. This results in a readable file that contains all nodes across the devicetrees included in the compilation process. The resulting file makes it easy to see all nodes and properties describing the peripherals. The DTB is found at *OUTPUT_DIR/a53/k3-am625-<boardname>.dtb*. To reverse compile the DTB, use the command below:

```
$ dtc -I dtb -O dts k3-am625-<boardname>.dtb -o <boardname>-reversed.dts
```

The reverse compiled DTB is the file *<boardname>-reversed.dts*. This output can be compared to the examples in the bindings files and other DTS files for completeness and correctness.

Another great resource to build a devicetree is to use other devicetrees as a guide to initialize peripherals. By compiling another devicetree, such as *k3-am625-sk.dts*, and reverse compiling the resulting DTB, it is easy to see other node configurations for peripherals. In general, the EVM devicetree nodes can be used as a good reference to represent a custom board's peripherals. If the use of a peripheral is similar to the EVM, only minor modifications may be needed, if any.

A good method for devicetree development on a custom board is to search the EVM devicetree files, *k3-am62x-sk-common.dtsi* and *k3-am625-sk.dts*, to see what properties are applied on top of the SoC definition of the node. These properties usually need to be set in the custom board's devicetree as well. Searching in the devicetree bindings directory for properties and compatible strings leads to documentation on what each property defines and how it can be set.

In *k3-am625-sk.dts*, the Ethernet PHYs are defined as child nodes of the MDIO bus node. This is added to the board devicetree below. Assume the pin configuration and the status properties have already been set.

```
cpsw3g_mdio {
    status = "okay";
    pinctrl-names = "default";
    pinctrl-0 = <mdio-custom_pins_default>;
+   cpsw3g_phy0: ethernet-phy@0 {
+       };
};
```

In *k3-am625-sk.dts*, there are properties defined in the Ethernet PHY nodes. A grep search on any of these properties from the kernel bindings directory points us to *ti,dp83867.txt*, which indicates that there are four required properties for the PHY nodes. These are the same properties specified in the EVM board devicetree. This document also points us to a header file with different settings for these properties, which are control registers for the Ethernet PHY. These settings can be set based on the EVM devicetree settings, or by referencing the Ethernet port design documentation for device specifications. These values and the method of incorporation in the devicetree will change if different PHYs are used. This is seen below:

```
cpsw3g_mdio {
    status = "okay";
    pinctrl-names = "default";
    pinctrl-0 = <mdio-custom_pins_default>;
    cpsw3g_phy0: ethernet-phy@0 {
+       reg = <0>;
+       ti,rx-internal-delay = <DP83867_RGMIIDCTL_2_25_NS>;
+       ti,tx-internal-delay = <DP83867_RGMIIDCTL_2_75_NS>;
+       ti,fifo-depth = <DP83867_PHYCR_FIFO_DEPTH_4_B_NIB>;
+   };
};
```

The EVM devicetree adds the "phy-handle" property for *&cpsw_port1*, which assigns the PHY node to the port node. This property can be set below after enabling the node.

```
+ &cpsw_port1 {
+     status = "okay";
+     phy-handle = <&cpsw3g_phy0>;
+ };
```

5.3 Revising the Devicetree Configuration

Suppose this devicetree is used to generate U-Boot binaries to test the Ethernet configuration. After transferring *u-boot.img* over UART per the sections above, the U-Boot console output is displayed below when any key is used to stop autoboot.

```
U-Boot 2024.04-00007-g344db2cf625-dirty (Jul 23 2024 - 09:05:14 -0500)

SoC:   AM62X SR1.0 HS-FS
Model: Texas Instruments AM625 SK
EEPROM not available at 0x50, trying to read at 0x51
Reading on-board EEPROM at 0x51 failed -121
DRAM:  2 GiB
Core:  40 devices, 22 uclasses, devicetree: separate
MMC:   mmc@fa10000: 0, mmc@fa00000: 1
Loading Environment from nowhere... OK
In:    serial
Out:   serial
Err:   serial
EEPROM not available at 0x50, trying to read at 0x51
Net:   am65_cpsw_nuss_port ethernet@8000000port@1: Invalid PHY mode, port 1
No ethernet found.
```



