n.danilyuk Add new settings in api for uart History

1 contributor

Raw Blame

1612 lines (1405 sloc) 65.2 KB

```
/**
 * $Id: $
 *
 * @file rp.h
 * @brief Red Pitaya library API interface
 *
 * @Author Red Pitaya
 *
 * (c) Red Pitaya  http://www.redpitaya.com
 *
 * This part of code is written in C programming language.
 * Please visit http://en.wikipedia.org/wiki/C_(programming_language)
 * for more details on the language used herein.
 */

#ifndef __RP_H
#define __RP_H

#ifdef __cplusplus
extern "C" {
#endif

#include <stdint.h>
#include <stdbool.h>

#define RP_MODEL "Z10"
```

```
27  #define ADC_SAMPLE_RATE 125e6
28  #define ADC_BITS 14
29  #define ADC_REG_BITS 14
30  #define ADC_BITS_MASK 0x3FFF
31  #define ADC_REG_BITS_MASK 0x3FFF
32  #define DAC_FREQUENCY 125e6
33
34  #define ADC_BUFFER_SIZE         (16 * 1024)
35  #define BUFFER_LENGTH           (16 * 1024)
36
37  /** @name Error codes
38   *  Various error codes returned by the API.
39   */
40  ///@{
41
42  /** Success */
43  #define RP_OK     0
44  /** Failed to Open EEPROM Device */
45  #define RP_EOED   1
46  /** Failed to Open Memory Device */
47  #define RP_EOMD   2
48  /** Failed to Close Memory Device*/
49  #define RP_ECMD   3
50  /** Failed to Map Memory Device */
51  #define RP_EMMD   4
52  /** Failed to Unmap Memory Device */
53  #define RP_EUMD   5
54  /** Value Out Of Range */
55  #define RP_EOOR   6
56  /** LED Input Direction is not valid */
57  #define RP_ELID   7
58  /** Modifying Read Only field */
59  #define RP_EMRO   8
60  /** Writing to Input Pin is not valid */
61  #define RP_EWIP   9
62  /** Invalid Pin number */
63  #define RP_EPN    10
64  /** Uninitialized Input Argument */
65  #define RP_UIA    11
66  /** Failed to Find Calibration Parameters */
67  #define RP_FCA    12
68  /** Failed to Read Calibration Parameters */
69  #define RP_RCA    13
70  /** Buffer too small */
71  #define RP_BTS    14
72  /** Invalid parameter value */
73  #define RP_EIPV   15
74  /** Unsupported Feature */
```

```
75  #define RP_EUF    16
76  /** Data not normalized */
77  #define RP_ENN    17
78  /** Failed to open bus */
79  #define RP_EFOB   18
80  /** Failed to close bus */
81  #define RP_EFCB   19
82  /** Failed to acquire bus access */
83  #define RP_EABA   20
84  /** Failed to read from the bus */
85  #define RP_EFRB   21
86  /** Failed to write to the bus */
87  #define RP_EFWB   22
88  /** Extension module not connected */
89  #define RP_EMNC   23
90  /** Command not supported */
91  #define RP_NOTS   24
92  /** Failed to init uart */
93  #define RP_EIU    25
94  /** Failed read from uart */
95  #define RP_ERU    26
96  /** Failed write to uart */
97  #define RP_EWU    27
98  /** Failed set settings to uart */
99  #define RP_ESU    28
100 /** Failed get settings from uart */
101 #define RP_EGU    29
```

```
102
103 #define SPECTR_OUT_SIG_LEN (2*1024)
104
105 ///@}
106
107 /**
108  * Type representing digital input output pins.
109  */
110 typedef enum {
111     RP_LED0,       //!< LED 0
112     RP_LED1,       //!< LED 1
113     RP_LED2,       //!< LED 2
114     RP_LED3,       //!< LED 3
115     RP_LED4,       //!< LED 4
116     RP_LED5,       //!< LED 5
117     RP_LED6,       //!< LED 6
118     RP_LED7,       //!< LED 7
119     RP_DIO0_P,     //!< DIO_P 0
120     RP_DIO1_P,     //!< DIO_P 1
121     RP_DIO2_P,     //!< DIO_P 2
122     RP_DIO3_P,     //!< DIO_P 3
123     RP_DIO4_P,     //!< DIO_P 4
124     RP_DIO5_P,     //!< DIO_P 5
125     RP_DIO6_P,     //!< DIO_P 6
126     RP_DIO7_P,     //!< DIO_P 7
127     RP_DIO0_N,     //!< DIO_N 0
128     RP_DIO1_N,     //!< DIO_N 1
129     RP_DIO2_N,     //!< DIO_N 2
130     RP_DIO3_N,     //!< DIO_N 3
131     RP_DIO4_N,     //!< DIO_N 4
132     RP_DIO5_N,     //!< DIO_N 5
133     RP_DIO6_N,     //!< DIO_N 6
134     RP_DIO7_N      //!< DIO_N 7
135 } rp_dpin_t;
136
137 /**
138  * Type representing pin's high or low state (on/off).
139  */
140 typedef enum {
141     RP_LOW, //!< Low state
142     RP_HIGH //!< High state
143 } rp_pinState_t;
144
145 /**
146  * Type representing pin's input or output direction.
147  */
148 typedef enum {
149     RP_IN, //!< Input direction
```

```
150     RP_OUT //!< Output direction
151 } rp_pinDirection_t;
152 
153 /**
154  * Type representing analog input output pins.
155  */
156 typedef enum {
157     RP_AOUT0,      //!< Analog output 0
158     RP_AOUT1,      //!< Analog output 1
159     RP_AOUT2,      //!< Analog output 2
160     RP_AOUT3,      //!< Analog output 3
161     RP_AIN0,       //!< Analog input 0
162     RP_AIN1,       //!< Analog input 1
163     RP_AIN2,       //!< Analog input 2
164     RP_AIN3        //!< Analog input 3
165 } rp_apin_t;
166 
167 typedef enum {
168     RP_WAVEFORM_SINE,       //!< Wave form sine
169     RP_WAVEFORM_SQUARE,     //!< Wave form square
170     RP_WAVEFORM_TRIANGLE,   //!< Wave form triangle
171     RP_WAVEFORM_RAMP_UP,    //!< Wave form sawtooth (/|)
172     RP_WAVEFORM_RAMP_DOWN,  //!< Wave form reversed sawtooth (|\)
173     RP_WAVEFORM_DC,         //!< Wave form dc
174     RP_WAVEFORM_PWM,        //!< Wave form pwm
175     RP_WAVEFORM_ARBITRARY,  //!< Use defined wave form
176     RP_WAVEFORM_DC_NEG      //!< Wave form negative dc
177 } rp_waveform_t;
178 
179 typedef enum {
180     RP_GEN_MODE_CONTINUOUS, //!< Continuous signal generation
181     RP_GEN_MODE_BURST,      //!< Signal is generated N times, wher N is defined with rp_GenBurs
182     RP_GEN_MODE_STREAM      //!< User can continuously write data to buffer
183 } rp_gen_mode_t;
184 
185 
186 typedef enum {
187     RP_GEN_TRIG_SRC_INTERNAL = 1,   //!< Internal trigger source
188     RP_GEN_TRIG_SRC_EXT_PE   = 2,   //!< External trigger source positive edge
189     RP_GEN_TRIG_SRC_EXT_NE   = 3,   //!< External trigger source negative edge
190     RP_GEN_TRIG_GATED_BURST  = 4    //!< External trigger gated burst
191 } rp_trig_src_t;
192 
193 /**
194  * Type representing Input/Output channels.
195  */
196 typedef enum {
197     RP_CH_1,    //!< Channel A
```

```
198     RP_CH_2     //!< Channel B
199 } rp_channel_t;
200
201
202 /**
203  * Type representing Input/Output channels in trigger.
204  */
205 typedef enum {
206     RP_T_CH_1,    //!< Channel A
207     RP_T_CH_2,    //!< Channel B
208     RP_T_CH_EXT,
209 } rp_channel_trigger_t;
210
211 /**
212  * The type represents the names of the coefficients in the filter.
