

Picochan

0.1

Generated by Doxygen 1.14.0

1 Picochan - a channel subsystem for Raspberry Pi Pico	1
1.1 Introduction	1
1.2 Channel Programs	2
1.3 Picochan APIs	2
1.4 Compiling	2
1.5 Tracing	3
1.6 Design	3
2 Channel programs	5
2.0.1 Introduction	5
2.0.2 When a CCW command in a channel program is finished	6
2.0.3 Channel program command chaining	7
2.0.4 Channel program ending	7
2.0.5 Notification to application	8
3 Compiling using Picochan	11
4 Channel Subsystem (CSS) API for Applications	13
4.0.1 Introduction	13
4.0.2 Compile-time constants and definitions - examples:	13
4.0.3 Debugging assertions:	13
4.0.4 Types	14
4.0.5 Initialisation of whole CSS	14
4.0.6 Allocation of subchannels in a channel to a CU	14
4.0.6.1 Initialise a channel to a UART CU	14
4.0.6.2 Initialise a channel to a memchan (cross-core) CU	14
4.0.6.3 Initialise a channel to a pio CU	14
4.0.7 Start a channel to a CU	14
4.0.8 Set PMCW flags of a subchannel to enable/disable, trace or change ISC	15
4.0.9 Start, monitor and control channel programs for a subchannel	15
4.0.10 Variations and wrappers for convenience or optimisation	15
5 Control Unit (CU) API for Device Drivers Overview	17
5.0.1 Introduction	17
5.0.2 Types	18
5.0.3 Compile-time constants and definitions - examples:	18
5.0.4 Debugging assertions:	18
5.0.5 Initialisation of whole CU subsystem	18
5.0.6 Initialisation of each CU	19
5.0.7 Convenience API for device driver to its CU	19
5.0.7.1 Convenience API with fully general arguments	19
5.0.7.2 Convenience API with some fixed arguments	19
5.0.8 Low-level API for device driver to its CU	20

6 Design of Picochan	21
6.0.1 Design	21
6.0.2 CSS <-> CU protocol	22
7 Tracing	23
7.0.1 Introduction	23
7.0.2 pch_dump_trace - parse and display traces (off-platform)	23
7.0.3 Interactive "offload trace buffers and parse/display" with gdb	24
7.0.4 API to enable/disable trace at various levels	24
7.0.4.1 Tracing for CSS	25
7.0.4.2 Tracing for CU	25
8 Topic Index	27
8.1 Topics	27
9 Data Structure Index	29
9.1 Data Structures	29
10 File Index	31
10.1 File List	31
11 Topic Documentation	33
11.1 picochan_base	33
11.1.1 Detailed Description	35
11.1.2 Macro Definition Documentation	35
11.1.2.1 PCH_BSIZE_ZERO	35
11.1.3 Typedef Documentation	35
11.1.3.1 pch_bsizex_t	35
11.1.3.2 pch_ccw_t	35
11.1.3.3 pch_chpid_t	36
11.1.3.4 pch_dma_irq_index_t	36
11.1.3.5 pch_dmaid_t	36
11.1.3.6 pch_schib_t	36
11.1.3.7 pch_unit_addr_t	37
11.1.4 Function Documentation	37
11.1.4.1 pch_bsizex_decode_inline()	37
11.1.4.2 pch_bsizex_decode_raw_inline()	37
11.1.4.3 pch_bsizex_encode_inline()	37
11.1.4.4 pch_bsizex_encode_raw_inline()	37
11.1.4.5 pch_bsizex_encode_dx_inline()	38
11.1.4.6 pch_bsizex_wrap()	38
11.2 internal_trc	38
11.2.1 Detailed Description	39
11.2.2 Macro Definition Documentation	39

11.2.2.1 PCH_CONFIG_ENABLE_TRACE	39
11.2.3 Typedef Documentation	39
11.2.3.1 pch_trc_bufferset_t	39
11.2.3.2 pch_trc_timestamp_t	39
11.3 internal_proto	39
11.3.1 Detailed Description	40
11.3.2 Typedef Documentation	40
11.3.2.1 proto_packet_t	40
11.4 internal_css	40
11.4.1 Detailed Description	40
11.4.2 Typedef Documentation	41
11.4.2.1 pch_chp_t	41
11.5 picochan_css	41
11.5.1 Detailed Description	43
11.5.2 Macro Definition Documentation	43
11.5.2.1 PCH_NUM_CHANNELS	43
11.5.2.2 PCH_NUM_ISCS	44
11.5.2.3 PCH_NUM_SCHIBS	44
11.5.3 Function Documentation	44
11.5.3.1 pch_ccw_get_addr()	44
11.5.3.2 pch_chp_alloc()	44
11.5.3.3 pch_chp_auto_configure_uartchan()	44
11.5.3.4 pch_chp_configure_memchan()	45
11.5.3.5 pch_chp_configure_uartchan()	45
11.5.3.6 pch_chp_get_tx_channel()	45
11.5.3.7 pch_chp_set_trace()	46
11.5.3.8 pch_chp_start()	46
11.5.3.9 pch_css_init()	46
11.5.3.10 pch_css_set_func_irq()	46
11.5.3.11 pch_css_set_io_callback()	47
11.5.3.12 pch_css_set_io_irq()	47
11.5.3.13 pch_css_set_trace()	48
11.5.3.14 pch_css_start()	48
11.5.3.15 pch_cus_auto_configure_uartcu()	48
11.5.3.16 pch_sch_cancel()	48
11.5.3.17 pch_sch_halt()	49
11.5.3.18 pch_sch_modify()	49
11.5.3.19 pch_sch_modify_enabled()	49
11.5.3.20 pch_sch_modify_flags()	49
11.5.3.21 pch_sch_modify_intparm()	50
11.5.3.22 pch_sch_modify_isc()	50
11.5.3.23 pch_sch_modify_traced()	50

11.5.3.24 pch_sch_resume()	50
11.5.3.25 pch_sch_run_wait()	50
11.5.3.26 pch_sch_run_wait_timeout()	51
11.5.3.27 pch_sch_start()	51
11.5.3.28 pch_sch_store()	51
11.5.3.29 pch_sch_store_pmcw()	51
11.5.3.30 pch_sch_store_scsw()	52
11.5.3.31 pch_sch_test()	52
11.5.3.32 pch_sch_wait()	52
11.5.3.33 pch_sch_wait_timeout()	52
11.5.3.34 pch_test_pending_interruption()	53
11.5.3.35 void()	53
11.6 picochan_cu	53
11.6.1 Detailed Description	55
11.6.2 Macro Definition Documentation	55
11.6.2.1 MAX_DEVIB_CALLBACKS	55
11.6.2.2 NUM_DEVIB_CALLBACKS	55
11.6.2.3 PCH CU INIT	56
11.6.2.4 PCH_NUM_CUS	56
11.6.3 Typedef Documentation	56
11.6.3.1 pch_cu_t	56
11.6.3.2 pch_devib_t	56
11.6.4 Function Documentation	57
11.6.4.1 pch_cu_get_tx_channel()	57
11.6.4.2 pch_cu_init()	57
11.6.4.3 pch_cu_register()	57
11.6.4.4 pch_cu_start()	58
11.6.4.5 pch_cus_init()	58
11.6.4.6 pch_cus_memcu_configure()	58
11.6.4.7 pch_cus_set_trace()	58
11.6.4.8 pch_cus_trace_cu()	59
11.6.4.9 pch_cus_trace_dev()	59
11.6.4.10 pch_cus_uartcu_configure()	59
11.6.4.11 pch_dev_receive_then()	60
11.6.4.12 pch_dev_send_then()	60
11.6.4.13 pch_dev_send_zeroes_then()	62
11.6.4.14 pch_dev_set_callback()	62
11.6.4.15 pch_devib_prepare_callback()	62
11.6.4.16 pch_devib_prepare_count()	63
11.6.4.17 pch_devib_prepare_read_data()	63
11.6.4.18 pch_devib_prepare_update_status()	63
11.6.4.19 pch_devib_prepare_write_data()	64

11.6.4.20 pch_devib_prepare_write_zeroes()	64
11.6.4.21 pch_get_cu()	64
11.6.4.22 pch_get_devib()	64
11.6.4.23 pch_register_devib_callback()	65
11.6.4.24 pch_register_unused_devib_callback()	65
12 Data Structure Documentation	67
12.1 addr_count Struct Reference	67
12.2 css Struct Reference	67
12.2.1 Detailed Description	68
12.3 dmachan_1way_config Struct Reference	68
12.4 dmachan_cmd Union Reference	68
12.5 dmachan_config Struct Reference	68
12.6 dmachan_link Struct Reference	69
12.7 dmachan_rx_channel Struct Reference	69
12.8 dmachan_tx_channel Struct Reference	69
12.9 dmairqix_config Struct Reference	70
12.10 pch_bsize Struct Reference	70
12.10.1 Detailed Description	70
12.11 pch_bsizex Struct Reference	71
12.11.1 Detailed Description	71
12.12 pch_ccw Struct Reference	71
12.12.1 Detailed Description	71
12.13 pch_chp Struct Reference	72
12.13.1 Detailed Description	72
12.14 pch_cu Struct Reference	72
12.14.1 Detailed Description	73
12.14.2 Field Documentation	73
12.14.2.1 num_devibs	73
12.15 pch_dev_range Struct Reference	74
12.16 pch_dev_sense Struct Reference	74
12.16.1 Detailed Description	74
12.17 pch_devib Struct Reference	74
12.17.1 Detailed Description	75
12.18 pch_intcode Struct Reference	75
12.18.1 Detailed Description	75
12.19 pch_pmcw Struct Reference	76
12.19.1 Detailed Description	76
12.20 pch_schib Struct Reference	76
12.20.1 Detailed Description	77
12.21 pch_schib_mda Struct Reference	77
12.21.1 Detailed Description	78

12.22 pch_scsw Struct Reference	78
12.22.1 Detailed Description	78
12.23 pch_trc_bufferset Struct Reference	78
12.23.1 Detailed Description	79
12.23.2 Field Documentation	79
12.23.2.1 buffers	79
12.23.2.2 irqnum	79
12.24 pch_trc_header Struct Reference	80
12.25 pch_trc_timestamp Struct Reference	80
12.25.1 Detailed Description	80
12.26 pch_trdata_address_change Struct Reference	80
12.27 pch_trdata_byte Struct Reference	81
12.28 pch_trdata_ccw_addr_sid Struct Reference	81
12.29 pch_trdata_chp_alloc Struct Reference	81
12.30 pch_trdata_cu_register Struct Reference	81
12.31 pch_trdata_cus_call_callback Struct Reference	82
12.32 pch_trdata_cus_init_mem_channel Struct Reference	82
12.33 pch_trdata_cus_tx_complete Struct Reference	82
12.34 pch_trdata_dev Struct Reference	82
12.35 pch_trdata_dev_byte Struct Reference	83
12.36 pch_trdata_dma_init Struct Reference	83
12.37 pch_trdata_dmachan Struct Reference	83
12.38 pch_trdata_dmachan_memstate Struct Reference	83
12.39 pch_trdata_dmachan_segment Struct Reference	84
12.40 pch_trdata_dmachan_segment_memstate Struct Reference	84
12.41 pch_trdata_func_irq Struct Reference	84
12.42 pch_trdata_id_byte Struct Reference	84
12.43 pch_trdata_id_irq Struct Reference	85
12.44 pch_trdata_intcode_scsw Struct Reference	85
12.45 pch_trdata_irq_handler Struct Reference	85
12.46 pch_trdata_irqnum_opt Struct Reference	85
12.47 pch_trdata_scsw_sid_cc Struct Reference	86
12.48 pch_trdata_sid_byte Struct Reference	86
12.49 pch_trdata_word_byte Struct Reference	86
12.50 pch_trdata_word_dev Struct Reference	86
12.51 pch_trdata_word_sid Struct Reference	87
12.52 pch_trdata_word_sid_byte Struct Reference	87
12.53 pch_txsm Struct Reference	87
12.54 proto_packet Struct Reference	87
12.54.1 Detailed Description	88
12.55 proto_parsed_devstatus_payload Struct Reference	88
12.56 proto_payload Struct Reference	88

12.57 ua_slist Struct Reference	88
13 File Documentation	89
13.1 dmachan_internal.h	89
13.2 dmachan_trace.h	89
13.3 base/include/picochan/bsize.h File Reference	90
13.3.1 Detailed Description	91
13.3.2 Typedef Documentation	91
13.3.2.1 pch_bsize_t	91
13.4 bsize.h	92
13.5 base/include/picochan/ccw.h File Reference	93
13.5.1 Detailed Description	94
13.6 ccw.h	94
13.7 base/include/picochan/dev_status.h File Reference	95
13.7.1 Detailed Description	95
13.8 dev_status.h	96
13.9 dmachan.h	96
13.10 dmachan_defs.h	99
13.11 base/include/picochan/ids.h File Reference	100
13.12 ids.h	100
13.13 base/include/picochan/intcode.h File Reference	101
13.13.1 Typedef Documentation	101
13.13.1.1 pch_intcode_t	101
13.14 intcode.h	102
13.15 base/include/picochan/scsw.h File Reference	102
13.15.1 Typedef Documentation	103
13.15.1.1 pch_scsw_t	103
13.16 scsw.h	103
13.17 base/include/picochan/trc.h File Reference	104
13.17.1 Macro Definition Documentation	106
13.17.1.1 PCH_TRC_RT	106
13.18 trc.h	106
13.19 trc_record_types.h	107
13.20 trc_records.h	108
13.21 txsm_state.h	110
13.22 chop.h	110
13.23 base/proto/packet.h File Reference	111
13.24 packet.h	112
13.25 payload.h	113
13.26 bufferset.h	113
13.27 trace.h	114
13.28 trace_lock.h	115

13.29 txsm.h	115
13.30 ccw_fetch.h	116
13.31 css/channel.h File Reference	116
13.32 channel.h	117
13.33 css/css_internal.h File Reference	119
13.33.1 Variable Documentation	120
13.33.1.1 CSS	120
13.34 css_internal.h	121
13.35 css_trace.h	122
13.36 css/include/picochan/css.h File Reference	123
13.37 css.h	126
13.38 css/include/picochan/pmcw.h File Reference	128
13.38.1 Detailed Description	129
13.38.2 Typedef Documentation	129
13.38.2.1 pch_pmcw_t	129
13.39 pmcw.h	130
13.40 css/include/picochan/schib.h File Reference	130
13.40.1 Detailed Description	131
13.40.2 Typedef Documentation	131
13.40.2.1 pch_schib_mda_t	131
13.41 schib.h	131
13.42 schib_dlist.h	132
13.43 schib_internal.h	132
13.44 schibs_lock.h	133
13.45 callback.h	133
13.46 cu_internal.h	134
13.47 cus_trace.h	134
13.48 devibs_lock.h	135
13.49 cu/include/picochan/cu.h File Reference	136
13.50 cu.h	138
13.51 cu/include/picochan/dev_api.h File Reference	141
13.51.1 Detailed Description	143
13.51.2 Function Documentation	143
13.51.2.1 pch_dev_call_final_then()	143
13.51.2.2 pch_dev_call_or_reject_then()	143
13.52 dev_api.h	143
13.53 cu/include/picochan/dev_sense.h File Reference	145
13.53.1 Detailed Description	145
13.54 dev_sense.h	145
13.55 cu/include/picochan/devib.h File Reference	146
13.55.1 Detailed Description	147
13.56 devib.h	147

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 divisible 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 "susends" 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_INTERRUPT`) 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_TRANSFER_IN_CHANNEL`). (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 Interruption" 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, sscsw)` 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, sscsw)` 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 mis-classifications.

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 obvious 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_devibs 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_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` #defined 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 concatenated 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 contiguous (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_FLAG_TRACE` 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_FLAG_TRACE;
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_TRACE` flag bit in the `devib`, e.g. with `pch_devib_set_trace(devib, true)`. With the `PCH_DEVIB_FLAG_TRACE` 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:

picochan_base	33
internal_trc	38
internal_proto	39
internal_css	40
picochan_css	41
picochan_cu	53

Chapter 9

Data Structure Index

9.1 Data Structures

Here are the data structures with brief descriptions:

addr_count	67
css	
Struct css is a channel subsystem (CSS)	67
dmachan_1way_config	68
dmachan_cmd	68
dmachan_config	68
dmachan_link	69
dmachan_rx_channel	69
dmachan_tx_channel	69
dmairqix_config	70
pch_bszie	
8-bit structure whose value encodes a 16-bit value for use as a count of bytes in a typical pic- ochan buffer or transfer request	70
pch_bsizes	
Pch_bszie together with a flag intended to indicate whether the bsizes encoded the original size exactly	71
pch_ccw	
I/O Channel-Command Word (CCW)	71
pch_chp	
Pch_chp_t is the CSS-side representation of a channel path to a control unit	72
pch_cu	
Pch_cu_t is a Control Unit (CU)	72
pch_dev_range	
The device sense structure by which a device can communicate additional error information on request by the CSS	74
pch_devib	
Pch_devib_t represents a device on a CU	74
pch_intcode	
.	75
pch_pmcw	
.	76
pch_schib	
Pch_schib_t is the Subchannel Information Block (SCHIB)	76
pch_schib_mda	
The Model Dependent Area (MDA) of a schib	77
pch_scsw	
.	78

pch_trc_bufferset	Set of buffers and metadata for a subsystem to use tracing	78
pch_trc_header	80
pch_trc_timestamp	Opaque timestamp of a 48-bit number of microseconds since boot	80
pch_trdata_address_change	80
pch_trdata_byte	81
pch_trdata_ccw_addr_sid	81
pch_trdata_chp_alloc	81
pch_trdata_cu_register	81
pch_trdata_cus_call_callback	82
pch_trdata_cus_init_mem_channel	82
pch_trdata_cus_tx_complete	82
pch_trdata_dev	82
pch_trdata_dev_byte	83
pch_trdata_dma_init	83
pch_trdata_dmachan	83
pch_trdata_dmachan_memstate	83
pch_trdata_dmachan_segment	84
pch_trdata_dmachan_segment_memstate	84
pch_trdata_func_irq	84
pch_trdata_id_byte	84
pch_trdata_id_irq	85
pch_trdata_intcode_scsw	85
pch_trdata_irq_handler	85
pch_trdata_irqnum_opt	85
pch_trdata_scsw_sid_cc	86
pch_trdata_sid_byte	86
pch_trdata_word_byte	86
pch_trdata_word_dev	86
pch_trdata_word_sid	87
pch_trdata_word_sid_byte	87
pch_txsm	87
proto_packet	4-byte command packet sent on a channel between CSS and CU or vice versa	87
proto_parsed_devstatus_payload	88
proto_payload	88
ua_slist	88

Chapter 10

File Index

10.1 File List

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

base/dmachan/ dmachan_internal.h	89
base/dmachan/ dmachan_trace.h	89
base/include/picochan/ bsize.h An encoding of 16-bit counts as 8-bit values for typical Pico-sized buffers	90
base/include/picochan/ ccw.h Channel-Command Word (CCW)	93
base/include/picochan/ dev_status.h Device status bit values	95
base/include/picochan/ dmachan.h	96
base/include/picochan/ dmachan_defs.h	99
base/include/picochan/ ids.h	100
base/include/picochan/ intcode.h	101
base/include/picochan/ scsw.h	102
base/include/picochan/ trc.h	104
base/include/picochan/ trc_record_types.h	107
base/include/picochan/ trc_records.h	108
base/include/picochan/ txsm_state.h	110
base/proto/ chop.h	110
base/proto/ packet.h	111
base/proto/ payload.h	113
base/trc/ bufferset.h	113
base/trc/ trace.h	114
base/trc/ trace_lock.h	115
base/txsm/ txsm.h	115
css/ ccw_fetch.h	116
css/ channel.h	116
css/ css_internal.h	119
css/ css_trace.h	122
css/ schib_dlist.h	132
css/ schib_internal.h	132
css/ schibs_lock.h	133
css/include/picochan/ css.h	123
css/include/picochan/ pmcw.h The Path Management Control World (PMCW)	128
css/include/picochan/ schib.h The Subchannel Information Block (SCHIB)	130

cu/callback.h	133
cu/cu_internal.h	134
cu/cus_trace.h	134
cu/devibs_lock.h	135
cu/include/picochan/cu.h	136
cu/include/picochan/dev_api.h	
The main API for a device on a CU	141
cu/include/picochan/dev_sense.h	
Device sense	145
cu/include/picochan/devib.h	
The structures and API for a device on a CU	146

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_bsizex 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_bsizex_encode (uint16_t n)**
Encode 16-bit count as an pch_bsizex_t.
- **pch_bsizex_t pch_bsizex_encode (uint16_t n)**
Encode 16-bit count as an 8-bit pch_bsizex_t.
- **uint16_t pch_bsizex_decode_raw (uint8_t esize)**
Decode an 8-bit raw value of a bsize (not in its pch_bsizex_t type-wrapping) into a 16-bit value.
- **uint16_t pch_bsizex_decode (pch_bsizex_t bsize)**
Decode an 8-bit pch_bsizex_t value into a 16-bit value.
- **static uint8_t pch_bsizex_unwrap (pch_bsizex_t s)**
Unwraps the uint8_t contained in a pch_bsizex_t.
- **static pch_bsizex_t pch_bsizex_wrap (uint8_t esize)**
wraps a uint8_t into a pch_bsizex_t
- **static uint8_t pch_bsizex_encode_raw_inline (uint16_t n)**
Perform a bsize encoding, returning the encoded value unwrapped.
- **static pch_bsizex_t pch_bsizex_encodeInline (uint16_t n)**
encode a 16-bit value into its pch_bsizex_t along with an "exact"
- **static pch_bsizex_t pch_bsizex_encodeInline (uint16_t n)**
encode a 16-bit value as a pch_bsizex_t
- **static uint16_t pch_bsizex_decode_raw_inline (uint8_t esize)**
decodes a raw bsize-encoded value
- **static uint16_t pch_bsizex_decodeInline (pch_bsizex_t bsize)**
decodes a pch_bsizex_t as the uint16_t it represents
- **uint8_t pch_bsizex_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_bsizet){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_bsizet

```
typedef struct pch_bsizet pch_bsizet
```

a [pch_bsizet](#) 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
```

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
MDA	+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
	DEVS/ccwflags SCHS Residual Count
	+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
	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_bsiz_decode_inline()

```
uint16_t pch_bsiz_decode_inline (
    pch_bsiz_t bsize) [inline], [static]
```

decodes a `pch_bsiz_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_bsiz_encode` is available as an ordinary function.

11.1.4.2 pch_bsiz_decode_raw_inline()

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

decodes a raw bsize-encoded value

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

11.1.4.3 pch_bsiz_encode_inline()

```
pch_bsiz_t pch_bsiz_encode_inline (
    uint16_t n) [inline], [static]
```

encode a 16-bit value as a `pch_bsiz_t`

This does the same as `pch_bsiz_encode` but does not return the exactness.

11.1.4.4 pch_bsiz_encode_raw_inline()

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

Perform a bsize encoding, returning the encoded value unwrapped.

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

11.1.4.5 pch_bsizex_encode_ex_inline()

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

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

This encodes n into its `pch_bsizex_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_bsizex_encode` is available as an ordinary function.

11.1.4.6 pch_bsizex_wrap()