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

The "Net" field indicates the status of the Ethernet configuration at this point. An error message shows that Ethernet port 1 is recognizing the PHY as operating in an invalid mode. This likely points to a devicetree issue, as the message indicates the PHY is either not operating in the correct mode, or the Ethernet port is not recognizing the mode the PHY is operating in.

To fix this error, look to the PHY and Ethernet port nodes as the possible causes. In the EVM devicetree, the port node has a property "phy-mode" that is set to "rgmii-rxid". A search on the property assignment, "rgmii-rxid", finds the binding file *ethernet-controller.yaml*. This file specifies that "phy-mode" is set in the port node to specify the interface between the PHY and Ethernet port. The bindings specify that "rgmii_rxid" should be set as the "phy-mode" if the PHY provides a RX delay. Since an RX delay was set as required by the PHY bindings in an earlier step, the property applies and is added to the devicetree.

```
&cpsw_port1 {
    phy-handle = <&cpsw3g_phy0>;
+   phy-mode = "rgmii-rxid";
};
```

This highlights the iterative process that eventually results in a functional board devicetree. Once this fix is applied, rebuilding and reloading U-Boot over UART should show a functional Ethernet port available to U-Boot. This port can now be used to transfer the larger images needed to load Linux.

6 Booting the Linux Kernel

6.1 Kernel Boot Overview

The process of booting the Linux kernel in the next section uses components that are provided in the SDK in order to streamline the bring-up process. The only component that needs to be rebuilt when testing a new configuration is the devicetree. The kernel image and filesystem are provided in the SDK and reused throughout the board bring-up process. This capability highlights the value of devicetree as a hardware description language as the same kernel and filesystem can be used by a different board with no code changes.

The U-Boot devicetree that was configured in the previous section is reused as the kernel devicetree. This is because a board's U-Boot and kernel devicetree should be developed in sync to simplify the debug process. Instructions in the next section detail reusing the devicetree.

The *initramfs* filesystem is used to boot the kernel during board bring-up. This is to reduce any hardware dependencies required to mount a physical root filesystem. After the board bring-up process is completed, a physical filesystem in non-volatile storage like eMMC or a developer friendly filesystem mounted via NFS can be used to enable full functionality.

6.2 Kernel Deployment Instructions

This section covers loading the Linux kernel image, devicetree, and filesystem once a peripheral capable of loading these components into RAM has been configured. To test the kernel configuration, load U-Boot binaries over UART and exit autoboot when prompted as in previous sections.

This method uses the kernel image and *initramfs* filesystem that is provided by the AM62x Processor SDK. The table below specifies where these images are found within the SDK.

Table 6-1. SDK Paths to Linux Kernel Components

Component	Path
Kernel Image	TI_SDK/board-support/prebuilt-images/am62xx-evm/Image
Ramdisk	TI_SDK/filesystem/am62xx-evm/tisdk-tiny-initramfs-am62xx-evm.rootfs.cpio

The kernel devicetree needs to be compiled. While this devicetree will be identical to the devicetree used for U-Boot to this point, it must be compiled in the Linux kernel repository.

Follow the commands below to copy the edited board devicetree file from U-Boot to the Linux kernel repository and generate the kernel DTB.

```
// copy board DTS from U-Boot to kernel
$ cp TI_U_BOOT/arch/arm/dts/k3-am625-<boardname>.dts TI_LINUX/arch/arm64/boot/dts/ti/

// from the root of 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" ti/k3-am625-<boardname>.dtb
```

The resulting DTB is found at `TI_LINUX/arch/arm64/boot/dts/ti/k3-am625-<boardname>.dtb`.