213  */
214 typedef enum {
215     AA,    //!< AA
216     BB,    //!< BB
217     PP,    //!< PP
218     KK     //!< KK
219 } rp_eq_filter_cof_t;
220
221 /**
222  * Type representing acquire signal sampling rate.
223  */
224 typedef enum {
225     RP_SMP_125M     = 0,     //!< Sample rate 125Msps; Buffer time length 131us; Decimation 1
226     RP_SMP_15_625M  = 1,  //!< Sample rate 15.625Msps; Buffer time length 1.048ms; Decimation 8
227     RP_SMP_1_953M   = 2,   //!< Sample rate 1.953Msps; Buffer time length 8.388ms; Decimation 6
228     RP_SMP_122_070K = 3, //!< Sample rate 122.070ksps; Buffer time length 134.2ms; Decimation 1
229     RP_SMP_15_258K  = 4,  //!< Sample rate 15.258ksps; Buffer time length 1.073s; Decimation 81
230     RP_SMP_1_907K   = 5   //!< Sample rate 1.907ksps; Buffer time length 8.589s; Decimation 655
231 } rp_acq_sampling_rate_t;
232
233
234 /**
235  * Type representing decimation used at acquiring signal.
236  */
237 typedef enum {
238     RP_DEC_1,     //!< Sample rate 125Msps; Buffer time length 131us; Decimation 1
239     RP_DEC_8,     //!< Sample rate 15.625Msps; Buffer time length 1.048ms; Decimation 8
240     RP_DEC_64,    //!< Sample rate 1.953Msps; Buffer time length 8.388ms; Decimation 64
241     RP_DEC_1024,  //!< Sample rate 122.070ksps; Buffer time length 134.2ms; Decimation 1024
242     RP_DEC_8192,  //!< Sample rate 15.258ksps; Buffer time length 1.073s; Decimation 8192
243     RP_DEC_65536  //!< Sample rate 1.907ksps; Buffer time length 8.589s; Decimation 65536
244 } rp_acq_decimation_t;
245
```

```
246
247 /**
248  * Type representing different trigger sources used at acquiring signal.
249  */
250 typedef enum {
251     RP_TRIG_SRC_DISABLED, //!< Trigger is disabled
252     RP_TRIG_SRC_NOW,      //!< Trigger triggered now (immediately)
253     RP_TRIG_SRC_CHA_PE,   //!< Trigger set to Channel A threshold positive edge
254     RP_TRIG_SRC_CHA_NE,   //!< Trigger set to Channel A threshold negative edge
255     RP_TRIG_SRC_CHB_PE,   //!< Trigger set to Channel B threshold positive edge
256     RP_TRIG_SRC_CHB_NE,   //!< Trigger set to Channel B threshold negative edge
257     RP_TRIG_SRC_EXT_PE,   //!< Trigger set to external trigger positive edge (DIO0_P pin)
258     RP_TRIG_SRC_EXT_NE,   //!< Trigger set to external trigger negative edge (DIO0_P pin)
259     RP_TRIG_SRC_AWG_PE,   //!< Trigger set to arbitrary wave generator application positive edg
260     RP_TRIG_SRC_AWG_NE    //!< Trigger set to arbitrary wave generator application negative edg
261 } rp_acq_trig_src_t;
262
263
264 /**
265  * Type representing different trigger states.
266  */
267 typedef enum {
268     RP_TRIG_STATE_TRIGGERED, //!< Trigger is triggered/disabled
269     RP_TRIG_STATE_WAITING,   //!< Trigger is set up and waiting (to be triggered)
270 } rp_acq_trig_state_t;
271
272
273 /**
274  * Calibration parameters, stored in the EEPROM device
275  */
276 typedef struct {
277     uint32_t fe_ch1_fs_g_hi;    //!< High gain front end full scale voltage, channel A
278     uint32_t fe_ch2_fs_g_hi;    //!< High gain front end full scale voltage, channel B
279     uint32_t fe_ch1_fs_g_lo;    //!< Low gain front end full scale voltage, channel A
280     uint32_t fe_ch2_fs_g_lo;    //!< Low gain front end full scale voltage, channel B
281     int32_t  fe_ch1_lo_offs;    //!< Front end DC offset, channel A
282     int32_t  fe_ch2_lo_offs;    //!< Front end DC offset, channel B
283     uint32_t be_ch1_fs;         //!< Back end full scale voltage, channel A
284     uint32_t be_ch2_fs;         //!< Back end full scale voltage, channel B
285     int32_t  be_ch1_dc_offs;    //!< Back end DC offset, channel A
286     int32_t  be_ch2_dc_offs;    //!< Back end DC offset, on channel B
287         uint32_t magic;                     //!
288     int32_t  fe_ch1_hi_offs;    //!< Front end DC offset, channel A
289     int32_t  fe_ch2_hi_offs;    //!< Front end DC offset, channel B
290     uint32_t low_filter_aa_ch1;  //!< Filter equalization coefficients AA for Low mode, channel
291     uint32_t low_filter_bb_ch1;  //!< Filter equalization coefficients BB for Low mode, channel
292     uint32_t low_filter_pp_ch1;  //!< Filter equalization coefficients PP for Low mode, channel
293     uint32_t low_filter_kk_ch1;  //!< Filter equalization coefficients KK for Low mode, channel
```

```
294     uint32_t low_filter_aa_ch2;  //!< Filter equalization coefficients AA for Low mode, channel
295     uint32_t low_filter_bb_ch2;  //!< Filter equalization coefficients BB for Low mode, channel
296     uint32_t low_filter_pp_ch2;  //!< Filter equalization coefficients PP for Low mode, channel
297     uint32_t low_filter_kk_ch2;  //!< Filter equalization coefficients KK for Low mode, channel
298     uint32_t  hi_filter_aa_ch1;  //!< Filter equalization coefficients AA for High mode, channe
299     uint32_t  hi_filter_bb_ch1;  //!< Filter equalization coefficients BB for High mode, channe
300     uint32_t  hi_filter_pp_ch1;  //!< Filter equalization coefficients PP for High mode, channe
301     uint32_t  hi_filter_kk_ch1;  //!< Filter equalization coefficients KK for High mode, channe
302     uint32_t  hi_filter_aa_ch2;  //!< Filter equalization coefficients AA for High mode, channe
303     uint32_t  hi_filter_bb_ch2;  //!< Filter equalization coefficients BB for High mode, channe
304     uint32_t  hi_filter_pp_ch2;  //!< Filter equalization coefficients PP for High mode, channe
305     uint32_t  hi_filter_kk_ch2;  //!< Filter equalization coefficients KK for High mode, channe
306
307 } rp_calib_params_t;
308
309 /**
310  * UART Character bits size
311  */
312 typedef enum {
313     RP_UART_CS6,      //!< Set 6 bits
314     RP_UART_CS7,      //!< Set 7 bits
315     RP_UART_CS8       //!< Set 8 bits
316 } rp_uart_bits_size_t;
317
318
319 /**
320  * UART stop bits
321  */
322 typedef enum {
323     RP_UART_STOP1,      //!< Set 1 bit
324     RP_UART_STOP2       //!< Set 2 bits
325 } rp_uart_stop_bits_t;
326
327 /**
328  * UART parity mode
329  */
330 typedef enum {
331     RP_UART_NONE,      //!< Disable parity check
332     RP_UART_EVEN,      //!< Set even mode for parity
333     RP_UART_ODD,       //!< Set odd mode for parity
334     RP_UART_MARK,      //!< Set Always 1
335     RP_UART_SPACE      //!< Set Always 0
336 } rp_uart_parity_t;
337
338 /** @name General
339  */
340 ///@{
341
```

```
342
343 /**
344  * Initializes the library. It must be called first, before any other library method.
345  * @return If the function is successful, the return value is RP_OK.
346  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
347  */
348
349 int rp_Init(void);
350
351 /**
352  * Initializes the library. It must be called first, before any other library method.
353  * @param reset Reset to default configuration on api
354  * @return If the function is successful, the return value is RP_OK.
