

GCC Code Coverage Report

Directory: ./	Exec	Total	Coverage
Date: 2022-03-19 23:06:30	Lines: 0	115	0.0 %
Legend: low: < 75.0 % medium: >= 75.0 % high: >= 90.0 %	Branches: 0	34	0.0 %

File	Lines	Branches
include/adp5587_common.hpp	0.0 % 0 / 2	- % 0 / 0
src/adp5587_common.cpp	0.0 % 0 / 113	0.0 % 0 / 34

Generated by: [GCOVR \(Version 4.2\)](#)

GCC Code Coverage Report

Directory: ./	Exec	Total	Coverage
Date: 2022-03-19 23:06:30	Lines: 0	115	0.0 %
Legend: low: < 75.0 % medium: >= 75.0 % high: >= 90.0 %	Branches: 0	34	0.0 %

File	Lines	Branches
include/adp5587_common.hpp	0.0 % 0 / 2	- % 0 / 0
src/adp5587_common.cpp	0.0 % 0 / 113	0.0 % 0 / 34

Generated by: [GCOVR \(Version 4.2\)](#)

GCC Code Coverage Report

Directory: ./	Exec	Total	Coverage
File: include/adp5587_common.hpp	Lines: 0	2	0.0 %
Date: 2022-03-19 23:06:30	Branches: 0	0	- %

