RP2350 Datasheet

A microcontroller
by Raspberry Pi

Colophon

build-date: 2025-07-29
build-version: d126e9e-clean

Legal disclaimer notice

TECHNICAL AND RELIABILITY DATA FOR RASPBERRY PI PRODUCTS (INCLUDING DATASHEETS) AS MODIFIED FROM TIME TO TIME ("RESOURCES") ARE PROVIDED BY RASPBERRY PI LTD ("RPL") "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW IN NO EVENT SHALL RPL BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THE RESOURCES, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

RPL reserves the right to make any enhancements, improvements, corrections or any other modifications to the RESOURCES or any products described in them at any time and without further notice.

The RESOURCES are intended for skilled users with suitable levels of design knowledge. Users are solely responsible for their selection and use of the RESOURCES and any application of the products described in them. User agrees to indemnify and hold RPL harmless against all liabilities, costs, damages or other losses arising out of their use of the RESOURCES.

RPL grants users permission to use the RESOURCES solely in conjunction with the Raspberry Pi products. All other use of the RESOURCES is prohibited. No licence is granted to any other RPL or other third party intellectual property right.

HIGH RISK ACTIVITIES. Raspberry Pi products are not designed, manufactured or intended for use in hazardous environments requiring fail safe performance, such as in the operation of nuclear facilities, aircraft navigation or communication systems, air traffic control, weapons systems or safety-critical applications (including life support systems and other medical devices), in which the failure of the products could lead directly to death, personal injury or severe physical or environmental damage ("High Risk Activities"). RPL specifically disclaims any express or implied warranty of fitness for High Risk Activities and accepts no liability for use or inclusions of Raspberry Pi products in High Risk Activities.

Raspberry Pi products are provided subject to RPL's Standard Terms. RPL's provision of the RESOURCES does not expand or otherwise modify RPL's Standard Terms including but not limited to the disclaimers and warranties expressed in them.

Table of contents

Chapter 1. Introduction

RP2350 is a new family of microcontrollers from Raspberry Pi that offers major enhancements over RP2040. Key features include:

- Dual Cortex-M33 or Hazard3 processors at 150 MHz
- 520 kB on-chip SRAM, in 10 independent banks
- 8 kB of one-time-programmable storage (OTP)
- Up to 16 MB of external QSPI flash or PSRAM through dedicated QSPI bus
- Additional 16 MB flash or PSRAM through optional second chip-select
- On-chip switched-mode power supply to generate core voltage
- Optional low-quiescent-current LDO mode for sleep states
- 2 × on-chip PLLs for internal or external clock generation
- GPIOs are 5 V-tolerant (powered) and 3.3 V-failsafe (unpowered)
- Security features:
  - Optional boot signing, enforced by on-chip mask ROM, with key fingerprint in OTP
  - Protected OTP storage for optional boot decryption key
  - Global bus filtering based on Arm or RISC-V security/privilege levels
  - Peripherals, GPIOs, and DMA channels individually assignable to security domains
  - Hardware mitigations for fault injection attacks
  - Hardware SHA-256 accelerator
- Peripherals:
  - 2 × UARTs
  - 2 × SPI controllers
  - 2 × I2C controllers
  - 24 × PWM channels
  - USB 1.1 controller and PHY, with host and device support
  - 12 × PIO state machines
  - 1 × HSTX peripheral

Table 1 shows the RP2350 family of devices, including options for QFN-80 (10 × 10 mm) and QFN-60 (7 × 7 mm) packages, with and without flash-in-package.

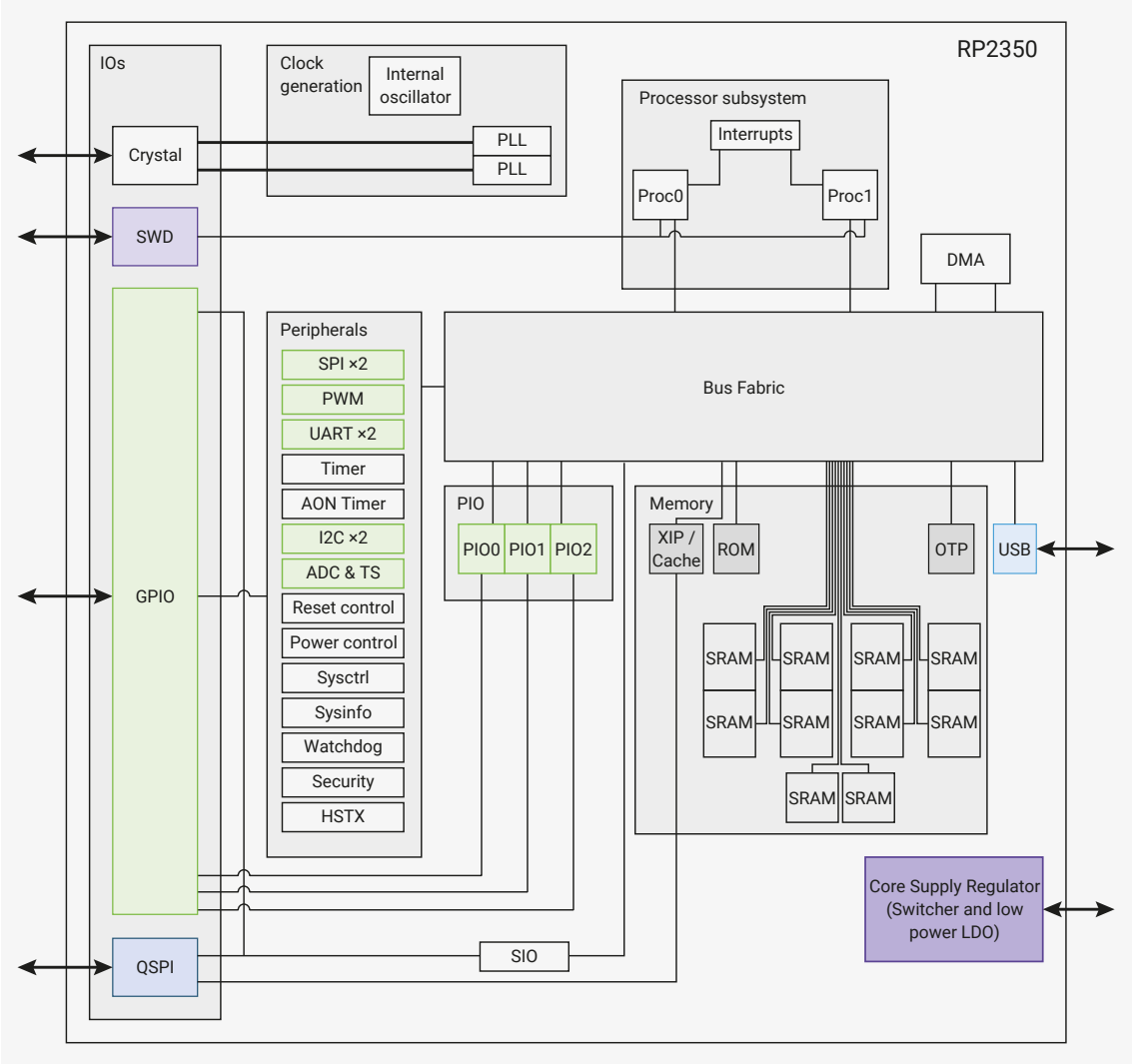
Table 1. RP2350 device family

Product	Package	Internal Flash	GPIO	Analogue Inputs
RP2350A	QFN-60	None	30	4
RP2350B	QFN-80	None	48	8
RP2354A	QFN-60	2 MB	30	4
RP2354B	QFN-80	2 MB	48	8

1.1. The chip

Dual Cortex-M33 or Hazard3 processors access RP2350's memory and peripherals via AHB and APB bus fabric.

Figure 1. A system overview of the RP2350 chip



Code may execute directly from external memory through a dedicated QSPI memory interface in the execute-in-place subsystem (XIP). The cache improves XIP performance significantly. Both flash and RAM can attach via this interface.

Debug is available via the SWD interface. This allows an external host to load, run, halt and inspect software running on the system, or configure the execution trace output.

Internal SRAM can contain code or data. It is addressed as a single 520 kB region, but physically partitioned into 10 banks to allow simultaneous parallel access from different managers. All SRAM supports single-cycle access.

A high-bandwidth system DMA offloads repetitive data transfer tasks from the processors.

GPIO pins can be driven directly via single-cycle IO (SIO), or from a variety of dedicated logic functions such as the hardware SPI, I2C, UART and PWM. Programmable IO controllers (PIO) can provide a wider variety of IO functions, or supplement the number of fixed-function peripherals.

A USB controller with embedded PHY provides FS/LS Host or Device connectivity under software control.

Four or eight ADC inputs (depending on package size) are shared with GPIO pins.

Two PLLs provide a fixed 48 MHz clock for USB or ADC, and a flexible system clock up to 150 MHz. A crystal oscillator provides a precise reference for the PLLs.

An internal voltage regulator supplies the core voltage, so you need generally only supply the IO voltage. It operates as a

switched mode buck converter when the system is awake, providing up to 200 mA at a variable output voltage, and can switch to a low-quiescent-current LDO mode when the system is asleep, providing up to 1 mA for state retention.

The system features low-power states where unused logic is powered off, supporting wakeup from timer or IO events. The amount of SRAM retained during power-down is configurable.

The internal 8 kB one-time-programmable storage (OTP) contains chip information such as unique identifiers, can be used to configure hardware and bootrom security features, and can be programmed with user-supplied code and data.

The built-in bootrom implements direct boot from flash or OTP, and serial boot from USB or UART. Code signature enforcement is supported for all boot media, using a key fingerprint registered in internal OTP storage. OTP can also store decryption keys for encrypted boot, preventing flash contents from being read externally.

RISC-V architecture support is implemented by dynamically swapping the Cortex-M33 (Armv8-M) processors with Hazard3 (RV32IMAC+) processors. Both architectures are available on all RP2350-family devices. The RISC-V cores support debug over SWD, and can be programmed with the same SDK as the Arm cores.

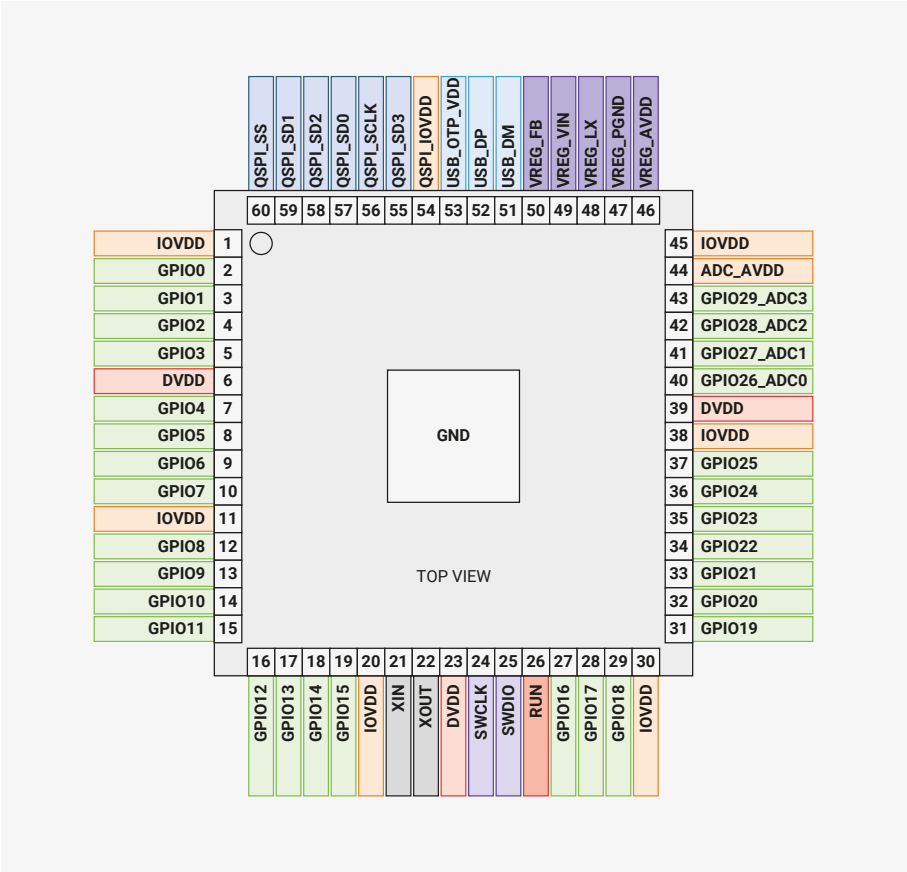
1.2. Pinout reference

This section provides a quick reference for pinout and pin functions. Full details, including electrical specifications and package drawings, can be found in Chapter 14.