355  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
356  */
357
358 int rp_InitReset(bool reset);
359
360 int rp_CalibInit();
361
362 /**
363  * Releases the library resources. It must be called last, after library is not used anymore. T
364  * application exits.
365  * @return If the function is successful, the return value is RP_OK.
366  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
367  */
368 int rp_Release();
369
370 /**
371 * Resets all modules. Typically calles after rp_Init()
372 * application exits.
373 * @return If the function is successful, the return value is RP_OK.
374 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
375 */
376 int rp_Reset();
377
378 /**
379  * Retrieves the library version number
380  * @return Library version
381  */
382 const char* rp_GetVersion();
383
384 /**
385  * Returns textual representation of error code.
386  * @param errorCode Error code returned from API.
387  * @return Textual representation of error given error code.
388  */
389 const char* rp_GetError(int errorCode);
```

```
390
391
392 ///@}
393 /** @name Digital loop
394 */
395 ///@{
396
397 /**
398 * Enable or disables digital loop. This internally connect output to input
399 * @param enable True if you want to enable this feature or false if you want to disable it
400 * Each rp_GetCalibrationSettings call returns the same cached setting values.
401 * @return Calibration settings
402 */
403 int rp_EnableDigitalLoop(bool enable);
404
405
406 ///@}
407 /** @name Calibrate
408 */
409 ///@{
410
411 /**
412 * Returns calibration settings.
413 * These calibration settings are populated only once from EEPROM at rp_Init().
414 * Each rp_GetCalibrationSettings call returns the same cached setting values.
415 * @return Calibration settings
416 */
417 rp_calib_params_t rp_GetCalibrationSettings();
418
419 /**
420 * Returns default calibration settings.
421 * These calibration settings are populated only once from EEPROM at rp_Init().
422 * Each rp_GetCalibrationSettings call returns the same cached setting values.
423 * @return Calibration settings
424 */
425 rp_calib_params_t rp_GetDefaultCalibrationSettings();
426
427 /**
428 * Calibrates input channel offset. This input channel must be grounded to calibrate properly.
429 * Calibration data is written to EPROM and repopulated so that rp_GetCalibrationSettings works
430 * @param channel Channel witch is going to be calibrated
431 * @return If the function is successful, the return value is RP_OK.
432 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
433 */
434 int rp_CalibrateFrontEndOffset(rp_channel_t channel, rp_pinState_t gain, rp_calib_params_t* out
435
436 /**
437 * Calibrates input channel low voltage scale. Jumpers must be set to LV.
```

```
438 * This input channel must be connected to stable positive source.
439 * Calibration data is written to EPROM and repopulated so that rp_GetCalibrationSettings works
440 * @param channel Channel witch is going to be calibrated
441 * @param referentialVoltage Voltage of the source.
442 * @return If the function is successful, the return value is RP_OK.
443 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
444 */
445 int rp_CalibrateFrontEndScaleLV(rp_channel_t channel, float referentialVoltage, rp_calib_params
446
447 /**
448 * Calibrates input channel high voltage scale. Jumpers must be set to HV.
449 * This input channel must be connected to stable positive source.
450 * Calibration data is written to EPROM and repopulated so that rp_GetCalibrationSettings works
451 * @param channel Channel witch is going to be calibrated
452 * @param referentialVoltage Voltage of the source.
453 * @return If the function is successful, the return value is RP_OK.
454 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
455 */
456 int rp_CalibrateFrontEndScaleHV(rp_channel_t channel, float referentialVoltage, rp_calib_params
457
458 /**
459 * Calibrates output channel offset.
460 * This input channel must be connected to calibrated input channel with came number (CH1 to CH1
461 * Calibration data is written to EPROM and repopulated so that rp_GetCalibrationSettings works
462 * @param channel Channel witch is going to be calibrated
463 * @return If the function is successful, the return value is RP_OK.
464 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
465 */
466 int rp_CalibrateBackEndOffset(rp_channel_t channel);
467
468 /**
469 * Calibrates output channel voltage scale.
470 * This input channel must be connected to calibrated input channel with came number (CH1 to CH1
471 * Calibration data is written to EPROM and repopulated so that rp_GetCalibrationSettings works
472 * @param channel Channel witch is going to be calibrated
473 * @return If the function is successful, the return value is RP_OK.
474 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
475 */
476 int rp_CalibrateBackEndScale(rp_channel_t channel);
477
478 /**
479 * Calibrates output channel.
480 * This input channel must be connected to calibrated input channel with came number (CH1 to CH1
481 * Calibration data is written to EPROM and repopulated so that rp_GetCalibrationSettings works
482 * @param channel Channel witch is going to be calibrated
483 * @return If the function is successful, the return value is RP_OK.
484 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
485 */
```

```
int rp_CalibrateBackEnd(rp_channel_t channel, rp_calib_params_t* out_params);

/**
* Set default calibration values.
* Calibration data is written to EPROM and repopulated so that rp_GetCalibrationSettings works
* @return If the function is successful, the return value is RP_OK.
* If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
*/
int rp_CalibrationReset();

/**
* Copy factory calibration values into user eeprom.
* @return If the function is successful, the return value is RP_OK.
* If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
*/
int rp_CalibrationFactoryReset();

/**
* Set saved calibration values in case of roll-back calibration.
* Calibration data is written to EPROM and repopulated so that rp_GetCalibrationSettings works
* @return If the function is successful, the return value is RP_OK.
* If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
*/
int rp_CalibrationSetCachedParams();

/**
* Write calibration values.
* Calibration data is written to EPROM and repopulated so that rp_GetCalibrationSettings works
* @return If the function is successful, the return value is RP_OK.
* If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
*/
int rp_CalibrationWriteParams(rp_calib_params_t calib_params);
///@}

/**
* Set calibration values in memory.
* Calibration values are written to temporary memory, but not permanently.
* @return If the function is successful, the return value is RP_OK.
* If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
*/
int rp_CalibrationSetParams(rp_calib_params_t calib_params);



/** @name Identification
 */
///@{

```

```
534 /**
535 * Gets FPGA Synthesized ID
536 */
537 int rp_IdGetID(uint32_t *id);
538
539 /**
540 * Gets FPGA Unique DNA
541 */
542 int rp_IdGetDNA(uint64_t *dna);
543
544 ///@}
545
546
547 /**
548  * LED methods
549  */
550
551 int rp_LEDSetState(uint32_t state);
552 int rp_LEDGetState(uint32_t *state);
553
554 /**
555  * GPIO methods
556  */
557
558 int rp_GPIOnSetDirection(uint32_t direction);
559 int rp_GPIOnGetDirection(uint32_t *direction);
560 int rp_GPIOnSetState(uint32_t state);
561 int rp_GPIOnGetState(uint32_t *state);
562 int rp_GPIOpSetDirection(uint32_t direction);
563 int rp_GPIOpGetDirection(uint32_t *direction);
564 int rp_GPIOpSetState(uint32_t state);
565 int rp_GPIOpGetState(uint32_t *state);
566
567
568 /** @name Digital Input/Output
569  */
570 ///@{
571
572 /**
573 * Sets digital pins to default values. Pins DIO1_P - DIO7_P, RP_DIO0_N - RP_DIO7_N are set al O
574 */
575 int rp_DpinReset();
576
577 /**
578  * Sets digital input output pin state.
579  * @param pin    Digital input output pin.
580  * @param state  High/Low state that will be set at the given pin.
581  * @return If the function is successful, the return value is RP_OK.
```

```
582  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
583  */
584 int rp_DpinSetState(rp_dpin_t pin, rp_pinState_t state);
585
586 /**
587  * Gets digital input output pin state.
588  * @param pin    Digital input output pin.
589  * @param state  High/Low state that is set at the given pin.
590  * @return If the function is successful, the return value is RP_OK.
591  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
592  */
593 int rp_DpinGetState(rp_dpin_t pin, rp_pinState_t* state);
594
595 /**
596  * Sets digital input output pin direction. LED pins are already automatically set to the outpu
597  * and they cannot be set to the input direction. DIOx_P and DIOx_N are must set either output
598  * before they can be used. When set to input direction, it is not allowed to write into these
599  * @param pin        Digital input output pin.