```
pch_bsizex_t pch_bsizex_wrap (
    uint8_t esize) [inline], [static]
```

wraps a `uint8_t` into a `pch_bsizex_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 all..

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 all..

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-NUL.
- void **pch_css_start** (**io_callback_t** io_callback, **uint8_t** isc_mask)

Starts CSS operation after setting the io_callback (if non-NUL), 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

<code>baudrate</code>	and <code>pch_chp_configure_uartchan</code> with <code>ctrl</code> argument bits taken from an appropriate <code>dma_channel_get_default_config()</code> value. The CU on the other side of the channel <i>MUST</i> 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, setting 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

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

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 nor 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 AchHaltPending 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_scs_w_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_scs_w_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 the 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

<i>cu</i>	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.
<i>num_devibs</i>	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

<i>cu</i>	the CU to register
<i>cua</i>	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

<i>cu</i>	- the control unit
<i>ua</i>	- the unit address of the device in control unit <i>cu</i>
<i>dstaddr</i>	- the address to receive the data sent by the CSS
<i>size</i>	- the number of data bytes requested - the number of bytes actually received will be at most <i>n</i> but may be strictly less.
<i>cbindex_opt</i>	- 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

<i>cu</i>	- the control unit
<i>ua</i>	- the unit address of the device in control unit <i>cu</i>
<i>flags</i>	<ul style="list-style-type: none"> - may contain the following flags: <ul style="list-style-type: none"> • 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.
<i>srcaddr</i>	- the address of the data to be sent (ignored if flags contains PROTO_CHOP_FLAG_SKIP)
<i>n</i>	- the number of data bytes to send
<i>cbindex_opt</i>	<ul style="list-style-type: none"> - 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: <ul style="list-style-type: none"> • 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

<i>cu</i>	the CU to which the device belongs
<i>ua</i>	the unit address of the device within its CU
<i>cbindex_opt</i>	<p>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:</p> <ul style="list-style-type: none"> • 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_UPDATE_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

```
00001 /*
00002  * Copyright (c) 2025 Malcolm Beattie
00003  * SPDX-License-Identifier: MIT
00004 */
00005
00006 #ifndef _PCH_DMACHAN_DMACHAN_INTERNAL_H
00007 #define _PCH_DMACHAN_DMACHAN_INTERNAL_H
00008
00009 #include "hardware/sync.h"
00010 #include "dmachan_trace.h"
00011
00012 // dmachan_mem_peer_spin_lock protects against test/update of
00013 // tx_channel.mem_src_state and rx_channel.mem_dst_state both
00014 // from interrupts and cross-core. It must be initialised before
00015 // use with pch_memchan_init().
00016 extern spin_lock_t *dmachan_mem_peer_spin_lock;
00017
00018 static inline uint32_t mem_peer_lock(void) {
00019 #if PCH_CONFIG_ENABLE_MEMCHAN
00020     return spin_lock_blocking(dmachan_mem_peer_spin_lock);
00021 #else
00022     return 0;
00023 #endif
00024 }
00025
00026 static inline void mem_peer_unlock(uint32_t saved_irq) {
00027 #if PCH_CONFIG_ENABLE_MEMCHAN
00028     spin_unlock(dmachan_mem_peer_spin_lock, saved_irq);
00029 #else
00030     (void)saved_irq;
00031 #endif
00032 }
00033
00034 #endif
```

13.2 dmachan_trace.h

```
00001 /*
00002  * Copyright (c) 2025 Malcolm Beattie
00003  * SPDX-License-Identifier: MIT
00004 */
00005
00006 #ifndef _PCH_DMACHAN_DMACHAN_TRACE_H
00007 #define _PCH_DMACHAN_DMACHAN_TRACE_H
00008
00009 #include "trc/trace.h"
00010 #include "picochan/trc_records.h"
00011
00012 #define PCH_DMACHAN_LINK_TRACE(rt, l, data) \
00013     PCH_TRC_WRITE(l->bs, l->bs, (rt), (data))
00014
00015 static inline void trace_dmachan(pch_trc_record_type_t rt, dmachan_link_t *l) {
00016     PCH_DMACHAN_LINK_TRACE(rt, l, ((struct pch_trdata_dmachan){
```

```

00017             .dmaid = l->dmaid
00018         });
00019     }
00020
00021 static inline void trace_dmachan_segment(pch_trc_record_type_t rt, dmachan_link_t *l, uint32_t addr,
00022     uint32_t count) {
00023     PCH_DMACHAN_LINK_TRACE(rt, l, ((struct pch_trdata_dmachan_segment){
00024         .addr = addr,
00025         .count = count,
00026         .dmaid = l->dmaid
00027     }));
00028
00029 static inline void trace_dmachan_memstate(pch_trc_record_type_t rt, dmachan_link_t *l, uint8_t state)
00030 {
00031     PCH_DMACHAN_LINK_TRACE(rt, l,
00032         ((struct pch_trdata_dmachan_memstate){
00033             .dmaid = l->dmaid,
00034             .state = state
00035         }));
00036
00037 static inline void trace_dmachan_segment_memstate(pch_trc_record_type_t rt, dmachan_link_t *l,
00038     uint32_t addr, uint32_t count, uint8_t state) {
00039     PCH_DMACHAN_LINK_TRACE(rt, l,
00040         ((struct pch_trdata_dmachan_segment_memstate){
00041             .addr = addr,
00042             .count = count,
00043             .dmaid = l->dmaid,
00044             .state = state
00045     ));
00046
00047 #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_BSIZZERO \(\(pch_bsize_t\){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_bsizex_encode (uint16_t n)**
Encode 16-bit count as an pch_bsizex_t.
- **pch_bsizex_t pch_bsizex_encode (uint16_t n)**
Encode 16-bit count as an 8-bit pch_bsizex_t.
- **uint16_t pch_bsizex_decode (uint8_t esize)**
Decode an 8-bit raw value of a bsize (not in its pch_bsizex_t type-wrapping) into a 16-bit value.
- **uint16_t pch_bsizex_decode (pch_bsizex_t bsizex)**
Decode an 8-bit pch_bsizex_t value into a 16-bit value.
- **static uint8_t pch_bsizex_unwrap (pch_bsizex_t s)**
Unwraps the uint8_t contained in a pch_bsizex_t.
- **static pch_bsizex_t pch_bsizex_wrap (uint8_t s)**
wraps a uint8_t into a pch_bsizex_t
- **static uint8_t pch_bsizex_encode_raw_inline (uint16_t n)**
Perform a bsize encoding, returning the encoded value unwrapped.
- **static pch_bsizex_t pch_bsizex_encodeex_inline (uint16_t n)**
encode a 16-bit value into its pch_bsizex_t along with an "exact"
- **static pch_bsizex_t pch_bsizex_encode_inline (uint16_t n)**
encode a 16-bit value as a pch_bsizex_t
- **static uint16_t pch_bsizex_decode_raw_inline (uint8_t esize)**
decodes a raw bsize-encoded value
- **static uint16_t pch_bsizex_decode_inline (pch_bsizex_t bsizex)**
decodes a pch_bsizex_t as the uint16_t it represents
- **uint8_t pch_bsizex_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_bsizex_t

```
typedef struct pch_bsizex pch_bsizex_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.](#)

```

00001 /*
00002  * Copyright (c) 2025 Malcolm Beattie
00003  * SPDX-License-Identifier: MIT
00004 */
00005
00006 #ifndef _PCH_API_BSIZE_H
00007 #define _PCH_API_BSIZE_H
00008
00009 #include <stdint.h>
00010
00016
00035 typedef struct pch_bsize {
00036     uint8_t esize;
00037 } pch_bsize_t;
00038
00046 #define PCH_BSIZE_ZERO ((pch_bsize_t){0})
00047
00055 typedef struct pch_bsizex {
00056     uint8_t exact;
00057     pch_bsize_t bsize;
00058 } pch_bsizex_t;
00059
00060 // Non-inlined API functions
00061
00065 pch_bsizex_t pch_bsize_encode(uint16_t n);
00066
00070 pch_bsize_t pch_bsize_encode(uint16_t n);
00071
00076 uint16_t pch_bsize_decode_raw(uint8_t esize);
00077
00082 uint16_t pch_bsize_decode(pch_bsize_t bsize);
00083
00084 // Inline encode/decode operations
00085
00089 static inline uint8_t pch_bsize_unwrap(pch_bsize_t s) {
00090     return s.esize;
00091 }
00092
00101 static inline pch_bsize_t pch_bsize_wrap(uint8_t esize) {
00102     return (pch_bsize_t){esize};
00103 }
00104
00112 static inline uint8_t pch_bsize_encode_raw_inline(uint16_t n) {
00113     // XXX TODO See if we can just call pch_bsize_encode_inline
00114     // and return the contained pch_bsize_t and have gcc
00115     // reliably optimise it as well as not calculating the
00116     // exact flag in the first place. For now we just spell it
00117     // all out again.
00118
00119     // 0b00nnnnnn - 1 x [0,63] -> 0,1,2,...,63
00120     if (n <= 63)
00121         return (uint8_t)n;
00122
00123     // 0b01nnnnnn - 2 x [32,95] -> 64,66,68,...,190
00124     if (n <= 191)
00125         return (uint8_t)((n>>1) - 32) | 0x40;
00126
00127     // 0b10nnnnnn - 8 x [24,87] -> 192,200,208,...,696
00128     if (n <= 703)
00129         return (uint8_t)((n>>3) - 24) | 0x80;
00130
00131     // 0b11nnnnnn - 64 x [11,74] -> 704,768,832,...,4736
00132     if (n <= 4736)
00133         return (uint8_t)((n>>6) - 11) | 0xc0;
00134
00135     return 0xff;
00136 }
00137
00149 static inline pch_bsizex_t pch_bsize_encodeex_inline(uint16_t n) {
00150     // 0b00nnnnnn - 1 x [0,63] -> 0,1,2,...,63
00151     if (n <= 63)
00152         return (pch_bsizex_t){1,pch_bsize_wrap(n)};
00153
00154     // 0b01nnnnnn - 2 x [32,95] -> 64,66,68,...,190
00155     if (n <= 191) {
00156         uint8_t exact = (n & 0x1) == 0;
00157         pch_bsize_t bsize = pch_bsize_wrap((n>>1) - 32) | 0x40;
00158         return (pch_bsizex_t){exact,bsize};
00159     }
00160
00161     // 0b10nnnnnn - 8 x [24,87] -> 192,200,208,...,696
00162     if (n <= 703) {

```

```

00163     uint8_t exact = (n & 0x7) == 0;
00164         pch_bsize_t bsize = pch_bsize_wrap((n>3) - 24) | 0x80;
00165     return (pch_bsizex_t){exact,bsize};
00166 }
00167
00168 // 0b11nnnnnn - 64 x [11,74] -> 704,768,832,...,4736
00169 if (n <= 4736) {
00170     uint8_t exact = (n & 0x3f) == 0;
00171         pch_bsize_t bsize = pch_bsize_wrap((n>6) - 11) | 0xc0;
00172     return (pch_bsizex_t){exact,bsize};
00173 }
00174
00175     return (pch_bsizex_t){0, pch_bsize_wrap(0xff)};
00176 }
00177
00184 static inline pch_bsize_t pch_bsize_encode_inline(uint16_t n) {
00185     return pch_bsize_wrap(pch_bsize_encode_raw_inline(n));
00186 }
00187
00195 static inline uint16_t pch_bsize_decode_raw_inline(uint8_t esize) {
00196     uint8_t flags = esize & 0xc0;
00197     uint16_t n = esize & 0x3f;
00198
00199     switch (flags) {
00200     case 0x00:
00201         // 0b00nnnnnn - 1 x [0,63] -> 0,1,2,...,63
00202         return n;
00203
00204     case 0x40:
00205         // 0b01nnnnnn - 2 x [32,95] -> 64,66,68,...,190
00206         return (n+32) << 1;
00207
00208     case 0x80:
00209         // 0b10nnnnnn - 8 x [24,87] -> 192,200,208,...,696
00210         return (n+24) << 3;
00211     }
00212
00213 // 0b11nnnnnn - 64 x [11,74] -> 704,768,832,...,4736
00214 return (n+11) << 6;
00215 }
00216
00224 static inline uint16_t pch_bsize_decode_inline(pch_bsize_t bsize) {
00225     return pch_bsize_decode_raw_inline(bsize.esize);
00226 }
00227
00231 uint8_t pch_bsize_encode_raw(uint16_t n);
00232
00233 #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.

```

00001 /*
00002  * Copyright (c) 2025 Malcolm Beattie
00003  * SPDX-License-Identifier: MIT
00004 */
00005
00006 #ifndef _PCH_API_CCW_H
00007 #define _PCH_API_CCW_H
00008
00009 #include <stdint.h>
00010 #include <assert.h>
00011
00017
00021 typedef uint8_t pch_ccw_flags_t;
00022
00023 // pch_ccw_flags_t: CCW flags
00024 // CD: Chain Data
00025 #define PCH_CCW_FLAG_CD          0x80
00026 // CC: Chain Command
00027 #define PCH_CCW_FLAG_CC          0x40
00028 // SLI: Suppress Length Indication
00029 #define PCH_CCW_FLAG_SLI         0x20
00030 // SKP: Skip/Discard data
00031 #define PCH_CCW_FLAG_SKP         0x10
00032 // PCI: Program Controlled Interruption
00033 #define PCH_CCW_FLAG_PCI         0x08
00034 // IDA: Indirect Data Address (not used in Picochan)
00035 #define PCH_CCW_FLAG_IDA         0x04
00036 // S: Suspend
00037 #define PCH_CCW_FLAG_S           0x02
00038 // MIDA: Modified Indirect Data Address (not used in Picochan)
00039 #define PCH_CCW_FLAG_MIDA        0x01
00040
00059 typedef struct __attribute__((aligned(4))) pch_ccw {

```

```

00060     uint8_t          cmd;
00061     pch_ccw_flags_t  flags;
00062     uint16_t         count;
00063     uint32_t         addr;
00064 } pch_ccw_t;
00065
00066 static_assert(sizeof(pch_ccw_t) == 8, "architected pch_ccw_t is 8 bytes");
00067
00068 // Architected values of CCW commands.
00069 // These do not match those for traditional CSS and we only divide
00070 // into "Read/Write" via the low bit instead of into Control/Read/
00071 // ReadBackward/Sense/Test/Write via various low-bit groups.
00072
00073 #define PCH_CCW_CMD_FIRST_RESERVED 0xf0
00074 // WRITE
00075 #define PCH_CCW_CMD_WRITE        0x01
00076 // READ
00077 #define PCH_CCW_CMD_READ         0x02
00078 // TIC: Transfer In Channel
00079 #define PCH_CCW_CMD_TIC          0xf0
00080 // SENSE: Read Sense data from devib
00081 #define PCH_CCW_CMD_SENSE        0xf2
00082
00083 // Architected bit tests of CCW commands
00084 static inline bool pch_is_ccw_cmd_write(uint8_t cmd) {
00085     return (cmd & 0x01) == 1;
00086 }
00087
00088 #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.](#)

```
00001 /*
00002  * Copyright (c) 2025 Malcolm Beattie
00003  * SPDX-License-Identifier: MIT
00004 */
00005
00019 #ifndef _PCH_DEV_STATUS_H
00020 #define _PCH_DEV_STATUS_H
00021
00022 #define PCH_DEVS_ATTENTION      0x80
00023 #define PCH_DEVS_STATUS_MODIFIER 0x40
00024 #define PCH_DEVS_CONTROL_UNIT_END 0x20
00025 #define PCH_DEVS_BUSY           0x10
00026 #define PCH_DEVS_CHANNEL_END    0x08
00027 #define PCH_DEVS_DEVICE_END     0x04
00028 #define PCH_DEVS_UNIT_CHECK     0x02
00029 #define PCH_DEVS_UNIT_EXCEPTION 0x01
00030
00031 #endif
```

13.9 dmachan.h

```
00001 /*
00002  * Copyright (c) 2025 Malcolm Beattie
00003  * SPDX-License-Identifier: MIT
00004 */
00005
00006 #ifndef _PCH_API_DMACHAN_H
00007 #define _PCH_API_DMACHAN_H
00008
00009 // PICO_CONFIG: PARAM_ASSERTIONS_ENABLED_PCH_DMACHAN, Enable/disable assertions in the pch_dmachan
00010 // module, type=bool, default=0, group=pch_dmachan
00010 #ifndef PARAM_ASSERTIONS_ENABLED_PCH_DMACHAN
00011 #define PARAM_ASSERTIONS_ENABLED_PCH_DMACHAN 0
00012 #endif
00013
00014 #ifndef PCH_CONFIG_ENABLE_MEMCHAN
00015 #define PCH_CONFIG_ENABLE_MEMCHAN 1
00016 #endif
00017
00018 #include "hardware/dma.h"
00019 #include "hardware/structs/dma_debug.h"
00020 #include "hardware/uart.h"
00021 #include "pico/platform/compiler.h"
00022 #include "picochan/dmachan_defs.h"
00023 #include "picochan/ids.h"
00024 #include "picochan/trc.h"
00025
00026 // General Pico SDK-like DMA-related functions that aren't in the SDK
00027 static inline enum dma_channel_transfer_size channel_config_get_transfer_data_size(dma_channel_config
config) {
00028     uint size = (config.ctrl & DMA_CH0_CTRL_TRIG_DATA_SIZE_BITS) »
00028 DMA_CH0_CTRL_TRIG_DATA_SIZE_LSB;
00029     return (enum dma_channel_transfer_size)size;
00030 }
00031
00032 static inline uint32_t dma_channel_get_transfer_count(uint channel) {
00033     return dma_channel_hw_addr(channel)->transfer_count;
00034 }
00035
00036 static inline dma_debug_channel_hw_t *dma_debug_channel_hw_addr(uint channel) {
00037     check_dma_channel_param(channel);
00038     return &dma_debug_hw->ch[channel];
00039 }
00040
00041 static inline uint32_t dma_channel_get_reload_count(uint channel) {
00042     return dma_debug_channel_hw_addr(channel)->dbg_tcr;
00043 }
00044
00045 static inline bool dma_irqn_get_channel_forced(uint irq_index, uint channel) {
00046     invalid_params_if(HARDWARE_DMA, irq_index >= NUM_DMA IRQS);
00047     check_dma_channel_param(channel);
00048
00049     return dma_hw->irq_ctrl[irq_index].intf & (lu « channel);
00050 }
00051
00052 static inline void dma_irqn_set_channel_forced(uint irq_index, uint channel, bool forced) {
00053     invalid_params_if(HARDWARE_DMA, irq_index >= NUM_DMA IRQS);
```

```

00054         if (forced)
00055             hw_set_bits(&dma_hw->irq_ctrl[irq_index].intf, lu << channel);
00056         else
00057             hw_clear_bits(&dma_hw->irq_ctrl[irq_index].intf, lu << channel);
00058     }
00059 }
00060
00061 // DMA configuration for one direction (tx or rx) of a dmachan channel
00062 typedef struct dmachan_lway_config {
00063     uint32_t           addr;
00064     dma_channel_config ctrl;
00065     pch_dmaid_t        dmaid;
00066     pch_dma_irq_index_t dmairqix;
00067 } dmachan_lway_config_t;
00068
00069 static inline dmachan_lway_config_t dmachan_lway_config_make(pch_dmaid_t dmaid, uint32_t addr,
00070     dma_channel_config ctrl, pch_dma_irq_index_t dmairqix) {
00071     return ((dmachan_lway_config_t){
00072         .addr = addr,
00073         .ctrl = ctrl,
00074         .dmaid = dmaid,
00075         .dmairqix = dmairqix
00076     });
00077 }
00078 static inline dmachan_lway_config_t dmachan_lway_config_claim(uint32_t addr, dma_channel_config ctrl,
00079     pch_dma_irq_index_t dmairqix) {
00080     pch_dmaid_t dmaid = (pch_dmaid_t)dma_claim_unused_channel(true);
00081     return dmachan_lway_config_make(dmaid, addr, ctrl, dmairqix);
00082 }
00083 static inline dmachan_lway_config_t dmachan_lway_config_memchan_make(pch_dmaid_t dmaid,
00084     pch_dma_irq_index_t dmairqix) {
00085     dma_channel_config ctrl = dma_channel_get_default_config(dmaid);
00086     channel_config_set_transfer_data_size(&ctrl, DMA_SIZE_8);
00087     channel_config_set_read_increment(&ctrl, true);
00088     channel_config_set_write_increment(&ctrl, true);
00089     return ((dmachan_lway_config_t){
00090         .addr = 0,
00091         .ctrl = ctrl,
00092         .dmaid = dmaid,
00093         .dmairqix = dmairqix
00094     });
00095 }
00096 // DMA configuration for both directions (tx and rx) of a dmachan
00097 // channel
00098 typedef struct dmachan_config {
00099     dmachan_lway_config_t tx;
00100     dmachan_lway_config_t rx;
00101 } dmachan_config_t;
00102
00103 static inline dmachan_config_t dmachan_config_claim(uint32_t txaddr, dma_channel_config txctrl,
00104     uint32_t rxaddr, dma_channel_config rxctrl, pch_dma_irq_index_t dmairqix) {
00105     return ((dmachan_config_t){
00106         .tx = dmachan_lway_config_claim(txaddr, txctrl, dmairqix),
00107         .rx = dmachan_lway_config_claim(rxaddr, rxctrl, dmairqix)
00108     });
00109 }
00110 static inline dmachan_config_t dmachan_config_memchan_make(pch_dmaid_t txdmaid, pch_dmaid_t rxdmaid,
00111     pch_dma_irq_index_t dmairqix) {
00112     return ((dmachan_config_t){
00113         .tx = dmachan_lway_config_memchan_make(txdmaid, dmairqix),
00114         .rx = dmachan_lway_config_memchan_make(rxdmaid, dmairqix)
00115     });
00116 }
00117 typedef union __aligned(4) dmachan_cmd {
00118     unsigned char    buf[4];
00119     uint32_t         raw;
00120 } dmachan_cmd_t;
00121
00122 #define DMACHAN_CMD_SIZE sizeof(dmachan_cmd_t)
00123 static_assert(DMACHAN_CMD_SIZE == 4, "dmachan_cmd_t must be 4 bytes");
00124
00125 static inline void dmachan_cmd_set_zero(dmachan_cmd_t *cmd) {
00126     cmd->raw = 0;
00127 }
00128
00129 static inline void dmachan_cmd_copy(dmachan_cmd_t *dst, dmachan_cmd_t *src) {
00130     dst->raw = src->raw;
00131 }
00132
00133 #define DMACHAN_CMD_COPY(cmdp, p) do { \
00134     dmachan_cmd_t *__cmdp = (cmdp); \
00135     static_assert(sizeof(*(p)) == DMACHAN_CMD_SIZE, \

```

