Picochan

0.1

Generated by Doxygen 1.14.0

Chapter 1

Picochan - a channel subsystem for Raspberry Pi Pico

1.1 Introduction

Picochan is

- library software that runs on Raspberry Pi Pico microcontrollers or, more generally, RP-series family chips such as RP2040 and RP2350.
- inspired by the I/O architecture of IBM mainframes which provides application-facing I/O machine instructions that trigger the *Channel Subsystem* (CSS) to run asynchronous channel programs of *Channel Command Words* (CCWs) communicated to a remote *Control Unit* (CU) that performs the low-level device I/O
- implements a CSS and CU(s) as low-level libraries to drive available Pico peripherals (e.g. DMA, UART, PIO) that allow separating the CSS from CU on separate cores of one Pico or separate Picos
- licensed as Open Source
- not designed particularly to be a replacement or "better than" any existing way of writing software for Pico that does I/O, whether with plain software libraries or addition peripherals (e.g. USB) or hardware (e.g. breakout boards)
- written for interest in order to find out whether this I/O model which proved useful for mainframes and their programmers when introduced 60+ years ago is a useful model now that Pico microcontroller I/O capabilities have caught up with those of mainframes 30-40 years ago
- **NOT** compatible in any way with actual mainframe I/O software, hardware, channels, CUs or I/O or device hardware
- **NOT** anything that does or would make sense to port to or compile on actual mainframe hardware, whether old or new. This is fundamental since Picochan is written to use the low-level microcontroller-style "bit-banging" of DMA and pin-based peripherals (UART, PIO) in order to implement its functionality - the functionality which actual IBM mainframe architectures (from S/360 right up to modern z/Architecture) hide invisibly behind their actual hardware/firmware-implemented I/O architecture.

1.2 Channel Programs

The Application API arranges for a series of I/O operations to happen to/from a device by using the CSS to start a *channel program* to its subchannel that runs asynchronously, managed by the CSS. Each I/O command - known as a *Channel Command Word* (CCW) - in the channel program gets the device to perform a "read-like" or "write-like" command that send/receives (offers/requests) a segment of data to/from the device. The command has an 8-bit command code field so a device may simply implement a basic "read" and/or "write" command or may offer a larger range of more complex commands for application use.

CCWs can be chained together in sequences and loops (with simplistic conditional branches) to form a channel program. Each CCW in the channel program and its completion on success or error can be flagged to notify the application by means of callbacks or interrupts. The notifications can be either passive (with the channel program continuing in parallel) or allowing suspend/resume that the program can use to inspect and/or update the CCWs of the channel program as it progresses.

For more information, see the Introduction to Channel Programs.

1.3 Picochan APIs

For writing an application that uses the CSS to perform I/O to devices on a CU, you use the Picochan application API for using the CSS along with documentation provided by the CU-side device driver author on what CCW commands are supported for that device and what they do.

Application API for using CSS

For writing a device driver that uses the CU to talk to a specific kind of device and offers a useful set of CCWs for a Picochan application on a CSS to use, you use the Picochan application API for using the CU along with documentation for the device you are driving.

Device driver API for using CU

1.4 Compiling

Picochan is written in C using the Pico SDK. It uses CMake in the way recommended by that SDK. Similarly to how the Pico SDK divides into multiple modules with names of the form `pico_foo` and `hardware_foo`, Picochan is divided into three CMake modules:

- `picochan_base`
- `picochan_css`
- `picochan_cu`

For which modules to use for which purposes and how to use CMake to compile your application and/or device driver programs, see Compiling.

1.5 Tracing

Picochan can be configured to write trace records for events happening in the CSS and CUs to help debugging of applications, device drivers and Picochan itself. Trace records are small binary records written with low-overhead (although compiled out by default) to in-memory ring buffers (e.g. 2 x 1KB). Offloading trace buffers can be done easily even with no application support if, e.g., SWD access is available (via openocd or gdb) and a Picochan-supplied `pch_dump_trace` program (a small C program intended to be compiled and run off-platform) parses and displays human-readable output of what each event represents. See Tracing.

1.6 Design

There is a brief summary of the design of Picochan.

Chapter 2

Channel programs

2.0.1 Introduction

A *channel program* is a series of (and often just one) 8-byte *Channel Command Words* (CCWs).

A CCW has API C type `pch_ccw_t` which is architecturally defined:

- 8-bit command code
- 8-bit flags
- 16-bit data segment size
- 32-bit data address

The required alignment for a CCW is 4-bytes, not 8-bytes, i.e. a CCW must be at an address divisble by 4 but need not be at an address that is divisible by 8.

Each subchannel can be running a channel program independently, started (asynchronously) by the application API `pch_sch_start(sid, ccwaddr)` which starts a channel program sending CCW commands to the device addressed by SID `sid` (via the channel to its CU) starting with the CCW at address `ccwaddr`

When the CSS executes a CCW, it sends the device (via communication over the channel to the CU that owns the device) a request with the given command code in the CCW. That can simply be a "Write" (command code 1) or "Read" (command code 2) - where Write and Read mean whatever the device driver chooses them to mean - or a different device-driver-documented command code (up to 239) for more complex commands.

- All odd command codes are Write-type (the CCW address and size provides a data segment for the device to receive and use to "write to the device")
- All even command codes are Read-type (the CCW address and size provides a data segment to be written to by the device).

Since the device driver is running on remotely on the CU (whether an entirely differnt Pico or the other core of the same Pico that the CSS runs on) the data transfers from/to the CCW data segment to/from the device happen via the channel and Pico peripherals (DMA, UART, PIO whatever), as driven by the CSS/CU software. This is the the whole point of this software.

The data area for the device to read/write begins with the `(address, count)` segment in that first CCW but can continue...

- if the "chain-data" flag (`PCH_CCW_FLAG_CD`) is set in the CCW flags field, then when the device exhausts that segment, the CSS fetches the next CCW in memory (next 8 bytes) and the device continues its reading/writing from/to the `(address,count)` in the new CCW. For that new ("data-chained") CCW, the command field is ignored.

When the data area is exhausted from that CCW command (the segment data-chained CCWs), the channel program will finish...unless the "chain-command" flag is set in the most recent CCW. See further down for what happens for command chaining.

When the channel program finishes, it is because the device has sent a channel operation to the CSS saying "UpdateStatus" (a CU->CSS operation code) with an 8-bit device status whose flags say "finish the channel program".

The "device status" is an 8-bit architected set of flags for the device to inform the CSS and the application

- The device can inform the CSS at any time about its status - it arrives at the CSS from the CU in an "Update↩ Status" protocol operation
- When a device has finished processing a CCW command, it will (and indeed must) send an UpdateStatus with a device status whose flags indicate that completion
- A device can send an UpdateStatus even when it is not in the process of dealing with a channel program - this is known as an "unsolicited" device status and it includes an "Alert" flag so that an application can be notified asynchronously about an event of interest at a device even when no channel program is in progress for that subchannel.
- The CSS makes that 8-bit device status visible to the application by storing it in the devs field in the Subchannel Status Word ("SCSW") part of the SCHIB at times in time for when the subchannel is notified. Some fields of the SCSW, including that for the device status, may not contain meaningful values at other times.

2.0.2 When a CCW command in a channel program is finished

When the device has finished with one CCW command (including any following data-chained CCWs) it sends an UpdateStatus with a device status whose flags indicate that it has finished that CCW command (flags including DeviceEnd and ChannelEnd)

At this point the CSS looks at the status of the subchannel and the flags field of the CCW to decide how to continue, testing the following conditions in order:

- If the subchannel has any unusual state (for example a CSS-side error or the application has done a `pch_↩ sch_halt(sid)`) then then the channel program ends - see the "Channel program ending" section below.
- If the CCW "chain-command" flag (`PCH_CCW_FLAG_CC`) is *not* set, then the channel program ends
- If the device status has flags that indicate any "unusual conditions" (anything other than a simple Device↩ End|ChannelEnd and an optional StatusModifier) then the channel program ends
- Here, the subchannel is OK, the device status indicates the CCW command was processed with no unusual conditions and the CCW CCW "chain-command" flag is present: the CSS proceeds with "command chaining" as described immediately below.

2.0.3 Channel program command chaining

When an individual CCW command has finished and command-chaining is appropriate (see above for when that is), rather than ending the channel program, the CSS proceeds by fetching another CCW.

- If the device status did *not* include the StatusModifier flag, then the next CCW fetched is the one in memory immediately after the previous CCW (i.e. at an address 8 bytes beyond the previous CCW address)
- If the device status *did* include the StatusModifier flag, then the CCW "skips" the next CCW and fetches the one after that (i.e. at an address 16 bytes beyond the previous CCW address)

The CSS then considers that newly fetched CCW for processing by testing CCW flags as though in the following order:

- If the "suspend" (S) flag (`PCH_CCW_FLAG_SUSPEND`) is set, then the CSS "suspends" the channel program:
  - the application is notified (see below) just as it would be if the channel program ended but the CCW address and other fields in the schib are set so that the channel program can be resumed from its current position
  - with the channel program suspended, the application can inspect the schib and do whatever it likes, including updating CCWs in memory. It can then resume the channel program by calling `pch_sch_↩ resume(sid)` and the CSS resumes the channel program from where it left off.
- If the "program controlled interruption" (PCI) flag (`PCH_CCW_FLAG_PROGRAM_CONTROLLED_↩ INTERRUPTION`) is set, then the CSS triggers a notification to the application (see below) with the schib indicating the current channel program information (CCW address, status and so on) at this point but immediately continues executing the channel program (as described below) without stopping. The continuing channel program may (and probably will) cause the SCSW to be updated as it progresses so what the application sees, if it looks, will depend on when it looks.

Here, the CSS is going to "command-chain" and thus continue the channel program. If the CCW command is a normal one (1-239) then the CSS sends the command for execution just as at the start of the channel program and the program continues in that fashion.

However, as well as those normal CCW commands that can be sent to the device (as described at the beginning of the description of CCWs and channel programs) there is an additional CCW command that can be used when chaining: "Transfer In Channel" (TIC) which has CCW command code decimal 240, hex 0xf0 (`PCH_CCW_CMD_↩ TIC`). (This specific command code value is different from that for mainframe I/O.)

The CCW command TIC is the equivalent of a "goto" or "jump" for the channel program and causes the CSS to get the memory address field of the CCW (usually used as a data segment pointer) and treat it as the memory address of the next CCW to be fetched. The CSS then fetches that new CCW and continues the channel program from there, subject to a few corner cases.

It is valid (and common) to have a channel program with a loop and it is valid to TIC to a TIC CCW but there are some corner case conditions which are not yet checked and are probably not handled correctly/sensibly by Picochan at the moment.

2.0.4 Channel program ending

When the channel program finishes, the CSS "notifies" the application unless the application has chosen to avoid that by setting various mask bits.

Similarly to how IRQs typically allow masking and enabling/disabling in various ways, the CSS provides ways for the application to choose how/when an event happening for a SCHIB (such as a channel program ending) triggers notification.

2.0.5 Notification to application

An event happens on a subchannel

- when a channel program ends
- when a device explicitly sends its device status to the CSS and includes the "Alert" flag
- when the CSS fetches a CCW while progressing a channel program and the CCW includes the "Suspend" (S) flag or the "Program Controlled Interruption" (PCI) flag.

The application can either detect and manage these events using API calls (see further down) or can arrange that the CSS notifies the application when they happen by means of an asynchronous notification.

Such an asynchronous notification is via an "I/O Interuption" which, on Pico, is implemented by the raising of the "CSS I/O IRQ". The IRQ number for that is (must be) set at or after software CSS initialisation time with the API call `pch_css_set_io_irq(io_irqnum)`. The IRQ chosen should be one not used by any real peripheral - RP2040 and RP2350 have quite a few non-externally-connected IRQs that are convenient for this purpose.

Each subchannel has an Interrupt Service Class ("ISC") which is a 3-bit number (0-7) which defaults to 0. The ISC for a schib is in the Path Management Control Word ("PMCW") field and can be modified (at any time) with the general API call `pch_sch_modify(sid, pmcw)` or its convenient more specific variant `pch_sch_↩ modify_isc(sid, iscnum)`.

A global 8-bit "I/O interruption mask" (one for each ISC) determines whether a SCHIB with an I/O notification pending actually raises the I/O IRQ. The mask can be set using API call `pch_css_set_isc_enable_mask(mask)` and, as usual with such an IRQ enablement mask, when a bit changes from 0 to 1 any pending SCHIBs with the ISC will cause the notification to happen at that point.

Although an application could write and set an IRQ handler itself to manage I/O interrupts, it may well instead want to set the IRQ handler to the provided handler function `pch_css_io_irq_handler` and set a callback function with `pch_css_set_io_callback(io_callback_t io_callback)`. In this case, schib notifications cause that provided handler to retrieve the state information, clear down the notification (see the "TEST SUBCHANNEL" function, `pch_sch_test()`) and call the callback function with the interruption code (`pch_intcode_t`) and SCSW (`pch_scsw_t`) as direct arguments.

Instead of getting an asynchronous notification by an I/O interrupt or callback, an application can choose to retrieve and reset the "notification pending" state of a subchannel itself - this should be while the ISC for the subchannel is masked (i.e. the bit in the ISC enablement mask for the subchannel's ISC should be zero) or else there is a race condition when the CSS handles the notification itself.

There are three main API calls related to inspecting and resetting interruption conditions and related state for subchannels. This state all resides in the SCSW part of the schib.

- `pch_sch_store(sid, schib)` fetches the current value of the schib and writes it to the `pch_schib_t *schib` pointer. The convenience function `pch_sch_store_scsw(sid, scsw)` just fetches the SCSW field of the schib and writes that to its pointer argument. Either way, this is a "look but not touch" API call which copies the SCSW (atomically) at the precise moment the function is called and no subchannel state is changed.
- `pch_sch_test(sid, scsw)` corresponds to the mainframe I/O instruction "TEST SUBCHANNEL" and this is the usual way to do deal with non-asynchronous notification of a subchannel but the naming is counter-intuitive. As well as fetching the current SCSW from the subchannel, it atomically tests to see whether the subchannel is in an "interruption condition" state and, if so, it *resets* that state:
  - After calling `pch_sch_test` on a subchannel that is causing an interruption condition, the subchannel

        * is removed from the pending list and will no longer cause an I/O interruption, even if the ISC bit corresponding to the subchannel's ISC is re-enabled in the global ISC mask.
        * has the relevant parts of the SCSW cleared/reset so that it is no longer "status pending" - in particular, the `PCH_SC_STATUS_PENDING` flag is cleared from `ctrl_flags`
    - *Without* calling `pch_sch_test` on a subchannel, a subchannel that is causing an interruption condition will remain in that state and, simply returning from the I/O interrupt handler will mean the handler is immediately re-entered. If the I/O interrupt handler is set to `pch_css_io_irq_handler` then that function calls `pch_sch_test` for you in order to retrieve the SCSW and call your callback function (set with `pch_css_set_io_callback`) with the retrieved SCSW. The other argument to your callback function is the `pch_intcode_t` that has the corresponding SID.

- `pch_test_pending_interruption()` (no arguments) tests whether any subchannel at all is currently causing an interrupt condition (whether masked or not). It should usually only be called when the ISC mask bits are disabled for the ISC of any subchannel that may possibly cause an interruption or else there is a race condition between the `pch_test_pending_interruption()` and the I/O interrupt handler being invoked. The type of the return value is `pch_intcode_t` which has two fields: a SID and a condition code (0-3).
    - If a subchannel is causing an interruption condition, the `pch_intcode_t` returned has its SID and cc=0. In this case, the interruption condition state is removed from the subchannel and it will no longer cause an interruption. However, the "status pending" and associated flags in the SCSW remain until inspected/cleared with `pch_sch_test`.
    - If no subchannel is causing an interruption condition, the `pch_intcode_t` returned has cc=1 (and the SID is zero but meaningless)
    - The order that subchannels are tested by `pch_test_pending_interruption` is in order of increasing ISC so subchannels with low ISC numbers have "higher priority" in terms of triggering interruption conditions than higher ISCs.

Chapter 3

Compiling using Picochan

Picochan is written in C using the Pico SDK. It uses CMake in the way recommended by that SDK. Similarly to how the Pico SDK divides into multiple modules with names of the form `pico_foo` and `hardware_foo`, Picochan is divided into three CMake modules:

- `picochan_base`
- `picochan_css`
- `picochan_cu`

Application code need only compile with the `picochan_base` and `picochan_css` modules.

Device driver code need only compile with the `picochan_base` and `picochan_cu` modules.

This Doxygen-format Picochan documentation is in its early stages and does not separate API information clearly enough from internal implementation details. In an attempt to make some sort of separation, much of the Doxygen documentation (generated from code comments) has been marked as belonging to a "Doxygen topic" with a name of either `picochan_base`, `picochan_css`, `picochan_cu` or `internal_foo` (for various values of `foo`). The intent is that any Doxygen topic with prefix `internal_` is not intended for API use but there may be misclassifications.

More documentation is needed on how to compile against Picochan but in brief:

- Prepare your CMakeLists.txt in the usual way for Pico SDK
- Set environment variable `PICO_SDK_PATH` to the path of the Picochan library source - the `src/picochan` subtree of the Picochan repository. This path is the one which contains the top-level `CMakeLists.txt` file starting:

  ```
  if (EXISTS ${CMAKE_CURRENT_LIST_DIR}/base/include/picochan/ccw.h)
  ```

  It is *not* the root directory of the repository (which contains subdirectories "src" and "tools") nor is it the "base" subdirectory of the library source subtree which contains a `CMakeLists.txt` file starting:

  ```
  add_library(picochan_base INTERFACE)
  ```

- In your own `CMakeLists.txt` file for your software, add the following in appropriate places
  - Before any `target_compile_definitions` or similar, add

    ```
    include($ENV{PICOCHAN_PATH}/CMakeLists.txt)
    ```

  - In the `target_link_libraries` section for your target
    - add `picochan_css` if using application API (to CSS)
    - add `picochan_cu` if using device driver API (on CU)

    * add `hardware_dma` and `hardware_pio` if using any pio channels
    * add `hardware_uart` if using any uart channels
- In the `target_compile_definitions` for your target, add any desired settings to any extra runtime sanity and argument checks for Debug builds or to enable tracing, such as

  ```
  PARAM_ASSERTIONS_ENABLED_PCH_CUS=1
  PARAM_ASSERTIONS_ENABLED_PCH_DMACHAN=1
  PARAM_ASSERTIONS_ENABLED_PCH_TRC=1
  PARAM_ASSERTIONS_ENABLED_PCH_TXSM=1
  PARAM_ASSERTIONS_ENABLED_PCH_CSS=1
  ```
- Add definitions to choose sizes for various global tables if the defaults are not suitable (and they may well not be).
    * Examples for using CSS:

      ```
      PCH_NUM_CSS_CUS=4
      PCH_NUM_SCHIBS=40
      ```
    * Examples for using CU:

      ```
      PCH_NUM_CUS=2
      ```

Chapter 4

Channel Subsystem (CSS) API for Applications

4.0.1 Introduction

- Application API for doing I/O just uses CSS
- The CSS represents each device it knows about as a *subchannel* and the application API interacts with a subchannel by using its 16-bit *Subchannel ID* (SID)
- The SID is an index into a CSS-managed global array of control blocks called *Subchannel Information Blocks* (SCHIBS)
- API is to start and manage channel programs of Channel Command Words (CCWs)
- `pch_sch_start(sid, addr)` to start a channel program from CCW address `addr`
- channel program runs async in CSS by talking over the channel to the CU which talks to device
- notification from CSS by irq or callback when
  - channel program complete
  - or at marked CCWs to notify partial progress
  - which can "just notify" or suspend then resume with `pch_sch_resume(sid)`

Since there are not yet enough code comment Doxygen annotations to divide the generated documentation into topics properly, there follows a summary of the main definitions (e.g. types, macros and API functions) for use by CSS-side code. They are described in the Doxygen-generated documentation but some may be in a Topics sub-section, some may be in "Data Structures" and some may be under "Files".

4.0.2 Compile-time constants and definitions - examples:

```
#define PCH_NUM_CHANNELS 4
#define PCH_NUM_SCHIBS 40
```

4.0.3 Debugging assertions:

```
#define PARAM_ASSERTIONS_ENABLED_PCH_CSS 1
#define PARAM_ASSERTIONS_ENABLED_PCH_TXSM 1
```

4.0.4 Types

```
typedef struct pch_schib pch_schib_t;

typedef struct pch_pmcw pch_pmcw_t;

typedef struct pch_intcode pch_intcode_t;

typedef void(*io_callback_t)(pch_intcode_t, pch_scsw_t);
```

4.0.5 Initialisation of whole CSS

```
void pch_css_init(void);

bool pch_css_set_trace(bool trace);

// Optionally use pch_css_auto_configure_..., pch_css_configure_...
// or pch_css_set functions to choose, configure and enable IRQ
// handlers for DMA, function and I/O IRQs, then auto-configure
// and enable any remaining ones with:

void pch_css_start(io_callback_t io_callback);

void pch_css_set_isc_enable_mask(uint8_t mask);
```

4.0.6 Allocation of subchannels in a channel to a CU

```
// Claim an unused channel (returns its pch_chpid_t or
// returns -1 or panics on failure)...
int pch_chp_claim_unused(bool required);
// ...or (less commonly) claim a specific chpid
// (panics on failure)
void pch_chp_claim(pch_chpid_t chpid);
// Allocate num_devices consecutive subchannels on the channel and
// return the SID of the first.
pch_sid_t pch_chp_alloc(pch_chpid_t chpid, uint16_t num_devices);
```

4.0.6.1 Initialise a channel to a UART CU

```
// Initialise and configure a UART channel with default parameters...
void pch_chp_auto_configure_uartchan(pch_chpid_t chpid, uart_inst_t *uart, dma_channel_config ctrl);
// ...or, less commonly, configure with non-default DMA control
// register flags after initialising the UART beforehand
void pch_chp_configure_uartchan(pch_chpid_t chpid, uart_inst_t *uart, dma_channel_config ctrl);
```

4.0.6.2 Initialise a channel to a memchan (cross-core) CU

```
void pch_memchan_init();

dmachan_tx_channel_t *pch_cu_get_tx_channel(pch_chpid_t chpid);

void pch_chp_configure_memchan(pch_chpid_t chpid, pch_dmaid_t txdmaid, pch_dmaid_t rxdmaid,
      dmachan_tx_channel_t *txpeer);
```

4.0.6.3 Initialise a channel to a pio CU

TBD

4.0.7 Start a channel to a CU

```
bool pch_chp_set_trace(pch_chpid_t chpid, bool trace);

void pch_chp_start(pch_chpid_t chpid);
```

4.0.8 Set PMCW flags of a subchannel to enable/disable, trace or change ISC

```
int pch_sch_modify_flags(pch_sid_t sid, uint16_t flags);
```

4.0.9 Start, monitor and control channel programs for a subchannel

```
int pch_sch_start(pch_sid_t sid, pch_ccw_t *ccw_addr);
int pch_sch_resume(pch_sid_t sid);
int pch_sch_test(pch_sid_t sid, pch_scsw_t *scsw);
int pch_sch_modify(pch_sid_t sid, pch_pmcw_t *pmcw);
int pch_sch_store(pch_sid_t sid, pch_schib_t *out_schib);
// int pch_sch_halt(pch_sid_t sid);
// int pch_sch_cancel(pch_sid_t sid);
pch_intcode_t pch_test_pending_interruption(void);
```

4.0.10 Variations and wrappers for convenience or optimisation

```
int pch_sch_wait(pch_sid_t sid, pch_scsw_t *scsw);
int pch_sch_wait_timeout(pch_sid_t sid, pch_scsw_t *scsw, absolute_time_t timeout_timestamp);
int pch_sch_run_wait(pch_sid_t sid, pch_ccw_t *ccw_addr, pch_scsw_t *scsw);
int pch_sch_run_wait_timeout(pch_sid_t sid, pch_ccw_t *ccw_addr, pch_scsw_t *scsw, absolute_time_t
      timeout_timestamp);

int pch_sch_store_pmcw(pch_sid_t sid, pch_pmcw_t *out_pmcw);
int pch_sch_store_scsw(pch_sid_t sid, pch_scsw_t *out_scsw);

int pch_sch_modify_intparm(pch_sid_t sid, uint32_t intparm);
int pch_sch_modify_flags(pch_sid_t sid, uint16_t flags);
int pch_sch_modify_isc(pch_sid_t sid, uint8_t isc);
int pch_sch_modify_enabled(pch_sid_t sid, bool enabled);
int pch_sch_modify_traced(pch_sid_t sid, bool traced);
```

Chapter 5

Control Unit (CU) API for Device Drivers Overview

5.0.1 Introduction

- "Device driver" software runs on core of CU and talks to actual devices
- "Device driver" is not a recognised term from the architectural view, and especially not from the application and channel subsystem side, but seems to be as good a term as any to use to refer to the software written to run CU-side to deal with the actual devices
- All device driver API calls are non-blocking (dozens to at most hundreds of cycles) and have no timing constraints
- API calls set some bits, update linked lists and cause the CU to send a single 4-byte operation packet down the channel to the CSS or, if already busy, queue it up so that CU will send it as soon as the current queue of operation commands have been sent
- At software init, register at least one callback function - there can be up to 239 per CU
- At device init time, `pch_dev_set_callback()` to set callback for "Start"
- When CSS fetches a CCW for the device with its `(command, flags, address, size)` fields, CSS sends Start request to CU which calls device's callback function
  - For a Read-type CCW, device uses `pch_dev_send...(...,srcaddr,size)` to send one or more chunks of data that (via CU->CSS) get written to the CCW data segments (CSS data-chains to following CCWs if needed)
  - For a Write-type CCW, device uses `pch_dev_receive...(...,dstaddr,size)` to request chunks of data from the CCW data segments (CSS data-chains to following CCWs if needed)
- Arguments to those API calls (or an explicit `pch_dev_set_callback`) can set the callback index to a different (already-registered) one to be used the next time the CU has reason to call the device driver
- Callbacks can happen
  - when a command has been sent (so CU is ready for another)
  - when a requested update is received from CSS about "how much room is left in the data segment"
  - or for (rare) "stop as soon as you can" requests (application "HALT SUBCHANNEL", `pch_sch_halt()`)
- When device has finished with that CCW command and its (data-chain of) 1 or more CCW segments, it uses `pch_update_status...(...,devstatus)` to cause the CSS to finish that CCW command. The devstatus can be

  - "normal" (CSS either command-chains to next CCW or notifies final state to application)
  - include "error" flags (prevents command-chaining and gets notified to application)
  - or "normal with StatusModifier" (CSS skips a CCW to allow for conditional logic in the channel program decided by device side)
- Device driver should document (for the application API user to see) what CCW command codes it recognises and what the associated data of the CCW (if any) is used for
  - may well be simply be "CCW command code 1 is Write" (when "Write" has an obvious device-specific meaning) and/or "CCW command code 2 is Read" (when "Read" has an obivous device-specific meaning).
  - More complex device drivers may go wild with many different recognised command codes and data segment formats.
  - Command codes available to device drivers are 1 to 239 (0xef) with even ones being Read-type (application reads from device) and odd ones being Write-type (application writes to device).

Since there are not yet enough code comment Doxygen annotations to divide the generated documentation into topics properly, there follows a summary of the main definitions (e.g. types, macros and API functions) for use by CU-side code. They are described in the Doxygen-generated documentation but some may be in a Topics sub-section, some may be in "Data Structures" and some may be under "Files".

5.0.2 Types

```
typedef struct pch_cu pch_cu_t;

typedef uint8_t pch_cbindex_t;

typedef struct pch_devib pch_devib_t;

typedef void (*pch_devib_callback_t)(pch_cu_t *cu, pch_devib_t *devib);

typedef struct pch_dev_sense pch_dev_sense_t;
```

5.0.3 Compile-time constants and definitions - examples:

```
#define PCH_NUM_CUS 2
```

5.0.4 Debugging assertions:

```
#define PARAM_ASSERTIONS_ENABLED_PCH_CUS 1
#define PARAM_ASSERTIONS_ENABLED_PCH_DMACHAN 1
#define PARAM_ASSERTIONS_ENABLED_PCH_TXSM 1
```

5.0.5 Initialisation of whole CU subsystem

```
void pch_cus_init(void);

bool pch_cus_set_trace(bool trace);

pch_cbindex_t pch_register_unused_devib_callback(pch_devib_callback_t cb);

// Optionally configure explicit DMA IRQ index(es) (or leave to auto-configure)
void pch_cus_configure_dma_irq_index_exclusive(pch_dma_irq_index_t dmairqix);
void pch_cus_configure_dma_irq_index_shared(pch_dma_irq_index_t dmairqix, uint8_t order_priority);
void pch_cus_configure_dma_irq_index_shared_default(pch_dma_irq_index_t dmairqix);
void pch_cus_ignore_dma_irq_index_t(pch_dma_irq_index_t dmairqix);
```

5.0.6 Initialisation of each CU

```
pch_cu_t foo_cu = PCH_CU_INIT(num_devibs);
// or, if num_devbs is not a compile-time constant, initialise at runtime with:
void pch_cu_init(pch_cu_t *cu, uint16_t num_devibs);

// register at a given control unit address:
pch_cu_register(pch_cu_t *cu, pch_cuaddr_t cua);

bool pch_cus_trace_cu(pch_cuaddr_t cua, bool trace);

// Configure connection as a UART channel...:
void pch_cus_auto_configure_uartcu(pch_cuaddr_t cua, uart_inst_t *uart, uint baudrate);
// ...or a memory channel (needs extra configuration):
void pch_cus_memcu_configure(pch_cuaddr_t cua, pch_dmaid_t txdmaid, pch_dmaid_t rxdmaid,
      dmachan_tx_channel_t *txpeer);

// Start CU. Returns immediately after setting all CU handling to
// happen via interrupt handlers and callbacks from those so follow
// with an infinite "__wfe()" loop if there is nothing else to be
// done from main().
void pch_cu_start(pch_cuaddr_t cua);
```

5.0.7 Convenience API for device driver to its CU

5.0.7.1 Convenience API with fully general arguments

```
int pch_dev_set_callback(pch_devib_t *devib, int cbindex_opt);
int pch_dev_call_or_reject_then(pch_devib_t *devib, pch_dev_call_func_t f, int reject_cbindex_opt);
void pch_dev_call_final_then(pch_devib_t *devib, pch_dev_call_func_t f, int cbindex_opt);

int pch_dev_send_then(pch_devib_t *devib, void *srcaddr, uint16_t n, proto_chop_flags_t flags, int
      cbindex_opt);
int pch_dev_send_zeroes_then(pch_devib_t *devib, uint16_t n, proto_chop_flags_t flags, int cbindex_opt);
int pch_dev_receive_then(pch_devib_t *devib, void *dstaddr, uint16_t size, int cbindex_opt);
int pch_dev_update_status_advert_then(pch_devib_t *devib, uint8_t devs, void *dstaddr, uint16_t size, int
      cbindex_opt);
```

5.0.7.2 Convenience API with some fixed arguments

- Omitting `_then` avoids setting devib callback by hardcoding -1 as the `cbindex_opt` argument of the full `_then` function.
- For `send` and `send_zeroes` family, the `flags` argument is set to
  - `PROTO_CHOP_FLAG_END` for the `_final` variant,
  - `PROTO_CHOP_FLAG_RESPONSE_REQUIRED` for the _respond variant
  - 0 for the `_norespond` variant
- For `pch_dev_update_status_ok` family, call the corresponding `pch_dev_update_status_`↩ function with `DeviceEnd|ChannelEnd`
- For `pch_dev_update_status_error` family, set `devib->sense` to the `sense` argument then call the corresponding `pch_dev_update_status_` function with a device status of `DeviceEnd|`↩ `ChannelEnd|UnitCheck`

```
int pch_dev_send(pch_devib_t *devib, void *srcaddr, uint16_t n, proto_chop_flags_t flags);
int pch_dev_send_final(pch_devib_t *devib, void *srcaddr, uint16_t n);
int pch_dev_send_final_then(pch_devib_t *devib, void *srcaddr, uint16_t n, int cbindex_opt);
int pch_dev_send_respond(pch_devib_t *devib, void *srcaddr, uint16_t n);
int pch_dev_send_respond_then(pch_devib_t *devib, void *srcaddr, uint16_t n, int cbindex_opt);
int pch_dev_send_norespond(pch_devib_t *devib, void *srcaddr, uint16_t n);
int pch_dev_send_norespond_then(pch_devib_t *devib, void *srcaddr, uint16_t n, int cbindex_opt);
int pch_dev_send_zeroes(pch_devib_t *devib, uint16_t n, proto_chop_flags_t flags);
int pch_dev_send_zeroes_respond_then(pch_devib_t *devib, uint16_t n, int cbindex_opt);
int pch_dev_send_zeroes_respond(pch_devib_t *devib, uint16_t n);
int pch_dev_send_zeroes_norespond_then(pch_devib_t *devib, uint16_t n, int cbindex_opt);
int pch_dev_send_zeroes_norespond(pch_devib_t *devib, uint16_t n);
int pch_dev_receive(pch_devib_t *devib, void *dstaddr, uint16_t size);
int pch_dev_update_status_then(pch_devib_t *devib, uint8_t devs, int cbindex_opt);
int pch_dev_update_status(pch_devib_t *devib, uint8_t devs);
int pch_dev_update_status_advert(pch_devib_t *devib, uint8_t devs, void *dstaddr, uint16_t size);
int pch_dev_update_status_ok_then(pch_devib_t *devib, int cbindex_opt);
int pch_dev_update_status_ok(pch_devib_t *devib);
int pch_dev_update_status_ok_advert(pch_devib_t *devib, void *dstaddr, uint16_t size);
int pch_dev_update_status_error_advert_then(pch_devib_t *devib, pch_dev_sense_t sense, void *dstaddr,
      uint16_t size, int cbindex_opt);
int pch_dev_update_status_error_then(pch_devib_t *devib, pch_dev_sense_t sense, int cbindex_opt);
int pch_dev_update_status_error_advert(pch_devib_t *devib, pch_dev_sense_t sense, void *dstaddr, uint16_t
      size);
int pch_dev_update_status_error(pch_devib_t *devib, pch_dev_sense_t sense);
```

5.0.8 Low-level API for device driver to its CU

The Convenience API functions above use this low-level API and are more likely to be suitable instead of using these directly.

```
static inline void pch_devib_prepare_callback(pch_devib_t *devib, pch_cbindex_t cbindex);
static inline void pch_devib_prepare_count(pch_devib_t *devib, uint16_t count);
static inline void pch_devib_prepare_write_data(pch_devib_t *devib, void *srcaddr, uint16_t n,
      proto_chop_flags_t flags);
static inline void pch_devib_prepare_write_zeroes(pch_devib_t *devib, uint16_t n, proto_chop_flags_t flags);
static inline void pch_devib_prepare_read_data(pch_devib_t *devib, void *dstaddr, uint16_t size);
void pch_devib_prepare_update_status(pch_devib_t *devib, uint8_t devs, void *dstaddr, uint16_t size);
void pch_devib_send_or_queue_command(pch_devib_t *devib);
```

Chapter 6

Design of Picochan

6.0.1 Design

- CSS (Channel Subsystem) runs on one core. One CSS only.
- Application API calls functions from this core
  - All application API calls are short and non-blocking (dozens of cycles), just set some bits, update linked lists and raise an IRQ
  - CSS runs from IRQ handlers (short, non-blocking - dozens of cycles, prod Pico peripherals (e.g. UART, PIO, DMA), no timing constraints
- Each Control Unit (CU) runs on its own core (same or different Pico)
- Can be just one CU or up to 256
- Each has a "Control Unit number" (CU number), 0-255
- Each CU can address up to 256 devices
- Each device on a CU has a "unit address" 0-255
- Connection between CU and its CSS is via a *channel*
- Currently implemented channels are:
  - uart channel ("uartchan")
    - uses one Pico UART on CSS and one on CU side
    - hardware connections: TX, RX, RTS, CTS, GND
    - RTS and CTS are absolutely required
  - memory channel ("memchan")
    - between two cores on same Pico: one core runs CSS; one core runs CU
    - no hardware connections needed
- An additional channel type is in development:
  - pio channel ("piochan")
    - uses PIO to drive the CSS<->CU protocol
    - hardware connections: TX, RX, CLK, RTS, CTS, GND
    - custom PIO state machine programs - all connections absolutely required
    - The CSS-driven synchronous clock and the customised protocol handling by the PIO on both CSS and CU side may allow for a faster and/or more robust connection than a uart channel

6.0.2 CSS <-> CU protocol

- CSS<->CU protocol is custom for Picochan - none of the CSS <-> CU connectivity and protocol options used for actual mainframes in the past or present (parallel channels, ESCON, FICON) is suitable for consideration for use with a microcontroller
- 4-byte operation command packets
  - 4-bit command
  - 4-bit flags
  - 8-bit unit address
  - 16-bit payload - operation specific, e.g. data segment count, CCW command or device status and encoded advertised room
- Operation commands:
  - Start (CSS -> CU) - start(/continue) a channel program
  - UpdateStatus (CU -> CSS) - end/progress a channel program or unsolicited notification to CSS of device state change (e.g. "ready")
  - RequestRead (CU -> CSS) - please send data from (Write-type) CCW
  - Data - immediately followed by bytes of data as per the count from the payload of the operations packet. Both CSS->CU (for responses to RequestRead) and CU->CSS (for transfer down the channel for the CSS to write to a segment of a Read-type CCW)
  - Signal (CSS -> CU) - mainly for "halt subchannel" (out-of-band)
- All channel types use DMA for data segment transfer to/from channel
- Channels are (for pio and uart channels) hardware FIFOs direct to/from Pico peripherals or (for mem channel) a single cross-memory 32-bit load/store with cross-memory DMA for data segments
- CSS represents each device to the application as a "subchannel"
  - A subchannel is represented in the CSS as a "Subchannel Information Block" (SCHIB), `pch_schib_t` (a 32-byte structure)
  - The CSS has a global array of SCHIBs (fixed size chosen at compile-time), addressed by a "Subsystem identification word" (SID)
  - SID is a 16-bit integer (`pch_sid_t` typdef for uint16_t)
  - The schib has fields for the device's control unit number and unit address for the CSS to use to contact the device's CU and identify a chosen device to that CU

Chapter 7

Tracing

7.0.1 Introduction

- Optional tracing to help debugging of CSS, CU and their users
- Code only present when compile-time `PCH_CONFIG_ENABLE_TRACE` `#define`d as non-zero
- Trace records are written for various events in CSS and CU when appropriate trace flags are set
- Trace records are small (∼8-28 bytes) binary structs written to a small set of statically allocated buffers (a `bufferset`) that is treated as a ring buffer
- A bufferset is compile-time defined for one or both of CSS (if used) and CUs (if used) as, by default, 2 x 1KB buffers. Offloading traces for processing (see below) if the buffers are consecutive in memory (e.g. a single 2KB chunk).
- Trace records consist of a 48-bit timestamp (microseconds since boot), an 8-bit "trace record type" and an 8-bit count of associated data
- When each individual trace buffer becomes full, an IRQ can optionally be raised so that the application can fetch and offload that buffer's data before the other buffer(s) in the bufferset fill and the ring returns to restart writing to the just-filled buffer.
- The resulting data in the bufferset is expected to be offloaded and processed off-platform
- Offloading the necessary data can be done simply by using openocd or gdb (when SWD access is available) to fetch
  - the 32-byte metadata global variables `CSS.trace_bs` (CSS) or `pch_cus_trace_bs` (for CU)
  - the trace buffers themselves

7.0.2 `pch_dump_trace` - parse and display traces (off-platform)

A `pch_dump_trace` program is provided - C source in the `tools/pch_dump_trace.c` directory of the Picochan repository - which is expected to be (compiled and) run off-platform rather than on the Pico itself.