600  * @param direction  In/Out direction that will be set at the given pin.
601  * @return If the function is successful, the return value is RP_OK.
602  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
603  */
604 int rp_DpinSetDirection(rp_dpin_t pin, rp_pinDirection_t direction);
605
606 /**
607  * Gets digital input output pin direction.
608  * @param pin        Digital input output pin.
609  * @param direction  In/Out direction that is set at the given pin.
610  * @return If the function is successful, the return value is RP_OK.
611  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
612  */
613 int rp_DpinGetDirection(rp_dpin_t pin, rp_pinDirection_t* direction);
614
615 ///@}
616
617
618 /** @name Analog Inputs/Outputs
619  */
620 ///@{
621
622 /**
623 * Sets analog outputs to default values (0V).
624 */
625 int rp_ApinReset();
626
627 /**
628  * Gets value from analog pin in volts.
629  * @param pin    Analog pin.
```

```
630  * @param value  Value on analog pin in volts
631  * @return If the function is successful, the return value is RP_OK.
632  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
633  */
634 int rp_ApinGetValue(rp_apin_t pin, float* value);
635
636 /**
637  * Gets raw value from analog pin.
638  * @param pin    Analog pin.
639  * @param value  Raw value on analog pin
640  * @return If the function is successful, the return value is RP_OK.
641  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
642  */
643 int rp_ApinGetValueRaw(rp_apin_t pin, uint32_t* value);
644
645 /**
646  * Sets value in volts on analog output pin.
647  * @param pin    Analog output pin.
648  * @param value  Value in volts to be set on given output pin.
649  * @return If the function is successful, the return value is RP_OK.
650  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
651  */
652 int rp_ApinSetValue(rp_apin_t pin, float value);
653
654 /**
655  * Sets raw value on analog output pin.
656  * @param pin    Analog output pin.
657  * @param value  Raw value to be set on given output pin.
658  * @return If the function is successful, the return value is RP_OK.
659  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
660  */
661 int rp_ApinSetValueRaw(rp_apin_t pin, uint32_t value);
662
663 /**
664  * Gets range in volts on specific pin.
665  * @param pin      Analog input output pin.
666  * @param min_val  Minimum value in volts on given pin.
667  * @param max_val  Maximum value in volts on given pin.
668  * @return If the function is successful, the return value is RP_OK.
669  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
670  */
671 int rp_ApinGetRange(rp_apin_t pin, float* min_val,  float* max_val);
672
673
674 /** @name Analog Inputs
675  */
676 ///@{
677
```

```
678 /**
679  * Gets value from analog pin in volts.
680  * @param pin    pin index
681  * @param value  voltage
682  * @return       RP_OK - successful, RP_E* - failure
683  */
684 int rp_AIpinGetValue(int unsigned pin, float* value);
685
686 /**
687  * Gets raw value from analog pin.
688  * @param pin    pin index
689  * @param value  raw 12 bit XADC value
690  * @return       RP_OK - successful, RP_E* - failure
691  */
692 int rp_AIpinGetValueRaw(int unsigned pin, uint32_t* value);
693
694
695 /** @name Analog Outputs
696  */
697 ///@{
698
699 /**
700 * Sets analog outputs to default values (0V).
701 */
702 int rp_AOpinReset();
703
704 /**
705  * Gets value from analog pin in volts.
706  * @param pin    Analog output pin index.
707  * @param value  Value on analog pin in volts
708  * @return       RP_OK - successful, RP_E* - failure
709  */
710 int rp_AOpinGetValue(int unsigned pin, float* value);
711
712 /**
713  * Gets raw value from analog pin.
714  * @param pin    Analog output pin index.
715  * @param value  Raw value on analog pin
716  * @return       RP_OK - successful, RP_E* - failure
717  */
718 int rp_AOpinGetValueRaw(int unsigned pin, uint32_t* value);
719
720 /**
721  * Sets value in volts on analog output pin.
722  * @param pin    Analog output pin index.
723  * @param value  Value in volts to be set on given output pin.
724  * @return       RP_OK - successful, RP_E* - failure
725  */
```

```
726 int rp_AOpinSetValue(int unsigned pin, float value);
727
728 /**
729  * Sets raw value on analog output pin.
730  * @param pin    Analog output pin index.
731  * @param value  Raw value to be set on given output pin.
732  * @return       RP_OK - successful, RP_E* - failure
733  */
734 int rp_AOpinSetValueRaw(int unsigned pin, uint32_t value);
735
736 /**
737  * Gets range in volts on specific pin.
738  * @param pin      Analog input output pin index.
739  * @param min_val  Minimum value in volts on given pin.
740  * @param max_val  Maximum value in volts on given pin.
741  * @return       RP_OK - successful, RP_E* - failure
742  */
743 int rp_AOpinGetRange(int unsigned pin, float* min_val,  float* max_val);
744
745
746 ///@}
747 /** @name Acquire
748  */
749 ///@{
750
751 /**
752  * Enables continous acquirement even after trigger has happened.
753  * @param enable True for enabling and false disabling
754  * @return If the function is successful, the return value is RP_OK.
755  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
756  */
757 int rp_AcqSetArmKeep(bool enable);
758
759 /**
760  * Gets status of continous acquirement even after trigger has happened.
761  * @param state Returns status
762  * @return If the function is successful, the return value is RP_OK.
763  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
764  */
765 int rp_AcqGetArmKeep(bool* state);
766
767 /**
768  * Indicates whether the ADC buffer was full of data. The length of the buffer is determined by
769  * @param state Returns status
770  * @return If the function is successful, the return value is RP_OK.
771  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
772  */
773 int rp_AcqGetBufferFillState(bool* state);
```

```
774
775 /**
776  * Sets the decimation used at acquiring signal. There is only a set of pre-defined decimation
777  * values which can be specified. See the #rp_acq_decimation_t enum values.
778  * @param decimation Specify one of pre-defined decimation values
779  * @return If the function is successful, the return value is RP_OK.
780  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
781  */
782 int rp_AcqSetDecimation(rp_acq_decimation_t decimation);
783
784 /**
785  * Gets the decimation used at acquiring signal. There is only a set of pre-defined decimation
786  * values which can be specified. See the #rp_acq_decimation_t enum values.
787  * @param decimation Returns one of pre-defined decimation values which is currently set.
788  * @return If the function is successful, the return value is RP_OK.
789  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
790  */
791 int rp_AcqGetDecimation(rp_acq_decimation_t* decimation);
792
793 /**
794  * Sets the decimation used at acquiring signal.
795  * You can specify values in the range (1,2,4,8,16-65536)
796  * @param decimation Decimation values
797  * @return If the function is successful, the return value is RP_OK.
798  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
799  */
800 int rp_AcqSetDecimationFactor(uint32_t decimation);
801
802 /**
803  * Gets the decimation factor used at acquiring signal in a numerical form. Although this metho
804  * value representing the current factor of the decimation, there is only a set of pre-defined
805  * factor values which can be returned. See the #rp_acq_decimation_t enum values.
806  * @param decimation Returns decimation factor value which is currently set.
807  * @return If the function is successful, the return value is RP_OK.
808  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
809  */
810 int rp_AcqGetDecimationFactor(uint32_t* decimation);
811
812 /**
813  * Sets the sampling rate for acquiring signal. There is only a set of pre-defined sampling rat
814  * values which can be specified. See the #rp_acq_sampling_rate_t enum values.
815  * @param sampling_rate Specify one of pre-defined sampling rate value
816  * @return If the function is successful, the return value is RP_OK.
817  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
818  */
819 int rp_AcqSetSamplingRate(rp_acq_sampling_rate_t sampling_rate);
820
821 /**
```

```
822  * Gets the sampling rate for acquiring signal. There is only a set of pre-defined sampling rat
823  * values which can be returned. See the #rp_acq_sampling_rate_t enum values.
824  * @param sampling_rate Returns one of pre-defined sampling rate value which is currently set
825  * @return If the function is successful, the return value is RP_OK.