1.2.1. Pin locations

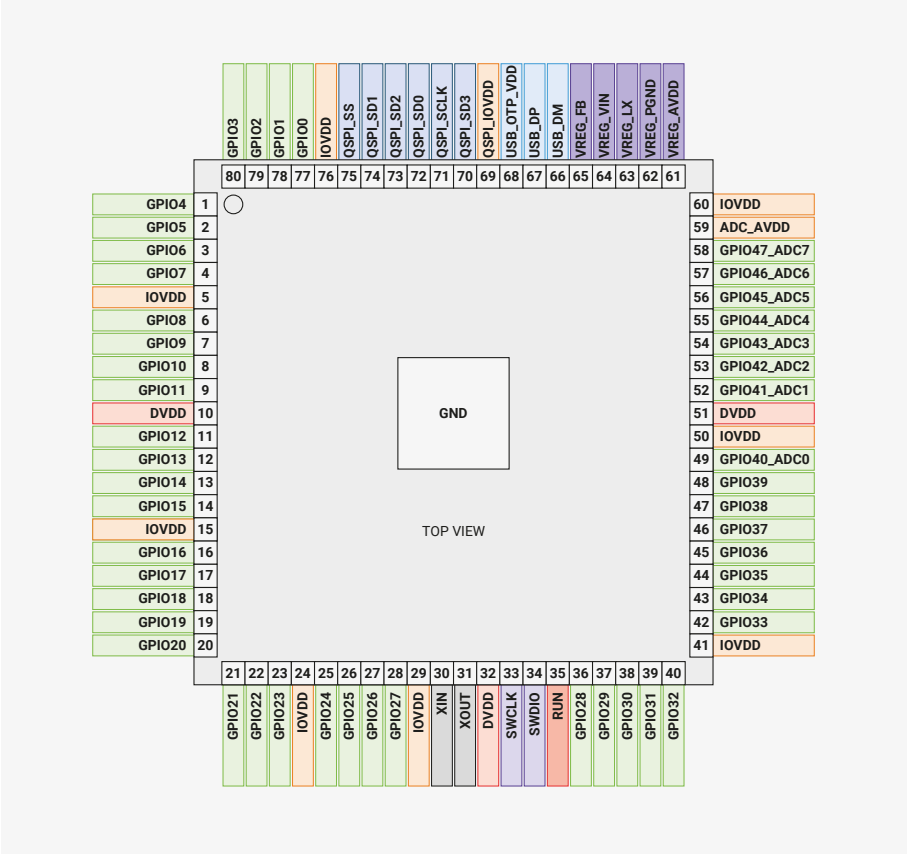
1.2.1.1. QFN-60 (RP2350A)

Figure 2. RP2350 Pinout for QFN-60 7×7mm (reduced ePad size)



1.2.1.2. QFN-80 (RP2350B)

Figure 3. RP2350 Pinout for QFN-80 10×10mm (reduced ePad size)



1.2.2. Pin descriptions

Table 2. The function of each pin is briefly described here. Full electrical specifications can be found in Chapter 14.

Name	Description
GPIOx	General-purpose digital input and output. RP2350 can connect one of a number of internal peripherals to each GPIO, or control GPIOs directly from software.
GPIOx/ADCy	General-purpose digital input and output, with analogue-to-digital converter function. The RP2350 ADC has an analogue multiplexer which can select any one of these pins, and sample the voltage.
QSPIx	Interface to a SPI, Dual-SPI or Quad-SPI flash or PSRAM device, with execute-in-place support. These pins can also be used as software-controlled GPIOs, if they are not required for flash access.
USB_DM and USB_DP	USB controller, supporting Full Speed device and Full/Low Speed host. A 27Ω series termination resistor is required on each pin, but bus pullups and pulldowns are provided internally. These pins can be used as software-controlled GPIOs, if USB is not required.
XIN and XOUT	Connect a crystal to RP2350's crystal oscillator. XIN can also be used as a single-ended CMOS clock input, with XOUT disconnected. The USB bootloader defaults to a 12MHz crystal or 12MHz clock input, but this can be configured via OTP.
RUN	Global asynchronous reset pin. Reset when driven low, run when driven high. If no external reset is required, this pin can be tied directly to IOVDD.
SWCLK and SWDIO	Access to the internal Serial Wire Debug multi-drop bus. Provides debug access to both processors, and can be used to download code.
GND	Single external ground connection, bonded to a number of internal ground pads on the RP2350 die.

Name	Description
IOVDD	Power supply for digital GPIOs, nominal voltage 1.8V to 3.3V
USB_OTP_VDD	Power supply for internal USB Full Speed PHY and OTP storage, nominal voltage 3.3V
ADC_AVDD	Power supply for analogue-to-digital converter, nominal voltage 3.3V
QSPI_IOVDD	Power supply for QSPI IOs, nominal voltage 1.8V to 3.3V
VREG_AVDD	Analogue power supply for internal core voltage regulator, nominal voltage 3.3V
VREG_PGND	Power-ground connection for internal core voltage regulator, tie to ground externally
VREG_LX	Switched-mode output for internal core voltage regulator, connected to external inductor. Max current 200 mA, nominal voltage 1.1V after filtering.
VREG_VIN	Power input for internal core voltage regulator, nominal voltage 2.7V to 5.5V
VREG_FB	Voltage feedback for internal core voltage regulator, connect to filtered VREG output (e.g. to DVDD, if the regulator is used to supply DVDD)
DVDD	Digital core power supply, nominal voltage 1.1V. Must be connected externally, either to the voltage regulator output, or an external board-level power supply.

1.2.3. GPIO functions (Bank 0)

Each individual GPIO pin can be connected to an internal peripheral via the GPIO functions defined below. Some internal peripheral connections appear in multiple places to allow some system level flexibility. SIO, PIO0, PIO1 and PIO2 can connect to all GPIO pins and are controlled by software (or software controlled state machines) so can be used to implement many functions.

Table 3. General Purpose Input/Output (GPIO) Bank 0 Functions

GPIO	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11
0		SPI0 RX	UART0 TX	I2C0 SDA	PWM0 A	SIO	PIO0	PIO1	PIO2	QMI CS1n	USB OVCUR DET	
1		SPI0 CSn	UART0 RX	I2C0 SCL	PWM0 B	SIO	PIO0	PIO1	PIO2	TRACECLK	USB VBUS DET	
2		SPI0 SCK	UART0 CTS	I2C1 SDA	PWM1 A	SIO	PIO0	PIO1	PIO2	TRACEDATA0	USB VBUS EN	UART0 TX
3		SPI0 TX	UART0 RTS	I2C1 SCL	PWM1 B	SIO	PIO0	PIO1	PIO2	TRACEDATA1	USB OVCUR DET	UART0 RX
4		SPI0 RX	UART1 TX	I2C0 SDA	PWM2 A	SIO	PIO0	PIO1	PIO2	TRACEDATA2	USB VBUS DET	
5		SPI0 CSn	UART1 RX	I2C0 SCL	PWM2 B	SIO	PIO0	PIO1	PIO2	TRACEDATA3	USB VBUS EN	
6		SPI0 SCK	UART1 CTS	I2C1 SDA	PWM3 A	SIO	PIO0	PIO1	PIO2		USB OVCUR DET	UART1 TX
7		SPI0 TX	UART1 RTS	I2C1 SCL	PWM3 B	SIO	PIO0	PIO1	PIO2		USB VBUS DET	UART1 RX
8		SPI1 RX	UART1 TX	I2C0 SDA	PWM4 A	SIO	PIO0	PIO1	PIO2	QMI CS1n	USB VBUS EN	
9		SPI1 CSn	UART1 RX	I2C0 SCL	PWM4 B	SIO	PIO0	PIO1	PIO2		USB OVCUR DET	
10		SPI1 SCK	UART1 CTS	I2C1 SDA	PWM5 A	SIO	PIO0	PIO1	PIO2		USB VBUS DET	UART1 TX
11		SPI1 TX	UART1 RTS	I2C1 SCL	PWM5 B	SIO	PIO0	PIO1	PIO2		USB VBUS EN	UART1 RX
12	HSTX	SPI1 RX	UART0 TX	I2C0 SDA	PWM6 A	SIO	PIO0	PIO1	PIO2	CLOCK GPIN0	USB OVCUR DET	
13	HSTX	SPI1 CSn	UART0 RX	I2C0 SCL	PWM6 B	SIO	PIO0	PIO1	PIO2	CLOCK GPOUT0	USB VBUS DET	
14	HSTX	SPI1 SCK	UART0 CTS	I2C1 SDA	PWM7 A	SIO	PIO0	PIO1	PIO2	CLOCK GPIN1	USB VBUS EN	UART0 TX
15	HSTX	SPI1 TX	UART0 RTS	I2C1 SCL	PWM7 B	SIO	PIO0	PIO1	PIO2	CLOCK GPOUT1	USB OVCUR DET	UART0 RX
16	HSTX	SPI0 RX	UART0 TX	I2C0 SDA	PWM0 A	SIO	PIO0	PIO1	PIO2		USB VBUS DET	
17	HSTX	SPI0 CSn	UART0 RX	I2C0 SCL	PWM0 B	SIO	PIO0	PIO1	PIO2		USB VBUS EN	
18	HSTX	SPI0 SCK	UART0 CTS	I2C1 SDA	PWM1 A	SIO	PIO0	PIO1	PIO2		USB OVCUR DET	UART0 TX
19	HSTX	SPI0 TX	UART0 RTS	I2C1 SCL	PWM1 B	SIO	PIO0	PIO1	PIO2	QMI CS1n	USB VBUS DET	UART0 RX
20		SPI0 RX	UART1 TX	I2C0 SDA	PWM2 A	SIO	PIO0	PIO1	PIO2	CLOCK GPIN0	USB VBUS EN	
21		SPI0 CSn	UART1 RX	I2C0 SCL	PWM2 B	SIO	PIO0	PIO1	PIO2	CLOCK GPOUT0	USB OVCUR DET	
22		SPI0 SCK	UART1 CTS	I2C1 SDA	PWM3 A	SIO	PIO0	PIO1	PIO2	CLOCK GPIN1	USB VBUS DET	UART1 TX

GPIO	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11
23		SPI0 TX	UART1 RTS	I2C1 SCL	PWM3 B	SIO	PIO0	PIO1	PIO2	CLOCK GPOUT1	USB VBUS EN	UART1 RX
24		SPI1 RX	UART1 TX	I2C0 SDA	PWM4 A	SIO	PIO0	PIO1	PIO2	CLOCK GPOUT2	USB OVCUR DET	
25		SPI1 CSn	UART1 RX	I2C0 SCL	PWM4 B	SIO	PIO0	PIO1	PIO2	CLOCK GPOUT3	USB VBUS DET	
26		SPI1 SCK	UART1 CTS	I2C1 SDA	PWM5 A	SIO	PIO0	PIO1	PIO2		USB VBUS EN	UART1 TX
27		SPI1 TX	UART1 RTS	I2C1 SCL	PWM5 B	SIO	PIO0	PIO1	PIO2		USB OVCUR DET	UART1 RX
28		SPI1 RX	UART0 TX	I2C0 SDA	PWM6 A	SIO	PIO0	PIO1	PIO2		USB VBUS DET	
29		SPI1 CSn	UART0 RX	I2C0 SCL	PWM6 B	SIO	PIO0	PIO1	PIO2		USB VBUS EN	
GPIOs 30 through 47 are QFN-80 only:												
30		SPI1 SCK	UART0 CTS	I2C1 SDA	PWM7 A	SIO	PIO0	PIO1	PIO2		USB OVCUR DET	UART0 TX
31		SPI1 TX	UART0 RTS	I2C1 SCL	PWM7 B	SIO	PIO0	PIO1	PIO2		USB VBUS DET	UART0 RX
32		SPI0 RX	UART0 TX	I2C0 SDA	PWM8 A	SIO	PIO0	PIO1	PIO2		USB VBUS EN	
33		SPI0 CSn	UART0 RX	I2C0 SCL	PWM8 B	SIO	PIO0	PIO1	PIO2		USB OVCUR DET	
34		SPI0 SCK	UART0 CTS	I2C1 SDA	PWM9 A	SIO	PIO0	PIO1	PIO2		USB VBUS DET	UART0 TX
35		SPI0 TX	UART0 RTS	I2C1 SCL	PWM9 B	SIO	PIO0	PIO1	PIO2		USB VBUS EN	UART0 RX
36		SPI0 RX	UART1 TX	I2C0 SDA	PWM10 A	SIO	PIO0	PIO1	PIO2		USB OVCUR DET	
37		SPI0 CSn	UART1 RX	I2C0 SCL	PWM10 B	SIO	PIO0	PIO1	PIO2		USB VBUS DET	
38		SPI0 SCK	UART1 CTS	I2C1 SDA	PWM11 A	SIO	PIO0	PIO1	PIO2		USB VBUS EN	UART1 TX
39		SPI0 TX	UART1 RTS	I2C1 SCL	PWM11 B	SIO	PIO0	PIO1	PIO2		USB OVCUR DET	UART1 RX
40		SPI1 RX	UART1 TX	I2C0 SDA	PWM8 A	SIO	PIO0	PIO1	PIO2		USB VBUS DET	
41		SPI1 CSn	UART1 RX	I2C0 SCL	PWM8 B	SIO	PIO0	PIO1	PIO2		USB VBUS EN	
42		SPI1 SCK	UART1 CTS	I2C1 SDA	PWM9 A	SIO	PIO0	PIO1	PIO2		USB OVCUR DET	UART1 TX
43		SPI1 TX	UART1 RTS	I2C1 SCL	PWM9 B	SIO	PIO0	PIO1	PIO2		USB VBUS DET	UART1 RX
44		SPI1 RX	UART0 TX	I2C0 SDA	PWM10 A	SIO	PIO0	PIO1	PIO2		USB VBUS EN	