`pch_dump_trace` takes two filenames as input which are expected to contain

- the raw data from the 32-byte bufferset structure

- the concetenated raw data from the trace buffers. This can simply be the contents of a single global `unsigned char pch_css_trace_buffer_space[]` array if there is room to have the buffers contigous (e.g. as 2KB instead of separate 1KB and 1KB)

`pch_dump_trace` parses the binary trace record data and, by default, extracts the trace record fields and explains them in human-readable output (although an option for raw "dump the timestamps, record type name/numbers and data in hex format" is available). For example, an extract from a basic test (with the UART channel intentionally slowed to 1200 baud), reformatted slightly (to combine the separate CSS and CU traces with indicators "+" for CSS-side and "-" for CU-side):

```
01:02.707412 +CSS Function IRQ raised for CU=0 with pending UA=4 while tx_active=0
01:02.707441 +CSS CCW fetch for SID:0004 CCW address=20003000 provides CCW{cmd:03 flags:40 count=4
      addr:20003300}
01:02.707474 +CSS-side SID:0004 sends packet{Start ua=4 CCWcmd:03 count=0(exact)}
01:02.707491 +CU tx channel DMAid=0 sets source to cmdbuf
01:02.707522 +IRQ for CSS-side CU=0 with DMA_IRQ_0 tx:irq_state=raised+complete,mem_src_state=idle
      rx:irq_state=none,mem_dst_state=idle
01:02.707537 +CSS-side CU=0 handling tx complete while txsm is idle
01:02.707565 +start subchannel SID:0004 CCW address=20003000 cc=0
01:02.707580 +test subchannel SID:0004 cc=1
01:02.707587 +test subchannel SID:0004 cc=1
01:02.743898 -IRQ for dev-side CU=0 with DMA_IRQ_1 tx:irq_state=none,mem_src_state=idle
      rx:irq_state=raised+complete,mem_dst_state=idle
01:02.743922 -dev-side CU=0 UA=4 received packet{Start ua=4 CCWcmd:03 count=0(exact)}
01:02.743953 -CU rx channel DMAid=3 sets destination to cmdbuf
01:02.743963 -dev-side CU=0 calls callback 1 for UA=4
01:02.744007 -dev-side CU=0 UA=4 sends packet{RequestRead ua=4 count=4}
01:02.744017 -CU tx channel DMAid=2 sets source to cmdbuf
01:02.744030 -IRQ for dev-side CU=0 with DMA_IRQ_1 tx:irq_state=raised+complete,mem_src_state=idle
      rx:irq_state=none,mem_dst_state=idle
01:02.744040 -dev-side CU=0 handling tx complete while txsm is idle
01:02.780445 +IRQ for CSS-side CU=0 with DMA_IRQ_0 tx:irq_state=none,mem_src_state=idle
      rx:irq_state=raised+complete,mem_dst_state=idle
01:02.780460 +CSS-side SID:0004 received packet{RequestRead ua=4 count=4}
01:02.780478 +CSS-side SID:0004 sends packet{Data|End ua=4 count=4}
```

7.0.3 Interactive "offload trace buffers and parse/display" with gdb

For gdb, an example when the buffers are defined as a single contiguous array:

```
unsigned char pch_css_trace_buffer_space[PCH_TRC_NUM_BUFFERS * PCH_TRC_BUFFER_SIZE] __aligned(4);
```

the following gdb definitions fetch and dump the current trace buffers to host-local files and run the `pch_dump_↩ trace` program on the results (see below) to parse and display human-readable explanations of the traced events:

```
define pch-show-css-trace
  dump binary value /tmp/gdb-css.bs CSS.trace_bs
  dump binary value /tmp/gdb-css.bufs pch_css_trace_buffer_space
  shell pch_dump_trace /tmp/gdb-css.bs /tmp/gdb-css.bufs
end
document pch-show-css-trace
  Dumps CSS trace buffers and uses pch_dump_trace to show them
end

define pch-show-cus-trace
  dump binary value /tmp/gdb-cus.bs pch_cus_trace_bs
  dump binary value /tmp/gdb-cus.bufs pch_cus_trace_buffer_space
  shell pch_dump_trace /tmp/gdb-cus.bs /tmp/gdb-cus.bufs
end
document pch-show-cus-trace
  Dumps CU trace buffers and uses pch_dump_trace to show them
end
```

7.0.4 API to enable/disable trace at various levels

- The compile-time default is that no tracing code is present at all
- To include the ability to enable tracing,

```
#define PCH_CONFIG_ENABLE_TRACE 1
```

- To change the default of 2 x 1KB buffers in each bufferset (where CSS, if present, uses one bufferset and CUs, if present, use one bufferset between them), define `PCH_TRC_NUM_BUFFERS` (default 2) and/or `PCH_TRC_BUFFER_SIZE` (default 1024) to different compile-time constants, e.g.

```
#define PCH_TRC_NUM_BUFFERS 3
#define PCH_TRC_BUFFER_SIZE 2048
```

7.0.4.1 Tracing for CSS

- No trace records are written at all unless/until `pch_css_set_trace(true)` is called, typically done immediately after `pch_css_init()`. With this enabled, trace records for CSS-global events are written.
- To enable trace records related to a given channel to be written, call `pch_chp_set_trace(chpid, true)`. If the trace flag is not set for a channel then no trace records for any subchannel on that channel are written. With the trace flag for a channel enabled, non-subchannel-specific trace records related to the channel are written.
- To enable trace records related to a given subchannel, set the `PCH_PMCW_TRACED` flag bit in the subchannel's `PMCW`, e.g. to set the trace flag at the same time as subchannel `sid` is enabled:

```
uint16_t flags = PCH_PMCW_ENABLED | PCH_PMCW_TRACED;
pch_sch_modify_flags(sid, flags);
```

7.0.4.2 Tracing for CU

- No trace records are written at all unless/until `pch_cus_set_trace(true)` is called, typically done immediately after `pch_cus_init()`.
- To enable trace records related to a given CU and all its devices to be written, call `pch_cus_trace_cu(cua, true)`. Unlike for the CSS API, setting the trace flag at CU level enables trace records for all its devices.
- To enable trace records related to a given device, set the `PCH_DEVIB_FLAG_TRACED` flag bit in the `devib`, e.g. with `pch_devib_set_traced(devib, true)`. With the `PCH_DEVIB_FLAG_TRACED` bit present in the `flags` field of a `devib` and the CU-global trace flag set (with `pch_cus_set_trace()`), records will be written for events related to the device regardless of whether the trace flag for its CU has been set with `pch_cus_trace_cu(cua, val)`.

Chapter 8

Topic Index

8.1 Topics

Here is a list of all topics with brief descriptions:

Chapter 9

Data Structure Index

9.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 10

File Index

10.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 11

Topic Documentation

11.1 picochan_base

The basic types used by picochan throughout both CSS and CU.

Files

- file bsize.h

  An encoding of 16-bit counts as 8-bit values for typical Pico-sized buffers.
- file ccw.h

  Channel-Command Word (CCW)
- file schib.h

  The Subchannel Information Block (SCHIB)
- file dev_sense.h

  Device sense.

Data Structures

- struct pch_bsizex

  a pch_bsize together with a flag intended to indicate whether the bsize encoded the original size exactly.
- struct pch_ccw

  I/O Channel-Command Word (CCW)
- struct pch_schib

  pch_schib_t is the Subchannel Information Block (SCHIB)
- struct pch_dev_sense

  The device sense structure by which a device can communicate additional error information on request by the CSS.

Macros

- #define PCH_BSIZE_ZERO ((pch_bsize_t){0})

  A constant struct initialiser for the bsize encoding of zero.

Typedefs

- typedef struct pch_bsizex pch_bsizex_t

  a pch_bsize together with a flag intended to indicate whether the bsize encoded the original size exactly.
- typedef uint8_t **pch_ccw_flags_t**

  the flags of a CCW
- typedef struct pch_ccw pch_ccw_t

  I/O Channel-Command Word (CCW)
- typedef uint16_t **pch_sid_t**

  a subchannel id (SID) between 0 and PCH_NUM_SCHIBS-1 (at most 65535)
- typedef uint8_t **pch_cuaddr_t**

  a control unit address between 0 and PCH_NUM_CUS-1 (at most 255) that identifies a control unit from the CU side.
- typedef uint8_t pch_unit_addr_t

  a unit address that identifies a device on a given CU on the control unit side.
- typedef uint8_t pch_chpid_t

  a channel path identifier between 0 and PCH_NUM_CHANNELS-1 (at most 255) that identifies a channel from the CSS side
- typedef uint8_t pch_dmaid_t

  a DMA id used by CSS or CU
- typedef int8_t pch_dma_irq_index_t

  a DMA IRQ index
- typedef struct pch_schib pch_schib_t

  pch_schib_t is the Subchannel Information Block (SCHIB)
- typedef struct pch_dev_sense **pch_dev_sense_t**

  The device sense structure by which a device can communicate additional error information on request by the CSS.

Functions

- pch_bsizex_t **pch_bsize_encodex** (uint16_t n)

  Encode 16-bit count as an pch_bsizex_t.
- pch_bsize_t **pch_bsize_encode** (uint16_t n)

  Encode 16-bit count as an 8-bit pch_bsize_t.
- uint16_t **pch_bsize_decode_raw** (uint8_t esize)

  Decode an 8-bit raw value of a bsize (not in its pch_bsize_t type-wrapping) into a 16-bit value.
- uint16_t **pch_bsize_decode** (pch_bsize_t bsize)

  Decode an 8-bit pch_bsize_t value into a 16-bit value.
- static uint8_t **pch_bsize_unwrap** (pch_bsize_t s)

  Unwraps the uint8_t contained in a pch_bsize_t.
- static pch_bsize_t pch_bsize_wrap (uint8_t esize)

  wraps a uint8_t into a pch_bsize_t
- static uint8_t pch_bsize_encode_raw_inline (uint16_t n)

  Perform a bsize encoding, returning the encoded value unwrapped.
- static pch_bsizex_t pch_bsize_encodex_inline (uint16_t n)

  encode a 16-bit value into its pch_bsize_t along with an "exact"
- static pch_bsize_t pch_bsize_encode_inline (uint16_t n)

  encode a 16-bit value as a pch_bsize_t
- static uint16_t pch_bsize_decode_raw_inline (uint8_t esize)

  decodes a raw bsize-encoded value
- static uint16_t pch_bsize_decode_inline (pch_bsize_t bsize)

  decodes a pch_bsize_t as the uint16_t it represents
- uint8_t **pch_bsize_encode_raw** (uint16_t n)

  Encode a 16-bit value into its raw 8-bit bsize encoding.

11.1.1 Detailed Description

The basic types used by picochan throughout both CSS and CU.

The subchannel-status word (SCSW)

The I/O interruption code.

11.1.2 Macro Definition Documentation

11.1.2.1 PCH_BSIZE_ZERO

```
#define PCH_BSIZE_ZERO ((pch_bsize_t){0})
```

A constant struct initialiser for the bsize encoding of zero.

This is simply a constant structure initialiser (the structure itself, not a pointer to a structure) containing a single byte of zero which is the bsize encoding of zero.

11.1.3 Typedef Documentation

11.1.3.1 pch_bsizex_t

```
typedef struct pch_bsizex pch_bsizex_t
```

a pch_bsize together with a flag intended to indicate whether the bsize encoded the original size exactly.

The flag is the low bit of the exact field. It is defined as a uint8_t rather than a bool to make its position clearer in any stored value of the structure.

11.1.3.2 pch_ccw_t

```
typedef struct pch_ccw pch_ccw_t
```

I/O Channel-Command Word (CCW)

pch_ccw_t is an architected 8-byte control block that must be 4-byte aligned. When marshalling/unmarshalling a CCW, unlike the original architected Format-1 CCW which was implicitly big-endian, the count and addr fields here are treated as native-endian and so will be little-endian on both ARM and RISC-V (in Pico configurations) and would also be so on x86, for example.

```
CCW +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |      cmd       |      flags      |            count               |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |                        data address                               |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

11.1.3.3 pch_chpid_t

```
typedef uint8_t pch_chpid_t
```

a channel path identifier between 0 and PCH_NUM_CHANNELS-1 (at most 255) that identifies a channel from the CSS side

Each channel connects to a single remote CU.

11.1.3.4 pch_dma_irq_index_t

```
typedef int8_t pch_dma_irq_index_t
```

a DMA IRQ index

Must be either -1 (meaning no DMA IRQ index set) or between 0 and the number of DMA IRQs on the platform (e.g. 2 for RP2040 and 4 for RP2350). Pico SDK uses the uint type for DMA IRQ index arguments but Picochan uses the pch_dma_irq_index_t type in its API and also for storing them so it can use a single byte instead of four.

11.1.3.5 pch_dmaid_t

```
typedef uint8_t pch_dmaid_t
```

a DMA id used by CSS or CU

Must be between 0 and the number of DMA channels on the platform. Pico SDK uses the uint type for DMA channel id arguments but picochan uses pch_dmaid_t type in its API and also for storing them in a single byte instead of four.

11.1.3.6 pch_schib_t

```
typedef struct pch_schib pch_schib_t
```

pch_schib_t is the Subchannel Information Block (SCHIB)

The SCHIB is formed from the Path Management Control Word (PMCW), Subchannel Status Word (SCSW) and Model Dependent Area (MDA). Of these, the PMCW and SCSW are architected formats and the MDA format is an internal implementation detail of the CSS.

```
PMCW    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |                            Intparm                            |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |                     |T|E| ISC |      CUAddr   | UnitAddr      |
SCSW    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |               | CC|P|I|U|Z| |N|W|  FC |     AC      |   SC    |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |                         CCW Address                           |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        | DEVS/ccwflags |     SCHS      |     Residual Count            |
MDA     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |                        data address                           |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |        reqcount/advcount      | prevua/ccwcmd |    nextua     |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |           prevsid             |           nextsid             |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

DEVS only needs to be valid when SC.StatusPending is set. Otherwise, we use the field to hold the current ccwflags.

11.1.3.7 pch_unit_addr_t

```
typedef uint8_t pch_unit_addr_t
```

a unit address that identifies a device on a given CU on the control unit side.

Must be between 0 and cu->num_devibs-1 (which is at most 255).

11.1.4 Function Documentation

11.1.4.1 pch_bsize_decode_inline()

```
uint16_t pch_bsize_decode_inline (
            pch_bsize_t bsize)  [inline], [static]
```

decodes a pch_bsize_t as the uint16_t it represents

This function is declared as "static inline" to be used in places where it is appropriate to have the code inlined. A corresponding function pch_bsize_encodex is available as an ordinary function.

11.1.4.2 pch_bsize_decode_raw_inline()

```
uint16_t pch_bsize_decode_raw_inline (
            uint8_t esize)  [inline], [static]
```

decodes a raw bsize-encoded value

This is a shortcut for pch_bsize_decode(pch_bsize_wrap(esize) which can be used when the benefits of the type-wrapping of the encoding are not needed.

11.1.4.3 pch_bsize_encode_inline()

```
pch_bsize_t pch_bsize_encode_inline (
            uint16_t n)  [inline], [static]
```

encode a 16-bit value as a pch_bsize_t

This does the same as pch_bsize_encodex_inline but does not return the exactness.

11.1.4.4 pch_bsize_encode_raw_inline()

```
uint8_t pch_bsize_encode_raw_inline (
            uint16_t n)  [inline], [static]
```

Perform a bsize encoding, returning the encoded value unwrapped.

This is a shortcut for pch_bsize_unwrap(pch_bsize_encode(size)) which can be used when the benefits of the type-wrapping of the encoding are not needed.

11.1.4.5 pch_bsize_encodex_inline()

```
pch_bsizex_t pch_bsize_encodex_inline (
             uint16_t n)  [inline], [static]
```

encode a 16-bit value into its pch_bsize_t along with an "exact"

This encodes n into its pch_bsize_t along with a flag bit that indicates whether decoding the result will produce exactly n.

This function is declared as "static inline" to be used in places where it is appropriate to have the code inlined. A corresponding function pch_bsize_encodex is available as an ordinary function.

11.1.4.6 pch_bsize_wrap()

```
pch_bsize_t pch_bsize_wrap (
             uint8_t esize)  [inline], [static]
```

wraps a uint8_t into a pch_bsize_t

This is typically used to produce a clearly-typed "bsize encoded" value after receiving an unwrapped bsize from a remote protocol

11.2 internal_trc

Internal. Tracing subsystem used by both CSS and CU.

Data Structures

- struct pch_trc_timestamp

  an opaque timestamp of a 48-bit number of microseconds since boot.
- struct pch_trc_bufferset

  set of buffers and metadata for a subsystem to use tracing

Macros

- #define PCH_CONFIG_ENABLE_TRACE 0

  Whether any tracing code should be compiled at at..

Typedefs

- typedef struct pch_trc_timestamp pch_trc_timestamp_t

  an opaque timestamp of a 48-bit number of microseconds since boot.
- typedef struct pch_trc_bufferset pch_trc_bufferset_t

  set of buffers and metadata for a subsystem to use tracing

11.2.1 Detailed Description

Internal. Tracing subsystem used by both CSS and CU.

11.2.2 Macro Definition Documentation

11.2.2.1 PCH_CONFIG_ENABLE_TRACE

```
#define PCH_CONFIG_ENABLE_TRACE 0
```

Whether any tracing code should be compiled at at..

Set to a compile-time non-empty non-zero constant to enable. Default 0.

11.2.3 Typedef Documentation

11.2.3.1 pch_trc_bufferset_t

```
typedef struct pch_trc_bufferset pch_trc_bufferset_t
```

set of buffers and metadata for a subsystem to use tracing

This struct holds an array of PCH_TRC_NUM_BUFFERS buffers, each which must be of size PCH_TRC_↩
BUFFER_SIZE.

When compile-time trace support is enabled (PCH_CONFIG_ENABLE_TRACE is defined to be non-zero), PCH↩
_TRC_NUM_BUFFERS is the number of trace buffers in a bufferset. These buffers form a ring - once the current buffer is full, the current buffer moves onto the next in the ring and, optionally, an interrupt is generated so that the previous buffer can be archived elsewhere before the ring wraps.

When compile-time trace support is not enabled, PCH_TRC_NUM_BUFFERS is defined as 0 so this struct can be instantiated but not used.

11.2.3.2 pch_trc_timestamp_t

```
typedef struct pch_trc_timestamp pch_trc_timestamp_t
```

an opaque timestamp of a 48-bit number of microseconds since boot.

The actual value is held as three consecutive 16-bit chunks (forming a little-endian encoding of the whole value) but the intended way of accessing the value is with pch_trc_timestamp_to_us().

11.3 internal_proto

The internal protocol between CSS and CU.

Data Structures

- struct proto_packet

  a 4-byte command packet sent on a channel between CSS and CU or vice versa

Typedefs

- typedef struct proto_packet proto_packet_t

  a 4-byte command packet sent on a channel between CSS and CU or vice versa

11.3.1 Detailed Description

The internal protocol between CSS and CU.

11.3.2 Typedef Documentation

11.3.2.1 proto_packet_t

```
typedef struct proto_packet proto_packet_t
```

a 4-byte command packet sent on a channel between CSS and CU or vice versa

Various parts of this implementation are tuned for and rely on the size being exactly 4 bytes. Note that the ARM ABI specifies that a return value of a composite type of up to 4 bytes (such as proto_packet_t) is passed in R0, thus behaving the same way as a 32-bit return value.

11.4 internal_css

A (CSS-side) channel that connects to a remote CU.

Data Structures

- struct pch_chp

  pch_chp_t is the CSS-side representation of a channel path to a control unit.

Typedefs

- typedef struct pch_chp pch_chp_t

  pch_chp_t is the CSS-side representation of a channel path to a control unit.

11.4.1 Detailed Description

A (CSS-side) channel that connects to a remote CU.

internal CSS implementations

11.4.2 Typedef Documentation

11.4.2.1 pch_chp_t

```
typedef struct pch_chp pch_chp_t
```

pch_chp_t is the CSS-side representation of a channel path to a control unit.

The application API usually refers to these by a channel path id (CHPID) which indexes into the global array CSS.↩ chps and so does not really need to care about the details of this struct. Currently, a channel only connects to a single control unit so the pch_chp_t is effectively a CSS-side "peer" object of the dev-side CU, pch_cu_t.

11.5 picochan_css

Channel Subsystem (CSS)

Files

- file pmcw.h

  The Path Management Control World (PMCW)

Macros

- #define PCH_NUM_SCHIBS

  The number of subchannels.
- #define PCH_NUM_CHANNELS

  The number of channels that the CSS can use.
- #define PCH_NUM_ISCS

  The number of interrupt service classes.

Typedefs

- typedef void(∗ **io_callback_t**) (pch_intcode_t, pch_scsw_t)

  A callback function to be invoked when a subchannel becomes status pending.

Functions

- static void ∗ pch_ccw_get_addr (pch_ccw_t ccw)

  Get the addr field of a CCW as a pointer.
- void pch_css_init (void)

  Initialise CSS.
- void pch_css_set_func_irq (irq_num_t irqnum)

  Low-level function to set the IRQ number that the CSS uses for application API notification to CSS.
- void pch_css_set_io_irq (irq_num_t irqnum)

  Low-level function to set the IRQ number that the CSS uses for I/O interrupt notification.
- io_callback_t pch_css_set_io_callback (io_callback_t io_callback)

  Low-level function to set the I/O callback function that the CSS invokes if its I/O interrupt handler has been set to pch_css_io_irq_handler. pch_css_start(io_callback, isc_mask) with io_callback non-NULL).
- void pch_css_start (io_callback_t io_callback, uint8_t isc_mask)

  Starts CSS operation after setting the io_callback (if non-NULL), configuring and enabling any needed CSS IRQ handlers that have not yet been set and setting the mask of ISCs that trigger I/O interrupts to be isc_mask.
- bool pch_css_set_trace (bool trace)

  Sets whether CSS tracing is enabled.
- bool pch_chp_set_trace (pch_chpid_t chpid, bool trace)

  Sets whether CSS tracing is enabled for channel chpid.
- void pch_chp_start (pch_chpid_t chpid)

  Starts channel chpid connection to its remote CU.
- void **pch_chp_claim** (pch_chpid_t chpid)

  Mark channel path chpid as claimed. Panics if it is already claimed or allocated.
- int **pch_chp_claim_unused** (bool required)

  Claims the next unclaimed and unallocated channel path and returns its CHPID (a pch_chpid_t cast to int). If no channel path is available, panics if required is true or else returns -1.
- pch_sid_t pch_chp_alloc (pch_chpid_t chpid, uint16_t num_devices)

  Allocates num_devices schibs for use by channel chpid.
- void pch_chp_configure_uartchan (pch_chpid_t chpid, uart_inst_t ∗uart, dma_channel_config ctrl)

  Configure a UART channel.
- void pch_chp_auto_configure_uartchan (pch_chpid_t chpid, uart_inst_t ∗uart, uint baudrate)

  Initialise and configure a hardware UART instance as a channel to the remote CU to which it is connected. Uses a default dma_channel_config control register.
- void pch_chp_configure_memchan (pch_chpid_t chpid, pch_dmaid_t txdmaid, pch_dmaid_t rxdmaid, dmachan_tx_channel_t ∗txpeer)

  Configure a memchan channel.
- dmachan_tx_channel_t ∗ pch_chp_get_tx_channel (pch_chpid_t chpid)

  Fetch the internal tx side of a channel from CSS to CU.
- int pch_sch_start (pch_sid_t sid, pch_ccw_t ∗ccw_addr)

  Start a channel program for a subchannel.
- int pch_sch_resume (pch_sid_t sid)

  Resume a channel program for a subchannel.
- int pch_sch_test (pch_sid_t sid, pch_scsw_t ∗scsw)

  Test the status of a subchannel, clearing various status conditions of status is pending.
- int pch_sch_modify (pch_sid_t sid, pch_pmcw_t ∗pmcw)

  Modifies the PMCW field of a subchannel.
- int pch_sch_store (pch_sid_t sid, pch_schib_t ∗out_schib)

  Stores the contents of the schib for subchannel sid to out_schib.
- int pch_sch_cancel (pch_sid_t sid)

  Cancel a channel program that has not yet started.
- int pch_sch_halt (pch_sid_t sid)

Halt a channel program.
- pch_intcode_t pch_test_pending_interruption (void)

    Test if there is a pending I/O interruption.
- int pch_sch_store_pmcw (pch_sid_t sid, pch_pmcw_t ∗out_pmcw)

    Stores the contents of the PMCW part of the schib for subchannel sid to out_pmcw.
- int pch_sch_store_scsw (pch_sid_t sid, pch_scsw_t ∗out_scsw)

    Stores the contents of the SCSW part of the schib for subchannel sid to out_scsw.
- int pch_sch_modify_intparm (pch_sid_t sid, uint32_t intparm)

    Modifies the intparm field of the PMCW part of the schib for subchannel sid.
- int pch_sch_modify_flags (pch_sid_t sid, uint16_t flags)

    Modifies the flags field of the PMCW part of the schib for subchannel sid.
- int pch_sch_modify_isc (pch_sid_t sid, uint8_t isc)

    Modifies the isc field of the PMCW part of the schib for subchannel sid.
- int pch_sch_modify_enabled (pch_sid_t sid, bool enabled)

    Modifies enabled flag of the schib for subchannel sid.
- int pch_sch_modify_traced (pch_sid_t sid, bool traced)

    Modifies traced flag of the schib for subchannel sid.
- **void** (pch_sid_t sid, uint count, uint8_t isc)

    Calls pch_sch_modify_isc() on count subchannels starting from sid, panicking if any call fails.
- void (pch_sid_t sid, uint count, bool enabled)

    Calls pch_sch_modify_enabled() on count subchannels starting from sid, panicking if any call fails.
- int pch_sch_wait (pch_sid_t sid, pch_scsw_t ∗scsw)

    Wait for an I/O interruption condition for subchannel sid.
- int pch_sch_wait_timeout (pch_sid_t sid, pch_scsw_t ∗scsw, absolute_time_t timeout_timestamp)

    Wait for an I/O interruption condition for subchannel sid with a timeout.
- int pch_sch_run_wait (pch_sid_t sid, pch_ccw_t ∗ccw_addr, pch_scsw_t ∗scsw)

    Start a channel program for a subchannel and wait for an I/O interruption condition.
- int pch_sch_run_wait_timeout (pch_sid_t sid, pch_ccw_t ∗ccw_addr, pch_scsw_t ∗scsw, absolute_time_↩
t timeout_timestamp)

    Start a channel program for a subchannel and wait for an I/O interruption condition with a timeout.
- void pch_cus_auto_configure_uartcu (pch_cuaddr_t cua, uart_inst_t ∗uart, uint baudrate)

    Initialise and configure a UART control unit with default dma_channel_config control register.

11.5.1 Detailed Description

Channel Subsystem (CSS)

11.5.2 Macro Definition Documentation

11.5.2.1 PCH_NUM_CHANNELS

```
#define PCH_NUM_CHANNELS
```

The number of channels that the CSS can use.

Must be a compile-time constant between 1 and 256. Default 4. One channel is needed to connect to each CU. Defines the size of the global array of CSS-side channel structures (see pch_chp_t).

11.5.2.2 PCH_NUM_ISCS

```
#define PCH_NUM_ISCS
```

The number of interrupt service classes.

Must be a compile-time constant between 1 and 8. Default 8. Defines the size of the global array of linked-list-headers for subchannels that are status pending.

11.5.2.3 PCH_NUM_SCHIBS

```
#define PCH_NUM_SCHIBS
```

The number of subchannels.

Must be a compile-time constant between 1 and 65536. Default 32. Defines the size of the global array of schibs (see pch_schib_t).

11.5.3 Function Documentation

11.5.3.1 pch_ccw_get_addr()

```
void * pch_ccw_get_addr (
            pch_ccw_t ccw)  [inline], [static]
```

Get the addr field of a CCW as a pointer.

This is a convenience function that cannot be put in ccw.h itself since the architected addr field is 32 bits and ccw.h must be usable on platforms where a (void∗) is longer without causing compiler warnings (for example for compiling pch_dump_trace off-platform).

11.5.3.2 pch_chp_alloc()

```
pch_sid_t pch_chp_alloc (
            pch_chpid_t chpid,
            uint16_t num_devices)
```

Allocates num_devices schibs for use by channel chpid.

Starting with the first unallocated schib in the CSS array of schibs, allocates num_devices consecutive schibs and initialises them to reference the devices with unit addresses 0 through num_devices-1 respectively on the CU to which channel chpid will connect. The total number of allocated schibs must not exceed the size of the array, PCH←
_NUM_SCHIBS. A check for this and other sanity checks on the arguments are made only if assertions are enabled. CSS must have been started (pch_css_start()) but this channel must not have been started yet (pch_chp_start()). Returns the SID of the first allocated schib.

11.5.3.3 pch_chp_auto_configure_uartchan()

```
void pch_chp_auto_configure_uartchan (
            pch_chpid_t chpid,
            uart_inst_t * uart,
            uint baudrate)
```

Initialise and configure a hardware UART instance as a channel to the remote CU to which it is connected. Uses a default dma_channel_config control register.

Calls pch_uart_init() with baud rate

Parameters

baudrate	and pch_chp_configure_uartchan with ctrl argument bits taken from an appropriate dma_channel_get_default_config() value. The CU on the other side of the channel *MUST* use the same baud rate and uart settings.

11.5.3.4 pch_chp_configure_memchan()

```
void pch_chp_configure_memchan (
            pch_chpid_t chpid,
            pch_dmaid_t txdmaid,
            pch_dmaid_t rxdmaid,
            dmachan_tx_channel_t * txpeer)
```

Configure a memchan channel.

A memchan channel allows the CSS to run on one core of a Pico while a CU runs on the other core. Instead of using physical pins or connections between CU and CSS, picochan uses the DMA channels to copy memory-to-memory between CSS and CU and an internal state machine and cross-core synchronisation to mediate CSS to CU communications. txdmaid and rxdmaid must be two unused DMA ids, typically allocated using dma_claim_↩ unused_channel(). In order for the CSS to find the CU-side information to cross-connect the sides in memory, the CU API function pch_cu_get_tx_channel() must be used to fetch the internal dmachan_tx_channel_t of the peer CU for passing to pch_chp_configure_memchan.

11.5.3.5 pch_chp_configure_uartchan()

```
void pch_chp_configure_uartchan (
            pch_chpid_t chpid,
            uart_inst_t * uart,
            dma_channel_config ctrl)
```

Configure a UART channel.

Configure the hardware UART instance uart as a channel to the remote CU to which it is connected. The UART must have been initialised already, be connected to a CU using the same baud rate as this channel has configured and the hardware flow control pins, CTS and RTS *MUST* be enabled and connected between channel and CU. ctrl should typically be a default dma_channel_config as returned from dma_channel_get_default_config(dmaid) invoked on any DMA id. Most bits in that dma_channel_config are overridden by the CSS (including the CHAIN↩ _TO which is why the dmaid above does not matter) but some applications may wish to set bits SNIFF_EN and HIGH_PRIORITY for their own purposes.

If you want to initialise and configure the UART channel using a given baud rate, suggested UART settings (8E1) and default DMA control register settings (no SNIFF_EN and no HIGH_PRIORITY), you can use pch_chp_auto_configure_uartchan() instead.

11.5.3.6 pch_chp_get_tx_channel()

```
dmachan_tx_channel_t * pch_chp_get_tx_channel (
            pch_chpid_t chpid)
```

Fetch the internal tx side of a channel from CSS to CU.

This function is only needed when configuring a memchan between a CSS and CU on different cores of a single Pico. The CU initialisation procedure uses this function to find its peer CSS structure in order to cross-connect the channels.

11.5.3.7 pch_chp_set_trace()

```
bool pch_chp_set_trace (
            pch_chpid_t chpid,
            bool trace)
```

Sets whether CSS tracing is enabled for channel chpid.

If this flag is not set to be true then no channel trace records are written for this channel and no subchannel trace records, regardless of any per-subchannel trace flags.

11.5.3.8 pch_chp_start()

```
void pch_chp_start (
            pch_chpid_t chpid)
```

Starts channel chpid connection to its remote CU.

The channel must be already configured but not have been started. Marks the channel as started and starts it, allowing it to receive commands from its remote CU.

11.5.3.9 pch_css_init()

```
void pch_css_init (
            void )
```

Initialise CSS.

Must be called before any other CSS function.

11.5.3.10 pch_css_set_func_irq()

```
void pch_css_set_func_irq (
            irq_num_t irqnum)
```

Low-level function to set the IRQ number that the CSS uses for application API notification to CSS.

Typically, should be a non-externally-used user IRQ (i.e. IRQ numbers 26-31 on RP2040 and IRQ numbers 46-51 (SPAREIRQ_IRQ0 through SPAREIRQ_IRQ5) on RP2350. In general, either the high-level convenience function pch_css_auto_configure_func_irq() should be used instead or, for mid-level control of the handler, variants on pch↩
_css_configure_func_irq...

11.5.3.11 pch_css_set_io_callback()

```
io_callback_t pch_css_set_io_callback (
            io_callback_t io_callback)
```

Low-level function to set the I/O callback function that the CSS invokes if its I/O interrupt handler has been set to pch_css_io_irq_handler. pch_css_start(io_callback, isc_mask) with io_callback non-NULL).

Sets a callback function which pch_css_io_irq_handler will invoke on subchannels with unmasked ISC and pending status.

Typically, this should instead be set implicitly by calling pch_css_start(io_callback, isc_mask) with io_callback non-NULL.

If pch_css_io_irq_handler is added as an ISR for the CSS I/O IRQ index (itself set with pch_css_set_io_irq), then when called, it pops each subchannel that is in an unmasked ISC and is status pending, retrieves the SCSW for that subchannel and calls the callback function.

Low-level function to set the I/O callback function that the CSS invokes if its I/O interrupt handler has been set to pch_css_io_irq_handler. pch_css_start(io_callback, isc_mask) with io_callback non-NULL).

If pch_css_io_irq_handler is added as an ISR for the CSS I/O IRQ index (itself set with pch_css_set_io_irq), then when called, it pops each subchannel that is in an unmasked ISC and is status pending, retrieves the SCSW for that subchannel and calls the callback function.

11.5.3.12 pch_css_set_io_irq()

```
void pch_css_set_io_irq (
            irq_num_t irqnum)
```

Low-level function to set the IRQ number that the CSS uses for I/O interrupt notification.

Sets the IRQ number that the CSS raises when a subchannel becomes status pending.

Typically, should be a non-externally-used user IRQ (i.e. IRQ numbers 26-31 on RP2040 and IRQ numbers 46-51 (SPAREIRQ_IRQ0 through SPAREIRQ_IRQ5) on RP2350. In general, either the high-level convenience function pch_css_auto_configure_io_irq() should be used instead or, for mid-level control of the handler, variants on pch_↩ css_configure_io_irq...

Typically, should be a non-externally used IRQ (i.e. IRQ numbers 26-31 on RP2040 and IRQ numbers 46-51 (SPAREIRQ_IRQ0 through SPAREIRQ_IRQ5) on RP2350. Although the application can use its own ISR if it wishes, adding function pch_css_io_irq_handler as an ISR for this interrupt lets the CSS itself handle callbacks for subchannels with pending status (see pch_css_set_io_callback).

Low-level function to set the IRQ number that the CSS uses for I/O interrupt notification.

Typically, should be a non-externally used IRQ (i.e. IRQ numbers 26-31 on RP2040 and IRQ numbers 46-51 (SPAREIRQ_IRQ0 through SPAREIRQ_IRQ5) on RP2350. Although the application can use its own ISR if it wishes, adding function pch_css_io_irq_handler as an ISR for this interrupt lets the CSS itself handle callbacks for subchannels with pending status (see pch_css_set_io_callback).

11.5.3.13 pch_css_set_trace()

```
bool pch_css_set_trace (
            bool trace)
```

Sets whether CSS tracing is enabled.

If this flag is not set to be true then no CSS trace records are written, regardless of any per-channel or per-subchannel trace flags.

11.5.3.14 pch_css_start()

```
void pch_css_start (
            io_callback_t io_callback,
            uint8_t isc_mask)
```

Starts CSS operation after setting the io_callback (if non-NULL), configuring and enabling any needed CSS IRQ handlers that have not yet been set and setting the mask of ISCs that trigger I/O interrupts to be isc_mask.

pch_css_init() must be called before calling this function. If the CSS DMA IRQ index is not yet set, it is configured using the index number corresponding to the current core number. If the function IRQ is not set, it is configured by claiming an unused user IRQ, setting the handler to pch_css_func_irq_handler and enabling the IRQ. If io_callback is non-NULL then it is set as the CSS io_callback function after, if the I/O IRQ is not set, configuring it by claiming an unused IRQ, settting the handler to pch_css_io_irq_handler and enabling the IRQ. Any IRQ handlers set from this function are added using irq_add_shared_handler() with an order_priority of PICO_SHARED_IRQ_HANDLER_↩
DEFAULT_ORDER_PRIORITY.

11.5.3.15 pch_cus_auto_configure_uartcu()

```
void pch_cus_auto_configure_uartcu (
            pch_cuaddr_t cua,
            uart_inst_t * uart,
            uint baudrate)
```

Initialise and configure a UART control unit with default dma_channel_config control register.

Calls pch_uart_init() with baud rate

Parameters

baudrate	and pch_cus_uartcu_configure with ctrl argument bits taken from an appropriate dma_channel_get_default_config() value. The CSS on the other side of the channel MUST use the same baud rate and uart settings set pch_uart_init().

