# GCC Code Coverage Report

| | | | Exec | Total | Coverage |
|---|---|---|---|---|---|
| Directory: | ./ | | | | |
| 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

| | | | Exec | Total | Coverage |
|---|---|---|---|---|---|
| Directory: | ./ | | | | |
| 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

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

Line Branch Exec Source

```
// MIT License

// Copyright (c) 2022 Chris Sutton

// Permission is hereby granted, free of charge, to any person obtaining a copy
// of this software and associated documentation files (the "Software"), to deal
// in the Software without restriction, including without limitation the rights
// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
// copies of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:

// The above copyright notice and this permission notice shall be included in all
// copies or substantial portions of the Software.

// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
// SOFTWARE.

#ifndef __ADP5587_COMMON_HPP__
#define __ADP5587_COMMON_HPP__

#include <restricted_base.hpp>
#include <i2c_utils.hpp>
#include <isr_manager_stm32g0.hpp>

#ifdef X86_UNIT_TESTING_ONLY
    #include <mock_cmsis.hpp>
#endif

namespace adp5587
{

class CommonFunctions
{
public:


    // @brief Incomplete list of ADP5587 device registers
    // see datasheet page 15 (https://www.analog.com/media/en/technical-documentation/data-sheets/adp5587.pdf)
    enum Registers
    {
        DEV_ID              = 0x00,
        CFG                 = 0x01,
        INT_STAT            = 0x02,
        KEY_LCK_EC_STAT     = 0x03,
        KEY_EVENTA          = 0x04,
        KEY_EVENTB          = 0x05,
        KEY_EVENTC          = 0x06,
        KEY_EVENTD          = 0x07,
        KEY_EVENTE          = 0x08,
        KEY_EVENTF          = 0x09,
        KEY_EVENTG          = 0x0A,
        KEY_EVENTH          = 0x0B,
        KEY_EVENTI          = 0x0C,
        KEY_EVENTJ          = 0x0D,
        GPIO_INT_STAT1      = 0x11,
        GPIO_INT_STAT2      = 0x12,
        GPIO_INT_STAT3      = 0x13,

        GPIO_INT_EN1        = 0x1A,
        GPIO_INT_EN2        = 0x1B,
        GPIO_INT_EN3        = 0x1C,
        KP_GPIO1            = 0x1D,
        KP_GPIO2            = 0x1E,
        KP_GPIO3            = 0x1F,
        GPI_EM_REG1         = 0x20,
        GPI_EM_REG2         = 0x21,
        GPI_EM_REG3         = 0x22,
        GPIO_DIR1           = 0x23,
        GPIO_DIR2           = 0x24,
        GPIO_DIR3           = 0x25,
        GPIO_INT_LVL1       = 0x26,
        GPIO_INT_LVL2       = 0x27,
        GPIO_INT_LVL3       = 0x28,
        DEBOUNCE_DIS1       = 0x29,
        DEBOUNCE_DIS2       = 0x30,
        DEBOUNCE_DIS3       = 0x31,
        GPIO_PULL1          = 0x32,
        GPIO_PULL2          = 0x33,
        GPIO_PULL3          = 0x34,

    };


    // @brief  Values for Keypad or GPIO selection registers
    enum KP_GPIO
    {
        R0 = 0b00000001,
        R1 = 0b00000010,
        R2 = 0b00000100,
        R3 = 0b00001000,
        R4 = 0b00010000,
        R5 = 0b00100000,
        R6 = 0b01000000,
        R7 = 0b10000000,

        C0 = 0b00000001,
        C1 = 0b00000010,
        C2 = 0b00000100,
        C3 = 0b00001000,
        C4 = 0b00010000,
        C5 = 0b00100000,
        C6 = 0b01000000,
        C7 = 0b10000000,
        C8 = 0b00000001,
        C9 = 0b00000010,
    };

    // Keypad release encodings. These values appear in the KeyEventReg entries after key press/release events
    // see datasheet page 9 (https://www.analog.com/media/en/technical-documentation/data-sheets/adp5587.pdf)
    // To get Key press events IDs, bitwise-OR the KeyPadMappings::XX_OFF values with KeyPadMappings::ON .