GPIO	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11
45		SPI1 CSn	UART0 RX	I2C0 SCL	PWM10 B	SIO	PIO0	PIO1	PIO2		USB OVCUR DET	
46		SPI1 SCK	UART0 CTS	I2C1 SDA	PWM11 A	SIO	PIO0	PIO1	PIO2		USB VBUS DET	UART0 TX
47		SPI1 TX	UART0 RTS	I2C1 SCL	PWM11 B	SIO	PIO0	PIO1	PIO2	QMI CS1n	USB VBUS EN	UART0 RX

Table 4. GPIO bank 0 function descriptions

Function Name	Description
SPIx	Connect one of the internal PL022 SPI peripherals to GPIO
UARTx	Connect one of the internal PL011 UART peripherals to GPIO
I2Cx	Connect one of the internal DW I2C peripherals to GPIO
PWMx A/B	Connect a PWM slice to GPIO. There are twelve PWM slices, each with two output channels (A/B). The B pin can also be used as an input, for frequency and duty cycle measurement.
SIO	Software control of GPIO, from the single-cycle IO (SIO) block. The SIO function (F5) must be selected for the processors to *drive* a GPIO, but the input is always connected, so software can check the state of GPIOs at any time.
PIOx	Connect one of the programmable IO blocks (PIO) to GPIO. PIO can implement a *wide* variety of interfaces, and has its own internal pin mapping hardware, allowing flexible placement of digital interfaces on bank 0 GPIOs. The PIO function (F6, F7, F8) must be selected for PIO to *drive* a GPIO, but the input is always connected, so the PIOs can always see the state of all pins.
HSTX	Connect the high-speed transmit peripheral (HSTX) to GPIO
CLOCK GPINx	General purpose clock inputs. Can be routed to a number of internal clock domains on RP2350, e.g. to provide a 1Hz clock for the AON Timer, or can be connected to an internal frequency counter.
CLOCK GPOUTx	General purpose clock outputs. Can drive a number of internal clocks (including PLL outputs) onto GPIOs, with optional integer divide.
TRACECLK, TRACEDATAx	CoreSight TPIU execution trace output from Cortex-M33 processors (Arm-only)
USB OVCUR DET/VBUS DET/VBUS EN	USB power control signals to/from the internal USB controller
QMI CS1n	Auxiliary chip select for QSPI bus, to allow execute-in-place from an additional flash or PSRAM device

NOTE

GPIOs 0 through 29 are available in all package variants. GPIOs 30 through 47 are available only in QFN-80 (RP2350B) package.

NOTE

Analogue input is available on GPIOs 26 through 29 in the QFN-60 package (RP2350A), for a total of four inputs, and on GPIOs 40 through 47 in the QFN-80 package (RP2350B), for a total of eight inputs.

1.2.4. GPIO functions (Bank 1)

GPIO functions are also available on the six dedicated QSPI pins, which are usually used for flash execute-in-place, and on the USB DP/DM pins. These may become available for general-purpose use depending on the use case, for example, QSPI pins may not be needed for code execution if RP2350 is booting from internal OTP storage, or being controlled externally via SWD.

Table 5. GPIO Bank 1 Functions

Pin	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11
USB DP			UART1 TX	I2C0 SDA		SIO						

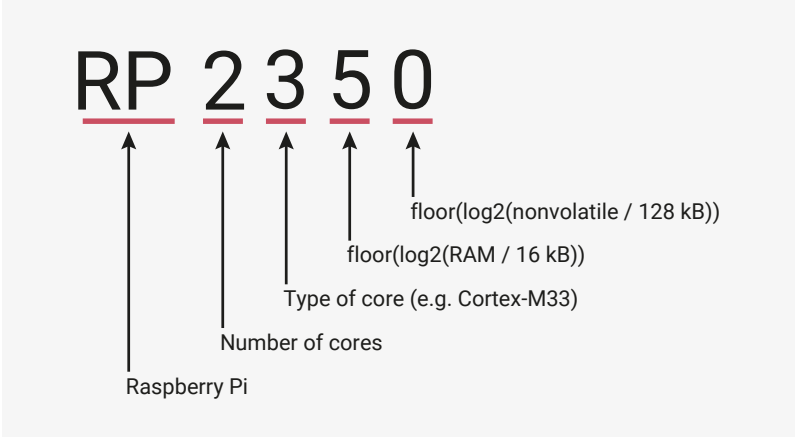
Pin	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11
USB DM			UART1 RX	I2C0 SCL		SIO						
QSPI SCK	QMI SCK		UART1 CTS	I2C1 SDA		SIO						UART1 TX
QSPI CSn	QMI CS0n		UART1 RTS	I2C1 SCL		SIO						UART1 RX
QSPI SD0	QMI SD0		UART0 TX	I2C0 SDA		SIO						
QSPI SD1	QMI SD1		UART0 RX	I2C0 SCL		SIO						
QSPI SD2	QMI SD2		UART0 CTS	I2C1 SDA		SIO						UART0 TX
QSPI SD3	QMI SD3		UART0 RTS	I2C1 SCL		SIO						UART0 RX

Table 6. GPIO bank 1 function descriptions

Function Name	Description
UARTx	Connect one of the internal PL011 UART peripherals to GPIO
I2Cx	Connect one of the internal DW I2C peripherals to GPIO
SIO	Software control of GPIO, from the single-cycle IO (SIO) block. The SIO function (F5) must be selected for the processors to *drive* a GPIO, but the input is always connected, so software can check the state of GPIOs at any time.
QMI	QSPI memory interface peripheral, used for execute-in-place from external QSPI flash or PSRAM memory devices.

1.3. Why is the chip called RP2350?

Figure 4. An explanation for the name of the RP2350 chip.



The post-fix numeral on RP2350 comes from the following,

1. Number of processor cores
   - **2** indicates a dual-core system
2. Loosely which type of processor
   - **3** indicates Cortex-M33 or Hazard3
3. Internal memory capacity: $\log_2\lfloor\frac{RAM}{16kB}\rfloor$
   - **5** indicates at least 2^5 × 16 kB = 512 kB
   - RP2350 has 520 kB of main system SRAM
4. Internal storage capacity: $\log_2\lfloor\frac{nonvolatile}{128kB}\rfloor$ (or 0 if no onboard nonvolatile storage)

- RP235**0** uses external flash
- RP235**4** has 2^4 × 128 kB = 2 MB of internal flash

1.4. Version History

Table 7 lists versions of RP2350. Later versions fix bugs in earlier versions. For more information about the changes made between versions, see Appendix C. Also refer to Product Change Notification (PCN) 28.

Table 7. RP2350 version history

Version	Use
A0	Internal development
A1	Internal development
A2	Initial release
A3	Internal development, samples, and limited production
A4	Production version

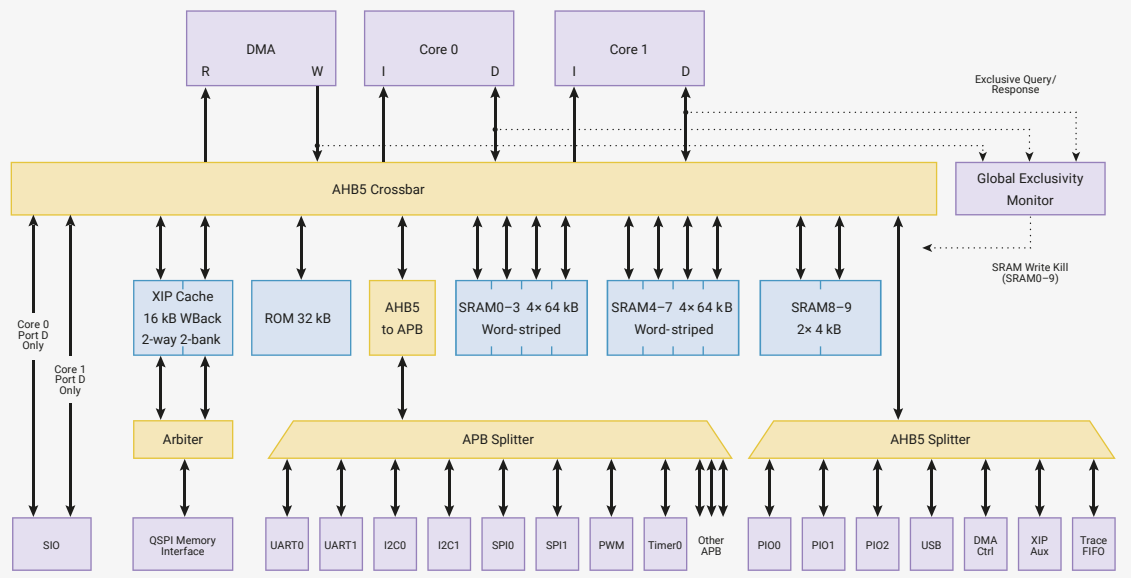
Chapter 2. System bus

2.1. Bus fabric

The RP2350 bus fabric routes addresses and data across the chip.

Figure 5 shows the high-level structure of the bus fabric. The main AHB5 crossbar routes addresses and data between its 6 upstream ports and 17 downstream ports, with up to six bus transfers taking place each cycle. All data paths are 32 bits wide. Memories connect to multiple dedicated ports on the main crossbar, for the best possible memory bandwidth. High-bandwidth AHB peripherals share a port on the crossbar. An APB bridge provides access to system control registers and lower-bandwidth peripherals. The SIO peripherals are accessed via a dedicated path from each processor.

Figure 5. RP2350 bus fabric overview.



The bus fabric connects 6 AHB5 managers, i.e. bus ports which generate addresses:

- Core 0: Instruction port (instruction fetch), and Data port (load/store access)
- Core 1: Instruction port (instruction fetch), and Data port (load/store access)
- DMA controller: Read port, Write port

The following 13 downstream ports are symmetrically accessible from all 6 upstream ports:

- Boot ROM (1 port)
- XIP (2 ports, striped)
- SRAM (10 ports, striped)

Additionally, the following 2 ports are accessible for processor load/store and DMA read/write only:

- 1 shared port for fast AHB5 peripherals: PIO0, PIO1, PIO2, USB, DMA control registers, XIP DMA FIFOs, HSTX FIFO, CoreSight trace DMA FIFO
- 1 port for the APB bridge, to all APB peripherals and control registers

NOTE

Instruction fetch from peripherals is *physically disconnected*, to avoid this IDAU-Exempt region ever becoming both Non-secure-writable and Secure-executable. This includes USB RAM, OTP and boot RAM. See Section 10.2.2.

The SIO block, which was connected to the Cortex-M0+ IOPORT on RP2040, provides two AHB ports, each dedicated to load/store access from one core.

The six managers can access any six *different* crossbar ports simultaneously. So, at a system clock of 150 MHz, the maximum sustained bus bandwidth is 3.6 GB/s.

2.1.1. Bus priority

The main AHB5 crossbar implements a two-level bus priority scheme. Priority levels are configured separately for core 0, core 1, DMA read and DMA write, using the BUS_PRIORITY register in the BUSCTRL register block.

When a downstream subordinate receives multiple simultaneous access requests, the port serves high-priority (priority level 1) managers before serving any requests from low-priority (priority 0) managers. If all requests come from managers with the same priority level, the port applies a round-robin tie break, granting access to each manager in turn.