11.5.3.16 pch_sch_cancel()

```
int pch_sch_cancel (
            pch_sid_t sid)
```

Cancel a channel program that has not yet started.

pch_sch_cancel tries to cancel a channel program before it has started. If pch_sch_cancel is called before the CSS has actually started the channel program (meaning that pch_sch_start() has set the AcStartPending in the subchannel's SCSW control flags but the function IRQ handler that would then process the Start has not yet run), then it cancels the start and returns condition code 0. Otherwise, it returns 1 meaning "too late to cancel" or 2 for "no such sid".

pch_sch_cancel only acts on the schib; it does not trigger any interrupt to cause any function IRQ not does it communicate with the CU in any way.

11.5.3.17 pch_sch_halt()

```
int pch_sch_halt (
            pch_sid_t sid)
```

Halt a channel program.

pch_sch_halt tries to halt a channel program. It sets the subchannel's AcHaltPending flag and triggers a CSS function IRQ which sends a Halt command to the CU for the device. The CU and device driver are responsible for acting on the Halt command in a timely manner and responding with an UpdateStatus to end the channel program as soon as reasonably convenient. Depending on the device driver, the Halt may or may not return a normal status.

11.5.3.18 pch_sch_modify()

```
int pch_sch_modify (
            pch_sid_t sid,
            pch_pmcw_t * pmcw)
```

Modifies the PMCW field of a subchannel.

Only the following parts of the PMCW of the subchannel are modified by this function; all other parts are ignored:

- intparm
- flags bits in mask PCH_PMCW_SCH_MODIFY_MASK

The bits in PCH_PMCW_SCH_MODIFY_MASK are PCH_PMCW_ENABLED, PCH_PMCW_TRACED and the ISC bits, PCH_PMCW_ISC_BITS.

11.5.3.19 pch_sch_modify_enabled()

```
int pch_sch_modify_enabled (
            pch_sid_t sid,
            bool enabled)
```

Modifies enabled flag of the schib for subchannel sid.

This is a convenience/optimised subset of pch_sch_modify that only modifies the enabled flag of the subchannel.

11.5.3.20 pch_sch_modify_flags()

```
int pch_sch_modify_flags (
            pch_sid_t sid,
            uint16_t flags)
```

Modifies the flags field of the PMCW part of the schib for subchannel sid.

This is a convenience/optimised subset of pch_sch_modify that only modifies the PMCW flags of the subchannel.

11.5.3.21 pch_sch_modify_intparm()

```
int pch_sch_modify_intparm (
            pch_sid_t sid,
            uint32_t intparm)
```

Modifies the intparm field of the PMCW part of the schib for subchannel sid.

This is a convenience/optimised subset of pch_sch_modify that only modifies the Interruption Parameter of the subchannel.

11.5.3.22 pch_sch_modify_isc()

```
int pch_sch_modify_isc (
            pch_sid_t sid,
            uint8_t isc)
```

Modifies the isc field of the PMCW part of the schib for subchannel sid.

This is a convenience/optimised subset of pch_sch_modify that only modifies the ISC of the subchannel.

11.5.3.23 pch_sch_modify_traced()

```
int pch_sch_modify_traced (
            pch_sid_t sid,
            bool traced)
```

Modifies traced flag of the schib for subchannel sid.

This is a convenience/optimised subset of pch_sch_modify that only modifies the traced flag of the subchannel.

11.5.3.24 pch_sch_resume()

```
int pch_sch_resume (
            pch_sid_t sid)
```

Resume a channel program for a subchannel.

Resumes a channel program that has been started for subchannel sid but has become suspended by reaching a CCW with the Suspend flag (PCH_CCW_FLAG_S) set.

The function updates an internal linked list and state then raises an IRQ for the CSS to resume the channel program asynchronously. For a Release-build, the function will typically take dozens rather than hundreds of CPU cycles.

11.5.3.25 pch_sch_run_wait()

```
int pch_sch_run_wait (
            pch_sid_t sid,
            pch_ccw_t * ccw_addr,
            pch_scsw_t * scsw)
```

Start a channel program for a subchannel and wait for an I/O interruption condition.

This is a convenience function which calls pch_sch_start to start a channel program for subchannel sid and then calls pch_sch_wait to wait for it to become status pending.

11.5.3.26 pch_sch_run_wait_timeout()

```
int pch_sch_run_wait_timeout (
            pch_sid_t sid,
            pch_ccw_t * ccw_addr,
            pch_scsw_t * scsw,
            absolute_time_t timeout_timestamp)
```

Start a channel program for a subchannel and wait for an I/O interruption condition with a timeout.

This is a convenience function which calls pch_sch_start to start a channel program for subchannel sid and then calls pch_sch_wait_timeout to wait for it to become status pending or for a timeout to expire.

11.5.3.27 pch_sch_start()

```
int pch_sch_start (
            pch_sid_t sid,
            pch_ccw_t * ccw_addr)
```

Start a channel program for a subchannel.

Starts a channel program running for subchannel sid starting with the CCW at address ccw_addr.

The function updates an internal linked list and state then raises an IRQ for the CSS to start the channel program asynchronously. For a Release-build, the function will typically take dozens rather than hundreds of CPU cycles.

11.5.3.28 pch_sch_store()

```
int pch_sch_store (
            pch_sid_t sid,
            pch_schib_t * out_schib)
```

Stores the contents of the schib for subchannel sid to out_schib.

Although the schib may be in memory that is addressable by the picochan CSS, it is architecturally independent and no part of the CSS API relies on that. pch_sch_store is the architectural API that provides access to the contents of the schib by copying it from its internal location to the application-visible memory pointed to by out_schib. The PMCW and SCSW parts of the schib are architectural and can be relied on to be as documented. The rest of the schib - the Model Dependent Area (MDA) - is intended to be an internal implementation detail.

11.5.3.29 pch_sch_store_pmcw()

```
int pch_sch_store_pmcw (
            pch_sid_t sid,
            pch_pmcw_t * out_pmcw)
```

Stores the contents of the PMCW part of the schib for subchannel sid to out_pmcw.

This is a convenience/optimised subset of pch_sch_store that only stores the PMCW part of the schib.

11.5.3.30 pch_sch_store_scsw()

```
int pch_sch_store_scsw (
            pch_sid_t sid,
            pch_scsw_t * out_scsw)
```

Stores the contents of the SCSW part of the schib for subchannel sid to out_scsw.

This is a convenience/optimised subset of pch_sch_store that only stores the SCSW part of the schib.

11.5.3.31 pch_sch_test()

```
int pch_sch_test (
            pch_sid_t sid,
            pch_scsw_t * scsw)
```

Test the status of a subchannel, clearing various status conditions of status is pending.

Retrieves a SCSW representing the current status of a subchannel. If the subchannel is "status pending", removes it from the list of subchannels that are the cause of an I/O interruption condition (or callback) and clears pending function conditions and, if set, the "Suspended" condition.

11.5.3.32 pch_sch_wait()

```
int pch_sch_wait (
            pch_sid_t sid,
            pch_scsw_t * scsw)
```

Wait for an I/O interruption condition for subchannel sid.

This is a convenience function which loops calling pch_sch_test on the subchannel, returning with the fetched SCSW when the subchannel becomes status pending. In between each call to pch_sch_test, the function calls __wfe() since the subchannel can only become status pending after the CSS processes an interrupt. This function must only be called while the ISC for the subchannel is masked or else there is a race condition with any I/O ISR such as pch_css_io_irq_handler which would process the I/O interruption itself.

Returns

Condition code - returned from pch_sch_test (will not be 1 since the function loops in this case)

11.5.3.33 pch_sch_wait_timeout()

```
int pch_sch_wait_timeout (
            pch_sid_t sid,
            pch_scsw_t * scsw,
            absolute_time_t timeout_timestamp)
```

Wait for an I/O interruption condition for subchannel sid with a timeout.

This is a convenience function which behaves the same as pch_sch_wait except that it also returns if the timeout expires (i.e. absolute time timeout_timestamp is reached) without the subchannel having become status pending.

11.5.3.34 pch_test_pending_interruption()

```
pch_intcode_t pch_test_pending_interruption (
            void )
```

Test if there is a pending I/O interruption.

If there is at least one subchannel which is "status pending" with an interruption condition then pch_test_pending↩
_interruption returns an pch_intcode_t containing the sid of the subchannel, its ISC, a condition code field of 1 and removes the subchannel from the list of those with a pending I/O interruption condition. If there is no such subchannel the condition code field of the returned pch_intcode_t is 0.

This function should only be called if the ISCs of any subchannels that may become pending are masked or else there is a race condition with any I/O ISR such as pch_css_io_irq_handler which would process the I/O interruption itself.

11.5.3.35 void()

```
void (
            pch_sid_t sid,
            uint count,
            bool enabled)
```

Calls pch_sch_modify_enabled() on count subchannels starting from sid, panicking if any call fails.

Calls pch_sch_modify_traced() on count subchannels starting from sid, panicking if any call fails.

Calls pch_sch_modify_enabled() on count subchannels starting from sid, panicking if any call fails.

11.6 picochan_cu

Control Unit (CU)

Files

- file dev_status.h

  Device status bit values.
- file dev_api.h

  The main API for a device on a CU.
- file devib.h

  The structures and API for a device on a CU.

Data Structures

- struct pch_cu

  pch_cu_t is a Control Unit (CU)
- struct pch_devib

  pch_devib_t represents a device on a CU

Macros

- #define PCH_NUM_CUS

  The number of control units.
- #define PCH_CU_INIT(num_devices)

  a compile-time initialiser for a pch_cu_t
- #define MAX_DEVIB_CALLBACKS 254

  The maximum number of registered callbacks.
- #define NUM_DEVIB_CALLBACKS 16

  The size of the global callbacks array.

Typedefs

- typedef struct pch_cu pch_cu_t

  pch_cu_t is a Control Unit (CU)
- typedef uint8_t **pch_cbindex_t**

  An 8-bit index into an array of callbacks that the CU can make to a device
  pch_cbindex_t is an 8-bit index into pch_devib_callbacks, an array of up to NUM_DEVIB_CALLBACKS registered callbacks on devibs.
- typedef struct pch_devib pch_devib_t

  pch_devib_t represents a device on a CU
- typedef void(∗ **pch_devib_callback_t**) (pch_devib_t ∗devib)

  pch_devib_callback_t is a function for the CU to callback a device

Functions

- static pch_devib_t ∗ pch_get_devib (pch_cu_t ∗cu, pch_unit_addr_t ua)

  Look up the pch_devib_t of a device from its CU and unit address.
- static pch_cu_t ∗ pch_get_cu (pch_cuaddr_t cua)

  Get the CU for a given control unit address.
- void pch_cus_init (void)

  Initialise CU subsystem.
- bool pch_cus_set_trace (bool trace)

  Sets whether CU subsystem tracing is enabled.
- void pch_cu_init (pch_cu_t ∗cu, uint16_t num_devibs)

  Initialises a CU with space for num_devibs devices.
- void pch_cu_register (pch_cu_t ∗cu, pch_cuaddr_t cua)

  Registers a CU at a control unit address.
- void pch_cus_uartcu_configure (pch_cuaddr_t cua, uart_inst_t ∗uart, dma_channel_config ctrl)

  Configure a UART control unit.
- void pch_cus_memcu_configure (pch_cuaddr_t cua, pch_dmaid_t txdmaid, pch_dmaid_t rxdmaid, dmachan_tx_channel_t ∗txpeer)

  Configure a memchan control unit.
- void pch_cu_start (pch_cuaddr_t cua)

  Starts the channel from CU cua to the CSS.
- bool pch_cus_trace_cu (pch_cuaddr_t cua, bool trace)

  Sets whether tracing is enabled for CU cua.
- bool pch_cus_trace_dev (pch_devib_t ∗devib, bool trace)

  Sets whether tracing is enabled for device.
- dmachan_tx_channel_t ∗ pch_cu_get_tx_channel (pch_cuaddr_t cua)

Fetch the internal tx side of a channel from CU to CSS.
- int pch_dev_set_callback (pch_devib_t ∗devib, int cbindex_opt)

  Set callback for device.
- int pch_dev_send_then (pch_devib_t ∗devib, void ∗srcaddr, uint16_t n, proto_chop_flags_t flags, int cbindex_opt)

  Sends data to the CSS.
- int pch_dev_send_zeroes_then (pch_devib_t ∗devib, uint16_t n, proto_chop_flags_t flags, int cbindex_opt)

  Sends zeroes to the CSS.
- int pch_dev_receive_then (pch_devib_t ∗devib, void ∗dstaddr, uint16_t size, int cbindex_opt)

  Receive data from the CSS.
- void pch_register_devib_callback (pch_cbindex_t n, pch_devib_callback_t cb)

  Registers a device callback function at a specific index.
- pch_cbindex_t pch_register_unused_devib_callback (pch_devib_callback_t cb)

  Registers a device callback function at an unused index.
- static void pch_devib_prepare_callback (pch_devib_t ∗devib, pch_cbindex_t cbindex)

  Low-level API to update devib->cbindex.
- static void pch_devib_prepare_count (pch_devib_t ∗devib, uint16_t count)

  Low-level API to update devib->payload with a count field.
- static void pch_devib_prepare_write_data (pch_devib_t ∗devib, void ∗srcaddr, uint16_t n, proto_chop_flags↵_t flags)

  Low-level API to prepare a Data channel operation command for a device.
- static void pch_devib_prepare_write_zeroes (pch_devib_t ∗devib, uint16_t n, proto_chop_flags_t flags)

  Low-level API to prepare a Data channel operation command for a device that will implicitly send zeroes.
- static void pch_devib_prepare_read_data (pch_devib_t ∗devib, void ∗dstaddr, uint16_t size)

  Low-level API to prepare a RequestRead channel operation command for a device.
- void pch_devib_prepare_update_status (pch_devib_t ∗devib, uint8_t devs, void ∗dstaddr, uint16_t size)

  Low-level API to prepare an UpdateStatus channel operation command for a device.

11.6.1 Detailed Description

Control Unit (CU)

11.6.2 Macro Definition Documentation

11.6.2.1 MAX_DEVIB_CALLBACKS

```
#define MAX_DEVIB_CALLBACKS 254
```

The maximum number of registered callbacks.

A callback index greater than this is handled internally.

11.6.2.2 NUM_DEVIB_CALLBACKS

```
#define NUM_DEVIB_CALLBACKS 16
```

The size of the global callbacks array.

Must be a compile-time definition, must not exceed MAX_DEVIB_CALLBACKS (254) and must provide room for any internal specially-defined callbacks. Default 16.

11.6.2.3 PCH_CU_INIT

```
#define PCH_CU_INIT(
              num_devices)
```

a compile-time initialiser for a pch_cu_t

PCH_CU_INIT relies on a non-standard C extension (supported by gcc) to initialise a pch_cu_t that includes the space for its devibs array (a Flexible Array Member) at the end of ths struct. The num_devices macro argument is evaluated more than once but since it must be a compile-time constant this should not be a problem.

11.6.2.4 PCH_NUM_CUS

```
#define PCH_NUM_CUS
```

The number of control units.

Must be a compile-time constant between 1 and 256. Default 4. Defines the size of the global array of pch_cu_t structures running on this Pico.

11.6.3 Typedef Documentation

11.6.3.1 pch_cu_t

```
typedef struct pch_cu pch_cu_t
```

pch_cu_t is a Control Unit (CU)

The struct starts with a fixed-size metadata section with state and communication information about its devices and channel to the CSS. Immediately following that (ignoring internal padding) is an array of pch_devib_t structures, one for each device on the CU. The size of that array is held in the num_devibs field of the pch_cu_t which is set at the time pch_cu_init is called and cannot be changed afterwards. The allocation of memory for a pch_cu_t, whether static or dynamic, is the responsibility of the application before calling pch_cu_init.

The alignment of pch_cu_t is enforced to be PCH_CU_ALIGN which is calculated at compile-time as PCH_MAX↩
_DEVIBS_PER_CU multiplied by the smallest power of 2 greater than or equal to sizeof(pch_devib_t). This allows address arithmetic and bit masking to determine the unit address and owning pch_cu_t of a devib. PCH_MAX↩
_DEVIBS_PER_CU, a preprocessor symbol, can be defined as any compile-time constant between 1 and 256, defaulting to 32. sizeof(pch_devib_t) is currently 16 so for the default PCH_MAX_DEVIBS_PER_CU, alignof(pch↩
_cu_t) is 512. With the maximum PCH_MAX_DEVIBS_PER_CU of 256, alignof(pch_cu_t) is 4096. Each individual pch_cu_t may be allocated at either compile-time or runtime with a smaller numbers of devibs than PCH_MAX_↩
DEVIBS_PER_CU but the alignment as calculated above is still required.

11.6.3.2 pch_devib_t

```
typedef struct pch_devib pch_devib_t
```

pch_devib_t represents a device on a CU

```
DEVIB  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       |     next      |    cbindex     |          size               |
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       |       op      |     flags      |         payload             |
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       |                          bufaddr                             |
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       |                            sense                             |
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

11.6.4 Function Documentation

11.6.4.1 pch_cu_get_tx_channel()

```
dmachan_tx_channel_t * pch_cu_get_tx_channel (
            pch_cuaddr_t cua)
```

Fetch the internal tx side of a channel from CU to CSS.

This function is only needed when configuring a memchan between a CU and the CSS on different cores of a single Pico. The CSS initialisation procedure uses this function to find its peer CU structure in order to cross-connect the channels.

11.6.4.2 pch_cu_init()

```
void pch_cu_init (
            pch_cu_t * cu,
            uint16_t num_devibs)
```

Initialises a CU with space for num_devibs devices.

Parameters

cu	Must be a pointer to enough space to hold the pch_cu_t structure including its flexible array that must itself have room for num_devibs pch_devib_t structures.
num_devibs	The number of devices to initialise

Typically, the PCH_CU_INIT macro is used as a static initialiser instead of needing to call this function on an uninitialised pch_cu_t.

11.6.4.3 pch_cu_register()

```
void pch_cu_register (
            pch_cu_t * cu,
            pch_cuaddr_t cua)
```

Registers a CU at a control unit address.

Parameters

cu	the CU to register
cua	control unit address to register as

No CU must yet have been registered as control unit address cua. cu must already have been initialised either with static initialiser PCH_CU_INIT() or by calling pch_cu_init().

11.6.4.4 pch_cu_start()

```
void pch_cu_start (
            pch_cuaddr_t cua)
```

Starts the channel from CU cua to the CSS.

The CU must already have been registered by calling pch_cu_register(). If the CU has already been started, this function returns without doing anything. If no DMA IRQ index has yet been explicitly configured for this CU then pch_cus_auto_configure_dma_irq_index(true) is called and pch_cu_set_dma_irq_index() is called to set the CU to use the returned index. Then it marks the CU as started and starts the channel to the CSS, allowing it to receive commands from the CSS.

11.6.4.5 pch_cus_init()

```
void pch_cus_init (
            void )
```

Initialise CU subsystem.

Must be called before any other CU function.

11.6.4.6 pch_cus_memcu_configure()

```
void pch_cus_memcu_configure (
            pch_cuaddr_t cua,
            pch_dmaid_t txdmaid,
            pch_dmaid_t rxdmaid,
            dmachan_tx_channel_t * txpeer)
```

Configure a memchan control unit.

A memchan control unit allows the CU to run on one core of a Pico while the CSS runs on the other core. Instead of using physical pins or connections between CU and CSS, picochan uses the DMA channels to copy memory-to-memory between CU and CSS and an internal state machine and cross-core synchronisation to mediate CU to CSS communications. txdmaid and rxdmaid must be two unused DMA ids, typically allocated using dma_claim_←
unused_channel(). In order for the CU to find the CSS-side information to cross-connect the sides in memory, the CSS API function pch_chp_get_tx_channel() must be used to fetch the internal dmachan_tx_channel_t of the peer CSS channel for passing to pch_cus_memcu_configure.

11.6.4.7 pch_cus_set_trace()

```
bool pch_cus_set_trace (
            bool trace)
```

Sets whether CU subsystem tracing is enabled.

If this flag is not set to be true then no CU trace records are written, regardless of any per-CU or per-device trace flags.

11.6.4.8 pch_cus_trace_cu()

```
bool pch_cus_trace_cu (
            pch_cuaddr_t cua,
            bool trace)
```

Sets whether tracing is enabled for CU cua.

If this flag is not set to be true then no CU trace records are written for this CU and no device trace records, regardless of any per-device trace flags.

11.6.4.9 pch_cus_trace_dev()

```
bool pch_cus_trace_dev (
            pch_devib_t * devib,
            bool trace)
```

Sets whether tracing is enabled for device.

If this flag is set to true and the trace flag is set for the CU subsystem as a whole (with pch_cus_set_trace) and the trace flag is set for the device's CU (with pch_cus_trace_cu) then device trace records are written for this device. If this function changes the setting of the device's trace flag then a trace record is written to indicate this (unlike using the low-level pch_devib_set_traced() function).

11.6.4.10 pch_cus_uartcu_configure()

```
void pch_cus_uartcu_configure (
            pch_cuaddr_t cua,
            uart_inst_t * uart,
            dma_channel_config ctrl)
```

Configure a UART control unit.

Configure the hardware UART instance uart as a channel from CU cua to the CSS. The UART must have been initialised already, be connected to the CSS using the same baud rate as the CSS has configured and the hardware flow control pins, CTS and RTS MUST be enabled and connected between CU and CSS. ctrl should typically be a default dma_channel_config as returned from dma_channel_get_default_config(dmaid) invoked on any DMA id. Most bits in that dma_channel_config are overridden by the CU (including the CHAIN_TO which is why the dmaid above does not matter) but some applications may wish to set bits SNIFF_EN and HIGH_PRIORITY for their own purposes.

If you want to initialise and configure the UART channel using a given baud rate, suggested UART settings (8E1) and default DMA control register settings (no SNIFF_EN and no HIGH_PRIORITY), you can use pch_cus_auto_configure_uartcu() instead.

11.6.4.11 pch_dev_receive_then()

```
int pch_dev_receive_then (
            pch_devib_t * devib,
            void * dstaddr,
            uint16_t size,
            int cbindex_opt)
```

Receive data from the CSS.

This, and related variants, is the primary function used to receive data from the CSS from the source address and count specified in a CCW segment with a Write-type command. Before calling this function, the device must have verified that the CSS is expecting to send data, i.e.

- the Start callback must have been called for the device and the device has not since sent an UpdateStatus including ChannelEnd
- and the CCW command must have been Write-Type (the devib->flags field must have the PCH_DEVIB_↩ FLAG_CMD_WRITE bit set).

If the device requests more data than the CCW segment contains then the amount of data sent to the device will be safely capped at the available amount but additional effects depend on flags set in the CCW and, possibly, the subchannel. A request by the device for more data than is available is an "Incorrect Length Condition" and, unless the channel program has included the PCH_CCW_FLAG_SLI ("Suppress Length Indication") flag in the CCW, will cause the channel program to stop any data chaining or command chaining and end (eventually) with a subchannel status field including the PCH_SCHS_INCORRECT_LENGTH flag. It is up to the device driver author to be aware of the effects the request counts may have on the channel program and, ideally, use them and document them in a way that allows the channel program author to construct channel programs that can make good use of the additional length checks or have them ignored where appropriate.

The `devib->size` field will have been filled in at Start time with a size that is no more than (and will typically be very close to) the size specified by the CCW segment itself. Following a call to `pch_dev_receive_then()` or its variants, the response from the CSS includes an exact up-to-date count of the remaining available room in the CCW segment and the CU updates the `devib->size` field with this value before invoking the next callback on the device.

Parameters

cu	- the control unit
ua	- the unit address of the device in control unit `cu`
dstaddr	- the address to receive the data sent by the CSS
size	- the number of data bytes requested - the number of bytes actually received will be at most `n` but may be strictly less.
cbindex_opt	- before sending, update the callback index in the devib (unless -1 is passed) ready for the next callback to the device, which will happen after the data has been received and the CU has updated the `devib->size` field with the remaining count of available data bytes.

11.6.4.12 pch_dev_send_then()

```
int pch_dev_send_then (
            pch_devib_t * devib,
            void * srcaddr,
            uint16_t n,
            proto_chop_flags_t flags,
            int cbindex_opt)
```

Sends data to the CSS.

This, and related variants, is the primary function used to send data to the CSS satisfying some or all of a CCW segment with a Read-type command. Before calling this function, the device must have verified that (1) the CSS is expecting data to be sent and (2) the amount of data it sends is no more than the maximum space advertised by the CSS. For (1),

- the Start callback must have been called for the device and the device has not since sent an UpdateStatus including ChannelEnd
- and the CCW command must have been Read-Type (the devib->flags field must have the PCH_DEVIB_↩ FLAG_CMD_WRITE bit as zero).

For (2), provided (1) holds, the devib->size field will have been filled in at Start time with a size that is no more than (and will typically be very close to) the size specified by the CCW segment itself. However, the size field is not affected by using this or related functions to send data to the CSS (and the field should not be updated in such a way by the device). Use the PROTO_CHOP_FLAG_RESPONSE_REQUIRED flag (see below) if up-to-date and/or exact size information is needed.

Parameters

cu	- the control unit
ua	- the unit address of the device in control unit `cu`
flags	- may contain the following flags: • PROTO_CHOP_FLAG_RESPONSE_REQUIRED -request that the CSS send an update (a Room operation) that causes the CU to update the `devib->size` field with up-to-date and exact information. • PROTO_CHOP_FLAG_END - after sending the data, the CSS will behave as though the device has sent a final device status with no unusual conditions (DeviceEnd|ChannelEnd and no other bits set). • PROTO_CHOP_FLAG_SKIP - instead of sending n data bytes down the channel, the CSS will behave as though n bytes of zeroes were sent. If this flag is set, srcaddr is ignored.
srcaddr	- the address of the data to be sent (ignored if flags contains PROTO_CHOP_FLAG_SKIP)
n	- the number of data bytes to send
cbindex_opt	- before sending, update the callback index in the devib (unless -1 is passed) ready for the next callback to the device. The event that will cause the next callback depends on the flags: • PROTO_CHOP_FLAG_RESPONSE_REQUIRED - the callback will happen after the CSS has replied with its Room operation and the CU has updated the `devib->size` field with an up-to-date and exact size. • PROTO_CHOP_FLAG_END - the next callback will be when the next CCW is processed causing a Start to the device (whether a CCW command-chained from the previous channel program or a new channel program - the difference is not visible to the device). • any other combination - the callback will happen as soon as the CU has completed sending the command+data to the CSS meaning that the device can invoke further API calls if it wishes. Whether any new API calls will cause commands to be sent to the CSS immediately depends on whether any other devices have commands that are being sent or are pending ahead of new requests from this device.

11.6.4.13 pch_dev_send_zeroes_then()

```
int pch_dev_send_zeroes_then (
            pch_devib_t * devib,
            uint16_t n,
            proto_chop_flags_t flags,
            int cbindex_opt)
```

Sends zeroes to the CSS.

Convenience function that calls pch_dev_send_then with a flags field that ORs in PROTO_CHOP_FLAG_SKIP and an (ignored) srcaddr of 0.

11.6.4.14 pch_dev_set_callback()

```
int pch_dev_set_callback (
            pch_devib_t * devib,
            int cbindex_opt)
```

Set callback for device.

Sets, changes or unsets the callback function that the CU invokes when action is needed from the device.

Parameters

cu	the CU to which the device belongs
ua	the unit address of the device within its CU
cbindex_opt	either a callback index (pch_devib_callback_t) of a callback function registered with pch_register_devib_callback or one of the following special values: • PCH_DEVIB_CALLBACK_DEFAULT - any attempt by the CSS to start a channel program for this device will result in the CU responding on its behalf with a final device status (ChannelEnd|DeviceEnd) with UnitCheck set and a sense code set with CommandReject with additional code EINVALIDDEV. Any attempt to callback the device at any other point in its lifecycle will result in the CU responding on its behalf with a final device status (ChannelEnd|DeviceEnd) with UnitCheck set and a sense code set with ProtoError, an additional code of the requested operation and ASC and ASCQ containing the bytes p0 and p1, respectively, of the operation packet payload. • PCH_DEVIB_CALLBACK_NOOP - any attempt to callback this device will be silently ignored. For this to be at all useful, the device must be specially written to determine any actions needed of it independently of the usual CU-to-device communication mechanisms. • -1 - the device callback is not changed

11.6.4.15 pch_devib_prepare_callback()

```
void pch_devib_prepare_callback (
            pch_devib_t * devib,
            pch_cbindex_t cbindex)  [inline], [static]
```

Low-level API to update devib->cbindex.

The cbindex field determines the callback that the CU will invoke the next time an event happens that needs handling by the device. For a Debug build, asserts if cbindex is invalid (out of range or unregistered).

Typically, device driver authors should use the higher-level pch_dev_ API rather than this low-level API.

11.6.4.16 pch_devib_prepare_count()

```
void pch_devib_prepare_count (
            pch_devib_t * devib,
            uint16_t count) [inline], [static]
```

Low-level API to update devib->payload with a count field.

The payload of a RequestRead or Data channel operation command provides the count of data bytes that are requested from the channel or are to be sent to the channel.

Typically, device driver authors should use the higher-level pch_dev_ API rather than this low-level API.

11.6.4.17 pch_devib_prepare_read_data()

```
void pch_devib_prepare_read_data (
            pch_devib_t * devib,
            void * dstaddr,
            uint16_t size) [inline], [static]
```

Low-level API to prepare a RequestRead channel operation command for a device.

Uses pch_devib_prepare_count to set the count of bytes that are to be requested, sets the destination address for the bytes and sets the channel operation command to be PROTO_CHOP_REQUEST_READ.

For a Debug build, asserts if the device has not received a Start operation.

Typically, device driver authors should use the higher-level pch_dev_ API rather than this low-level API.

11.6.4.18 pch_devib_prepare_update_status()

```
void pch_devib_prepare_update_status (
            pch_devib_t * devib,
            uint8_t devs,
            void * dstaddr,
            uint16_t size)
```

Low-level API to prepare an UpdateStatus channel operation command for a device.

Sets the channel operation command to be PROTO_CHOP_UDPATE_STATUS. Sets the device status (devs) in the payload. If it's either an unsolicited status (neither ChannelEnd nor DeviceEnd set) or it's end-of-channel-program (both ChannelEnd and DeviceEnd set) then it also sets the devib addr field to dstaddr, the size field to field and encodes the (16-bit) size as an 8-bit "bsize" value within the payload. A non-zero value of the size advertises to the CSS the buffer and length to which the next CCW Write-type command can immediately send data during Start.

For a Debug build, asserts if the device has not received a Start operation.

Typically, device driver authors should use the higher-level pch_dev_ API rather than this low-level API.

11.6.4.19 pch_devib_prepare_write_data()

```
void pch_devib_prepare_write_data (
            pch_devib_t * devib,
            void * srcaddr,
            uint16_t n,
            proto_chop_flags_t flags)  [inline], [static]
```

Low-level API to prepare a Data channel operation command for a device.

Uses pch_devib_prepare_count to set the count of bytes to be written, sets the source address for the bytes and sets the channel operation command to be PROTO_CHOP_DATA along with any provided flags.

For a Debug build, asserts if the device has not received a Start operation.

Typically, device driver authors should use the higher-level pch_dev_ API rather than this low-level API.

11.6.4.20 pch_devib_prepare_write_zeroes()

```
void pch_devib_prepare_write_zeroes (
            pch_devib_t * devib,
            uint16_t n,
            proto_chop_flags_t flags)  [inline], [static]
```

Low-level API to prepare a Data channel operation command for a device that will implicitly send zeroes.

Uses pch_devib_prepare_count to set the count of zero bytes to be written and sets the channel operation command to be PROTO_CHOP_DATA together with the PROTO_CHOP_FLAG_SKIP flag that means that the CU does not have to send any actual data bytes down the channel and causes the CSS to write zero bytes itself directly to the CCW's destination address.

For a Debug build, asserts if the device has not received a Start operation.

Typically, device driver authors should use the higher-level pch_dev_ API rather than this low-level API.

11.6.4.21 pch_get_cu()

```
pch_cu_t * pch_get_cu (
            pch_cuaddr_t cua)  [inline], [static]
```

Get the CU for a given control unit address.

For a Debug build, asserts when cua exceeds the (compile-time defined) number of CUs, PCH_NUM_CUS, or if the CU has not been initialised with pch_cu_init.

11.6.4.22 pch_get_devib()

```
pch_devib_t * pch_get_devib (
            pch_cu_t * cu,
            pch_unit_addr_t ua)  [inline], [static]
```

Look up the pch_devib_t of a device from its CU and unit address.

This is a direct array member dereference into the devibs array in the CU. There is no checking that ua is in range.

11.6.4.23 pch_register_devib_callback()

```
void pch_register_devib_callback (
            pch_cbindex_t n,
            pch_devib_callback_t cb)
```

Registers a device callback function at a specific index.

For a Debug build, asserts if n is out of range in the global array of callbacks or if the callback index is already registered.

11.6.4.24 pch_register_unused_devib_callback()

```
pch_cbindex_t pch_register_unused_devib_callback (
            pch_devib_callback_t cb)
```

Registers a device callback function at an unused index.

Panics if no more unused indices are available in the global array of callbacks. This performs a simple linear iteration of the array to find the first unused slot so is not intended to be used at performance sensitive times.

Returns

The allocated callback index number

Chapter 12

Data Structure Documentation

12.1 addr_count Struct Reference

Data Fields

- uint32_t **addr**
- uint16_t **count**
- bool **discard**

The documentation for this struct was generated from the following file:

- css/rx_handle.c

12.2 css Struct Reference

struct css is a channel subsystem (CSS)

```
#include <css_internal.h>
```

Data Fields

- schib_dlist_t **isc_dlists** [8]
- io_callback_t **io_callback**
- int16_t **io_irqnum**

  -1 or Irq raised for schib notify
- int16_t **func_irqnum**

  raised by API to schedule schib function
- uint8_t **isc_enable_mask**
- uint8_t **isc_status_mask**
- pch_dma_irq_index_t **dmairqix**

  completions raise irq dma.IRQ_BASE+dmairqix
- int8_t **core_num**
- pch_sid_t **next_sid**

  starting SID for next pch_chp_claim
- pch_trc_bufferset_t **trace_bs**
- pch_chp_t **chps** [4]
- pch_schib_t **schibs** [32]

12.2.1 Detailed Description

struct css is a channel subsystem (CSS)

It is intended to be a singleton and is just a convenience for gathering together the global variables associated with the CSS.

The documentation for this struct was generated from the following file:

- css/css_internal.h

12.3 dmachan_1way_config Struct Reference

Data Fields

- uint32_t **addr**
- dma_channel_config **ctrl**
- pch_dmaid_t **dmaid**
- pch_dma_irq_index_t **dmairqix**

The documentation for this struct was generated from the following file:

- base/include/picochan/dmachan.h

12.4 dmachan_cmd Union Reference

Data Fields

- unsigned char **buf** [4]
- uint32_t **raw**

The documentation for this union was generated from the following file:

- base/include/picochan/dmachan.h

12.5 dmachan_config Struct Reference

Data Fields

- dmachan_1way_config_t **tx**
- dmachan_1way_config_t **rx**

The documentation for this struct was generated from the following file:

- base/include/picochan/dmachan.h

12.6 dmachan_link Struct Reference

Data Fields

- dmachan_cmd_t **cmd**
- pch_trc_bufferset_t ∗ **bs**
- pch_dmaid_t **dmaid**
- pch_dma_irq_index_t **dmairqix**
- bool **complete**
- bool **resetting**

The documentation for this struct was generated from the following file:

- base/include/picochan/dmachan.h

12.7 dmachan_rx_channel Struct Reference

Data Fields

- dmachan_link_t **link**
- dmachan_tx_channel_t ∗ **mem_tx_peer**
- uint32_t **srcaddr**
- dma_channel_config **ctrl**
- dmachan_mem_dst_state_t **mem_dst_state**

The documentation for this struct was generated from the following file:

- base/include/picochan/dmachan.h

12.8 dmachan_tx_channel Struct Reference

Data Fields

- dmachan_link_t **link**
- dmachan_rx_channel_t ∗ **mem_rx_peer**
- dmachan_mem_src_state_t **mem_src_state**

The documentation for this struct was generated from the following file:

- base/include/picochan/dmachan.h

12.9 dmairqix_config Struct Reference

Data Fields

- dmairqix_config_state_t **state**
- uint8_t **core_num**

The documentation for this struct was generated from the following file:

- cu/cu.c

12.10 pch_bsize Struct Reference

an 8-bit structure whose value encodes a 16-bit value for use as a count of bytes in a typical picochan buffer or transfer request

```
#include <bsize.h>
```

Data Fields

- uint8_t **esize**

12.10.1 Detailed Description

an 8-bit structure whose value encodes a 16-bit value for use as a count of bytes in a typical picochan buffer or transfer request

The 8-bit encoding is wrapped as a structure to provide type clarity (even if not full type safety is not possible) when being passed around via the API and stored.

The encoding is not 1-1 (of course) but the decoding of the value obtained by encoding n is always less than or equal to n and "close" when n is a size typically used as a buffer size for workloads using picochan.

The encoding/decoding is exact for the following values:

- 1 x [0, 63] -> 0, 1, 2, ..., 63
- 2 x [32, 95] -> 64, 66, 68, ..., 190
- 8 x [24, 87] -> 192, 200, 208, ..., 696
- 64 x [11, 74] -> 704, 768, 832, ..., 4736

The documentation for this struct was generated from the following file:

- base/include/picochan/bsize.h

12.11 pch_bsizex Struct Reference

a pch_bsize together with a flag intended to indicate whether the bsize encoded the original size exactly.

```
#include <bsize.h>
```

Data Fields

- uint8_t **exact**
- pch_bsize_t **bsize**

12.11.1 Detailed Description

a pch_bsize together with a flag intended to indicate whether the bsize encoded the original size exactly.

The flag is the low bit of the exact field. It is defined as a uint8_t rather than a bool to make its position clearer in any stored value of the structure.

The documentation for this struct was generated from the following file:

- base/include/picochan/bsize.h

12.12 pch_ccw Struct Reference

I/O Channel-Command Word (CCW)

```
#include <ccw.h>
```

Data Fields

- uint8_t **cmd**
- pch_ccw_flags_t **flags**
- uint16_t **count**
- uint32_t **addr**

12.12.1 Detailed Description

I/O Channel-Command Word (CCW)

pch_ccw_t is an architected 8-byte control block that must be 4-byte aligned. When marshalling/unmarshalling a CCW, unlike the original architected Format-1 CCW which was implicitly big-endian, the count and addr fields here are treated as native-endian and so will be little-endian on both ARM and RISC-V (in Pico configurations) and would also be so on x86, for example.