```

00136             "DMACHAN_CMD_COPY sizeof(*p) must be DMACHAN_CMD_SIZE"); \
00137             __cmdp->raw = *(uint32_t*)(p); \
00138 } while (0)
00139
00140 // dmachan_link_t collects the common fields in tx and rx channels
00141 typedef struct __aligned(4) dmachan_link {
00142     dmachan_cmd_t           cmd;
00143     pch_trc_bufferset_t    *bs;           // only when tracing
00144     pch_dmaid_t            dmaid;
00145     pch_dma_irq_index_t   dmairqix;
00146     bool                   complete;
00147     bool                   resetting;
00148 } dmachan_link_t;
00149
00150 #define DMACHAN_LINK_CMD_COPY(l, p) do { \
00151     dmachan_link_t *__l = (l); \
00152     DMACHAN_CMD_COPY(&__l->cmd, p); \
00153 } while (0)
00154
00155 static inline void dmachan_set_link_bs(dmachan_link_t *l, pch_trc_bufferset_t *bs) {
00156     l->bs = bs;
00157 }
00158
00159 static inline void dmachan_link_cmd_set_zero(dmachan_link_t *l) {
00160     dmachan_cmd_set_zero(&l->cmd);
00161 }
00162
00163 static inline void dmachan_link_cmd_copy(dmachan_link_t *dst, dmachan_link_t *src) {
00164     dmachan_cmd_copy(&dst->cmd, &src->cmd);
00165 }
00166
00167 static inline void dmachan_set_link_irq_enabled(dmachan_link_t *l, bool enabled) {
00168     pch_dma_irq_index_t dmairqix = l->dmairqix;
00169     assert(dmairqix >= 0 && dmairqix < NUM_DMA IRQS);
00170     dma_irqn_set_channel_enabled(dmairqix, l->dmaid, enabled);
00171 }
00172
00173 static inline bool dmachan_link_irq_raised(dmachan_link_t *l) {
00174     return dma_irqn_get_channel_status(l->dmairqix, l->dmaid);
00175 }
00176
00177 static inline bool dmachan_get_link_irq_forced(dmachan_link_t *l) {
00178     return dma_irqn_get_channel_forced(l->dmairqix, l->dmaid);
00179 }
00180
00181 static inline void dmachan_set_link_irq_forced(dmachan_link_t *l, bool forced) {
00182     dma_irqn_set_channel_forced(l->dmairqix, l->dmaid, forced);
00183 }
00184
00185 static inline void dmachan_ack_link_irq(dmachan_link_t *l) {
00186     dma_irqn_acknowledge_channel(l->dmairqix, l->dmaid);
00187 }
00188
00189 // tx and rx channels, starting with forward declarations because
00190 // for memchans there is a field pointing at the peer channel
00191 typedef struct __aligned(4) dmachan_tx_channel dmachan_tx_channel_t;
00192 typedef struct __aligned(4) dmachan_rx_channel dmachan_rx_channel_t;
00193
00194 typedef struct __aligned(4) dmachan_tx_channel {
00195     dmachan_link_t           link;
00196     dmachan_rx_channel_t    *mem_rx_peer; // only for memchan
00197     dmachan_mem_src_state_t mem_src_state; // only for memchan
00198 } dmachan_tx_channel_t;
00199
00200 typedef struct __aligned(4) dmachan_rx_channel {
00201     dmachan_link_t           link;
00202     dmachan_tx_channel_t    *mem_tx_peer; // only for memchan
00203     uint32_t                 srcaddr;
00204     dma_channel_config       ctrl;
00205     dmachan_mem_dst_state_t mem_dst_state; // only for memchan
00206 } dmachan_rx_channel_t;
00207
00208 static inline dmachan_irq_state_t dmachan_make_irq_state(bool raised, bool forced, bool complete) {
00209     return ((dmachan_irq_state_t)raised)
00210         | ((dmachan_irq_state_t)forced) << 1
00211         | ((dmachan_irq_state_t)complete) << 2;
00212 }
00213
00214 // tx channel irq and memory source state handling
00215 static inline void dmachan_set_mem_src_state(dmachan_tx_channel_t *tx, dmachan_mem_src_state_t
00216 new_state) {
00217     valid_params_if(PCH_DMACHAN,
00218                     new_state == DMACHAN_MEM_SRC_IDLE
00219                     || tx->mem_src_state == DMACHAN_MEM_SRC_IDLE);
00220     tx->mem_src_state = new_state;
00221 }

```

```

00222
00223 dmachan_irq_state_t dmachan_handle_tx_irq(dmachan_tx_channel_t *tx);
00224
00225 // rx channel irq and memory destination state handling
00226 static inline void dmachan_set_mem_dst_state(dmachan_rx_channel_t *rx, dmachan_mem_dst_state_t
00227 new_state) {
00228     valid_params_if(PCH_DMACHAN,
00229                     new_state == DMACHAN_MEM_DST_IDLE
00230                     || rx->mem_dst_state == DMACHAN_MEM_DST_IDLE);
00231     rx->mem_dst_state = new_state;
00232 }
00233
00234 dmachan_irq_state_t dmachan_handle_rx_irq(dmachan_rx_channel_t *rx);
00235
00236 void dmachan_panic_unless_memchan_initialised(void);
00237
00238 void dmachan_init_tx_channel(dmachan_tx_channel_t *tx, dmachan_1way_config_t *cfg);
00239 void dmachan_start_src_cmdbuf(dmachan_tx_channel_t *tx);
00240 void dmachan_write_src_reset(dmachan_tx_channel_t *tx);
00241 void dmachan_start_src_data(dmachan_tx_channel_t *tx, uint32_t srcaddr, uint32_t count);
00242
00243 void dmachan_init_rx_channel(dmachan_rx_channel_t *rx, dmachan_1way_config_t *cfg);
00244 void dmachan_start_dst_reset(dmachan_rx_channel_t *rx);
00245 void dmachan_start_dst_cmdbuf(dmachan_rx_channel_t *rx);
00246 void dmachan_start_dst_data(dmachan_rx_channel_t *rx, uint32_t dstaddr, uint32_t count);
00247 void dmachan_start_dst_discard(dmachan_rx_channel_t *rx, uint32_t count);
00248 void dmachan_start_dst_data_src_zeroes(dmachan_rx_channel_t *rx, uint32_t dstaddr, uint32_t count);
00249
00250 // Convenience functions for configuring UART channels
00251 void pch_uart_init(uart_inst_t *uart, uint baudrate);
00252
00253 static inline dma_channel_config dmachan_uart_make_txctrl(uart_inst_t *uart, dma_channel_config ctrl)
00254 {
00255     channel_config_set_transfer_data_size(&ctrl, DMA_SIZE_8);
00256     channel_config_set_write_increment(&ctrl, false);
00257     uint txdreq = uart_get_dreq_num(uart, true);
00258     channel_config_set_dreq(&ctrl, txdreq);
00259 }
00260
00261 static inline dma_channel_config dmachan_uart_make_rxctrl(uart_inst_t *uart, dma_channel_config ctrl)
00262 {
00263     channel_config_set_transfer_data_size(&ctrl, DMA_SIZE_8);
00264     channel_config_set_read_increment(&ctrl, false);
00265     uint rxdreq = uart_get_dreq_num(uart, false);
00266     channel_config_set_dreq(&ctrl, rxdreq);
00267 }
00268
00269 // pch_memchan_init must be called before configuring either side of
00270 // any memchan CU with pch_cus_memcu_configure or
00271 // pch_chp_configure_memchan
00272 void pch_memchan_init(void);
00273
00274 #endif

```

13.10 dmachan_defs.h

```

00001 /*
00002  * Copyright (c) 2025 Malcolm Beattie
00003  * SPDX-License-Identifier: MIT
00004 */
00005
00006 #ifndef _PCH_API_DMACHAN_DEFS_H
00007 #define _PCH_API_DMACHAN_DEFS_H
00008
00009 // dmachan_mem_src_state_t is the DMA state of a tx channel
0010 typedef enum __attribute__((packed)) dmachan_mem_src_state {
0011     DMACHAN_MEM_SRC_IDLE = 0,
0012     DMACHAN_MEM_SRC_CMDBUF,
0013     DMACHAN_MEM_SRC_DATA
0014 } dmachan_mem_src_state_t;
0015
0016 // dmachan_mem_dst_state_t is the DMA state of an rx channel
0017 typedef enum __attribute__((packed)) dmachan_mem_dst_state {
0018     DMACHAN_MEM_DST_IDLE = 0,
0019     DMACHAN_MEM_DST_CMDBUF,
0020     DMACHAN_MEM_DST_DATA,
0021     DMACHAN_MEM_DST_DISCARD,
0022     DMACHAN_MEM_DST_SRC_ZEROES
0023 } dmachan_mem_dst_state_t;
0024

```

```

00025 // dmachan_irq_state_t represents the state of a given DMA id
00026 // with respect to an interrupt for a given DMA IRQ number.
00027 // REASON_RAISED means either there was a DMA engine completion
00028 // causing the bit for the DMA id to be set in register INTSn for
00029 // that DMA IRQ index or, apparently, the INTFn forced bit was set
00030 // explicitly.
00031 // REASON_FORCED means the bit for the DMA id was explicitly set in
00032 // register INTFn for that DMA IRQ index, ignoring the value of the
00033 // enable bit in the corresponding INTEn register. It also seems to
00034 // cause the corresponding INTSn bit to be seen as 1 too so
00035 // REASON_RAISED will (always?) be set if REASON_FORCED is.
00036 // COMPLETE is the value of the link's complete field at the end of
00037 // the dmachan_handle_tx_irq and dmachan_handle_rx_irq functions:
00038 // it will be 1 if either the RAISED or FORCED conditions hold or if
00039 // the field was set explicitly beforehand as a way of causing
00040 // completion handling locally without an irq being triggered.
00041 typedef uint8_t dmachan_irq_state_t;
00042 #define DMACHAN_IRQ_REASON_RAISED      0x1
00043 #define DMACHAN_IRQ_REASON_FORCED     0x2
00044 #define DMACHAN_IRQ_COMPLETE        0x4
00045
00046 #define DMACHAN_IRQ_REASON_MASK      0x3
00047
00048 // ASCII 'C' is the byte we look for when resetting a dmachan
00049 // receive link so that we can resynchronise by dropping any zero
00050 // bytes that are generated by Break conditions from the sender.
00051 #define DMACHAN_RESET_BYTE 0x43
00052
00053 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.](#)

```

00001 /*
00002  * Copyright (c) 2025 Malcolm Beattie
00003  * SPDX-License-Identifier: MIT
00004 */
00005
00006 #ifndef _PCH_API_IDS_H
00007 #define _PCH_API_IDS_H
00008

```

```

00009 #include <stdint.h>
00010
00016
00020 typedef uint16_t pch_sid_t;
00021
00026 typedef uint8_t pch_cuaddr_t;
00027
00034 typedef uint8_t pch_unit_addr_t;
00035
00042 typedef uint8_t pch_chpid_t;
00043
00052 typedef uint8_t pch_dmaid_t;
00053
00064 typedef int8_t pch_dma_irq_index_t;
00065
00066 #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.](#)

```
00001 /*
00002  * Copyright (c) 2025 Malcolm Beattie
00003  * SPDX-License-Identifier: MIT
00004 */
00005 #ifndef _PCH_API_INTCODE_H
00006 #define _PCH_API_INTCODE_H
00007
00008 #include "picochan/ids.h"
00009
00016
00039 typedef struct pch_intcode {
00040     uint32_t      intparm;
00041     pch_sid_t    sid;
00042     uint8_t       flags;
00043     uint8_t       cc;
00044 } pch_intcode_t;
00045 static_assert(sizeof(pch_intcode_t) == 8,
00046                 "architected pch_intcode_t is 8 bytes");
00047
00048 #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_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.](#)

```
00001 /*
00002 * Copyright (c) 2025 Malcolm Beattie
00003 * SPDX-License-Identifier: MIT
00004 */
00005
00006 #ifndef _PCH_API_SCSW_H
00007 #define _PCH_API_SCSW_H
00008
00009 #include <stdint.h>
00010 #include <assert.h>
00011
00017
00018 #define PCH_SF_CC_MASK 0xc0
```

```

00019 #define PCH_SF_CC_SHIFT 6
00020 #define PCH_SF_P      0x20
00021 #define PCH_SF_I      0x10
00022 #define PCH_SF_U      0x08
00023 #define PCH_SF_Z      0x04
00024 #define PCH_SF_UNUSED 0x02
00025 #define PCH_SF_N      0x01
00026
00027 // uint16_t of SCSW Control flags W,FC,AC,SC (Function, Activity, Status)
00028 #define PCH_SCSW_CCW_WRITE 0x8000
00029
00030 #define PCH_FC_MASK      0x7000
00031 #define PCH_FC_START     0x4000
00032 #define PCH_FC_HALT     0x2000
00033 #define PCH_FC_CLEAR    0x1000
00034
00035 #define PCH_AC_MASK      0x0fe0
00036 #define PCH_AC_RESUME_PENDING 0x0800
00037 #define PCH_AC_START_PENDING 0x0400
00038 #define PCH_AC_HALT_PENDING 0x0200
00039 #define PCH_AC_CLEAR_PENDING 0x0100
00040 #define PCH_AC_SUBCHANNEL_ACTIVE 0x0080
00041 #define PCH_AC_DEVICE_ACTIVE 0x0040
00042 #define PCH_AC_SUSPENDED   0x0020
00043
00044 #define PCH_SC_MASK      0x001f
00045 #define PCH_SC_ALERT     0x0010
00046 #define PCH_SC_INTERMEDIATE 0x0008
00047 #define PCH_SC_PRIMARY   0x0004
00048 #define PCH_SC_SECONDARY 0x0002
00049 #define PCH_SC_PENDING   0x0001
00050
00051 // uint8_t of Subchannel Status (SCHS) flags
00052 #define PCH_SCHS_PROGRAM_CONTROLLED INTERRUPTION 0x80
00053 #define PCH_SCHS_INCORRECT_LENGTH 0x40
00054 #define PCH_SCHS_PROGRAM_CHECK 0x20
00055 #define PCH_SCHS_PROTECTION_CHECK 0x10
00056 #define PCH_SCHS_CHANNEL_DATA_CHECK 0x08
00057 #define PCH_SCHS_CHANNEL_CONTROL_CHECK 0x04
00058 #define PCH_SCHS_INTERFACE_CONTROL_CHECK 0x02
00059 #define PCH_SCHS_CHAINING_CHECK 0x01
00060
00080 typedef struct __attribute__((aligned(4))) pch_scsw {
00081     uint8_t __unused_flags;
00082     uint8_t user_flags;
00083     uint16_t ctrl_flags;
00084     uint32_t ccw_addr;
00085     uint8_t devs;
00086     uint8_t schs;
00087     uint16_t count;
00088 } pch_scsw_t;
00089 static_assert(sizeof(pch_scsw_t) == 12, "architected pch_scsw_t is 12 bytes");
00090
00091 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 all.
- `#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_TRADED , 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_TRADED , PCH_TRC_RT_CUS CU_STARTED ,`
- `PCH_TRC_RT_CUS CU IRQ , PCH_TRC_RT_CUS DEV_TRADED , 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.](#)

```
00001 /*
00002  * Copyright (c) 2025 Malcolm Beattie
00003  * SPDX-License-Identifier: MIT
00004 */
00005
00006 #ifndef _PCH_API_TRC_H
00007 #define _PCH_API_TRC_H
00008
00009 #include "picochan/ids.h"
00010
00016
00025 typedef struct pch_trc_timestamp {
00026     uint16_t low;
00027     uint16_t mid;
00028     uint16_t high;
00029 } pch_trc_timestamp_t;
00030
00031 static inline uint64_t pch_trc_timestamp_to_us(pch_trc_timestamp_t t) {
00032     return (((uint64_t) t.high) « 32)
00033         + (((uint64_t) t.mid) « 16)
00034         + (((uint64_t) t.low));
00035 }
00036
00037 static inline void pch_trc_write_timestamp(pch_trc_timestamp_t *tp, uint64_t us) {
00038     uint16_t *hp = (uint16_t*)tp;
00039     hp[0] = (uint16_t)us; // low 16 bits: t.low
00040     hp[1] = (uint16_t)(us » 16); // middle 16 bits: t.mid
00041     hp[2] = (uint16_t)(us » 32); // low 16 bits of top 32: t.high
00042 }
00043
00044 // The following macro definition nastiness allows a host-based
00045 // trace dump program to redefine these macros to build a list
00046 // of the record type names along with the enum values themselves.
00047 #define PCH_TRC_RT(rt) PCH_TRC_RT_ ## rt
00048
00049 typedef enum __attribute__((__packed__)) pch_trc_record_type {
00050 #include "picochan/trc_record_types.h"
00051 } pch_trc_record_type_t;
00052
00053 typedef struct __attribute__((__packed__,__aligned__(2))) pch_trc_header {
00054     pch_trc_timestamp_t timestamp;
00055     uint8_t size; // includes header and following data
00056     pch_trc_record_type_t rec_type;
00057 } pch_trc_header_t;
00058
00065 #ifndef PCH_CONFIG_ENABLE_TRACE
00066 #define PCH_CONFIG_ENABLE_TRACE 0
00067 #endif
00068
00069 #if PCH_CONFIG_ENABLE_TRACE
00070
```

```

00071 #ifndef PCH_TRC_BUFFER_SIZE
00072 #define PCH_TRC_BUFFER_SIZE 1024
00073 #endif
00080
00081 #ifndef PCH_TRC_NUM_BUFFERS
00082 #define PCH_TRC_NUM_BUFFERS 2
00083 #endif
00090
00095 extern uint32_t pch_trc_buffer_size;
00096
00101 extern uint32_t pch_trc_num_buffers;
00102
00103 #else
00104 // PCH_CONFIG_ENABLE_TRACE is not defined
00105
00106 #ifndef PCH_TRC_BUFFER_SIZE
00107 #define PCH_TRC_BUFFER_SIZE 0
00108 #endif
00109
00110 #ifndef PCH_TRC_NUM_BUFFERS
00111 #define PCH_TRC_NUM_BUFFERS 1
00112 #endif
00113
00114 #endif
00115 // end of PCH_CONFIG_ENABLE_TRACE section
00116
00133 typedef struct pch_trc_bufferset {
00135     uint32_t      current_buffer_num;
00136
00139     uint32_t      current_buffer_pos;
00140
00147     int16_t       irqnum;
00148
00152     bool          enable;
00153
00156     uint32_t      magic;
00157     uint32_t      buffer_size;
00158     uint16_t      num_buffers;
00167     void          *buffers[PCH_TRC_NUM_BUFFERS];
00168 } pch_trc_bufferset_t;
00169
00170 #endif

```

13.19 trc_record_types.h

```

00001 /*
00002  * Copyright (c) 2025 Malcolm Beattie
00003  * SPDX-License-Identifier: MIT
00004 */
00005
00006 PCH_TRC_RT(INVALID),
00007 PCH_TRC_RT(CSS_SCH_START),
00008 PCH_TRC_RT(CSS_SCH_RESUME),
00009 PCH_TRC_RT(CSS_SCH_TEST),
00010 PCH_TRC_RT(CSS_SCH MODIFY),
00011 PCH_TRC_RT(CSS_SCH_STORE),
00012 PCH_TRC_RT(CSS_SCH_CANCEL),
00013 PCH_TRC_RT(CSS_SCH_CLEAR),
00014 PCH_TRC_RT(CSS_SCH_HALT),
00015 PCH_TRC_RT(CSS_CCW_FETCH),
00016 PCH_TRC_RT(CSS_CHP_ALLOC),
00017 PCH_TRC_RT(CSS_CHP_TX_DMA_INIT),
00018 PCH_TRC_RT(CSS_CHP_RX_DMA_INIT),
00019 PCH_TRC_RT(CSS_CHP_CONFIGURED),
00020 PCH_TRC_RT(CSS_CHP_TRADED),
00021 PCH_TRC_RT(CSS_CHP_STARTED),
00022 PCH_TRC_RT(CSS_CHP_IRQ),
00023 PCH_TRC_RT(CSS_RX_COMMAND_COMPLETE),
00024 PCH_TRC_RT(CSS_RX_DATA_COMPLETE),
00025 PCH_TRC_RT(CSS_SEND_TX_PACKET),
00026 PCH_TRC_RT(CSS_TX_COMPLETE),
00027 PCH_TRC_RT(CSS_CORE_NUM),
00028 PCH_TRC_RT(CSS_SET_DMA IRQ),
00029 PCH_TRC_RT(CSS_SET_FUNC IRQ),
00030 PCH_TRC_RT(CSS_SET_IO IRQ),
00031 PCH_TRC_RT(CSS_SET_IO_CALLBACK),
00032 PCH_TRC_RT(CSS_NOTIFY),
00033 PCH_TRC_RT(CSS_FUNC IRQ),
00034 PCH_TRC_RT(CSS_IO_CALLBACK),
00035 PCH_TRC_RT(CSS_INIT_DMA IRQ_HANDLER),
00036 PCH_TRC_RT(CSS_INIT_FUNC IRQ_HANDLER),
00037 PCH_TRC_RT(CSS_INIT_IO IRQ_HANDLER),
00038 PCH_TRC_RT(CUS_QUEUE_COMMAND),

```

```

00039 PCH_TRC_RT(CUS_INIT_DMA IRQ_HANDLER),
00040 PCH_TRC_RT(CUS CU REGISTER),
00041 PCH_TRC_RT(CUS CU TX DMA INIT),
00042 PCH_TRC_RT(CUS CU RX DMA INIT),
00043 PCH_TRC_RT(CUS CU CONFIGURED),
00044 PCH_TRC_RT(CUS CU TRACED),
00045 PCH_TRC_RT(CUS CU STARTED),
00046 PCH_TRC_RT(CUS CU IRQ),
00047 PCH_TRC_RT(CUS DEV TRACED),
00048 PCH_TRC_RT(CUS SEND TX PACKET),
00049 PCH_TRC_RT(CUS TX COMPLETE),
00050 PCH_TRC_RT(CUS REGISTER CALLBACK),
00051 PCH_TRC_RT(CUS CALL CALLBACK),
00052 PCH_TRC_RT(CUS RX COMMAND COMPLETE),
00053 PCH_TRC_RT(CUS RX DATA COMPLETE),
00054 PCH_TRC_RT(DMACHAN DST CMDBUF REMOTE),
00055 PCH_TRC_RT(DMACHAN DST CMDBUF MEM),
00056 PCH_TRC_RT(DMACHAN DST RESET REMOTE),
00057 PCH_TRC_RT(DMACHAN DST RESET MEM),
00058 PCH_TRC_RT(DMACHAN DST DATA REMOTE),
00059 PCH_TRC_RT(DMACHAN DST DATA MEM),
00060 PCH_TRC_RT(DMACHAN DST DISCARD REMOTE),
00061 PCH_TRC_RT(DMACHAN DST DISCARD MEM),
00062 PCH_TRC_RT(DMACHAN SRC CMDBUF REMOTE),
00063 PCH_TRC_RT(DMACHAN SRC CMDBUF MEM),
00064 PCH_TRC_RT(DMACHAN SRC RESET REMOTE),
00065 PCH_TRC_RT(DMACHAN SRC RESET MEM),
00066 PCH_TRC_RT(DMACHAN SRC DATA REMOTE),
00067 PCH_TRC_RT(DMACHAN SRC DATA MEM),
00068 PCH_TRC_RT(TRC_ENABLE)

```

13.20 trc_records.h

```

00001 /*
00002  * Copyright (c) 2025 Malcolm Beattie
00003  * SPDX-License-Identifier: MIT
00004 */
00005
00006 #ifndef _PCH_API_TRC_RECORDS_H
00007 #define _PCH_API_TRC_RECORDS_H
00008
00009 #include "picochan/ids.h"
00010 #include "picochan/ccw.h"
00011 #include "picochan/scsw.h"
00012 #include "picochan/intcode.h"
00013
00014 // Common structs for the data parts of trace records
00015
00016 struct pch_trdata_byte {
00017     uint8_t byte;
00018 };
00019
00020 struct pch_trdata_id_byte {
00021     uint8_t id;
00022     uint8_t byte;
00023 };
00024
00025 struct pch_trdata_irq_handler {
00026     uint32_t handler;
00027     int16_t order_priority; // -1 for exclusive
00028     uint8_t irqnum;
00029 };
00030
00031 struct pch_trdata_cu_register {
00032     uint16_t num_devices;
00033     pch_cuaddr_t cuaddr;
00034 };
00035
00036 struct pch_trdata_id_irq {
00037     uint8_t id;
00038     pch_dma_irq_index_t dmairqix;
00039     uint8_t tx_state;
00040     uint8_t rx_state;
00041 };
00042
00043 struct pch_trdata_dev {
00044     pch_cuaddr_t cuaddr;
00045     pch_unit_addr_t ua;
00046 };
00047
00048 struct pch_trdata_dev_byte {
00049     pch_cuaddr_t cuaddr;
00050     pch_unit_addr_t ua;

```