NOTE

Priority arbitration only applies when multiple managers attempt to access the **same** subordinate on the same cycle. When multiple managers access different subordinates, e.g. different SRAM banks, the requests proceed simultaneously.

A subordinate with zero wait states can be accessed once per system clock cycle. When accessing a subordinate with zero wait states (e.g. SRAM), high-priority managers never experience delays caused by accesses from low-priority managers. This *guarantees* latency and throughput for real-time use cases. However, it also means that low-priority managers may stall until there is a free cycle.

2.1.2. Bus security filtering

Every point where the fabric connects to a downstream AHB or APB peripheral is interposed by a bus security filter, which enforces the following access control lists as defined by the ACCESSCTRL registers (Section 10.6):

- A list of who can access the port: core 0, core 1, DMA, debugger
- A list of the security states from which the port can be accessed: the four combinations of Secure/Non-secure and Privileged/Unprivileged.

Accesses that fail either check are prevented from accessing the downstream port, and return a bus error upstream.

There are three exceptions, which do not implement bus security filters because they implement their own security filtering internally:

- The ACCESSCTRL block itself, which is always world-readable, but filters writes on security and privilege
- Boot RAM, which is hardwired to Secure access only
- The single-cycle IO subsystem (SIO), which is internally banked over Secure and Non-secure

The Cortex-M Private Peripheral Bus (PPB) registers also lack ACCESSCTRL permissions because they are internal to the processors, not accessed through the system bus. The PPB registers are internally banked over Secure and Non-secure.

2.1.3. Atomic register access

Each peripheral register block is allocated 4 kB of address space, with registers accessed using one of 4 methods, selected by address decode.

- `Addr + 0x0000` : normal read write access
- `Addr + 0x1000` : atomic XOR on write
- `Addr + 0x2000` : atomic bitmask set on write
- `Addr + 0x3000` : atomic bitmask clear on write

This allows software to modify individual fields of a control register without performing a read-modify-write sequence. Instead, the peripheral itself modifies its contents in-place. Without this capability, it is difficult to safely access IO registers when an interrupt service routine is concurrent with code running in the foreground, or when the two processors run code in parallel.

The four atomic access aliases occupy a total of 16 kB. Native atomic writes take the same number of clock cycles as normal writes. Most peripherals on RP2350 provide this functionality natively, but some peripherals (I2C, UART, SPI and SSI) add this functionality using a bus interposer. The bus interposer translates upstream atomic writes into downstream read-modify-write sequences at the boundary of the peripheral, at the cost of additional clock cycles. Atomic writes that use a bus interposer take two additional clock cycles compared to normal writes.

The following registers do not support atomic register access:

- SIO (Section 3.1), though some individual registers (for example, GPIO) have set, clear, and XOR aliases.
- Any register accessed through the self-hosted CoreSight window, including Arm Mem-APs and the RISC-V Debug Module.
- Standard Arm control registers on the Cortex-M33 private peripheral bus (PPB), except for Raspberry Pi-specific registers on the EPPB.
- OTP programming registers accessed through the SBPI bridge.

2.1.4. APB bridge

The APB bridge provides an interface between the high-speed main AHB5 interconnect and the lower-bandwidth peripherals. Unlike the AHB5 fabric, which offers zero-wait-state accesses everywhere, APB accesses take a minimum of three cycles for a read, and four cycles for a write.

As a result, the throughput of the APB portion of the bus fabric is lower than the AHB5 portion. However, there is more than sufficient bandwidth to saturate the APB serial peripherals.

The following APB ports contain asynchronous bus crossings, which insert additional stall cycles on top of the typical cost of a read or write in the APB bridge:

- ADC
- HSTX_CTRL
- OTP
- POWMAN

The APB bridge implements a fixed timeout for stalled downstream transfers. The downstream bus may stall indefinitely, such as when accessing an asynchronous bus crossing when the destination clock is stopped, or deadlock conditions when accessing system APB registers through Mem-APs in the self-hosted debug window (Section 3.5.6). When an APB transfer exceeds 65,535 cycles the APB bridge abandons the transfer and returns a bus fault. This keeps the system bus available so that software or the debugger can diagnose the reason for the overly long transfer.

2.1.5. Narrow IO register writes

The majority of memory-mapped IO registers on RP2350 ignore the width of bus read/write accesses. They treat all writes as though they were 32 bits in size. This means software cannot use byte or halfword writes to modify part of an IO register: any write to an address where the 30 address MSBs match the register address affects the contents of the entire register.

To update part of an IO register without a read-modify-write sequence, the best solution on RP2350 is atomic set/clear/XOR (see Section 2.1.3). This is more flexible than byte or halfword writes, as any combination of fields can be updated in one operation.

Upon a 8-bit or 16-bit write (such as a `strb` instruction on the Cortex-M33), the narrow value is replicated multiple times across the 32-bit data bus, so that it is broadcast to all 8-bit or 16-bit segments of the destination register:

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/system/narrow_io_write/narrow_io_write.c Lines 19 - 62

```
19 int main() {
20     stdio_init_all();
21
22     // We'll use WATCHDOG_SCRATCH0 as a convenient 32 bit read/write register
23     // that we can assign arbitrary values to
24     io_rw_32 *scratch32 = &watchdog_hw->scratch[0];
25     // Alias the scratch register as two halfwords at offsets +0x0 and +0x2
26     volatile uint16_t *scratch16 = (volatile uint16_t *) scratch32;
27     // Alias the scratch register as four bytes at offsets +0x0, +0x1, +0x2, +0x3:
28     volatile uint8_t *scratch8 = (volatile uint8_t *) scratch32;
29
30     // Show that we can read/write the scratch register as normal:
31     printf("Writing 32 bit value\n");
32     *scratch32 = 0xdeadbeef;
33     printf("Should be 0xdeadbeef: 0x%08x\n", *scratch32);
34
35     // We can do narrow reads just fine -- IO registers treat this as a 32 bit
36     // read, and the processor/DMA will pick out the correct byte lanes based
37     // on transfer size and address LSBs
38     printf("\nReading back 1 byte at a time\n");
39     // Little-endian!
40     printf("Should be ef be ad de: %02x ", scratch8[0]);
41     printf("%02x ", scratch8[1]);
42     printf("%02x ", scratch8[2]);
43     printf("%02x\n", scratch8[3]);
44
45     // Byte writes are replicated four times across the 32-bit bus, and IO
46     // registers usually sample the entire write bus.
47     printf("\nWriting 8 bit value 0xa5 at offset 0\n");
48     scratch8[0] = 0xa5;
49     // Read back the whole scratch register in one go
50     printf("Should be 0xa5a5a5a5: 0x%08x\n", *scratch32);
51
52     // The IO register ignores the address LSBs [1:0] as well as the transfer
53     // size, so it doesn't matter what byte offset we use
54     printf("\nWriting 8 bit value at offset 1\n");
55     scratch8[1] = 0x3c;
56     printf("Should be 0x3c3c3c3c: 0x%08x\n", *scratch32);
57
58     // Halfword writes are also replicated across the write data bus
59     printf("\nWriting 16 bit value at offset 0\n");
60     scratch16[0] = 0xf00d;
61     printf("Should be 0xf00df00d: 0x%08x\n", *scratch32);
62 }
```

To disable this behaviour on RP2350, set bit `14` of the address by accessing the peripheral at an offset of `+0x4000`. This

causes invalid byte lanes to be driven to zero, rather than being driven with replicated data. In some situations, such as DMA of 8-bit values to the PWM peripheral, the default replication behaviour is not desirable.

2.1.6. Global Exclusive Monitor

The Global Exclusive Monitor enables standard Arm and RISC-V atomic instructions to safely access shared variables in SRAM from both cores. This underpins software libraries for manipulating shared variables, such as `stdatomic.h` in C11. For detailed rules governing the monitor's operation, see the Armv8-M Architecture Reference Manual.

Arm describes exclusive monitor interactions in terms of a *processing element*, PE, which performs a sequence of bus accesses. For RP2350 purposes, this is one AHB5 manager out of the following three: core 0 load/store, core 1 load/store, and DMA write. The DMA does not itself perform exclusive accesses, but its writes are monitored with respect to exclusive sequences on either processor. No distinction is made between debugger and non-debugger accesses from a processor.

The monitor observes all transfers on SRAM initiated by the DMA write and processor load/store ports, and pays particular attention to two types of transfer:

- AHB5 *exclusive reads*: Arm `ldrex*` instructions, RISC-V `lr.w` instructions, and the read phase of RISC-V AMOs (The Hazard3 cores on RP2350 implement AMOs as an exclusive read/write pair that retries until the write succeeds).
- AHB5 *exclusive writes*: Arm `strex*` instructions, RISC-V `sc.w` instructions, and the writeback phase of RISC-V AMOs

Based on these observations, the monitor enforces that an atomic read-modify-write sequence (formed of an exclusive read followed by a successful exclusive write by the same PE) is not interleaved with another PE's successful write (exclusive or not) to the same reservation granule. A reservation granule is any 16-byte, naturally aligned area of SRAM. An exclusive write succeeds when all of the following are true:

- It is preceded by an exclusive read by the same PE
- No other exclusive writes were performed by this PE since that exclusive read
- The exclusive read was to the same reservation granule
- The exclusive read was of the same size (byte/halfword/word)
- The exclusive read was from the same security and privilege state
- No other PEs successfully wrote to the same granule since that exclusive read

If the above conditions are *not* met, the Global Exclusive Monitor shoots down the exclusive write before SRAM can commit the write data. The failure is reported to the originating PE, for example by a non-zero return value from an Arm `strex` instruction.

This implementation of the Armv8-M Global Exclusive Monitor also meets the requirements for RISC-V `lr/sc` and `amo*` instructions, with the caveat that the *RsrvEventual* PMA is not supported. (In practice, whilst it is quite easy to come up with contrived examples of starvation such as the DMA writing to a shared variable on every single cycle, bounded LR/SC and AMO sequences will generally complete quickly.)

CAUTION

Secure software should avoid shared variables in Non-secure-accessible memory. Such variables are vulnerable to deliberate starvation from exclusive accesses by repeatedly performing non-exclusive writes.

Exclusive accesses are only supported on SRAM. The system treats exclusive accesses to other memory regions as normal reads and writes, reporting exclusivity failure to the originating PE, for example by a non-zero return value from an Arm `strex` instruction.

2.1.6.1. Implementation-defined monitor behaviour

The Armv8-M Architecture Reference Manual leaves several aspects of the Global Exclusive Monitor up to the implementation. For completeness, the RP2350 implementation defines them as follows:

- The reservation granule size is fixed at 16 bytes
- A single reservation is tracked per PE
- The Arm `clrex` instruction does not affect global monitor state
- Any exclusive write by a PE clears that PE's global reservation
- A non-exclusive write by a PE does *not* clear that PE's global reservation, no matter the address

Only the following updates a PE's reservation tag, setting its reservation state to **Exclusive**:

- An exclusive read on SRAM

Only the following changes a PE's reservation state from **Exclusive** to **Open**:

- A *successful* exclusive write from another PE to this PE's reservation
- A non-exclusive write from another PE to this PE's reservation
- Any exclusive write by this PE
- An exclusive read by this PE, *not* on SRAM

A reservation granule can span multiple SRAM banks, so multiple operations on the same reservation granule may complete on the same cycle. This can result in the following problematic situations:

- Multiple exclusive writes to the same reservation granule, reserved on each PE: in this case the lowest-numbered PE succeeds (in the order DMA < core 0 < core 1), and all others fail.
- A mixture of non-exclusive and exclusive writes to the same reservation granule on the same cycle: in this case, the exclusive writes fail.
- One PE *x* can write to a reservation granule on the same cycle that another PE *y* attempts to reserve the *same* reservation granule via exclusive load: in this case, *y*'s reservation is granted (i.e. the write takes place logically before the load).
- One PE *x* can write to a reservation granule reserved by another PE *y*, on the same cycle that PE *y* makes a new reservation on a *different* reservation granule: in this case, again, *y*'s reservation is granted.

These rules can be summarised by a *logical* ordering of all possible events on a reservation granule that can occur on the same cycle: first all normal writes in arbitrary order, then all exclusive writes in ascending PE order (DMA, core 0, core 1), then all loads in arbitrary order.

2.1.6.2. Regions without exclusives support

The Global Exclusive monitor only supports exclusive transactions on certain address ranges. The main system SRAM supports exclusive transactions throughout its entire range: `0x20000000` through `0x20082000`. Within ranges that support exclusive transactions, the Global Exclusive monitor:

- Tracks exclusive sequences across all participating PEs.
- Drives the exclusive success/failure response correctly based on the observed ordering.
- Shoots down failing exclusive writes so that they have no effect.