    // See the templated overload operator function below.
    enum class KeyPadMappings
    {
        INIT=0,
        // these default to key release events
        A7_OFF=71, A6_OFF=61, A5_OFF=51, A4_OFF=41, A3_OFF=31, A2_OFF=21, A1_OFF=11, A0_OFF=1,
        B7_OFF=72, B6_OFF=62, B5_OFF=52, B4_OFF=42, B3_OFF=32, B2_OFF=22, B1_OFF=12, B0_OFF=2,
        C7_OFF=73, C6_OFF=63, C5_OFF=53, C4_OFF=43, C3_OFF=33, C2_OFF=23, C1_OFF=13, C0_OFF=3,
        D7_OFF=74, D6_OFF=64, D5_OFF=54, D4_OFF=44, D3_OFF=34, D2_OFF=24, D1_OFF=14, D0_OFF=4,
        E7_OFF=75, E6_OFF=65, E5_OFF=55, E4_OFF=45, E3_OFF=35, E2_OFF=25, E1_OFF=15, E0_OFF=5,
        F7_OFF=76, F6_OFF=66, F5_OFF=56, F4_OFF=46, F3_OFF=36, F2_OFF=26, F1_OFF=16, F0_OFF=6,
        G7_OFF=77, G6_OFF=67, G5_OFF=57, G4_OFF=47, G3_OFF=37, G2_OFF=27, G1_OFF=17, G0_OFF=7,
        H7_OFF=78, H6_OFF=68, H5_OFF=58, H4_OFF=48, H3_OFF=38, H2_OFF=28, H1_OFF=18, H0_OFF=8,
        I7_OFF=79, I6_OFF=69, I5_OFF=59, I4_OFF=49, I3_OFF=39, I2_OFF=29, I1_OFF=19, I0_OFF=9,
        J7_OFF=80, J6_OFF=70, J5_OFF=60, J4_OFF=50, J3_OFF=40, J2_OFF=30, J1_OFF=20, J0_OFF=10,
        // this bit will be set if the key was pressed
        ON=128,
    };

    enum class GPIKeyMappings
    {
        // these default to key release events
        R0=97, R1=98, R2=99, R3=100, R4=101, R5=102, R6=103, R7=104,
        C0=105,C1=106,C2=107,C3=108, C4=109, C5=110, C6=111, C7=112, C8=113, C9=114,
        // this bit will be set if the key was pressed
        ON=128,
    };


    // @brief Bitwise-OR two scoped enum literals together
    // @tparam SCOPED_ENUM The scoped enum type.
    // @param L The left literal operand
    // @param R The right literal operand
    // @return constexpr SCOPED_ENUM Returns the combined value as SCOPED_ENUM enum type
    template<typename SCOPED_ENUM>
    constexpr friend SCOPED_ENUM operator | (SCOPED_ENUM L, SCOPED_ENUM R)
    {
        return static_cast<SCOPED_ENUM>(static_cast<int>(L) | static_cast<int>(R));
    }


    // @brief Write the byte array to the ADP5587 register
    // @tparam REG_SIZE
    // @param reg The register to modify
    // @param tx_bytes The value to write
    void write_register(const uint8_t reg, uint8_t tx_byte);

 // @brief callback function for IsrManagerStm32g0
 // see stm32_interrupt_managers/inc/stm32g0_interrupt_manager_functional.hpp
    void exti_isr();

    // @brief global enable keypad interrupts
    void enable_keypad_isr();

    // @brief global disable keypad interrupts
    void disable_keypad_isr();
    // @brief global enable GPIO interrupts
    void enable_gpio_isr();

    // @brief global disable GPIO interrupts
    void disable_gpio_isr();

    // @brief Select inidividual row/col connections as keypad input. Omitted connections will be configured as GPI.