```
CCW +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |      cmd      |     flags     |             count             |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |                          data address                         |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The documentation for this struct was generated from the following file:

- base/include/picochan/ccw.h

12.13 pch_chp Struct Reference

pch_chp_t is the CSS-side representation of a channel path to a control unit.

```
#include <channel.h>
```

Data Fields

- dmachan_tx_channel_t **tx_channel**
- dmachan_rx_channel_t **rx_channel**
- pch_txsm_t **tx_pending**
- pch_sid_t **first_sid**
- uint16_t **num_devices**
- int16_t **rx_data_for_ua**
- uint8_t **rx_data_end_ds**
- bool **rx_response_required**
- bool **traced**
- bool **claimed**
- bool **allocated**
- bool **configured**
- bool **started**
- bool **tx_active**
- ua_dlist_t **ua_func_dlist**
- ua_slist_t **ua_response_slist**

12.13.1 Detailed Description

pch_chp_t is the CSS-side representation of a channel path to a control unit.

The application API usually refers to these by a channel path id (CHPID) which indexes into the global array CSS.↩ chps and so does not really need to care about the details of this struct. Currently, a channel only connects to a single control unit so the pch_chp_t is effectively a CSS-side "peer" object of the dev-side CU, pch_cu_t.

The documentation for this struct was generated from the following file:

- css/channel.h

12.14 pch_cu Struct Reference

pch_cu_t is a Control Unit (CU)

```
#include <cu.h>
```

Data Fields

- dmachan_tx_channel_t **tx_channel**
- dmachan_rx_channel_t **rx_channel**
- pch_txsm_t **tx_pending**
- pch_cuaddr_t **cuaddr**
- int16_t **tx_callback_ua**

  when tx_pending in use, the ua to callback or -1
- int16_t **rx_active**

  active ua for rx data to dev or -1 if none
- int16_t **tx_head**

  head (active) ua on tx side or -1 if none
- int16_t **tx_tail**

  tail ua on tx side pending list of -1 if none
- pch_dma_irq_index_t **dmairqix**

  completions raise irq dma.IRQ_BASE+dmairqix, -1 before configuration
- bool **traced**
- bool **configured**
- bool **started**
- uint16_t num_devibs
- pch_devib_t **devibs** [ ]

  Flexible Array Member (FAM) of size num_devibs.

12.14.1 Detailed Description

pch_cu_t is a Control Unit (CU)

The struct starts with a fixed-size metadata section with state and communication information about its devices and channel to the CSS. Immediately following that (ignoring internal padding) is an array of pch_devib_t structures, one for each device on the CU. The size of that array is held in the num_devibs field of the pch_cu_t which is set at the time pch_cu_init is called and cannot be changed afterwards. The allocation of memory for a pch_cu_t, whether static or dynamic, is the responsibility of the application before calling pch_cu_init.

The alignment of pch_cu_t is enforced to be PCH_CU_ALIGN which is calculated at compile-time as PCH_MAX↩_DEVIBS_PER_CU multiplied by the smallest power of 2 greater than or equal to sizeof(pch_devib_t). This allows address arithmetic and bit masking to determine the unit address and owning pch_cu_t of a devib. PCH_MAX↩_DEVIBS_PER_CU, a preprocessor symbol, can be defined as any compile-time constant between 1 and 256, defaulting to 32. sizeof(pch_devib_t) is currently 16 so for the default PCH_MAX_DEVIBS_PER_CU, alignof(pch↩_cu_t) is 512. With the maximum PCH_MAX_DEVIBS_PER_CU of 256, alignof(pch_cu_t) is 4096. Each individual pch_cu_t may be allocated at either compile-time or runtime with a smaller numbers of devibs than PCH_MAX_↩DEVIBS_PER_CU but the alignment as calculated above is still required.

12.14.2 Field Documentation

12.14.2.1 num_devibs

```
uint16_t pch_cu::num_devibs
```

[0, 256]

The documentation for this struct was generated from the following file:

- cu/include/picochan/cu.h

12.15 pch_dev_range Struct Reference

Data Fields

- pch_cu_t ∗ **cu**
- uint16_t **num_devices**
- pch_unit_addr_t **first_ua**

The documentation for this struct was generated from the following file:

- cu/include/picochan/cu.h

12.16 pch_dev_sense Struct Reference

The device sense structure by which a device can communicate additional error information on request by the CSS.

```
#include <dev_sense.h>
```

Data Fields

- uint8_t **flags**
- uint8_t **code**
- uint8_t **asc**
- uint8_t **ascq**

12.16.1 Detailed Description

The device sense structure by which a device can communicate additional error information on request by the CSS.

The documentation for this struct was generated from the following file:

- cu/include/picochan/dev_sense.h

12.17 pch_devib Struct Reference

pch_devib_t represents a device on a CU

```
#include <devib.h>
```

Data Fields

- pch_unit_addr_t **next**
- pch_cbindex_t **cbindex**
- uint16_t **size**
- proto_chop_t **op**
- uint8_t **flags**
- proto_payload_t **payload**
- uint32_t **addr**
- pch_dev_sense_t **sense**

12.17.1 Detailed Description

pch_devib_t represents a device on a CU

```
DEVIB  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       |     next      |    cbindex    |           size                |
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       |       op      |     flags     |          payload              |
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       |                          bufaddr                              |
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       |                            sense                              |
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The documentation for this struct was generated from the following file:

- cu/include/picochan/devib.h

12.18 pch_intcode Struct Reference

```
#include <intcode.h>
```

Data Fields

- uint32_t **intparm**
- pch_sid_t **sid**
- uint8_t **flags**
- uint8_t **cc**

12.18.1 Detailed Description

pch_intcode_t is the I/O interruption code which is returned from pch_test_pending_interruption.

The original expansion of the acronym SID is Subsystem-Identification Word which is 32 bits and includes some bits of data beyond just the subchannel number. For Picochan we only use the 16-bit subchannel number so calling this the SID is more appropriate.

```
pch_intcode_t
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |               Interruption Parameter (Intparm)                |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |  Subchannel ID (SID)          |      ISC      |           |cc |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

cc is the condition code which, for a return from pch_test_pending_interruption, only uses two values: 0 means there was no interrupt pending and the rest of the pch_intcode_t is meaningless; 1 means an interrupt was pending and its information has been returned.

The documentation for this struct was generated from the following file:

- base/include/picochan/intcode.h

12.19 pch_pmcw Struct Reference

```
#include <pmcw.h>
```

Data Fields

- uint32_t **intparm**
- uint16_t **flags**
- pch_chpid_t **chpid**
- pch_unit_addr_t **unit_addr**

12.19.1 Detailed Description

pch_pmcw_t is the Path Management Control World (PMCW)

This is an architected part of the schib. It contains

- the addressing information for the CSS to communicate with the device on its CU (see below)
- An Interruption Parameter (intparm) - a 32-bit value which is not modified by the CSS and can be used by the application for any purpose
- An Interrupt Service Class (ISC) so that groups of subchannels can be masked/unmasked together from delivering I/O interruptions
- The flag which indicates that the subchannel is enabled and can thus run channel programs
- A "trace" flag to indicate whether events for this subchannel can cause trace records to be written

Although for a mainframe channel subsystem, the addressing information in the PMCW contains 8 x 8-bit channel path id numbers referencing one or more channels that can reach the control unit, for picochan, the addressing information is simply a single channel path id (CHPID) and and the unit address of the device on the single remote CU to which it is connected.

The addressing information (CHPID and UnitAddr) must be set by the application (by using pch_chp_alloc) before the channel is started.

```
PMCW    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |               Interruption Parameter (Intparm)                |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |                     |T|E| ISC |     CHPID     | UnitAddr      |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The documentation for this struct was generated from the following file:

- css/include/picochan/pmcw.h

12.20 pch_schib Struct Reference

pch_schib_t is the Subchannel Information Block (SCHIB)

```
#include <schib.h>
```

Data Fields

- pch_pmcw_t **pmcw**
- pch_scsw_t **scsw**
- pch_schib_mda_t **mda**

12.20.1 Detailed Description

pch_schib_t is the Subchannel Information Block (SCHIB)

The SCHIB is formed from the Path Management Control Word (PMCW), Subchannel Status Word (SCSW) and Model Dependent Area (MDA). Of these, the PMCW and SCSW are architected formats and the MDA format is an internal implementation detail of the CSS.

```
PMCW    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |                            Intparm                            |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |                     |T|E| ISC |      CUAddr   | UnitAddr      |
SCSW    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |               | CC|P|I|U|Z| |N|W|  FC |     AC      |   SC    |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |                         CCW Address                           |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        | DEVS/ccwflags |     SCHS      |     Residual Count            |
MDA     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |                        data address                           |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |        reqcount/advcount      | prevua/ccwcmd |    nextua     |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |           prevsid             |           nextsid             |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

DEVS only needs to be valid when SC.StatusPending is set. Otherwise, we use the field to hold the current ccwflags.

The documentation for this struct was generated from the following file:

- css/include/picochan/schib.h

12.21 pch_schib_mda Struct Reference

The Model Dependent Area (MDA) of a schib.

```
#include <schib.h>
```

Data Fields

- uint32_t **data_addr**
- uint16_t **devcount**
- pch_unit_addr_t **prevua**
- pch_unit_addr_t **nextua**
- pch_sid_t **prevsid**
- pch_sid_t **nextsid**

12.21.1 Detailed Description

The Model Dependent Area (MDA) of a schib.

Although this structure is part of the schib, pch_schib_t, and thus is visible to applications, the contents are for internal use by the CSS.

The documentation for this struct was generated from the following file:

- css/include/picochan/schib.h

12.22 pch_scsw Struct Reference

```
#include <scsw.h>
```

Data Fields

- uint8_t **__unused_flags**
- uint8_t **user_flags**
- uint16_t **ctrl_flags**
- uint32_t **ccw_addr**
- uint8_t **devs**
- uint8_t **schs**
- uint16_t **count**

12.22.1 Detailed Description

pch_scsw_t is the Subchannel Status Word (SCSW) which must be 4-byte aligned. When marshalling/unmarshalling an SCSW, unlike the original architected SCSW which was implicitly big-endian, the ccw_addr and count fields here are treated as native-endian and so will be little-endian on both ARM and RISC-V (in Pico configurations) and would also be so on x86, for example. The flags fields are slightly rearranged from their original architected positions and some have been dropped and one or two added.

```
SCSW    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |               | CC|P|I|U|Z| |N|W|  FC |     AC      |   SC    |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |                         CCW Address                           |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |      DEVS     |     SCHS      |     Residual Count            |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The documentation for this struct was generated from the following file:

- base/include/picochan/scsw.h

12.23 pch_trc_bufferset Struct Reference

set of buffers and metadata for a subsystem to use tracing

```
#include <trc.h>
```

Data Fields

- uint32_t **current_buffer_num**

  the index in buffers of the current buffer being appended to
- uint32_t **current_buffer_pos**

  the byte offset in the current buffer where the next trace record will be written.
- int16_t irqnum

  the irq_num_t of an IRQ or -1
- bool **enable**

  the bufferset enablement flag for tracing. When false, no trace records will be written and all of the buffer arrays, pointers and indexes above are ignored.
- uint32_t **magic**

  subsystem-specific magic number for identifying dumped trace buffers
- uint32_t **buffer_size**
- uint16_t **num_buffers**
- void ∗ buffers [1]

  the array of trace buffers.

12.23.1 Detailed Description

set of buffers and metadata for a subsystem to use tracing

This struct holds an array of PCH_TRC_NUM_BUFFERS buffers, each which must be of size PCH_TRC_↩ BUFFER_SIZE.

When compile-time trace support is enabled (PCH_CONFIG_ENABLE_TRACE is defined to be non-zero), PCH↩ _TRC_NUM_BUFFERS is the number of trace buffers in a bufferset. These buffers form a ring - once the current buffer is full, the current buffer moves onto the next in the ring and, optionally, an interrupt is generated so that the previous buffer can be archived elsewhere before the ring wraps.

When compile-time trace support is not enabled, PCH_TRC_NUM_BUFFERS is defined as 0 so this struct can be instantiated but not used.

12.23.2 Field Documentation

12.23.2.1 buffers

```
void* pch_trc_bufferset::buffers[1]
```

the array of trace buffers.

It is treated as a single ring buffer of trace records. Each trace record is of the form of an 8-byte header (pch↩ _trc_header_t) followed by a number of bytes of associated trace data. The total size of header plus its following associated data is in the size field of the header.

12.23.2.2 irqnum

```
int16_t pch_trc_bufferset::irqnum
```

the irq_num_t of an IRQ or -1

When not -1, raised when pch_trc_switch_to_next_buffer is called either by explicit invocation or when writing a trace record skips to the next trace buffer because the current buffer is full.

The documentation for this struct was generated from the following file:

- base/include/picochan/trc.h

12.24 pch_trc_header Struct Reference

Data Fields

- pch_trc_timestamp_t **timestamp**
- uint8_t **size**
- pch_trc_record_type_t **rec_type**

The documentation for this struct was generated from the following file:

- base/include/picochan/trc.h

12.25 pch_trc_timestamp Struct Reference

an opaque timestamp of a 48-bit number of microseconds since boot.

```
#include <trc.h>
```

Data Fields

- uint16_t **low**
- uint16_t **mid**
- uint16_t **high**

12.25.1 Detailed Description

an opaque timestamp of a 48-bit number of microseconds since boot.

The actual value is held as three consecutive 16-bit chunks (forming a little-endian encoding of the whole value) but the intended way of accessing the value is with pch_trc_timestamp_to_us().

The documentation for this struct was generated from the following file:

- base/include/picochan/trc.h

12.26 pch_trdata_address_change Struct Reference

Data Fields

- uint32_t **old_addr**
- uint32_t **new_addr**

The documentation for this struct was generated from the following file:

- base/include/picochan/trc_records.h

12.27 pch_trdata_byte Struct Reference

Data Fields

- uint8_t **byte**

The documentation for this struct was generated from the following file:

- base/include/picochan/trc_records.h

12.28 pch_trdata_ccw_addr_sid Struct Reference

Data Fields

- pch_ccw_t **ccw**
- uint32_t **addr**
- pch_sid_t **sid**

The documentation for this struct was generated from the following file:

- base/include/picochan/trc_records.h

12.29 pch_trdata_chp_alloc Struct Reference

Data Fields

- pch_sid_t **first_sid**
- uint16_t **num_devices**
- pch_chpid_t **chpid**

The documentation for this struct was generated from the following file:

- base/include/picochan/trc_records.h

12.30 pch_trdata_cu_register Struct Reference

Data Fields

- uint16_t **num_devices**
- pch_cuaddr_t **cuaddr**

The documentation for this struct was generated from the following file:

- base/include/picochan/trc_records.h

12.31 pch_trdata_cus_call_callback Struct Reference

Data Fields

- pch_cuaddr_t **cuaddr**
- pch_unit_addr_t **ua**
- uint8_t **cbindex**

The documentation for this struct was generated from the following file:

- base/include/picochan/trc_records.h

12.32 pch_trdata_cus_init_mem_channel Struct Reference

Data Fields

- pch_cuaddr_t **cuaddr**
- pch_dmaid_t **txdmaid**
- pch_dmaid_t **rxdmaid**

The documentation for this struct was generated from the following file:

- base/include/picochan/trc_records.h

12.33 pch_trdata_cus_tx_complete Struct Reference

Data Fields

- int16_t **uaopt**
- pch_cuaddr_t **cuaddr**
- uint8_t **txpstate**

The documentation for this struct was generated from the following file:

- base/include/picochan/trc_records.h

12.34 pch_trdata_dev Struct Reference

Data Fields

- pch_cuaddr_t **cuaddr**
- pch_unit_addr_t **ua**

The documentation for this struct was generated from the following file:

- base/include/picochan/trc_records.h

12.35 pch_trdata_dev_byte Struct Reference

Data Fields

- pch_cuaddr_t **cuaddr**
- pch_unit_addr_t **ua**
- uint8_t **byte**

The documentation for this struct was generated from the following file:

- base/include/picochan/trc_records.h

12.36 pch_trdata_dma_init Struct Reference

Data Fields

- uint32_t **addr**
- uint32_t **ctrl**
- uint8_t **id**
- pch_dmaid_t **dmaid**
- pch_dma_irq_index_t **dmairqix**
- uint8_t **core_num**

The documentation for this struct was generated from the following file:

- base/include/picochan/trc_records.h

12.37 pch_trdata_dmachan Struct Reference

Data Fields

- pch_dmaid_t **dmaid**

The documentation for this struct was generated from the following file:

- base/include/picochan/trc_records.h

12.38 pch_trdata_dmachan_memstate Struct Reference

Data Fields

- pch_dmaid_t **dmaid**
- uint8_t **state**

The documentation for this struct was generated from the following file:

- base/include/picochan/trc_records.h

12.39 pch_trdata_dmachan_segment Struct Reference

Data Fields

- uint32_t **addr**
- uint32_t **count**
- pch_dmaid_t **dmaid**

The documentation for this struct was generated from the following file:

- base/include/picochan/trc_records.h

12.40 pch_trdata_dmachan_segment_memstate Struct Reference

Data Fields

- uint32_t **addr**
- uint32_t **count**
- pch_dmaid_t **dmaid**
- uint8_t **state**

The documentation for this struct was generated from the following file:

- base/include/picochan/trc_records.h

12.41 pch_trdata_func_irq Struct Reference

Data Fields

- int16_t **ua_opt**
- pch_chpid_t **chpid**
- uint8_t **tx_active**

The documentation for this struct was generated from the following file:

- base/include/picochan/trc_records.h

12.42 pch_trdata_id_byte Struct Reference

Data Fields

- uint8_t **id**
- uint8_t **byte**

The documentation for this struct was generated from the following file:

- base/include/picochan/trc_records.h

12.43 pch_trdata_id_irq Struct Reference

Data Fields

- uint8_t **id**
- pch_dma_irq_index_t **dmairqix**
- uint8_t **tx_state**
- uint8_t **rx_state**

The documentation for this struct was generated from the following file:

- base/include/picochan/trc_records.h

12.44 pch_trdata_intcode_scsw Struct Reference

Data Fields

- pch_intcode_t **intcode**
- pch_scsw_t **scsw**

The documentation for this struct was generated from the following file:

- base/include/picochan/trc_records.h

12.45 pch_trdata_irq_handler Struct Reference

Data Fields

- uint32_t **handler**
- int16_t **order_priority**
- uint8_t **irqnum**

The documentation for this struct was generated from the following file:

- base/include/picochan/trc_records.h

12.46 pch_trdata_irqnum_opt Struct Reference

Data Fields

- int16_t **irqnum_opt**

The documentation for this struct was generated from the following file:

- base/include/picochan/trc_records.h

12.47 pch_trdata_scsw_sid_cc Struct Reference

Data Fields

- pch_scsw_t **scsw**
- pch_sid_t **sid**
- uint8_t **cc**

The documentation for this struct was generated from the following file:

- base/include/picochan/trc_records.h

12.48 pch_trdata_sid_byte Struct Reference

Data Fields

- pch_sid_t **sid**
- uint8_t **byte**

The documentation for this struct was generated from the following file:

- base/include/picochan/trc_records.h

12.49 pch_trdata_word_byte Struct Reference

Data Fields

- uint32_t **word**
- uint8_t **byte**

The documentation for this struct was generated from the following file:

- base/include/picochan/trc_records.h

12.50 pch_trdata_word_dev Struct Reference

Data Fields

- uint32_t **word**
- pch_cuaddr_t **cuaddr**
- pch_unit_addr_t **ua**

The documentation for this struct was generated from the following file:

- base/include/picochan/trc_records.h

12.51 pch_trdata_word_sid Struct Reference

Data Fields

- uint32_t **word**
- pch_sid_t **sid**

The documentation for this struct was generated from the following file:

- base/include/picochan/trc_records.h

12.52 pch_trdata_word_sid_byte Struct Reference

Data Fields

- uint32_t **word**
- pch_sid_t **sid**
- uint8_t **byte**

The documentation for this struct was generated from the following file:

- base/include/picochan/trc_records.h

12.53 pch_txsm Struct Reference

Data Fields

- pch_txsm_state_t **state**
- uint16_t **count**
- uint32_t **addr**

The documentation for this struct was generated from the following file:

- base/txsm/txsm.h

12.54 proto_packet Struct Reference

a 4-byte command packet sent on a channel between CSS and CU or vice versa

```
#include <packet.h>
```

Data Fields

- proto_chop_t **chop**
- pch_unit_addr_t **unit_addr**
- uint8_t **p0**
- uint8_t **p1**

12.54.1 Detailed Description

a 4-byte command packet sent on a channel between CSS and CU or vice versa

Various parts of this implementation are tuned for and rely on the size being exactly 4 bytes. Note that the ARM ABI specifies that a return value of a composite type of up to 4 bytes (such as proto_packet_t) is passed in R0, thus behaving the same way as a 32-bit return value.

The documentation for this struct was generated from the following file:

- base/proto/packet.h

12.55 proto_parsed_devstatus_payload Struct Reference

Data Fields

- uint16_t **count**
- uint8_t **devs**

The documentation for this struct was generated from the following file:

- base/proto/payload.h

12.56 proto_payload Struct Reference

Data Fields

- uint8_t **p0**
- uint8_t **p1**

The documentation for this struct was generated from the following file:

- base/proto/payload.h

12.57 ua_slist Struct Reference

Data Fields

- int16_t **head**
- int16_t **tail**

The documentation for this struct was generated from the following file:

- css/channel.h

Chapter 13

File Documentation

13.1 dmachan_internal.h

```
/*
 * Copyright (c) 2025 Malcolm Beattie
 * SPDX-License-Identifier: MIT
 */

#ifndef _PCH_DMACHAN_DMACHAN_INTERNAL_H
#define _PCH_DMACHAN_DMACHAN_INTERNAL_H

#include "hardware/sync.h"
#include "dmachan_trace.h"

// dmachan_mem_peer_spin_lock protects against test/update of
// tx_channel.mem_src_state and rx_channel.mem_dst_state both
// from interrupts and cross-core. It must be initialised before
// use with pch_memchan_init().
extern spin_lock_t *dmachan_mem_peer_spin_lock;

static inline uint32_t mem_peer_lock(void) {
#if PCH_CONFIG_ENABLE_MEMCHAN
        return spin_lock_blocking(dmachan_mem_peer_spin_lock);
#else
        return 0;
#endif
}

static inline void mem_peer_unlock(uint32_t saved_irq) {
#if PCH_CONFIG_ENABLE_MEMCHAN
        spin_unlock(dmachan_mem_peer_spin_lock, saved_irq);
#else
        (void)saved_irq;
#endif
}

#endif
```

13.2 dmachan_trace.h

```
/*
 * Copyright (c) 2025 Malcolm Beattie
 * SPDX-License-Identifier: MIT
 */

#ifndef _PCH_DMACHAN_DMACHAN_TRACE_H
#define _PCH_DMACHAN_DMACHAN_TRACE_H

#include "trc/trace.h"
#include "picochan/trc_records.h"

#define PCH_DMACHAN_LINK_TRACE(rt, l, data) \
        PCH_TRC_WRITE(l->bs, l->bs, (rt), (data))

static inline void trace_dmachan(pch_trc_record_type_t rt, dmachan_link_t *l) {
        PCH_DMACHAN_LINK_TRACE(rt, l, ((struct pch_trdata_dmachan){
                .dmaid = l->dmaid
        }));
}

static inline void trace_dmachan_segment(pch_trc_record_type_t rt, dmachan_link_t *l, uint32_t addr,
uint32_t count) {
        PCH_DMACHAN_LINK_TRACE(rt, l, ((struct pch_trdata_dmachan_segment){
                .addr = addr,
                .count = count,
                .dmaid = l->dmaid
        }));
}

static inline void trace_dmachan_memstate(pch_trc_record_type_t rt, dmachan_link_t *l, uint8_t state)
{
        PCH_DMACHAN_LINK_TRACE(rt, l,
                ((struct pch_trdata_dmachan_memstate){
                        .dmaid = l->dmaid,
                        .state = state
                }));
}

static inline void trace_dmachan_segment_memstate(pch_trc_record_type_t rt, dmachan_link_t *l,
uint32_t addr, uint32_t count, uint8_t state) {
        PCH_DMACHAN_LINK_TRACE(rt, l,
                ((struct pch_trdata_dmachan_segment_memstate){
                        .addr = addr,
                        .count = count,
                        .dmaid = l->dmaid,
                        .state = state
                }));
}

#endif
```

13.3 base/include/picochan/bsize.h File Reference

An encoding of 16-bit counts as 8-bit values for typical Pico-sized buffers.

```
#include <stdint.h>
```

Data Structures

- struct pch_bsize

  an 8-bit structure whose value encodes a 16-bit value for use as a count of bytes in a typical picochan buffer or transfer request
- struct pch_bsizex

  a pch_bsize together with a flag intended to indicate whether the bsize encoded the original size exactly.

Macros

- #define PCH_BSIZE_ZERO ((pch_bsize_t){0})

  A constant struct initialiser for the bsize encoding of zero.

Typedefs

- typedef struct pch_bsize pch_bsize_t

  an 8-bit structure whose value encodes a 16-bit value for use as a count of bytes in a typical picochan buffer or transfer request
- typedef struct pch_bsizex pch_bsizex_t

  a pch_bsize together with a flag intended to indicate whether the bsize encoded the original size exactly.

Functions

- pch_bsizex_t **pch_bsize_encodex** (uint16_t n)

  Encode 16-bit count as an pch_bsizex_t.
- pch_bsize_t **pch_bsize_encode** (uint16_t n)

  Encode 16-bit count as an 8-bit pch_bsize_t.
- uint16_t **pch_bsize_decode_raw** (uint8_t esize)

  Decode an 8-bit raw value of a bsize (not in its pch_bsize_t type-wrapping) into a 16-bit value.
- uint16_t **pch_bsize_decode** (pch_bsize_t bsize)

  Decode an 8-bit pch_bsize_t value into a 16-bit value.
- static uint8_t **pch_bsize_unwrap** (pch_bsize_t s)

  Unwraps the uint8_t contained in a pch_bsize_t.
- static pch_bsize_t pch_bsize_wrap (uint8_t esize)

  wraps a uint8_t into a pch_bsize_t
- static uint8_t pch_bsize_encode_raw_inline (uint16_t n)

  Perform a bsize encoding, returning the encoded value unwrapped.
- static pch_bsizex_t pch_bsize_encodex_inline (uint16_t n)

  encode a 16-bit value into its pch_bsize_t along with an "exact"
- static pch_bsize_t pch_bsize_encode_inline (uint16_t n)

  encode a 16-bit value as a pch_bsize_t
- static uint16_t pch_bsize_decode_raw_inline (uint8_t esize)

  decodes a raw bsize-encoded value
- static uint16_t pch_bsize_decode_inline (pch_bsize_t bsize)

  decodes a pch_bsize_t as the uint16_t it represents
- uint8_t **pch_bsize_encode_raw** (uint16_t n)

  Encode a 16-bit value into its raw 8-bit bsize encoding.

13.3.1 Detailed Description

An encoding of 16-bit counts as 8-bit values for typical Pico-sized buffers.

13.3.2 Typedef Documentation

13.3.2.1 pch_bsize_t

```
typedef struct pch_bsize pch_bsize_t
```

an 8-bit structure whose value encodes a 16-bit value for use as a count of bytes in a typical picochan buffer or transfer request

The 8-bit encoding is wrapped as a structure to provide type clarity (even if not full type safety is not possible) when being passed around via the API and stored.

The encoding is not 1-1 (of course) but the decoding of the value obtained by encoding n is always less than or equal to n and "close" when n is a size typically used as a buffer size for workloads using picochan.

The encoding/decoding is exact for the following values:

- 1 x [0, 63] -> 0, 1, 2, ..., 63
- 2 x [32, 95] -> 64, 66, 68, ..., 190
- 8 x [24, 87] -> 192, 200, 208, ..., 696
- 64 x [11, 74] -> 704, 768, 832, ..., 4736

13.4 bsize.h

Go to the documentation of this file.

```
/*
 * Copyright (c) 2025 Malcolm Beattie
 * SPDX-License-Identifier: MIT
 */

#ifndef _PCH_API_BSIZE_H
#define _PCH_API_BSIZE_H

#include <stdint.h>


typedef struct pch_bsize {
        uint8_t esize;
} pch_bsize_t;

#define PCH_BSIZE_ZERO ((pch_bsize_t){0})

typedef struct pch_bsizex {
        uint8_t         exact;
        pch_bsize_t     bsize;
} pch_bsizex_t;

// Non-inlined API functions

pch_bsizex_t pch_bsize_encodex(uint16_t n);

pch_bsize_t pch_bsize_encode(uint16_t n);

uint16_t pch_bsize_decode_raw(uint8_t esize);

uint16_t pch_bsize_decode(pch_bsize_t bsize);

// Inline encode/decode operations

static inline uint8_t pch_bsize_unwrap(pch_bsize_t s) {
        return s.esize;
}

static inline pch_bsize_t pch_bsize_wrap(uint8_t esize) {
        return (pch_bsize_t){esize};
}

static inline uint8_t pch_bsize_encode_raw_inline(uint16_t n) {
    // XXX TODO See if we can just call pch_bsize_encodex_inline
    // and return the contained pch_bsize_t and have gcc
    // reliably optimise it as well as not calculating the
    // exact flag in the first place. For now we just spell it
    // all out again.

    // 0b00nnnnnn - 1 x [0,63] -> 0,1,2,...,63
    if (n <= 63)
                return (uint8_t)n;

    // 0b01nnnnnn - 2 x [32,95] -> 64,66,68,...,190
    if (n <= 191)
                return (uint8_t)(((n»1) - 32) | 0x40);

    // 0b10nnnnnn - 8 x [24,87] -> 192,200,208,...,696
    if (n <= 703)
                return (uint8_t)(((n»3) - 24) | 0x80);

    // 0b11nnnnnn - 64 x [11,74] -> 704,768,832,...,4736
    if (n <= 4736)
                return (uint8_t)(((n»6) - 11) | 0xc0);

        return 0xff;
}

static inline pch_bsizex_t pch_bsize_encodex_inline(uint16_t n) {
    // 0b00nnnnnn - 1 x [0,63] -> 0,1,2,...,63
    if (n <= 63)
                return (pch_bsizex_t){1,pch_bsize_wrap(n)};

    // 0b01nnnnnn - 2 x [32,95] -> 64,66,68,...,190
    if (n <= 191) {
        uint8_t exact = (n & 0x1) == 0;
                pch_bsize_t bsize = pch_bsize_wrap(((n»1) - 32) | 0x40);
        return (pch_bsizex_t){exact,bsize};
    }

    // 0b10nnnnnn - 8 x [24,87] -> 192,200,208,...,696
    if (n <= 703) {
        uint8_t exact = (n & 0x7) == 0;
                pch_bsize_t bsize = pch_bsize_wrap(((n»3) - 24) | 0x80);
        return (pch_bsizex_t){exact,bsize};
    }

    // 0b11nnnnnn - 64 x [11,74] -> 704,768,832,...,4736
    if (n <= 4736) {
        uint8_t exact = (n & 0x3f) == 0;
                pch_bsize_t bsize = pch_bsize_wrap(((n»6) - 11) | 0xc0);
                return (pch_bsizex_t){exact,bsize};
    }

        return (pch_bsizex_t){0, pch_bsize_wrap(0xff)};
}

static inline pch_bsize_t pch_bsize_encode_inline(uint16_t n) {
        return pch_bsize_wrap(pch_bsize_encode_raw_inline(n));
}

static inline uint16_t pch_bsize_decode_raw_inline(uint8_t esize) {
        uint8_t flags = esize & 0xc0;
        uint16_t n = esize & 0x3f;

    switch (flags) {
    case 0x00:
        // 0b00nnnnnn - 1 x [0,63] -> 0,1,2,...,63
        return n;

    case 0x40:
        // 0b01nnnnnn - 2 x [32,95] -> 64,66,68,...,190
        return (n+32) « 1;

    case 0x80:
        // 0b10nnnnnn - 8 x [24,87] -> 192,200,208,...,696
        return (n+24) « 3;
    }

    // 0b11nnnnnn - 64 x [11,74] -> 704,768,832,...,4736
    return (n+11) « 6;
}

static inline uint16_t pch_bsize_decode_inline(pch_bsize_t bsize) {
        return pch_bsize_decode_raw_inline(bsize.esize);
}

uint8_t pch_bsize_encode_raw(uint16_t n);

#endif
```

13.5 base/include/picochan/ccw.h File Reference

Channel-Command Word (CCW)

```
#include <stdint.h>
#include <assert.h>
```

Data Structures

- struct pch_ccw

  I/O Channel-Command Word (CCW)

Macros

- #define **PCH_CCW_FLAG_CD** 0x80
- #define **PCH_CCW_FLAG_CC** 0x40
- #define **PCH_CCW_FLAG_SLI** 0x20
- #define **PCH_CCW_FLAG_SKP** 0x10

- #define **PCH_CCW_FLAG_PCI** 0x08
- #define **PCH_CCW_FLAG_IDA** 0x04
- #define **PCH_CCW_FLAG_S** 0x02
- #define **PCH_CCW_FLAG_MIDA** 0x01
- #define **PCH_CCW_CMD_FIRST_RESERVED** 0xf0
- #define **PCH_CCW_CMD_WRITE** 0x01
- #define **PCH_CCW_CMD_READ** 0x02
- #define **PCH_CCW_CMD_TIC** 0xf0
- #define **PCH_CCW_CMD_SENSE** 0xf2

Typedefs

- typedef uint8_t **pch_ccw_flags_t**

  the flags of a CCW
- typedef struct pch_ccw pch_ccw_t

  I/O Channel-Command Word (CCW)

Functions

- static bool **pch_is_ccw_cmd_write** (uint8_t cmd)

13.5.1 Detailed Description

Channel-Command Word (CCW)

13.6 ccw.h

Go to the documentation of this file.

```
/*
 * Copyright (c) 2025 Malcolm Beattie
 * SPDX-License-Identifier: MIT
 */

#ifndef _PCH_API_CCW_H
#define _PCH_API_CCW_H

#include <stdint.h>
#include <assert.h>


typedef uint8_t pch_ccw_flags_t;

// pch_ccw_flags_t: CCW flags
// CD: Chain Data
#define PCH_CCW_FLAG_CD          0x80
// CC: Chain Command
#define PCH_CCW_FLAG_CC          0x40
// SLI: Suppress Length Indication
#define PCH_CCW_FLAG_SLI         0x20
// SKP: Skip/Discard data
#define PCH_CCW_FLAG_SKP         0x10
// PCI: Program Controlled Interruption
#define PCH_CCW_FLAG_PCI         0x08
// IDA: Indirect Data Address (not used in Picochan)
#define PCH_CCW_FLAG_IDA         0x04
// S: Suspend
#define PCH_CCW_FLAG_S           0x02
// MIDA: Modified Indirect Data Address (not used in Picochan)
#define PCH_CCW_FLAG_MIDA        0x01

typedef struct __attribute__((aligned(4))) pch_ccw {
    uint8_t         cmd;
    pch_ccw_flags_t flags;
    uint16_t        count;
    uint32_t        addr;
} pch_ccw_t;

static_assert(sizeof(pch_ccw_t) == 8, "architected pch_ccw_t is 8 bytes");

// Architected values of CCW commands.
// These do not match those for traditional CSS and we only divide
// into "Read/Write" via the low bit instead of into Control/Read/
// ReadBackward/Sense/Test/Write via various low-bit groups.

#define PCH_CCW_CMD_FIRST_RESERVED 0xf0
// WRITE
#define PCH_CCW_CMD_WRITE       0x01
// READ
#define PCH_CCW_CMD_READ        0x02
// TIC: Transfer In Channel
#define PCH_CCW_CMD_TIC         0xf0
// SENSE: Read Sense data from devib
#define PCH_CCW_CMD_SENSE       0xf2

// Architected bit tests of CCW commands
static inline bool pch_is_ccw_cmd_write(uint8_t cmd) {
        return (cmd & 0x01) == 1;
}

#endif
```

13.7 base/include/picochan/dev_status.h File Reference

Device status bit values.

Macros

- #define **PCH_DEVS_ATTENTION** 0x80
- #define **PCH_DEVS_STATUS_MODIFIER** 0x40
- #define **PCH_DEVS_CONTROL_UNIT_END** 0x20
- #define **PCH_DEVS_BUSY** 0x10
- #define **PCH_DEVS_CHANNEL_END** 0x08
- #define **PCH_DEVS_DEVICE_END** 0x04
- #define **PCH_DEVS_UNIT_CHECK** 0x02
- #define **PCH_DEVS_UNIT_EXCEPTION** 0x01

13.7.1 Detailed Description

Device status bit values.

The device status is an 8-bit architected value that is sent from a device (via its CU) to the CSS at the end of (and sometimes during) the device's execution of a CCW. The device status sent by the device is never modified by the CU or CSS but its bits drive the CSS logic for how to progress/end the channel program and the final device status of a channel program is visible to the application in the SCSW part of the architected schib.

13.8 dev_status.h

Go to the documentation of this file.