Exclusive transactions aren't supported outside of this range; all exclusive accesses report exclusive failure (both exclusive reads and exclusive writes), and exclusive writes aren't suppressed.

Outside of regions with exclusive transaction support, load/store exclusive loops run forever while still affecting SRAM contents. This applies to both Arm processors performing exclusive reads/writes and RISC-V processors performing `lr.w`/`sc.w` instructions. However, an `amo*.w` instruction on Hazard3 will result in a Store/AMO Fault, as the hardware

detects the failed exclusive read and bails out to avoid an infinite loop.

It is recommended not to perform exclusive accesses on regions outside of main SRAM. Shared variables outside of main SRAM can be protected using either lock variables in main SRAM, the SIO spinlocks, or a locking protocol that does not require exclusive accesses, such as a lock-free queue.

2.1.7. Bus performance counters

Bus performance counters automatically count accesses to the main AHB5 crossbar arbiters. These counters can help diagnose high-traffic performance issues.

There are four performance counters, starting at PERFCTR0. Each is a 24-bit saturating counter. Counter values can be read from `BUSCTRL_PERFCTRx` and cleared by writing any value to `BUSCTRL_PERFCTRx`. Each counter can count one of the 20 available events at a time, as selected by `BUSCTRL_PERFSELx`. For more information, see Section 12.15.4.

2.2. Address map

The address map for the device is split into sections as shown in Table 8. Details are shown in the following sections. Unmapped address ranges raise a bus error when accessed.

Each link in the left-hand column of Table 8 goes to a detailed address map for that address range. The detailed address maps have a link for each address to the relevant documentation for that address.

Rough address decode is first performed on bits `31:28` of the address:

Table 8. Address Map Summary

Bus Segment	Base Address
ROM	`0x00000000`
XIP	`0x10000000`
SRAM	`0x20000000`
APB Peripherals	`0x40000000`
AHB Peripherals	`0x50000000`
Core-local Peripherals (SIO)	`0xd0000000`
Cortex-M33 private registers	`0xe0000000`

2.2.1. ROM

ROM is accessible to DMA, processor load/store, and processor instruction fetch. It is located at address zero, which is the starting point for both Arm processors when the device is reset.

Table 9. Address map for ROM bus segment

Bus Endpoint	Base Address
ROM_BASE	`0x00000000`

2.2.2. XIP

XIP is accessible to DMA, processor load/store, and processor instruction fetch. This address range contains various mirrors of a 64 MB space which is mapped to external memory devices. On RP2350 the lower 32 MB is occupied by the QSPI Memory Interface (QMI), and the remainder is reserved. QMI controls are in the APB register section.

Table 10. Address map for XIP bus segment

Bus Endpoint	Base Address
XIP_BASE	0x10000000
XIP_NOCACHE_NOALLOC_BASE	0x14000000
XIP_MAINTENANCE_BASE	0x18000000
XIP_NOCACHE_NOALLOC_NOTRANSLATE_BASE	0x1c000000

NOTE

XIP_SRAM_BASE no longer exists as a separate address range. Cache-as-SRAM is now achieved by pinning cache lines within the cached XIP address space.

2.2.3. SRAM

SRAM is accessible to DMA, processor load/store, and processor instruction fetch.

SRAM0-3 and SRAM4-7 are always striped on bits 3:2 of the address:

Table 11. Address map for SRAM bus segment, SRAM0-7 (striped)

Bus Endpoint	Base Address
SRAM_BASE	0x20000000
SRAM_STRIPED_BASE	0x20000000
SRAM0_BASE	0x20000000
SRAM4_BASE	0x20040000
SRAM_STRIPED_END	0x20080000

There are two striped regions, each 256 kB in size, and each striped over 4 SRAM banks. SRAM0-3 are in the SRAM0 power domain, and SRAM4-7 are in the SRAM1 power domain.

SRAM 8-9 are always non-striped:

Table 12. Address map for SRAM bus segment, SRAM8-9 (non-striped)

Bus Endpoint	Base Address
SRAM8_BASE	0x20080000
SRAM9_BASE	0x20081000
SRAM_END	0x20082000

These smaller blocks of SRAM are useful for hoisting high-bandwidth data structures like the processor stacks. They are in the SRAM1 power domain.

2.2.4. APB registers

APB peripheral registers are accessible to processor load/store and DMA only. Instruction fetch will always fail.

The APB peripheral segment provides access to control and configuration registers, as well as data access for lower-bandwidth peripherals. APB writes cost a minimum of four cycles, and APB reads a minimum of three.

Table 13. Address map for APB bus segment

Bus Endpoint	Base Address
SYSINFO_BASE	0x40000000
SYSCFG_BASE	0x40008000

Bus Endpoint	Base Address
CLOCKS_BASE	0x40010000
PSM_BASE	0x40018000
RESETS_BASE	0x40020000
IO_BANK0_BASE	0x40028000
IO_QSPI_BASE	0x40030000
PADS_BANK0_BASE	0x40038000
PADS_QSPI_BASE	0x40040000
XOSC_BASE	0x40048000
PLL_SYS_BASE	0x40050000
PLL_USB_BASE	0x40058000
ACCESSCTRL_BASE	0x40060000
BUSCTRL_BASE	0x40068000
UART0_BASE	0x40070000
UART1_BASE	0x40078000
SPI0_BASE	0x40080000
SPI1_BASE	0x40088000
I2C0_BASE	0x40090000
I2C1_BASE	0x40098000
ADC_BASE	0x400a0000
PWM_BASE	0x400a8000
TIMER0_BASE	0x400b0000
TIMER1_BASE	0x400b8000
HSTX_CTRL_BASE	0x400c0000
XIP_CTRL_BASE	0x400c8000
XIP_QMI_BASE	0x400d0000
WATCHDOG_BASE	0x400d8000
BOOTRAM_BASE	0x400e0000
ROSC_BASE	0x400e8000
TRNG_BASE	0x400f0000
SHA256_BASE	0x400f8000
POWMAN_BASE	0x40100000
TICKS_BASE	0x40108000
OTP_BASE	0x40120000
OTP_DATA_BASE	0x40130000
OTP_DATA_RAW_BASE	0x40134000
OTP_DATA_GUARDED_BASE	0x40138000

Bus Endpoint	Base Address
OTP_DATA_RAW_GUARDED_BASE	0x4013c000
CORESIGHT_PERIPH_BASE	0x40140000
CORESIGHT_ROMTABLE_BASE	0x40140000
CORESIGHT_AHB_AP_CORE0_BASE	0x40142000
CORESIGHT_AHB_AP_CORE1_BASE	0x40144000
CORESIGHT_TIMESTAMP_GEN_BASE	0x40146000
CORESIGHT_ATB_FUNNEL_BASE	0x40147000
CORESIGHT_TPIU_BASE	0x40148000
CORESIGHT_CTI_BASE	0x40149000
CORESIGHT_APB_AP_RISCV_BASE	0x4014a000
GLITCH_DETECTOR_BASE	0x40158000
TBMAN_BASE	0x40160000

2.2.5. AHB registers

AHB peripheral registers are accessible to processor load/store and DMA only. Instruction fetch will always fail.

The AHB peripheral segment provides access to higher-bandwidth peripherals. The minimum read/write cost is one cycle, and peripherals may insert up to one wait state.

Table 14. Address map for AHB peripheral bus segment

Bus Endpoint	Base Address
DMA_BASE	0x50000000
USBCTRL_BASE	0x50100000
USBCTRL_DPRAM_BASE	0x50100000
USBCTRL_REGS_BASE	0x50110000
PIO0_BASE	0x50200000
PIO1_BASE	0x50300000
PIO2_BASE	0x50400000
XIP_AUX_BASE	0x50500000
HSTX_FIFO_BASE	0x50600000
CORESIGHT_TRACE_BASE	0x50700000

2.2.6. Core-local peripherals (SIO)

SIO is accessible to processor load/store only. It contains registers which need single-cycle access from both cores concurrently, such as the GPIO registers. Access is always zero-wait-state.

Table 15. Address map for SIO bus segment

Bus Endpoint	Base Address
SIO_BASE	0xd0000000
SIO_NONSEC_BASE	0xd0020000

2.2.7. Cortex-M33 private peripherals

The PPB is accessible to processor load/store only.

The PPB region contains standard control registers defined by Arm, Non-secure aliases of some of those registers, and a handful of other core-local registers defined by Raspberry Pi (the EPPB).

These addresses are only accessible to Arm processors: RISC-V processors will return a bus fault.

Table 16. Address map for PPB bus segment

Bus Endpoint	Base Address
PPB_BASE	0xe0000000
PPB_NONSEC_BASE	0xe0020000
EPPB_BASE	0xe0080000

Chapter 3. Processor subsystem

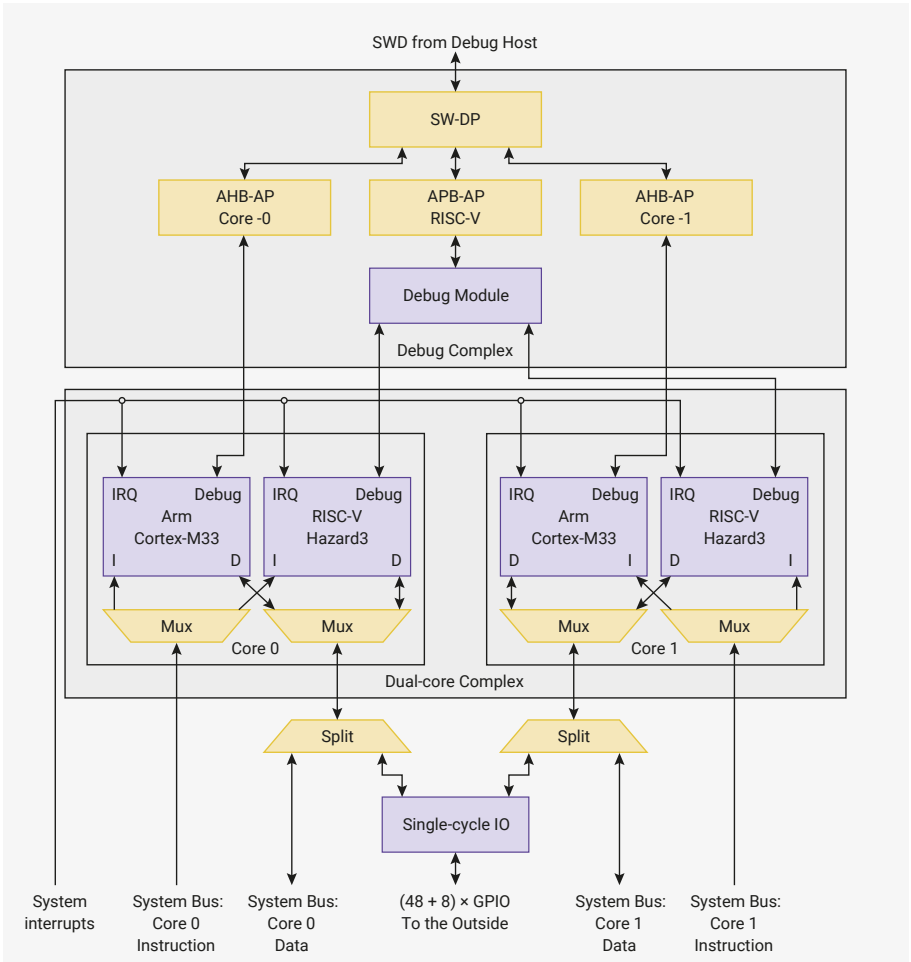


Figure 6. The RP2350 processor subsystem connects two processors to the system bus, peripheral interrupts, GPIOs, and a Serial Wire Debug (SWD) connection from an external debug host. It also contains closely-coupled peripherals, and peripherals used for synchronisation and communication, which are collectively referred to as the single-cycle IO subsystem (SIO).

RP2350 is a symmetric dual-core system. Two cores operate simultaneously and independently, offering high processing throughput and the ability to route interrupts to different cores to improve throughput and latency of interrupt handling. The two cores have a symmetric view of the system bus; all memory resources on RP2350 are accessible equally on both cores, with the same performance.

Each core has a pair of 32-bit AHB5 links to the system bus. One is used exclusively for instruction fetch, the other exclusively for load or store instructions and debugger access. Each core can perform one instruction fetch *and* one load or store access per cycle, provided there are no conflicts on the downstream bus ports.