```
00051     uint8_t      byte;
00052 };
00053
00054 struct pch_trdata_word_dev {
00055     uint32_t      word;
00056     pch_cuaddr_t  cuaddr;
00057     pch_unit_addr_t ua;
00058 };
00059
00060 struct pch_trdata_word_sid_byte {
00061     uint32_t      word;
00062     pch_sid_t     sid;
00063     uint8_t       byte;
00064 };
00065
00066 struct pch_trdata_word_byte {
00067     uint32_t      word;
00068     uint8_t       byte;
00069 };
00070
00071 struct pch_trdata_word_sid {
00072     uint32_t      word;
00073     pch_sid_t     sid;
00074 };
00075
00076 struct pch_trdata_sid_byte {
00077     pch_sid_t     sid;
00078     uint8_t       byte;
00079 };
00080
00081 struct pch_trdata_ccw_addr_sid {
00082     pch_ccw_t     ccw;
00083     uint32_t      addr;
00084     pch_sid_t     sid;
00085 };
00086
00087 struct pch_trdata_intcode_scsw {
00088     pch_intcode_t intcode;
00089     pch_scsw_t    scsw;
00090 };
00091
00092 struct pch_trdata_scsw_sid_cc {
00093     pch_scsw_t    scsw;
00094     pch_sid_t     sid;
00095     uint8_t       cc;
00096 };
00097
00098 struct pch_trdata_dma_init {
00099     uint32_t      addr;
00100    uint32_t      ctrl;
00101    uint8_t       id;
00102    pch_dmaid_t   dmaid;
00103    pch_dma_irq_index_t dmairqix;
00104    uint8_t       core_num;
00105 };
00106
00107 struct pch_trdata_chp_alloc {
00108     pch_sid_t     first_sid;
00109     uint16_t      num_devices;
00110     pch_chpid_t   chpid;
00111 };
00112
00113 struct pch_trdata_irqnum_opt {
00114     int16_t       irqnum_opt;
00115 };
00116
00117 struct pch_trdata_address_change {
00118     uint32_t      old_addr;
00119     uint32_t      new_addr;
00120 };
00121
00122 struct pch_trdata_func_irq {
00123     int16_t       ua_opt;
00124     pch_chpid_t   chpid;
00125     uint8_t       tx_active;
00126 };
00127
00128 struct pch_trdata_cus_init_mem_channel {
00129     pch_cuaddr_t  cuaddr;
00130     pch_dmaid_t   txdmaid;
00131     pch_dmaid_t   rxdmaid;
00132 };
00133
00134 struct pch_trdata_cus_tx_complete {
00135     int16_t       uaopt;
00136     pch_cuaddr_t  cuaddr;
00137     uint8_t       txpstate;
```

```

00138 };
00139
00140 struct pch_trdata_cus_call_callback {
00141     pch_cuaddr_t cuaddr;
00142     pch_unit_addr_t ua;
00143     uint8_t cbindex;
00144 };
00145
00146 struct pch_trdata_dmachan {
00147     pch_dmaid_t dmaid;
00148 };
00149
00150 struct pch_trdata_dmachan_memstate {
00151     pch_dmaid_t dmaid;
00152     uint8_t state;
00153 };
00154
00155 struct pch_trdata_dmachan_segment {
00156     uint32_t addr;
00157     uint32_t count;
00158     pch_dmaid_t dmaid;
00159 };
00160
00161 struct pch_trdata_dmachan_segment_memstate {
00162     uint32_t addr;
00163     uint32_t count;
00164     pch_dmaid_t dmaid;
00165     uint8_t state;
00166 };
00167
00168 #endif

```

13.21 txsm_state.h

```

00001 /*
00002  * Copyright (c) 2025 Malcolm Beattie
00003  * SPDX-License-Identifier: MIT
00004 */
00005
00006 #ifndef _PCH_API_TXSM_STATE_H
00007 #define _PCH_API_TXSM_STATE_H
00008
00009 typedef enum __attribute__((packed)) pch_txsm_state {
0010     PCH_TXSM_IDLE = 0,
0011     PCH_TXSM_PENDING,
0012     PCH_TXSM_SENDING
0013 } pch_txsm_state_t;
0014
0015 typedef enum __attribute__((packed)) pch_txsm_run_result {
0016     PCH_TXSM_NOOP = 0,
0017     PCH_TXSM_ACTED,
0018     PCH_TXSM_FINISHED
0019 } pch_txsm_run_result_t;
0020
0021 #endif

```

13.22 chop.h

```

00001 /*
00002  * Copyright (c) 2025 Malcolm Beattie
00003  * SPDX-License-Identifier: MIT
00004 */
00005
00006 #ifndef _PCH_PROTO_CHOP_H
00007 #define _PCH_PROTO_CHOP_H
00008
00009 // proto_chop_t represents a channel operation in a packet sent
00010 // between CSS and CU in either direction.
00011 // It is 8 bits with the top 4 as flag bits (with only 3 currently
00012 // in use) and the bottom 4 as the operation command itself.
00013 // The meaning of the flag bits depends on the operation command.
00014 typedef uint8_t proto_chop_t;
00015
00016 typedef enum __attribute__((packed)) proto_chop_cmd {
00017     PROTO_CHOP_START          = 0,
00018     PROTO_CHOP_ROOM           = 1,
00019     PROTO_CHOP_DATA           = 2,
00020     PROTO_CHOP_UPDATE_STATUS  = 3,

```

```

00021     PROTO_CHOP_REQUEST_READ      = 4,
00022     PROTO_CHOP_HALT             = 5
00023 } proto_chop_cmd_t;
00024 static_assert(sizeof(proto_chop_cmd_t) == 1, "proto_chop_cmd_t must be 1 byte");
00025
00026 typedef uint8_t proto_chop_flags_t;
00027
00028 // PROTO_CHOP_FLAG_SKIP is valid in CSS -> CU Room, Data and Start
00029 // and in CU -> CSS Data
00030 #define PROTO_CHOP_FLAG_SKIP    0x80
00031
00032 // PROTO_CHOP_FLAG_END is valid in CSS <-> CU Data
00033 #define PROTO_CHOP_FLAG_END     0x40
00034
00035 // PROTO_CHOP_FLAG_STOP is valid in CSS -> CU Data
00036 #define PROTO_CHOP_FLAG_STOP    0x20
00037
00038 // PROTO_CHOP_FLAG_RESPONSE_REQUIRED is valid in CU -> CSS Data
00039 #define PROTO_CHOP_FLAG_RESPONSE_REQUIRED 0x20
00040
00041 static inline proto_chop_flags_t proto_chop_flags(proto_chop_t c) {
00042     return (proto_chop_flags_t)(c & 0xf0);
00043 }
00044
00045 static inline proto_chop_cmd_t proto_chop_cmd(proto_chop_t c) {
00046     return (proto_chop_cmd_t)(c & 0x0f);
00047 }
00048
00049 #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.](#)

```

00001 /*
00002  * Copyright (c) 2025 Malcolm Beattie
00003  * SPDX-License-Identifier: MIT
00004 */
00005
00006 #ifndef _PCH_PROTO_PACKET_H
00007 #define _PCH_PROTO_PACKET_H
00008
00009 #include <assert.h>
00010 #include "chop.h"
00011 #include "payload.h"
00012 #include "picochan/ids.h"
00013 #include "picochan/bsize.h"
00014
00020
00030 typedef struct __attribute__((aligned(4))) proto_packet {
00031     proto_chop_t    chop;
00032     pch_unit_addr_t unit_addr;
00033     uint8_t         p0;
00034     uint8_t         p1;
00035 } proto_packet_t;
00036
00037 static_assert(sizeof(proto_packet_t) == 4, "proto_packet_t must be 4 bytes");
00038
00039 static inline proto_payload_t proto_get_payload(proto_packet_t p) {
00040     return ((proto_payload_t){p.p0, p.p1});
00041 }
00042
00043 static inline uint32_t proto_packet_as_word(proto_packet_t p) {
00044     return *(uint32_t *)p;
00045 }
00046
00047 // proto_get_count parses the payload of the packet as a 2-byte
00048 // big-endian value
00049 static inline uint16_t proto_get_count(proto_packet_t p) {
00050     return ((uint16_t)p.p0 « 8) + (uint16_t)p.p1; // big endian
00051 }
00052
00053 // proto_decode_esize_payload decodes the second byte of the payload
00054 // of the packet (p.p1), treating it as a pch_decode_bsize and using
00055 // pch_bsize_decode_raw to return the resulting count.
00056 static inline uint16_t proto_decode_esize_payload(proto_packet_t p) {
00057     return pch_bsize_decode_raw(p.p1);
00058 }
00059
00060 static inline proto_packet_t proto_make_packet(proto_chop_t chop, pch_unit_addr_t ua, proto_payload_t
00061     payload) {
00062     return ((proto_packet_t){
00063         .chop = chop,
00064         .unit_addr = ua,
00065         .p0 = payload.p0,
00066         .p1 = payload.p1
00067     });
00068
00069 static inline proto_packet_t proto_make_count_packet(proto_chop_t chop, pch_unit_addr_t ua, uint16_t
00070     count) {
00071     return ((proto_packet_t){
00072         .chop = chop,
00073         .unit_addr = ua,
00074         .p0 = count / 256, // big-endian: high byte
00075         .p1 = count % 256 // big-endian: low byte
00076     });
00077
00078 static inline proto_packet_t proto_make_esize_packet(proto_chop_t chop, pch_unit_addr_t ua, uint8_t
00079     p0, pch_bsize_t esize) {
00080     return ((proto_packet_t){
00081         .chop = chop,
00082         .unit_addr = ua,
00083         .p0 = p0,
00084         .p1 = pch_bsize_unwrap(esize)
00085     });
00086
00087 #endif

```

13.25 payload.h

```

00001 /*
00002  * Copyright (c) 2025 Malcolm Beattie
00003  * SPDX-License-Identifier: MIT
00004 */
00005
00006 #ifndef _PCH_PROTO_PAYLOAD_H
00007 #define _PCH_PROTO_PAYLOAD_H
00008
00009 #include "picochan/bsize.h"
00010
00011 // proto_payload_t is a 2-byte channel operation payload. It can be a
00012 // count, a pair of bytes "ccwcmd", "esize" for START-like or a
00013 // byte of device status followed by an (optional) advertised write
00014 // window esize for a device status update operation. A payload of a
00015 // uint16_t is decoded as big endian.
00016 typedef struct proto_payload {
00017     uint8_t p0;
00018     uint8_t p1;
00019 } proto_payload_t;
00020
00021 // proto_parse_count_payload parses the payload as a 2-byte
00022 // big-endian value
00023 static inline uint16_t proto_parse_count_payload(proto_payload_t p) {
00024     return ((uint16_t)p.p0 << 8) + (uint16_t)p.p1; // big endian
00025 }
00026
00027 static inline uint8_t proto_parse_devstatus_payload_devs(proto_payload_t p) {
00028     return p.p0;
00029 }
00030
00031 static inline pch_bsize_t proto_parse_devstatus_payload_esize(proto_payload_t p) {
00032     return pch_bsize_wrap(p.p1);
00033 }
00034
00035 struct proto_parsed_devstatus_payload {
00036     uint16_t count;
00037     uint8_t devs;
00038 };
00039
00040 static inline proto_payload_t proto_make_count_payload(uint16_t count) {
00041     // big-endian encoding
00042     return ((proto_payload_t){
00043         .p0 = (uint8_t)(count / 256),
00044         .p1 = (uint8_t)(count % 256)
00045     });
00046 }
00047
00048 proto_payload_t proto_make_devstatus_payload(uint8_t devs, pch_bsize_t esize);
00049 proto_payload_t proto_make_start_payload(uint8_t ccwcmd, pch_bsize_t esize);
00050 struct proto_parsed_devstatus_payload proto_parse_devstatus_payload(proto_payload_t p);
00051
00052#endif

```

13.26 bufferset.h

```

00001 /*
00002  * Copyright (c) 2025 Malcolm Beattie
00003  * SPDX-License-Identifier: MIT
00004 */
00005
00006 #ifndef _PCH_TRC_BUFFERSET_H
00007 #define _PCH_TRC_BUFFERSET_H
00008
00009 #include "hardware/irq.h"
00010 #include "assert.h"
00011 #include "trace_lock.h"
00012
00013 // pch_trc_init_bufferset initialises the bufferset by filling in
00014 // the num_buffers, buffer_size and magic fields and zeroing out the
00015 // other fields
00016 void pch_trc_init_bufferset(pch_trc_bufferset_t *bs, uint32_t magic);
00017
00018 // pch_trc_init_buffer initialises buffer index n to buf.
00019 static inline void pch_trc_init_buffer(pch_trc_bufferset_t *bs, uint n, void *buf) {
00020     valid_params_if(PCH_TRC, n < PCH_TRC_NUM_BUFFERS);
00021     valid_params_if(PCH_TRC, ((uint32_t)buf & 0x3) == 0);
00022     bs->buffers[n] = buf;
00023 }
00024
00025 // pch_trc_init_contiguous_buffers initialises all buffers of bs

```

```

00026 // to be pointers to the PCH_TRC_NUM_BUFFERS consecutive
00027 // PCH_BUFFER_SIZE-bytes-sized buffers in the contiguous space
00028 // in buf. buf must therefore be a pointer to at least
00029 // PCH_TRC_NUM_BUFFERS*PCH_BUFFER_SIZE available bytes.
00030 void pch_trc_init_all_buffers(pch_trc_bufferset_t *bs, void *buf);
00031
00032 // pch_trc_switch_to_next_buffer_unsafe is for internal use.
00033 // The external API is pch_trc_switch_to_next_buffer which takes
00034 // the trace_lock and then calls this with a 0 for position.
00035 // Internally, this is used when allocating a slot for a new trace
00036 // record (which has already taken trace_lock) and in that situation
00037 // it is often called with a non-zero pos following the
00038 // newly-allocated trace record.
00039 static inline unsigned char *pch_trc_switch_to_next_buffer_unsafe(pch_trc_bufferset_t *bs, uint32_t
pos) {
00040     bs->current_buffer_num = (bs->current_buffer_num + 1) % PCH_TRC_NUM_BUFFERS;
00041     bs->current_buffer_pos = pos;
00042     if (bs->irqnum > -1)
00043         irq_set_pending((irq_num_t)(bs->irqnum));
00044
00045     return bs->buffers[bs->current_buffer_num];
00046 }
00047
00048 // pch_trc_switch_to_next_buffer switches to the next trace buffer in
00049 // the bufferset. If bs->irqnum is non-negative, that IRQ is raised.
00050 // When the IRQ is raised, current_buffer_num has already been
00051 // incremented (modulo PCH_TRC_NUM_BUFFERS) and a trace record may be
00052 // in the process of writing to the new buffer. The IRQ handler will
00053 // typically want to start copying or sending the contents of
00054 // bs->buffers[bs->current_buffer_num-1] elsewhere and aim for
00055 // completion before the trace records fill remaining buffers and
00056 // wrap back around to overwrite that buffer.
00057 static inline unsigned char *pch_trc_switch_to_next_buffer(pch_trc_bufferset_t *bs) {
00058     uint32_t status = trace_lock();
00059     unsigned char *rec = pch_trc_switch_to_next_buffer_unsafe(bs, 0);
00060     trace_unlock(status);
00061     return rec;
00062 }
00063
00064 #endif

```

13.27 trace.h

```

00001 /*
00002  * Copyright (c) 2025 Malcolm Beattie
00003  * SPDX-License-Identifier: MIT
00004 */
00005
00006 #ifndef _PCH_TRC_TRACE_H
00007 #define _PCH_TRC_TRACE_H
00008
00009 // PICO_CONFIG: PARAM_ASSERTIONS_ENABLED_PCH_TRC, Enable/disable assertions in the pch_trc module,
00010 // type=bool, default=0, group=pch_trc
00011 #ifndef PARAM_ASSERTIONS_ENABLED_PCH_TRC
00012 #define PARAM_ASSERTIONS_ENABLED_PCH_TRC 0
00013 #endif
00014 #include <stddef.h>
00015 #include <string.h>
00016 #include "picochan/trc.h"
00017 #include "picochan/trc_records.h"
00018 #include "bufferset.h"
00019
00020 static_assert(sizeof(pch_trc_timestamp_t) == 6,
00021     "pch_trc_timestamp_t must be 6 bytes");
00022
00023 static_assert(sizeof(pch_trc_header_t) == 8,
00024     "pch_trc_header_t must be 8 bytes");
00025
00026 void *pch_trc_write_uncond(pch_trc_bufferset_t *bs, pch_trc_record_type_t rt, uint8_t data_size);
00027
00028 // pch_trc_write allocates and writes the header of a trace record
00029 // with the current timestamp and record type rt and returns a pointer
00030 // to the location where data_size bytes of associated trace should be
00031 // written. It returns NULL (without writing any header or taking any
00032 // other action) if no trace record should be written. This will be
00033 // the case if tracing was disabled globally at compile time
00034 // (PCH_CONFIG_ENABLE_TRACE was not defined or defined as 0) or if
00035 // tracing has been disabled (perhaps temporarily) at runtime by
00036 // setting pch_trc_enable to false or if the cond function argument is
00037 // false.
00038 static inline void *pch_trc_write(pch_trc_bufferset_t *bs, bool cond, pch_trc_record_type_t rt,
00039     uint8_t data_size) {

```

```

00039 #if PCH_CONFIG_ENABLE_TRACE
00040     if (!bs->enable // per-bufferset runtime tracing flag not enabled
00041         || !cond) // per-function-call condition flag not enabled
00042         return NULL;
00043
00044     return pch_trc_write_uncond(bs, rt, data_size);
00045 #else
00046     (void)bs;
00047     (void)cond;
00048     (void)rt;
00049     (void)data_size;
00050 #endif
00051     return NULL;
00052 }
00053
00054 #define PCH_TRC_WRITE(bs, cond, rt, data) do { \
00055     size_t __data_size = sizeof (data); \
00056     void *__rec = pch_trc_write((bs), (cond), (rt), __data_size); \
00057     if (__rec) \
00058         memcpy(__rec, &(data), __data_size); \
00059     } while (0)
00060
00061 bool pch_trc_set_enable(pch_trc_bufferset_t *bs, bool enable);
00062
00063 #endif

```

13.28 trace_lock.h

```

00001 /*
00002  * Copyright (c) 2025 Malcolm Beattie
00003  * SPDX-License-Identifier: MIT
00004 */
00005
00006 #ifndef _PCH_TRC_TRACE_LOCK_H
00007 #define _PCH_TRC_TRACE_LOCK_H
00008
00009 #include "hardware/sync.h"
00010
00011 static inline uint32_t trace_lock(void) {
00012     return save_and_disable_interrupts();
00013 }
00014
00015 static inline void trace_unlock(uint32_t status) {
00016     restore_interrupts(status);
00017 }
00018
00019 #endif

```

13.29 txsm.h

```

00001 /*
00002  * Copyright (c) 2025 Malcolm Beattie
00003  * SPDX-License-Identifier: MIT
00004 */
00005
00006 #ifndef _PCH_TXSM_PENDING_XFER_H
00007 #define _PCH_TXSM_PENDING_XFER_H
00008
00009 // PICO_CONFIG: PARAM_ASSERTIONS_ENABLED_PCH_TXSM, Enable/disable assertions in the pch_txsm module,
00010 // type=bool, default=0, group=pch_txsm
00011 #ifndef PARAM_ASSERTIONS_ENABLED_PCH_TXSM
00012 #define PARAM_ASSERTIONS_ENABLED_PCH_TXSM 0
00013 #endif
00014
00015 #include <stdint.h>
00016 #include "picochan/dmachan.h"
00017 #include "picochan/txsm_state.h"
00018
00019 // txsm provides a state machine that manages using a
00020 // dmachan_tx_channel to transmit a data buffer, driven by
00021 // tx completion handler calls.
00022
00023 // +-----+-----+-----+-----+-----+
00024 // |           |   flags   |       count    |
00025 // +-----+-----+-----+-----+-----+-----+
00026 // |           |           addr      |
00027 // +-----+-----+-----+-----+-----+-----+

```

```

00028
00029 typedef struct pch_txsm {
00030     pch_txsm_state_t state;
00031     uint16_t count;
00032     uint32_t addr;
00033 } pch_txsm_t;
00034
00035 // pch_txsm_busy returns whether px is non-Idle (i.e. it returns true
00036 // if and only if px is in state Pending or Sending).
00037 static inline bool pch_txsm_busy(pch_txsm_t *px) {
00038     return px->state != PCH_TXSM_IDLE;
00039 }
00040
00041 // Reset resets the state to Idle but does not change any
00042 // owner, addr or count set by SetPending
00043 static inline void pch_txsm_reset(pch_txsm_t *px) {
00044     px->state = PCH_TXSM_IDLE;
00045 }
00046
00047 // pch_txsm_set_pending stashies (addr, count) in px and moves its
00048 // state from Idle to Pending. It panics if px is Busy.
00049 static inline void pch_txsm_set_pending(pch_txsm_t *px, uint32_t addr, uint16_t count) {
00050     valid_params_if(PCH_TXSM, px->state == PCH_TXSM_IDLE);
00051
00052     px->state = PCH_TXSM_PENDING;
00053     px->addr = addr;
00054     px->count = count;
00055 }
00056
00057 enum pch_txsm_run_result pch_txsm_run(pch_txsm_t *px, dmachan_tx_channel_t *txch);
00058
00059 #endif

```

13.30 ccw_fetch.h

```

00001 /*
00002  * Copyright (c) 2025 Malcolm Beattie
00003  * SPDX-License-Identifier: MIT
00004 */
00005
00006 #ifndef _PCH_CSS_CCW_FETCH_H
00007 #define _PCH_CSS_CCW_FETCH_H
00008
00009 void fetch_chain_data_ccw(pch_schib_t *schib);
00010
00011 uint8_t fetch_first_command_ccw(pch_schib_t *schib);
00012
00013 uint8_t fetch_resume_ccw(pch_schib_t *schib);
00014
00015 uint8_t fetch_chain_ccw(pch_schib_t *schib);
00016
00017 uint8_t fetch_chain_command_ccw(pch_schib_t *schib);
00018
00019 #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_DLST** ((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 *)
- void **push_ua_dlist_unsafe** (ua_dlist_t *, pch_chp_t *chp, pch_schib_t *schib)
- static void **push_ua_dlist** (ua_dlist_t *, pch_chp_t *chp, pch_schib_t *schib)
- pch_schib_t * **remove_from_ua_dlist_unsafe** (ua_dlist_t *, pch_chp_t *chp, pch_unit_addr_t ua)
- static pch_schib_t * **remove_from_ua_dlist** (ua_dlist_t *, pch_chp_t *chp, pch_unit_addr_t ua)
- static pch_schib_t * **pop_ua_dlist_unsafe** (ua_dlist_t *, pch_chp_t *chp)
- static pch_schib_t * **pop_ua_dlist** (ua_dlist_t *, pch_chp_t *chp)
- static ua_slist_t **make_ua_slist** (void)
- static void **reset_ua_slist** (ua_slist_t *)
- pch_schib_t * **pop_ua_slist_unsafe** (ua_slist_t *, pch_chp_t *chp)
- static pch_schib_t * **pop_ua_slist** (ua_slist_t *, pch_chp_t *chp)
- bool **push_ua_slist_unsafe** (ua_slist_t *, pch_chp_t *chp, pch_sid_t sid)
- static bool **push_ua_slist** (ua_slist_t *, 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.](#)

```

00001 /*
00002  * Copyright (c) 2025 Malcolm Beattie
00003  * SPDX-License-Identifier: MIT
00004 */
00005
00006 #ifndef _PCH_CSS_CHANNEL_H
00007 #define _PCH_CSS_CHANNEL_H
00008
00009 #include <stdint.h>
00010 #include <stdbool.h>
00011 #include "css_internal.h"
00012 #include "txsm/txsm.h"
00013 #include "proto/packet.h"
00014
00020
00021 // ua_dlist_t is the head of a circular double-linked list of schibs
00022 // which all belong to the same channel, linked by the prevua/nextua
00023 // fields of schib.mda. It is the unit_addr_t of the head of the list
00024 // or else -1 if the list is empty.
00025 typedef int16_t ua_dlist_t;
00026
00027 // ua_slist_t is the head and tail of a single-linked list of schibs
00028 // which all belong to the same channel, linked by the nextua field of

```

