Implementing a DSI Driver on the Raspberry Pi

Raspberry Pi Ltd

2024-01-03: githash: a97226e-clean

Colophon

build-date: 2024-01-03
build-version: githash: a97226e-clean

Legal Disclaimer Notice

Document version history

Release	Date	Description
1.0	1 Sept 2021	Initial release
1.1	27 April 2022	Copy edit, public release
1.2	3 Jan 2024	Added information on Raspberry Pi 5

Scope of document

This document applies to the following Raspberry Pi products:

Pi 0			Pi 1		Pi 2		Pi 3	Pi 4	Pi 400	Pi 5	CM 1	CM 3	CM 4	Pico
0	W	H	A	B	A	B	B	All	All	All	All	All	All	All
			*	*	*	*	*	*		*	*	*	*	

Introduction

This whitepaper is an introduction to writing a Display Serial Interface (DSI) driver for liquid crystal display (LCD) panels running under the Kernel Mode Setting (KMS) graphics system. It does *not* cover DSI drivers for the legacy graphics system, or Fake Kernel Mode Setting (FKMS).

It assumes that the Raspberry Pi is running the Raspberry Pi operating system (OS; Linux), and is fully up to date with the latest firmware and kernels.

Terminology

DSI: A specification for interfacing with display devices, typically LCD panels.

Legacy graphics stack: A graphics stack wholly implemented in the VideoCore firmware blob with a shim application programming interface (API) exposed to the kernel. This is what has been used on the majority of Raspberry Pi Ltd's Pi devices since launch, but has been replaced by KMS/DRM in Raspberry Pi OS Bullseye and later OSs.

vc4-kms-v3d: The full KMS driver for Raspberry Pi devices. Controls the entire display process, including talking to the hardware directly with no firmware interaction.

FKMS: While the firmware still controls the low-level hardware (for example the High-Definition Multimedia Interface (HDMI) ports, DSI, etc.), standard Linux libraries are used in the kernel itself.

KMS: Kernel Mode Setting API.

DRM: Direct Rendering Manager, a subsystem of the Linux kernel used to communicate with graphics processing units (GPUs). Used in partnership with FKMS and KMS.

SoC: System on a chip. The main processor on Raspberry Pi boards, which varies according to model.

MIPI: Mobile Industry Processor Interface.

MIPI Alliance: The standards body for MIPI protocols.

D-PHY: MIPI physical/electrical standard for DSI (and Camera Serial Interface 2). The C-PHY and M-PHY standards are *not* supported on Raspberry Pi devices.

DT: Device Tree, a mechanism for defining the hardware characteristics of a device.

Prerequisites

In order to develop a DSI driver there are several requirements:

- A Raspberry Pi (preferably 4B 4GB or 8GB if you are working and compiling on the Pi itself). Ensure that you have the line `dtoverlay=vc4-kms-v3d` in `config.txt` so you are using the KMS driver.
- A buildable kernel tree. The Raspberry Pi kernel tree can be found at `https://www.github.com/raspberrypi/linux`.
- A datasheet for the LCD panel showing the resolution and timings. The register sets from the manufacturer for panel initialisation are also very useful, sometimes critical, here.

Concepts

DSI

DSI specifies a high-speed point-to-point serial bus that has a clock lane plus one to four data lanes.

The DSI and D-PHY specifications are maintained by the MIPI Alliance.

A number of introductions to these standards are available, so no detail will be given here on the electrical level or low-level protocol — please refer to the specifications.

The SoCs used on Raspberry Pi devices implement two DSI interfaces. On Raspberry Pi 1 to Raspberry Pi 4, only one port, the 4-lane DSI1, is exposed. On the CM devices up to the Raspberry Pi CM4, two ports are exposed, the 4-lane DSI1 and the 2-lane DSI0. The devices implement DSI 1.0 over D-PHY 1.01.

On the Raspberry Pi 5, both ports have moved from the SoC to the Raspberry Pi RP1 device and are exposed. Both DSI0 and DSI1 are 4-lane and implement DSI 1.1.

DSI links normally operate in Low Power (LP) mode or High Speed (HS) mode. LP mode is typically around 10MHz, which is not fast enough to carry video. It is frequently used for display commands and configuration (LP signalling). Displays can accept video in either Command or Video modes. Command mode is relatively rare, and generally requires the display to have a full frame buffer. This mode is currently not supported by the Raspberry Pi Ltd drivers. There is also an Ultra-Low-Power State (ULPS), but this is rarely used.