There are two sockets for cores to attach to the system bus, referred to as **core 0** and **core 1** throughout this datasheet. (They may synonymously be referred to as core0, core1, proc0 and proc1 in register documentation.) The processor plugged into each socket is selectable at boot time:

- A Cortex-M33 processor, implementing the Armv8-M Main instruction set, plus extensions
- A Hazard3 processor, implementing the RV32IMAC instruction set, plus extensions

Cortex-M33 is the default option. Whichever processor is unused is held in reset with its clock gated at the top level. Unused processors use zero dynamic power. See Section 3.9 for information about the architecture selection hardware.

The two Cortex-M33 instances are identical. They are configured with the Security, DSP and FPU extensions, as well as 8× SAU regions, 8× Secure MPU regions and 8× Non-secure MPU regions. Section 3.7 documents the Cortex-M33 processor as well as the specific configuration used on RP2350. The two Hazard3 instances are also identical to one another; see Section 3.8 for the features and operation of the Hazard3 processors.

The Cortex-M33 implementation of the Armv8-M Security extension (also known as TrustZone-M) isolates trusted and untrusted software running on-device. RP2350 extends the strict partitioning of the Arm Secure and Non-secure states throughout the system, including the ability to assign peripherals, GPIOs and DMA channels to each security domain. See Section 10.2 for a high-level overview of Armv8-M Security extension features in the context of the RP2350 security architecture.

Not shown on Figure 6 are the coprocessors for the Cortex-M33. These are closely coupled to the core, offering a transfer rate of 64 bits per cycle in and out of the Arm register file. You may consider them to be inside the Cortex-M33 block on the diagram. RP2350 equips each Cortex-M33 with the following coprocessors:

- Coprocessor `0`: GPIO coprocessor (GPIOC), described in Section 3.6.1
- Coprocessors `4` and `5`: Secure and Non-secure instances of the double-precision coprocessor (DCP), described in Section 3.6.2
- Coprocessor `7`: redundancy coprocessor (RCP), described in Section 3.6.3

An external debug host can access both cores over a Serial Wire Debug (SWD) bus. The host can:

- run, halt and reset the cores
- inspect internal core state such as registers
- access memory from the core's point of view
- load code onto the device and run it

Section 3.5 describes the debug hardware in addition to the instruction trace hardware available on the Arm processors.

Peripherals throughout the system assert **interrupt requests** (IRQs) to demand attention from the processors. For example, a UART peripheral asserts its interrupt when it has received a character, so the processor can collect it from the receive FIFO. All interrupts route to both cores, and the core's internal interrupt controller selects the interrupt signals it wishes to subscribe to. Section 3.2 defines the system-level IRQ numbering as well as details of the Arm non-maskable interrupt (NMI).

The event signals described in Section 3.3 are a mechanism for processors to sleep when waiting for other processors in the system to complete a task or free up some resource. Each processor sees events emitted by the other processor. They also see exclusivity events generated by the Global Exclusive Monitor described in Section 2.1.6, which is the piece of hardware that allows the processors to safely manipulate shared variables using atomic read-modify-write sequences.

3.1. SIO

The Single-cycle IO subsystem (SIO) contains peripherals that require low-latency, deterministic access from the processors. It is accessed via the AHB Fabric. The SIO has a dedicated bus interface for each processor, as shown in Figure 7.

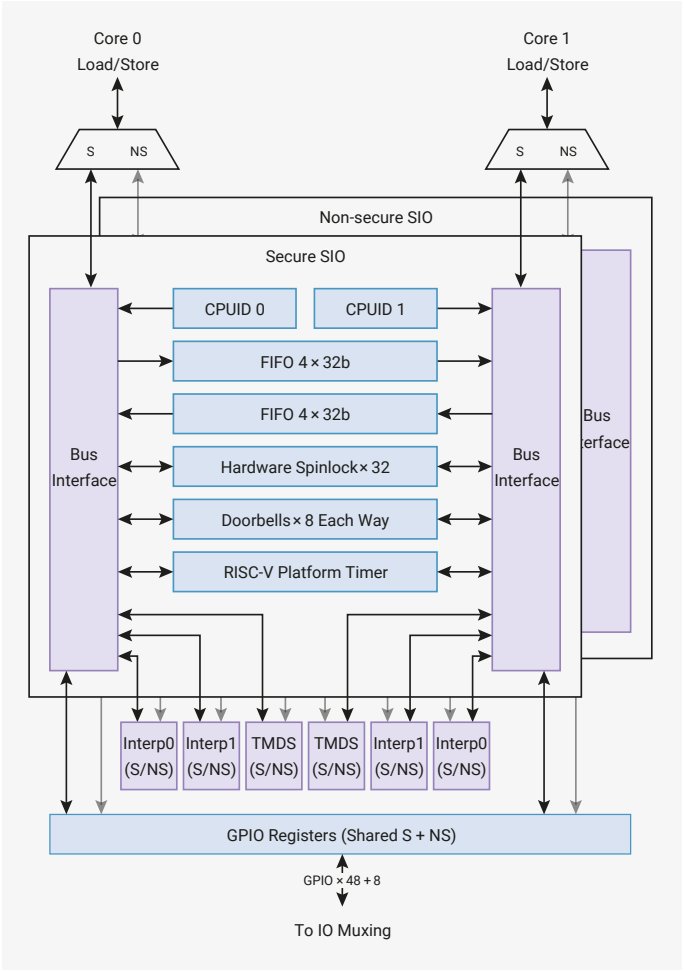


Figure 7. The single-cycle IO block contains registers which processors must access quickly. FIFOs, doorbells and spinlocks support message passing and synchronisation between the two cores. The shared GPIO registers provide fast, direct access to GPIO-capable pins. Interpolators can accelerate common software tasks. Most SIO hardware is banked (duplicated) for Secure and Non-secure access. Grey arrows show bus connections for Non-secure access.

The SIO contains:

- CPUID registers which read as 0/1 on core 0/1 (Section 3.1.2)
- Mailbox FIFOs for passing ordered messages between cores (Section 3.1.5)
- Doorbells for interrupting the opposite core on cumulative and unordered events (Section 3.1.6)
- Hardware spinlocks for implementing critical sections without using exclusive bus accesses (Section 3.1.4)
- Interpolators (Section 3.1.10) and TMDS encoders (Section 3.1.9)
- Standard RISC-V 64-bit platform timer (Section 3.1.8) which is usable by both Arm and RISC-V software
- GPIO registers for fast software bitbanging (Section 3.1.3), with shared access from both cores

Most SIO hardware is duplicated for Secure/Non-secure access. Non-secure access to the FIFO registers will see a physically different FIFO than Secure access to the same address, so that messages belonging to Secure and Non-secure software are not mixed: Section 3.1.1 describes this Secure/Non-secure banking in more detail.

3.1.1. Secure and Non-secure SIO

To allow isolation of Secure and Non-secure software, whilst keeping a consistent programming model for software written to run in either domain, the SIO is duplicated into a Secure and a Non-secure bank. Most hardware is duplicated between the two banks, including:

- Mailbox FIFOs
- Doorbell registers

- Interrupt outputs to processors
- Spinlocks

For example, Non-secure code on core 0 can pass messages to Non-secure code on core 1 through the Non-secure instance of the mailbox FIFO. In turn, this message will generate a Non-secure interrupt, which is separate from the Secure FIFO interrupt line. This does not interfere with any Secure message passing that might be going on at the same time, and Non-secure code can not snoop Secure messages because it does not have access to the Secure mailboxes. The software running in the Secure and Non-secure domain can be identical, and the processors' bus accesses to the SIO will automatically be routed to the Secure or Non-secure version of the mailbox registers.

The following hardware is *not* duplicated:

- The GPIO registers are shared, and Non-secure accesses are filtered on a per-GPIO basis by the Non-secure GPIO mask defined in the ACCESSCTRL GPIO_NSMASK0 and GPIO_NSMASK1 registers
- The RISC-V standard platform timer (MTIME, MTIMEH), which is also usable by Arm processors, is present only in the Secure SIO, as it is a Machine-mode peripheral on RISC-V
- The interpolator and TMDS encoder peripherals are assignable to either the Secure or Non-secure SIO using the PERI_NONSEC register

Accesses to the SIO register address range, starting at `0xd0000000` (SIO_BASE), are mapped to the SIO bank which matches the security attribute of the bus access. This means accesses from the Arm Secure state, or RISC-V Machine mode, will access the Secure SIO bank, and accesses from the Arm Non-secure state, or RISC-V User mode, will access the Non-secure SIO bank.

Additionally, Secure accesses can use the mirrored address range starting at `0xd0020000` (SIO_NONSEC_BASE) to access the Non-secure view of SIO, for example, using the Non-secure doorbells to interrupt Non-secure code running on the other core. Attempting to access this address range from Non-secure code will generate a bus fault.

NOTE

The `0x20000` offset of the Secure-to-Non-secure mirror matches the PPB mirrors at `0xe0000000` (PPB_BASE) and `0xe0020000` (PPB_NONSEC_BASE), which function similarly.

NOTE

Debug access is mapped to the Secure/Non-secure SIO using the security attribute of the debugger's bus access, which may differ from the security state that the core was halted in.

3.1.2. CPUID

The CPUID SIO register returns a value of 0 when read by core 0, and 1 when read by core 1. This helps software identify the core running the current application. The initial boot sequence also relies on this check: both cores start running simultaneously, core 1 goes into a deep sleep state, and core 0 continues the main boot sequence.

IMPORTANT

Don't confuse the SIO CPUID register with the Cortex-M33 CPUID register on each processor's internal Private Peripheral Bus, which lists the processor's part number and version.

NOTE

Reading the MHARTID CSR on each Hazard3 core returns the same values as CPUID: 0 on core 0, and 1 on core 1.

3.1.3. GPIO control

The SIO GPIO registers control GPIOs which have the SIO function selected (function 5). This function is supported on the following pins:

- All user GPIOs (GPIOs 0 through 29, or 0 through 47, depending on package option)
- QSPI pins
- USB DP/DM pins

All SIO GPIO control registers come in pairs. The lower-addressed register in each pair (for example, GPIO_IN) is connected to GPIOs 0 through 31, and the higher-addressed register in each pair (for example, GPIO_HI_IN) is connected to GPIOs 32 through 47, the QSPI pins, and the USB DP/DM pins.

NOTE

To drive a pin with the SIO's GPIO registers, the GPIO multiplexer for this pin must first be configured to select the SIO GPIO function. See Table 646.

These GPIO registers are *shared* between the two cores: both cores can access them simultaneously. There are three groups of registers:

- Output registers, GPIO_OUT and GPIO_HI_OUT set the output level of the GPIO. 0 for low output, 1 for high output.
- Output enable registers, GPIO_OE and GPIO_HI_OE, are used to enable the output driver. 0 for high-impedance, 1 for drive high or low based on GPIO_OUT and GPIO_HI_OUT.
- Input registers, GPIO_IN and GPIO_HI_IN, allow the processor to sample the current state of the GPIOs.

Reading GPIO_IN returns up to 32 input values in a single read, and software then masks out individual pins it is interested in.

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_gpio/include/hardware/gpio.h Lines 869 - 879

```
869 static inline bool gpio_get(uint gpio) {
870 #ifdef NUM_BANK0_GPIOS <= 32
871     return sio_hw->gpio_in & (1u << gpio);
872 #else
873     if (gpio < 32) {
874         return sio_hw->gpio_in & (1u << gpio);
875     } else {
876         return sio_hw->gpio_hi_in & (1u << (gpio - 32));
877     }
878 #endif
879 }
```

The OUT and OE registers also have atomic SET, CLR, and XOR aliases. This allows software to update a subset of the pins in one operation. This ensures safety for concurrent GPIO access, both between the two cores and between a single core's interrupt handler and foreground code.