```
/*
 * Copyright (c) 2025 Malcolm Beattie
 * SPDX-License-Identifier: MIT
 */

#ifndef _PCH_DEV_STATUS_H
#define _PCH_DEV_STATUS_H

#define PCH_DEVS_ATTENTION        0x80
#define PCH_DEVS_STATUS_MODIFIER  0x40
#define PCH_DEVS_CONTROL_UNIT_END 0x20
#define PCH_DEVS_BUSY             0x10
#define PCH_DEVS_CHANNEL_END      0x08
#define PCH_DEVS_DEVICE_END       0x04
#define PCH_DEVS_UNIT_CHECK       0x02
#define PCH_DEVS_UNIT_EXCEPTION   0x01

#endif
```

13.9 dmachan.h

```
/*
 * Copyright (c) 2025 Malcolm Beattie
 * SPDX-License-Identifier: MIT
 */

#ifndef _PCH_API_DMACHAN_H
#define _PCH_API_DMACHAN_H

// PICO_CONFIG: PARAM_ASSERTIONS_ENABLED_PCH_DMACHAN, Enable/disable assertions in the pch_dmachan
module, type=bool, default=0, group=pch_dmachan
#ifndef PARAM_ASSERTIONS_ENABLED_PCH_DMACHAN
#define PARAM_ASSERTIONS_ENABLED_PCH_DMACHAN 0
#endif

#ifndef PCH_CONFIG_ENABLE_MEMCHAN
#define PCH_CONFIG_ENABLE_MEMCHAN 1
#endif

#include "hardware/dma.h"
#include "hardware/structs/dma_debug.h"
#include "hardware/uart.h"
#include "pico/platform/compiler.h"
#include "picochan/dmachan_defs.h"
#include "picochan/ids.h"
#include "picochan/trc.h"

// General Pico SDK-like DMA-related functions that aren't in the SDK
static inline enum dma_channel_transfer_size channel_config_get_transfer_data_size(dma_channel_config
config) {
        uint size = (config.ctrl & DMA_CH0_CTRL_TRIG_DATA_SIZE_BITS) »
DMA_CH0_CTRL_TRIG_DATA_SIZE_LSB;
        return (enum dma_channel_transfer_size)size;
}

static inline uint32_t dma_channel_get_transfer_count(uint channel) {
        return dma_channel_hw_addr(channel)->transfer_count;
}

static inline dma_debug_channel_hw_t *dma_debug_channel_hw_addr(uint channel) {
        check_dma_channel_param(channel);
        return &dma_debug_hw->ch[channel];
}

static inline uint32_t dma_channel_get_reload_count(uint channel) {
        return dma_debug_channel_hw_addr(channel)->dbg_tcr;
}

static inline bool dma_irqn_get_channel_forced(uint irq_index, uint channel) {
        invalid_params_if(HARDWARE_DMA, irq_index >= NUM_DMA_IRQS);
        check_dma_channel_param(channel);

        return dma_hw->irq_ctrl[irq_index].intf & (1u « channel);
}

static inline void dma_irqn_set_channel_forced(uint irq_index, uint channel, bool forced) {
        invalid_params_if(HARDWARE_DMA, irq_index >= NUM_DMA_IRQS);

        if (forced)
                hw_set_bits(&dma_hw->irq_ctrl[irq_index].intf, 1u « channel);
        else
                hw_clear_bits(&dma_hw->irq_ctrl[irq_index].intf, 1u « channel);
}

// DMA configuration for one direction (tx or rx) of a dmachan channel
typedef struct dmachan_1way_config {
        uint32_t                addr;
        dma_channel_config      ctrl;
        pch_dmaid_t             dmaid;
        pch_dma_irq_index_t     dmairqix;
} dmachan_1way_config_t;

static inline dmachan_1way_config_t dmachan_1way_config_make(pch_dmaid_t dmaid, uint32_t addr,
dma_channel_config ctrl, pch_dma_irq_index_t dmairqix) {
        return ((dmachan_1way_config_t){
                .addr = addr,
                .ctrl = ctrl,
                .dmaid = dmaid,
                .dmairqix = dmairqix
        });
}

static inline dmachan_1way_config_t dmachan_1way_config_claim(uint32_t addr, dma_channel_config ctrl,
pch_dma_irq_index_t dmairqix) {
        pch_dmaid_t dmaid = (pch_dmaid_t)dma_claim_unused_channel(true);
        return dmachan_1way_config_make(dmaid, addr, ctrl, dmairqix);
}

static inline dmachan_1way_config_t dmachan_1way_config_memchan_make(pch_dmaid_t dmaid,
pch_dma_irq_index_t dmairqix) {
        dma_channel_config ctrl = dma_channel_get_default_config(dmaid);
        channel_config_set_transfer_data_size(&ctrl, DMA_SIZE_8);
        channel_config_set_read_increment(&ctrl, true);
        channel_config_set_write_increment(&ctrl, true);
        return ((dmachan_1way_config_t){
                .addr = 0,
                .ctrl = ctrl,
                .dmaid = dmaid,
                .dmairqix = dmairqix
        });
}

// DMA configuration for both directions (tx and rx) of a dmachan
// channel
typedef struct dmachan_config {
        dmachan_1way_config_t   tx;
        dmachan_1way_config_t   rx;
} dmachan_config_t;

static inline dmachan_config_t dmachan_config_claim(uint32_t txaddr, dma_channel_config txctrl,
uint32_t rxaddr, dma_channel_config rxctrl, pch_dma_irq_index_t dmairqix) {
        return ((dmachan_config_t){
                .tx = dmachan_1way_config_claim(txaddr, txctrl, dmairqix),
                .rx = dmachan_1way_config_claim(rxaddr, rxctrl, dmairqix)
        });
}

static inline dmachan_config_t dmachan_config_memchan_make(pch_dmaid_t txdmaid, pch_dmaid_t rxdmaid,
pch_dma_irq_index_t dmairqix) {
        return ((dmachan_config_t){
                .tx = dmachan_1way_config_memchan_make(txdmaid, dmairqix),
                .rx = dmachan_1way_config_memchan_make(rxdmaid, dmairqix)
        });
}

typedef union __aligned(4) dmachan_cmd {
        unsigned char   buf[4];
        uint32_t        raw;
} dmachan_cmd_t;

#define DMACHAN_CMD_SIZE sizeof(dmachan_cmd_t)
static_assert(DMACHAN_CMD_SIZE == 4, "dmachan_cmd_t must be 4 bytes");

static inline void dmachan_cmd_set_zero(dmachan_cmd_t *cmd) {
        cmd->raw = 0;
}

static inline void dmachan_cmd_copy(dmachan_cmd_t *dst, dmachan_cmd_t *src) {
        dst->raw = src->raw;
}

#define DMACHAN_CMD_COPY(cmdp, p) do { \
        dmachan_cmd_t *__cmdp = (cmdp); \
        static_assert(sizeof(*(p)) == DMACHAN_CMD_SIZE, \
                "DMACHAN_CMD_COPY sizeof(*p) must be DMACHAN_CMD_SIZE"); \
        __cmdp->raw = *(uint32_t*)(p); \
} while (0)

// dmachan_link_t collects the common fields in tx and rx channels
typedef struct __aligned(4) dmachan_link {
        dmachan_cmd_t           cmd;
        pch_trc_bufferset_t     *bs;            // only when tracing
        pch_dmaid_t             dmaid;
        pch_dma_irq_index_t     dmairqix;
        bool                    complete;
        bool                    resetting;
} dmachan_link_t;

#define DMACHAN_LINK_CMD_COPY(l, p) do { \
        dmachan_link_t *__l = (l); \
        DMACHAN_CMD_COPY(&__l->cmd, p); \
} while (0)

static inline void dmachan_set_link_bs(dmachan_link_t *l, pch_trc_bufferset_t *bs) {
        l->bs = bs;
}

static inline void dmachan_link_cmd_set_zero(dmachan_link_t *l) {
        dmachan_cmd_set_zero(&l->cmd);
}

static inline void dmachan_link_cmd_copy(dmachan_link_t *dst, dmachan_link_t *src) {
        dmachan_cmd_copy(&dst->cmd, &src->cmd);
}

static inline void dmachan_set_link_irq_enabled(dmachan_link_t *l, bool enabled) {
        pch_dma_irq_index_t dmairqix = l->dmairqix;
        assert(dmairqix >= 0 && dmairqix < NUM_DMA_IRQS);
        dma_irqn_set_channel_enabled(dmairqix, l->dmaid, enabled);
}

static inline bool dmachan_link_irq_raised(dmachan_link_t *l) {
        return dma_irqn_get_channel_status(l->dmairqix, l->dmaid);
};

static inline bool dmachan_get_link_irq_forced(dmachan_link_t *l) {
        return dma_irqn_get_channel_forced(l->dmairqix, l->dmaid);
}

static inline void dmachan_set_link_irq_forced(dmachan_link_t *l, bool forced) {
        dma_irqn_set_channel_forced(l->dmairqix, l->dmaid, forced);
}

static inline void dmachan_ack_link_irq(dmachan_link_t *l) {
        dma_irqn_acknowledge_channel(l->dmairqix, l->dmaid);
}

// tx and rx channels, starting with forward declarations because
// for memchans there is a field pointing at the peer channel
typedef struct __aligned(4) dmachan_tx_channel dmachan_tx_channel_t;
typedef struct __aligned(4) dmachan_rx_channel dmachan_rx_channel_t;

typedef struct __aligned(4) dmachan_tx_channel {
        dmachan_link_t          link;
        dmachan_rx_channel_t    *mem_rx_peer;   // only for memchan
        dmachan_mem_src_state_t mem_src_state;  // only for memchan
} dmachan_tx_channel_t;

typedef struct __aligned(4) dmachan_rx_channel {
        dmachan_link_t          link;
        dmachan_tx_channel_t    *mem_tx_peer;   // only for memchan
        uint32_t                srcaddr;
        dma_channel_config      ctrl;
        dmachan_mem_dst_state_t mem_dst_state;  // only for memchan
} dmachan_rx_channel_t;

static inline dmachan_irq_state_t dmachan_make_irq_state(bool raised, bool forced, bool complete) {
        return ((dmachan_irq_state_t)raised)
                | ((dmachan_irq_state_t)forced) « 1
                | ((dmachan_irq_state_t)complete) « 2;
}

// tx channel irq and memory source state handling
static inline void dmachan_set_mem_src_state(dmachan_tx_channel_t *tx, dmachan_mem_src_state_t
new_state) {
        valid_params_if(PCH_DMACHAN,
                new_state == DMACHAN_MEM_SRC_IDLE
                || tx->mem_src_state == DMACHAN_MEM_SRC_IDLE);

        tx->mem_src_state = new_state;
}
```

```c

dmachan_irq_state_t dmachan_handle_tx_irq(dmachan_tx_channel_t *tx);

// rx channel irq and memory destination state handling
static inline void dmachan_set_mem_dst_state(dmachan_rx_channel_t *rx, dmachan_mem_dst_state_t
new_state) {
        valid_params_if(PCH_DMACHAN,
                new_state == DMACHAN_MEM_DST_IDLE
                || rx->mem_dst_state == DMACHAN_MEM_DST_IDLE);

        rx->mem_dst_state = new_state;
}

dmachan_irq_state_t dmachan_handle_rx_irq(dmachan_rx_channel_t *rx);

void dmachan_panic_unless_memchan_initialised(void);

void dmachan_init_tx_channel(dmachan_tx_channel_t *tx, dmachan_1way_config_t *cfg);
void dmachan_start_src_cmdbuf(dmachan_tx_channel_t *tx);
void dmachan_write_src_reset(dmachan_tx_channel_t *tx);
void dmachan_start_src_data(dmachan_tx_channel_t *tx, uint32_t srcaddr, uint32_t count);

void dmachan_init_rx_channel(dmachan_rx_channel_t *rx, dmachan_1way_config_t *cfg);
void dmachan_start_dst_reset(dmachan_rx_channel_t *rx);
void dmachan_start_dst_cmdbuf(dmachan_rx_channel_t *rx);
void dmachan_start_dst_data(dmachan_rx_channel_t *rx, uint32_t dstaddr, uint32_t count);
void dmachan_start_dst_discard(dmachan_rx_channel_t *rx, uint32_t count);
void dmachan_start_dst_data_src_zeroes(dmachan_rx_channel_t *rx, uint32_t dstaddr, uint32_t count);

// Convenience functions for configuring UART channels
void pch_uart_init(uart_inst_t *uart, uint baudrate);

static inline dma_channel_config dmachan_uart_make_txctrl(uart_inst_t *uart, dma_channel_config ctrl)
{
        channel_config_set_transfer_data_size(&ctrl, DMA_SIZE_8);
        channel_config_set_write_increment(&ctrl, false);
        uint txdreq = uart_get_dreq_num(uart, true);
        channel_config_set_dreq(&ctrl, txdreq);
        return ctrl;
}

static inline dma_channel_config dmachan_uart_make_rxctrl(uart_inst_t *uart, dma_channel_config ctrl)
{
        channel_config_set_transfer_data_size(&ctrl, DMA_SIZE_8);
        channel_config_set_read_increment(&ctrl, false);
        uint rxdreq = uart_get_dreq_num(uart, false);
        channel_config_set_dreq(&ctrl, rxdreq);
        return ctrl;
}

// pch_memchan_init must be called before configuring either side of
// any memchan CU with pch_cus_memcu_configure or
// pch_chp_configure_memchan
void pch_memchan_init(void);

#endif
```

13.10 dmachan_defs.h

```c
/*
 * Copyright (c) 2025 Malcolm Beattie
 * SPDX-License-Identifier: MIT
 */

#ifndef _PCH_API_DMACHAN_DEFS_H
#define _PCH_API_DMACHAN_DEFS_H

// dmachan_mem_src_state_t is the DMA state of a tx channel
typedef enum __attribute__((packed)) dmachan_mem_src_state {
        DMACHAN_MEM_SRC_IDLE = 0,
        DMACHAN_MEM_SRC_CMDBUF,
        DMACHAN_MEM_SRC_DATA
} dmachan_mem_src_state_t;

// dmachan_mem_dst_state_t is the DMA state of an rx channel
typedef enum __attribute__((packed)) dmachan_mem_dst_state {
        DMACHAN_MEM_DST_IDLE = 0,
        DMACHAN_MEM_DST_CMDBUF,
        DMACHAN_MEM_DST_DATA,
        DMACHAN_MEM_DST_DISCARD,
        DMACHAN_MEM_DST_SRC_ZEROES
} dmachan_mem_dst_state_t;

```

```
// dmachan_irq_state_t represents the state of a given DMA id
// with respect to an interrupt for a given DMA IRQ number.
// REASON_RAISED means either there was a DMA engine completion
// causing the bit for the DMA id to be set in register INTSn for
// that DMA IRQ index or, apparently, the INTFn forced bit was set
// explicitly.
// REASON_FORCED means the bit for the DMA id was explicitly set in
// register INTFn for that DMA IRQ index, ignoring the value of the
// enable bit in the corresponding INTEn register. It also seems to
// cause the corresponding INTSn bit to be seen as 1 too so
// REASON_RAISED will (always?) be set if REASON_FORCED is.
// COMPLETE is the value of the link's complete field at the end of
// the dmachan_handle_tx_irq and dmachan_handle_rx_irq functions:
// it will be 1 if either the RAISED or FORCED conditions hold or if
// the field was set explicitly beforehand as a way of causing
// completion handling locally without an irq being triggered.
typedef uint8_t dmachan_irq_state_t;
#define DMACHAN_IRQ_REASON_RAISED       0x1
#define DMACHAN_IRQ_REASON_FORCED       0x2
#define DMACHAN_IRQ_COMPLETE            0x4

#define DMACHAN_IRQ_REASON_MASK         0x3

// ASCII 'C' is the byte we look for when resetting a dmachan
// receive link so that we can resynchronise by dropping any zero
// bytes that are generated by Break conditions from the sender.
#define DMACHAN_RESET_BYTE 0x43

#endif
```

13.11 base/include/picochan/ids.h File Reference

```
#include <stdint.h>
```

Typedefs

- typedef uint16_t **pch_sid_t**

  a subchannel id (SID) between 0 and PCH_NUM_SCHIBS-1 (at most 65535)
- typedef uint8_t **pch_cuaddr_t**

  a control unit address between 0 and PCH_NUM_CUS-1 (at most 255) that identifies a control unit from the CU side.
- typedef uint8_t pch_unit_addr_t

  a unit address that identifies a device on a given CU on the control unit side.
- typedef uint8_t pch_chpid_t

  a channel path identifier between 0 and PCH_NUM_CHANNELS-1 (at most 255) that identifies a channel from the CSS side
- typedef uint8_t pch_dmaid_t

  a DMA id used by CSS or CU
- typedef int8_t pch_dma_irq_index_t

  a DMA IRQ index

13.12 ids.h

Go to the documentation of this file.

```
/*
 * Copyright (c) 2025 Malcolm Beattie
 * SPDX-License-Identifier: MIT
 */

#ifndef _PCH_API_IDS_H
#define _PCH_API_IDS_H

```

```
#include <stdint.h>

typedef uint16_t pch_sid_t;

typedef uint8_t pch_cuaddr_t;

typedef uint8_t pch_unit_addr_t;

typedef uint8_t pch_chpid_t;

typedef uint8_t pch_dmaid_t;

typedef int8_t pch_dma_irq_index_t;

#endif
```

13.13 base/include/picochan/intcode.h File Reference

```
#include "picochan/ids.h"
```

Data Structures

- struct pch_intcode

Typedefs

- typedef struct pch_intcode pch_intcode_t

13.13.1 Typedef Documentation

13.13.1.1 pch_intcode_t

```
typedef struct pch_intcode pch_intcode_t
```

pch_intcode_t is the I/O interruption code which is returned from pch_test_pending_interruption.

The original expansion of the acronym SID is Subsystem-Identification Word which is 32 bits and includes some bits of data beyond just the subchannel number. For Picochan we only use the 16-bit subchannel number so calling this the SID is more appropriate.

```
pch_intcode_t
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |               Interruption Parameter (Intparm)                |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |  Subchannel ID (SID)           |      ISC       |           |cc |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

cc is the condition code which, for a return from pch_test_pending_interruption, only uses two values: 0 means there was no interrupt pending and the rest of the pch_intcode_t is meaningless; 1 means an interrupt was pending and its information has been returned.

13.14 intcode.h

Go to the documentation of this file.

```
/*
 * Copyright (c) 2025 Malcolm Beattie
 * SPDX-License-Identifier: MIT
 */
#ifndef _PCH_API_INTCODE_H
#define _PCH_API_INTCODE_H

#include "picochan/ids.h"


typedef struct pch_intcode {
        uint32_t        intparm;
        pch_sid_t       sid;
        uint8_t         flags;
        uint8_t         cc;
} pch_intcode_t;
static_assert(sizeof(pch_intcode_t) == 8,
        "architected pch_intcode_t is 8 bytes");

#endif
```

13.15 base/include/picochan/scsw.h File Reference

```
#include <stdint.h>
#include <assert.h>
```

Data Structures

- struct pch_scsw

Macros

- #define **PCH_SF_CC_MASK** 0xc0
- #define **PCH_SF_CC_SHIFT** 6
- #define **PCH_SF_P** 0x20
- #define **PCH_SF_I** 0x10
- #define **PCH_SF_U** 0x08
- #define **PCH_SF_Z** 0x04
- #define **PCH_SF_UNUSED** 0x02
- #define **PCH_SF_N** 0x01
- #define **PCH_SCSW_CCW_WRITE** 0x8000
- #define **PCH_FC_MASK** 0x7000
- #define **PCH_FC_START** 0x4000
- #define **PCH_FC_HALT** 0x2000
- #define **PCH_FC_CLEAR** 0x1000
- #define **PCH_AC_MASK** 0x0fe0
- #define **PCH_AC_RESUME_PENDING** 0x0800
- #define **PCH_AC_START_PENDING** 0x0400
- #define **PCH_AC_HALT_PENDING** 0x0200
- #define **PCH_AC_CLEAR_PENDING** 0x0100
- #define **PCH_AC_SUBCHANNEL_ACTIVE** 0x0080
- #define **PCH_AC_DEVICE_ACTIVE** 0x0040
- #define **PCH_AC_SUSPENDED** 0x0020

- #define **PCH_SC_MASK** 0x001f
- #define **PCH_SC_ALERT** 0x0010
- #define **PCH_SC_INTERMEDIATE** 0x0008
- #define **PCH_SC_PRIMARY** 0x0004
- #define **PCH_SC_SECONDARY** 0x0002
- #define **PCH_SC_PENDING** 0x0001
- #define **PCH_SCHS_PROGRAM_CONTROLLED_INTERRUPTION** 0x80
- #define **PCH_SCHS_INCORRECT_LENGTH** 0x40
- #define **PCH_SCHS_PROGRAM_CHECK** 0x20
- #define **PCH_SCHS_PROTECTION_CHECK** 0x10
- #define **PCH_SCHS_CHANNEL_DATA_CHECK** 0x08
- #define **PCH_SCHS_CHANNEL_CONTROL_CHECK** 0x04
- #define **PCH_SCHS_INTERFACE_CONTROL_CHECK** 0x02
- #define **PCH_SCHS_CHAINING_CHECK** 0x01

Typedefs

- typedef struct pch_scsw pch_scsw_t

13.15.1 Typedef Documentation

13.15.1.1 pch_scsw_t

```
typedef struct pch_scsw pch_scsw_t
```

pch_scsw_t is the Subchannel Status Word (SCSW) which must be 4-byte aligned. When marshalling/unmarshalling an SCSW, unlike the original architected SCSW which was implicitly big-endian, the ccw_addr and count fields here are treated as native-endian and so will be little-endian on both ARM and RISC-V (in Pico configurations) and would also be so on x86, for example. The flags fields are slightly rearranged from their original architected positions and some have been dropped and one or two added.

```
SCSW    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |               | CC|P|I|U|Z| |N|W|  FC |     AC      |   SC    |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |                         CCW Address                           |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |      DEVS     |     SCHS      |     Residual Count            |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

13.16 scsw.h

Go to the documentation of this file.

```
/*
 * Copyright (c) 2025 Malcolm Beattie
 * SPDX-License-Identifier: MIT
 */

#ifndef _PCH_API_SCSW_H
#define _PCH_API_SCSW_H

#include <stdint.h>
#include <assert.h>


#define PCH_SF_CC_MASK  0xc0
```

```
#define PCH_SF_CC_SHIFT 6
#define PCH_SF_P        0x20
#define PCH_SF_I        0x10
#define PCH_SF_U        0x08
#define PCH_SF_Z        0x04
#define PCH_SF_UNUSED   0x02
#define PCH_SF_N        0x01

// uint16_t of SCSW Control flags W,FC,AC,SC (Function, Activity, Status)
#define PCH_SCSW_CCW_WRITE       0x8000

#define PCH_FC_MASK              0x7000
#define PCH_FC_START             0x4000
#define PCH_FC_HALT              0x2000
#define PCH_FC_CLEAR             0x1000

#define PCH_AC_MASK              0x0fe0
#define PCH_AC_RESUME_PENDING    0x0800
#define PCH_AC_START_PENDING     0x0400
#define PCH_AC_HALT_PENDING      0x0200
#define PCH_AC_CLEAR_PENDING     0x0100
#define PCH_AC_SUBCHANNEL_ACTIVE 0x0080
#define PCH_AC_DEVICE_ACTIVE     0x0040
#define PCH_AC_SUSPENDED         0x0020

#define PCH_SC_MASK              0x001f
#define PCH_SC_ALERT             0x0010
#define PCH_SC_INTERMEDIATE      0x0008
#define PCH_SC_PRIMARY           0x0004
#define PCH_SC_SECONDARY         0x0002
#define PCH_SC_PENDING           0x0001

// uint8_t of Subchannel Status (SCHS) flags
#define PCH_SCHS_PROGRAM_CONTROLLED_INTERRUPTION 0x80
#define PCH_SCHS_INCORRECT_LENGTH                0x40
#define PCH_SCHS_PROGRAM_CHECK                   0x20
#define PCH_SCHS_PROTECTION_CHECK                0x10
#define PCH_SCHS_CHANNEL_DATA_CHECK              0x08
#define PCH_SCHS_CHANNEL_CONTROL_CHECK           0x04
#define PCH_SCHS_INTERFACE_CONTROL_CHECK         0x02
#define PCH_SCHS_CHAINING_CHECK                  0x01

typedef struct __attribute__((aligned(4))) pch_scsw {
    uint8_t     __unused_flags;
    uint8_t     user_flags;
    uint16_t    ctrl_flags;
    uint32_t    ccw_addr;
    uint8_t     devs;
    uint8_t     schs;
    uint16_t    count;
} pch_scsw_t;
static_assert(sizeof(pch_scsw_t) == 12, "architected pch_scsw_t is 12 bytes");

#endif
```

13.17 base/include/picochan/trc.h File Reference

```
#include "picochan/ids.h"
#include "picochan/trc_record_types.h"
```

Data Structures

- struct pch_trc_timestamp

  an opaque timestamp of a 48-bit number of microseconds since boot.
- struct pch_trc_header
- struct pch_trc_bufferset

  set of buffers and metadata for a subsystem to use tracing

Macros

- #define PCH_TRC_RT(rt)
- #define PCH_CONFIG_ENABLE_TRACE 0

  Whether any tracing code should be compiled at at..
- #define **PCH_TRC_BUFFER_SIZE** 0
- #define **PCH_TRC_NUM_BUFFERS** 1

Typedefs

- typedef struct pch_trc_timestamp pch_trc_timestamp_t

  an opaque timestamp of a 48-bit number of microseconds since boot.
- typedef enum pch_trc_record_type **pch_trc_record_type_t**
- typedef struct pch_trc_header **pch_trc_header_t**
- typedef struct pch_trc_bufferset pch_trc_bufferset_t

  set of buffers and metadata for a subsystem to use tracing

Enumerations

- enum **pch_trc_record_type** {
  PCH_TRC_RT_INVALID , **PCH_TRC_RT_CSS_SCH_START** , **PCH_TRC_RT_CSS_SCH_RESUME** , **PCH_TRC_RT_CSS_SCH_TEST** ,
  PCH_TRC_RT_CSS_SCH_MODIFY , **PCH_TRC_RT_CSS_SCH_STORE** , **PCH_TRC_RT_CSS_SCH_CANCEL** , **PCH_TRC_RT_CSS_SCH_CLEAR** ,
  PCH_TRC_RT_CSS_SCH_HALT , **PCH_TRC_RT_CSS_CCW_FETCH** , **PCH_TRC_RT_CSS_CHP_ALLOC** , **PCH_TRC_RT_CSS_CHP_TX_DMA_INIT** ,
  PCH_TRC_RT_CSS_CHP_RX_DMA_INIT , **PCH_TRC_RT_CSS_CHP_CONFIGURED** , **PCH_TRC_RT_CSS_CHP_TRACED** , **PCH_TRC_RT_CSS_CHP_STARTED** ,
  PCH_TRC_RT_CSS_CHP_IRQ , **PCH_TRC_RT_CSS_RX_COMMAND_COMPLETE** , **PCH_TRC_RT_CSS_RX_DATA_COMPLETE** , **PCH_TRC_RT_CSS_SEND_TX_PACKET** ,
  PCH_TRC_RT_CSS_TX_COMPLETE , **PCH_TRC_RT_CSS_CORE_NUM** , **PCH_TRC_RT_CSS_SET_DMA_IRQ** , **PCH_TRC_RT_CSS_SET_FUNC_IRQ** ,
  PCH_TRC_RT_CSS_SET_IO_IRQ , **PCH_TRC_RT_CSS_SET_IO_CALLBACK** , **PCH_TRC_RT_CSS_NOTIFY** , **PCH_TRC_RT_CSS_FUNC_IRQ** ,
  PCH_TRC_RT_CSS_IO_CALLBACK , **PCH_TRC_RT_CSS_INIT_DMA_IRQ_HANDLER** , **PCH_TRC_RT_CSS_INIT_FUNC_IRQ_HANDLER** , **PCH_TRC_RT_CSS_INIT_IO_IRQ_HANDLER** ,
  PCH_TRC_RT_CUS_QUEUE_COMMAND , **PCH_TRC_RT_CUS_INIT_DMA_IRQ_HANDLER** , **PCH_TRC_RT_CUS_CU_REGISTER** , **PCH_TRC_RT_CUS_CU_TX_DMA_INIT** ,
  PCH_TRC_RT_CUS_CU_RX_DMA_INIT , **PCH_TRC_RT_CUS_CU_CONFIGURED** , **PCH_TRC_RT_CUS_CU_TRACED** , **PCH_TRC_RT_CUS_CU_STARTED** ,
  PCH_TRC_RT_CUS_CU_IRQ , **PCH_TRC_RT_CUS_DEV_TRACED** , **PCH_TRC_RT_CUS_SEND_TX_PACKET** , **PCH_TRC_RT_CUS_TX_COMPLETE** ,
  PCH_TRC_RT_CUS_REGISTER_CALLBACK , **PCH_TRC_RT_CUS_CALL_CALLBACK** , **PCH_TRC_RT_CUS_RX_COMMAND_COMPLETE** , **PCH_TRC_RT_CUS_RX_DATA_COMPLETE** ,
  PCH_TRC_RT_DMACHAN_DST_CMDBUF_REMOTE , **PCH_TRC_RT_DMACHAN_DST_CMDBUF_MEM** , **PCH_TRC_RT_DMACHAN_DST_RESET_REMOTE** , **PCH_TRC_RT_DMACHAN_DST_RESET_MEM** ,
  PCH_TRC_RT_DMACHAN_DST_DATA_REMOTE , **PCH_TRC_RT_DMACHAN_DST_DATA_MEM** , **PCH_TRC_RT_DMACHAN_DST_DISCARD_REMOTE** , **PCH_TRC_RT_DMACHAN_DST_DISCARD_MEM** ,
  PCH_TRC_RT_DMACHAN_SRC_CMDBUF_REMOTE , **PCH_TRC_RT_DMACHAN_SRC_CMDBUF_MEM** , **PCH_TRC_RT_DMACHAN_SRC_RESET_REMOTE** , **PCH_TRC_RT_DMACHAN_SRC_RESET_MEM** ,
  PCH_TRC_RT_DMACHAN_SRC_DATA_REMOTE , **PCH_TRC_RT_DMACHAN_SRC_DATA_MEM** , **PCH_TRC_RT_TRC_ENABLE** }

Functions

- static uint64_t **pch_trc_timestamp_to_us** (pch_trc_timestamp_t t)
- static void **pch_trc_write_timestamp** (pch_trc_timestamp_t ∗tp, uint64_t us)

13.17.1 Macro Definition Documentation

13.17.1.1 PCH_TRC_RT

```
#define PCH_TRC_RT(
                rt)
```

Value:

```
PCH_TRC_RT_ ## rt
```

13.18 trc.h

Go to the documentation of this file.

```
/*
 * Copyright (c) 2025 Malcolm Beattie
 * SPDX-License-Identifier: MIT
 */

#ifndef _PCH_API_TRC_H
#define _PCH_API_TRC_H

#include "picochan/ids.h"


typedef struct pch_trc_timestamp {
        uint16_t low;
        uint16_t mid;
        uint16_t high;
} pch_trc_timestamp_t;

static inline uint64_t pch_trc_timestamp_to_us(pch_trc_timestamp_t t) {
        return (((uint64_t) t.high) « 32)
                + (((uint64_t) t.mid) « 16)
                + (((uint64_t) t.low));
}

static inline void pch_trc_write_timestamp(pch_trc_timestamp_t *tp, uint64_t us) {
        uint16_t *hp = (uint16_t*)tp;
        hp[0] = (uint16_t)us; // low 16 bits: t.low
        hp[1] = (uint16_t)(us » 16); // middle 16 bits: t.mid
        hp[2] = (uint16_t)(us » 32); // low 16 bits of top 32: t.high
}

// The following macro definition nastiness allows a host-based
// trace dump program to redefine these macros to build a list
// of the record type names along with the enum values themselves.
#define PCH_TRC_RT(rt) PCH_TRC_RT_ ## rt

typedef enum __attribute__((__packed__)) pch_trc_record_type {
#include "picochan/trc_record_types.h"
} pch_trc_record_type_t;

typedef struct __attribute__((__packed__,__aligned__(2))) pch_trc_header {
        pch_trc_timestamp_t     timestamp;
        uint8_t                 size; // includes header and following data
        pch_trc_record_type_t   rec_type;
} pch_trc_header_t;

#ifndef PCH_CONFIG_ENABLE_TRACE
#define PCH_CONFIG_ENABLE_TRACE 0
#endif

#if PCH_CONFIG_ENABLE_TRACE

```

```
#ifndef PCH_TRC_BUFFER_SIZE
#define PCH_TRC_BUFFER_SIZE 1024
#endif

#ifndef PCH_TRC_NUM_BUFFERS
#define PCH_TRC_NUM_BUFFERS 2
#endif

extern uint32_t pch_trc_buffer_size;

extern uint32_t pch_trc_num_buffers;

#else
// PCH_CONFIG_ENABLE_TRACE is not defined

#ifndef PCH_TRC_BUFFER_SIZE
#define PCH_TRC_BUFFER_SIZE 0
#endif

#ifndef PCH_TRC_NUM_BUFFERS
#define PCH_TRC_NUM_BUFFERS 1
#endif

#endif
// end of PCH_CONFIG_ENABLE_TRACE section

typedef struct pch_trc_bufferset {
        uint32_t        current_buffer_num;

        uint32_t        current_buffer_pos;

        int16_t         irqnum;

        bool            enable;

        uint32_t        magic;
        uint32_t        buffer_size;
        uint16_t        num_buffers;
        void            *buffers[PCH_TRC_NUM_BUFFERS];
} pch_trc_bufferset_t;

#endif
```

13.19 trc_record_types.h

```
/*
 * Copyright (c) 2025 Malcolm Beattie
 * SPDX-License-Identifier: MIT
 */

PCH_TRC_RT(INVALID),
PCH_TRC_RT(CSS_SCH_START),
PCH_TRC_RT(CSS_SCH_RESUME),
PCH_TRC_RT(CSS_SCH_TEST),
PCH_TRC_RT(CSS_SCH_MODIFY),
PCH_TRC_RT(CSS_SCH_STORE),
PCH_TRC_RT(CSS_SCH_CANCEL),
PCH_TRC_RT(CSS_SCH_CLEAR),
PCH_TRC_RT(CSS_SCH_HALT),
PCH_TRC_RT(CSS_CCW_FETCH),
PCH_TRC_RT(CSS_CHP_ALLOC),
PCH_TRC_RT(CSS_CHP_TX_DMA_INIT),
PCH_TRC_RT(CSS_CHP_RX_DMA_INIT),
PCH_TRC_RT(CSS_CHP_CONFIGURED),
PCH_TRC_RT(CSS_CHP_TRACED),
PCH_TRC_RT(CSS_CHP_STARTED),
PCH_TRC_RT(CSS_CHP_IRQ),
PCH_TRC_RT(CSS_RX_COMMAND_COMPLETE),
PCH_TRC_RT(CSS_RX_DATA_COMPLETE),
PCH_TRC_RT(CSS_SEND_TX_PACKET),
PCH_TRC_RT(CSS_TX_COMPLETE),
PCH_TRC_RT(CSS_CORE_NUM),
PCH_TRC_RT(CSS_SET_DMA_IRQ),
PCH_TRC_RT(CSS_SET_FUNC_IRQ),
PCH_TRC_RT(CSS_SET_IO_IRQ),
PCH_TRC_RT(CSS_SET_IO_CALLBACK),
PCH_TRC_RT(CSS_NOTIFY),
PCH_TRC_RT(CSS_FUNC_IRQ),
PCH_TRC_RT(CSS_IO_CALLBACK),
PCH_TRC_RT(CSS_INIT_DMA_IRQ_HANDLER),
PCH_TRC_RT(CSS_INIT_FUNC_IRQ_HANDLER),
PCH_TRC_RT(CSS_INIT_IO_IRQ_HANDLER),
PCH_TRC_RT(CUS_QUEUE_COMMAND),
PCH_TRC_RT(CUS_INIT_DMA_IRQ_HANDLER),
PCH_TRC_RT(CUS_CU_REGISTER),
PCH_TRC_RT(CUS_CU_TX_DMA_INIT),
PCH_TRC_RT(CUS_CU_RX_DMA_INIT),
PCH_TRC_RT(CUS_CU_CONFIGURED),
PCH_TRC_RT(CUS_CU_TRACED),
PCH_TRC_RT(CUS_CU_STARTED),
PCH_TRC_RT(CUS_CU_IRQ),
PCH_TRC_RT(CUS_DEV_TRACED),
PCH_TRC_RT(CUS_SEND_TX_PACKET),
PCH_TRC_RT(CUS_TX_COMPLETE),
PCH_TRC_RT(CUS_REGISTER_CALLBACK),
PCH_TRC_RT(CUS_CALL_CALLBACK),
PCH_TRC_RT(CUS_RX_COMMAND_COMPLETE),
PCH_TRC_RT(CUS_RX_DATA_COMPLETE),
PCH_TRC_RT(DMACHAN_DST_CMDBUF_REMOTE),
PCH_TRC_RT(DMACHAN_DST_CMDBUF_MEM),
PCH_TRC_RT(DMACHAN_DST_RESET_REMOTE),
PCH_TRC_RT(DMACHAN_DST_RESET_MEM),
PCH_TRC_RT(DMACHAN_DST_DATA_REMOTE),
PCH_TRC_RT(DMACHAN_DST_DATA_MEM),
PCH_TRC_RT(DMACHAN_DST_DISCARD_REMOTE),
PCH_TRC_RT(DMACHAN_DST_DISCARD_MEM),
PCH_TRC_RT(DMACHAN_SRC_CMDBUF_REMOTE),
PCH_TRC_RT(DMACHAN_SRC_CMDBUF_MEM),
PCH_TRC_RT(DMACHAN_SRC_RESET_REMOTE),
PCH_TRC_RT(DMACHAN_SRC_RESET_MEM),
PCH_TRC_RT(DMACHAN_SRC_DATA_REMOTE),
PCH_TRC_RT(DMACHAN_SRC_DATA_MEM),
PCH_TRC_RT(TRC_ENABLE)
```

13.20 trc_records.h

```
/*
 * Copyright (c) 2025 Malcolm Beattie
 * SPDX-License-Identifier: MIT
 */

#ifndef _PCH_API_TRC_RECORDS_H
#define _PCH_API_TRC_RECORDS_H

#include "picochan/ids.h"
#include "picochan/ccw.h"
#include "picochan/scsw.h"
#include "picochan/intcode.h"

// Common structs for the data parts of trace records

struct pch_trdata_byte {
        uint8_t         byte;
};

struct pch_trdata_id_byte {
        uint8_t         id;
        uint8_t         byte;
};

struct pch_trdata_irq_handler {
        uint32_t        handler;
        int16_t         order_priority; // -1 for exclusive
        uint8_t         irqnum;
};

struct pch_trdata_cu_register {
        uint16_t                num_devices;
        pch_cuaddr_t            cuaddr;
};

struct pch_trdata_id_irq {
        uint8_t                 id;
        pch_dma_irq_index_t     dmairqix;
        uint8_t                 tx_state;
        uint8_t                 rx_state;
};

struct pch_trdata_dev {
        pch_cuaddr_t    cuaddr;
        pch_unit_addr_t ua;
};

struct pch_trdata_dev_byte {
        pch_cuaddr_t    cuaddr;
        pch_unit_addr_t ua;
        uint8_t         byte;
};

struct pch_trdata_word_dev {
        uint32_t        word;
        pch_cuaddr_t    cuaddr;
        pch_unit_addr_t ua;
};

struct pch_trdata_word_sid_byte {
        uint32_t        word;
        pch_sid_t       sid;
        uint8_t         byte;
};

struct pch_trdata_word_byte {
        uint32_t        word;
        uint8_t         byte;
};

struct pch_trdata_word_sid {
        uint32_t        word;
        pch_sid_t       sid;
};

struct pch_trdata_sid_byte {
        pch_sid_t       sid;
        uint8_t         byte;
};

struct pch_trdata_ccw_addr_sid {
        pch_ccw_t       ccw;
        uint32_t        addr;
        pch_sid_t       sid;
};

struct pch_trdata_intcode_scsw {
        pch_intcode_t   intcode;
        pch_scsw_t      scsw;
};

struct pch_trdata_scsw_sid_cc {
        pch_scsw_t      scsw;
        pch_sid_t       sid;
        uint8_t         cc;
};

struct pch_trdata_dma_init {
        uint32_t                addr;
        uint32_t                ctrl;
        uint8_t                 id;
        pch_dmaid_t             dmaid;
        pch_dma_irq_index_t     dmairqix;
        uint8_t                 core_num;
};

struct pch_trdata_chp_alloc {
        pch_sid_t       first_sid;
        uint16_t        num_devices;
        pch_chpid_t     chpid;
};

struct pch_trdata_irqnum_opt {
        int16_t         irqnum_opt;
};

struct pch_trdata_address_change {
        uint32_t        old_addr;
        uint32_t        new_addr;
};

struct pch_trdata_func_irq {
        int16_t         ua_opt;
        pch_chpid_t     chpid;
        uint8_t         tx_active;
};

struct pch_trdata_cus_init_mem_channel {
        pch_cuaddr_t    cuaddr;
        pch_dmaid_t     txdmaid;
        pch_dmaid_t     rxdmaid;
};

struct pch_trdata_cus_tx_complete {
        int16_t         uaopt;
        pch_cuaddr_t    cuaddr;
        uint8_t         txpstate;
```

```
};

struct pch_trdata_cus_call_callback {
        pch_cuaddr_t    cuaddr;
        pch_unit_addr_t ua;
        uint8_t         cbindex;
};

struct pch_trdata_dmachan {
        pch_dmaid_t     dmaid;
};

struct pch_trdata_dmachan_memstate {
        pch_dmaid_t     dmaid;
        uint8_t         state;
};

struct pch_trdata_dmachan_segment {
        uint32_t        addr;
        uint32_t        count;
        pch_dmaid_t     dmaid;
};

struct pch_trdata_dmachan_segment_memstate {
        uint32_t        addr;
        uint32_t        count;
        pch_dmaid_t     dmaid;
        uint8_t         state;
};

#endif
```

13.21 txsm_state.h

```
/*
 * Copyright (c) 2025 Malcolm Beattie
 * SPDX-License-Identifier: MIT
 */

#ifndef _PCH_API_TXSM_STATE_H
#define _PCH_API_TXSM_STATE_H

typedef enum __attribute__((packed)) pch_txsm_state {
        PCH_TXSM_IDLE = 0,
        PCH_TXSM_PENDING,
        PCH_TXSM_SENDING
} pch_txsm_state_t;

typedef enum __attribute__((packed)) pch_txsm_run_result {
        PCH_TXSM_NOOP = 0,
        PCH_TXSM_ACTED,
        PCH_TXSM_FINISHED
} pch_txsm_run_result_t;

#endif
```

13.22 chop.h

```
/*
 * Copyright (c) 2025 Malcolm Beattie
 * SPDX-License-Identifier: MIT
 */

#ifndef _PCH_PROTO_CHOP_H
#define _PCH_PROTO_CHOP_H

// proto_chop_t represents a channel operation in a packet sent
// between CSS and CU in either direction.
// It is 8 bits with the top 4 as flag bits (with only 3 currently
// in use) and the bottom 4 as the operation command itself.
// The meaning of the flag bits depends on the operation command.
typedef uint8_t proto_chop_t;

typedef enum __attribute__((packed)) proto_chop_cmd {
        PROTO_CHOP_START                = 0,
        PROTO_CHOP_ROOM                 = 1,
        PROTO_CHOP_DATA                 = 2,
        PROTO_CHOP_UPDATE_STATUS        = 3,
        PROTO_CHOP_REQUEST_READ         = 4,
        PROTO_CHOP_HALT                 = 5
} proto_chop_cmd_t;
static_assert(sizeof(proto_chop_cmd_t) == 1, "proto_chop_cmd_t must be 1 byte");

typedef uint8_t proto_chop_flags_t;

// PROTO_CHOP_FLAG_SKIP is valid in CSS -> CU Room, Data and Start
// and in CU -> CSS Data
#define PROTO_CHOP_FLAG_SKIP    0x80

// PROTO_CHOP_FLAG_END is valid in CSS <-> CU Data
#define PROTO_CHOP_FLAG_END     0x40

// PROTO_CHOP_FLAG_STOP is valid in CSS -> CU Data
#define PROTO_CHOP_FLAG_STOP    0x20

// PROTO_CHOP_FLAG_RESPONSE_REQUIRED is valid in CU -> CSS Data
#define PROTO_CHOP_FLAG_RESPONSE_REQUIRED     0x20

static inline proto_chop_flags_t proto_chop_flags(proto_chop_t c) {
        return (proto_chop_flags_t)(c & 0xf0);
}

static inline proto_chop_cmd_t proto_chop_cmd(proto_chop_t c) {
        return (proto_chop_cmd_t)(c & 0x0f);
}

#endif
```

13.23 base/proto/packet.h File Reference

```
#include <assert.h>
#include "chop.h"
#include "payload.h"
#include "picochan/ids.h"
#include "picochan/bsize.h"
```

Data Structures

- struct proto_packet

  a 4-byte command packet sent on a channel between CSS and CU or vice versa

Typedefs

- typedef struct proto_packet proto_packet_t

  a 4-byte command packet sent on a channel between CSS and CU or vice versa

Functions

- static proto_payload_t **proto_get_payload** (proto_packet_t p)
- static uint32_t **proto_packet_as_word** (proto_packet_t p)
- static uint16_t **proto_get_count** (proto_packet_t p)
- static uint16_t **proto_decode_esize_payload** (proto_packet_t p)
- static proto_packet_t **proto_make_packet** (proto_chop_t chop, pch_unit_addr_t ua, proto_payload_t payload)
- static proto_packet_t **proto_make_count_packet** (proto_chop_t chop, pch_unit_addr_t ua, uint16_t count)
- static proto_packet_t **proto_make_esize_packet** (proto_chop_t chop, pch_unit_addr_t ua, uint8_t p0, pch_bsize_t esize)

13.24 packet.h

Go to the documentation of this file.