826  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
827  */
828 int rp_AcqGetSamplingRate(rp_acq_sampling_rate_t* sampling_rate);
829
830 /**
831  * Gets the sampling rate for acquiring signal in a numerical form in Hz. Although this method
832  * value representing the current value of the sampling rate, there is only a set of pre-define
833  * values which can be returned. See the #rp_acq_sampling_rate_t enum values.
834  * @param sampling_rate returns currently set sampling rate in Hz
835  * @return If the function is successful, the return value is RP_OK.
836  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
837  */
838 int rp_AcqGetSamplingRateHz(float* sampling_rate);
839
840 /**
841  * Enables or disables averaging of data between samples.
842  * Data between samples can be averaged by setting the averaging flag in the Data decimation re
843  * @param enabled When true, the averaging is enabled, otherwise it is disabled.
844  * @return If the function is successful, the return value is RP_OK.
845  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
846  */
847 int rp_AcqSetAveraging(bool enabled);
848
849 /**
850  * Returns information if averaging of data between samples is enabled or disabled.
851  * Data between samples can be averaged by setting the averaging flag in the Data decimation re
852  * @param enabled Set to true when the averaging is enabled, otherwise is it set to false.
853  * @return If the function is successful, the return value is RP_OK.
854  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
855  */
856 int rp_AcqGetAveraging(bool *enabled);
857
858 /**
859  * Sets the trigger source used at acquiring signal. When acquiring is started,
860  * the FPGA waits for the trigger condition on the specified source and when the condition is m
861  * starts writing the signal to the buffer.
862  * @param source Trigger source.
863  * @return If the function is successful, the return value is RP_OK.
864  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
865  */
866 int rp_AcqSetTriggerSrc(rp_acq_trig_src_t source);
867
868 /**
869  * Gets the trigger source used at acquiring signal. When acquiring is started,
```

```
 * the FPGA waits for the trigger condition on the specified source and when the condition is m
 * starts writing the signal to the buffer.
 * @param source Currently set trigger source.
 * @return If the function is successful, the return value is RP_OK.
 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
 */
int rp_AcqGetTriggerSrc(rp_acq_trig_src_t* source);

/**
 * Returns the trigger state. Either it is waiting for a trigger to happen, or it has already b
 * By default it is in the triggered state, which is treated the same as disabled.
 * @param state Trigger state
 * @return If the function is successful, the return value is RP_OK.
 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
 */
int rp_AcqGetTriggerState(rp_acq_trig_state_t* state);

/**
 * Sets the number of decimated data after trigger written into memory.
 * @param decimated_data_num Number of decimated data. It must not be higher than the ADC buffe
 * @return If the function is successful, the return value is RP_OK.
 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
 */
int rp_AcqSetTriggerDelay(int32_t decimated_data_num);

/**
 * Returns current number of decimated data after trigger written into memory.
 * @param decimated_data_num Number of decimated data.
 * @return If the function is successful, the return value is RP_OK.
 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
 */
int rp_AcqGetTriggerDelay(int32_t* decimated_data_num);

/**
 * Sets the amount of decimated data in nanoseconds after trigger written into memory.
 * @param time_ns Time in nanoseconds. Number of ADC samples within the specified
 * time must not be higher than the ADC buffer size.
 * @return If the function is successful, the return value is RP_OK.
 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
 */
int rp_AcqSetTriggerDelayNs(int64_t time_ns);

/**
 * Returns the current amount of decimated data in nanoseconds after trigger written into memor
 * @param time_ns Time in nanoseconds.
 * @return If the function is successful, the return value is RP_OK.
 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
 */
```

```
int rp_AcqGetTriggerDelayNs(int64_t* time_ns);

/**
 * Returns the number of valid data ponts before trigger.
 * @param time_ns number of data points.
 * @return If the function is successful, the return value is RP_OK.
 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
 */
int rp_AcqGetPreTriggerCounter(uint32_t* value);

/**
 * Sets the trigger threshold value in volts. Makes the trigger when ADC value crosses this val
 * @param voltage Threshold value for the channel
 * @return If the function is successful, the return value is RP_OK.
 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
 */
int rp_AcqSetTriggerLevel(rp_channel_trigger_t channel, float voltage);

/**
 * Gets currently set trigger threshold value in volts
 * @param voltage Current threshold value for the channel
 * @return If the function is successful, the return value is RP_OK.
 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
 */
int rp_AcqGetTriggerLevel(rp_channel_trigger_t channel, float* voltage);

/**
 * Sets the trigger threshold hysteresis value in volts.
 * Value must be outside to enable the trigger again.
 * @param voltage Threshold hysteresis value for the channel
 * @return If the function is successful, the return value is RP_OK.
 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
 */
int rp_AcqSetTriggerHyst(float voltage);

/**
 * Gets currently set trigger threshold hysteresis value in volts
 * @param voltage Current threshold hysteresis value for the channel
 * @return If the function is successful, the return value is RP_OK.
 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
 */
int rp_AcqGetTriggerHyst(float* voltage);

/**
 * Sets the acquire gain state. The gain should be set to the same value as it is set on the Re
 * hardware by the LV/HV gain jumpers. LV = 1V; HV = 20V.
 * @param channel Channel A or B
 * @param state High or Low state
```

```
 * @return If the function is successful, the return value is RP_OK.
 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
 */
int rp_AcqSetGain(rp_channel_t channel, rp_pinState_t state);

/**
 * Returns the currently set acquire gain state in the library. It may not be set to the same v
 * it is set on the Red Pitaya hardware by the LV/HV gain jumpers. LV = 1V; HV = 20V.
 * @param channel Channel A or B
 * @param state Currently set High or Low state in the library.
 * @return If the function is successful, the return value is RP_OK.
 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
 */
int rp_AcqGetGain(rp_channel_t channel, rp_pinState_t* state);

/**
 * Returns the currently set acquire gain in the library. It may not be set to the same value a
 * it is set on the Red Pitaya hardware by the LV/HV gain jumpers. Returns value in Volts.
 * @param channel Channel A or B
 * @param voltage Currently set gain in the library. 1.0 or 20.0 Volts
 * @return If the function is successful, the return value is RP_OK.
 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
 */
int rp_AcqGetGainV(rp_channel_t channel, float* voltage);

/**
 * Returns current position of ADC write pointer.
 * @param pos Write pointer position
 * @return If the function is successful, the return value is RP_OK.
 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
 */
int rp_AcqGetWritePointer(uint32_t* pos);

/**
 * Returns position of ADC write pointer at time when trigger arrived.
 * @param pos Write pointer position
 * @return If the function is successful, the return value is RP_OK.
 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
 */
int rp_AcqGetWritePointerAtTrig(uint32_t* pos);

/**
 * Starts the acquire. Signals coming from the input channels are acquired and written into mem
 * @return If the function is successful, the return value is RP_OK.
 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
 */
int rp_AcqStart();

```

```
1014  /**
1015  * Stops the acquire.
1016  * @return If the function is successful, the return value is RP_OK.
1017  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1018  */
1019  int rp_AcqStop();
1020
1021  /**
1022   * Resets the acquire writing state machine.
1023   * @return If the function is successful, the return value is RP_OK.
1024   * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
1025   */
1026  int rp_AcqReset();
1027
1028  /**
1029   * Normalizes the ADC buffer position. Returns the modulo operation of ADC buffer size...
1030   * @param pos position to be normalized
1031   * @return Normalized position (pos % ADC_BUFFER_SIZE)
1032   */
1033  uint32_t rp_AcqGetNormalizedDataPos(uint32_t pos);
1034
1035  /**
1036   * Returns the ADC buffer in raw units from start to end position.
1037   *
1038   * @param channel Channel A or B for which we want to retrieve the ADC buffer.
1039   * @param start_pos Starting position of the ADC buffer to retrieve.
1040   * @param end_pos Ending position of the ADC buffer to retrieve.
1041   * @param buffer The output buffer gets filled with the selected part of the ADC buffer.
1042   * @param buffer_size Length of input buffer. Returns length of filled buffer. In case of too s
1043   * @return If the function is successful, the return value is RP_OK.