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_gpio/include/hardware/gpio.h Lines 918 - 924

```
918 static inline void gpio_set_mask(uint32_t mask) {
919 #ifdef PICO_USE_GPIO_COPROCESSOR
920     gpioc_lo_out_set(mask);
```

```
921 #else
922     sio_hw->gpio_set = mask;
923 #endif
924 }
```

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_gpio/include/hardware/gpio.h Lines 965 - 971

```
965 static inline void gpio_clr_mask(uint32_t mask) {
966 #ifdef PICO_USE_GPIO_COPROCESSOR
967     gpioc_lo_out_clr(mask);
968 #else
969     sio_hw->gpio_clr = mask;
970 #endif
971 }
```

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_gpio/include/hardware/gpio.h Lines 1155 - 1180

```
1155 static inline void gpio_put(uint gpio, bool value) {
1156 #ifdef PICO_USE_GPIO_COPROCESSOR
1157     gpioc_bit_out_put(gpio, value);
1158 #elif NUM_BANK0_GPIOS <= 32
1159     uint32_t mask = 1ul << gpio;
1160     if (value)
1161         gpio_set_mask(mask);
1162     else
1163         gpio_clr_mask(mask);
1164 #else
1165     uint32_t mask = 1ul << (gpio & 0x1fu);
1166     if (gpio < 32) {
1167         if (value) {
1168             sio_hw->gpio_set = mask;
1169         } else {
1170             sio_hw->gpio_clr = mask;
1171         }
1172     } else {
1173         if (value) {
1174             sio_hw->gpio_hi_set = mask;
1175         } else {
1176             sio_hw->gpio_hi_clr = mask;
1177         }
1178     }
1179 #endif
1180 }
```

If both processors write to an `OUT` or `OE` register (or any of its `SET`/`CLR`/`XOR` aliases) on the same clock cycle, the result is as though core 0 wrote first, then core 1 wrote immediately afterward. For example, if core 0 SETs a bit and core 1 XORs it on the same clock cycle, the bit ends up with a value of `0`.

NOTE

This is a conceptual model for the result produced when two cores write to a GPIO register simultaneously. The register never contains the intermediate values at any point. In the previous example, if the pin is initially `0`, and core 0 performs a `SET` while core 1 performs a `XOR`, the GPIO output remains low throughout the clock cycle.

As well as being shared between cores, the GPIO registers are also shared between security domains. The Secure and Non-secure SIO offer alternative views of the same GPIO registers, which are always mapped as GPIO function `5`. However, the Non-secure SIO can only access pins which are enabled in the GPIO Non-secure mask configured by the ACCESSCTRL registers GPIO_NSMASK0 and GPIO_NSMASK1. The layout of the NSMASK registers matches the layout of the SIO registers — for example, `QSPI_SCK` is bit `26` in both GPIO_HI_IN and GPIO_NSMASK1.

When a pin is not enabled in Non-secure code:

- Writes to the corresponding GPIO registers from a Non-secure context have no effect.
- Reads from a Non-secure context return zeroes.
- Reads and writes from a Secure context function as usual using the Secure bank.

The GPIO coprocessor port (Section 3.6.1) provides dedicated instructions for accessing the SIO GPIO registers from the Cortex-M33 processors. This includes the ability to read and write 64 bits in a single operation.

3.1.4. Hardware spinlocks

The SIO provides 32 hardware spinlocks, which can be used to manage mutually-exclusive access to shared software resources. Each spinlock is a one-bit flag, mapped to a different address (from SPINLOCK0 to SPINLOCK31). Software interacts with each spinlock with one of the following operations:

- Read: Attempt to claim the lock. Read value is non-zero if the lock was successfully claimed, or zero if the lock had already been claimed by a previous read.
- Write (any value): Release the lock. The next attempt to claim the lock will succeed.

If both cores try to claim the same lock on the same clock cycle, core 0 succeeds.

Generally software will acquire a lock by repeatedly polling the lock bit ("spinning" on the lock) until it is successfully claimed. This is inefficient if the lock is held for long periods, so generally the spinlocks should be used to protect short critical sections of higher-level primitives such as mutexes, semaphores and queues.

For debugging purposes, the current state of all 32 spinlocks can be observed via SPINLOCK_ST.

NOTE

RP2350 has separate spinlocks for Secure and Non-secure SIO banks because sharing these registers would allow Non-secure code to deliberately starve Secure code that attempts to acquire a lock. See Section 3.1.1.

NOTE

The processors on RP2350 support standard atomic/exclusive access instructions which, in concert with the global exclusive monitor (Section 2.1.6), allow both cores to safely share variables in SRAM. The SIO spinlocks are still included for compatibility with RP2040.

NOTE

Due to RP2350-E2, writes to new SIO registers above an offset of `+0x180` alias the spinlocks, causing spurious lock releases. The SDK by default uses atomic memory accesses to implement the `hardware_sync_spin_lock` API, as a workaround on RP2350 A2.

3.1.5. Inter-processor FIFOs (Mailboxes)

The SIO contains two FIFOs for passing data, messages or ordered events between the two cores. Each FIFO is 32 bits wide and four entries deep. One of the FIFOs can only be written by core 0 and read by core 1. The other can only be written by core 1 and read by core 0.

Each core writes to its outgoing FIFO by writing to FIFO_WR and reads from its incoming FIFO by reading from FIFO_RD. A status register, FIFO_ST, provides the following status signals:

- Incoming FIFO contains data (`VLD`).
- Outgoing FIFO has room for more data (`RDY`).
- The incoming FIFO was read from while empty at some point in the past (`ROE`).
- The outgoing FIFO was written to while full at some point in the past (`WOF`).

Writing to the outgoing FIFO while full, or reading from the incoming FIFO while empty, does not affect the FIFO state. The current contents and level of the FIFO is preserved. However, this does represent some loss of data or reception of invalid data by the software accessing the FIFO, so a sticky error flag is raised (`ROE` or `WOF`).

The SIO has a FIFO IRQ output for each core to notify the core that it has received FIFO data. This is a *core-local interrupt*, mapped to the same IRQ number on each core (`SIO_IRQ_FIFO`, interrupt number `25`). Non-secure FIFO interrupts use a separate interrupt line, (`SIO_IRQ_FIFO_NS`, interrupt number `27`). It is not possible to interrupt on the opposite core's FIFO.

Each IRQ output is the logical OR of the `VLD`, `ROE` and `WOF` bits in that core's FIFO_ST register: that is, the IRQ is asserted if any of these three bits is high, and clears again when they are all low. To clear the `ROE` and `WOF` flags, write any value to FIFO_ST. To clear the `VLD` flag, read data from the FIFO until it is empty.

If the corresponding interrupt line is enabled in the processor's interrupt controller, the processor takes an interrupt each time data appears in its FIFO, or if it has performed some invalid FIFO operation (read on empty, write on full).

NOTE

`ROE` and `WOF` only become set if software misbehaves in some way. Generally, the interrupt handler triggers when data appears in the FIFO, raising the `VLD` flag. Then, the interrupt handler clears the IRQ by reading data from the FIFO until `VLD` goes low once more.

The inter-processor FIFOs and the Event signals are used by the bootrom (Chapter 5) `wait_for_vector` routine, where core 1 remains in a sleep state until it is woken, and provided with its initial stack pointer, entry point and vector table through the FIFO.

NOTE

RP2350 has separate FIFOs and interrupts for Secure and Non-secure SIO banks. See Section 3.1.1

3.1.6. Doorbells

The doorbell registers raise an interrupt on the opposite core. There are 8 doorbell flags in each direction, combined into a single doorbell interrupt per core. This is a core-local interrupt: the same interrupt number on each core (`SIO_IRQ_BELL`, interrupt number `26`) notifies that core of incoming doorbell interrupts.

Whereas the mailbox FIFOs are used for cross-core events whose count and order is important, doorbells are used for events which are accumulative (i.e. may post multiple times, but only answered once) and which can be responded to in any order.

Writing a non-zero value to the DOORBELL_OUT_SET register raises the opposite core's doorbell interrupt. The interrupt remains raised until all bits are cleared. Generally, the opposite core enters its doorbell interrupt handler, reads its DOORBELL_IN_CLR register to get the mask of active doorbell flags, and then writes back to acknowledge and clear the interrupt.

The DOORBELL_IN_SET register allows a processor to ring its own doorbell. This is useful when the routine which rings a doorbell can be scheduled on either core. Likewise, for symmetry, a processor can clear the opposite core's doorbell flags using the DOORBELL_OUT_CLR register: this is useful for setup code, but should be avoided in general because of the potential for race conditions when acknowledging interrupts meant for the opposite core.

At any time, a core can read back its DOORBELL_OUT_SET or DOORBELL_OUT_CLR register (they return the same result) to see the status of doorbell interrupts posted to the opposite core. Likewise, reading either DOORBELL_IN_SET or DOORBELL_IN_CLR returns the status of doorbell interrupts posted to this core.

NOTE

RP2350 has separate per-core doorbell interrupt signals and doorbell registers for Secure and Non-secure SIO banks. Non-secure doorbells are posted on `SIO_IRQ_BELL_NS`, interrupt number `28`. See Section 3.1.1.

3.1.7. Integer divider

RP2040's memory-mapped integer divider peripheral is not present on RP2350, since the processors support divide instructions. The address space previously allocated for the divider registers is now reserved.

3.1.8. RISC-V platform timer

This 64-bit timer is a standard peripheral described in the RISC-V privileged specification, usable equally by the Arm and RISC-V processors on RP2350. It drives the per-core `SIO_IRQ_MTIMECMP` system-level interrupt (Section 3.2), as well as the `mip.mtip` timer interrupt on the RISC-V processors.

There is a single 64-bit counter, shared between both cores. The low and high half can be accessed through the MTIME and MTIMEH SIO registers. Use the following procedure to safely read the 64-bit time using 32-bit register accesses:

1. Read the upper half, MTIMEH.
2. Read the lower half, MTIME.
3. Read the upper half again.
4. Loop if the two upper-half reads returned different values.

This is similar to the procedure for reading RP2350 system timers (Section 12.8). The loop should only happen once, when the timer is read at exactly the instant of a 32-bit rollover, and even this is only occasional. If you require constant-time operation, you can instead zero the lower half when the two upper-half reads differ.

Timer interrupts are generated based on a per-core 64-bit time comparison value, accessed through the MTIMECMP and MTIMECMPH SIO registers. Each core gets its own copy of these registers, accessed at the same address. The per-core interrupt is asserted whenever the current time indicated in the MTIME registers is greater than or equal to that core's MTIMECMP. Use the following sequence to write a new 64-bit timer comparison value without causing spurious interrupts:

1. Write all-ones to MTIMECMP (guaranteed greater than or equal to the old value, *and* the lower half of the target value).
2. Write the upper half of the target value to MTIMECMPH (combined 64-bit value is still greater than or equal to the target value).

3. Write the lower half of the target value to MTIMECMP.

The RISC-V timer can count either ticks from the system-level tick generator (Section 8.5), or system clock cycles, selected by the MTIME_CTRL register. Use a 1 microsecond time base for compatibility with most RISC-V software.

3.1.9. TMDS encoder

Each core is equipped with an implementation of the TMDS encode algorithm described in chapter 3 of the DVI 1.0 specification. In general, the HSTX peripheral (Section 12.11) supports lower processor overhead for DVI-D output as well as a wider range of pixel formats, but the SIO TMDS encoders are included for use with non-HSTX-capable GPIOs.

The TMDS_CTRL register allows configuration of a number of input pixel formats, from 16-bit RGB down to 1-bit monochrome. Once the encoder has been set up, the processor writes 32 bits of colour data at a time to TMDS_WDATA, and then reads TMDS data symbols from the output registers. Depending on the pixel format, there may be multiple TMDS symbols read for each write to TMDS_WDATA. There are no stalls: encoding is limited entirely by the processor's load/store bandwidth, up to one 32-bit read or write per cycle per core.

To allow for framebuffer/scanbuffer resolution lower than the display resolution, the output registers have both peek and pop aliases (e.g. TMDS_PEEK_SINGLE and TMDS_POP_SINGLE). Reading either register advances the encoder's DC balance counter, but only the pop alias shifts the colour data in TMDS_WDATA so that multiple correctly-DC-balanced TMDS symbols can be generated from the same input pixel.

The TMDS encoder peripherals are not duplicated over security domains. They are assigned to the Secure SIO at reset, and can be reassigned to the Non-secure SIO using the PERI_NONSEC register.

3.1.10. Interpolator

Each core is equipped with two *interpolators* (`INTERP0` and `INTERP1`) that can accelerate tasks by combining certain pre-configured operations into a single processor cycle. Intended for cases where the pre-configured operation repeats many times, interpolators result in code which uses both fewer CPU cycles and fewer CPU registers in time-critical sections.