```

00029 // schib.mda. It contains the unit_addr_t of the head and tail of
00030 // the list or else both fields are -1 if the list is empty.
00031 typedef struct ua_slist {
00032     int16_t head;
00033     int16_t tail;
00034 } ua_slist_t;
00035
00047 typedef struct __aligned(4) pch_chp {
00048     dmachan_tx_channel_t tx_channel;
00049     dmachan_rx_channel_t rx_channel;
00050     pch_txsm_t tx_pending;
00051     pch_sid_t first_sid;
00052     uint16_t num_devices; // [0, 256]
00053     // rx_data_for_ua: rx dma is active writing to CCW for this ua
00054     int16_t rx_data_for_ua;
00055     // rx_data_end_ds: if non-zero then, when rx data complete,
00056     // treat as an immediate implicit device status for update_status
00057     uint8_t rx_data_end_ds;
00058     // TODO combine following bools into a uint8_t flags
00059     // rx_response_required: when rx data complete, peer wants response
00060     bool rx_response_required;
00061     bool traced;
00062     bool claimed;
00063     bool allocated;
00064     bool configured;
00065     bool started;
00066     // tx_active: tx dma is active
00067     bool tx_active;
00068     // ua_func_dlist: links via schib.prevua and .nextua
00069     ua_dlist_t ua_func_dlist;
00070     // ua_response_slist: link via schib.nextua
00071     ua_slist_t ua_response_slist;
00072 } pch_chp_t;
00073
00074 //
00075 // ua_dlist
00076 //
00077 #define EMPTY_UA_DLST ((ua_dlist_t)-1)
00078
00079 static inline ua_dlist_t make_ua_dlist(void) {
00080     return EMPTY_UA_DLST;
00081 }
00082
00083 static inline int16_t peek_ua_dlist(ua_dlist_t *l) {
00084     return (int16_t)*l;
00085 }
00086
00087 void push_ua_dlist_unsafe(ua_dlist_t *l, pch_chp_t *chp, pch_schib_t *schib);
00088
00089 static inline void push_ua_dlist(ua_dlist_t *l, pch_chp_t *chp, pch_schib_t *schib) {
00090     uint32_t status = schibs_lock();
00091     push_ua_dlist_unsafe(l, chp, schib);
00092     schibs_unlock(status);
00093 }
00094
00095 pch_schib_t *remove_from_ua_dlist_unsafe(ua_dlist_t *l, pch_chp_t *chp, pch_unit_addr_t ua);
00096
00097 static inline pch_schib_t *remove_from_ua_dlist(ua_dlist_t *l, pch_chp_t *chp, pch_unit_addr_t ua) {
00098     uint32_t status = schibs_lock();
00099     pch_schib_t *schib = remove_from_ua_dlist_unsafe(l, chp, ua);
00100     schibs_unlock(status);
00101     return schib;
00102 }
00103
00104 static inline pch_schib_t *pop_ua_dlist_unsafe(ua_dlist_t *l, pch_chp_t *chp) {
00105     if (*l == -1)
00106         return NULL;
00107
00108     return remove_from_ua_dlist_unsafe(l, chp, (pch_unit_addr_t)*l);
00109 }
00110
00111 static inline pch_schib_t *pop_ua_dlist(ua_dlist_t *l, pch_chp_t *chp) {
00112     uint32_t status = schibs_lock();
00113     pch_schib_t *schib = pop_ua_dlist_unsafe(l, chp);
00114     schibs_unlock(status);
00115     return schib;
00116 }
00117
00118 //
00119 // ua_slist
00120 //
00121
00122 static inline ua_slist_t make_ua_slist(void) {
00123     return ((ua_slist_t){-1, -1});
00124 }
00125
00126 static inline void reset_ua_slist(ua_slist_t *l) {

```

```

00127     l->head = -1;
00128     l->tail = -1;
00129 }
00130
00131 pch_schib_t *pop_ua_slist_unsafe(ua_slist_t *l, pch_chp_t *chp);
00132
00133 static inline pch_schib_t *pop_ua_slist(ua_slist_t *l, pch_chp_t *chp) {
00134     uint32_t status = schibs_lock();
00135     pch_schib_t *schib = pop_ua_slist_unsafe(l, chp);
00136     schibs_unlock(status);
00137     return schib;
00138 }
00139
00140 bool push_ua_slist_unsafe(ua_slist_t *l, pch_chp_t *chp, pch_sid_t sid);
00141
00142 static inline bool push_ua_slist(ua_slist_t *l, pch_chp_t *chp, pch_sid_t sid) {
00143     uint32_t status = schibs_lock();
00144     bool was_empty = push_ua_slist_unsafe(l, chp, sid);
00145     schibs_unlock(status);
00146     return was_empty;
00147 }
00148
00149 // popping from and pushing to the channel ua_response_slist of schibs
00150 // with response packets pending to be sent to their CUs
00151 static inline pch_schib_t *pop_ua_response_slist(pch_chp_t *chp) {
00152     return pop_ua_slist(&chp->ua_response_slist, chp);
00153 }
00154
00155 static inline void push_ua_response_slist(pch_chp_t *chp, pch_sid_t sid) {
00156     push_ua_slist(&chp->ua_response_slist, chp, sid);
00157 }
00158
00159 //
00160 // getting packets to/from the channel command buffers
00161 //
00162 static inline proto_packet_t get_rx_packet(pch_chp_t *chp) {
00163     // chp.rx_channel is a dmachan_rx_channel_t which is
00164     // __aligned(4) and cmd is the first member of rx_channel
00165     // so is 4-byte aligned. proto_packet_t is 4-bytes and also
00166     // __aligned(4) (and needing no more than 4-byte alignment)
00167     // but omitting the __builtin_assume_aligned below causes
00168     // gcc 14.1.0 to produce error
00169     // error: cast increases required alignment of target type
00170     // [-Werror=cast-align]
00171     proto_packet_t *pp = (proto_packet_t *)
00172         __builtin_assume_aligned(&chp->rx_channel.link.cmd, 4);
00173     return *pp;
00174 }
00175
00176 static inline proto_packet_t get_tx_packet(pch_chp_t *chp) {
00177     // chp.tx_channel is a dmachan_tx_channel_t which is the
00178     // first member of chp which is a pch_chp_t which is
00179     // __aligned(4) and cmd is the first member of tx_channel
00180     // so is 4-byte aligned. proto_packet_t is 4-bytes and also
00181     // __aligned(4) (and needing no more than 4-byte alignment)
00182     // but omitting the __builtin_assume_aligned below causes
00183     // gcc 14.1.0 to produce error
00184     // error: cast increases required alignment of target type
00185     // [-Werror=cast-align]
00186     proto_packet_t *pp = (proto_packet_t *)
00187         __builtin_assume_aligned(&chp->tx_channel.link.cmd, 4);
00188     return *pp;
00189 }
00190
00191 void send_tx_packet(pch_chp_t *chp, proto_packet_t p);
00192
00193 #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 iscnm)`
- 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 iscnm)`
- void `remove_from_isc_dlist (uint8_t iscnm, 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.](#)

```

00001 /*
00002  * Copyright (c) 2025 Malcolm Beattie
00003  * SPDX-License-Identifier: MIT
00004 */
00005
00006 #ifndef _PCH_CSS_CSS_INTERNAL_H
00007 #define _PCH_CSS_CSS_INTERNAL_H
00008
00009 // PICO_CONFIG: PARAM_ASSERTIONS_ENABLED_PCH_CSS, Enable/disable assertions in the pch_css module,
00010 type=bool, default=0, group=pch_css
00011 #ifndef PARAM_ASSERTIONS_ENABLED_PCH_CSS
00012 #define PARAM_ASSERTIONS_ENABLED_PCH_CSS 0
00013 #endif
00014 #include <stdint.h>
00015 #include <assert.h>
00016 #include "hardware/sync.h"
00017 #include "hardware/irq.h"
00018 #include "picochan/css.h"
00019 #include "schibs_lock.h"
00020 #include "schib_internal.h"
00021 #include "schib_dlist.h"
00022 #include "channel.h"
00023 #include "picochan/dmachan.h"
00024 #include "trc/trace.h"
00025
00026
00027 struct css {
00028     schib_dlist_t    isc_dlists[PCH_NUM_ISCS]; // indexed by ISC
00029     io_callback_t    io_callback;
00030     int16_t          io_irqnum;
00031     int16_t          func_irqnum;
00032     uint8_t          isc_enable_mask;
00033     uint8_t          isc_status_mask;
00034     pch_dma_irq_index_t dmairqix;
00035     int8_t           core_num;
00036     pch_sid_t        next_sid;
00037     pch_trc_bufferset_t trace_bs;
00038     pch_chp_t        chps[PCH_NUM_CHANNELS];
00039     pch_schib_t      schibs[PCH_NUM_SCHIBS];
00040 };
00041
00042 extern struct css CSS;
00043
00044 static inline pch_schib_t *get_schib(pch_sid_t sid) {
00045     return &CSS.schibs[sid];
00046 }
00047
00048 static inline pch_chp_t *pch_get_chp(pch_chpid_t chpid) {
00049     return &CSS.chps[chpid];
00050 }
00051
00052 static inline pch_chpid_t pch_get_chpid(pch_chp_t *chp) {
00053     int32_t n = chp - &CSS.chps[0];
00054     assert(n >= 0 && n < PCH_NUM_CHANNELS);
00055     return (pch_chpid_t)n;
00056 }
00057
00058 static inline schib_dlist_t *get_isc_dlist(uint8_t iscnum) {
00059     valid_params_if(PCH_CSS, iscnum < PCH_NUM_ISCS);
00060     return &CSS.isc_dlists[iscnum];
00061 }
00062
00063 static inline pch_schib_t *get_schib_by_chp(pch_chp_t *chp, pch_unit_addr_t ua) {
00064     valid_params_if(PCH_CSS, (uint16_t)ua < chp->num_devices);
00065     return get_schib(chp->first_sid + (pch_sid_t)ua);
00066 }
00067
00068 static inline pch_schib_t *get_schib(pch_schib_t *schib) {
00069     // if we definitely decide to include intparm in the PMCW then
00070     // the schib size is 32 bytes so we could easily check the low
00071     // 5 bits are all zero as a valid_params_if check too.
00072     valid_params_if(PCH_CSS,
00073                     schib >= &CSS.schibs[0]
00074                     && schib < &CSS.schibs[PCH_NUM_SCHIBS]);
00075
00076     return schib - CSS.schibs;
00077 }
00078
00079 static inline pch_sid_t get_sid(pch_schib_t *schib) {
00080     // if we definitely decide to include intparm in the PMCW then
00081     // the schib size is 32 bytes so we could easily check the low
00082     // 5 bits are all zero as a valid_params_if check too.
00083     valid_params_if(PCH_CSS,
00084                     schib >= &CSS.schibs[0]
00085                     && schib < &CSS.schibs[PCH_NUM_SCHIBS]);
00086
00087     return schib - CSS.schibs;
00088 }
00089
00090 static inline bool css_is_started(void) {
00091     return CSS.dmairqix >= 0;
00092 }
```

```

00092 static inline void reset_subchannel_to_idle(pch_schib_t *schib) {
00093     const uint16_t mask = PCH_FC_START|PCH_FC_HALT|PCH_FC_CLEAR
00094         | PCH_AC_RESUME_PENDING|PCH_AC_START_PENDING
00095         | PCH_AC_HALT_PENDING|PCH_AC_CLEAR_PENDING
00096         | PCH_AC_SUSPENDED | PCH_SC_PENDING;
00097
00098     schib->scsw.ctrl_flags &= ~mask;
00099 }
00100
00101 static inline void css_clear_pending_subchannel(pch_schib_t *schib) {
00102     valid_params_if(PCH_CSS, schib_is_status_pending(schib));
00103
00104     if (schib->scsw.ctrl_flags & PCH_SC_INTERMEDIATE) {
00105         // TODO Don't do clearing unless various flag
00106         // combinations are set.
00107     }
00108
00109     reset_subchannel_to_idle(schib);
00110 }
00111
00112 void __isr pch_css_dma_irq_handler(void);
00113
00114 void suspend_or_send_start_packet(pch_chp_t *chp, pch_schib_t *schib, uint8_t ccwcmd);
00115 void do_command_chain_and_send_start(pch_chp_t *chp, pch_schib_t *schib);
00116 void send_command_with_data(pch_chp_t *chp, pch_schib_t *schib, proto_packet_t p, uint16_t count);
00117 void send_update_room(pch_chp_t *chp, pch_schib_t *schib);
00118 void send_data_response(pch_chp_t *chp, pch_schib_t *schib);
00119 void css_handle_rx_complete(pch_chp_t *chp);
00120 void css_handle_tx_complete(pch_chp_t *chp);
00121
00122 /**
00123 // isc_dlists
00124 // isc
00125 /**
00126
00127 pch_schib_t *pop_pending_schib_from_isc(uint8_t iscnnum);
00128 void remove_from_isc_dlist(uint8_t iscnnum, pch_sid_t sid);
00129 void push_to_isc_dlist(pch_schib_t *schib);
00130 pch_schib_t *pop_pending_schib(void);
00131 void css_notify(pch_schib_t *schib, uint8_t devs);
00132
00133 static inline pch_intcode_t css_make_intcode(pch_schib_t *schib) {
00134     pch_intcode_t ic = { 0 }; // all fields zeroes, including cc
00135     if (schib) {
00136         pch_sid_t sid = get_sid(schib);
00137         ic.intparm = schib->pmcw.intparm;
00138         ic.sid = sid;
00139         ic.flags = pch_pmcw_isc(&schib->pmcw);
00140         ic.cc = 1; // cc=1 means intcode stored [sic]
00141     }
00142
00143     return ic;
00144 }
00145
00146 #endif

```

13.35 css_trace.h

```

00001 /*
00002  * Copyright (c) 2025 Malcolm Beattie
00003  * SPDX-License-Identifier: MIT
00004 */
00005
00006 #ifndef _PCH_CSS_CSS_TRACE_H
00007 #define _PCH_CSS_CSS_TRACE_H
00008
00009 #include "css_internal.h"
0010
0011 #include "picochan/trc_records.h"
0012 #include "trc/trace.h"
0013
0014 #define PCH_CSS_TRACE_COND(rt, cond, data) \
0015     PCH_TRC_WRITE(&CSS.trace_bs, (cond), (rt), (data))
0016
0017 #define PCH_CSS_TRACE(rt, data) PCH_CSS_TRACE_COND((rt), true, (data))
0018
0019 static inline void trace_schib_byte(pch_trc_record_type_t rt, pch_schib_t *schib, uint8_t byte) {
0020     PCH_CSS_TRACE_COND(rt, schib_is_traced(schib),
0021                         ((struct pch_trdata_sid_byte){get_sid(schib), byte}));
0022 }
0023
0024 static inline void trace_schib_word_byte(pch_trc_record_type_t rt, pch_schib_t *schib, uint32_t word,
0025                                         uint8_t byte) {

```

```

00025     PCH_CSS_TRACE_COND(rt, schib_is_traced(schib),
00026                         ((struct pch_trdata_word_sid_byte){word, get_sid(schib), byte}));
00027 }
00028
00029 static inline void trace_schib_packet(pch_trc_record_type_t rt, pch_schib_t *schib, proto_packet_t p)
00030 {
00031     PCH_CSS_TRACE_COND(rt, schib_is_traced(schib),
00032                         ((struct pch_trdata_word_sid){proto_packet_as_word(p), get_sid(schib)}));
00033 }
00034
00035 static inline void trace_schib_ccw(pch_trc_record_type_t rt, pch_schib_t *schib, pch_ccw_t *ccw_addr,
00036                                   pch_ccw_t ccw) {
00037     PCH_CSS_TRACE_COND(rt, schib_is_traced(schib),
00038                         ((struct pch_trdata_ccw_addr_sid){
00039                             .ccw = ccw,
00040                             .addr = (uint32_t)ccw_addr,
00041                             .sid = get_sid(schib)
00042                         }));
00043
00044 static inline void trace_schib_callback(pch_trc_record_type_t rt, pch_schib_t *schib, pch_intcode_t
00045 *ic) {
00046     PCH_CSS_TRACE_COND(rt, schib_is_traced(schib),
00047                         ((struct pch_trdata_intcode_scsw){
00048                             .intcode = *ic,
00049                             .scsw = schib->scsw,
00050                         }));
00051
00052 static inline void trace_schib_scsw_cc(pch_trc_record_type_t rt, pch_schib_t *schib, pch_scsw_t *scsw,
00053                                   uint8_t cc) {
00054     PCH_CSS_TRACE_COND(rt, schib_is_traced(schib),
00055                         ((struct pch_trdata_scsw_sid_cc){
00056                             .scsw = *scsw,
00057                             .sid = get_sid(schib),
00058                             .cc = cc
00059                         }));
00060
00061 static inline void trace_chp_irq(pch_trc_record_type_t rt, pch_chp_t *chp, pch_dma_irq_index_t
00062 dmairqix, uint8_t tx_irq_state, uint8_t rx_irq_state) {
00063     PCH_CSS_TRACE_COND(rt,
00064                         chp->traced, ((struct pch_trdata_id_irq){
00065                             .id = pch_get_chpid(chp),
00066                             .dmairqix = dmairqix,
00067                             .tx_state = tx_irq_state << 4
00068                             | chp->tx_channel.mem_src_state,
00069                             .rx_state = rx_irq_state << 4
00070                             | chp->rx_channel.mem_dst_state
00071                         }));
00072 #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 irqn)`
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 irqn, uint8_t order_priority)`
- `void pch_css_configure_func_irq_exclusive (irq_num_t irqn)`
- `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 irqn)`
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 irqn, uint8_t order_priority)`
- `void pch_css_configure_io_irq_exclusive (irq_num_t irqn)`
- `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 the enabled field of the PMCW part of the schib for subchannel sid.

- **int pch_sch_modify_traced (pch_sid_t sid, bool traced)**
Modifies enabled flag of the schib for subchannel sid.
- **void (pch_sid_t sid, uint count, uint8_t isc)**
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.](#)

```

00001 /*
00002  * Copyright (c) 2025 Malcolm Beattie
00003  * SPDX-License-Identifier: MIT
00004 */
00005
00006 #ifndef _PCH_API_CSS_H
00007 #define _PCH_API_CSS_H
00008
00009 #include "hardware/irq.h"
00010 #include "hardware/dma.h"
00011 #include "hardware/sync.h"
00012 #include "hardware/uart.h"
00013 #include "pico/time.h"
00014 #include "picochan/schib.h"
00015 #include "picochan/ccw.h"
00016 #include "picochan/intcode.h"
00017 #include "picochan/dmachan.h"
00018
00024
00035 #ifndef PCH_NUM_SCHIBS
00036 #define PCH_NUM_SCHIBS 32
00037 #endif
00038 static_assert(PCH_NUM_SCHIBS >= 1 && PCH_NUM_SCHIBS <= 65536,
00039             "PCH_NUM_SCHIBS must be between 1 and 65536");
00040
00052 #ifndef PCH_NUM_CHANNELS
00053 #define PCH_NUM_CHANNELS 4
00054 #endif
00055 static_assert(PCH_NUM_CHANNELS >= 1 && PCH_NUM_CHANNELS <= 256,
00056             "PCH_NUM_CHANNELS must be between 1 and 256");
00057
00069 #ifndef PCH_NUM_ISCS
00070 #define PCH_NUM_ISCS 8
00071 #endif
00072 static_assert(PCH_NUM_ISCS >= 1 && PCH_NUM_ISCS <= 8,
00073             "PCH_NUM_ISCS must be between 1 and 8");
00074
00075 #define PCH_CSS_BUFFERSET_MAGIC 0x70437353
00076
00080
00081 typedef void(*io_callback_t)(pch_intcode_t, pch_scsw_t);
00082
00092 static inline void *pch_ccw_get_addr(pch_ccw_t ccw) {
00093     return (void *)ccw.addr;
00094 }
00095
00101 void pch_css_init(void);
00102
00103 // Accessor functions for basic CSS settings
00104 int8_t pch_css_get_core_num(void);

```

```
00105 int8_t pch_css_get_dma_irq_index(void);
00106 int16_t pch_css_get_func_irq(void);
00107 int16_t pch_css_get_io_irq(void);
00108
00109 // A variety of different initialisation functions for configuring
00110 // CSS IRQ numbers and handlers for DMA IRQ index, function IRQ
00111 // and I/O IRQ.
00112
00113 void pch_css_set_dma_irq_index(pch_dma_irq_index_t dmairqix);
00114 void pch_css_configure_dma_irq_index_shared(pch_dma_irq_index_t dmairqix, uint8_t order_priority);
00115 void pch_css_configure_dma_irq_index_exclusive(pch_dma_irq_index_t dmairqix);
00116 void pch_css_configure_dma_irq_index_default_shared(uint8_t order_priority);
00117 void pch_css_configure_dma_irq_index_default_exclusive();
00118 void pch_css_auto_configure_dma_irq_index();
00119
00120
00121 void pch_css_set_func_irq(irq_num_t irqnum);
00122 void pch_css_configure_func_irq_shared(irq_num_t irqnum, uint8_t order_priority);
00123 void pch_css_configure_func_irq_exclusive(irq_num_t irqnum);
00124 void pch_css_configure_func_irq_unused_shared(bool required, uint8_t order_priority);
00125 void pch_css_configure_func_irq_unused_exclusive(bool required);
00126 void pch_css_auto_configure_func_irq(bool required);
00127
00128
00129 void pch_css_set_io_irq(irq_num_t irqnum);
00130 void pch_css_configure_io_irq_shared(irq_num_t irqnum, uint8_t order_priority);
00131 void pch_css_configure_io_irq_exclusive(irq_num_t irqnum);
00132 void pch_css_configure_io_irq_unused_shared(bool required, uint8_t order_priority);
00133 void pch_css_configure_io_irq_unused_exclusive(bool required);
00134 void pch_css_auto_configure_io_irq(bool required);
00135
00136
00137 io_callback_t pch_css_set_io_callback(io_callback_t io_callback);
00138
00139
00140 void pch_css_start(io_callback_t io_callback, uint8_t isc_mask);
00141
00142
00143 bool pch_css_set_trace(bool trace);
00144
00145
00146 bool pch_chp_set_trace(pch_chpid_t chpid, bool trace);
00147
00148
00149 void __isr pch_css_func_irq_handler(void);
00150 void __isr pch_css_io_irq_handler(void);
00151
00152
00153 void pch_css_set_io_irq(irq_num_t irqnum);
00154
00155
00156 io_callback_t pch_css_set_io_callback(io_callback_t io_callback);
00157
00158
00159 void pch_chp_start(pch_chpid_t chpid);
00160
00161
00162 // Channel initialisation
00163
00164
00165 void pch_chp_claim(pch_chpid_t chpid);
00166
00167
00168 int pch_chp_claim_unused(bool required);
00169
00170
00171 pch_sid_t pch_chp_alloc(pch_chpid_t chpid, uint16_t num_devices);
00172
00173
00174
00175 void pch_chp_configure_uartchan(pch_chpid_t chpid, uart_inst_t *uart, dma_channel_config ctrl);
00176
00177
00178 void pch_chp_auto_configure_uartchan(pch_chpid_t chpid, uart_inst_t *uart, uint baudrate);
00179
00180
00181 void pch_chp_configure_memchan(pch_chpid_t chpid, pch_dmaid_t txdmaid, pch_dmaid_t rxdmaid,
00182     dmachan_tx_channel_t *txpeer);
00183
00184
00185 // Channel initialisation low-level helpers
00186
00187 void pch_chp_dma_configure(pch_chpid_t chpid, dmachan_config_t *dc);
00188
00189 void pch_chp_set_configured(pch_chpid_t chpid, bool configured);
00190
00191 dmachan_tx_channel_t *pch_chp_get_tx_channel(pch_chpid_t chpid);
00192
00193 dmachan_rx_channel_t *pch_chp_get_rx_channel(pch_chpid_t chpid);
00194
00195
00196 // Architectural API for subchannels and channel programs
00197
00198
00199 int pch_sch_start(pch_sid_t sid, pch_ccw_t *ccw_addr);
00200
00201
00202 int pch_sch_resume(pch_sid_t sid);
00203
00204
00205 int pch_sch_test(pch_sid_t sid, pch_scsw_t *scsw);
00206
00207
00208 int pch_sch_modify(pch_sid_t sid, pch_pmcw_t *pmcw);
00209
00210
00211 int pch_sch_store(pch_sid_t sid, pch_schib_t *out_schib);
00212
00213
00214 int pch_sch_cancel(pch_sid_t sid);
00215
00216
00217 int pch_sch_halt(pch_sid_t sid);
00218
00219
00220 pch_intcode_t pch_test_pending_interruption(void);
00221
```