Line	Branch	Exec	Source
1			// MIT License
2			
3			// Copyright (c) 2022 Chris Sutton
4			
5			// Permission is hereby granted, free of charge, to any person obtaining a copy
6			// of this software and associated documentation files (the "Software"), to deal
7			// in the Software without restriction, including without limitation the rights
8			// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
9			// copies of the Software, and to permit persons to whom the Software is
10			// furnished to do so, subject to the following conditions:
11			
12			// The above copyright notice and this permission notice shall be included in all
13			// copies or substantial portions of the Software.
14			
15			// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
16			// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
17			// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
18			// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
19			// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
20			// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
21			// SOFTWARE.
22			
23			#ifndef __ADP5587_COMMON_HPP__
24			#define __ADP5587_COMMON_HPP__
25			
26			#include <restricted_base.hpp>
27			#include <i2c_utils.hpp>
28			#include <isr_manager_stm32g0.hpp>
29			
30			#ifdef X86_UNIT_TESTING_ONLY
31			#include <mock_cmsis.hpp>
32			#endif
33			
34			namespace adp5587
35			{
36			
37			class CommonFunctions
38			{
39			public:
40			
41			
42			// @brief Incomplete list of ADP5587 device registers
43			// see datasheet page 15 (https://www.analog.com/media/en/technical-documentation/data-sheets/adp5587.pdf)
44			enum Registers
45			{
46			DEV_ID = 0x00,
47			CFG = 0x01,
48			INT_STAT = 0x02,
49			KEY_LCK_EC_STAT = 0x03,
50			KEY_EVENTA = 0x04,
51			KEY_EVENTB = 0x05,
52			KEY_EVENTC = 0x06,
53			KEY_EVENTD = 0x07,
54			KEY_EVENTE = 0x08,
55			KEY_EVENTF = 0x09,
56			KEY_EVENTG = 0x0A,
57			KEY_EVENTH = 0x0B,
58			KEY_EVENTI = 0x0C,
59			KEY_EVENTJ = 0x0D,
60			GPIO_INT_STAT1 = 0x11,
61			GPIO_INT_STAT2 = 0x12,
62			GPIO_INT_STAT3 = 0x13,
63			
64			GPIO_INT_EN1 = 0x1A,
65			GPIO_INT_EN2 = 0x1B,
66			GPIO_INT_EN3 = 0x1C,
67			KP_GPIO1 = 0x1D,
68			KP_GPIO2 = 0x1E,
69			
70			KP_GPIO3 = 0x1F,
71			GPI_EM_REG1 = 0x20,
72			GPI_EM_REG2 = 0x21,
73			GPI_EM_REG3 = 0x22,
74			GPIO_DIR1 = 0x23,
75			GPIO_DIR2 = 0x24,
76			GPIO_DIR3 = 0x25,
77			GPIO_INT_LVL1 = 0x26,
78			GPIO_INT_LVL2 = 0x27,
79			GPIO_INT_LVL3 = 0x28,
80			DEBOUNCE_DIS1 = 0x29,
81			DEBOUNCE_DIS2 = 0x30,
			DEBOUNCE_DIS3 = 0x31,

```

82     GPIO_PULL1          = 0x32,
83     GPIO_PULL2          = 0x33,
84     GPIO_PULL3          = 0x34,
85
86 };
87
88
89 // @brief Values for Keypad or GPIO selection registers
90 enum KP_GPIO
91 {
92     R0 = 0b00000001,
93     R1 = 0b00000010,
94     R2 = 0b00000100,
95     R3 = 0b00001000,
96     R4 = 0b00010000,
97     R5 = 0b00100000,
98     R6 = 0b01000000,
99     R7 = 0b10000000,
100
101     C0 = 0b00000001,
102     C1 = 0b00000010,
103     C2 = 0b00000100,
104     C3 = 0b00001000,