The kernel image, devicetree, and filesystem are loaded separately into RAM through a peripheral or copied from non-volatile storage. This process depends on the peripheral or interface that was enabled on the custom board in the above sections. The Linux images need to be copied to the external media or host directory that is used to transfer the kernel components to the device through the available interface. In the above section, Ethernet was discussed and can be used if it is functional. In the below steps, SD Card is used as it is readily available on TI boards and is easy to use at this stage of board bring up.

To use SD card to load kernel components, a partitioned SD card is required. One way to create this is to flash the SD card with a default image provided by the SDK. This is done using an application like *balenaEtcher*. The default image is found at the [AM62x Processor SDK Download Page](#). Select "Downloads" and then select "Download options" for PROCESSOR-SDK-LINUX-AM62X. Download the AM62x Yocto SD card image and flash it to the SD card. This will partition the card correctly and place a working version of the SDK filesystem on the card. This filesystem will not be used at this point.

To avoid confusion, remove all files from the boot partition and copy the kernel components to this location. A guide to flashing the SD card can be found in the [AM62x Processor SDK Guide](#).

Use UART boot to get to a U-Boot prompt and halt autoboot. Use the U-Boot load command for the peripheral used to load kernel components into RAM. Refer to the [U-Boot Documentation](#) to determine what commands to use for each different peripheral. The following example is for using SD card to load the required kernel components.

```
/* using MMC device 0 (SD), partition 1 */
=> load mmc 0:1 $loadaddr Image
=> load mmc 0:1 $fdtaddr k3-am625-<boardname>.dtb
=> load mmc 0:1 $rdaddr tisdk-tiny-initramfs-am62xx-evm.rootfs.cpio
```

Note

If the `loadaddr`, `fdtaddr`, and `rdaddr` U-Boot environment variables are not set at this point, U-Boot was not properly configured for the custom setup. Return to [Section 3](#) to correct the U-Boot environment for the custom board.

If the image fails to load, there is likely an issue with the device configuration. Return to [Section 5.1](#) to attempt to reconfigure the device.

Now that the required kernel components have been loaded into device RAM, the next step is to specify the kernel initialization process. For board bring-up, this is to enter a kernel command shell and use `initramfs` as the filesystem. To do this, set the `bootargs` environment variable, which is passed from U-Boot to the Linux kernel when it begins to boot. The following U-Boot command sets this initialization process.

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

The device is now prepared to attempt to boot the kernel using the images copied to RAM. The command below boots the kernel components stored in RAM.

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

The Linux kernel will attempt to boot. If successful, the kernel bootlog will populate with initialization messages and eventually enter a kernel command shell. In order to have access to the system and hardware information, mount the following pseudo-file systems using the commands below.

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

If you did not reach the kernel command shell, there are multiple methods to debug this issue. For methods on how to attempt to reconfigure a peripheral correctly to successfully boot the kernel, see [Section 5.1](#). For help debugging, see [Section 7](#).

Reaching the kernel command line provides access to tools that are useful in building the devicetree and debugging boot failures. It is important to continue the same iterative approach of enabling each peripheral one by one, and committing code to the U-Boot repository once a working configuration has been established. The devicetree configuration methods provided in [Section 5.1](#) as well as tools in the next section are useful in building a devicetree to initialize all peripherals on the board.

Common next steps to complete the board bring-up process include configuring the custom board to use an external filesystem, enabling high speed data features, and customizing the kernel image. For more details, see the [AM62x Processor SDK Guide](#).

7 Tools and Debugging

When attempting to bring-up a custom board, the expectation is to encounter issues during the boot process. This section explains common issues that arise when booting custom boards and how to approach the debugging process.