```
/*
 * Copyright (c) 2025 Malcolm Beattie
 * SPDX-License-Identifier: MIT
 */

#ifndef _PCH_PROTO_PACKET_H
#define _PCH_PROTO_PACKET_H

#include <assert.h>
#include "chop.h"
#include "payload.h"
#include "picochan/ids.h"
#include "picochan/bsize.h"


typedef struct __attribute__((aligned(4))) proto_packet {
        proto_chop_t    chop;
        pch_unit_addr_t unit_addr;
        uint8_t         p0;
        uint8_t         p1;
} proto_packet_t;

static_assert(sizeof(proto_packet_t) == 4, "proto_packet_t must be 4 bytes");

static inline proto_payload_t proto_get_payload(proto_packet_t p) {
        return ((proto_payload_t){p.p0, p.p1});
}

static inline uint32_t proto_packet_as_word(proto_packet_t p) {
        return *(uint32_t *)&p;
}

// proto_get_count parses the payload of the packet as a 2-byte
// big-endian value
static inline uint16_t proto_get_count(proto_packet_t p) {
        return ((uint16_t)p.p0 « 8) + (uint16_t)p.p1; // big endian
}

// proto_decode_esize_payload decodes the second byte of the payload
// of the packet (p.p1), treating it as a pch_decode_bsize and using
// pch_bsize_decode_raw to return the resulting count.
static inline uint16_t proto_decode_esize_payload(proto_packet_t p) {
        return pch_bsize_decode_raw(p.p1);
}

static inline proto_packet_t proto_make_packet(proto_chop_t chop, pch_unit_addr_t ua, proto_payload_t
payload) {
        return ((proto_packet_t){
                .chop = chop,
                .unit_addr = ua,
                .p0 = payload.p0,
                .p1 = payload.p1
        });
}

static inline proto_packet_t proto_make_count_packet(proto_chop_t chop, pch_unit_addr_t ua, uint16_t
count) {
        return ((proto_packet_t){
                .chop = chop,
                .unit_addr = ua,
                .p0 = count / 256, // big-endian: high byte
                .p1 = count % 256  // big-endian: low byte
        });
}

static inline proto_packet_t proto_make_esize_packet(proto_chop_t chop, pch_unit_addr_t ua, uint8_t
p0, pch_bsize_t esize) {
        return ((proto_packet_t){
                .chop = chop,
                .unit_addr = ua,
                .p0 = p0,
                .p1 = pch_bsize_unwrap(esize)
        });
}

#endif
```

13.25 payload.h

```
/*
 * Copyright (c) 2025 Malcolm Beattie
 * SPDX-License-Identifier: MIT
 */

#ifndef _PCH_PROTO_PAYLOAD_H
#define _PCH_PROTO_PAYLOAD_H

#include "picochan/bsize.h"

// proto_payload_t is a 2-byte channel operation payload. It can be a
// count, a pair of bytes "ccwcmd", "esize" for START-like or a
// byte of device status followed by an (optional) advertised write
// window esize for a device status update operation. A payload of a
// uint16_t is decoded as big endian.
typedef struct proto_payload {
        uint8_t p0;
        uint8_t p1;
} proto_payload_t;

// proto_parse_count_payload parses the payload as a 2-byte
// big-endian value
static inline uint16_t proto_parse_count_payload(proto_payload_t p) {
        return ((uint16_t)p.p0 « 8) + (uint16_t)p.p1; // big endian
}

static inline uint8_t proto_parse_devstatus_payload_devs(proto_payload_t p) {
        return p.p0;
}

static inline pch_bsize_t proto_parse_devstatus_payload_esize(proto_payload_t p) {
        return pch_bsize_wrap(p.p1);
}

struct proto_parsed_devstatus_payload {
        uint16_t count;
        uint8_t devs;
};

static inline proto_payload_t proto_make_count_payload(uint16_t count) {
        // big-endian encoding
        return ((proto_payload_t){
                .p0 = (uint8_t)(count / 256),
                .p1 = (uint8_t)(count % 256)
        });
}

proto_payload_t proto_make_devstatus_payload(uint8_t devs, pch_bsize_t esize);
proto_payload_t proto_make_start_payload(uint8_t ccwcmd, pch_bsize_t esize);
struct proto_parsed_devstatus_payload proto_parse_devstatus_payload(proto_payload_t p);

#endif
```

13.26 bufferset.h

```
/*
 * Copyright (c) 2025 Malcolm Beattie
 * SPDX-License-Identifier: MIT
 */

#ifndef _PCH_TRC_BUFFERSET_H
#define _PCH_TRC_BUFFERSET_H

#include "hardware/irq.h"
#include "assert.h"
#include "trace_lock.h"

// pch_trc_init_bufferset initialises the bufferset by filling in
// the num_buffers, buffer_size and magic fields and zeroing out the
// other fields
void pch_trc_init_bufferset(pch_trc_bufferset_t *bs, uint32_t magic);

// pch_trc_init_buffer initialises buffer index n to buf.
static inline void pch_trc_init_buffer(pch_trc_bufferset_t *bs, uint n, void *buf) {
        valid_params_if(PCH_TRC, n < PCH_TRC_NUM_BUFFERS);
        valid_params_if(PCH_TRC, ((uint32_t)buf & 0x3) == 0);
        bs->buffers[n] = buf;
}

// pch_trc_init_contiguous_buffers initialises all buffers of bs
// to be pointers to the PCH_TRC_NUM_BUFFERS consecutive
// PCH_TRC_BUFFER_SIZE-bytes-sized buffers in the contiguous space
// in buf. buf must therefore be a pointer to at least
// PCH_TRC_NUM_BUFFERS*PCH_TRC_BUFFER_SIZE available bytes.
void pch_trc_init_all_buffers(pch_trc_bufferset_t *bs, void *buf);

// pch_trc_switch_to_next_buffer_unsafe is for internal use.
// The external API is pch_trc_switch_to_next_buffer which takes
// the trace_lock and then calls this with a 0 for position.
// Internally, this is used when allocating a slot for a new trace
// record (which has already taken trace_lock) and in that situation
// it is often called with a non-zero pos following the
// newly-allocated trace record.
static inline unsigned char *pch_trc_switch_to_next_buffer_unsafe(pch_trc_bufferset_t *bs, uint32_t
pos) {
        bs->current_buffer_num = (bs->current_buffer_num + 1) % PCH_TRC_NUM_BUFFERS;
        bs->current_buffer_pos = pos;
        if (bs->irqnum > -1)
                irq_set_pending((irq_num_t)(bs->irqnum));

        return bs->buffers[bs->current_buffer_num];
}

// pch_trc_switch_to_next_buffer switches to the next trace buffer in
// the bufferset. If bs->irqnum is non-negative, that IRQ is raised.
// When the IRQ is raised, current_buffer_num has already been
// incremented (modulo PCH_TRC_NUM_BUFFERS) and a trace record may be
// in the process of writing to the new buffer. The IRQ handler will
// typically want to start copying or sending the contents of
// bs->buffers[bs->current_buffer_num-1] elsewhere and aim for
// completion before the trace records fill remaining buffers and
// wrap back around to overwrite that buffer.
static inline unsigned char *pch_trc_switch_to_next_buffer(pch_trc_bufferset_t *bs) {
        uint32_t status = trace_lock();
        unsigned char *rec = pch_trc_switch_to_next_buffer_unsafe(bs, 0);
        trace_unlock(status);
        return rec;
}

#endif
```

13.27 trace.h

```
/*
 * Copyright (c) 2025 Malcolm Beattie
 * SPDX-License-Identifier: MIT
 */

#ifndef _PCH_TRC_TRACE_H
#define _PCH_TRC_TRACE_H

// PICO_CONFIG: PARAM_ASSERTIONS_ENABLED_PCH_TRC, Enable/disable assertions in the pch_trc module,
type=bool, default=0, group=pch_trc
#ifndef PARAM_ASSERTIONS_ENABLED_PCH_TRC
#define PARAM_ASSERTIONS_ENABLED_PCH_TRC 0
#endif

#include <stddef.h>
#include <string.h>
#include "picochan/trc.h"
#include "picochan/trc_records.h"
#include "bufferset.h"

static_assert(sizeof(pch_trc_timestamp_t) == 6,
        "pch_trc_timestamp_t must be 6 bytes");

static_assert(sizeof(pch_trc_header_t) == 8,
        "pch_trc_header_t must be 8 bytes");

void *pch_trc_write_uncond(pch_trc_bufferset_t *bs, pch_trc_record_type_t rt, uint8_t data_size);

// pch_trc_write allocates and writes the header of a trace record
// with the current timestamp and record type rt and returns a pointer
// to the location where data_size bytes of associated trace should be
// written. It returns NULL (without writing any header or taking any
// other action) if no trace record should be written. This will be
// the case if tracing was disabled globally at compile time
// (PCH_CONFIG_ENABLE_TRACE was not defined or defined as 0) or if
// tracing has been disabled (perhaps temporarily) at runtime by
// setting pch_trc_enable to false or if the cond function argument is
// false.
static inline void *pch_trc_write(pch_trc_bufferset_t *bs, bool cond, pch_trc_record_type_t rt,
uint8_t data_size) {
```

```
#if PCH_CONFIG_ENABLE_TRACE
        if (!bs->enable // per-bufferset runtime tracing flag not enabled
                || !cond) // per-function-call condition flag not enabled
                return NULL;

        return pch_trc_write_uncond(bs, rt, data_size);
#else
        (void)bs;
        (void)cond;
        (void)rt;
        (void)data_size;
#endif
        return NULL;
}

#define PCH_TRC_WRITE(bs, cond, rt, data) do { \
                size_t __data_size = sizeof (data); \
                void *__rec = pch_trc_write((bs), (cond), (rt), __data_size); \
                if (__rec) \
                        memcpy(__rec, &(data), __data_size); \
        } while (0)

bool pch_trc_set_enable(pch_trc_bufferset_t *bs, bool enable);

#endif
```

13.28 trace_lock.h

```
/*
 * Copyright (c) 2025 Malcolm Beattie
 * SPDX-License-Identifier: MIT
 */

#ifndef _PCH_TRC_TRACE_LOCK_H
#define _PCH_TRC_TRACE_LOCK_H

#include "hardware/sync.h"

static inline uint32_t trace_lock(void) {
        return save_and_disable_interrupts();
}

static inline void trace_unlock(uint32_t status) {
        restore_interrupts(status);
}

#endif
```

13.29 txsm.h

```
/*
 * Copyright (c) 2025 Malcolm Beattie
 * SPDX-License-Identifier: MIT
 */

#ifndef _PCH_TXSM_PENDING_XFER_H
#define _PCH_TXSM_PENDING_XFER_H

// PICO_CONFIG: PARAM_ASSERTIONS_ENABLED_PCH_TXSM, Enable/disable assertions in the pch_txsm module,
type=bool, default=0, group=pch_txsm
#ifndef PARAM_ASSERTIONS_ENABLED_PCH_TXSM
#define PARAM_ASSERTIONS_ENABLED_PCH_TXSM 0
#endif

#include <stdint.h>
#include "picochan/dmachan.h"
#include "picochan/txsm_state.h"

// txsm provides a state machine that manages using a
// dmachan_tx_channel to transmit a data buffer, driven by
// tx completion handler calls.
//
// pch_txsm_t represents a pending data transfer.
//        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
//        |                |     flags      |           count               |
//        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
//        |                             addr                               |
//        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
typedef struct pch_txsm {
        pch_txsm_state_t        state;
        uint16_t                count;
        uint32_t                addr;
} pch_txsm_t;

// pch_txsm_busy returns whether px is non-Idle (i.e. it returns true
// if and only if px is in state Pending or Sending).
static inline bool pch_txsm_busy(pch_txsm_t *px) {
        return px->state != PCH_TXSM_IDLE;
}

// Reset resets the state to Idle but does not change any
// owner, addr or count set by SetPending
static inline void pch_txsm_reset(pch_txsm_t *px) {
        px->state = PCH_TXSM_IDLE;
}

// pch_txsm_set_pending stashes (addr, count) in px and moves its
// state from Idle to Pending. It panics if px is Busy.
static inline void pch_txsm_set_pending(pch_txsm_t *px, uint32_t addr, uint16_t count) {
        valid_params_if(PCH_TXSM, px->state == PCH_TXSM_IDLE);

        px->state = PCH_TXSM_PENDING;
        px->addr = addr;
        px->count = count;
}

enum pch_txsm_run_result pch_txsm_run(pch_txsm_t *px, dmachan_tx_channel_t *txch);

#endif
```

13.30 ccw_fetch.h

```
/*
 * Copyright (c) 2025 Malcolm Beattie
 * SPDX-License-Identifier: MIT
 */

#ifndef _PCH_CSS_CCW_FETCH_H
#define _PCH_CSS_CCW_FETCH_H

void fetch_chain_data_ccw(pch_schib_t *schib);

uint8_t fetch_first_command_ccw(pch_schib_t *schib);

uint8_t fetch_resume_ccw(pch_schib_t *schib);

uint8_t fetch_chain_ccw(pch_schib_t *schib);

uint8_t fetch_chain_command_ccw(pch_schib_t *schib);

#endif
```

13.31 css/channel.h File Reference

```
#include <stdint.h>
#include <stdbool.h>
#include "css_internal.h"
#include "txsm/txsm.h"
#include "proto/packet.h"
```

Data Structures

- struct ua_slist
- struct pch_chp

  pch_chp_t is the CSS-side representation of a channel path to a control unit.

Macros

- #define **EMPTY_UA_DLIST** ((ua_dlist_t)-1)

Typedefs

- typedef int16_t **ua_dlist_t**
- typedef struct ua_slist **ua_slist_t**
- typedef struct pch_chp pch_chp_t

  pch_chp_t is the CSS-side representation of a channel path to a control unit.

Functions

- static ua_dlist_t **make_ua_dlist** (void)
- static int16_t **peek_ua_dlist** (ua_dlist_t ∗l)
- void **push_ua_dlist_unsafe** (ua_dlist_t ∗l, pch_chp_t ∗chp, pch_schib_t ∗schib)
- static void **push_ua_dlist** (ua_dlist_t ∗l, pch_chp_t ∗chp, pch_schib_t ∗schib)
- pch_schib_t ∗ **remove_from_ua_dlist_unsafe** (ua_dlist_t ∗l, pch_chp_t ∗chp, pch_unit_addr_t ua)
- static pch_schib_t ∗ **remove_from_ua_dlist** (ua_dlist_t ∗l, pch_chp_t ∗chp, pch_unit_addr_t ua)
- static pch_schib_t ∗ **pop_ua_dlist_unsafe** (ua_dlist_t ∗l, pch_chp_t ∗chp)
- static pch_schib_t ∗ **pop_ua_dlist** (ua_dlist_t ∗l, pch_chp_t ∗chp)
- static ua_slist_t **make_ua_slist** (void)
- static void **reset_ua_slist** (ua_slist_t ∗l)
- pch_schib_t ∗ **pop_ua_slist_unsafe** (ua_slist_t ∗l, pch_chp_t ∗chp)
- static pch_schib_t ∗ **pop_ua_slist** (ua_slist_t ∗l, pch_chp_t ∗chp)
- bool **push_ua_slist_unsafe** (ua_slist_t ∗l, pch_chp_t ∗chp, pch_sid_t sid)
- static bool **push_ua_slist** (ua_slist_t ∗l, pch_chp_t ∗chp, pch_sid_t sid)
- static pch_schib_t ∗ **pop_ua_response_slist** (pch_chp_t ∗chp)
- static void **push_ua_response_slist** (pch_chp_t ∗chp, pch_sid_t sid)
- static proto_packet_t **get_rx_packet** (pch_chp_t ∗chp)
- static proto_packet_t **get_tx_packet** (pch_chp_t ∗chp)
- void **send_tx_packet** (pch_chp_t ∗chp, proto_packet_t p)

13.32 channel.h

Go to the documentation of this file.

```
/*
 * Copyright (c) 2025 Malcolm Beattie
 * SPDX-License-Identifier: MIT
 */

#ifndef _PCH_CSS_CHANNEL_H
#define _PCH_CSS_CHANNEL_H

#include <stdint.h>
#include <stdbool.h>
#include "css_internal.h"
#include "txsm/txsm.h"
#include "proto/packet.h"


// ua_dlist_t is the head of a circular double-linked list of schibs
// which all belong to the same channel, linked by the prevua/nextua
// fields of schib.mda. It is the unit_addr_t of the head of the list
// or else -1 if the list is empty.
typedef int16_t ua_dlist_t;

// ua_slist_t is the head and tail of a single-linked list of schibs
// which all belong to the same channel, linked by the nextua field of
// schib.mda. It contains the unit_addr_t of the head and tail of
// the list or else both fields are -1 if the list is empty.
typedef struct ua_slist {
        int16_t head;
        int16_t tail;
} ua_slist_t;

typedef struct __aligned(4) pch_chp {
        dmachan_tx_channel_t    tx_channel;
        dmachan_rx_channel_t    rx_channel;
        pch_txsm_t              tx_pending;
        pch_sid_t               first_sid;
        uint16_t                num_devices; // [0, 256]
        // rx_data_for_ua: rx dma is active writing to CCW for this ua
        int16_t                 rx_data_for_ua;
        // rx_data_end_ds: if non-zero then, when rx data complete,
        // treat as an immediate implicit device status for update_status
        uint8_t                 rx_data_end_ds;
        // TODO combine following bools into a uint8_t flags
        // rx_response_required: when rx data complete, peer wants response
        bool                    rx_response_required;
        bool                    traced;
        bool                    claimed;
        bool                    allocated;
        bool                    configured;
        bool                    started;
        // tx_active: tx dma is active
        bool                    tx_active;
        // ua_func_dlist: links via schib.prevua and .nextua
        ua_dlist_t              ua_func_dlist;
        // ua_response_slist: link via schib.nextua
        ua_slist_t              ua_response_slist;
} pch_chp_t;

//
// ua_dlist
//
#define EMPTY_UA_DLIST ((ua_dlist_t)-1)

static inline ua_dlist_t make_ua_dlist(void) {
        return EMPTY_UA_DLIST;
}

static inline int16_t peek_ua_dlist(ua_dlist_t *l) {
        return (int16_t)*l;
}

void push_ua_dlist_unsafe(ua_dlist_t *l, pch_chp_t *chp, pch_schib_t *schib);

static inline void push_ua_dlist(ua_dlist_t *l, pch_chp_t *chp, pch_schib_t *schib) {
        uint32_t status = schibs_lock();
        push_ua_dlist_unsafe(l, chp, schib);
        schibs_unlock(status);
}

pch_schib_t *remove_from_ua_dlist_unsafe(ua_dlist_t *l, pch_chp_t *chp, pch_unit_addr_t ua);

static inline pch_schib_t *remove_from_ua_dlist(ua_dlist_t *l, pch_chp_t *chp, pch_unit_addr_t ua) {
        uint32_t status = schibs_lock();
        pch_schib_t *schib = remove_from_ua_dlist_unsafe(l, chp, ua);
        schibs_unlock(status);
        return schib;
}

static inline pch_schib_t *pop_ua_dlist_unsafe(ua_dlist_t *l, pch_chp_t *chp) {
        if (*l == -1)
                return NULL;

        return remove_from_ua_dlist_unsafe(l, chp, (pch_unit_addr_t)*l);
}

static inline pch_schib_t *pop_ua_dlist(ua_dlist_t *l, pch_chp_t *chp) {
        uint32_t status = schibs_lock();
        pch_schib_t *schib = pop_ua_dlist_unsafe(l, chp);
        schibs_unlock(status);
        return schib;
}

//
// ua_slist
//

static inline ua_slist_t make_ua_slist(void) {
        return ((ua_slist_t){-1, -1});
}

static inline void reset_ua_slist(ua_slist_t *l) {
        l->head = -1;
        l->tail = -1;
}

pch_schib_t *pop_ua_slist_unsafe(ua_slist_t *l, pch_chp_t *chp);

static inline pch_schib_t *pop_ua_slist(ua_slist_t *l, pch_chp_t *chp) {
        uint32_t status = schibs_lock();
        pch_schib_t *schib = pop_ua_slist_unsafe(l, chp);
        schibs_unlock(status);
        return schib;
}

bool push_ua_slist_unsafe(ua_slist_t *l, pch_chp_t *chp, pch_sid_t sid);

static inline bool push_ua_slist(ua_slist_t *l, pch_chp_t *chp, pch_sid_t sid) {
        uint32_t status = schibs_lock();
        bool was_empty = push_ua_slist_unsafe(l, chp, sid);
        schibs_unlock(status);
        return was_empty;
}

// popping from and pushing to the channel ua_response_slist of schibs
// with response packets pending to be sent to their CUs
static inline pch_schib_t *pop_ua_response_slist(pch_chp_t *chp) {
        return pop_ua_slist(&chp->ua_response_slist, chp);
}

static inline void push_ua_response_slist(pch_chp_t *chp, pch_sid_t sid) {
        push_ua_slist(&chp->ua_response_slist, chp, sid);
}

//
// getting packets to/from the channel command buffers
//
static inline proto_packet_t get_rx_packet(pch_chp_t *chp) {
        // chp.rx_channel is a dmachan_rx_channel_t which is
        // __aligned(4) and cmd is the first member of rx_channel
        // so is 4-byte aligned. proto_packet_t is 4-bytes and also
        // __aligned(4) (and needing no more than 4-byte alignment)
        // but omitting the __builtin_assume_aligned below causes
        // gcc 14.1.0 to produce error
        // error: cast increases required alignment of target type
        // [-Werror=cast-align]
        proto_packet_t *pp = (proto_packet_t *)
                __builtin_assume_aligned(&chp->rx_channel.link.cmd, 4);
        return *pp;
}

static inline proto_packet_t get_tx_packet(pch_chp_t *chp) {
        // chp.tx_channel is a dmachan_tx_channel_t which is the
        // first member of chp which is a pch_chp_t which is
        // __aligned(4) and cmd is the first member of tx_channel
        // so is 4-byte aligned. proto_packet_t is 4-bytes and also
        // __aligned(4) (and needing no more than 4-byte alignment)
        // but omitting the __builtin_assume_aligned below causes
        // gcc 14.1.0 to produce error
        // error: cast increases required alignment of target type
        // [-Werror=cast-align]
        proto_packet_t *pp = (proto_packet_t *)
                __builtin_assume_aligned(&chp->tx_channel.link.cmd, 4);
        return *pp;
}

void send_tx_packet(pch_chp_t *chp, proto_packet_t p);

#endif
```

13.33 css/css_internal.h File Reference

```
#include <stdint.h>
#include <assert.h>
#include "hardware/sync.h"
#include "hardware/irq.h"
#include "picochan/css.h"
#include "schibs_lock.h"
#include "schib_internal.h"
#include "schib_dlist.h"
```

```
#include "channel.h"
#include "picochan/dmachan.h"
#include "trc/trace.h"
```

Data Structures

- struct css

  struct css is a channel subsystem (CSS)

Macros

- #define **PARAM_ASSERTIONS_ENABLED_PCH_CSS** 0

Functions

- static pch_schib_t ∗ **get_schib** (pch_sid_t sid)
- static pch_chp_t ∗ **pch_get_chp** (pch_chpid_t chpid)
- static pch_chpid_t **pch_get_chpid** (pch_chp_t ∗chp)
- static schib_dlist_t ∗ **get_isc_dlist** (uint8_t iscnum)
- static pch_schib_t ∗ **get_schib_by_chp** (pch_chp_t ∗chp, pch_unit_addr_t ua)
- static pch_sid_t **get_sid** (pch_schib_t ∗schib)
- static bool **css_is_started** (void)
- static void **reset_subchannel_to_idle** (pch_schib_t ∗schib)
- static void **css_clear_pending_subchannel** (pch_schib_t ∗schib)
- void __isr **pch_css_dma_irq_handler** (void)
- void **suspend_or_send_start_packet** (pch_chp_t ∗chp, pch_schib_t ∗schib, uint8_t ccwcmd)
- void **do_command_chain_and_send_start** (pch_chp_t ∗chp, pch_schib_t ∗schib)
- void **send_command_with_data** (pch_chp_t ∗chp, pch_schib_t ∗schib, proto_packet_t p, uint16_t count)
- void **send_update_room** (pch_chp_t ∗chp, pch_schib_t ∗schib)
- void **send_data_response** (pch_chp_t ∗chp, pch_schib_t ∗schib)
- void **css_handle_rx_complete** (pch_chp_t ∗chp)
- void **css_handle_tx_complete** (pch_chp_t ∗chp)
- pch_schib_t ∗ **pop_pending_schib_from_isc** (uint8_t iscnum)
- void **remove_from_isc_dlist** (uint8_t iscnum, pch_sid_t sid)
- void **push_to_isc_dlist** (pch_schib_t ∗schib)
- pch_schib_t ∗ **pop_pending_schib** (void)
- void **css_notify** (pch_schib_t ∗schib, uint8_t devs)
- static pch_intcode_t **css_make_intcode** (pch_schib_t ∗schib)

Variables

- struct css CSS

13.33.1 Variable Documentation

13.33.1.1 CSS

```
struct css CSS  [extern]
```

CSS is a channel subsystem. It is intended to be a singleton and is just a convenience for gathering together the global variables associated with the CSS.

13.34 css_internal.h

Go to the documentation of this file.

```
/*
 * Copyright (c) 2025 Malcolm Beattie
 * SPDX-License-Identifier: MIT
 */

#ifndef _PCH_CSS_CSS_INTERNAL_H
#define _PCH_CSS_CSS_INTERNAL_H

// PICO_CONFIG: PARAM_ASSERTIONS_ENABLED_PCH_CSS, Enable/disable assertions in the pch_css module,
type=bool, default=0, group=pch_css
#ifndef PARAM_ASSERTIONS_ENABLED_PCH_CSS
#define PARAM_ASSERTIONS_ENABLED_PCH_CSS 0
#endif

#include <stdint.h>
#include <assert.h>
#include "hardware/sync.h"
#include "hardware/irq.h"
#include "picochan/css.h"
#include "schibs_lock.h"
#include "schib_internal.h"
#include "schib_dlist.h"
#include "channel.h"
#include "picochan/dmachan.h"
#include "trc/trace.h"


struct css {
        schib_dlist_t   isc_dlists[PCH_NUM_ISCS]; // indexed by ISC
        io_callback_t   io_callback;
        int16_t         io_irqnum;
        int16_t         func_irqnum;
        uint8_t         isc_enable_mask;
        uint8_t         isc_status_mask;
        pch_dma_irq_index_t dmairqix;
        int8_t          core_num;
        pch_sid_t       next_sid;
        pch_trc_bufferset_t trace_bs;
        pch_chp_t       chps[PCH_NUM_CHANNELS];
        pch_schib_t     schibs[PCH_NUM_SCHIBS];
};

extern struct css CSS;

static inline pch_schib_t *get_schib(pch_sid_t sid) {
        return &CSS.schibs[sid];
}

static inline pch_chp_t *pch_get_chp(pch_chpid_t chpid) {
        return &CSS.chps[chpid];
}

static inline pch_chpid_t pch_get_chpid(pch_chp_t *chp) {
        int32_t n = chp - &CSS.chps[0];
        assert(n >= 0 && n < PCH_NUM_CHANNELS);
        return (pch_chpid_t)n;
}

static inline schib_dlist_t *get_isc_dlist(uint8_t iscnum) {
        valid_params_if(PCH_CSS, iscnum < PCH_NUM_ISCS);
        return &CSS.isc_dlists[iscnum];
}

static inline pch_schib_t *get_schib_by_chp(pch_chp_t *chp, pch_unit_addr_t ua) {
        valid_params_if(PCH_CSS, (uint16_t)ua < chp->num_devices);
        return get_schib(chp->first_sid + (pch_sid_t)ua);
}

static inline pch_sid_t get_sid(pch_schib_t *schib) {
        // if we definitely decide to include intparm in the PMCW then
        // the schib size is 32 bytes so we could easily check the low
        // 5 bits are all zero as a valid_params_if check too.
        valid_params_if(PCH_CSS,
                schib >= &CSS.schibs[0]
                && schib < &CSS.schibs[PCH_NUM_SCHIBS]);

        return schib - CSS.schibs;
}

static inline bool css_is_started(void) {
        return CSS.dmairqix >= 0;
}
```

```

static inline void reset_subchannel_to_idle(pch_schib_t *schib) {
        const uint16_t mask = PCH_FC_START|PCH_FC_HALT|PCH_FC_CLEAR
                | PCH_AC_RESUME_PENDING|PCH_AC_START_PENDING
                | PCH_AC_HALT_PENDING|PCH_AC_CLEAR_PENDING
                | PCH_AC_SUSPENDED | PCH_SC_PENDING;

        schib->scsw.ctrl_flags &= ~mask;
}

static inline void css_clear_pending_subchannel(pch_schib_t *schib) {
        valid_params_if(PCH_CSS, schib_is_status_pending(schib));

        if (schib->scsw.ctrl_flags & PCH_SC_INTERMEDIATE) {
                // TODO Don't do clearing unless various flag
                // combinations are set.
        }

        reset_subchannel_to_idle(schib);
}

void __isr pch_css_dma_irq_handler(void);

void suspend_or_send_start_packet(pch_chp_t *chp, pch_schib_t *schib, uint8_t ccwcmd);
void do_command_chain_and_send_start(pch_chp_t *chp, pch_schib_t *schib);
void send_command_with_data(pch_chp_t *chp, pch_schib_t *schib, proto_packet_t p, uint16_t count);
void send_update_room(pch_chp_t *chp, pch_schib_t *schib);
void send_data_response(pch_chp_t *chp, pch_schib_t *schib);
void css_handle_rx_complete(pch_chp_t *chp);
void css_handle_tx_complete(pch_chp_t *chp);

//
// isc dlists
//

pch_schib_t *pop_pending_schib_from_isc(uint8_t iscnum);
void remove_from_isc_dlist(uint8_t iscnum, pch_sid_t sid);
void push_to_isc_dlist(pch_schib_t *schib);
pch_schib_t *pop_pending_schib(void);
void css_notify(pch_schib_t *schib, uint8_t devs);

static inline pch_intcode_t css_make_intcode(pch_schib_t *schib) {
        pch_intcode_t ic = { 0 }; // all fields zeroes, including cc
        if (schib) {
                pch_sid_t sid = get_sid(schib);
                ic.intparm = schib->pmcw.intparm;
                ic.sid = sid;
                ic.flags = pch_pmcw_isc(&schib->pmcw);
                ic.cc = 1; // cc=1 means intcode stored [sic]
        }

        return ic;
}

#endif
```

13.35 css_trace.h

```
/*
 * Copyright (c) 2025 Malcolm Beattie
 * SPDX-License-Identifier: MIT
 */

#ifndef _PCH_CSS_CSS_TRACE_H
#define _PCH_CSS_CSS_TRACE_H

#include "css_internal.h"

#include "picochan/trc_records.h"
#include "trc/trace.h"

#define PCH_CSS_TRACE_COND(rt, cond, data) \
        PCH_TRC_WRITE(&CSS.trace_bs, (cond), (rt), (data))

#define PCH_CSS_TRACE(rt, data) PCH_CSS_TRACE_COND((rt), true, (data))

static inline void trace_schib_byte(pch_trc_record_type_t rt, pch_schib_t *schib, uint8_t byte) {
        PCH_CSS_TRACE_COND(rt, schib_is_traced(schib),
                ((struct pch_trdata_sid_byte){get_sid(schib), byte}));
}

static inline void trace_schib_word_byte(pch_trc_record_type_t rt, pch_schib_t *schib, uint32_t word,
uint8_t byte) {
        PCH_CSS_TRACE_COND(rt, schib_is_traced(schib),
                ((struct pch_trdata_word_sid_byte){word, get_sid(schib), byte}));
}

static inline void trace_schib_packet(pch_trc_record_type_t rt, pch_schib_t *schib, proto_packet_t p)
{
        PCH_CSS_TRACE_COND(rt, schib_is_traced(schib),
                ((struct pch_trdata_word_sid){proto_packet_as_word(p), get_sid(schib)}));
}

static inline void trace_schib_ccw(pch_trc_record_type_t rt, pch_schib_t *schib, pch_ccw_t *ccw_addr,
pch_ccw_t ccw) {
        PCH_CSS_TRACE_COND(rt, schib_is_traced(schib),
                ((struct pch_trdata_ccw_addr_sid){
                        .ccw = ccw,
                        .addr = (uint32_t)ccw_addr,
                        .sid = get_sid(schib)
                }));
}

static inline void trace_schib_callback(pch_trc_record_type_t rt, pch_schib_t *schib, pch_intcode_t
*ic) {
        PCH_CSS_TRACE_COND(rt, schib_is_traced(schib),
                ((struct pch_trdata_intcode_scsw){
                        .intcode = *ic,
                        .scsw = schib->scsw,
                }));
}

static inline void trace_schib_scsw_cc(pch_trc_record_type_t rt, pch_schib_t *schib, pch_scsw_t *scsw,
uint8_t cc) {
        PCH_CSS_TRACE_COND(rt, schib_is_traced(schib),
                ((struct pch_trdata_scsw_sid_cc){
                        .scsw = *scsw,
                        .sid = get_sid(schib),
                        .cc = cc
                }));
}

static inline void trace_chp_irq(pch_trc_record_type_t rt, pch_chp_t *chp, pch_dma_irq_index_t
dmairqix, uint8_t tx_irq_state, uint8_t rx_irq_state) {
        PCH_CSS_TRACE_COND(rt,
                chp->traced, ((struct pch_trdata_id_irq){
                        .id = pch_get_chpid(chp),
                        .dmairqix = dmairqix,
                        .tx_state = tx_irq_state « 4
                                | chp->tx_channel.mem_src_state,
                        .rx_state = rx_irq_state « 4
                                | chp->rx_channel.mem_dst_state
                }));
}