The lowest speed at which HS mode runs depends on the display bit rate (size and colour depth) and the number of lanes in use. DSI supports a burst mode which runs HS mode faster than normally required, so the lane(s) can drop back to an LP state during the blanking periods to save power.

Display configuration has two parts: the Display Command Set (DCS), which contains a standard set of commands, and the Manufacturer Command Set (MCS), which contains display-specific commands. Commands can be sent in either LP or HS modes, although some devices will have restrictions on which modes are supported.

In general, there is very little commonality between displays, and the DCS is insufficient to set them up ready for use, which means that manufacturer-specific commands are almost always needed when trying to program a display. This means that, on the whole, each display requires its own driver, or at the very least its own set of register setup information.

DRM

Refer to the Linux kernel documentation for the latest (and definitive) documentation of DRM.

A number of online introductions are available to give an overview, for example:

- https://bootlin.com/pub/conferences/2017/kr/ripard-drm/ripard-drm.pdf
- https://events.static.linuxfound.org/sites/events/files/slides/brezillon-drm-kms.pdf

DRM splits the display composition pipe into **planes** that are composed and fed into a **CRTC** (cathode-ray tube controller) to implement display timings. That feeds an **encoder**, e.g. HDMI, Display Parallel Interface (DPI), DSI, or low-voltage differential signalling (LVDS), potentially through one or more **bridges** to either a **connector** or a **panel**.

- Plane: A memory object from which a scanout engine (a CRTC) is fed.
- CRTC: Reads the pixel data and generates the video mode timing.
- Encoder: Encodes the video mode timings to something suitable for the connector or panel.
- Connector: Represents where the display controller sends out the signal, for example, an HDMI connector.

- Panel: An alternative to a connector for display devices attached directly to the device rather than via a connector. Conceptually very similar to a connector.
- Bridge: A device that converts one form of video signal to another.

All display pipelines will have planes and CRTCs as standard. Encoders, bridges, connectors, and panels are specific to the computing device and the display device.

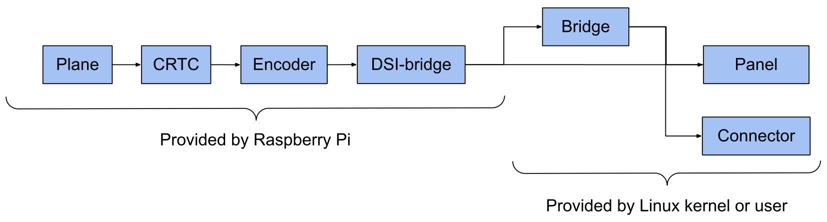
NOTE

The encoder is a partially deprecated concept, but as it was exposed via a userspace API it has to be retained for backward compatibility. When not used, the CRTC will feed a connector or panel, possibly via a bridge.

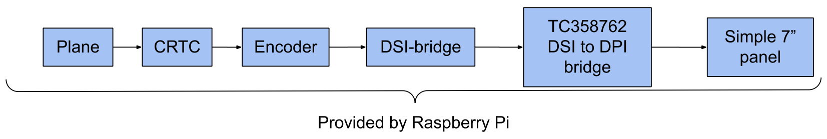
Drivers for DRM/KMS normally live under `/drivers/gpu/drm/` in the kernel source tree, or under `/drivers/staging` for those that have not yet been fully accepted.

DRM pipeline on Raspberry Pi devices

The following diagram shows a generic DRM pipeline on a Raspberry Pi device.



The next diagram shows the specific pipeline used for the Raspberry Pi 7" LCD panel.



DSI under DRM/KMS

When adding a DSI display, you will need to configure bridge drivers, if present, and panels. If drivers are not available for the panels or bridge, then they will need to be written.

In common with the majority of Linux implementations, one driver should do one thing and do it well. Where a panel is connected via a generic bridge chip (e.g. TI SN65DSI83 DSI to LVDS, or Toshiba TC358762 DSI to DPI), then the bridge chip is normally expected to have a driver, and it should *not* be integrated into the panel driver.

Most panels connected via bridges can be configured by extending the panel-simple driver. If the panel is fully integrated and takes DSI directly in, then there are example panel drivers such as Ilitek ILI9881C and Sitronix ST7701.