105     C4 = 0b00010000,
106     C5 = 0b00100000,
107     C6 = 0b01000000,
108     C7 = 0b10000000,
109     C8 = 0b00000001,
110     C9 = 0b00000010,
111 };
112
113 // Keypad release encodings. These values appear in the KeyEventReg entries after key press/release events
114 // see datasheet page 9 (https://www.analog.com/media/en/technical-documentation/data-sheets/adp5587.pdf)
115 // To get Key press events IDs, bitwise-OR the KeyPadMappings::XX_OFF values with KeyPadMappings::ON .
116 // See the templated overload operator function below.
117 enum class KeyPadMappings
118 {
119     INIT=0,
120     // these default to key release events
121     A7_OFF=71, A6_OFF=61, A5_OFF=51, A4_OFF=41, A3_OFF=31, A2_OFF=21, A1_OFF=11, A0_OFF=1,
122     B7_OFF=72, B6_OFF=62, B5_OFF=52, B4_OFF=42, B3_OFF=32, B2_OFF=22, B1_OFF=12, B0_OFF=2,
123     C7_OFF=73, C6_OFF=63, C5_OFF=53, C4_OFF=43, C3_OFF=33, C2_OFF=23, C1_OFF=13, C0_OFF=3,
124     D7_OFF=74, D6_OFF=64, D5_OFF=54, D4_OFF=44, D3_OFF=34, D2_OFF=24, D1_OFF=14, D0_OFF=4,
125     E7_OFF=75, E6_OFF=65, E5_OFF=55, E4_OFF=45, E3_OFF=35, E2_OFF=25, E1_OFF=15, E0_OFF=5,
126     F7_OFF=76, F6_OFF=66, F5_OFF=56, F4_OFF=46, F3_OFF=36, F2_OFF=26, F1_OFF=16, F0_OFF=6,
127     G7_OFF=77, G6_OFF=67, G5_OFF=57, G4_OFF=47, G3_OFF=37, G2_OFF=27, G1_OFF=17, G0_OFF=7,
128     H7_OFF=78, H6_OFF=68, H5_OFF=58, H4_OFF=48, H3_OFF=38, H2_OFF=28, H1_OFF=18, H0_OFF=8,
129     I7_OFF=79, I6_OFF=69, I5_OFF=59, I4_OFF=49, I3_OFF=39, I2_OFF=29, I1_OFF=19, I0_OFF=9,
130     J7_OFF=80, J6_OFF=70, J5_OFF=60, J4_OFF=50, J3_OFF=40, J2_OFF=30, J1_OFF=20, J0_OFF=10,
131     // this bit will be set if the key was pressed
132     ON=128,
133 };
134
135 enum class GPIKeyMappings
136 {
137     // these default to key release events
138     R0=97, R1=98, R2=99, R3=100, R4=101, R5=102, R6=103, R7=104,
139     C0=105, C1=106, C2=107, C3=108, C4=109, C5=110, C6=111, C7=112, C8=113, C9=114,
140     // this bit will be set if the key was pressed
141     ON=128,
142 };
143
144
145 // @brief Bitwise-OR two scoped enum literals together
146
147 // @tparam SCOPED_ENUM The scoped enum type.
148 // @param L The left literal operand
149 // @param R The right literal operand
150 // @return constexpr SCOPED_ENUM Returns the combined value as SCOPED_ENUM enum type
151 template<typename SCOPED_ENUM>
152 constexpr friend SCOPED_ENUM operator | (SCOPED_ENUM L, SCOPED_ENUM R)
153 {
154     return static_cast<SCOPED_ENUM>(static_cast<int>(L) | static_cast<int>(R));
155 }
156
157
158 // @brief Write the byte array to the ADP5587 register
159 // @tparam REG_SIZE
160 // @param reg The register to modify
161 // @param tx_bytes The value to write
162 void write_register(const uint8_t reg, uint8_t tx_byte);
163
164 // @brief callback function for IsrManagerStm32g0
165 // see stm32_interrupt_managers/inc/stm32g0_interrupt_manager_functional.hpp
166 void exti_isr();
167
168 // @brief global enable keypad interrupts
169 void enable_keypad_isr();
170
171 // @brief global disable keypad interrupts
172 void disable_keypad_isr();