    void keypad_gpio_select(uint8_t row_mask, uint8_t col_mask0_7, uint8_t col_mask8_9);

    // @brief Select if GPI is included in event FIFO
    void gpio_fifo_select(uint8_t row_mask, uint8_t col_mask0_7, uint8_t col_mask8_9);

    // @brief Enable GPI interrupts on inidividual row/col
    void gpio_interrupt_select(uint8_t row_mask, uint8_t col_mask0_7, uint8_t col_mask8_9);

    // @brief Set the GPIO direction as output on indiviudal rows/cols
    void set_gpo_out(uint8_t row_mask, uint8_t col_mask0_7, uint8_t col_mask8_9);

    // @brief Set the GPIO lvl as active high on indiviudal rows/cols
    void set_gpi_active_high(uint8_t row_mask, uint8_t col_mask0_7, uint8_t col_mask8_9);

    // @brief Disable the GPIO debounce on indiviudal rows/cols
    void disable_debounce(uint8_t row_mask, uint8_t col_mask0_7, uint8_t col_mask8_9);

    // @brief Disable the GPIO pullup on indiviudal rows/cols
    void disable_gpio_pullup(uint8_t row_mask, uint8_t col_mask0_7, uint8_t col_mask8_9);

protected:

    // @brief The CMSIS mem-mapped I2C periph. Set in the c'tor
    // std::unique_ptr<I2C_TypeDef> m_i2c_handle;
    I2C_TypeDef* m_i2c_handle;

    // @brief local store for ADP5587 key event FIFO
    std::array<KeyPadMappings, 10> m_key_event_fifo {KeyPadMappings::INIT};

    // @brief Confirm ADP5587 replies to write_addr and read_addr with ACK
    // @return true if both are successful, false if either fail.
    bool probe_i2c();

    // @brief clear the Key Event Registers (KEY_EVENTx) by reading them and
    // clear the Interrupt status register (INT_STAT) by writing 1 to each bit
    void clear_fifo_and_isr();

private:

    // @brief The i2c slave address for ADP5587ACPZ-1-R7
    const uint8_t m_i2c_addr {0x60};

    // @brief Configuration Register 1
    enum ConfigReg
    {
        KE_IEN              = (1 << 0), // Key events interrupt enable.
        GPI_IEN             = (1 << 1), // GPI interrupt enable.
        K_LCK_IM            = (1 << 2), // Keypad lock interrupt mask.
        OVR_FLOW_IEN        = (1 << 3), // Overflow interrupt enable.
        INT_CFG             = (1 << 4), // Interrupt configuration.
        OVR_FLOW_M          = (1 << 5), // Overflow mode.
        GPIEM_CFG           = (1 << 6), // GPI event mode configuration.
        AUTO_INC            = (1 << 7), // 2C auto-increment. Burst read is supported; burst write is not supported.
    };

    // Interrupt status register
    enum IntStatusReg
    {
        KE_INT              = (1 << 0), // Key events interrupt status. When set, write 1 to clear.
        GPI_INT             = (1 << 1), // GPI interrupt status. When set, write 1 to clear.
        K_LCK_INT           = (1 << 2), // Keylock interrupt status. When set, write 1 to clear.
        OVR_FLOW_INT        = (1 << 3), // Overflow interrupt status. When set, write 1 to clear.
    };

    // Keylock and event counter register
    enum KeyLckEvCntReg
    {
        KEC1                = (1 << 0), // 3-bit key event count of key event register.
        KEC2                = (1 << 1),
        KEC3                = (1 << 2),
        KEC4                = (1 << 3),
        LCK1                = (1 << 4), // 2-bit keypad lock status[1:0] (00 = unlocked; 11 = locked; read-only bits).
        LCK2                = (1 << 5),
        K_LCK_EN            = (1 << 6), // 0: lock feature is disabled. 1: lock feature is enabled.
    };

    // @brief Read some bytes from the ADP5587 register
    // @param reg The register to read
    void read_register(const uint8_t reg, uint8_t &rx_byte);

    // @brief Updates the stored key events FIFO data and resets the HW ISR
    void update_key_events();

    // @brief Read the FIFO bytes into "m_key_event_fifo" member byte array
    void read_fifo_bytes_from_hw();

    void write_config_bits(uint8_t config_bits);
    void clear_config_bits(uint8_t config_bits);






};

} // namespace adp5587


#endif // __ADP5587_COMMON_HPP__
```

Generated by: GCOVR (Version 4.2)

# GCC Code Coverage Report

| | | | Exec | Total | Coverage |
|---|---|---|---|---|---|
| Directory: | ./ | | | | |
| 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

```
// MIT License

// Copyright (c) 2022 Chris Sutton

// Permission is hereby granted, free of charge, to any person obtaining a copy
// of this software and associated documentation files (the "Software"), to deal
// in the Software without restriction, including without limitation the rights
// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
// copies of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:

// The above copyright notice and this permission notice shall be included in all
// copies or substantial portions of the Software.

// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
// SOFTWARE.

#include <adp5587_common.hpp>
#if defined(USE_RTT)
#include <SEGGER_RTT.h>
#endif

namespace adp5587
{

void CommonFunctions::write_register(uint8_t reg [[maybe_unused]], uint8_t tx_byte [[maybe_unused]])
{
    #if not defined(X86_UNIT_TESTING_ONLY)
        // write this number of bytes: The data byte(s) AND the address byte
        stm32::i2c::set_numbytes(m_i2c_handle, 2);

        // send AD5587 write address and the register we want to write
        stm32::i2c::send_addr(m_i2c_handle, m_i2c_addr, stm32::i2c::MsgType::WRITE);
        stm32::i2c::send_byte(m_i2c_handle, reg);

        // send AD5587 read address and get received data
        stm32::i2c::send_byte(m_i2c_handle, tx_byte);

        stm32::i2c::generate_stop_condition(m_i2c_handle);
    #endif
}

void CommonFunctions::exti_isr()
{
#if not defined(X86_UNIT_TESTING_ONLY)

            // tell the driver to read keypad FIFO data and clear adp5587 HW interrupt registers
            update_key_events();
            // clear the falling flag for EXTI Line 5
            EXTI->FPR1 = EXTI->FPR1 | EXTI_IMR1_IM5;


#endif
}

void CommonFunctions::enable_keypad_isr() { write_config_bits((ConfigReg::KE_IEN)); }


void CommonFunctions::disable_keypad_isr() { clear_config_bits(ConfigReg::KE_IEN); }


void CommonFunctions::enable_gpio_isr() { write_config_bits(ConfigReg::GPI_IEN); }


void CommonFunctions::disable_gpio_isr() { clear_config_bits(ConfigReg::GPI_IEN); }


void CommonFunctions::gpio_fifo_select(uint8_t row_mask, uint8_t col_mask0_7, uint8_t col_mask8_9)
{
    write_register(Registers::GPI_EM_REG1, row_mask);
    write_register(Registers::GPI_EM_REG2, col_mask0_7);
    write_register(Registers::GPI_EM_REG3, col_mask8_9);
}


void CommonFunctions::keypad_gpio_select(uint8_t row_mask, uint8_t col_mask0_7, uint8_t col_mask8_9)
{
    write_register(Registers::KP_GPIO1, row_mask);
    write_register(Registers::KP_GPIO2, col_mask0_7);
    write_register(Registers::KP_GPIO3, col_mask8_9);
}


void CommonFunctions::gpio_interrupt_select(uint8_t row_mask, uint8_t col_mask0_7, uint8_t col_mask8_9)
{
    write_register(Registers::GPIO_INT_EN1, row_mask);
    write_register(Registers::GPIO_INT_EN2, col_mask0_7);
    write_register(Registers::GPIO_INT_EN3, col_mask8_9);
}


void CommonFunctions::set_gpo_out(uint8_t row_mask, uint8_t col_mask0_7, uint8_t col_mask8_9)
```

```
{
    write_register(Registers::GPIO_DIR1, row_mask);
    write_register(Registers::GPIO_DIR2, col_mask0_7);
    write_register(Registers::GPIO_DIR3, col_mask8_9);
}


void CommonFunctions::set_gpi_active_high(uint8_t row_mask, uint8_t col_mask0_7, uint8_t col_mask8_9)
{
    write_register(Registers::GPIO_INT_LVL1, row_mask);
    write_register(Registers::GPIO_INT_LVL2, col_mask0_7);
    write_register(Registers::GPIO_INT_LVL3, col_mask8_9);
}


void CommonFunctions::disable_debounce(uint8_t row_mask, uint8_t col_mask0_7, uint8_t col_mask8_9)
{
    write_register(Registers::DEBOUNCE_DIS1, row_mask);
    write_register(Registers::DEBOUNCE_DIS2, col_mask0_7);
    write_register(Registers::DEBOUNCE_DIS3, col_mask8_9);
}


void CommonFunctions::disable_gpio_pullup(uint8_t row_mask, uint8_t col_mask0_7, uint8_t col_mask8_9)
{
    write_register(Registers::GPIO_PULL1, row_mask);
    write_register(Registers::GPIO_PULL2, col_mask0_7);
    write_register(Registers::GPIO_PULL3, col_mask8_9);
}


bool CommonFunctions::probe_i2c()
{
 bool success {true};

    // check ADP5587 is listening on 0x60 (write). Left-shift of address is *not* required.