All configuration happens via the DT. The base DT will already define the `dsi0` and `dsi1` interfaces, but they will be disabled. The Raspberry Pi Ltd kernel tree uses DT overlays to extend the base DT. This is different from mainline Linux which generally expects the base DT to be a complete description of the hardware. The details of DT are covered in other documentation, so will not be duplicated here.

There are already `dtoverlays` merged into the Raspberry Pi Ltd kernel tree for the JDI LT070ME05000 1200 × 1920 native DSI panel, the Raspberry Pi 7" panel, which uses a Toshiba TC358762 DSI to DPI bridge chip, and the TI SN65DSI83 DSI to LVDS bridge.

All three are drivers that are merged into the mainline Linux kernel, and therefore the DT bindings documentation will be found in the mainline tree.

Note that regulator and reset general-purpose input/output (GPIO) control is generally abstracted out from any driver, as then it can be used on a variety of hardware with only configuration changes.

DSI device drivers

Please check to see if a driver for your panel/bridge has been sent to the dri-devel kernel mailing list before starting on a new one from scratch. Many drivers get started but not cleaned up for the final merge, and those are generally better starting points than basing a new driver on something else. Likewise, the Raspberry Pi Ltd kernel tree generally sticks to the Long Term Stable (LTS) kernel releases, so there may be drivers that have been merged in more recent kernel versions that can be used as a starting point.

Bridges and panels have slightly different APIs, but they support the same concepts.

For a bridge, please implement the DRM atomic API calls, not the legacy non-atomic ones. That means using the following functions in `struct drm_bridge_funcs`:

- `atomic_pre_enable`
- `atomic_enable`
- `atomic_disable`
- `atomic_post_disable`

The SN65DSI83 driver is a good example of a bridge driver using the atomic API.

With the atomic APIs the mode_set call is deprecated, and the bridge should retrieve the mode from the `crtc_state` when required; see `sn65dsi83_atomic_enable`.

DRM calls all the `*_pre_enable` calls first, starting at the furthest from the encoder (generally the panel or connector) and working backwards. This is generally the correct call to enable the regulators, clear any reset GPIOs, and generally configure the bridge. Configuration is normally done with either inter-integrated circuit (I2C) commands or using the `mipi_dsi_*` commands or `mipi_dsi_generic_wri_*te`, defined in the kernel tree in drm_mipi_dsi.h.

Once back to the encoder, it enables the CRTC and encoder to start video down the pipeline.

It then calls the `*_enable` calls down the chain from the encoder to the panel/connector. This is normally used to enable backlights and similar.

When being disabled the reverse happens: `*_disable` functions are called first from the panel/connector to the encoder, video is disabled, and then `*_post_disable` are called from the encoder to the panel/connector.

The same principles hold for a panel, but the calls are `prepare`/`enable` and `disable`/`unprepare` in `struct drm_panel_funcs`.

The number of lanes and various other mode flags are configured via fields in the `struct mipi_dsi_device` given to the panel/bridge probe function.

DSI supports several virtual channels on a single link. While the SoC has some support for this, it is rarely used on displays. It can be configured via the `channel` field in `struct mipi_dsi_device` and `struct mipi_dsi_msg`.

Panel timings

The panel, via its driver, specifies the timings it desires, and this is passed down the chain.

The Raspberry Pi Ltd DSI block on the Raspberry Pi 1 to Raspberry Pi 4 is driven off an integer divider from a fixed phase-locked loop (PLL), and can therefore only achieve specific link frequencies, meaning that DSI's burst mode is used under most conditions. The function `vc4_dsi_bridge_mode_fixup` can be used to increase the horizontal front porch to account for the increased pixel clocks while keeping the delay between sync pulse and image data, and the overall line time, the same. This means the link frequency used may not be the same as that requested.

On the Raspberry Pi 5, the DSI block is part of the Raspberry Pi RP1 device and is allocated its own PLL, which means it is capable of any link frequency. This means that the link frequency will be the same as that requested.

NOTE

When moving from using a display on the Raspberry Pi 4 to a Raspberry Pi 5, this change in link frequency handling may cause problems if the driver inadvertently relies on this adjustment. You may need to correct the link frequency and porch values appropriately.

Further reading

Jagan Teki's *Demystifying Linux MIPI-DSI Subsystem*

Raspberry Pi