7.1 Kernel Debug Traces

Early on in the board bring up process, it can be difficult to detect the root cause of a boot failure if the kernel fails prior to bootlogs being printed. Additionally, kernel bootlogs can be difficult to interpret. Adding kernel debug features to the kernel image through *menuconfig* selection helps isolate the cause of the failure.

Enabling additional debug features requires building a custom kernel image. These steps come from the [AM62x Processor SDK Guide](#). These commands are run from the root of the SDK Linux kernel repository.

1. Install tools required to build the kernel image using the command below.

```
sudo apt install git xz-utils build-essential flex bison bc libssl-dev libncurses-dev
```

2. Clean source to begin a fresh build of the kernel.

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

3. Enter the configuration menu.

To enable extra debugging features, enable additional config options. This is done by using *menuconfig*. To apply the *menuconfig* settings over the recommended configuration file fragment, use the following command.

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

4. Enable debug features.

Navigate to "Kernel hacking" to view the kernel debug options. Enable debugging features in *menuconfig* sets them to be built into the kernel image. A description of these debug configuration options is found at *TI_LINUX/lib/Kconfig.debug*. After enabling debug features, save the configuration and exit the configuration menu. Build the kernel image using the following command.

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

The resulting image is found in *TI_LINUX/arch/arm64/boot/Image*. Using this image enables access to the added debug features.

7.2 OpenOCD Debugging

It is recommended to debug the board using JTAG to find the root of the issue. 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](#).

8 Future Work

This minimal configuration was developed for the AM62x using concepts that apply more broadly to other TI SoCs and even other Arm® processors running U-Boot and/or Linux. The configuration specifically can be extended to other devices in the K3 family (AM64x, AM67x, and so forth). It is recommended to use this minimal configuration as an example to build similar devicetrees for other SoCs. The following steps can serve as a general approach to creating a minimal configuration.

1. Create minimal devicetrees for U-Boot and the Linux Kernel

```
k3-{SOC}-minimal.dts k3-{SOC}-r5-minimal.dts k3-{SOC}-minimal-u-boot.dtsi
```

2. Create minimal configuration fragments to incorporate minimal devicetrees

```
{SOC}_minimal_a53.config {SOC}_minimal_r5.config
```

9 Summary

By reducing the number of enabled peripherals and reducing their functionality, a minimal configuration can be created for an SoC as an initial test for bringing up a custom board. This application note aims to ease the process of board bring up by outlining a step-by-step process to use the minimal configuration as a starting point to setup a custom board configuration. The minimal configuration streamlines this process and reduces the time of the initial bring up process from days or weeks to hours.

10 References

- Texas Instruments, [AM62x Processors Silicon Revision 1.0 Technical Reference Manual](#).
- Texas Instruments, [Processor SDK Linux for AM62x](#).
- Bootlin, [Device Tree for Dummies](#).
- [U-Boot Documentation](#).
- [OpenOCD User's Guide](#).
- Relevant software files:
 - [k3-am625-minimal.dts](#)
 - [k3-am625-r5-minimal.dts](#)
 - [k3-am625-minimal-u-boot.dtsi](#)

Revision History

Changes from Revision * (August 2023) to Revision A (September 2024)	Page
• Updated Section Abstract	1
• Updated Section 1	2
• Added Section 2	2
• Added Section 3	2
• Updated Section 4	4
• Added Section 4.1	4
• Updated Section 4.2	5
• Updated Section 4.3	6
• Added Section 4.4	8
• Added Section 4.5	9
• Updated Section 4.6	11
• Added Section 5	12
• Updated Section 5.1	12
• Added Section 5.2	13
• Added Section 5.3	14
• Added Section 6	15
• Added Section 6.1	15
• Added Section 6.2	15
• Updated Section 7	17
• Added Section 7.1	17
• Updated Section 7.2	18
• Updated Section 8	18
• Updated Section 9	18

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATA SHEETS), 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 will 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](#) or other applicable terms available either on [ti.com](https://www.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.

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

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2024, Texas Instruments Incorporated