```

00461 // API additions with internal optimisation
00462
00469 int pch_sch_store_pmcw(pch_sid_t sid, pch_pmcw_t *out_pmcw);
00470
00477 int pch_sch_store_scsw(pch_sid_t sid, pch_scsw_t *out_scsw);
00478
00479 // Convenience API functions that wrap the architectural API
00480
00487 int pch_sch_modify_intparm(pch_sid_t sid, uint32_t intparm);
00488
00495 int pch_sch_modify_flags(pch_sid_t sid, uint16_t flags);
00496
00503 int pch_sch_modify_isc(pch_sid_t sid, uint8_t isc);
00504
00511 int pch_sch_modify_enabled(pch_sid_t sid, bool enabled);
00512
00519 int pch_sch_modify_traced(pch_sid_t sid, bool traced);
00520
00525 void __time_critical_func(pch_sch_modify_isc_range)(pch_sid_t sid, uint count, uint8_t isc);
00526
00531 void __time_critical_func(pch_sch_modify_enabled_range)(pch_sid_t sid, uint count, bool enabled);
00532
00537 void __time_critical_func(pch_sch_modify_traced_range)(pch_sid_t sid, uint count, bool traced);
00538
00539 // These functions should only be called while the ISC for the
00540 // subchannel has been disabled
00541
00547 int pch_sch_wait(pch_sid_t sid, pch_scsw_t *scsw);
00558
00567 int pch_sch_wait_timeout(pch_sid_t sid, pch_scsw_t *scsw, absolute_time_t timeout_timestamp);
00568
00576 int pch_sch_run_wait(pch_sid_t sid, pch_ccw_t *ccw_addr, pch_scsw_t *scsw);
00577
00586 int pch_sch_run_wait_timeout(pch_sid_t sid, pch_ccw_t *ccw_addr, pch_scsw_t *scsw, absolute_time_t
    timeout_timestamp);
00587
00588 #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 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.](#)

```

00001 /*
00002  * Copyright (c) 2025 Malcolm Beattie
00003  * SPDX-License-Identifier: MIT
00004 */
00005
00006 #ifndef _PCH_CSS_PMCW_H
00007 #define _PCH_CSS_PMCW_H
00008
00009 #include <stdbool.h>
00010 #include "picochan/ids.h"
00011
00012
00013 typedef struct pch_pmcw {
00014     uint32_t      intparm;
00015     uint16_t      flags;
00016     pch_chpid_t   chpid;
00017     pch_unit_addr_t unit_addr;
00018 } pch_pmcw_t;
00019
00020 // PCH_PMCW_SCH MODIFY_MASK are the bits of the PMCW flags
00021 // which can be set with the Modify Subchannel function
00022 #define PCH_PMCW_SCH MODIFY_MASK 0x001f
00023
00024 // ISC: Interrupt Service Class - the low 3 bits of the PMCW.
00025 // We define PCH_PMCW_ISC_LSB as the shift count to get the ISC bits
00026 // in case we ever want to move them, even though it's currently 0.
00027 #define PCH_PMCW_ISC_BITS      0x07
00028 #define PCH_PMCW_ISC_LSB       0
00029 #define PCH_PMCW_ENABLED      0x08
00030 #define PCH_PMCW_TRACED       0x10
00031
00032 static inline uint8_t pch_pmcw_isc(pch_pmcw_t *pmcw) {
00033     return (pmcw->flags & PCH_PMCW_ISC_BITS) >> PCH_PMCW_ISC_LSB;
00034 }
00035
00036 bool pch_css_is_isc_enabled(uint8_t iscnm);
00037 void pch_css_set_isc_enabled(uint8_t iscnm, bool enabled);
00038 void pch_css_disable_isc(uint8_t iscnm);
00039 void pch_css_disable_isc_mask(uint8_t mask);
00040 void pch_css_enable_isc(uint8_t iscnm);
00041 void pch_css_enable_isc_mask(uint8_t mask);
00042 void pch_css_set_isc_enable_mask(uint8_t mask);
00043
00044 #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.](#)

```
00001 /*
00002  * Copyright (c) 2025 Malcolm Beattie
00003  * SPDX-License-Identifier: MIT
00004 */
00005
00006 #ifndef _PCH_API_SCHIB_H
00007 #define _PCH_API_SCHIB_H
00008
00009 #include "picochan/ids.h"
00010 #include "picochan/pmcw.h"
00011 #include "picochan/scsw.h"
00012
00013
00025 typedef struct pch_schib_mda {
00026     uint32_t          data_addr;
00027     uint16_t          devcount;
00028     pch_unit_addr_t  prevua;
00029     pch_unit_addr_t  nextua;
00030     pch_sid_t         prevsid;
00031     pch_sid_t         nextsid;
00032 } pch_schib_mda_t;
00033 static_assert(sizeof(pch_schib_mda_t) == 12,
00034               "pch_schib_mda_t should be 12 bytes");
00035
00065 typedef struct pch_schib {
00066     pch_pmcw_t        pmcw;
```

```

00067     pch_scsw_t      scsw;
00068     pch_schib_mda_t mda;
00069 } pch_schib_t;
00070 static_assert(sizeof(pch_schib_t) == 32,
00071     "pch_schib_t should be 32 bytes");
00072
00073 static inline bool schib_is_enabled(pch_schib_t *schib) {
00074     return schib->pmcw.flags & PCH_PMCW_ENABLED;
00075 }
00076
00077 static inline bool schib_is_traced(pch_schib_t *schib) {
00078     return schib->pmcw.flags & PCH_PMCW_TRACED;
00079 }
00080
00081 static inline bool schib_has_function_in_progress(pch_schib_t *schib) {
00082     const uint16_t mask = PCH_FC_START|PCH_FC_HALT|PCH_FC_CLEAR;
00083     return schib->scsw.ctrl_flags & mask;
00084 }
00085
00086 static inline bool schib_is_status_pending(pch_schib_t *schib) {
00087     return schib->scsw.ctrl_flags & PCH_SC_PENDING;
00088 }
00089
00090 #endif

```

13.42 schib_dlist.h

```

00001 /*
00002  * Copyright (c) 2025 Malcolm Beattie
00003  * SPDX-License-Identifier: MIT
00004 */
00005
00006 #ifndef _PCH_CSS_SCHIB_DLST_H
00007 #define _PCH_CSS_SCHIB_DLST_H
00008
00009 // schib_dlist_t is a doubly linked (by sid) list of schibs
00010 typedef int32_t schib_dlist_t;
00011
00012 pch_schib_t *remove_from_schib_dlist_unsafe(schib_dlist_t *l, pch_sid_t sid);
00013 bool push_to_schib_dlist_unsafe(schib_dlist_t *l, pch_sid_t sid);
00014
00015 static inline pch_schib_t *remove_from_schib_dlist(schib_dlist_t *l, pch_sid_t sid) {
00016     uint32_t status = schibs_lock();
00017     pch_schib_t *schib = remove_from_schib_dlist_unsafe(l, sid);
00018     schibs_unlock(status);
00019     return schib;
00020 }
00021
00022 static inline pch_schib_t *pop_schib_dlist_unsafe(schib_dlist_t *l) {
00023     if (*l == -1)
00024         return NULL;
00025
00026     return remove_from_schib_dlist_unsafe(l, (pch_sid_t)*l);
00027 }
00028
00029 static inline pch_schib_t *pop_schib_dlist(schib_dlist_t *l) {
00030     uint32_t status = schibs_lock();
00031     pch_schib_t *schib = pop_schib_dlist_unsafe(l);
00032     schibs_unlock(status);
00033     return schib;
00034 }
00035
00036 static inline bool push_to_schib_dlist(schib_dlist_t *l, pch_sid_t sid) {
00037     uint32_t status = schibs_lock();
00038     bool was_empty = push_to_schib_dlist_unsafe(l, sid);
00039     schibs_unlock(status);
00040     return was_empty;
00041 }
00042
00043 #endif

```

13.43 schib_internal.h

```

00001 /*
00002  * Copyright (c) 2025 Malcolm Beattie
00003  * SPDX-License-Identifier: MIT
00004 */
00005

```

```

00006 #ifndef _PCH_CSS_SCHIB_INTERNAL_H
00007 #define _PCH_CSS_SCHIB_INTERNAL_H
00008
00009 #include "picochan/schib.h"
00010 #include "picochan/dev_status.h"
00011 #include "picochan/ccw.h"
00012
00013 // get_stashed_ccw_flags is a CSS-internal function that fetches the
00014 // CCW flags that we stash in the SCSW device status field during
00015 // execution of a channel program. The SCSW device status field is
00016 // only architected to be valid when Status Pending is set in the
00017 // Status Control flags and we have to be careful only to stash
00018 // CCW flags in this field when Status Pending is not set.
00019 static inline pch_ccw_flags_t get_stashed_ccw_flags(pch_schib_t *schib) {
00020     return (pch_ccw_flags_t)schib->scsw.devs; // sic
00021 }
00022
00023 #endif

```

13.44 schibs_lock.h

```

00001 /*
00002  * Copyright (c) 2025 Malcolm Beattie
00003  * SPDX-License-Identifier: MIT
00004 */
00005
00006 #ifndef _PCH_CSS_SCHIBS_LOCK_H
00007 #define _PCH_CSS_SCHIBS_LOCK_H
00008
00009 #include "hardware/sync.h"
00010
00011 // schibs_lock() and schibs_unlock() protect manipulation of the
00012 // linked lists of schib's with pending functions (i.e. API
00013 // functions such as Start Subchannel). The user API uses a
00014 // critical section protected by schibs_lock() /schibs_unlock() to
00015 // update the Function Control flags in the target schib with the
00016 // request, add itself to the ua_func_dlist headed by the channel
00017 // responsible for the subchannel (linked via mda.prevua/nextua) and
00018 // ping the CSS with raise_func_irq.
00019 // At the moment, we assume the user API invocations and the CSS
00020 // itself run on the same core and so the ping is raising a
00021 // (non-hardware-connected) IRQ and lock/unlock is a simple
00022 // disable/restore of (all) interrupts. If we want to separate out
00023 // the user invocations onto a different core from the CSS itself
00024 // (and there's no inherent problem with that since the CSS runs
00025 // entirely asynchronously and can cope with that) then we can
00026 // change the ping to be a doorbell interrupt to the other core
00027 // and change the lock/unlock to be a (hardware) spinlock plus the
00028 // disable/restore of interrupts.
00029
00030 static inline uint32_t schibs_lock(void) {
00031     return save_and_disable_interrupts();
00032 }
00033
00034 static inline void schibs_unlock(uint32_t status) {
00035     restore_interrupts(status);
00036 }
00037
00038 #endif

```

13.45 callback.h

```

00001 /*
00002  * Copyright (c) 2025 Malcolm Beattie
00003  * SPDX-License-Identifier: MIT
00004 */
00005
00006 #ifndef _PCH_CUS_CALLBACK_H
00007 #define _PCH_CUS_CALLBACK_H
00008
00009 #include "picochan/cu.h"
00010 #include "picochan/devib.h"
00011 #include "picochan/dev_status.h"
00012 #include "cus_trace.h"
00013
00014 static inline void pch_call_devib_callback(pch_cbindex_t cbindex, pch_devib_t *devib) {
00015     assert(pch_cbindex_is_callable(cbindex));
00016

```

```

00017     if (cbindex == PCH_DEVIB_CALLBACK_NOOP)
00018         return;
00019
00020     pch_devib_callback_t cb = pch_devib_callbacks[cbindex];
00021     cb(devib);
00022 }
00023
00024 static inline void callback_devib(pch_devib_t *devib) {
00025     pch_cbindex_t cbindex = devib->cbindex;
00026
00027     trace_call_callback(PCH_TRC_RT_CUS_CALL_CALLBACK,
00028                         devib, cbindex);
00029     pch_call_devib_callback(cbindex, devib);
00030 }
00031
00032 #endif

```

13.46 cu_internal.h

```

00001 /*
00002  * Copyright (c) 2025 Malcolm Beattie
00003  * SPDX-License-Identifier: MIT
00004 */
00005
00006 #ifndef _PCH_CUS CU_INTERNAL_H
00007 #define _PCH_CUS CU_INTERNAL_H
00008
00009 #include "picochan/cu.h"
00010 #include "proto/packet.h"
00011 #include "devibs_lock.h"
00012
00013 static inline proto_packet_t get_rx_packet(pch_cu_t *cu) {
00014     // cu.rx_channel is a dmachan_rx_channel_t which is
00015     // __aligned(4) and cmd is the first member of rx_channel
00016     // so is 4-byte aligned. proto_packet_t is 4-bytes and also
00017     // __aligned(4) (and needing no more than 4-byte alignment)
00018     // but omitting the __builtin_assume_aligned below causes
00019     // gcc 14.1.0 to produce error
00020     // error: cast increases required alignment of target type
00021     // [-Werror=cast-align]
00022     proto_packet_t *pp = (proto_packet_t *)
00023         __builtin_assume_aligned(&cu->rx_channel.link.cmd, 4);
00024     return *pp;
00025 }
00026
00027 void pch_cus_send_command_to_css(pch_cu_t *cu);
00028 void pch_cus_handle_rx_complete(pch_cu_t *cu);
00029 void pch_cus_handle_tx_complete(pch_cu_t *cu);
00030
00031 #endif

```

13.47 cus_trace.h

```

00001 /*
00002  * Copyright (c) 2025 Malcolm Beattie
00003  * SPDX-License-Identifier: MIT
00004 */
00005
00006 #ifndef _PCH_CUS_CUS_TRACE_H
00007 #define _PCH_CUS_CUS_TRACE_H
00008
00009 #include "picochan/devib.h"
00010 #include "picochan/cu.h"
00011 #include "picochan/trc_records.h"
00012 #include "trc/trace.h"
00013 #include "proto/packet.h"
00014 #include "txsm/txsm.h"
00015
00016 extern pch_trc_bufferset_t pch_cus_trace_bs;
00017
00018 #define PCH_CUS_TRACE_COND(rt, cond, data) \
00019     PCH_TRC_WRITE(&pch_cus_trace_bs, (cond), (rt), (data))
00020
00021 #define PCH_CUS_TRACE(rt, data) PCH_CUS_TRACE_COND((rt), true, (data))
00022
00023 static inline void trace_dev(pch_trc_record_type_t rt, pch_devib_t *devib) {
00024     PCH_CUS_TRACE_COND(rt, cu_or_devib_is_traced(devib),
00025         ((struct pch_trdata_dev) {

```

```

00026             .cuaddr = pch_dev_get_cuaddr(devib),
00027             .ua = pch_dev_get_ua(devib)
00028         });
00029     }
00030
00031 static inline void trace_dev_byte(pch_trc_record_type_t rt, pch_devib_t *devib, uint8_t byte) {
00032     PCH_CUS_TRACE_COND(rt, cu_or_devib_is_traced(devib),
00033     ((struct pch_trdata_dev_byte){
00034         .cuaddr = pch_dev_get_cuaddr(devib),
00035         .ua = pch_dev_get_ua(devib),
00036         .byte = byte
00037     }));
00038 }
00039
00040 static inline void trace_dev_packet(pch_trc_record_type_t rt, pch_devib_t *devib, proto_packet_t p) {
00041     PCH_CUS_TRACE_COND(rt,
00042         cu_or_devib_is_traced(devib),
00043         ((struct pch_trdata_word_dev){
00044             .word = proto_packet_as_word(p),
00045             .cuaddr = pch_dev_get_cuaddr(devib),
00046             .ua = pch_dev_get_ua(devib)
00047         }));
00048 }
00049
00050 static inline void trace_tx_complete(pch_trc_record_type_t rt, pch_cu_t *cu, int16_t uaopt,
00051     pch_txsm_state_t txpstate) {
00052     PCH_CUS_TRACE_COND(rt, cu->traced,
00053     ((struct pch_trdata_cus_tx_complete){
00054         .uaopt = uaopt,
00055         .cuaddr = cu->cuaddr,
00056         .txpstate = (uint8_t)txpstate
00057     }));
00058
00059 static inline void trace_register_callback(pch_trc_record_type_t rt, pch_cbindex_t n,
00060     pch_devib_callback_t cb) {
00061     PCH_CUS_TRACE(rt,
00062         ((struct pch_trdata_word_byte){(uint32_t)cb,n}));
00063 }
00064 static inline void trace_call_callback(pch_trc_record_type_t rt, pch_devib_t *devib, pch_cbindex_t
00065     cbindex) {
00066     PCH_CUS_TRACE_COND(rt,
00067         cu_or_devib_is_traced(devib),
00068         ((struct pch_trdata_cus_call_callback){
00069             .cuaddr = pch_dev_get_cuaddr(devib),
00070             .ua = pch_dev_get_ua(devib),
00071             .cbindex = (uint8_t)cbindex
00072     }));
00073
00074 static inline void trace_cu_irq(pch_trc_record_type_t rt, pch_cu_t *cu, pch_dma_irq_index_t dmairqix,
00075     uint8_t tx_irq_state, uint8_t rx_irq_state) {
00076     PCH_CUS_TRACE_COND(rt,
00077         cu->traced, ((struct pch_trdata_id_irq){
00078             .id = cu->cuaddr,
00079             .dmairqix = dmairqix,
00080             .tx_state = tx_irq_state << 4
00081                 | cu->tx_channel.mem_src_state,
00082             .rx_state = rx_irq_state << 4
00083                 | cu->rx_channel.mem_dst_state
00084     }));
00085 }
00086 #endif

```

13.48 devibs_lock.h

```

00001 /*
00002  * Copyright (c) 2025 Malcolm Beattie
00003  * SPDX-License-Identifier: MIT
00004 */
00005
00006 #ifndef _PCH_CUS_DEVIBS_LOCK_H
00007 #define _PCH_CUS_DEVIBS_LOCK_H
00008
00009 #include "hardware/sync.h"
00010
00011 // devibs_lock() and devibs_unlock() protect manipulation of the
00012 // linked lists of devibs's with pending functions (i.e. API
00013 // functions such as Start Subchannel). The device API uses a
00014 // critical section protected by devibs_lock() / devibs_unlock() to
00015 // add itself to the tx pending list headed by the devices' CU

```

```

00016 // fields tx_head and tx_tail and linked via devib->next. The list
00017 // is traversed and the pending packets sent (from the devib fields
00018 // op and payload and using the devib's ua) whenever the CU's tx
00019 // engine is free, driven by DMA completion on the tx channel.
00020 // We assume the device API invocations and the CU itself run on the
00021 // same core and so simply disable/restore (all) interrupts without
00022 // needing to worry about cross-core locking.
00023 static inline uint32_t devibs_lock(void) {
00024     return save_and_disable_interrupts();
00025 }
00026
00027 static inline void devibs_unlock(uint32_t status) {
00028     restore_interrupts(status);
00029 }
00030
00031 #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.](#)

```

00001 /*
00002  * Copyright (c) 2025 Malcolm Beattie
00003  * SPDX-License-Identifier: MIT
00004 */
00005
00006 #ifndef _PCH_CUS_CU_H
00007 #define _PCH_CUS_CU_H
00008
00009 // PICO_CONFIG: PARAM_ASSERTIONS_ENABLED_PCH_CUS, Enable/disable assertions in the pch_cus module,
00010 // type=bool, default=0, group=pch_cus
00011 #ifndef PARAM_ASSERTIONS_ENABLED_PCH_CUS
00012 #define PARAM_ASSERTIONS_ENABLED_PCH_CUS 0
00013 #endif
00014 #ifndef PCH_MAX_DEVIBS_PER CU
00015 #define PCH_MAX_DEVIBS_PER CU 32
00016 #endif
00017
00018 #include <stdint.h>
00019 #include <assert.h>
00020 #include "hardware/uart.h"
00021 #include "picochan/dev_api.h"
00022 #include "picochan/dmchan.h"
00023 #include "txsm/txsm.h"
00024
00025 static_assert(__builtin_constant_p(PCH_MAX_DEVIBS_PER CU),
00026                 "PCH_MAX_DEVIBS_PER CU must be a compile-time constant");
00027
00028 static_assert(PCH_MAX_DEVIBS_PER CU >= 1 && PCH_MAX_DEVIBS_PER CU <= 256,
00029                 "PCH_MAX_DEVIBS_PER CU must be between 1 and 256");
00030
00031 #define PCH_MAX_DEVIBS_PER CU_ALIGN_SHIFT (31U - __builtin_clz(2 * (PCH_MAX_DEVIBS_PER CU) - 1))
00032
00033 static_assert(__builtin_constant_p(PCH_MAX_DEVIBS_PER CU_ALIGN_SHIFT),
00034                 "__builtin_clz() did not produce compile-time constant for
00035 PCH_MAX_DEVIBS_PER CU_ALIGN_SHIFT");
00036
00037 #define PCH_CU_ALIGN (1U << (PCH_DEVIB_SPACE_SHIFT+PCH_MAX_DEVIBS_PER CU_ALIGN_SHIFT))
00038
00039 static_assert(__builtin_constant_p(PCH_CU_ALIGN),
00040                 "could not produce compile-time constant for PCH_CU_ALIGN");
00041
00042
00043 #ifndef PCH_NUM_CUS
00044 #define PCH_NUM_CUS 4
00045 #endif
00046
00047 static_assert(PCH_NUM_CUS >= 1 && PCH_NUM_CUS <= 256,
00048                 "PCH_NUM_CUS must be between 1 and 256");
00049
00050 #define PCH_CUS_BUFFERSET_MAGIC 0x70437553
00051
00052 typedef struct __aligned(PCH_CU_ALIGN) pch_cu {
00053     dmchan_tx_channel_t tx_channel;
00054     dmchan_rx_channel_t rx_channel;
00055     pch_txsm_t tx_pending;
00056     pch_cuaddr_t cuaddr;
00057     int16_t tx_callback_ua;
00058     int16_t rx_active;
00059     int16_t tx_head;
00060     int16_t tx_tail;
00061     pch_dma_irq_index_t dmairqix;
00062     bool traced;
00063     bool configured;
00064     bool started;
00065     uint16_t num_devibs;
00066     pch_devib_t devibs[];
00067 } pch_cu_t;
00068
00069 static inline pch_dma_irq_index_t pch_cu_get_dma_irq_index(pch_cu_t *cu) {
00070     return cu->dmairqix;
00071 }
```

```

00118
00119 void pch_cu_set_dma_irq_index(pch_cu_t *cu, pch_dma_irq_index_t dmairqix);
00120
00121 #define PCH CU INIT(num_devices) { \
00122     .rx_active = -1, \
00123     .tx_head = -1, \
00124     .tx_tail = -1, \
00125     .dmairqix = -1, \
00126     .num_devibs = (num_devices), \
00127     .devibs = { [(num_devices)-1] = {0} } \
00128 }
00129
00130
00131 static inline pch_cu_t *pch_dev_get_cu(pch_devib_t *devib) {
00132     unsigned long p = (unsigned long)devib;
00133     p -= __builtin_offsetof(pch_cu_t, devibs);
00134     p &= ~(PCH CU ALIGN-1);
00135     return (pch_cu_t *)p;
00136 }
00137
00138 static inline pch_cuaddr_t pch_dev_get_cuaddr(pch_devib_t *devib) {
00139     pch_cu_t *cu = pch_dev_get_cu(devib);
00140     return cu->cuaddr;
00141 }
00142
00143 static inline pch_unit_addr_t pch_dev_get_ua(pch_devib_t *devib) {
00144     pch_cu_t *cu = pch_dev_get_cu(devib);
00145     return (pch_unit_addr_t)(devib - cu->devibs);
00146 }
00147
00148 static inline pch_devib_t *pch_get_devib(pch_cu_t *cu, pch_unit_addr_t ua) {
00149     return &cu->devibs[ua];
00150 }
00151
00152 static inline bool cu_or_devib_is_traced(pch_devib_t *devib) {
00153     pch_cu_t *cu = pch_dev_get_cu(devib);
00154     return cu->traced || pch_devib_is_traced(devib);
00155 }
00156
00157 extern pch_cu_t *pchip_cus[PCH_NUM_CUS];
00158
00159 extern bool pchip_cus_init_done;
00160
00161 static inline pch_cu_t *pch_get_cu(pch_cuaddr_t cua) {
00162     valid_params_if(PCH_CUS, cua < PCH_NUM_CUS);
00163     pch_cu_t *cu = pchip_cus[cua];
00164     assert(cu != NULL);
00165     return cu;
00166 }
00167
00168 void pchip_cus_init(void);
00169
00170 bool pchip_cus_set_trace(bool trace);
00171
00172 /*
00173  * \brief Configure an explicit DMA IRQ for use by CUs started from the
00174  * calling core and set an exclusive IRQ handler for it.
00175  * \ingroup picochan_cu
00176  *
00177  * If a CSS is to be used on the same Pico, it must be initialised on
00178  * a different core, using a different DMA IRQ index. A convenient way
00179  * to still allow the CU subsystem to auto-configure its DMA IRQ
00180  * choice is to call pchip_cus_ignore_dma_irq_index_t() on the DMA IRQ
00181  * index of the CSS.
00182 */
00183 void pchip_cus_configure_dma_irq_index_exclusive(pch_dma_irq_index_t dmairqix);
00184
00185 /*
00186  * \brief Configure an explicit DMA IRQ for use by CUs started from
00187  * the calling core and add a shared IRQ handler for it.
00188  * \ingroup picochan_cu
00189  *
00190  * If a CSS is to be used on the same Pico, it must be initialised on
00191  * a different core, using a different DMA IRQ index. A convenient way
00192  * to still allow the CU subsystem to auto-configure its DMA IRQ
00193  * choice is to call pchip_cus_ignore_dma_irq_index_t() on the DMA IRQ
00194  * index of the CSS.
00195 */
00196 void pchip_cus_configure_dma_irq_index_shared(pch_dma_irq_index_t dmairqix, uint8_t order_priority);
00197
00198 /*
00199  * \brief Configure an explicit DMA IRQ for use by CUs started from
00200  * the calling core and add a shared IRQ handler for it using an
00201  * order_priority of PICO_SHARED_IRQ_HANDLER_DEFAULT_ORDER_PRIORITY.
00202  * \ingroup picochan_cu
00203  *
00204  * If a CSS is to be used on the same Pico, it must be initialised on
00205  * a different core, using a different DMA IRQ index. A convenient way