    #ifndef X86_UNIT_TESTING_ONLY
 if (stm32::i2c::send_addr(m_i2c_handle, m_i2c_addr, stm32::i2c::MsgType::PROBE) == stm32::i2c::Status::NACK)
    {
        success = false;
    }
    #endif
    return success;
}

void CommonFunctions::clear_fifo_and_isr()
{
    // clear the key event FIFO by reading each register
    uint8_t ke_byte {0};
    read_register(Registers::KEY_EVENTA, ke_byte);
    read_register(Registers::KEY_EVENTB, ke_byte);
    read_register(Registers::KEY_EVENTC, ke_byte);
    read_register(Registers::KEY_EVENTD, ke_byte);
    read_register(Registers::KEY_EVENTE, ke_byte);
    read_register(Registers::KEY_EVENTF, ke_byte);
    read_register(Registers::KEY_EVENTG, ke_byte);
    read_register(Registers::KEY_EVENTH, ke_byte);
    read_register(Registers::KEY_EVENTI, ke_byte);

    // clear all interrupts
    write_register(Registers::INT_STAT, (IntStatusReg::KE_INT | IntStatusReg::GPI_INT | IntStatusReg::K_LCK_INT | IntStatusReg::OVR_FLOW_INT));
}

void CommonFunctions::read_register(const uint8_t reg [[maybe_unused]], uint8_t &rx_byte [[maybe_unused]])
{
    #if not defined(X86_UNIT_TESTING_ONLY)
        // read this number of bytes
        stm32::i2c::set_numbytes(m_i2c_handle, 1);

        // send AD5587 write address and the register we want to read
        stm32::i2c::send_addr(m_i2c_handle, m_i2c_addr, stm32::i2c::MsgType::WRITE);
        stm32::i2c::send_byte(m_i2c_handle, reg);

        // send AD5587 read address and get received data
        stm32::i2c::send_addr(m_i2c_handle, m_i2c_addr, stm32::i2c::MsgType::READ);
        stm32::i2c::receive_byte(m_i2c_handle, rx_byte);

        stm32::i2c::generate_stop_condition(m_i2c_handle);

        #if defined(USE_RTT)
            switch(reg)
            {
                case 0x00:
                    SEGGER_RTT_printf(0, "\n\nDeviceID (%u): %u", +reg, +rx_byte);
                    break;
                case 0x01:
                    SEGGER_RTT_printf(0, "\nConfiguration Register 1 (%u): %u", +reg, +rx_byte);
                    break;
                case 0x02:
                    SEGGER_RTT_printf(0, "\nInterrupt status register (%u): %u", +reg, +rx_byte);
                    break;
                case 0x03:
                    SEGGER_RTT_printf(0, "\nKeylock and event counter register (%u): %u", +reg, +rx_byte);
                    break;
                case 0x04:
                    SEGGER_RTT_printf(0, "\nKey Event Register A (%u): %u", +reg, +rx_byte);
                    break;
                case 0x05:
                    SEGGER_RTT_printf(0, "\nKey Event Register B (%u): %u", +reg, +rx_byte);
                    break;
                case 0x06:
                    SEGGER_RTT_printf(0, "\nKey Event Register C (%u): %u", +reg, +rx_byte);
                    break;
                case 0x07:
                    SEGGER_RTT_printf(0, "\nKey Event Register D (%u): %u", +reg, +rx_byte);
                    break;
                case 0x08:
                    SEGGER_RTT_printf(0, "\nKey Event Register E (%u): %u", +reg, +rx_byte);
                    break;
                case 0x09:
                    SEGGER_RTT_printf(0, "\nKey Event Register F (%u): %u", +reg, +rx_byte);
                    break;
                case 0x11:
                    SEGGER_RTT_printf(0, "\nGPIO Interrupt Status 1: (%u): %u", +reg, +rx_byte);
                    break;
                case 0x12:
                    SEGGER_RTT_printf(0, "\nGPIO Interrupt Status 2: (%u): %u", +reg, +rx_byte);
                    break;
                case 0x13:
                    SEGGER_RTT_printf(0, "\nGPIO Interrupt Status 3: (%u): %u", +reg, +rx_byte);
                    break;
                case 0x0A:
                    SEGGER_RTT_printf(0, "\nKey Event Register G (%u): %u", +reg, +rx_byte);
                    break;
                case 0x0B:
                    SEGGER_RTT_printf(0, "\nKey Event Register H (%u): %u", +reg, +rx_byte);
                    break;
                case 0x0C:
                    SEGGER_RTT_printf(0, "\nKey Event Register I (%u): %u", +reg, +rx_byte);
                    break;
                case 0x0D:
                    SEGGER_RTT_printf(0, "\nKey Event Register J (%u): %u", +reg, +rx_byte);
                    break;
                case 0x1D:
                    SEGGER_RTT_printf(0, "\nR0-R7 Keypad selection (%u): %u", +reg, +rx_byte);
                    break;
                case 0x1E:
                    SEGGER_RTT_printf(0, "\nC0-C7 Keypad selection (%u): %u", +reg, +rx_byte);
                    break;
                case 0x1F:
                    SEGGER_RTT_printf(0, "\nC8-C9 Keypad selection (%u): %u", +reg, +rx_byte);
                    break;
                case 0x20:
                    SEGGER_RTT_printf(0, "\nGPI Key Mode 1 (%u): %u", +reg, +rx_byte);
                    break;
                case 0x21:
                    SEGGER_RTT_printf(0, "\nGPI Key Mode 2 (%u): %u", +reg, +rx_byte);
                    break;
                case 0x22:
                    SEGGER_RTT_printf(0, "\nGPI Key Mode 3 (%u): %u", +reg, +rx_byte);
                    break;
            }

        #endif // USE_RTT
    #endif
}


void CommonFunctions::update_key_events()
{
    // Steps for Key interrupt events
    // 1. Check the specific type of interrupt in the Interrupt Status regsister (INT_STAT)
    // 2. Check if there is event data in the FIFO by reading the event counter (KEY_LCK_EC_STAT:KEC[0:3])
    // 3. Read (and implicitly clear the data) in the FIFO
    // 4. Reset the interrupt statuses in Interrupt Status regsister (INT_STAT)

    // 1. check if the INT_STAT bits are set
    uint8_t int_stat_byte {0};
    read_register(Registers::INT_STAT, int_stat_byte);

    // if the ADP5587 interrupt register shows key or gpio event we need to process and reset the registers
    if ((int_stat_byte & IntStatusReg::KE_INT) == IntStatusReg::KE_INT)
    {
        // 2. non-zero event counter means there is FIFO data to read (which also clears the FIFO)
        uint8_t key_lck_ec_stat_byte {0};
        read_register(Registers::KEY_LCK_EC_STAT, key_lck_ec_stat_byte);
        if ((key_lck_ec_stat_byte & (KeyLckEvCntReg::KEC1 | KeyLckEvCntReg::KEC2 | KeyLckEvCntReg::KEC3 | KeyLckEvCntReg::KEC4)) > 0)
        {
            // 3. read the FIFO data (which also clears the FIFO and event counter)
            read_fifo_bytes_from_hw();

        }
        // 4. Make sure we clear the interrupt status (by writing 1). Interrupts are blocked until the register is cleared.
        write_register(Registers::INT_STAT, IntStatusReg::KE_INT);
    }

    // Steps for GPI interrupt events
    // 1. Check the specific type of interrupt in the Interrupt Status regsister (INT_STAT)
    // 2. Check if GPIO events were configured to be sent to key event FIFO (GPI_EM_REG1, GPI_EM_REG2, GPI_EM_REG3)
    //    If so, read (and implicitly clear the data) in the FIFO
    // 3. Read (and implicitly clear the GPI interrupt data) in GPIO_INT_STAT1, GPIO_INT_STAT2, GPIO_INT_STAT3
    // 4. The Interrupt Status regsister (INT_STAT) can now be reset

    // 1. confirm GPIO interrupt status
    if ((int_stat_byte & IntStatusReg::GPI_INT) == IntStatusReg::GPI_INT)
    {

        // 2. if we enabled GPI interrupts in the event FIFO then we must read the event FIFO to clear that data
        uint8_t read_gpi_em1_value{0};
        uint8_t read_gpi_em2_value{0};
        uint8_t read_gpi_em3_value{0};
        read_register(Registers::GPI_EM_REG1, read_gpi_em1_value);
        read_register(Registers::GPI_EM_REG2, read_gpi_em2_value);
        read_register(Registers::GPI_EM_REG3, read_gpi_em3_value);

        if ((read_gpi_em1_value | read_gpi_em2_value | read_gpi_em3_value) > 0)
        {
            read_fifo_bytes_from_hw();
        }

        // 3. We need to clear the GPI Interrupt Status before we can continue,
        // Datasheet says read twice to clear but they can be stubborn so keep reading (usually 10x) until they clear.