```

```

172 // @brief global enable GPIO interrupts
173 void enable_gpio_isr();
174
175 // @brief global disable GPIO interrupts
176 void disable_gpio_isr();
177
178 // @brief Select individual row/col connections as keypad input. Omitted connections will be configured as GPI.
179 void keypad_gpio_select(uint8_t row_mask, uint8_t col_mask0_7, uint8_t col_mask8_9);
180
181 // @brief Select if GPI is included in event FIFO
182 void gpio_fifo_select(uint8_t row_mask, uint8_t col_mask0_7, uint8_t col_mask8_9);
183
184 // @brief Enable GPI interrupts on individual row/col
185 void gpio_interrupt_select(uint8_t row_mask, uint8_t col_mask0_7, uint8_t col_mask8_9);
186
187 // @brief Set the GPIO direction as output on individual rows/cols
188 void set_gpo_out(uint8_t row_mask, uint8_t col_mask0_7, uint8_t col_mask8_9);
189
190 // @brief Set the GPIO lvl as active high on individual rows/cols
191 void set_gpi_active_high(uint8_t row_mask, uint8_t col_mask0_7, uint8_t col_mask8_9);
192
193 // @brief Disable the GPIO debounce on individual rows/cols
194 void disable_debounce(uint8_t row_mask, uint8_t col_mask0_7, uint8_t col_mask8_9);
195
196 // @brief Disable the GPIO pullup on individual rows/cols
197 void disable_gpio_pullup(uint8_t row_mask, uint8_t col_mask0_7, uint8_t col_mask8_9);
198
199 protected:
200
201 // @brief The CMSIS mem-mapped I2C periph. Set in the c'tor
202 // std::unique_ptr<I2C_TypeDef> m_i2c_handle;
203 I2C_TypeDef* m_i2c_handle;
204
205 // @brief local store for ADP5587 key event FIFO
206 std::array<KeyPadMappings, 10> m_key_event_fifo {KeyPadMappings::INIT};
207
208 // @brief Confirm ADP5587 replies to write_addr and read_addr with ACK
209 // @return true if both are successful, false if either fail.
210 bool probe_i2c();
211
212 // @brief clear the Key Event Registers (KEY_EVENTx) by reading them and
213 // clear the Interrupt status register (INT_STAT) by writing 1 to each bit
214 void clear_fifo_and_isr();
215
216 private:
217
218 // @brief The i2c slave address for ADP5587ACPZ-1-R7
219 const uint8_t m_i2c_addr {0x60};
220
221 // @brief Configuration Register 1
222
223 enum ConfigReg
224 {
225     KE_IEN          = (1 << 0), // Key events interrupt enable.
226     GPI_IEN         = (1 << 1), // GPI interrupt enable.
227     K_LCK_IM        = (1 << 2), // Keypad lock interrupt mask.
228     OVR_FLOW_IEN    = (1 << 3), // Overflow interrupt enable.
229     INT_CFG         = (1 << 4), // Interrupt configuration.
230     OVR_FLOW_M      = (1 << 5), // Overflow mode.
231     GPIEM_CFG       = (1 << 6), // GPI event mode configuration.
232     AUTO_INC        = (1 << 7), // 2C auto-increment. Burst read is supported; burst write is not supported.
233 };
234
235 // Interrupt status register
236 enum IntStatusReg
237 {
238     KE_INT          = (1 << 0), // Key events interrupt status. When set, write 1 to clear.
239     GPI_INT         = (1 << 1), // GPI interrupt status. When set, write 1 to clear.
240     K_LCK_INT       = (1 << 2), // Keylock interrupt status. When set, write 1 to clear.
241     OVR_FLOW_INT    = (1 << 3), // Overflow interrupt status. When set, write 1 to clear.
242 };
243
244 // Keylock and event counter register
245 enum KeyLckEvCntReg
246 {
247     KEC1            = (1 << 0), // 3-bit key event count of key event register.
248     KEC2            = (1 << 1),
249     KEC3            = (1 << 2),
250     KEC4            = (1 << 3),
251     LCK1            = (1 << 4), // 2-bit keypad lock status[1:0] (00 = unlocked; 11 = locked; read-only bits).
252     LCK2            = (1 << 5),
253     K_LCK_EN        = (1 << 6), // 0: lock feature is disabled. 1: lock feature is enabled.
254 };
255
256 // @brief Read some bytes from the ADP5587 register
257 // @param reg The register to read
258 void read_register(const uint8_t reg, uint8_t &rx_byte);
259
260 // @brief Updates the stored key events FIFO data and resets the HW ISR
261 void update_key_events();