```

```

00240 * to still allow the CU subsystem to auto-configure its DMA IRQ
00241 * choice is to call pch_cus_ignore_dma_irq_index_t() on the DMA IRQ
00242 * index of the CSS.
00243 */
00244 void pch_cus_configure_dma_irq_index_shared_default(pch_dma_irq_index_t dmairqix);
00245
00246 /*
00247 * \brief Automatically choose and configure a suitable DMA IRQ for
00248 * use by CUs started from the calling core.
00249 * \ingroup picochan_cu
00250 *
00251 * If one of the explicit pch_cus_configure_dma_irq_index_...()
00252 * family of functions has already been called from the calling core
00253 * then the lowest such DMA IRQ index is returned. Otherwise, the
00254 * lowest DMA IRQ index is chosen that has not already been either
00255 * configured to any core or explicitly marked as not-to-use by
00256 * pch_cus_ignore_dma_irq_index_t(). It is then configured with
00257 * pch_cus_configure_dma_irq_index_shared_default() and returned.
00258 * In the case that no such unused index is available, the function
00259 * panics if required is true, otherwise -1 is returned.
00260 *
00261 * If a CSS is to be used on the same Pico, it must be initialised on
00262 * a different core, using a different DMA IRQ index. A convenient way
00263 * to still allow the CU subsystem to auto-configure its DMA IRQ
00264 * choice is to call pch_cus_ignore_dma_irq_index_t() on the DMA IRQ
00265 * index of the CSS.
00266 */
00267 pch_dma_irq_index_t pch_cus_auto_configure_dma_irq_index(bool required);
00268
00269 /* \brief Marks dmairqix such that any call to
00270 * pch_cus_auto_configure_dma_irq_index(), whether explicit or
00271 * implicitly from pch_cu_start(), will not choose that DMA IRQ index.
00272 * \ingroup picochan_cu
00273 *
00274 * This function is convenient for avoiding the need to configure
00275 * explicit DMA IRQ index numbers for the CU subsystem while ensuring
00276 * that its auto-configuration of DMA IRQ index numbers does not
00277 * conflict with those of a CSS in use on the same Pico or just some
00278 * other DMA IRQ index that needs to be reserved for application use.
00279 */
00280 void pch_cus_ignore_dma_irq_index_t(pch_dma_irq_index_t dmairqix);
00281
00282 // CU initialisation and configuration
00283
00284 void pch_cu_init(pch_cu_t *cu, uint16_t num_devibs);
00285
00286 void pch_cu_register(pch_cu_t *cu, pch_cuaddr_t cua);
00287
00288 void pch_cus_uartcu_configure(pch_cuaddr_t cua, uart_inst_t *uart, dma_channel_config ctrl);
00289
00290 void pch_cus_auto_configure_uartcu(pch_cuaddr_t cua, uart_inst_t *uart, uint baudrate);
00291
00292 void pch_cus_memcu_configure(pch_cuaddr_t cua, pch_dmaid_t txdmaid, pch_dmaid_t rxdmaid,
00293     dmachan_tx_channel_t *txpeer);
00294
00295 void pch_cu_start(pch_cuaddr_t cua);
00296
00297 bool pch_cus_trace_cu(pch_cuaddr_t cua, bool trace);
00298
00299 bool pch_cus_trace_dev(pch_devib_t *devib, bool trace);
00300
00301 // CU initialisation low-level helpers
00302 void pch_cu_dma_configure(pch_cuaddr_t cua, dmachan_config_t *dc);
00303 void pch_cu_set_configured(pch_cuaddr_t cua, bool configured);
00304
00305 dmachan_tx_channel_t *pch_cu_get_tx_channel(pch_cuaddr_t cua);
00306
00307 dmachan_rx_channel_t *pch_cu_get_rx_channel(pch_cuaddr_t cua);
00308
00309 void __isr pch_cus_handle_dma_irq(void);
00310
00311 typedef struct pch_dev_range {
00312     pch_cu_t             *cu;
00313     uint16_t              num_devices;    // 0 to 256
00314     pch_unit_addr_t       first_ua;
00315 } pch_dev_range_t;
00316
00317 static inline pch_unit_addr_t pch_dev_range_get_ua(pch_dev_range_t *dr, uint i) {
00318     assert(dr->cu);
00319     assert(i < dr->num_devices);
00320     assert((uint)dr->first_ua + i < dr->cu->num_devibs);
00321
00322     return dr->first_ua + i;
00323 }
00324
00325 static inline int pch_dev_range_get_index_nocheck(pch_dev_range_t *dr, pch_devib_t *devib) {
00326     return (int)pch_dev_get_ua(devib) - dr->first_ua;
00327 }
```

```

00432 }
00433
00434 static inline int pch_dev_range_get_index(pch_dev_range_t *dr, pch_devib_t *devib) {
00435     assert(dr->cu == pch_dev_get_cu(devib));
00436
00437     int i = pch_dev_range_get_index_nocheck(dr, devib);
00438     if (i < 0 || i >= dr->num_devices)
00439         return -1;
00440
00441     return i;
00442 }
00443
00444 static inline int pch_dev_range_get_index_required(pch_dev_range_t *dr, pch_devib_t *devib) {
00445     int i = pch_dev_range_get_index(dr, devib);
00446     if (i < 0)
00447         panic("devib not found in dev_range");
00448
00449     return i;
00450 }
00451
00452 static inline pch_devib_t *pch_dev_range_get_devib_by_index(pch_dev_range_t *dr, uint i) {
00453     assert(dr->cu);
00454
00455     pch_unit_addr_t ua = pch_dev_range_get_ua(dr, i);
00456     return pch_get_devib(dr->cu, ua);
00457 }
00458
00459 static inline pch_devib_t *pch_dev_range_get_devib_by_ua_nocheck(pch_dev_range_t *dr, pch_unit_addr_t
00460 ua) {
00461     assert(dr->cu);
00462
00463     return pch_get_devib(dr->cu, ua);
00464 }
00465
00466 static inline pch_devib_t *pch_dev_range_get_devib_by_ua(pch_dev_range_t *dr, pch_unit_addr_t ua) {
00467     assert(dr->cu);
00468
00469     if (ua < dr->first_ua
00470         || (uint)ua >= (uint)dr->first_ua + (uint)dr->num_devices) {
00471         return NULL;
00472     }
00473
00474     return pch_get_devib(dr->cu, ua);
00475 }
00476
00477 static inline pch_devib_t *pch_dev_range_get_devib_by_ua_required(pch_dev_range_t *dr, pch_unit_addr_t
00478 ua) {
00479     assert(dr->cu);
00480
00481     if (ua < dr->first_ua
00482         || (uint)ua >= (uint)dr->first_ua + (uint)dr->num_devices) {
00483         panic("ua not in dev_range");
00484     }
00485
00486     return pch_get_devib(dr->cu, ua);
00487 }
00488
00489 static inline void pch_dev_range_init(pch_dev_range_t *drout, pch_cu_t *cu, pch_unit_addr_t first_ua,
00490 uint16_t num_devices) {
00491     assert(cu);
00492     assert((uint)first_ua + (uint)num_devices <= cu->num_devibs);
00493
00494     drout->cu = cu;
00495     drout->num_devices = num_devices;
00496     drout->first_ua = first_ua;
00497 }
00498
00499 static inline void pch_dev_range_set_callback(pch_dev_range_t *dr, pch_cbindex_t cbindex) {
00500     assert(dr->cu);
00501
00502     for (uint i = 0; i < dr->num_devices; i++) {
00503         pch_devib_t *devib = pch_dev_range_get_devib_by_index(dr, i);
00504         pch_dev_set_callback(devib, cbindex);
00505     }
00506 }
00507
00508 #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, EINVALDCALLBACK = 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 value 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.](#)

```
00001 /*
00002  * Copyright (c) 2025 Malcolm Beattie
00003  * SPDX-License-Identifier: MIT
00004 */
00005
00006 #ifndef _PCH CU DEV API H
00007 #define _PCH CU DEV API H
00008
00009 #include "picochan/devib.h"
00010
00019
00020 // Main API for dev implementation, slightly higher level than
00021 // devib ones. They return negative error values on error
00022 // (e.g. -EINVAL). They do various parameter checks and return
00023 // errors instead of asserting like the low-level API does. Those
00024 // with cbindex_opt arguments leave the devib cbindex field alone
00025 // if called with a negative value, otherwise they validate it
00026 // as a callback cbindex and set the field or return a negative
00027 // error value, as appropriate. For sends (of data or zeroes), the
00028 // length sent is validated to be under the CSS-advertised window
00029 // (devib->size) and capped at that if not, with the actual count
```

```

00030 // returned. Many functions are variants of the full generic ones
00031 // that simply specialise the callback and flags fields.
00032 // Values between 1 and 255 are typically used to fit into the ASC
00033 // byte of a pch_dev_sense_t with sense code
00034 // PCH_DEV_SENSE_COMMAND_REJECT. ECANCEL is associated with sense
00035 // code PCH_DEV_SENSE_CANCEL.
00036
00037 enum {
00038     ENOSUCHERROR          = 1,
00039     EINVALIDCALLBACK      = 2,
00040     ENOTSTARTED           = 3,
00041     ECMNDNOTREAD         = 4,
00042     ECMDNOTWRITE          = 5,
00043     EWRITETOOBIG          = 6,
00044     EINVALIDSTATUS        = 7,
00045     EINVALIDIDDEV         = 8,
00046     EINVALIDIDCMD         = 9,
00047     EINVALIDVALUE          = 10,
00048     EDATALENZERO          = 11,
00049     EBUFFERTOOSHORT       = 12,
00050     //
00051     ECANCEL               = 256
00052 };
00053
00054 // dev API with fully general arguments
00055
00056 int pch_dev_set_callback(pch_devib_t *devib, int cbindex_opt);
00057
00058 int pch_dev_send_then(pch_devib_t *devib, void *srcaddr, uint16_t n, proto_chop_flags_t flags, int
00059 cbindex_opt);
00060
00061 int pch_dev_send_zeroes_then(pch_devib_t *devib, uint16_t n, proto_chop_flags_t flags, int
00062 cbindex_opt);
00063
00064 int pch_dev_receive_then(pch_devib_t *devib, void *dstaddr, uint16_t size, int cbindex_opt);
00065
00066 int pch_dev_update_status_advert_then(pch_devib_t *devib, uint8_t devs, void *dstaddr, uint16_t size,
00067 int cbindex_opt);
00068
00069 // dev API convenience functions with some fixed arguments:
00070 // * Omitting _then avoids setting devib callback by hardcoding -1
00071 // as the cbindex_opt argument of the full _then function.
00072 // * For send and send_zeroes family, the flags argument is set to
00073 //   * PROTO_CHOP_FLAG_END for the _final variant,
00074 //   * PROTO_CHOP_FLAG_RESPONSE_REQUIRED for the _respond variant
00075 //   * 0 for the _norespond variant
00076 // * For pch_dev_update_status_ok family, call the corresponding
00077 //   pch_dev_update_status_ function with DeviceEnd|ChannelEnd
00078 // * For pch_dev_update_status_error family, set devib->sense to the
00079 //   sense argument then call the corresponding pch_dev_update_status_
00080 //   function with a device status of DeviceEnd|ChannelEnd|UnitCheck
00081 int pch_dev_send(pch_devib_t *devib, void *srcaddr, uint16_t n, proto_chop_flags_t flags);
00082 int pch_dev_send_final(pch_devib_t *devib, void *srcaddr, uint16_t n);
00083 int pch_dev_send_final_then(pch_devib_t *devib, void *srcaddr, uint16_t n, int cbindex_opt);
00084 int pch_dev_send_respond(pch_devib_t *devib, void *srcaddr, uint16_t n);
00085 int pch_dev_send_respond_then(pch_devib_t *devib, void *srcaddr, uint16_t n, int cbindex_opt);
00086 int pch_dev_send_norespond(pch_devib_t *devib, void *srcaddr, uint16_t n);
00087 int pch_dev_send_norespond_then(pch_devib_t *devib, void *srcaddr, uint16_t n, int cbindex_opt);
00088 int pch_dev_send_zeroes(pch_devib_t *devib, uint16_t n, proto_chop_flags_t flags);
00089 int pch_dev_send_zeroes_then(pch_devib_t *devib, uint16_t n, int cbindex_opt);
00090 int pch_dev_send_zeroes_respond(pch_devib_t *devib, uint16_t n);
00091 int pch_dev_send_zeroes_respond_then(pch_devib_t *devib, uint16_t n, int cbindex_opt);
00092 int pch_dev_send_zeroes_norespond(pch_devib_t *devib, uint16_t n);
00093 int pch_dev_receive(pch_devib_t *devib, void *dstaddr, uint16_t size);
00094 int pch_dev_update_status_then(pch_devib_t *devib, uint8_t devs, int cbindex_opt);
00095 int pch_dev_update_status(pch_devib_t *devib, uint8_t devs);
00096 int pch_dev_update_status_advert(pch_devib_t *devib, uint8_t devs, void *dstaddr, uint16_t size);
00097 int pch_dev_update_status_ok_then(pch_devib_t *devib, int cbindex_opt);
00098 int pch_dev_update_status_ok(pch_devib_t *devib);
00099 int pch_dev_update_status_ok_advert(pch_devib_t *devib, void *dstaddr, uint16_t size);
00100 int pch_dev_update_status_error_advert_then(pch_devib_t *devib, pch_dev_sense_t sense, void *dstaddr,
00101 uint16_t size, int cbindex_opt);
00102 int pch_dev_update_status_error_then(pch_devib_t *devib, pch_dev_sense_t sense, int cbindex_opt);
00103 int pch_dev_update_status_error_advert(pch_devib_t *devib, pch_dev_sense_t sense, void *dstaddr,
00104 uint16_t size);
00105 int pch_dev_update_status_error(pch_devib_t *devib, pch_dev_sense_t sense);
00106
00107 #endif
00108
00109
00110
00111
00112
00113
00114
00115
00116
00117
00118
00119
00120
00121
00122
00123
00124
00125
00126
00127
00128
00129
00130
00131
00132
00133
00134
00135
00136
00137
00138
00139
00140
00141
00142
00143
00144
00145
00146
00147
00148
00149
00150
00151
00152
00153
00154
00155
00156
00157
00158
00159
00160
00161
00162
00163
00164
00165
00166
00167
00168
00169
00170
00171
00172
00173
00174
00175
00176
00177
00178
00179
00180
00181
00182
00183
00184
00185
00186
00187
00188
00189
00190
00191
00192
00193
00194
00195
00196
00197
00198
00199
00200
00201
00202
00203
00204
00205
00206
00207
00208
00209
00210
00211
00212
00213
00214
00215
00216
00217
00218
00219
00220
00221
00222
00223
00224
00225
00226
00227
00228
00229
00230
00231
00232
00233
00234
00235
00236
00237
00238
00239
00240
00241
00242
00243
00244
00245
00246
00247
00248
00249
00250
00251
00252
00253
00254
00255
00256
00257
00258
00259
00260
00261
00262
00263
00264
00265
00266
00267
00268
00269
00270
00271
00272
00273
00274
00275
00276
00277
00278
00279
00280
00281
00282
00283
00284
00285
00286
00287
00288
00289
00290
00291
00292
00293
00294
00295
00296
00297
00298
00299
00300
00301
00302
00303
00304
00305
00306
00307
00308
00309
00310
00311
00312
00313
00314
00315
00316
00317
00318
00319
00320
00321
00322
00323
00324
00325
00326
00327
00328
00329
00330
00331
00332
00333
00334
00335
00336
00337
00338
00339
00340
00341
00342
00343
00344
00345
00346
00347
00348
00349
00350
00351
00352
00353
00354
00355
00356
00357
00358
00359
00360
00361
00362
00363
00364
00365
00366
00367
00368
00369
00370
00371
00372
00373
00374
00375
00376
00377
00378
00379
00380
00381
00382
00383
00384
00385
00386
00387
00388
00389
00390
00391
00392
00393
00394
00395
00396
00397
00398
00399
00400
00401
00402
00403
00404
00405
00406
00407
00408
00409
00410
00411
00412
00413
00414
00415
00416
00417
00418
00419
00420
00421
00422
00423
00424
00425
00426
00427
00428
00429
00430
00431
00432
00433
00434
00435
00436
00437
00438
00439
00440
00441
00442
00443
00444
00445
00446
00447
00448
00449
00450
00451
00452
00453
00454
00455
00456
00457
00458
00459
00460
00461
00462
00463
00464
00465
00466
00467
00468
00469
00470
00471
00472
00473
00474
00475
00476
00477
00478
00479
00480
00481
00482
00483
00484
00485
00486
00487
00488
00489
00490
00491
00492
00493
00494
00495
00496
00497
00498
00499
00500
00501
00502
00503
00504
00505
00506
00507
00508
00509
00510
00511
00512
00513
00514
00515
00516
00517
00518
00519
00520
00521
00522
00523
00524
00525
00526
00527
00528
00529
00530
00531
00532
00533
00534
00535
00536
00537
00538
00539
00540
00541
00542
00543
00544
00545
00546
00547
00548
00549
00550
00551
00552
00553
00554
00555
00556
00557
00558
00559
00560
00561
00562
00563
00564
00565
00566
00567
00568
00569
00570
00571
00572
00573
00574
00575
00576
00577
00578
00579
00580
00581
00582
00583
00584
00585
00586
00587
00588
00589
00590
00591
00592
00593
00594
00595
00596
00597
00598
00599
00600
00601
00602
00603
00604
00605
00606
00607
00608
00609
00610
00611
00612
00613
00614
00615
00616
00617
00618
00619
00620
00621
00622
00623
00624
00625
00626
00627
00628
00629
00630
00631
00632
00633
00634
00635
00636
00637
00638
00639
00640
00641
00642
00643
00644
00645
00646
00647
00648
00649
00650
00651
00652
00653
00654
00655
00656
00657
00658
00659
00660
00661
00662
00663
00664
00665
00666
00667
00668
00669
00670
00671
00672
00673
00674
00675
00676
00677
00678
00679
00680
00681
00682
00683
00684
00685
00686
00687
00688
00689
00690
00691
00692
00693
00694
00695
00696
00697
00698
00699
00700
00701
00702
00703
00704
00705
00706
00707
00708
00709
00710
00711
00712
00713
00714
00715
00716
00717
00718
00719
00720
00721
00722
00723
00724
00725
00726
00727
00728
00729
00730
00731
00732
00733
00734
00735
00736
00737
00738
00739
00740
00741
00742
00743
00744
00745
00746
00747
00748
00749
00750
00751
00752
00753
00754
00755
00756
00757
00758
00759
00760
00761
00762
00763
00764
00765
00766
00767
00768
00769
00770
00771
00772
00773
00774
00775
00776
00777
00778
00779
00780
00781
00782
00783
00784
00785
00786
00787
00788
00789
00790
00791
00792
00793
00794
00795
00796
00797
00798
00799
00800
00801
00802
00803
00804
00805
00806
00807
00808
00809
00810
00811
00812
00813
00814
00815
00816
00817
00818
00819
00820
00821
00822
00823
00824
00825
00826
00827
00828
00829
00830
00831
00832
00833
00834
00835
00836
00837
00838
00839
00840
00841
00842
00843
00844
00845
00846
00847
00848
00849
00850
00851
00852
00853
00854
00855
00856
00857
00858
00859
00860
00861
00862
00863
00864
00865
00866
00867
00868
00869
00870
00871
00872
00873
00874
00875
00876
00877
00878
00879
00880
00881
00882
00883
00884
00885
00886
00887
00888
00889
00890
00891
00892
00893
00894
00895
00896
00897
00898
00899
00900
00901
00902
00903
00904
00905
00906
00907
00908
00909
00910
00911
00912
00913
00914
00915
00916
00917
00918
00919
00920
00921
00922
00923
00924
00925
00926
00927
00928
00929
00930
00931
00932
00933
00934
00935
00936
00937
00938
00939
00940
00941
00942
00943
00944
00945
00946
00947
00948
00949
00950
00951
00952
00953
00954
00955
00956
00957
00958
00959
00960
00961
00962
00963
00964
00965
00966
00967
00968
00969
00970
00971
00972
00973
00974
00975
00976
00977
00978
00979
00980
00981
00982
00983
00984
00985
00986
00987
00988
00989
00990
00991
00992
00993
00994
00995
00996
00997
00998
00999
00999
01000
01001
01002
01003
01004
01005
01006
01007
01008
01009
010010
010011
010012
010013
010014
010015
010016
010017
010018
010019
010020
010021
010022
010023
010024
010025
010026
010027
010028
010029
010030
010031
010032
010033
010034
010035
010036
010037
010038
010039
010040
010041
010042
010043
010044
010045
010046
010047
010048
010049
010050
010051
010052
010053
010054
010055
010056
010057
010058
010059
010060
010061
010062
010063
010064
010065
010066
010067
010068
010069
010070
010071
010072
010073
010074
010075
010076
010077
010078
010079
010080
010081
010082
010083
010084
010085
010086
010087
010088
010089
010090
010091
010092
010093
010094
010095
010096
010097
010098
010099
0100100
0100101
0100102
0100103
0100104
0100105
0100106
0100107
0100108
0100109
0100110
0100111
0100112
0100113
0100114
0100115
0100116
0100117
0100118
0100119
0100120
0100121
0100122
0100123
0100124
0100125
0100126
0100127
0100128
0100129
0100130
0100131
0100132
0100133
0100134
0100135
0100136
0100137
0100138
0100139
0100140
0100141
0100142
0100143
0100144
0100145
0100146
0100147
0100148
0100149
0100150
0100151
0100152
0100153
0100154
0100155
0100156
0100157
0100158
0100159
0100160
0100161
0100162
0100163
0100164
0100165
0100166
0100167
0100168
0100169
0100170
0100171
0100172
0100173
0100174
0100175
0100176
0100177
0100178
0100179
0100180
0100181
0100182
0100183
0100184
0100185
0100186
0100187
0100188
0100189
0100190
0100191
0100192
0100193
0100194
0100195
0100196
0100197
0100198
0100199
0100200
0100201
0100202
0100203
0100204
0100205
0100206
0100207
0100208
0100209
0100210
0100211
0100212
0100213
0100214
0100215
0100216
0100217
0100218
0100219
0100220
0100221
0100222
0100223
0100224
0100225
0100226
0100227
0100228
0100229
0100230
0100231
0100232
0100233
0100234
0100235
0100236
0100237
0100238
0100239
0100240
0100241
0100242
0100243
0100244
0100245
0100246
0100247
0100248
0100249
0100250
0100251
0100252
0100253
0100254
0100255
0100256
0100257
0100258
0100259
0100260
0100261
0100262
0100263
0100264
0100265
0100266
0100267
0100268
0100269
0100270
0100271
0100272
0100273
0100274
0100275
0100276
0100277
0100278
0100279
0100280
0100281
0100282
0100283
0100284
0100285
0100286
0100287
0100288
0100289
0100290
0100291
0100292
0100293
0100294
0100295
0100296
0100297
0100298
0100299
0100300
0100301
0100302
0100303
0100304
0100305
0100306
0100307
0100308
0100309
0100310
0100311
0100312
0100313
0100314
0100315
0100316
0100317
0100318
0100319
0100320
0100321
0100322
0100323
0100324
0100325
0100326
0100327
0100328
0100329
0100330
0100331
0100332
0100333
0100334
0100335
0100336
0100337
0100338
0100339
0100340
0100341
0100342
0100343
0100344
0100345
0100346
0100347
0100348
0100349
0100350
0100351
0100352
0100353
0100354
0100355
0100356
0100357
0100358
0100359
0100360
0100361
0100362
0100363
0100364
0100365
0100366
0100367
0100368
0100369
0100370
0100371
0100372
0100373
0100374
0100375
0100376
0100377
0100378
0100379
0100380
0100381
0100382
0100383
0100384
0100385
0100386
0100387
0100388
0100389
0100390
0100391
0100392
0100393
0100394
0100395
0100396
0100397
0100398
0100399
0100400
0100401
0100402
0100403
0100404
0100405
0100406
0100407
0100408
0100409
0100410
0100411
0100412
0100413
0100414
0100415
0100416
0100417
0100418
0100419
0100420
0100421
0100422
0100423
0100424
0100425
0100426
0100427
0100428
0100429
0100430
0100431
0100432
0100433
0100434
0100435
0100436
0100437
0100438
0100439
0100440
0100441
0100442
0100443
0100444
0100445
0100446
0100447
0100448
0100449
0100450
0100451
0100452
0100453
0100454
0100455
0100456
0100457
0100458
0100459
0100460
0100461
0100462
0100463
0100464
0100465
0100466
0100467
0100468
0100469
0100470
0100471
0100472
0100473
0100474
0100475
0100476
0100477
0100478
0100479
0100480
0100481
0100482
0100483
0100484
0100485
0100486
0100487
0100488
0100489
0100490
0100491
0100492
0100493
0100494
0100495
0100496
0100497
0100498
0100499
0100500
0100501
0100502
0100503
0100504
0100505
0100506
0100507
0100508
0100509
0100510
0100511
0100512
0100513
0100514
0100515
0100516
0100517
0100518
0100519
0100520
0100521
0100522
0100523
0100524
0100525
0100
```

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.](#)