1044   * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
1045   */
1046  int rp_AcqGetDataPosRaw(rp_channel_t channel, uint32_t start_pos, uint32_t end_pos, int16_t* bu
1047
1048  /**
1049   * Returns the ADC buffer in Volt units from start to end position.
1050   *
1051   * @param channel Channel A or B for which we want to retrieve the ADC buffer.
1052   * @param start_pos Starting position of the ADC buffer to retrieve.
1053   * @param end_pos Ending position of the ADC buffer to retrieve.
1054   * @param buffer The output buffer gets filled with the selected part of the ADC buffer.
1055   * @param buffer_size Length of input buffer. Returns length of filled buffer. In case of too s
1056   * @return If the function is successful, the return value is RP_OK.
1057   * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
1058   */
1059  int rp_AcqGetDataPosV(rp_channel_t channel, uint32_t start_pos, uint32_t end_pos, float* buffer
1060
1061  /**
```

```
1062  * Returns the ADC buffer in raw units from specified position and desired size.
1063  * Output buffer must be at least 'size' long.
1064  * @param channel Channel A or B for which we want to retrieve the ADC buffer.
1065  * @param pos Starting position of the ADC buffer to retrieve.
1066  * @param size Length of the ADC buffer to retrieve. Returns length of filled buffer. In case o
1067  * @param buffer The output buffer gets filled with the selected part of the ADC buffer.
1068  * @return If the function is successful, the return value is RP_OK.
1069  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
1070  */
1071 int rp_AcqGetDataRaw(rp_channel_t channel,  uint32_t pos, uint32_t* size, int16_t* buffer);
1072
1073 /**
1074  * Returns the ADC buffer in raw units from specified position and desired size.
1075  * Output buffer must be at least 'size' long.
1076  * @param channel Channel A or B for which we want to retrieve the ADC buffer.
1077  * @param pos Starting position of the ADC buffer to retrieve.
1078  * @param size Length of the ADC buffer to retrieve. Returns length of filled buffer. In case o
1079  * @param buffer The output buffer gets filled with the selected part of the ADC buffer.
1080  * @return If the function is successful, the return value is RP_OK.
1081  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
1082  */
1083 int rp_AcqGetDataRawV2(uint32_t pos, uint32_t* size, uint16_t* buffer, uint16_t* buffer2);
1084
1085 /**
1086  * Returns the ADC buffer in raw units from the oldest sample to the newest one.
1087  * Output buffer must be at least 'size' long.
1088  * CAUTION: Use this method only when write pointer has stopped (Trigger happened and writing s
1089  * @param channel Channel A or B for which we want to retrieve the ADC buffer.
1090  * @param size Length of the ADC buffer to retrieve. Returns length of filled buffer. In case o
1091  * @param buffer The output buffer gets filled with the selected part of the ADC buffer.
1092  * @return If the function is successful, the return value is RP_OK.
1093  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
1094  */
1095 int rp_AcqGetOldestDataRaw(rp_channel_t channel, uint32_t* size, int16_t* buffer);
1096
1097 /**
1098  * Returns the latest ADC buffer samples in raw units.
1099  * Output buffer must be at least 'size' long.
1100  * @param channel Channel A or B for which we want to retrieve the ADC buffer.
1101  * @param size Length of the ADC buffer to retrieve. Returns length of filled buffer. In case o
1102  * @param buffer The output buffer gets filled with the selected part of the ADC buffer.
1103  * @return If the function is successful, the return value is RP_OK.
1104  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
1105  */
1106 int rp_AcqGetLatestDataRaw(rp_channel_t channel, uint32_t* size, int16_t* buffer);
1107
1108 /**
1109  * Returns the ADC buffer in Volt units from specified position and desired size.
```

```
1110  * Output buffer must be at least 'size' long.
1111  * @param channel Channel A or B for which we want to retrieve the ADC buffer.
1112  * @param pos Starting position of the ADC buffer to retrieve
1113  * @param size Length of the ADC buffer to retrieve. Returns length of filled buffer. In case o
1114  * @param buffer The output buffer gets filled with the selected part of the ADC buffer.
1115  * @return If the function is successful, the return value is RP_OK.
1116  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
1117  */
1118 int rp_AcqGetDataV(rp_channel_t channel, uint32_t pos, uint32_t* size, float* buffer);
1119
1120 /**
1121  * Returns the ADC buffer in Volt units from specified position and desired size.
1122  * Output buffer must be at least 'size' long.
1123  * @param pos Starting position of the ADC buffer to retrieve
1124  * @param size Length of the ADC buffer to retrieve. Returns length of filled buffer. In case o
1125  * @param buffer1 The output buffer gets filled with the selected part of the ADC buffer for ch
1126  * @param buffer2 The output buffer gets filled with the selected part of the ADC buffer for ch
1127  * @return If the function is successful, the return value is RP_OK.
1128  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
1129  */
1130 int rp_AcqGetDataV2(uint32_t pos, uint32_t* size, float* buffer1, float* buffer2);
1131
1132 /**
1133  * Returns the ADC buffer in Volt units from the oldest sample to the newest one.
1134  * Output buffer must be at least 'size' long.
1135  * CAUTION: Use this method only when write pointer has stopped (Trigger happened and writing s
1136  * @param channel Channel A or B for which we want to retrieve the ADC buffer.
1137  * @param size Length of the ADC buffer to retrieve. Returns length of filled buffer. In case o
1138  * @param buffer The output buffer gets filled with the selected part of the ADC buffer.
1139  * @return If the function is successful, the return value is RP_OK.
1140  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
1141  */
1142 int rp_AcqGetOldestDataV(rp_channel_t channel, uint32_t* size, float* buffer);
1143
1144 /**
1145  * Returns the latest ADC buffer samples in Volt units.
1146  * Output buffer must be at least 'size' long.
1147  * @param channel Channel A or B for which we want to retrieve the ADC buffer.
1148  * @param size Length of the ADC buffer to retrieve. Returns length of filled buffer. In case o
1149  * @param buffer The output buffer gets filled with the selected part of the ADC buffer.
1150  * @return If the function is successful, the return value is RP_OK.
1151  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
1152  */
1153 int rp_AcqGetLatestDataV(rp_channel_t channel, uint32_t* size, float* buffer);
1154
1155
1156 int rp_AcqGetBufSize(uint32_t* size);
1157
```

```
1158  /**
1159  * Sets the current calibration values from temporary memory to the FPGA filter
1160  * @return If the function is successful, the return value is RP_OK.
1161  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1162  */
1163  int rp_AcqUpdateAcqFilter(rp_channel_t channel);
1164
1165  /**
1166  * Sets the current calibration values from temporary memory to the FPGA filter
1167  * @param channel Channel A or B for which we want to retrieve the ADC buffer.
1168  * @param coef_aa Return AA coefficient.
1169  * @param coef_bb Return BB coefficient.
1170  * @param coef_kk Return KK coefficient.
1171  * @param coef_pp Return PP coefficient.
1172  * @return If the function is successful, the return value is RP_OK.
1173  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1174  */
1175  int rp_AcqGetFilterCalibValue(rp_channel_t channel,uint32_t* coef_aa, uint32_t* coef_bb, uint32
1176
1177  ///@}
1178  /** @name Generate
1179  */
1180  ///@{
1181
1182
1183  /**
1184  * Sets generate to default values.
1185  */
1186  int rp_GenReset();
1187
1188  /**
1189  * Enables output
1190  * @param channel Channel A or B which we want to enable
1191  * @return If the function is successful, the return value is RP_OK.
1192  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1193  */
1194  int rp_GenOutEnable(rp_channel_t channel);
1195
1196  /**
1197  * Runs/Stop two channels synchronously
1198  * @return If the function is successful, the return value is RP_OK.
1199  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1200  */
1201  int rp_GenOutEnableSync(bool enable);
1202
1203  /**
1204  * Disables output
1205  * @param channel Channel A or B which we want to disable
```

```
1206  * @return If the function is successful, the return value is RP_OK.