```

```
262
263 // @brief Read the FIFO bytes into "m_key_event_fifo" member byte array
264 void read_fifo_bytes_from_hw();
265
266 void write_config_bits(uint8_t config_bits);
267 void clear_config_bits(uint8_t config_bits);
268
269
270
271
272
273
274 };
275
276 } // namespace adp5587
277
278
279 #endif // __ADP5587_COMMON_HPP__
```

GCC Code Coverage Report

Directory: ./		Exec	Total	Coverage
File: src/adp5587_common.cpp	Lines:	0	113	0.0 %
Date: 2022-03-19 23:06:30	Branches:	0	34	0.0 %

Line	Branch	Exec	Source
1			// MIT License
2			
3			// Copyright (c) 2022 Chris Sutton
4			
5			// Permission is hereby granted, free of charge, to any person obtaining a copy
6			// of this software and associated documentation files (the "Software"), to deal
7			// in the Software without restriction, including without limitation the rights
8			// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
9			// copies of the Software, and to permit persons to whom the Software is
10			// furnished to do so, subject to the following conditions:
11			
12			// The above copyright notice and this permission notice shall be included in all
13			// copies or substantial portions of the Software.
14			
15			// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
16			// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
17			// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
18			// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
19			// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
20			// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
21			// SOFTWARE.
22			
23			#include <adp5587_common.hpp>
24			#if defined(USE_RTT)
25			#include <SEGGER_RTT.h>
26			#endif
27			
28			namespace adp5587
29			{
30			
31			void CommonFunctions::write_register(uint8_t reg [[maybe_unused]], uint8_t tx_byte [[maybe_unused]])
32			{
33			#if not defined(X86_UNIT_TESTING_ONLY)
34			// write this number of bytes: The data byte(s) AND the address byte
35			stm32::i2c::set_numbytes(m_i2c_handle, 2);
36			
37			// send AD5587 write address and the register we want to write
38			stm32::i2c::send_addr(m_i2c_handle, m_i2c_addr, stm32::i2c::MsgType::WRITE);
39			stm32::i2c::send_byte(m_i2c_handle, reg);
40			
41			// send AD5587 read address and get received data
42			stm32::i2c::send_byte(m_i2c_handle, tx_byte);
43			
44			stm32::i2c::generate_stop_condition(m_i2c_handle);
45			#endif
46			}
47			
48			void CommonFunctions::exti_isr()
49			{
50			#if not defined(X86_UNIT_TESTING_ONLY)
51			
52			// tell the driver to read keypad FIFO data and clear adp5587 HW interrupt registers
53			update_key_events();
54			// clear the falling flag for EXTI Line 5
55			EXTI->FPR1 = EXTI->FPR1 EXTI_IMR1_IM5;
56			
57			#endif
58			}
59			
60			
61			void CommonFunctions::enable_keypad_isr() { write_config_bits((ConfigReg::KE_IEN)); }
62			
63			
64			void CommonFunctions::disable_keypad_isr() { clear_config_bits(ConfigReg::KE_IEN); }
65			
66			
67			void CommonFunctions::enable_gpio_isr() { write_config_bits(ConfigReg::GPI_IEN); }
68			
69			
70			void CommonFunctions::disable_gpio_isr() { clear_config_bits(ConfigReg::GPI_IEN); }
71			
72			
73			void CommonFunctions::gpio_fifo_select(uint8_t row_mask, uint8_t col_mask0_7, uint8_t col_mask8_9)
74			{
75			write_register(Registers::GPI_EM_REG1, row_mask);
76			write_register(Registers::GPI_EM_REG2, col_mask0_7);
77			write_register(Registers::GPI_EM_REG3, col_mask8_9);
78			}
79			
80			
81			void CommonFunctions::keypad_gpio_select(uint8_t row_mask, uint8_t col_mask0_7, uint8_t col_mask8_9)
82			{
83			write_register(Registers::KP_GPIO1, row_mask);
84			write_register(Registers::KP_GPIO2, col_mask0_7);
85			write_register(Registers::KP_GPIO3, col_mask8_9);
86			}
87			
88			
89			void CommonFunctions::gpio_interrupt_select(uint8_t row_mask, uint8_t col_mask0_7, uint8_t col_mask8_9)
90			{
91			write_register(Registers::GPIO_INT_EN1, row_mask);
92			write_register(Registers::GPIO_INT_EN2, col_mask0_7);
93			write_register(Registers::GPIO_INT_EN3, col_mask8_9);
94			}
95			
96			
97			void CommonFunctions::set_gpo_out(uint8_t row_mask, uint8_t col_mask0_7, uint8_t col_mask8_9)

```

98 {