```
00001 /*
00002  * Copyright (c) 2025 Malcolm Beattie
00003  * SPDX-License-Identifier: MIT
00004 */
00005
00006 #ifndef _PCH_CU_DEV_SENSE_H
00007 #define _PCH_CU_DEV_SENSE_H
00008
00014
00018 typedef struct __attribute__((__packed__, __aligned__(4))) pch\_dev\_sense {
00019     uint8_t flags;
00020     uint8_t code;
00021     uint8_t asc;
00022     uint8_t ascq;
00023 } pch\_dev\_sense\_t;
00024
00025 #define PCH_DEV_SENSE_COMMAND_REJECT      0x80
00026 #define PCH_DEV_SENSE_INTERVENTION_REQUIRED 0x40
00027 #define PCH_DEV_SENSE_BUS_OUT_CHECK        0x20
00028 #define PCH_DEV_SENSE_EQUIPMENT_CHECK      0x10
00029 #define PCH_DEV_SENSE_DATA_CHECK          0x08
00030 #define PCH_DEV_SENSE_OVERRUN            0x04
00031 #define PCH_DEV_SENSE_PROTO_ERROR         0x02
00032 #define PCH_DEV_SENSE_CANCEL              0x01
00033
00034 #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.

```
00001 /*
00002  * Copyright (c) 2025 Malcolm Beattie
00003  * SPDX-License-Identifier: MIT
00004 */
00005
00006 #ifndef _PCH_CU_DEVIB_H
00007 #define _PCH_CU_DEVIB_H
00008
00009 #include "pico/platform/compiler.h"
00010 #include "picochan/ids.h"
00011 #include "picochan/dev_status.h"
00012 #include "picochan/dev_sense.h"
00013 #include "proto/chop.h"
00014 #include "proto/payload.h"
00015
00021
```

```

00027 typedef uint8_t pch_cbindex_t;
00028
00029 #define PCH_DEVIB_CALLBACK_DEFAULT 0
00030 #define PCH_DEVIB_CALLBACK_NOOP 255
00031
00032 #define MAX_DEVIB_CALLBACKS 254
00033
00034 #define NUM_DEVIB_CALLBACKS 16
00035 static_assert(NUM_DEVIB_CALLBACKS <= MAX_DEVIB_CALLBACKS,
00036     "NUM_DEVIB_CALLBACKS must not exceed MAX_DEVIB_CALLBACKS");
00037
00038 static_assert(sizeof(pch_dev_sense_t) == 4,
00039     "pch_dev_sense_t must be 4 bytes");
00040
00041 typedef struct __aligned(4) pch_devib {
00042     pch_unit_addr_t next;
00043     pch_cbindex_t cbindex;
00044     uint16_t size;
00045     proto_chop_t op;
00046     uint8_t flags;
00047     proto_payload_t payload;
00048     uint32_t addr;
00049     pch_dev_sense_t sense;
00050 } pch_devib_t;
00051
00052
00053 #define PCH_DEVIB_SPACE_SHIFT (31U - __builtin_clz(2 * sizeof(pch_devib_t) - 1))
00054
00055 static_assert(__builtin_constant_p(PCH_DEVIB_SPACE_SHIFT),
00056     "__builtin_clz() did not produce compile-time constant for PCH_DEVIB_SPACE_SHIFT");
00057
00058
00059 #define PCH_DEVIB_FLAG_STARTED 0x80
00060 #define PCH_DEVIB_FLAG_CMD_WRITE 0x40
00061 #define PCH_DEVIB_FLAG_RX_DATA_REQUIRED 0x20
00062 #define PCH_DEVIB_FLAG_TX_CALLBACK 0x10
00063 #define PCH_DEVIB_FLAG_TRACED 0x08
00064 #define PCH_DEVIB_FLAG_STOPPING 0x04
00065
00066
00067 static inline bool pch_devib_is_started(pch_devib_t *devib) {
00068     return devib->flags & PCH_DEVIB_FLAG_STARTED;
00069 }
00070
00071 static inline bool pch_devib_is_cmd_write(pch_devib_t *devib) {
00072     return devib->flags & PCH_DEVIB_FLAG_CMD_WRITE;
00073 }
00074
00075 static inline bool pch_devib_is_traced(pch_devib_t *devib) {
00076     return devib->flags & PCH_DEVIB_FLAG_TRACED;
00077 }
00078
00079 static inline bool pch_devib_set_traced(pch_devib_t *devib, bool trace) {
00080     bool old_trace = pch_devib_is_traced(devib);
00081     if (trace)
00082         devib->flags |= PCH_DEVIB_FLAG_TRACED;
00083     else
00084         devib->flags &= ~PCH_DEVIB_FLAG_TRACED;
00085
00086     return old_trace;
00087 }
00088
00089 static inline bool pch_devib_is_stopping(pch_devib_t *devib) {
00090     return devib->flags & PCH_DEVIB_FLAG_STOPPING;
00091 }
00092
00093
00094 // Forward declaration of pch_cu_t for identifying devib by
00095 // (pch_cu_t, pch_unit_addr_t) for callbacks and dev implementations.
00096 typedef struct pch_cu pch_cu_t;
00097
00098 // Callbacks
00099
00100 typedef void (*pch_devib_callback_t)(pch_devib_t *devib);
00101
00102 extern pch_devib_callback_t pch_devib_callbacks[];
00103
00104 static inline bool pch_cbindex_is_callable(uint cbindex) {
00105     if (cbindex == PCH_DEVIB_CALLBACK_NOOP)
00106         return true;
00107
00108     if (cbindex >= NUM_DEVIB_CALLBACKS)
00109         return false;
00110
00111     return pch_devib_callbacks[cbindex] != NULL;
00112 }
00113
00114 // Callback registration API
00115
00116 void pch_register_devib_callback(pch_cbindex_t n, pch_devib_callback_t cb);
00117

```

```
00160 pch_cbindex_t pch_register_unused_devib_callback(pch_devib_callback_t cb);
00161
00162 void pch_default_devib_callback(pch_devib_t *devib);
00163
00164 // Low-level API for dev implementation updating devib
00165
00177 static inline void pch_devib_prepare_callback(pch_devib_t *devib, pch_cbindex_t cbindex) {
00178     assert(pch_cbindex_is_callable(cbindex));
00179     devib->cbindex = cbindex;
00180 }
00181
00192 static inline void pch_devib_prepare_count(pch_devib_t *devib, uint16_t count) {
00193     devib->payload = proto_make_count_payload(count);
00194 }
00195
00210 static inline void pch_devib_prepare_write_data(pch_devib_t *devib, void *srcaddr, uint16_t n,
00211     proto_chop_flags_t flags) {
00212     assert(devib->flags & PCH_DEVIB_FLAG_STARTED);
00213     pch_devib_prepare_count(devib, n);
00214     devib->op = PROTO_CHOP_DATA | flags;
00215     devib->addr = (uint32_t)srcaddr;
00216 }
00217
00233 static inline void pch_devib_prepare_write_zeroes(pch_devib_t *devib, uint16_t n, proto_chop_flags_t
00234     flags) {
00235     assert(devib->flags & PCH_DEVIB_FLAG_STARTED);
00236     pch_devib_prepare_count(devib, n);
00237     // hard-code the ResponseRequired flag for now
00238     devib->op = PROTO_CHOP_DATA | PROTO_CHOP_FLAG_SKIP | flags;
00239 }
00240
00253 static inline void pch_devib_prepare_read_data(pch_devib_t *devib, void *dstaddr, uint16_t size) {
00254     assert(devib->flags & PCH_DEVIB_FLAG_STARTED);
00255     pch_devib_prepare_count(devib, size);
00256     devib->op = PROTO_CHOP_REQUEST_READ;
00257     devib->flags |= PCH_DEVIB_FLAG_RX_DATA_REQUIRED;
00258     devib->addr = (uint32_t)dstaddr;
00259 }
00260
00280 void pch_devib_prepare_update_status(pch_devib_t *devib, uint8_t devs, void *dstaddr, uint16_t size);
00281
00282 void pch_devib_send_or_queue_command(pch_devib_t *devib);
00283 #endif
```


Index

addr_count, 67
base/dmachan/dmachan_internal.h, 89
base/dmachan/dmachan_trace.h, 89
base/include/picochan/bsize.h, 90, 92
base/include/picochan/ccw.h, 93, 94
base/include/picochan/dev_status.h, 95, 96
base/include/picochan/dmachan.h, 96
base/include/picochan/dmachan_defs.h, 99
base/include/picochan/ids.h, 100
base/include/picochan/intcode.h, 101, 102
base/include/picochan/scsw.h, 102, 103
base/include/picochan/trc.h, 104, 106
base/include/picochan/trc_record_types.h, 107
base/include/picochan/trc_records.h, 108
base/include/picochan/txsm_state.h, 110
base/proto/chop.h, 110
base/proto/packet.h, 111, 112
base/proto/payload.h, 113
base/trc/bufferset.h, 113
base/trc/trace.h, 114
base/trc/trace_lock.h, 115
base/txsm/txsm.h, 115
bsize.h
 pch_bsize_t, 91
buffers
 pch_trc_bufferset, 79
Channel programs, 5
Channel Subsystem (CSS) API for Applications, 13
Compiling using Picochan, 11
Control Unit (CU) API for Device Drivers Overview, 17
CSS
 css_internal.h, 120
css, 67
css/ccw_fetch.h, 116
css/channel.h, 116, 117
css/css_internal.h, 119, 121
css/css_trace.h, 122
css/include/picochan/css.h, 123, 126
css/include/picochan/pmcw.h, 128, 130
css/include/picochan/schib.h, 130, 131
css/schib_dlist.h, 132
css/schib_internal.h, 132
css/schibs_lock.h, 133
css_internal.h
 CSS, 120
cu/callback.h, 133
cu/cu_internal.h, 134
cu/cus_trace.h, 134
cu/devibs_lock.h, 135
cu/include/picochan/cu.h, 136, 138
cu/include/picochan/dev_api.h, 141, 143
cu/include/picochan/dev_sense.h, 145
cu/include/picochan/devib.h, 146, 147
Design of Picochan, 21
dev_api.h
 pch_dev_call_final_then, 143
 pch_dev_call_or_reject_then, 143
dmachan_1way_config, 68
dmachan_cmd, 68
dmachan_config, 68
dmachan_link, 69
dmachan_rx_channel, 69
dmachan_tx_channel, 69
dmairqix_config, 70
intcode.h
 pch_intcode_t, 101
internal_css, 40
 pch_chp_t, 41
internal_proto, 39
 proto_packet_t, 40
internal_trc, 38
 PCH_CONFIG_ENABLE_TRACE, 39
 pch_trc_bufferset_t, 39
 pch_trc_timestamp_t, 39
irqnum
 pch_trc_bufferset, 79
MAX_DEVIB_CALLBACKS
 picochan_cu, 55
NUM_DEVIB_CALLBACKS
 picochan_cu, 55
num_devibs
 pch_cu, 73
pch_bsize, 70
pch_bsize_decode_inline
 picochan_base, 37
pch_bsize_decode_raw_inline
 picochan_base, 37
pch_bsize_encode_inline
 picochan_base, 37
pch_bsize_encode_raw_inline
 picochan_base, 37
pch_bsize_encode_index_inline
 picochan_base, 37
pch_bsize_t

bsize.h, 91
 pch_bsize_wrap
 picochan_base, 38
 PCH_BSIZE_ZERO
 picochan_base, 35
 pch_bsizex, 71
 pch_bsizex_t
 picochan_base, 35
 pch_ccw, 71
 pch_ccw_get_addr
 picochan_css, 44
 pch_ccw_t
 picochan_base, 35
 pch_chp, 72
 pch_chp_alloc
 picochan_css, 44
 pch_chp_auto_configure_uartchan
 picochan_css, 44
 pch_chp_configure_memchan
 picochan_css, 45
 pch_chp_configure_uartchan
 picochan_css, 45
 pch_chp_get_tx_channel
 picochan_css, 45
 pch_chp_set_trace
 picochan_css, 45
 pch_chp_start
 picochan_css, 46
 pch_chp_t
 internal_css, 41
 pch_chpid_t
 picochan_base, 35
 PCH_CONFIG_ENABLE_TRACE
 internal_trc, 39
 pch_css_init
 picochan_css, 46
 pch_css_set_func_irq
 picochan_css, 46
 pch_css_set_io_callback
 picochan_css, 46
 pch_css_set_io_irq
 picochan_css, 47
 pch_css_set_trace
 picochan_css, 47
 pch_css_start
 picochan_css, 48
 pch_cu, 72
 num_devibs, 73
 pch_cu_get_tx_channel
 picochan_cu, 57
 PCH_CU_INIT
 picochan_cu, 55
 pch_cu_init
 picochan_cu, 57
 pch_cu_register
 picochan_cu, 57
 pch_cu_start
 picochan_cu, 57
 pch_cu_t
 picochan_cu, 56
 pch_cus_auto_configure_uartcu
 picochan_css, 48
 pch_cus_init
 picochan_cu, 58
 pch_cus_memcu_configure
 picochan_cu, 58
 pch_cus_set_trace
 picochan_cu, 58
 pch_cus_trace_cu
 picochan_cu, 58
 pch_cus_trace_dev
 picochan_cu, 59
 pch_cus_uartcu_configure
 picochan_cu, 59
 pch_dev_call_final_then
 dev_api.h, 143
 pch_dev_call_or_reject_then
 dev_api.h, 143
 pch_dev_range, 74
 pch_dev_receive_then
 picochan_cu, 59
 pch_dev_send_then
 picochan_cu, 60
 pch_dev_send_zeroes_then
 picochan_cu, 61
 pch_dev_sense, 74
 pch_dev_set_callback
 picochan_cu, 62
 pch_devib, 74
 pch_devib_prepare_callback
 picochan_cu, 62
 pch_devib_prepare_count
 picochan_cu, 62
 pch_devib_prepare_read_data
 picochan_cu, 63
 pch_devib_prepare_update_status
 picochan_cu, 63
 pch_devib_prepare_write_data
 picochan_cu, 63
 pch_devib_prepare_write_zeroes
 picochan_cu, 64
 pch_devib_t
 picochan_cu, 56
 pch_dma_irq_index_t
 picochan_base, 36
 pch_dmaid_t
 picochan_base, 36
 pch_get_cu
 picochan_cu, 64
 pch_get_devib
 picochan_cu, 64
 pch_intcode, 75
 pch_intcode_t
 intcode.h, 101
 PCH_NUM_CHANNELS
 picochan_css, 43

PCH_NUM_CUS
 picochan_cu, 56

PCH_NUM_ISCS
 picochan_css, 43

PCH_NUM_SCHIBS
 picochan_css, 44

pch_pmcw, 76

pch_pmcw_t
 pmcw.h, 129

pch_register_devib_callback
 picochan_cu, 64

pch_register_unused_devib_callback
 picochan_cu, 65

pch_sch_cancel
 picochan_css, 48

pch_sch_halt
 picochan_css, 48

pch_sch_modify
 picochan_css, 49

pch_sch_modify_enabled
 picochan_css, 49

pch_sch_modify_flags
 picochan_css, 49

pch_sch_modify_intparm
 picochan_css, 49

pch_sch_modify_isc
 picochan_css, 50

pch_sch_modify_traced
 picochan_css, 50

pch_sch_resume
 picochan_css, 50

pch_sch_run_wait
 picochan_css, 50

pch_sch_run_wait_timeout
 picochan_css, 50

pch_sch_start
 picochan_css, 51

pch_sch_store
 picochan_css, 51

pch_sch_store_pmcw
 picochan_css, 51

pch_sch_store_scsw
 picochan_css, 51

pch_sch_test
 picochan_css, 52

pch_sch_wait
 picochan_css, 52

pch_sch_wait_timeout
 picochan_css, 52

pch_schib, 76

pch_schib_mda, 77

pch_schib_mda_t
 schib.h, 131

pch_schib_t
 picochan_base, 36

pch_scsw, 78

pch_scsw_t
 scsw.h, 103

pch_test_pending_interruption
 picochan_css, 52

pch_trc_bufferset, 78
 buffers, 79
 irqnum, 79

pch_trc_bufferset_t
 internal_trc, 39

pch_trc_header, 80

PCH_TRC_RT
 trc.h, 106

pch_trc_timestamp, 80

pch_trc_timestamp_t
 internal_trc, 39

pch_trdata_address_change, 80

pch_trdata_byte, 81

pch_trdata_ccw_addr_sid, 81

pch_trdata_chp_alloc, 81

pch_trdata_cu_register, 81

pch_trdata_cus_call_callback, 82

pch_trdata_cus_init_mem_channel, 82

pch_trdata_cus_tx_complete, 82

pch_trdata_dev, 82

pch_trdata_dev_byte, 83

pch_trdata_dma_init, 83

pch_trdata_dmachan, 83

pch_trdata_dmachan_memstate, 83

pch_trdata_dmachan_segment, 84

pch_trdata_dmachan_segment_memstate, 84

pch_trdata_func_irq, 84

pch_trdata_id_byte, 84

pch_trdata_id_irq, 85

pch_trdata_intcode_scsw, 85

pch_trdata_irq_handler, 85

pch_trdata_irqnum_opt, 85

pch_trdata_scsw_sid_cc, 86

pch_trdata_sid_byte, 86

pch_trdata_word_byte, 86

pch_trdata_word_dev, 86

pch_trdata_word_sid, 87

pch_trdata_word_sid_byte, 87

pch_txsm, 87

pch_unit_addr_t
 picochan_base, 36

Picochan - a channel subsystem for Raspberry Pi Pico, 1

picochan_base, 33

 pch_bsize_decode_inline, 37

 pch_bsize_decode_raw_inline, 37

 pch_bsize_encode_inline, 37

 pch_bsize_encode_raw_inline, 37

 pch_bsize_encodex_inline, 37

 pch_bsize_wrap, 38

 PCH_BSIZE_ZERO, 35

 pch_bsizex_t, 35

 pch_ccw_t, 35

 pch_chpid_t, 35

 pch_dma_irq_index_t, 36

 pch_dmaid_t, 36

pch_schib_t, 36
 pch_unit_addr_t, 36
 picochan_css, 41
 pch_ccw_get_addr, 44
 pch_chp_alloc, 44
 pch_chp_auto_configure_uartchan, 44
 pch_chp_configure_memchan, 45
 pch_chp_configure_uartchan, 45
 pch_chp_get_tx_channel, 45
 pch_chp_set_trace, 45
 pch_chp_start, 46
 pch_css_init, 46
 pch_css_set_func_irq, 46
 pch_css_set_io_callback, 46
 pch_css_set_io_irq, 47
 pch_css_set_trace, 47
 pch_css_start, 48
 pch_cus_auto_configure_uartcu, 48
 PCH_NUM_CHANNELS, 43
 PCH_NUM_ISCS, 43
 PCH_NUM_SCHIBS, 44
 pch_sch_cancel, 48
 pch_sch_halt, 48
 pch_sch_modify, 49
 pch_sch_modify_enabled, 49
 pch_sch_modify_flags, 49
 pch_sch_modify_intparm, 49
 pch_sch_modify_isc, 50
 pch_sch_modify_traced, 50
 pch_sch_resume, 50
 pch_sch_run_wait, 50
 pch_sch_run_wait_timeout, 50
 pch_sch_start, 51
 pch_sch_store, 51
 pch_sch_store_pmcw, 51
 pch_sch_store_scsw, 51
 pch_sch_test, 52
 pch_sch_wait, 52
 pch_sch_wait_timeout, 52
 pch_test_pending_interruption, 52
 void, 53
 picochan_cu, 53
 MAX_DEVIB_CALLBACKS, 55
 NUM_DEVIB_CALLBACKS, 55
 pch_cu_get_tx_channel, 57
 PCH CU INIT, 55
 pch_cu_init, 57
 pch_cu_register, 57
 pch_cu_start, 57
 pch_cu_t, 56
 pch_cus_init, 58
 pch_cus_memcu_configure, 58
 pch_cus_set_trace, 58
 pch_cus_trace_cu, 58
 pch_cus_trace_dev, 59
 pch_cus_uartcu_configure, 59
 pch_dev_receive_then, 59
 pch_dev_send_then, 60
 pch_dev_send_zeroes_then, 61
 pch_dev_set_callback, 62
 pch_devib_prepare_callback, 62
 pch_devib_prepare_count, 62
 pch_devib_prepare_read_data, 63
 pch_devib_prepare_update_status, 63
 pch_devib_prepare_write_data, 63
 pch_devib_prepare_write_zeroes, 64
 pch_devib_t, 56
 pch_get_cu, 64
 pch_get_devib, 64
 PCH_NUM_CUS, 56
 pch_register_devib_callback, 64
 pch_register_unused_devib_callback, 65
 pmcw.h
 pch_pmcw_t, 129
 proto_packet, 87
 proto_packet_t
 internal_proto, 40
 proto_parsed_devstatus_payload, 88
 proto_payload, 88
 schib.h
 pch_schib_mda_t, 131
 scsw.h
 pch_scsw_t, 103
 Tracing, 23
 trc.h
 PCH TRC RT, 106
 ua_slist, 88
 void
 picochan_css, 53