        uint8_t gpio_int_stat1_value{0};
        uint8_t gpio_int_stat2_value{0};
        uint8_t gpio_int_stat3_value{0};
        read_register(Registers::GPIO_INT_STAT1, gpio_int_stat1_value);
        read_register(Registers::GPIO_INT_STAT2, gpio_int_stat2_value);
        read_register(Registers::GPIO_INT_STAT3, gpio_int_stat3_value);
        while((gpio_int_stat1_value | gpio_int_stat2_value | gpio_int_stat3_value) > 0)
        {
            read_register(Registers::GPIO_INT_STAT1, gpio_int_stat1_value);
            read_register(Registers::GPIO_INT_STAT2, gpio_int_stat2_value);
            read_register(Registers::GPIO_INT_STAT3, gpio_int_stat3_value);
        }

        // 4. now we have cleared the cause of the interrupt,
        // we can clear the GPI_INT bit in the shared Interrupt Status Register.
        uint8_t int_stat_value{0};
        write_register(Registers::INT_STAT, IntStatusReg::GPI_INT);
        read_register(Registers::INT_STAT, int_stat_value);
    }
}

void CommonFunctions::read_fifo_bytes_from_hw()
{
    // read the FIFO bytes into class member byte array

    uint8_t read_value {0};
    read_register(Registers::KEY_EVENTA, read_value);
    m_key_event_fifo.at(0) = static_cast<KeyPadMappings>(read_value);

    read_register(Registers::KEY_EVENTB, read_value);
    m_key_event_fifo.at(1) = static_cast<KeyPadMappings>(read_value);

    read_register(Registers::KEY_EVENTC, read_value);
    m_key_event_fifo.at(2) = static_cast<KeyPadMappings>(read_value);

    read_register(Registers::KEY_EVENTD, read_value);
    m_key_event_fifo.at(3) = static_cast<KeyPadMappings>(read_value);

    read_register(Registers::KEY_EVENTE, read_value);
    m_key_event_fifo.at(4) = static_cast<KeyPadMappings>(read_value);

    read_register(Registers::KEY_EVENTF, read_value);
    m_key_event_fifo.at(5) = static_cast<KeyPadMappings>(read_value);

    read_register(Registers::KEY_EVENTG, read_value);
    m_key_event_fifo.at(6) = static_cast<KeyPadMappings>(read_value);

    read_register(Registers::KEY_EVENTH, read_value);
    m_key_event_fifo.at(7) = static_cast<KeyPadMappings>(read_value);

    read_register(Registers::KEY_EVENTI, read_value);
    m_key_event_fifo.at(8) = static_cast<KeyPadMappings>(read_value);

    read_register(Registers::KEY_EVENTJ, read_value);
    m_key_event_fifo.at(9) = static_cast<KeyPadMappings>(read_value);


}


void CommonFunctions::write_config_bits(uint8_t config_bits)
{
    uint8_t existing_byte {0};
    read_register(Registers::CFG, existing_byte);
    write_register(Registers::CFG, existing_byte | config_bits);
    // maybe should read back and return bool based on comparison?
    uint8_t new_byte {0};
    read_register(Registers::CFG, new_byte);

}

void CommonFunctions::clear_config_bits(uint8_t config_bits)
{
    uint8_t existing_byte {0};
    read_register(Registers::CFG, existing_byte);
    write_register(Registers::CFG, (existing_byte &= ~(config_bits)));
}








} // namespace adp5587
```

Generated by: GCOVR (Version 4.2)