#endif
```

13.36 css/include/picochan/css.h File Reference

```
#include "hardware/irq.h"
#include "hardware/dma.h"
#include "hardware/sync.h"
#include "hardware/uart.h"
#include "pico/time.h"
#include "picochan/schib.h"
#include "picochan/ccw.h"
#include "picochan/intcode.h"
#include "picochan/dmachan.h"
```

Macros

- #define PCH_NUM_SCHIBS

  The number of subchannels.

- #define PCH_NUM_CHANNELS

  The number of channels that the CSS can use.
- #define PCH_NUM_ISCS

  The number of interrupt service classes.
- #define **PCH_CSS_BUFFERSET_MAGIC** 0x70437353

Typedefs

- typedef void(∗ **io_callback_t**) (pch_intcode_t, pch_scsw_t)

  A callback function to be invoked when a subchannel becomes status pending.

Functions

- static void ∗ pch_ccw_get_addr (pch_ccw_t ccw)

  Get the addr field of a CCW as a pointer.
- void pch_css_init (void)

  Initialise CSS.
- int8_t **pch_css_get_core_num** (void)
- int8_t **pch_css_get_dma_irq_index** (void)
- int16_t **pch_css_get_func_irq** (void)
- int16_t **pch_css_get_io_irq** (void)
- void **pch_css_set_dma_irq_index** (pch_dma_irq_index_t dmairqix)
- void **pch_css_configure_dma_irq_index_shared** (pch_dma_irq_index_t dmairqix, uint8_t order_priority)
- void **pch_css_configure_dma_irq_index_exclusive** (pch_dma_irq_index_t dmairqix)
- void **pch_css_configure_dma_irq_index_default_shared** (uint8_t order_priority)
- void **pch_css_configure_dma_irq_index_default_exclusive** ()
- void **pch_css_auto_configure_dma_irq_index** ()
- void pch_css_set_func_irq (irq_num_t irqnum)

  Low-level function to set the IRQ number that the CSS uses for application API notification to CSS.
- void **pch_css_configure_func_irq_shared** (irq_num_t irqnum, uint8_t order_priority)
- void **pch_css_configure_func_irq_exclusive** (irq_num_t irqnum)
- void **pch_css_configure_func_irq_unused_shared** (bool required, uint8_t order_priority)
- void **pch_css_configure_func_irq_unused_exclusive** (bool required)
- void **pch_css_auto_configure_func_irq** (bool required)
- void pch_css_set_io_irq (irq_num_t irqnum)

  Low-level function to set the IRQ number that the CSS uses for I/O interrupt notification.
- void **pch_css_configure_io_irq_shared** (irq_num_t irqnum, uint8_t order_priority)
- void **pch_css_configure_io_irq_exclusive** (irq_num_t irqnum)
- void **pch_css_configure_io_irq_unused_shared** (bool required, uint8_t order_priority)
- void **pch_css_configure_io_irq_unused_exclusive** (bool required)
- void **pch_css_auto_configure_io_irq** (bool required)
- io_callback_t pch_css_set_io_callback (io_callback_t io_callback)

  Low-level function to set the I/O callback function that the CSS invokes if its I/O interrupt handler has been set to pch_css_io_irq_handler. pch_css_start(io_callback, isc_mask) with io_callback non-NULL).
- void pch_css_start (io_callback_t io_callback, uint8_t isc_mask)

  Starts CSS operation after setting the io_callback (if non-NULL), configuring and enabling any needed CSS IRQ handlers that have not yet been set and setting the mask of ISCs that trigger I/O interrupts to be isc_mask.
- bool pch_css_set_trace (bool trace)

  Sets whether CSS tracing is enabled.
- bool pch_chp_set_trace (pch_chpid_t chpid, bool trace)

  Sets whether CSS tracing is enabled for channel chpid.

- void __isr **pch_css_func_irq_handler** (void)
- void __isr **pch_css_io_irq_handler** (void)
- void pch_chp_start (pch_chpid_t chpid)

    Starts channel chpid connection to its remote CU.
- void **pch_chp_claim** (pch_chpid_t chpid)

    Mark channel path chpid as claimed. Panics if it is already claimed or allocated.
- int **pch_chp_claim_unused** (bool required)

    Claims the next unclaimed and unallocated channel path and returns its CHPID (a pch_chpid_t cast to int). If no channel path is available, panics if required is true or else returns -1.
- pch_sid_t pch_chp_alloc (pch_chpid_t chpid, uint16_t num_devices)

    Allocates num_devices schibs for use by channel chpid.
- void pch_chp_configure_uartchan (pch_chpid_t chpid, uart_inst_t ∗uart, dma_channel_config ctrl)

    Configure a UART channel.
- void pch_chp_auto_configure_uartchan (pch_chpid_t chpid, uart_inst_t ∗uart, uint baudrate)

    Initialise and configure a hardware UART instance as a channel to the remote CU to which it is connected. Uses a default dma_channel_config control register.
- void pch_chp_configure_memchan (pch_chpid_t chpid, pch_dmaid_t txdmaid, pch_dmaid_t rxdmaid, dmachan_tx_channel_t ∗txpeer)

    Configure a memchan channel.
- void **pch_chp_dma_configure** (pch_chpid_t chpid, dmachan_config_t ∗dc)
- void **pch_chp_set_configured** (pch_chpid_t chpid, bool configured)
- dmachan_tx_channel_t ∗ pch_chp_get_tx_channel (pch_chpid_t chpid)

    Fetch the internal tx side of a channel from CSS to CU.
- dmachan_rx_channel_t ∗ **pch_chp_get_rx_channel** (pch_chpid_t chpid)
- int pch_sch_start (pch_sid_t sid, pch_ccw_t ∗ccw_addr)

    Start a channel program for a subchannel.
- int pch_sch_resume (pch_sid_t sid)

    Resume a channel program for a subchannel.
- int pch_sch_test (pch_sid_t sid, pch_scsw_t ∗scsw)

    Test the status of a subchannel, clearing various status conditions of status is pending.
- int pch_sch_modify (pch_sid_t sid, pch_pmcw_t ∗pmcw)

    Modifies the PMCW field of a subchannel.
- int pch_sch_store (pch_sid_t sid, pch_schib_t ∗out_schib)

    Stores the contents of the schib for subchannel sid to out_schib.
- int pch_sch_cancel (pch_sid_t sid)

    Cancel a channel program that has not yet started.
- int pch_sch_halt (pch_sid_t sid)

    Halt a channel program.
- pch_intcode_t pch_test_pending_interruption (void)

    Test if there is a pending I/O interruption.
- int pch_sch_store_pmcw (pch_sid_t sid, pch_pmcw_t ∗out_pmcw)

    Stores the contents of the PMCW part of the schib for subchannel sid to out_pmcw.
- int pch_sch_store_scsw (pch_sid_t sid, pch_scsw_t ∗out_scsw)

    Stores the contents of the SCSW part of the schib for subchannel sid to out_scsw.
- int pch_sch_modify_intparm (pch_sid_t sid, uint32_t intparm)

    Modifies the intparm field of the PMCW part of the schib for subchannel sid.
- int pch_sch_modify_flags (pch_sid_t sid, uint16_t flags)

    Modifies the flags field of the PMCW part of the schib for subchannel sid.
- int pch_sch_modify_isc (pch_sid_t sid, uint8_t isc)

    Modifies the isc field of the PMCW part of the schib for subchannel sid.
- int pch_sch_modify_enabled (pch_sid_t sid, bool enabled)

Modifies enabled flag of the schib for subchannel sid.
- int pch_sch_modify_traced (pch_sid_t sid, bool traced)

  Modifies traced flag of the schib for subchannel sid.
- **void** (pch_sid_t sid, uint count, uint8_t isc)

  Calls pch_sch_modify_isc() on count subchannels starting from sid, panicking if any call fails.
- void (pch_sid_t sid, uint count, bool enabled)

  Calls pch_sch_modify_enabled() on count subchannels starting from sid, panicking if any call fails.
- int pch_sch_wait (pch_sid_t sid, pch_scsw_t ∗scsw)

  Wait for an I/O interruption condition for subchannel sid.
- int pch_sch_wait_timeout (pch_sid_t sid, pch_scsw_t ∗scsw, absolute_time_t timeout_timestamp)

  Wait for an I/O interruption condition for subchannel sid with a timeout.
- int pch_sch_run_wait (pch_sid_t sid, pch_ccw_t ∗ccw_addr, pch_scsw_t ∗scsw)

  Start a channel program for a subchannel and wait for an I/O interruption condition.
- int pch_sch_run_wait_timeout (pch_sid_t sid, pch_ccw_t ∗ccw_addr, pch_scsw_t ∗scsw, absolute_time_←
t timeout_timestamp)

  Start a channel program for a subchannel and wait for an I/O interruption condition with a timeout.

13.37 css.h

Go to the documentation of this file.

```
/*
 * Copyright (c) 2025 Malcolm Beattie
 * SPDX-License-Identifier: MIT
 */

#ifndef _PCH_API_CSS_H
#define _PCH_API_CSS_H

#include "hardware/irq.h"
#include "hardware/dma.h"
#include "hardware/sync.h"
#include "hardware/uart.h"
#include "pico/time.h"
#include "picochan/schib.h"
#include "picochan/ccw.h"
#include "picochan/intcode.h"
#include "picochan/dmachan.h"


#ifndef PCH_NUM_SCHIBS
#define PCH_NUM_SCHIBS 32
#endif
static_assert(PCH_NUM_SCHIBS >= 1 && PCH_NUM_SCHIBS <= 65536,
        "PCH_NUM_SCHIBS must be between 1 and 65536");

#ifndef PCH_NUM_CHANNELS
#define PCH_NUM_CHANNELS 4
#endif
static_assert(PCH_NUM_CHANNELS >= 1 && PCH_NUM_CHANNELS <= 256,
        "PCH_NUM_CHANNELS must be between 1 and 256");

#ifndef PCH_NUM_ISCS
#define PCH_NUM_ISCS 8
#endif
static_assert(PCH_NUM_ISCS >= 1 && PCH_NUM_ISCS <= 8,
        "PCH_NUM_ISCS must be between 1 and 8");

#define PCH_CSS_BUFFERSET_MAGIC 0x70437353


typedef void(*io_callback_t)(pch_intcode_t, pch_scsw_t);

static inline void *pch_ccw_get_addr(pch_ccw_t ccw) {
        return (void *)ccw.addr;
}

void pch_css_init(void);

// Accessor functions for basic CSS settings
int8_t pch_css_get_core_num(void);
```

```
int8_t pch_css_get_dma_irq_index(void);
int16_t pch_css_get_func_irq(void);
int16_t pch_css_get_io_irq(void);

// A variety of different initialisation functions for configuring
// CSS IRQ numbers and handlers for DMA IRQ index, function IRQ
// and I/O IRQ.

void pch_css_set_dma_irq_index(pch_dma_irq_index_t dmairqix);
void pch_css_configure_dma_irq_index_shared(pch_dma_irq_index_t dmairqix, uint8_t order_priority);
void pch_css_configure_dma_irq_index_exclusive(pch_dma_irq_index_t dmairqix);
void pch_css_configure_dma_irq_index_default_shared(uint8_t order_priority);
void pch_css_configure_dma_irq_index_default_exclusive();
void pch_css_auto_configure_dma_irq_index();

void pch_css_set_func_irq(irq_num_t irqnum);
void pch_css_configure_func_irq_shared(irq_num_t irqnum, uint8_t order_priority);
void pch_css_configure_func_irq_exclusive(irq_num_t irqnum);
void pch_css_configure_func_irq_unused_shared(bool required, uint8_t order_priority);
void pch_css_configure_func_irq_unused_exclusive(bool required);
void pch_css_auto_configure_func_irq(bool required);

void pch_css_set_io_irq(irq_num_t irqnum);
void pch_css_configure_io_irq_shared(irq_num_t irqnum, uint8_t order_priority);
void pch_css_configure_io_irq_exclusive(irq_num_t irqnum);
void pch_css_configure_io_irq_unused_shared(bool required, uint8_t order_priority);
void pch_css_configure_io_irq_unused_exclusive(bool required);
void pch_css_auto_configure_io_irq(bool required);

io_callback_t pch_css_set_io_callback(io_callback_t io_callback);

void pch_css_start(io_callback_t io_callback, uint8_t isc_mask);

bool pch_css_set_trace(bool trace);

bool pch_chp_set_trace(pch_chpid_t chpid, bool trace);

void __isr pch_css_func_irq_handler(void);
void __isr pch_css_io_irq_handler(void);

void pch_css_set_io_irq(irq_num_t irqnum);


io_callback_t pch_css_set_io_callback(io_callback_t io_callback);

void pch_chp_start(pch_chpid_t chpid);

// Channel initialisation

void pch_chp_claim(pch_chpid_t chpid);

int pch_chp_claim_unused(bool required);

pch_sid_t pch_chp_alloc(pch_chpid_t chpid, uint16_t num_devices);


void pch_chp_configure_uartchan(pch_chpid_t chpid, uart_inst_t *uart, dma_channel_config ctrl);

void pch_chp_auto_configure_uartchan(pch_chpid_t chpid, uart_inst_t *uart, uint baudrate);

void pch_chp_configure_memchan(pch_chpid_t chpid, pch_dmaid_t txdmaid, pch_dmaid_t rxdmaid,
dmachan_tx_channel_t *txpeer);

// Channel initialisation low-level helpers
void pch_chp_dma_configure(pch_chpid_t chpid, dmachan_config_t *dc);
void pch_chp_set_configured(pch_chpid_t chpid, bool configured);
dmachan_tx_channel_t *pch_chp_get_tx_channel(pch_chpid_t chpid);
dmachan_rx_channel_t *pch_chp_get_rx_channel(pch_chpid_t chpid);

// Architectural API for subchannels and channel programs

int pch_sch_start(pch_sid_t sid, pch_ccw_t *ccw_addr);

int pch_sch_resume(pch_sid_t sid);

int pch_sch_test(pch_sid_t sid, pch_scsw_t *scsw);

int pch_sch_modify(pch_sid_t sid, pch_pmcw_t *pmcw);

int pch_sch_store(pch_sid_t sid, pch_schib_t *out_schib);

int pch_sch_cancel(pch_sid_t sid);

int pch_sch_halt(pch_sid_t sid);

pch_intcode_t pch_test_pending_interruption(void);

```

```
// API additions with internal optimisation

int pch_sch_store_pmcw(pch_sid_t sid, pch_pmcw_t *out_pmcw);

int pch_sch_store_scsw(pch_sid_t sid, pch_scsw_t *out_scsw);

// Convenience API functions that wrap the architectural API

int pch_sch_modify_intparm(pch_sid_t sid, uint32_t intparm);

int pch_sch_modify_flags(pch_sid_t sid, uint16_t flags);

int pch_sch_modify_isc(pch_sid_t sid, uint8_t isc);

int pch_sch_modify_enabled(pch_sid_t sid, bool enabled);

int pch_sch_modify_traced(pch_sid_t sid, bool traced);

void __time_critical_func(pch_sch_modify_isc_range)(pch_sid_t sid, uint count, uint8_t isc);

void __time_critical_func(pch_sch_modify_enabled_range)(pch_sid_t sid, uint count, bool enabled);

void __time_critical_func(pch_sch_modify_traced_range)(pch_sid_t sid, uint count, bool traced);

// These functions should only be called while the ISC for the
// subchannel has been disabled

int pch_sch_wait(pch_sid_t sid, pch_scsw_t *scsw);

int pch_sch_wait_timeout(pch_sid_t sid, pch_scsw_t *scsw, absolute_time_t timeout_timestamp);

int pch_sch_run_wait(pch_sid_t sid, pch_ccw_t *ccw_addr, pch_scsw_t *scsw);

int pch_sch_run_wait_timeout(pch_sid_t sid, pch_ccw_t *ccw_addr, pch_scsw_t *scsw, absolute_time_t
    timeout_timestamp);

#endif
```

13.38 css/include/picochan/pmcw.h File Reference

The Path Management Control World (PMCW)

```
#include <stdbool.h>
#include "picochan/ids.h"
```

Data Structures

- struct pch_pmcw

Macros

- #define **PCH_PMCW_SCH_MODIFY_MASK** 0x001f
- #define **PCH_PMCW_ISC_BITS** 0x07
- #define **PCH_PMCW_ISC_LSB** 0
- #define **PCH_PMCW_ENABLED** 0x08
- #define **PCH_PMCW_TRACED** 0x10

Typedefs

- typedef struct pch_pmcw pch_pmcw_t

Functions

- static uint8_t **pch_pmcw_isc** (pch_pmcw_t ∗pmcw)
- bool **pch_css_is_isc_enabled** (uint8_t iscnum)
- void **pch_css_set_isc_enabled** (uint8_t iscnum, bool enabled)
- void **pch_css_disable_isc** (uint8_t iscnum)
- void **pch_css_disable_isc_mask** (uint8_t mask)
- void **pch_css_enable_isc** (uint8_t iscnum)
- void **pch_css_enable_isc_mask** (uint8_t mask)
- void **pch_css_set_isc_enable_mask** (uint8_t mask)

13.38.1 Detailed Description

The Path Management Control World (PMCW)

13.38.2 Typedef Documentation

13.38.2.1 pch_pmcw_t

```
typedef struct pch_pmcw pch_pmcw_t
```

pch_pmcw_t is the Path Management Control World (PMCW)

This is an architected part of the schib. It contains

- the addressing information for the CSS to communicate with the device on its CU (see below)
- An Interruption Parameter (intparm) - a 32-bit value which is not modified by the CSS and can be used by the application for any purpose
- An Interrupt Service Class (ISC) so that groups of subchannels can be masked/unmasked together from delivering I/O interruptions
- The flag which indicates that the subchannel is enabled and can thus run channel programs
- A "trace" flag to indicate whether events for this subchannel can cause trace records to be written

Although for a mainframe channel subsystem, the addressing information in the PMCW contains 8 x 8-bit channel path id numbers referencing one or more channels that can reach the control unit, for picochan, the addressing information is simply a single channel path id (CHPID) and and the unit address of the device on the single remote CU to which it is connected.

The addressing information (CHPID and UnitAddr) must be set by the application (by using pch_chp_alloc) before the channel is started.

```
PMCW    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |               Interruption Parameter (Intparm)                |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |                     |T|E| ISC |      CHPID    | UnitAddr      |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

13.39 pmcw.h

Go to the documentation of this file.

```
/*
 * Copyright (c) 2025 Malcolm Beattie
 * SPDX-License-Identifier: MIT
 */

#ifndef _PCH_CSS_PMCW_H
#define _PCH_CSS_PMCW_H

#include <stdbool.h>
#include "picochan/ids.h"


typedef struct pch_pmcw {
        uint32_t        intparm;
        uint16_t        flags;
        pch_chpid_t     chpid;
        pch_unit_addr_t unit_addr;
} pch_pmcw_t;

// PCH_PMCW_SCH_MODIFY_MASK are the bits of the PMCW flags
// which can be set with the Modify Subchannel function
#define PCH_PMCW_SCH_MODIFY_MASK 0x001f

// ISC: Interrupt Service Class - the low 3 bits of the PMCW.
// We define PCH_PMCW_ISC_LSB as the shift count to get the ISC bits
// in case we ever want to move them, even though it's currently 0.
#define PCH_PMCW_ISC_BITS       0x07
#define PCH_PMCW_ISC_LSB        0
#define PCH_PMCW_ENABLED        0x08
#define PCH_PMCW_TRACED         0x10

static inline uint8_t pch_pmcw_isc(pch_pmcw_t *pmcw) {
        return (pmcw->flags & PCH_PMCW_ISC_BITS) » PCH_PMCW_ISC_LSB;
}

bool pch_css_is_isc_enabled(uint8_t iscnum);
void pch_css_set_isc_enabled(uint8_t iscnum, bool enabled);
void pch_css_disable_isc(uint8_t iscnum);
void pch_css_disable_isc_mask(uint8_t mask);
void pch_css_enable_isc(uint8_t iscnum);
void pch_css_enable_isc_mask(uint8_t mask);
void pch_css_set_isc_enable_mask(uint8_t mask);

#endif
```

13.40 css/include/picochan/schib.h File Reference

The Subchannel Information Block (SCHIB)

```
#include "picochan/ids.h"
#include "picochan/pmcw.h"
#include "picochan/scsw.h"
```

Data Structures

- struct pch_schib_mda

    The Model Dependent Area (MDA) of a schib.

- struct pch_schib

    pch_schib_t is the Subchannel Information Block (SCHIB)

Typedefs

- typedef struct pch_schib_mda pch_schib_mda_t

    The Model Dependent Area (MDA) of a schib.
- typedef struct pch_schib pch_schib_t

    pch_schib_t is the Subchannel Information Block (SCHIB)

Functions

- static bool **schib_is_enabled** (pch_schib_t ∗schib)
- static bool **schib_is_traced** (pch_schib_t ∗schib)
- static bool **schib_has_function_in_progress** (pch_schib_t ∗schib)
- static bool **schib_is_status_pending** (pch_schib_t ∗schib)

13.40.1 Detailed Description

The Subchannel Information Block (SCHIB)

13.40.2 Typedef Documentation

13.40.2.1 pch_schib_mda_t

```
typedef struct pch_schib_mda pch_schib_mda_t
```

The Model Dependent Area (MDA) of a schib.

Although this structure is part of the schib, pch_schib_t, and thus is visible to applications, the contents are for internal use by the CSS.

13.41 schib.h

Go to the documentation of this file.

```
/*
 * Copyright (c) 2025 Malcolm Beattie
 * SPDX-License-Identifier: MIT
 */

#ifndef _PCH_API_SCHIB_H
#define _PCH_API_SCHIB_H

#include "picochan/ids.h"
#include "picochan/pmcw.h"
#include "picochan/scsw.h"


typedef struct pch_schib_mda {
        uint32_t        data_addr;
        uint16_t        devcount;
        pch_unit_addr_t prevua;
        pch_unit_addr_t nextua;
        pch_sid_t       prevsid;
        pch_sid_t       nextsid;
} pch_schib_mda_t;
static_assert(sizeof(pch_schib_mda_t) == 12,
        "pch_schib_mda_t should be 12 bytes");

typedef struct pch_schib {
        pch_pmcw_t      pmcw;
        pch_scsw_t      scsw;
        pch_schib_mda_t mda;
} pch_schib_t;
static_assert(sizeof(pch_schib_t) == 32,
        "pch_schib_t should be 32 bytes");

static inline bool schib_is_enabled(pch_schib_t *schib) {
        return schib->pmcw.flags & PCH_PMCW_ENABLED;
}

static inline bool schib_is_traced(pch_schib_t *schib) {
        return schib->pmcw.flags & PCH_PMCW_TRACED;
}

static inline bool schib_has_function_in_progress(pch_schib_t *schib) {
        const uint16_t mask = PCH_FC_START|PCH_FC_HALT|PCH_FC_CLEAR;
        return schib->scsw.ctrl_flags & mask;
}

static inline bool schib_is_status_pending(pch_schib_t *schib) {
        return schib->scsw.ctrl_flags & PCH_SC_PENDING;
}

#endif
```

13.42 schib_dlist.h

```
/*
 * Copyright (c) 2025 Malcolm Beattie
 * SPDX-License-Identifier: MIT
 */

#ifndef _PCH_CSS_SCHIB_DLIST_H
#define _PCH_CSS_SCHIB_DLIST_H

// schib_dlist_t is a doubly linked (by sid) list of schibs
typedef int32_t schib_dlist_t;

pch_schib_t *remove_from_schib_dlist_unsafe(schib_dlist_t *l, pch_sid_t sid);
bool push_to_schib_dlist_unsafe(schib_dlist_t *l, pch_sid_t sid);

static inline pch_schib_t *remove_from_schib_dlist(schib_dlist_t *l, pch_sid_t sid) {
        uint32_t status = schibs_lock();
        pch_schib_t *schib = remove_from_schib_dlist_unsafe(l, sid);
        schibs_unlock(status);
        return schib;
}

static inline pch_schib_t *pop_schib_dlist_unsafe(schib_dlist_t *l) {
        if (*l == -1)
                return NULL;

        return remove_from_schib_dlist_unsafe(l, (pch_sid_t)*l);
}

static inline pch_schib_t *pop_schib_dlist(schib_dlist_t *l) {
        uint32_t status = schibs_lock();
        pch_schib_t *schib = pop_schib_dlist_unsafe(l);
        schibs_unlock(status);
        return schib;
}

static inline bool push_to_schib_dlist(schib_dlist_t *l, pch_sid_t sid) {
        uint32_t status = schibs_lock();
        bool was_empty = push_to_schib_dlist_unsafe(l, sid);
        schibs_unlock(status);
        return was_empty;
}

#endif
```

13.43 schib_internal.h

```
/*
 * Copyright (c) 2025 Malcolm Beattie
 * SPDX-License-Identifier: MIT
 */

```

```
#ifndef _PCH_CSS_SCHIB_INTERNAL_H
#define _PCH_CSS_SCHIB_INTERNAL_H

#include "picochan/schib.h"
#include "picochan/dev_status.h"
#include "picochan/ccw.h"

// get_stashed_ccw_flags is a CSS-internal function that fetches the
// CCW flags that we stash in the SCSW device status field during
// execution of a channel program. The SCSW device status field is
// only architected to be valid when Status Pending is set in the
// Status Control flags and we have to be careful only to stash
// CCW flags in this field when Status Pending is not set.
static inline pch_ccw_flags_t get_stashed_ccw_flags(pch_schib_t *schib) {
        return (pch_ccw_flags_t)schib->scsw.devs; // sic
}

#endif
```

13.44 schibs_lock.h

```
/*
 * Copyright (c) 2025 Malcolm Beattie
 * SPDX-License-Identifier: MIT
 */

#ifndef _PCH_CSS_SCHIBS_LOCK_H
#define _PCH_CSS_SCHIBS_LOCK_H

#include "hardware/sync.h"

// schibs_lock() and schibs_unlock() protect manipulation of the
// linked lists of schib's with pending functions (i.e. API
// functions such as Start Subchannel). The user API uses a
// critical section protected by schibs_lock()/schibs_unlock() to
// update the Function Control flags in the target schib with the
// request, add itself to the ua_func_dlist headed by the channel
// responsible for the subchannel (linked via mda.prevua/nextua) and
// ping the CSS with raise_func_irq.
// At the moment, we assume the user API invocations and the CSS
// itself run on the same core and so the ping is raising a
// (non-hardware-connected) IRQ and lock/unlock is a simple
// disable/restore of (all) interrupts. If we want to separate out
// the user invocations onto a different core from the CSS itself
// (and there's no inherent problem with that since the CSS runs
// entirely asynchronously and can cope with that) then we can
// change the ping to be a doorbell interrupt to the other core
// and change the lock/unlock to be a (hardware) spinlock plus the
// disable/restore of interrupts.

static inline uint32_t schibs_lock(void) {
        return save_and_disable_interrupts();
}

static inline void schibs_unlock(uint32_t status) {
        restore_interrupts(status);
}

#endif
```

13.45 callback.h

```
/*
 * Copyright (c) 2025 Malcolm Beattie
 * SPDX-License-Identifier: MIT
 */

#ifndef _PCH_CUS_CALLBACK_H
#define _PCH_CUS_CALLBACK_H

#include "picochan/cu.h"
#include "picochan/devib.h"
#include "picochan/dev_status.h"
#include "cus_trace.h"

static inline void pch_call_devib_callback(pch_cbindex_t cbindex, pch_devib_t *devib) {
        assert(pch_cbindex_is_callable(cbindex));

```

```
    if (cbindex == PCH_DEVIB_CALLBACK_NOOP)
        return;

    pch_devib_callback_t cb = pch_devib_callbacks[cbindex];
    cb(devib);
}

static inline void callback_devib(pch_devib_t *devib) {
        pch_cbindex_t cbindex = devib->cbindex;

        trace_call_callback(PCH_TRC_RT_CUS_CALL_CALLBACK,
                devib, cbindex);
        pch_call_devib_callback(cbindex, devib);
}

#endif
```

13.46 cu_internal.h

```
/*
 * Copyright (c) 2025 Malcolm Beattie
 * SPDX-License-Identifier: MIT
 */

#ifndef _PCH_CUS_CU_INTERNAL_H
#define _PCH_CUS_CU_INTERNAL_H

#include "picochan/cu.h"
#include "proto/packet.h"
#include "devibs_lock.h"

static inline proto_packet_t get_rx_packet(pch_cu_t *cu) {
        // cu.rx_channel is a dmachan_rx_channel_t which is
        // __aligned(4) and cmd is the first member of rx_channel
        // so is 4-byte aligned. proto_packet_t is 4-bytes and also
        // __aligned(4) (and needing no more than 4-byte alignment)
        // but omitting the __builtin_assume_aligned below causes
        // gcc 14.1.0 to produce error
        // error: cast increases required alignment of target type
        // [-Werror=cast-align]
        proto_packet_t *pp = (proto_packet_t *)
                __builtin_assume_aligned(&cu->rx_channel.link.cmd, 4);
        return *pp;
}

void pch_cus_send_command_to_css(pch_cu_t *cu);
void pch_cus_handle_rx_complete(pch_cu_t *cu);
void pch_cus_handle_tx_complete(pch_cu_t *cu);

#endif
```

13.47 cus_trace.h

```
/*
 * Copyright (c) 2025 Malcolm Beattie
 * SPDX-License-Identifier: MIT
 */

#ifndef _PCH_CUS_CUS_TRACE_H
#define _PCH_CUS_CUS_TRACE_H

#include "picochan/devib.h"
#include "picochan/cu.h"
#include "picochan/trc_records.h"
#include "trc/trace.h"
#include "proto/packet.h"
#include "txsm/txsm.h"

extern pch_trc_bufferset_t pch_cus_trace_bs;

#define PCH_CUS_TRACE_COND(rt, cond, data) \
        PCH_TRC_WRITE(&pch_cus_trace_bs, (cond), (rt), (data))

#define PCH_CUS_TRACE(rt, data) PCH_CUS_TRACE_COND((rt), true, (data))

static inline void trace_dev(pch_trc_record_type_t rt, pch_devib_t *devib) {
        PCH_CUS_TRACE_COND(rt, cu_or_devib_is_traced(devib),
                ((struct pch_trdata_dev){
                        .cuaddr = pch_dev_get_cuaddr(devib),
                        .ua = pch_dev_get_ua(devib)
                }));
}

static inline void trace_dev_byte(pch_trc_record_type_t rt, pch_devib_t *devib, uint8_t byte) {
        PCH_CUS_TRACE_COND(rt, cu_or_devib_is_traced(devib),
                ((struct pch_trdata_dev_byte){
                        .cuaddr = pch_dev_get_cuaddr(devib),
                        .ua = pch_dev_get_ua(devib),
                        .byte = byte
                }));
}

static inline void trace_dev_packet(pch_trc_record_type_t rt, pch_devib_t *devib, proto_packet_t p) {
        PCH_CUS_TRACE_COND(rt,
                cu_or_devib_is_traced(devib),
                ((struct pch_trdata_word_dev){
                        .word = proto_packet_as_word(p),
                        .cuaddr = pch_dev_get_cuaddr(devib),
                        .ua = pch_dev_get_ua(devib)
                }));
}

static inline void trace_tx_complete(pch_trc_record_type_t rt, pch_cu_t *cu, int16_t uaopt,
pch_txsm_state_t txpstate) {
        PCH_CUS_TRACE_COND(rt, cu->traced,
                ((struct pch_trdata_cus_tx_complete){
                        .uaopt = uaopt,
                        .cuaddr = cu->cuaddr,
                        .txpstate = (uint8_t)txpstate
                }));
}

static inline void trace_register_callback(pch_trc_record_type_t rt, pch_cbindex_t n,
pch_devib_callback_t cb) {
        PCH_CUS_TRACE(rt,
                ((struct pch_trdata_word_byte){(uint32_t)cb,n}));
}

static inline void trace_call_callback(pch_trc_record_type_t rt, pch_devib_t *devib, pch_cbindex_t
cbindex) {
        PCH_CUS_TRACE_COND(rt,
                cu_or_devib_is_traced(devib),
                ((struct pch_trdata_cus_call_callback){
                        .cuaddr = pch_dev_get_cuaddr(devib),
                        .ua = pch_dev_get_ua(devib),
                        .cbindex = (uint8_t)cbindex
                }));
}

static inline void trace_cu_irq(pch_trc_record_type_t rt, pch_cu_t *cu, pch_dma_irq_index_t dmairqix,
uint8_t tx_irq_state, uint8_t rx_irq_state) {
        PCH_CUS_TRACE_COND(rt,
                cu->traced, ((struct pch_trdata_id_irq){
                        .id = cu->cuaddr,
                        .dmairqix = dmairqix,
                        .tx_state = tx_irq_state « 4
                                | cu->tx_channel.mem_src_state,
                        .rx_state = rx_irq_state « 4
                                | cu->rx_channel.mem_dst_state
                }));
}

#endif
```

13.48 devibs_lock.h

```
/*
 * Copyright (c) 2025 Malcolm Beattie
 * SPDX-License-Identifier: MIT
 */

#ifndef _PCH_CUS_DEVIBS_LOCK_H
#define _PCH_CUS_DEVIBS_LOCK_H

#include "hardware/sync.h"

// devibs_lock() and devibs_unlock() protect manipulation of the
// linked lists of devibs 's with pending functions (i.e. API
// functions such as Start Subchannel). The device API uses a
// critical section protected by devibs_lock()/devibs_unlock() to
// add itself to the tx pending list headed by the devices' CU
// fields tx_head and tx_tail and linked via devib->next. The list
// is traversed and the pending packets sent (from the devib fields
// op and payload and using the devib's ua) whenever the CU's tx
// engine is free, driven by DMA completion on the tx channel.
// We assume the device API invocations and the CU itself run on the
// same core and so simply disable/restore (all) interrupts without
// needing to worry about cross-core locking.
static inline uint32_t devibs_lock(void) {
        return save_and_disable_interrupts();
}

static inline void devibs_unlock(uint32_t status) {
        restore_interrupts(status);
}

#endif
```

13.49 cu/include/picochan/cu.h File Reference

```
#include <stdint.h>
#include <assert.h>
#include "hardware/uart.h"
#include "picochan/dev_api.h"
#include "picochan/dmachan.h"
#include "txsm/txsm.h"
```

Data Structures

- struct pch_cu

  pch_cu_t is a Control Unit (CU)
- struct pch_dev_range

Macros

- #define **PARAM_ASSERTIONS_ENABLED_PCH_CUS** 0
- #define **PCH_MAX_DEVIBS_PER_CU** 32
- #define **PCH_MAX_DEVIBS_PER_CU_ALIGN_SHIFT** (31U - __builtin_clz(2 ∗ (PCH_MAX_DEVIBS_←PER_CU) - 1))
- #define **PCH_CU_ALIGN** (1U << (PCH_DEVIB_SPACE_SHIFT+PCH_MAX_DEVIBS_PER_CU_ALIGN←_SHIFT))
- #define PCH_NUM_CUS

  The number of control units.
- #define **PCH_CUS_BUFFERSET_MAGIC** 0x70437553
- #define PCH_CU_INIT(num_devices)

  a compile-time initialiser for a pch_cu_t

Typedefs

- typedef struct pch_cu pch_cu_t

  pch_cu_t is a Control Unit (CU)
- typedef struct pch_dev_range **pch_dev_range_t**

Functions

- static pch_dma_irq_index_t **pch_cu_get_dma_irq_index** (pch_cu_t ∗cu)
- void **pch_cu_set_dma_irq_index** (pch_cu_t ∗cu, pch_dma_irq_index_t dmairqix)
- static pch_cu_t ∗ **pch_dev_get_cu** (pch_devib_t ∗devib)
- static pch_cuaddr_t **pch_dev_get_cuaddr** (pch_devib_t ∗devib)
- static pch_unit_addr_t **pch_dev_get_ua** (pch_devib_t ∗devib)
- static pch_devib_t ∗ pch_get_devib (pch_cu_t ∗cu, pch_unit_addr_t ua)

  Look up the pch_devib_t of a device from its CU and unit address.
- static bool **cu_or_devib_is_traced** (pch_devib_t ∗devib)
- static pch_cu_t ∗ pch_get_cu (pch_cuaddr_t cua)

  Get the CU for a given control unit address.
- void pch_cus_init (void)

  Initialise CU subsystem.
- bool pch_cus_set_trace (bool trace)

  Sets whether CU subsystem tracing is enabled.
- void **pch_cus_configure_dma_irq_index_exclusive** (pch_dma_irq_index_t dmairqix)
- void **pch_cus_configure_dma_irq_index_shared** (pch_dma_irq_index_t dmairqix, uint8_t order_priority)
- void **pch_cus_configure_dma_irq_index_shared_default** (pch_dma_irq_index_t dmairqix)
- pch_dma_irq_index_t **pch_cus_auto_configure_dma_irq_index** (bool required)
- void **pch_cus_ignore_dma_irq_index_t** (pch_dma_irq_index_t dmairqix)
- void pch_cu_init (pch_cu_t ∗cu, uint16_t num_devibs)

  Initialises a CU with space for num_devibs devices.
- void pch_cu_register (pch_cu_t ∗cu, pch_cuaddr_t cua)

  Registers a CU at a control unit address.
- void pch_cus_uartcu_configure (pch_cuaddr_t cua, uart_inst_t ∗uart, dma_channel_config ctrl)

  Configure a UART control unit.
- void pch_cus_auto_configure_uartcu (pch_cuaddr_t cua, uart_inst_t ∗uart, uint baudrate)

  Initialise and configure a UART control unit with default dma_channel_config control register.
- void pch_cus_memcu_configure (pch_cuaddr_t cua, pch_dmaid_t txdmaid, pch_dmaid_t rxdmaid, dmachan_tx_channel_t ∗txpeer)

  Configure a memchan control unit.
- void pch_cu_start (pch_cuaddr_t cua)

  Starts the channel from CU cua to the CSS.
- bool pch_cus_trace_cu (pch_cuaddr_t cua, bool trace)

  Sets whether tracing is enabled for CU cua.
- bool pch_cus_trace_dev (pch_devib_t ∗devib, bool trace)