The interpolators already accelerate audio operations within the SDK. Their flexible configuration makes it possible to optimise many other tasks, including:

- quantization
- dithering
- table lookup address generation
- affine texture mapping
- decompression
- linear feedback

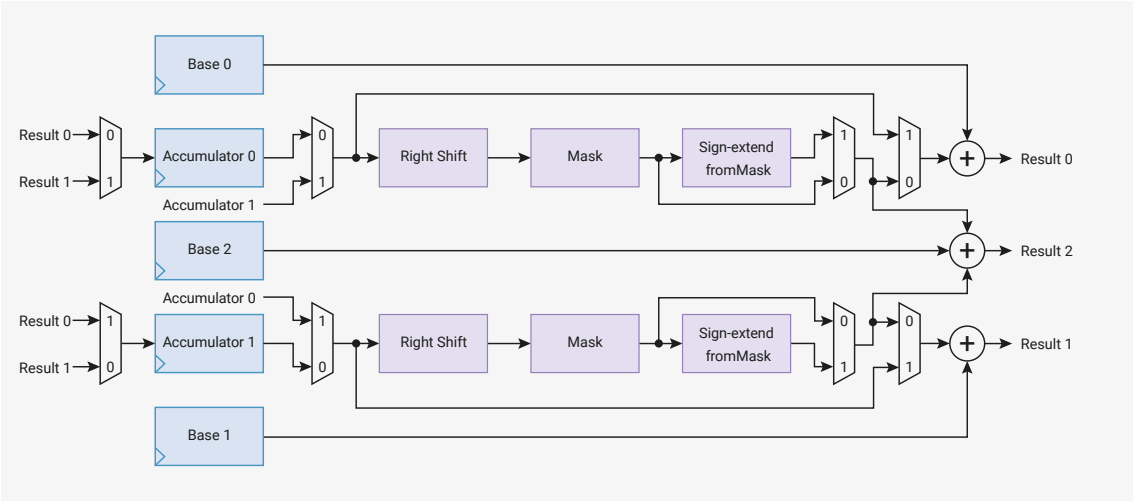


Figure 8. An interpolator. The two accumulator registers and three base registers have single-cycle read/write access from the processor. The interpolator is organised into two lanes, which perform masking, shifting and sign-extension operations on the two accumulators. This produces three possible results, by adding the intermediate shift/mask values to the three base registers. From left to right, the multiplexers on each lane are controlled by the following flags in the CTRL registers: CROSS_RESULT, CROSS_INPUT, SIGNED, and ADD_RAW.

The processor can write or read any interpolator register in one cycle, and the results are ready on the next cycle. The processor can also perform an addition on one of the two accumulators ACCUM0 or ACCUM1 by writing to the corresponding ACCUMx_ADD register.

The three results are available in the read-only locations PEEK0, PEEK1, PEEK2. Reading from these locations does not change the state of the interpolator. The results are also aliased at the locations POP0, POP1, POP2; reading from a POPx alias returns the same result as the corresponding PEEKx, and simultaneously writes back the lane results to the accumulators. Use the POPx aliases to advance the state of interpolator each time a result is read.

You can adjust interpolator behaviour with the following operational modes:

- fractional blending between two values
- *clamping* values to restrict them within a given range.

The following example shows a trivial example of *popping* a lane result to produce simple iterative feedback.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/interp/hello_interp/hello_interp.c Lines 11 - 23

```
11 void times_table() {
12     puts("9 times table:");
13
14     // Initialise lane 0 on interp0 on this core
15     interp_config cfg = interp_default_config();
16     interp_set_config(interp0, 0, &cfg);
17
18     interp0->accum[0] = 0;
19     interp0->base[0] = 9;
20
21     for (int i = 0; i < 10; ++i)
22         printf("%d\n", interp0->pop[0]);
23 }
```

3.1.10.1. Lane operations

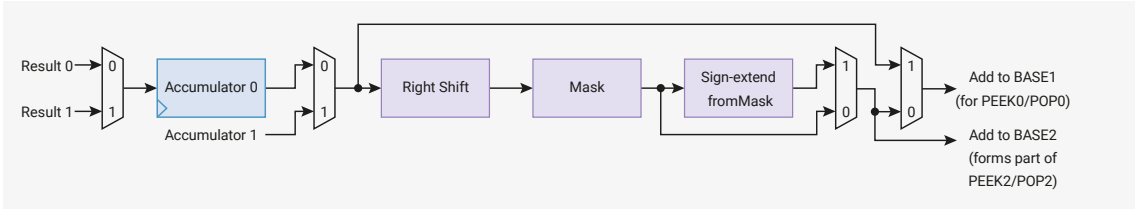


Figure 9. Each lane of each interpolator can be configured to perform mask, shift and sign-extension on one of the accumulators. This is fed into adders which produce final results, which may optionally be fed back into the accumulators with each read. The datapath can be configured using a handful of 32-bit multiplexers. From left to right, these are controlled by the following CTRL flags: CROSS_RESULT, CROSS_INPUT, SIGNED, and ADD_RAW.

Each lane performs these three operations, in sequence:

- A right shift by `CTRL_LANEx_SHIFT` (0 to 31 bits)
- A mask of bits from `CTRL_LANEx_MASK_LSB` to `CTRL_LANEx_MASK_MSB` inclusive (each ranging from bit 0 to bit 31)
- A sign extension from the top of the mask, i.e. take bit `CTRL_LANEx_MASK_MSB` and OR it into all more-significant bits, if `CTRL_LANEx_SIGNED` is set

For example, if:

- `ACCUM0` = `0xdeadbeef`
- `CTRL_LANE0_SHIFT` = 8
- `CTRL_LANE0_MASK_LSB` = 4
- `CTRL_LANE0_MASK_MSB` = 7
- `CTRL_SIGNED` = 1

Then lane 0 would produce the following results at each stage:

- Right shift by 8 to produce `0x00deadbe`
- Mask bits 7 to 4 to produce `0x00deadbe & 0x000000f0` = `0x000000b0`
- Sign-extend up from bit 7 to produce `0xffffffb0`

In software:

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/interp/hello_interp/hello_interp.c Lines 25 - 46

```
25 void moving_mask() {
26     interp_config cfg = interp_default_config();
27     interp0->accum[0] = 0x1234abcd;
28
29     puts("Masking:");
30     printf("ACCUM0 = %08x\n", interp0->accum[0]);
31     for (int i = 0; i < 8; ++i) {
32         // LSB, then MSB. These are inclusive, so 0,31 means "the entire 32 bit register"
33         interp_config_set_mask(&cfg, i * 4, i * 4 + 3);
34         interp_set_config(interp0, 0, &cfg);
35         // Reading from ACCUMx_ADD returns the raw lane shift and mask value, without BASEx
   added
36         printf("Nibble %d: %08x\n", i, interp0->add_raw[0]);
37     }
38
39     puts("Masking with sign extension:");
40     interp_config_set_signed(&cfg, true);
41     for (int i = 0; i < 8; ++i) {
42         interp_config_set_mask(&cfg, i * 4, i * 4 + 3);
43         interp_set_config(interp0, 0, &cfg);
44         printf("Nibble %d: %08x\n", i, interp0->add_raw[0]);
45     }
46 }
```

The above example should print the following:

```
ACCUM0 = 1234abcd
Nibble 0: 0000000d
Nibble 1: 000000c0
Nibble 2: 00000b00
Nibble 3: 0000a000
Nibble 4: 00040000
Nibble 5: 00300000
Nibble 6: 02000000
Nibble 7: 10000000
Masking with sign extension:
Nibble 0: fffffffd
Nibble 1: ffffffc0
Nibble 2: fffffb00
Nibble 3: ffffa000
Nibble 4: 00040000
Nibble 5: 00300000
Nibble 6: 02000000
Nibble 7: 10000000
```

Changing the result and input multiplexers can create feedback between the accumulators. This is useful for audio dithering.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/interp/hello_interp/hello_interp.c Lines 48 - 66

```
48 void cross_lanes() {
49     interp_config cfg = interp_default_config();
50     interp_config_set_cross_result(&cfg, true);
51     // ACCUM0 gets lane 1 result:
52     interp_set_config(interp0, 0, &cfg);
53     // ACCUM1 gets lane 0 result:
54     interp_set_config(interp0, 1, &cfg);
55
56     interp0->accum[0] = 123;
57     interp0->accum[1] = 456;
58     interp0->base[0] = 1;
59     interp0->base[1] = 0;
60     puts("Lane result crossover:");
61     for (int i = 0; i < 10; ++i) {
62         uint32_t peek0 = interp0->peek[0];
63         uint32_t pop1 = interp0->pop[1];
64         printf("PEEK0, POP1: %d, %d\n", peek0, pop1);
65     }
66 }
```

This should print the following :

```
PEEK0, POP1: 124, 456
PEEK0, POP1: 457, 124
PEEK0, POP1: 125, 457
PEEK0, POP1: 458, 125
PEEK0, POP1: 126, 458
PEEK0, POP1: 459, 126
PEEK0, POP1: 127, 459
PEEK0, POP1: 460, 127
PEEK0, POP1: 128, 460
PEEK0, POP1: 461, 128
```

3.1.10.2. Blend mode

Blend mode is available on INTERP0 on each core, and is enabled by the CTRL_LANE0_BLEND control flag. It performs linear interpolation, which we define as follows:

$$x = x_0 + \alpha(x_1 - x_0), \text{ for } 0 \leq \alpha < 1$$

Where x_0 is the register BASE0, x_1 is the register BASE1, and α is a fractional value formed from the least significant 8 bits of the lane 1 shift and mask value.

Blend mode differs from normal mode in the following ways:

- PEEK0, POP0 return the 8-bit alpha value (the 8 LSBs of the lane 1 shift and mask value), with zeroes in result bits 31 down to 24.
- PEEK1, POP1 return the linear interpolation between BASE0 and BASE1
- PEEK2, POP2 do not include lane 1 result in the addition (i.e. it is BASE2 + lane 0 shift and mask value)

The result of the linear interpolation is equal to BASE0 when the alpha value is 0, and equal to BASE0 + 255/256 * (BASE1 - BASE0) when the alpha value is all-ones.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/interp/hello_interp/hello_interp.c Lines 68 - 87

```
68 void simple_blend1() {
69     puts("Simple blend 1:");
70
71     interp_config cfg = interp_default_config();
72     interp_config_set_blend(&cfg, true);
73     interp_set_config(interp0, 0, &cfg);
74
75     cfg = interp_default_config();
76     interp_set_config(interp0, 1, &cfg);
77
78     interp0->base[0] = 500;
79     interp0->base[1] = 1000;
80
81     for (int i = 0; i <= 6; i++) {
82         // set fraction to value between 0 and 255
83         interp0->accum[1] = 255 * i / 6;
84         // ≈ 500 + (1000 - 500) * i / 6;
85         printf("%d\n", (int) interp0->peek[1]);
86     }
87 }
```

This should print the following (note the 255/256 resulting in 998 not 1000):

```
500
582
666
748
832
914
998
```

CTRL_LANE1_SIGNED controls whether BASE0 and BASE1 are sign-extended for this interpolation (this sign extension is required because the interpolation produces an intermediate product value 40 bits in size). CTRL_LANE0_SIGNED continues to control the sign extension of the lane 0 intermediate result in PEEK2, POP2 as normal.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/interp/hello_interp/hello_interp.c Lines 90 - 121

```
 90 void print_simple_blend2_results(bool is_signed) {
 91     // lane 1 signed flag controls whether base 0/1 are treated as signed or unsigned
 92     interp_config cfg = interp_default_config();
 93     interp_config_set_signed(&cfg, is_signed);
 94     interp_set_config(interp0, 1, &cfg);
 95
 96     for (int i = 0; i <= 6; i++) {
 97         interp0->accum[1] = 255 * i / 6;
 98         if (is_signed) {
 99             printf("%d\n", (int) interp0->peek[1]);
100         } else {
101             printf("0x%08x\n", (uint) interp0->peek[1]);
102         }
103     }
104 }
105
106 void simple_blend2() {
107     puts("Simple blend 2:");
108
109     interp_config cfg = interp_default_config();
110     interp_config_set_blend(&cfg, true);
111     interp_set_config(interp0, 0, &cfg);
112
113     interp0->base[0] = (uint32_t) -1000;
114     interp0->base[1] = 1000;
115
116     puts("signed:");
117     print_simple_blend2_results(true);
118
119     puts("unsigned:");
120     print_simple_blend2_results(false);
121 }
```

This should print the following:

```
signed:
-1000
-672
-336
-8
328
656
992
unsigned:
0xfffffc18
0xd5fffd60
0xaafffeb0
0x80fffff8
0x56000148
0x2c000290
0x010003e0
```

Finally, in blend mode when using the `BASE_1AND0` register to send a 16-bit value to each of `BASE0` and `BASE1` with a single 32-bit write, the sign-extension of these 16-bit values to full 32-bit values during the write is controlled by `CTRL_LANE1_SIGNED` for both bases, as opposed to non-blend-mode operation, where `CTRL_LANE0_SIGNED` affects extension into `BASE0` and `CTRL_LANE1_SIGNED` affects extension into `BASE1`.