99     write_register(Registers::GPIO_DIR1, row_mask);
100     write_register(Registers::GPIO_DIR2, col_mask0_7);
101     write_register(Registers::GPIO_DIR3, col_mask8_9);
102 }
103
104
105 void CommonFunctions::set_gpi_active_high(uint8_t row_mask, uint8_t col_mask0_7, uint8_t col_mask8_9)
106 {
107     write_register(Registers::GPIO_INT_LVL1, row_mask);
108     write_register(Registers::GPIO_INT_LVL2, col_mask0_7);
109     write_register(Registers::GPIO_INT_LVL3, col_mask8_9);
110 }
111
112
113 void CommonFunctions::disable_debounce(uint8_t row_mask, uint8_t col_mask0_7, uint8_t col_mask8_9)
114 {
115     write_register(Registers::DEBOUNCE_DIS1, row_mask);
116     write_register(Registers::DEBOUNCE_DIS2, col_mask0_7);
117     write_register(Registers::DEBOUNCE_DIS3, col_mask8_9);
118 }
119
120
121 void CommonFunctions::disable_gpio_pullup(uint8_t row_mask, uint8_t col_mask0_7, uint8_t col_mask8_9)
122 {
123     write_register(Registers::GPIO_PULL1, row_mask);
124     write_register(Registers::GPIO_PULL2, col_mask0_7);
125     write_register(Registers::GPIO_PULL3, col_mask8_9);
126 }
127
128
129 bool CommonFunctions::probe_i2c()
130 {
131     bool success {true};
132
133     // check ADP5587 is listening on 0x60 (write). Left-shift of address is *not* required.
134     #ifndef X86_UNIT_TESTING_ONLY
135     if (stm32::i2c::send_addr(m_i2c_handle, m_i2c_addr, stm32::i2c::MsgType::PROBE) == stm32::i2c::Status::NACK)
136     {
137         success = false;
138     }
139     #endif
140     return success;
141 }
142
143 void CommonFunctions::clear_fifo_and_isr()
144 {
145     // clear the key event FIFO by reading each register
146
147     uint8_t ke_byte {0};
148     read_register(Registers::KEY_EVENTA, ke_byte);
149     read_register(Registers::KEY_EVENTB, ke_byte);
150     read_register(Registers::KEY_EVENTC, ke_byte);
151     read_register(Registers::KEY_EVENTD, ke_byte);
152     read_register(Registers::KEY_EVENTE, ke_byte);
153     read_register(Registers::KEY_EVENTF, ke_byte);
154     read_register(Registers::KEY_EVENTG, ke_byte);
155     read_register(Registers::KEY_EVENTH, ke_byte);
156     read_register(Registers::KEY_EVENTI, ke_byte);
157
158     // clear all interrupts
159     write_register(Registers::INT_STAT, (IntStatusReg::KE_INT | IntStatusReg::GPI_INT | IntStatusReg::K_LCK_INT | IntStatusReg::OVR_FLOW_INT));
160 }
161
162 void CommonFunctions::read_register(const uint8_t reg [[maybe_unused]], uint8_t &rx_byte [[maybe_unused]])
163 {
164     #if not defined(X86_UNIT_TESTING_ONLY)
165     // read this number of bytes
166     stm32::i2c::set_numbytes(m_i2c_handle, 1);
167
168     // send AD5587 write address and the register we want to read
169     stm32::i2c::send_addr(m_i2c_handle, m_i2c_addr, stm32::i2c::MsgType::WRITE);
170     stm32::i2c::send_byte(m_i2c_handle, reg);
171
172     // send AD5587 read address and get received data
173     stm32::i2c::send_addr(m_i2c_handle, m_i2c_addr, stm32::i2c::MsgType::READ);
174     stm32::i2c::receive_byte(m_i2c_handle, rx_byte);
175
176     stm32::i2c::generate_stop_condition(m_i2c_handle);
177
178     #if defined(USE_RTT)
179     switch(reg)
180     {
181     case 0x00:
182         SEGGER_RTT_printf(0, "\n\nDeviceID (%u): %u", +reg, +rx_byte);
183         break;
184     case 0x01:
185         SEGGER_RTT_printf(0, "\nConfiguration Register 1 (%u): %u", +reg, +rx_byte);
186         break;
187     case 0x02:
188         SEGGER_RTT_printf(0, "\nInterrupt status register (%u): %u", +reg, +rx_byte);
189         break;
190     case 0x03:
191         SEGGER_RTT_printf(0, "\nKeylock and event counter register (%u): %u", +reg, +rx_byte);
192         break;
193     case 0x04:
194         SEGGER_RTT_printf(0, "\nKey Event Register A (%u): %u", +reg, +rx_byte);
195         break;
196     case 0x05:
197         SEGGER_RTT_printf(0, "\nKey Event Register B (%u): %u", +reg, +rx_byte);
198         break;
199     case 0x06:
200         SEGGER_RTT_printf(0, "\nKey Event Register C (%u): %u", +reg, +rx_byte);
201         break;
202     case 0x07:
203         SEGGER_RTT_printf(0, "\nKey Event Register D (%u): %u", +reg, +rx_byte);
204         break;
205     case 0x08:

```



```

205         SEGGER_RTT_printf(0, "\nKey Event Register E (%u): %u", +reg, +rx_byte);
206         break;
207     case 0x09:
208         SEGGER_RTT_printf(0, "\nKey Event Register F (%u): %u", +reg, +rx_byte);
209         break;
210     case 0x11:
211         SEGGER_RTT_printf(0, "\nGPIO Interrupt Status 1: (%u): %u", +reg, +rx_byte);
212         break;
213     case 0x12:
214         SEGGER_RTT_printf(0, "\nGPIO Interrupt Status 2: (%u): %u", +reg, +rx_byte);
215         break;
216     case 0x13:
217         SEGGER_RTT_printf(0, "\nGPIO Interrupt Status 3: (%u): %u", +reg, +rx_byte);
218         break;
219     case 0x0A:
220         SEGGER_RTT_printf(0, "\nKey Event Register G (%u): %u", +reg, +rx_byte);
221         break;
222     case 0x0B:
223         SEGGER_RTT_printf(0, "\nKey Event Register H (%u): %u", +reg, +rx_byte);
224         break;
225     case 0x0C:
226         SEGGER_RTT_printf(0, "\nKey Event Register I (%u): %u", +reg, +rx_byte);
227         break;
228     case 0x0D:
229         SEGGER_RTT_printf(0, "\nKey Event Register J (%u): %u", +reg, +rx_byte);
230         break;
231     case 0x1D:
232         SEGGER_RTT_printf(0, "\nR0-R7 Keypad selection (%u): %u", +reg, +rx_byte);
233         break;
234     case 0x1E:
235         SEGGER_RTT_printf(0, "\nC0-C7 Keypad selection (%u): %u", +reg, +rx_byte);
236         break;
237     case 0x1F:
238         SEGGER_RTT_printf(0, "\nC8-C9 Keypad selection (%u): %u", +reg, +rx_byte);
239         break;
240     case 0x20:
241         SEGGER_RTT_printf(0, "\nGPI Key Mode 1 (%u): %u", +reg, +rx_byte);
242         break;
243     case 0x21:
244         SEGGER_RTT_printf(0, "\nGPI Key Mode 2 (%u): %u", +reg, +rx_byte);
245         break;
246     case 0x22:
247         SEGGER_RTT_printf(0, "\nGPI Key Mode 3 (%u): %u", +reg, +rx_byte);
248         break;
249     }
250
251     #endif // USE_RTT
252 #endif
253 }
254
255
256 void CommonFunctions::update_key_events()
257 {
258     // Steps for Key interrupt events
259     // 1. Check the specific type of interrupt in the Interrupt Status register (INT_STAT)
260     // 2. Check if there is event data in the FIFO by reading the event counter (KEY_LCK_EC_STAT:KEC[0:3])
261     // 3. Read (and implicitly clear the data) in the FIFO
262     // 4. Reset the interrupt statuses in Interrupt Status register (INT_STAT)
263
264     // 1. check if the INT_STAT bits are set
265     uint8_t int_stat_byte {0};
266     read_register(Registers::INT_STAT, int_stat_byte);
267
268     // if the ADP5587 interrupt register shows key or gpio event we need to process and reset the registers
269     if ((int_stat_byte & IntStatusReg::KE_INT) == IntStatusReg::KE_INT)
270     {
271         // 2. non-zero event counter means there is FIFO data to read (which also clears the FIFO)
272         uint8_t key_lck_ec_stat_byte {0};
273         read_register(Registers::KEY_LCK_EC_STAT, key_lck_ec_stat_byte);
274         if ((key_lck_ec_stat_byte & (KeyLckEvCntReg::KEC1 | KeyLckEvCntReg::KEC2 | KeyLckEvCntReg::KEC3 | KeyLckEvCntReg::KEC4)) > 0)
275         {
276             // 3. read the FIFO data (which also clears the FIFO and event counter)
277             read_fifo_bytes_from_hw();
278
279         }
280         // 4. Make sure we clear the interrupt status (by writing 1). Interrupts are blocked until the register is cleared.
281         write_register(Registers::INT_STAT, IntStatusReg::KE_INT);
282     }
283
284     // Steps for GPI interrupt events
285     // 1. Check the specific type of interrupt in the Interrupt Status register (INT_STAT)
286     // 2. Check if GPIO events were configured to be sent to key event FIFO (GPI_EM_REG1, GPI_EM_REG2, GPI_EM_REG3)
287     //    If so, read (and implicitly clear the data) in the FIFO
288     // 3. Read (and implicitly clear the GPI interrupt data) in GPIO_INT_STAT1, GPIO_INT_STAT2, GPIO_INT_STAT3
289     // 4. The Interrupt Status register (INT_STAT) can now be reset
290
291     // 1. confirm GPIO interrupt status
292     if ((int_stat_byte & IntStatusReg::GPI_INT) == IntStatusReg::GPI_INT)
293     {
294         // 2. if we enabled GPI interrupts in the event FIFO then we must read the event FIFO to clear that data
295         uint8_t read_gpi_em1_value{0};
296         uint8_t read_gpi_em2_value{0};
297         uint8_t read_gpi_em3_value{0};
298         read_register(Registers::GPI_EM_REG1, read_gpi_em1_value);
299         read_register(Registers::GPI_EM_REG2, read_gpi_em2_value);
300         read_register(Registers::GPI_EM_REG3, read_gpi_em3_value);
301
302         if ((read_gpi_em1_value | read_gpi_em2_value | read_gpi_em3_value) > 0)
303         {
304             read_fifo_bytes_from_hw();
305         }
306
307         // 3. We need to clear the GPI Interrupt Status before we can continue,
308         // Datasheet says read twice to clear but they can be stubborn so keep reading (usually 10x) until they clear.
309         uint8_t gpio_int_stat1_value{0};