  Sets whether tracing is enabled for device.
- void **pch_cu_dma_configure** (pch_cuaddr_t cua, dmachan_config_t ∗dc)
- void **pch_cu_set_configured** (pch_cuaddr_t cua, bool configured)
- dmachan_tx_channel_t ∗ pch_cu_get_tx_channel (pch_cuaddr_t cua)

  Fetch the internal tx side of a channel from CU to CSS.
- dmachan_rx_channel_t ∗ **pch_cu_get_rx_channel** (pch_cuaddr_t cua)
- void __isr **pch_cus_handle_dma_irq** (void)
- static pch_unit_addr_t **pch_dev_range_get_ua** (pch_dev_range_t ∗dr, uint i)
- static int **pch_dev_range_get_index_nocheck** (pch_dev_range_t ∗dr, pch_devib_t ∗devib)
- static int **pch_dev_range_get_index** (pch_dev_range_t ∗dr, pch_devib_t ∗devib)
- static int **pch_dev_range_get_index_required** (pch_dev_range_t ∗dr, pch_devib_t ∗devib)
- static pch_devib_t ∗ **pch_dev_range_get_devib_by_index** (pch_dev_range_t ∗dr, uint i)
- static pch_devib_t ∗ **pch_dev_range_get_devib_by_ua_nocheck** (pch_dev_range_t ∗dr, pch_unit_addr_t ua)
- static pch_devib_t ∗ **pch_dev_range_get_devib_by_ua** (pch_dev_range_t ∗dr, pch_unit_addr_t ua)
- static pch_devib_t ∗ **pch_dev_range_get_devib_by_ua_required** (pch_dev_range_t ∗dr, pch_unit_addr_t ua)
- static void **pch_dev_range_init** (pch_dev_range_t ∗drout, pch_cu_t ∗cu, pch_unit_addr_t first_ua, uint16_t num_devices)
- static void **pch_dev_range_set_callback** (pch_dev_range_t ∗dr, pch_cbindex_t cbindex)

Variables

- pch_cu_t ∗ **pch_cus** [4]
- bool **pch_cus_init_done**

13.50 cu.h

Go to the documentation of this file.

```
/*
 * Copyright (c) 2025 Malcolm Beattie
 * SPDX-License-Identifier: MIT
 */

#ifndef _PCH_CUS_CU_H
#define _PCH_CUS_CU_H

// PICO_CONFIG: PARAM_ASSERTIONS_ENABLED_PCH_CUS, Enable/disable assertions in the pch_cus module, type=bool, default=0, group=pch_cus
#ifndef PARAM_ASSERTIONS_ENABLED_PCH_CUS
#define PARAM_ASSERTIONS_ENABLED_PCH_CUS 0
#endif

#ifndef PCH_MAX_DEVIBS_PER_CU
#define PCH_MAX_DEVIBS_PER_CU 32
#endif

#include <stdint.h>
#include <assert.h>
#include "hardware/uart.h"
#include "picochan/dev_api.h"
#include "picochan/dmachan.h"
#include "txsm/txsm.h"

static_assert(__builtin_constant_p(PCH_MAX_DEVIBS_PER_CU),
        "PCH_MAX_DEVIBS_PER_CU must be a compile-time constant");

static_assert(PCH_MAX_DEVIBS_PER_CU >= 1 && PCH_MAX_DEVIBS_PER_CU <= 256,
        "PCH_MAX_DEVIBS_PER_CU must be between 1 and 256");

#define PCH_MAX_DEVIBS_PER_CU_ALIGN_SHIFT (31U - __builtin_clz(2 * (PCH_MAX_DEVIBS_PER_CU) - 1))

static_assert(__builtin_constant_p(PCH_MAX_DEVIBS_PER_CU_ALIGN_SHIFT),
        "__builtin_clz() did not produce compile-time constant for PCH_MAX_DEVIBS_PER_CU_ALIGN_SHIFT");

#define PCH_CU_ALIGN (1U « (PCH_DEVIB_SPACE_SHIFT+PCH_MAX_DEVIBS_PER_CU_ALIGN_SHIFT))

static_assert(__builtin_constant_p(PCH_CU_ALIGN),
        "could not produce compile-time constant for PCH_CU_ALIGN");


#ifndef PCH_NUM_CUS
#define PCH_NUM_CUS 4
#endif
static_assert(PCH_NUM_CUS >= 1 && PCH_NUM_CUS <= 256,
        "PCH_NUM_CUS must be between 1 and 256");

#define PCH_CUS_BUFFERSET_MAGIC 0x70437553

typedef struct __aligned(PCH_CU_ALIGN) pch_cu {
        dmachan_tx_channel_t    tx_channel;
        dmachan_rx_channel_t    rx_channel;
        pch_txsm_t              tx_pending;
        pch_cuaddr_t            cuaddr;
    int16_t                 tx_callback_ua;
    int16_t                 rx_active;
    int16_t                 tx_head;
    int16_t                 tx_tail;
    pch_dma_irq_index_t     dmairqix;
    bool                    traced;
    bool                    configured;
    bool                    started;
        uint16_t                num_devibs;
    pch_devib_t             devibs[];
} pch_cu_t;

static inline pch_dma_irq_index_t pch_cu_get_dma_irq_index(pch_cu_t *cu) {
        return cu->dmairqix;
}
```

```

void pch_cu_set_dma_irq_index(pch_cu_t *cu, pch_dma_irq_index_t dmairqix);

#define PCH_CU_INIT(num_devices) { \
                .rx_active = -1, \
                .tx_head = -1, \
                .tx_tail = -1, \
                .dmairqix = -1, \
                .num_devibs = (num_devices), \
                .devibs = { [(num_devices)-1] = {0} } \
        }

static inline pch_cu_t *pch_dev_get_cu(pch_devib_t *devib) {
        unsigned long p = (unsigned long)devib;
        p -= __builtin_offsetof(pch_cu_t, devibs);
        p &= ~(PCH_CU_ALIGN-1);
        return (pch_cu_t *)p;
}

static inline pch_cuaddr_t pch_dev_get_cuaddr(pch_devib_t *devib) {
        pch_cu_t *cu = pch_dev_get_cu(devib);
        return cu->cuaddr;
}

static inline pch_unit_addr_t pch_dev_get_ua(pch_devib_t *devib) {
        pch_cu_t *cu = pch_dev_get_cu(devib);
        return (pch_unit_addr_t)(devib - cu->devibs);
}

static inline pch_devib_t *pch_get_devib(pch_cu_t *cu, pch_unit_addr_t ua) {
        return &cu->devibs[ua];
}

static inline bool cu_or_devib_is_traced(pch_devib_t *devib) {
        pch_cu_t *cu = pch_dev_get_cu(devib);
        return cu->traced || pch_devib_is_traced(devib);
}

extern pch_cu_t *pch_cus[PCH_NUM_CUS];

extern bool pch_cus_init_done;

static inline pch_cu_t *pch_get_cu(pch_cuaddr_t cua) {
        valid_params_if(PCH_CUS, cua < PCH_NUM_CUS);
        pch_cu_t *cu = pch_cus[cua];
        assert(cu != NULL);
        return cu;
}

void pch_cus_init(void);

bool pch_cus_set_trace(bool trace);

/*
 * \brief Configure an explicit DMA IRQ for use by CUs started from the
 * calling core and set an exclusive IRQ handler for it.
 * \ingroup picochan_cu
 *
 * If a CSS is to be used on the same Pico, it must be initialised on
 * a different core, using a different DMA IRQ index. A convenient way
 * to still allow the CU subsystem to auto-configure its DMA IRQ
 * choice is to call pch_cus_ignore_dma_irq_index_t() on the DMA IRQ
 * index of the CSS.
*/
void pch_cus_configure_dma_irq_index_exclusive(pch_dma_irq_index_t dmairqix);

/*
 * \brief Configure an explicit DMA IRQ for use by CUs started from
 * the calling core and add a shared IRQ handler for it.
 * \ingroup picochan_cu
 *
 * If a CSS is to be used on the same Pico, it must be initialised on
 * a different core, using a different DMA IRQ index. A convenient way
 * to still allow the CU subsystem to auto-configure its DMA IRQ
 * choice is to call pch_cus_ignore_dma_irq_index_t() on the DMA IRQ
 * index of the CSS.
*/
void pch_cus_configure_dma_irq_index_shared(pch_dma_irq_index_t dmairqix, uint8_t order_priority);

/*
 * \brief Configure an explicit DMA IRQ for use by CUs started from
 * the calling core and add a shared IRQ handler for it using an
 * order_priority of PICO_SHARED_IRQ_HANDLER_DEFAULT_ORDER_PRIORITY.
 * \ingroup picochan_cu
 *
 * If a CSS is to be used on the same Pico, it must be initialised on
 * a different core, using a different DMA IRQ index. A convenient way
 * to still allow the CU subsystem to auto-configure its DMA IRQ
 * choice is to call pch_cus_ignore_dma_irq_index_t() on the DMA IRQ
 * index of the CSS.
*/
void pch_cus_configure_dma_irq_index_shared_default(pch_dma_irq_index_t dmairqix);

/*
 * \brief Automatically choose and configure a suitable DMA IRQ for
 * use by CUs started from the calling core.
 * \ingroup picochan_cu
 *
 * If one of the explicit pch_cus_configure_dma_irq_index_...()
 * family of functions has already been called from the calling core
 * then the lowest such DMA IRQ index is returned. Otherwise, the
 * lowest DMA IRQ index is chosen that has not already been either
 * configured to any core or explicitly marked as not-to-use by
 * pch_cus_ignore_dma_irq_index_t(). It is then configured with
 * pch_cus_configure_dma_irq_index_shared_default() and returned.
 * In the case that no such unused index is available, the function
 * panics if required is true, otherwise -1 is returned.
 *
 * If a CSS is to be used on the same Pico, it must be initialised on
 * a different core, using a different DMA IRQ index. A convenient way
 * to still allow the CU subsystem to auto-configure its DMA IRQ
 * choice is to call pch_cus_ignore_dma_irq_index_t() on the DMA IRQ
 * index of the CSS.
*/
pch_dma_irq_index_t pch_cus_auto_configure_dma_irq_index(bool required);

/* \brief Marks dmairqix such that any call to
 * pch_cus_auto_configure_dma_irq_index(), whether explicit or
 * implicitly from pch_cu_start(), will not choose that DMA IRQ index.
 * \ingroup picochan_cu
 *
 * This function is convenient for avoiding the need to configure
 * explicit DMA IRQ index numbers for the CU subsystem while ensuring
 * that its auto-configuration of DMA IRQ index numbers does not
 * conflict with those of a CSS in use on the same Pico or just some
 * other DMA IRQ index that needs to be reserved for application use.
 */
void pch_cus_ignore_dma_irq_index_t(pch_dma_irq_index_t dmairqix);

// CU initialisation and configuration

void pch_cu_init(pch_cu_t *cu, uint16_t num_devibs);

void pch_cu_register(pch_cu_t *cu, pch_cuaddr_t cua);

void pch_cus_uartcu_configure(pch_cuaddr_t cua, uart_inst_t *uart, dma_channel_config ctrl);

void pch_cus_auto_configure_uartcu(pch_cuaddr_t cua, uart_inst_t *uart, uint baudrate);

void pch_cus_memcu_configure(pch_cuaddr_t cua, pch_dmaid_t txdmaid, pch_dmaid_t rxdmaid,
      dmachan_tx_channel_t *txpeer);

void pch_cu_start(pch_cuaddr_t cua);

bool pch_cus_trace_cu(pch_cuaddr_t cua, bool trace);

bool pch_cus_trace_dev(pch_devib_t *devib, bool trace);

// CU initialisation low-level helpers
void pch_cu_dma_configure(pch_cuaddr_t cua, dmachan_config_t *dc);
void pch_cu_set_configured(pch_cuaddr_t cua, bool configured);

dmachan_tx_channel_t *pch_cu_get_tx_channel(pch_cuaddr_t cua);

dmachan_rx_channel_t *pch_cu_get_rx_channel(pch_cuaddr_t cua);

void __isr pch_cus_handle_dma_irq(void);

typedef struct pch_dev_range {
        pch_cu_t         *cu;
        uint16_t         num_devices;    // 0 to 256
        pch_unit_addr_t first_ua;
} pch_dev_range_t;

static inline pch_unit_addr_t pch_dev_range_get_ua(pch_dev_range_t *dr, uint i) {
        assert(dr->cu);
        assert(i < dr->num_devices);
        assert((uint)dr->first_ua + i < dr->cu->num_devibs);

        return dr->first_ua + i;
}

static inline int pch_dev_range_get_index_nocheck(pch_dev_range_t *dr, pch_devib_t *devib) {
        return (int)pch_dev_get_ua(devib) - dr->first_ua;
}

static inline int pch_dev_range_get_index(pch_dev_range_t *dr, pch_devib_t *devib) {
        assert(dr->cu == pch_dev_get_cu(devib));

        int i = pch_dev_range_get_index_nocheck(dr, devib);
        if (i < 0 || i >= dr->num_devices)
                return -1;

        return i;
}

static inline int pch_dev_range_get_index_required(pch_dev_range_t *dr, pch_devib_t *devib) {
        int i = pch_dev_range_get_index(dr, devib);
        if (i < 0)
                panic("devib not found in dev_range");

        return i;
}

static inline pch_devib_t *pch_dev_range_get_devib_by_index(pch_dev_range_t *dr, uint i) {
        assert(dr->cu);

        pch_unit_addr_t ua = pch_dev_range_get_ua(dr, i);
        return pch_get_devib(dr->cu, ua);
}

static inline pch_devib_t *pch_dev_range_get_devib_by_ua_nocheck(pch_dev_range_t *dr, pch_unit_addr_t
ua) {
        assert(dr->cu);

        return pch_get_devib(dr->cu, ua);
}

static inline pch_devib_t *pch_dev_range_get_devib_by_ua(pch_dev_range_t *dr, pch_unit_addr_t ua) {
        assert(dr->cu);

        if (ua < dr->first_ua
                || (uint)ua >= (uint)dr->first_ua + (uint)dr->num_devices) {
                return NULL;
        }

        return pch_get_devib(dr->cu, ua);
}

static inline pch_devib_t *pch_dev_range_get_devib_by_ua_required(pch_dev_range_t *dr, pch_unit_addr_t
ua) {
        assert(dr->cu);

        if (ua < dr->first_ua
                || (uint)ua >= (uint)dr->first_ua + (uint)dr->num_devices) {
                panic("ua not in dev_range");
        }

        return pch_get_devib(dr->cu, ua);
}

static inline void pch_dev_range_init(pch_dev_range_t *drout, pch_cu_t *cu, pch_unit_addr_t first_ua,
uint16_t num_devices) {
        assert(cu);
        assert((uint)first_ua + (uint)num_devices <= cu->num_devibs);

        drout->cu = cu;
        drout->num_devices = num_devices;
        drout->first_ua = first_ua;
}

static inline void pch_dev_range_set_callback(pch_dev_range_t *dr, pch_cbindex_t cbindex) {
        assert(dr->cu);

        for (uint i = 0; i < dr->num_devices; i++) {
                pch_devib_t *devib = pch_dev_range_get_devib_by_index(dr, i);
                pch_dev_set_callback(devib, cbindex);
        }
}

#endif
```

13.51 cu/include/picochan/dev_api.h File Reference

The main API for a device on a CU.

```
#include "picochan/devib.h"
```

Typedefs

- typedef int(∗ **pch_dev_call_func_t**) (pch_devib_t ∗devib)

Enumerations

- enum {
ENOSUCHERROR = 1 , **EINVALIDCALLBACK** = 2 , **ENOTSTARTED** = 3 , **ECMDNOTREAD** = 4 ,
ECMDNOTWRITE = 5 , **EWRITETOOBIG** = 6 , **EINVALIDSTATUS** = 7 , **EINVALIDDEV** = 8 ,
EINVALIDCMD = 9 , **EINVALIDVALUE** = 10 , **EDATALENZERO** = 11 , **EBUFFERTOOSHORT** = 12 ,
ECANCEL = 256 }

Functions

- int pch_dev_set_callback (pch_devib_t ∗devib, int cbindex_opt)

  Set callback for device.
- int pch_dev_send_then (pch_devib_t ∗devib, void ∗srcaddr, uint16_t n, proto_chop_flags_t flags, int cbindex_opt)

  Sends data to the CSS.
- int pch_dev_send_zeroes_then (pch_devib_t ∗devib, uint16_t n, proto_chop_flags_t flags, int cbindex_opt)

  Sends zeroes to the CSS.
- int pch_dev_receive_then (pch_devib_t ∗devib, void ∗dstaddr, uint16_t size, int cbindex_opt)

  Receive data from the CSS.
- int **pch_dev_update_status_advert_then** (pch_devib_t ∗devib, uint8_t devs, void ∗dstaddr, uint16_t size, int cbindex_opt)
- int **pch_dev_send** (pch_devib_t ∗devib, void ∗srcaddr, uint16_t n, proto_chop_flags_t flags)
- int **pch_dev_send_final** (pch_devib_t ∗devib, void ∗srcaddr, uint16_t n)
- int **pch_dev_send_final_then** (pch_devib_t ∗devib, void ∗srcaddr, uint16_t n, int cbindex_opt)
- int **pch_dev_send_respond** (pch_devib_t ∗devib, void ∗srcaddr, uint16_t n)
- int **pch_dev_send_respond_then** (pch_devib_t ∗devib, void ∗srcaddr, uint16_t n, int cbindex_opt)
- int **pch_dev_send_norespond** (pch_devib_t ∗devib, void ∗srcaddr, uint16_t n)
- int **pch_dev_send_norespond_then** (pch_devib_t ∗devib, void ∗srcaddr, uint16_t n, int cbindex_opt)
- int **pch_dev_send_zeroes** (pch_devib_t ∗devib, uint16_t n, proto_chop_flags_t flags)
- int **pch_dev_send_zeroes_respond_then** (pch_devib_t ∗devib, uint16_t n, int cbindex_opt)
- int **pch_dev_send_zeroes_respond** (pch_devib_t ∗devib, uint16_t n)
- int **pch_dev_send_zeroes_norespond_then** (pch_devib_t ∗devib, uint16_t n, int cbindex_opt)
- int **pch_dev_send_zeroes_norespond** (pch_devib_t ∗devib, uint16_t n)
- int **pch_dev_receive** (pch_devib_t ∗devib, void ∗dstaddr, uint16_t size)
- int **pch_dev_update_status_then** (pch_devib_t ∗devib, uint8_t devs, int cbindex_opt)
- int **pch_dev_update_status** (pch_devib_t ∗devib, uint8_t devs)
- int **pch_dev_update_status_advert** (pch_devib_t ∗devib, uint8_t devs, void ∗dstaddr, uint16_t size)
- int **pch_dev_update_status_ok_then** (pch_devib_t ∗devib, int cbindex_opt)
- int **pch_dev_update_status_ok** (pch_devib_t ∗devib)
- int **pch_dev_update_status_ok_advert** (pch_devib_t ∗devib, void ∗dstaddr, uint16_t size)
- int **pch_dev_update_status_error_advert_then** (pch_devib_t ∗devib, pch_dev_sense_t sense, void ∗dstaddr, uint16_t size, int cbindex_opt)
- int **pch_dev_update_status_error_then** (pch_devib_t ∗devib, pch_dev_sense_t sense, int cbindex_opt)
- int **pch_dev_update_status_error_advert** (pch_devib_t ∗devib, pch_dev_sense_t sense, void ∗dstaddr, uint16_t size)
- int **pch_dev_update_status_error** (pch_devib_t ∗devib, pch_dev_sense_t sense)
- int pch_dev_call_or_reject_then (pch_devib_t ∗devib, pch_dev_call_func_t f, int reject_cbindex_opt)
- void pch_dev_call_final_then (pch_devib_t ∗devib, pch_dev_call_func_t f, int cbindex_opt)

13.51.1 Detailed Description

The main API for a device on a CU.

These provide a slightly higher-level API by wrapping the low-level pch_devib_ API functions.

13.51.2 Function Documentation

13.51.2.1 pch_dev_call_final_then()

```
void pch_dev_call_final_then (
            pch_devib_t * devib,
            pch_dev_call_func_t f,
            int cbindex_opt)
```

Calls f, sends an UpdateStatus with an appropriate payload based on its return value then sets cbindex_opt as the next callback. If f returns a negative value, the UpdateStatus payload is UnitCheck with sense CommandReject with the associated negated (positive) error value or else, if f returns a non-negative valuem the UpdateStatus payload is normal "no error" with ChannelEnd|DeviceEnd.

13.51.2.2 pch_dev_call_or_reject_then()

```
int pch_dev_call_or_reject_then (
            pch_devib_t * devib,
            pch_dev_call_func_t f,
            int reject_cbindex_opt)
```

Calls f and, if it returns a negative value, sets an appropriate sense, triggers an UpdateStatus to report the error and sets the "next callback" index. If f returns a non-negative value, no action is taken. In either case, the return value of f is propagated to the caller.

When f returns a negative value between -1 and -255, the sense set is CommandReject with an ASC byte of the associated negated (positive) error value. When f returns -ECANCEL (-256), the sense set is Cancel.

13.52 dev_api.h

Go to the documentation of this file.

```
/*
 * Copyright (c) 2025 Malcolm Beattie
 * SPDX-License-Identifier: MIT
 */

#ifndef _PCH_CU_DEV_API_H
#define _PCH_CU_DEV_API_H

#include "picochan/devib.h"


// Main API for dev implementation, slightly higher level than
// devib ones. They return negative error values on error
// (e.g. -EINVAL). They do various parameter checks and return
// errors instead of asserting like the low-level API does. Those
// with cbindex_opt arguments leave the devib cbindex field alone
// if called with a negative value, otherwise they validate it
// as a callback cbindex and set the field or return a negative
// error value, as appropriate.  For sends (of data or zeroes), the
// length sent is validated to be under the CSS-advertised window
// (devib->size) and capped at that if not, with the actual count
// returned. Many functions are variants of the full generic ones
// that simply specialise the callback and flags fields.
// Values between 1 and 255 are typically used to fit into the ASC
// byte of a pch_dev_sense_t with sense code
// PCH_DEV_SENSE_COMMAND_REJECT. ECANCEL is associated with sense
// code PCH_DEV_SENSE_CANCEL.

enum {
        ENOSUCHERROR            = 1,
        EINVALIDCALLBACK    = 2,
        ENOTSTARTED     = 3,
        ECMDNOTREAD     = 4,
        ECMDNOTWRITE        = 5,
        EWRITETOOBIG        = 6,
        EINVALIDSTATUS      = 7,
        EINVALIDDEV             = 8,
        EINVALIDCMD             = 9,
        EINVALIDVALUE           = 10,
        EDATALENZERO            = 11,
        EBUFFERTOOSHORT         = 12,
        //
        ECANCEL                 = 256
};

// dev API with fully general arguments

int pch_dev_set_callback(pch_devib_t *devib, int cbindex_opt);

int pch_dev_send_then(pch_devib_t *devib, void *srcaddr, uint16_t n, proto_chop_flags_t flags, int
cbindex_opt);

int pch_dev_send_zeroes_then(pch_devib_t *devib, uint16_t n, proto_chop_flags_t flags, int
cbindex_opt);

int pch_dev_receive_then(pch_devib_t *devib, void *dstaddr, uint16_t size, int cbindex_opt);

int pch_dev_update_status_advert_then(pch_devib_t *devib, uint8_t devs, void *dstaddr, uint16_t size,
int cbindex_opt);

// dev API convenience functions with some fixed arguments:
// * Omitting _then avoids setting devib callback by hardcoding -1
// as the cbindex_opt argument of the full _then function.
// * For send and send_zeroes family, the flags argument is set to
//     * PROTO_CHOP_FLAG_END for the _final variant,
//     * PROTO_CHOP_FLAG_RESPONSE_REQUIRED for the _respond variant
//     * 0 for the _norespond variant
// * For pch_dev_update_status_ok family, call the corresponding
// pch_dev_update_status_ function with DeviceEnd|ChannelEnd
// * For pch_dev_update_status_error family, set devib->sense to the
// sense argument then call the corresponding pch_dev_update_status_
// function with a device status of DeviceEnd|ChannelEnd|UnitCheck
int pch_dev_send(pch_devib_t *devib, void *srcaddr, uint16_t n, proto_chop_flags_t flags);
int pch_dev_send_final(pch_devib_t *devib, void *srcaddr, uint16_t n);
int pch_dev_send_final_then(pch_devib_t *devib, void *srcaddr, uint16_t n, int cbindex_opt);
int pch_dev_send_respond(pch_devib_t *devib, void *srcaddr, uint16_t n);
int pch_dev_send_respond_then(pch_devib_t *devib, void *srcaddr, uint16_t n, int cbindex_opt);
int pch_dev_send_norespond(pch_devib_t *devib, void *srcaddr, uint16_t n);
int pch_dev_send_norespond_then(pch_devib_t *devib, void *srcaddr, uint16_t n, int cbindex_opt);
int pch_dev_send_zeroes(pch_devib_t *devib, uint16_t n, proto_chop_flags_t flags);
int pch_dev_send_zeroes_respond_then(pch_devib_t *devib, uint16_t n, int cbindex_opt);
int pch_dev_send_zeroes_respond(pch_devib_t *devib, uint16_t n);
int pch_dev_send_zeroes_norespond_then(pch_devib_t *devib, uint16_t n, int cbindex_opt);
int pch_dev_send_zeroes_norespond(pch_devib_t *devib, uint16_t n);
int pch_dev_receive(pch_devib_t *devib, void *dstaddr, uint16_t size);
int pch_dev_update_status_then(pch_devib_t *devib, uint8_t devs, int cbindex_opt);
int pch_dev_update_status(pch_devib_t *devib, uint8_t devs);
int pch_dev_update_status_advert(pch_devib_t *devib, uint8_t devs, void *dstaddr, uint16_t size);
int pch_dev_update_status_ok_then(pch_devib_t *devib, int cbindex_opt);
int pch_dev_update_status_ok(pch_devib_t *devib);
int pch_dev_update_status_ok_advert(pch_devib_t *devib, void *dstaddr, uint16_t size);
int pch_dev_update_status_error_advert_then(pch_devib_t *devib, pch_dev_sense_t sense, void *dstaddr,
uint16_t size, int cbindex_opt);
int pch_dev_update_status_error_then(pch_devib_t *devib, pch_dev_sense_t sense, int cbindex_opt);
int pch_dev_update_status_error_advert(pch_devib_t *devib, pch_dev_sense_t sense, void *dstaddr,
uint16_t size);
int pch_dev_update_status_error(pch_devib_t *devib, pch_dev_sense_t sense);

typedef int (*pch_dev_call_func_t)(pch_devib_t *devib);

int pch_dev_call_or_reject_then(pch_devib_t *devib, pch_dev_call_func_t f, int reject_cbindex_opt);

void pch_dev_call_final_then(pch_devib_t *devib, pch_dev_call_func_t f, int cbindex_opt);

#endif
```

13.53 cu/include/picochan/dev_sense.h File Reference

Device sense.

Data Structures

- struct pch_dev_sense
  The device sense structure by which a device can communicate additional error information on request by the CSS.

Macros

- #define **PCH_DEV_SENSE_COMMAND_REJECT** 0x80
- #define **PCH_DEV_SENSE_INTERVENTION_REQUIRED** 0x40
- #define **PCH_DEV_SENSE_BUS_OUT_CHECK** 0x20
- #define **PCH_DEV_SENSE_EQUIPMENT_CHECK** 0x10
- #define **PCH_DEV_SENSE_DATA_CHECK** 0x08
- #define **PCH_DEV_SENSE_OVERRUN** 0x04
- #define **PCH_DEV_SENSE_PROTO_ERROR** 0x02
- #define **PCH_DEV_SENSE_CANCEL** 0x01

Typedefs

- typedef struct pch_dev_sense **pch_dev_sense_t**
  The device sense structure by which a device can communicate additional error information on request by the CSS.

13.53.1 Detailed Description

Device sense.

13.54 dev_sense.h

Go to the documentation of this file.

```
/*
 * Copyright (c) 2025 Malcolm Beattie
 * SPDX-License-Identifier: MIT
 */

#ifndef _PCH_CU_DEV_SENSE_H
#define _PCH_CU_DEV_SENSE_H


typedef struct __attribute__((__packed__,__aligned__(4))) pch_dev_sense {
        uint8_t flags;
        uint8_t code;
        uint8_t asc;
        uint8_t ascq;
} pch_dev_sense_t;

#define PCH_DEV_SENSE_COMMAND_REJECT            0x80
#define PCH_DEV_SENSE_INTERVENTION_REQUIRED     0x40
#define PCH_DEV_SENSE_BUS_OUT_CHECK             0x20
#define PCH_DEV_SENSE_EQUIPMENT_CHECK           0x10
#define PCH_DEV_SENSE_DATA_CHECK                0x08
#define PCH_DEV_SENSE_OVERRUN                   0x04
#define PCH_DEV_SENSE_PROTO_ERROR               0x02
#define PCH_DEV_SENSE_CANCEL                    0x01

#endif
```

13.55 cu/include/picochan/devib.h File Reference

The structures and API for a device on a CU.

```
#include "pico/platform/compiler.h"
#include "picochan/ids.h"
#include "picochan/dev_status.h"
#include "picochan/dev_sense.h"
#include "proto/chop.h"
#include "proto/payload.h"
```

Data Structures

- struct pch_devib

  pch_devib_t represents a device on a CU

Macros

- #define **PCH_DEVIB_CALLBACK_DEFAULT** 0
- #define **PCH_DEVIB_CALLBACK_NOOP** 255
- #define MAX_DEVIB_CALLBACKS 254

  The maximum number of registered callbacks.
- #define NUM_DEVIB_CALLBACKS 16

  The size of the global callbacks array.
- #define **PCH_DEVIB_SPACE_SHIFT** (31U - __builtin_clz(2 ∗ sizeof(pch_devib_t) - 1))
- #define **PCH_DEVIB_FLAG_STARTED** 0x80
- #define **PCH_DEVIB_FLAG_CMD_WRITE** 0x40
- #define **PCH_DEVIB_FLAG_RX_DATA_REQUIRED** 0x20
- #define **PCH_DEVIB_FLAG_TX_CALLBACK** 0x10
- #define **PCH_DEVIB_FLAG_TRACED** 0x08
- #define **PCH_DEVIB_FLAG_STOPPING** 0x04

Typedefs

- typedef uint8_t **pch_cbindex_t**

  An 8-bit index into an array of callbacks that the CU can make to a device

  pch_cbindex_t is an 8-bit index into pch_devib_callbacks, an array of up to NUM_DEVIB_CALLBACKS registered callbacks on devibs.
- typedef struct pch_devib pch_devib_t

  pch_devib_t represents a device on a CU
- typedef void(∗ **pch_devib_callback_t**) (pch_devib_t ∗devib)

  pch_devib_callback_t is a function for the CU to callback a device

Functions

- static bool **pch_devib_is_started** (pch_devib_t ∗devib)
- static bool **pch_devib_is_cmd_write** (pch_devib_t ∗devib)
- static bool **pch_devib_is_traced** (pch_devib_t ∗devib)
- static bool **pch_devib_set_traced** (pch_devib_t ∗devib, bool trace)
- static bool **pch_devib_is_stopping** (pch_devib_t ∗devib)
- static bool **pch_cbindex_is_callable** (uint cbindex)
- void pch_register_devib_callback (pch_cbindex_t n, pch_devib_callback_t cb)

  Registers a device callback function at a specific index.
- pch_cbindex_t pch_register_unused_devib_callback (pch_devib_callback_t cb)

  Registers a device callback function at an unused index.
- void **pch_default_devib_callback** (pch_devib_t ∗devib)
- static void pch_devib_prepare_callback (pch_devib_t ∗devib, pch_cbindex_t cbindex)

  Low-level API to update devib->cbindex.
- static void pch_devib_prepare_count (pch_devib_t ∗devib, uint16_t count)

  Low-level API to update devib->payload with a count field.
- static void pch_devib_prepare_write_data (pch_devib_t ∗devib, void ∗srcaddr, uint16_t n, proto_chop_flags↩_t flags)

  Low-level API to prepare a Data channel operation command for a device.
- static void pch_devib_prepare_write_zeroes (pch_devib_t ∗devib, uint16_t n, proto_chop_flags_t flags)

  Low-level API to prepare a Data channel operation command for a device that will implicitly send zeroes.
- static void pch_devib_prepare_read_data (pch_devib_t ∗devib, void ∗dstaddr, uint16_t size)

  Low-level API to prepare a RequestRead channel operation command for a device.
- void pch_devib_prepare_update_status (pch_devib_t ∗devib, uint8_t devs, void ∗dstaddr, uint16_t size)

  Low-level API to prepare an UpdateStatus channel operation command for a device.
- void **pch_devib_send_or_queue_command** (pch_devib_t ∗devib)

Variables

- pch_devib_callback_t **pch_devib_callbacks** [ ]

13.55.1 Detailed Description

The structures and API for a device on a CU.

13.56 devib.h

Go to the documentation of this file.

```
/*
 * Copyright (c) 2025 Malcolm Beattie
 * SPDX-License-Identifier: MIT
 */

#ifndef _PCH_CU_DEVIB_H
#define _PCH_CU_DEVIB_H

#include "pico/platform/compiler.h"
#include "picochan/ids.h"
#include "picochan/dev_status.h"
#include "picochan/dev_sense.h"
#include "proto/chop.h"
#include "proto/payload.h"


```

```c
typedef uint8_t pch_cbindex_t;

#define PCH_DEVIB_CALLBACK_DEFAULT      0
#define PCH_DEVIB_CALLBACK_NOOP         255

#define MAX_DEVIB_CALLBACKS 254

#define NUM_DEVIB_CALLBACKS 16
static_assert(NUM_DEVIB_CALLBACKS <= MAX_DEVIB_CALLBACKS,
        "NUM_DEVIB_CALLBACKS must not exceed MAX_DEVIB_CALLBACKS");

static_assert(sizeof(pch_dev_sense_t) == 4,
        "pch_dev_sense_t must be 4 bytes");

typedef struct __aligned(4) pch_devib {
        pch_unit_addr_t next;
        pch_cbindex_t   cbindex;
        uint16_t        size;
        proto_chop_t    op;
        uint8_t         flags;
        proto_payload_t payload;
        uint32_t        addr;
        pch_dev_sense_t sense;
} pch_devib_t;

#define PCH_DEVIB_SPACE_SHIFT (31U - __builtin_clz(2 * sizeof(pch_devib_t) - 1))

static_assert(__builtin_constant_p(PCH_DEVIB_SPACE_SHIFT),
        "__builtin_clz() did not produce compile-time constant for PCH_DEVIB_SPACE_SHIFT");

#define PCH_DEVIB_FLAG_STARTED          0x80
#define PCH_DEVIB_FLAG_CMD_WRITE        0x40
#define PCH_DEVIB_FLAG_RX_DATA_REQUIRED 0x20
#define PCH_DEVIB_FLAG_TX_CALLBACK      0x10
#define PCH_DEVIB_FLAG_TRACED           0x08
#define PCH_DEVIB_FLAG_STOPPING         0x04

static inline bool pch_devib_is_started(pch_devib_t *devib) {
        return devib->flags & PCH_DEVIB_FLAG_STARTED;
}

static inline bool pch_devib_is_cmd_write(pch_devib_t *devib) {
        return devib->flags & PCH_DEVIB_FLAG_CMD_WRITE;
}

static inline bool pch_devib_is_traced(pch_devib_t *devib) {
        return devib->flags & PCH_DEVIB_FLAG_TRACED;
}

static inline bool pch_devib_set_traced(pch_devib_t *devib, bool trace) {
        bool old_trace = pch_devib_is_traced(devib);
        if (trace)
                devib->flags |= PCH_DEVIB_FLAG_TRACED;
        else
                devib->flags &= ~PCH_DEVIB_FLAG_TRACED;

        return old_trace;
}

static inline bool pch_devib_is_stopping(pch_devib_t *devib) {
        return devib->flags & PCH_DEVIB_FLAG_STOPPING;
}

// Forward declaration of pch_cu_t for identifying devib by
// (pch_cu_t, pch_unit_addr_t) for callbacks and dev implementations.
typedef struct pch_cu pch_cu_t;

// Callbacks

typedef void (*pch_devib_callback_t)(pch_devib_t *devib);

extern pch_devib_callback_t pch_devib_callbacks[];

static inline bool pch_cbindex_is_callable(uint cbindex) {
        if (cbindex == PCH_DEVIB_CALLBACK_NOOP)
                return true;

        if (cbindex >= NUM_DEVIB_CALLBACKS)
                return false;

        return pch_devib_callbacks[cbindex] != NULL;
}

// Callback registration API

void pch_register_devib_callback(pch_cbindex_t n, pch_devib_callback_t cb);

```

```c
pch_cbindex_t pch_register_unused_devib_callback(pch_devib_callback_t cb);

void pch_default_devib_callback(pch_devib_t *devib);

// Low-level API for dev implementation updating devib

static inline void pch_devib_prepare_callback(pch_devib_t *devib, pch_cbindex_t cbindex) {
        assert(pch_cbindex_is_callable(cbindex));
        devib->cbindex = cbindex;
}

static inline void pch_devib_prepare_count(pch_devib_t *devib, uint16_t count) {
        devib->payload = proto_make_count_payload(count);
}

static inline void pch_devib_prepare_write_data(pch_devib_t *devib, void *srcaddr, uint16_t n, proto_chop_flags_t flags) {
        assert(devib->flags & PCH_DEVIB_FLAG_STARTED);
        pch_devib_prepare_count(devib, n);
        devib->op = PROTO_CHOP_DATA | flags;
        devib->addr = (uint32_t)srcaddr;
}

static inline void pch_devib_prepare_write_zeroes(pch_devib_t *devib, uint16_t n, proto_chop_flags_t flags) {
        assert(devib->flags & PCH_DEVIB_FLAG_STARTED);
        pch_devib_prepare_count(devib, n);
        // hard-code the ResponseRequired flag for now
        devib->op = PROTO_CHOP_DATA | PROTO_CHOP_FLAG_SKIP | flags;
}

static inline void pch_devib_prepare_read_data(pch_devib_t *devib, void *dstaddr, uint16_t size) {
        assert(devib->flags & PCH_DEVIB_FLAG_STARTED);
        pch_devib_prepare_count(devib, size);
        devib->op = PROTO_CHOP_REQUEST_READ;
        devib->flags |= PCH_DEVIB_FLAG_RX_DATA_REQUIRED;
        devib->addr = (uint32_t)dstaddr;
}

void pch_devib_prepare_update_status(pch_devib_t *devib, uint8_t devs, void *dstaddr, uint16_t size);

void pch_devib_send_or_queue_command(pch_devib_t *devib);
#endif
```

Index