1207  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1208  */
1209 int rp_GenOutDisable(rp_channel_t channel);
1210
1211 /**
1212  * Gets value true if channel is enabled otherwise return false.
1213  * @param channel Channel A or B.
1214  * @param value Pointer where value will be returned
1215  * @return If the function is successful, the return value is RP_OK.
1216  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1217  */
1218 int rp_GenOutIsEnabled(rp_channel_t channel, bool *value);
1219
1220 /**
1221  * Sets channel signal peak to peak amplitude.
1222  * @param channel Channel A or B for witch we want to set amplitude
1223  * @param amplitude Amplitude of the generated signal. From 0 to max value. Max amplitude is 1
1224  * @return If the function is successful, the return value is RP_OK.
1225  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1226  */
1227 int rp_GenAmp(rp_channel_t channel, float amplitude);
1228
1229 /**
1230  * Gets channel signal peak to peak amplitude.
1231  * @param channel Channel A or B for witch we want to get amplitude.
1232  * @param amplitude Pointer where value will be returned.
1233  * @return If the function is successful, the return value is RP_OK.
1234  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1235  */
1236 int rp_GenGetAmp(rp_channel_t channel, float *amplitude);
1237
1238 /**
1239  * Sets DC offset of the signal. signal = signal + DC_offset.
1240  * @param channel Channel A or B for witch we want to set DC offset.
1241  * @param offset DC offset of the generated signal. Max offset is 2.
1242  * @return If the function is successful, the return value is RP_OK.
1243  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1244  */
1245 int rp_GenOffset(rp_channel_t channel, float offset);
1246
1247 /**
1248  * Gets DC offset of the signal.
1249  * @param channel Channel A or B for witch we want to get amplitude.
1250  * @param offset Pointer where value will be returned.
1251  * @return If the function is successful, the return value is RP_OK.
1252  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1253  */
```

```
1254 int rp_GenGetOffset(rp_channel_t channel, float *offset);
1255
1256 /**
1257 * Sets channel signal frequency.
1258 * @param channel Channel A or B for witch we want to set frequency.
1259 * @param frequency Frequency of the generated signal in Hz.
1260 * @return If the function is successful, the return value is RP_OK.
1261 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1262 */
1263 int rp_GenFreq(rp_channel_t channel, float frequency);
1264
1265 /**
1266 * Gets channel signal frequency.
1267 * @param channel Channel A or B for witch we want to get frequency.
1268 * @param frequency Pointer where value will be returned.
1269 * @return If the function is successful, the return value is RP_OK.
1270 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1271 */
1272 int rp_GenGetFreq(rp_channel_t channel, float *frequency);
1273
1274 /**
1275 * Sets channel signal phase. This shifts the signal in time.
1276 * @param channel Channel A or B for witch we want to set phase.
1277 * @param phase Phase in degrees of the generated signal. From 0 deg to 180 deg.
1278 * @return If the function is successful, the return value is RP_OK.
1279 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1280 */
1281 int rp_GenPhase(rp_channel_t channel, float phase);
1282
1283 /**
1284 * Gets channel signal phase.
1285 * @param channel Channel A or B for witch we want to get phase.
1286 * @param phase Pointer where value will be returned.
1287 * @return If the function is successful, the return value is RP_OK.
1288 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1289 */
1290 int rp_GenGetPhase(rp_channel_t channel, float *phase);
1291
1292 /**
1293 * Sets channel signal waveform. This determines how the signal looks.
1294 * @param channel Channel A or B for witch we want to set waveform type.
1295 * @param form Wave form of the generated signal [SINE, SQUARE, TRIANGLE, SAWTOOTH, PWM, DC, ARB
1296 * @return If the function is successful, the return value is RP_OK.
1297 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1298 */
1299 int rp_GenWaveform(rp_channel_t channel, rp_waveform_t type);
1300
1301 /**
```

```
1302 * Gets channel signal waveform.
1303 * @param channel Channel A or B for witch we want to get waveform.
1304 * @param type Pointer where value will be returned.
1305 * @return If the function is successful, the return value is RP_OK.
1306 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1307 */
1308 int rp_GenGetWaveform(rp_channel_t channel, rp_waveform_t *type);
1309
1310 /**
1311 * Sets user defined waveform.
1312 * @param channel Channel A or B for witch we want to set waveform.
1313 * @param waveform Use defined wave form, where min is -1V an max is 1V.
1314 * @param length Length of waveform.
1315 * @return If the function is successful, the return value is RP_OK.
1316 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1317 */
1318 int rp_GenArbWaveform(rp_channel_t channel, float *waveform, uint32_t length);
1319
1320 /**
1321 * Gets user defined waveform.
1322 * @param channel Channel A or B for witch we want to get waveform.
1323 * @param waveform Pointer where waveform will be returned.
1324 * @param length Pointer where waveform length will be returned.
1325 * @return If the function is successful, the return value is RP_OK.
1326 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1327 */
1328 int rp_GenGetArbWaveform(rp_channel_t channel, float *waveform, uint32_t *length);
1329
1330 /**
1331 * Sets duty cycle of PWM signal.
1332 * @param channel Channel A or B for witch we want to set duty cycle.
1333 * @param ratio Ratio betwen the time when signal in HIGH vs the time when signal is LOW.
1334 * @return If the function is successful, the return value is RP_OK.
1335 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1336 */
1337 int rp_GenDutyCycle(rp_channel_t channel, float ratio);
1338
1339 /**
1340 * Gets duty cycle of PWM signal.
1341 * @param channel Channel A or B for witch we want to get duty cycle.
1342 * @param ratio Pointer where value will be returned.
1343 * @return If the function is successful, the return value is RP_OK.
1344 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1345 */
1346 int rp_GenGetDutyCycle(rp_channel_t channel, float *ratio);
1347
1348 /**
1349 * Sets generation mode.
```

```
1350 * @param channel Channel A or B for witch we want to set generation mode.
1351 * @param mode Type of signal generation (CONTINUOUS, BURST, STREAM).
1352 * @return If the function is successful, the return value is RP_OK.
1353 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1354 */
1355 int rp_GenMode(rp_channel_t channel, rp_gen_mode_t mode);
1356
1357 /**
1358 * Gets generation mode.
1359 * @param channel Channel A or B for witch we want to get generation mode.
1360 * @param mode Pointer where value will be returned.
1361 * @return If the function is successful, the return value is RP_OK.
1362 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1363 */
1364 int rp_GenGetMode(rp_channel_t channel, rp_gen_mode_t *mode);
1365
1366 /**
1367 * Sets number of generated waveforms in a burst.
1368 * @param channel Channel A or B for witch we want to set number of generated waveforms in a bur
1369 * @param num Number of generated waveforms. If -1 a continuous signal will be generated.
1370 * @return If the function is successful, the return value is RP_OK.
1371 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1372 */
1373 int rp_GenBurstCount(rp_channel_t channel, int num);
1374
1375 /**
1376 * Gets number of generated waveforms in a burst.
1377 * @param channel Channel A or B for witch we want to get number of generated waveforms in a bur
1378 * @param num Pointer where value will be returned.
1379 * @return If the function is successful, the return value is RP_OK.
1380 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1381 */
1382 int rp_GenGetBurstCount(rp_channel_t channel, int *num);
1383
1384 /**
1385 * Sets the value to be set at the end of the generated signal in burst mode.
1386 * @param channel Channel A or B for witch we want to set number of generated waveforms in a bur
1387 * @param amplitude Amplitude level at the end of the signal (Volt).
1388 * @return If the function is successful, the return value is RP_OK.
1389 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1390 */
1391 int rp_GenBurstLastValue(rp_channel_t channel, float amlitude);
1392
1393 /**
1394 * Gets the value to be set at the end of the generated signal in burst mode.
1395 * @param channel Channel A or B for witch we want to get number of generated waveforms in a bur
1396 * @param amplitude Amplitude where value will be returned (Volt).
1397 * @return If the function is successful, the return value is RP_OK.
```

```
1398 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1399 */
1400 int rp_GenGetBurstLastValue(rp_channel_t channel, float *amlitude);
1401
1402 /**
1403 * Sets number of burst repetitions. This determines how many bursts will be generated.