```

```

311     uint8_t gpio_int_stat2_value{0};
312     uint8_t gpio_int_stat3_value{0};
313     read_register(Registers::GPIO_INT_STAT1, gpio_int_stat1_value);
314     read_register(Registers::GPIO_INT_STAT2, gpio_int_stat2_value);
315     read_register(Registers::GPIO_INT_STAT3, gpio_int_stat3_value);
316     while((gpio_int_stat1_value | gpio_int_stat2_value | gpio_int_stat3_value) > 0)
317     {
318         read_register(Registers::GPIO_INT_STAT1, gpio_int_stat1_value);
319         read_register(Registers::GPIO_INT_STAT2, gpio_int_stat2_value);
320         read_register(Registers::GPIO_INT_STAT3, gpio_int_stat3_value);
321     }
322
323     // 4. now we have cleared the cause of the interrupt,
324     // we can clear the GPI_INT bit in the shared Interrupt Status Register.
325     uint8_t int_stat_value{0};
326     write_register(Registers::INT_STAT, IntStatusReg::GPI_INT);
327     read_register(Registers::INT_STAT, int_stat_value);
328 }
329 }
330
331 void CommonFunctions::read_fifo_bytes_from_hw()
332 {
333     // read the FIFO bytes into class member byte array
334
335     uint8_t read_value {0};
336     read_register(Registers::KEY_EVENTA, read_value);
337     m_key_event_fifo.at(0) = static_cast<KeyPadMappings>(read_value);
338
339     read_register(Registers::KEY_EVENTB, read_value);
340     m_key_event_fifo.at(1) = static_cast<KeyPadMappings>(read_value);
341
342     read_register(Registers::KEY_EVENTC, read_value);
343     m_key_event_fifo.at(2) = static_cast<KeyPadMappings>(read_value);
344
345     read_register(Registers::KEY_EVENTD, read_value);
346     m_key_event_fifo.at(3) = static_cast<KeyPadMappings>(read_value);
347
348     read_register(Registers::KEY_EVENTE, read_value);
349     m_key_event_fifo.at(4) = static_cast<KeyPadMappings>(read_value);
350
351     read_register(Registers::KEY_EVENTF, read_value);
352     m_key_event_fifo.at(5) = static_cast<KeyPadMappings>(read_value);
353
354     read_register(Registers::KEY_EVENTG, read_value);
355     m_key_event_fifo.at(6) = static_cast<KeyPadMappings>(read_value);
356
357     read_register(Registers::KEY_EVENTH, read_value);
358     m_key_event_fifo.at(7) = static_cast<KeyPadMappings>(read_value);
359
360     read_register(Registers::KEY_EVENTI, read_value);
361     m_key_event_fifo.at(8) = static_cast<KeyPadMappings>(read_value);
362
363     read_register(Registers::KEY_EVENTJ, read_value);
364     m_key_event_fifo.at(9) = static_cast<KeyPadMappings>(read_value);
365
366 }
367
368
369 void CommonFunctions::write_config_bits(uint8_t config_bits)
370 {
371     uint8_t existing_byte {0};
372     read_register(Registers::CFG, existing_byte);
373     write_register(Registers::CFG, existing_byte | config_bits);
374     // maybe should read back and return bool based on comparison?
375     uint8_t new_byte {0};
376     read_register(Registers::CFG, new_byte);
377
378 }
379
380 void CommonFunctions::clear_config_bits(uint8_t config_bits)
381 {
382     uint8_t existing_byte {0};
383     read_register(Registers::CFG, existing_byte);
384     write_register(Registers::CFG, (existing_byte &= ~(config_bits)));
385 }
386
387
388
389
390
391
392
393 } // namespace adp5587
394

```