1404 * @param channel Channel A or B for witch we want to set number of burst repetitions.
1405 * @param repetitions Number of generated bursts. If -1, infinite bursts will be generated.
1406 * @return If the function is successful, the return value is RP_OK.
1407 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1408 */
1409 int rp_GenBurstRepetitions(rp_channel_t channel, int repetitions);
1410
1411 /**
1412 * Gets number of burst repetitions.
1413 * @param channel Channel A or B for witch we want to get number of burst repetitions.
1414 * @param repetitions Pointer where value will be returned.
1415 * @return If the function is successful, the return value is RP_OK.
1416 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1417 */
1418 int rp_GenGetBurstRepetitions(rp_channel_t channel, int *repetitions);
1419
1420 /**
1421 * Sets the time/period of one burst in micro seconds. Period must be equal or greater then the
1422 * If it is greater than the difference will be the delay between two consequential bursts.
1423 * @param channel Channel A or B for witch we want to set burst period.
1424 * @param period Time in micro seconds.
1425 * @return If the function is successful, the return value is RP_OK.
1426 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1427 */
1428 int rp_GenBurstPeriod(rp_channel_t channel, uint32_t period);
1429
1430 /**
1431 * Gets the period of one burst in micro seconds.
1432 * @param channel Channel A or B for witch we want to get burst period.
1433 * @param period Pointer where value will be returned.
1434 * @return If the function is successful, the return value is RP_OK.
1435 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1436 */
1437 int rp_GenGetBurstPeriod(rp_channel_t channel, uint32_t *period);
1438
1439 /**
1440 * Sets trigger source.
1441 * @param channel Channel A or B for witch we want to set trigger source.
1442 * @param src Trigger source (INTERNAL, EXTERNAL_PE, EXTERNAL_NE, GATED_BURST).
1443 * @return If the function is successful, the return value is RP_OK.
1444 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1445 */
```

```
1446 int rp_GenTriggerSource(rp_channel_t channel, rp_trig_src_t src);
1447
1448 /**
1449 * Gets trigger source.
1450 * @param channel Channel A or B for witch we want to get burst period.
1451 * @param src Pointer where value will be returned.
1452 * @return If the function is successful, the return value is RP_OK.
1453 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1454 */
1455 int rp_GenGetTriggerSource(rp_channel_t channel, rp_trig_src_t *src);
1456
1457 /**
1458 * Sets Trigger for specified channel/channels.
1459 * @param mask Mask determines channel: 1->ch1, 2->ch2, 3->ch1&ch2.
1460 * @return If the function is successful, the return value is RP_OK.
1461 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1462 */
1463 int rp_GenTrigger(uint32_t channel);
1464
1465 /**
1466 * The generator is reset on both channels.
1467 * @return If the function is successful, the return value is RP_OK.
1468 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1469 */
1470 int rp_GenSynchronise();
1471
1472 /**
1473 * Sets the DAC protection mode from overheating. Only works with Redpitaya 250-12 otherwise ret
1474 * @param channel Channel A or B for witch we want to set protection.
1475 * @param enable Flag enabling protection mode.total
1476 * @return If the function is successful, the return value is RP_OK.
1477 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1478 */
1479 int rp_SetEnableTempProtection(rp_channel_t channel, bool enable);
1480
1481 /**
1482 * Get status of DAC protection mode from overheating. Only works with Redpitaya 250-12 otherwis
1483 * @param channel Channel A or B for witch we want to set protection.
1484 * @param enable Flag return current status.
1485 * @return If the function is successful, the return value is RP_OK.
1486 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1487 */
1488 int rp_GetEnableTempProtection(rp_channel_t channel, bool *enable);
1489
1490 /**
1491 * Resets the flag indicating that the DAC is overheated. Only works with Redpitaya 250-12 other
1492 * @param channel Channel A or B.
1493 * @param status  New status for latch trigger.
```

```
1494  * @return If the function is successful, the return value is RP_OK.
1495  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1496  */
1497 int rp_SetLatchTempAlarm(rp_channel_t channel, bool status);
1498
1499 /**
1500  * Returns the status that there was an overheat. Only works with Redpitaya 250-12 otherwise ret
1501  * @param channel Channel A or B.
1502  * @param status  State of latch trigger.
1503  * @return If the function is successful, the return value is RP_OK.
1504  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1505  */
1506 int rp_GetLatchTempAlarm(rp_channel_t channel, bool *status);
1507
1508 /**
1509  * Returns the current DAC overheat status in real time. Only works with Redpitaya 250-12 otherw
1510  * @param channel Channel A or B.
1511  * @param status  Get current state.
1512  * @return If the function is successful, the return value is RP_OK.
1513  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1514  */
1515 int rp_GetRuntimeTempAlarm(rp_channel_t channel, bool *status);
1516
1517
1518 /**
1519  * Only works with Redpitaya 250-12 otherwise returns RP_NOTS
1520  * @param enable return current state.
1521  * @return If the function is successful, the return value is RP_OK.
1522  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1523  */
1524 int rp_GetPllControlEnable(bool *enable);
1525
1526 /**
1527  * Only works with Redpitaya 250-12 otherwise returns RP_NOTS
1528  * @param enable Flag enabling PLL control.
1529  * @return If the function is successful, the return value is RP_OK.
1530  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1531  */
1532 int rp_SetPllControlEnable(bool enable);
1533
1534 /**
1535  * Only works with Redpitaya 250-12 otherwise returns RP_NOTS
1536  * @param status Get current state.
1537  * @return If the function is successful, the return value is RP_OK.
1538  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1539  */
1540 int rp_GetPllControlLocked(bool *status);
1541
```

```
1542 float rp_CmnCnvCntToV(uint32_t field_len, uint32_t cnts, float adc_max_v, uint32_t calibScale,
1543
1544 /**
1545  * Opens the UART device (/dev/ttyPS1). Initializes the default settings.
1546  * @return If the function is successful, the return value is RP_OK.
1547  * If the function is unsuccessful, the return value is any of RP_E* values that indicate an er
1548  */
1549 int rp_UartInit();
1550
1551 /**
1552 * Closes device UART
1553 * @return If the function is successful, the return value is RP_OK.
1554 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1555 */
1556 int rp_UartRelease();
1557
1558 /**
1559 * Reading values into the buffer from the UART device
1560 * @param buffer Non-zero buffer for writing data.
1561 * @param size Buffer size. Returns the amount of data read.
1562 * @return If the function is successful, the return value is RP_OK.
1563 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1564 */
1565 int rp_UartRead(unsigned char *buffer, int *size);
1566
1567 /**
1568 * Writes data to UART
1569 * @param buffer The buffer to be written to the UART.
1570 * @param size Buffer size.
1571 * @return If the function is successful, the return value is RP_OK.
1572 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1573 */
1574 int rp_UartWrite(unsigned char *buffer, int size);
1575
1576 /**
1577 * Set speed for the UART.
1578 * @param speed Value of speed
1579 * @return If the function is successful, the return value is RP_OK.
1580 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1581 */
1582 int rp_UartSpeed(int speed);
1583
1584 /**
1585 * Set character size for the UART.
1586 * @param size Value of size
1587 * @return If the function is successful, the return value is RP_OK.
1588 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1589 */
```

```
1590 int rp_UartSetBits(rp_uart_bits_size_t size);
1591
1592 /**
1593 * Set stop bits size for the UART.
1594 * @param mode Value of size
1595 * @return If the function is successful, the return value is RP_OK.
1596 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1597 */
1598 int rp_UartSetStopBits(rp_uart_stop_bits_t mode);
1599
1600 /**
1601 * Set parity check mode for the UART.
1602 * @param mode Value of size
1603 * @return If the function is successful, the return value is RP_OK.
1604 * If the function is unsuccessful, the return value is any of RP_E* values that indicate an err
1605 */
1606 int rp_UartSetParityMode(rp_uart_parity_t mode);
1607
1608 #ifdef __cplusplus
1609 }
1610 #endif
1611
1612 #endif //__RP_H
```