






GCC Code Coverage Report

Directory: source/	Exec	Total	Coverage
Date: 2021-11-27 04:38:01	Lines: 253	404	62.6 %
Legend: low: < 75.0 % medium: >= 75.0 % high: >= 90.0 %	Branches: 187	377	49.6 %

File	Lines	Branches
cpp_ssd1306/inc/font.hpp	 100.0 % 8 / 8	100.0 % 2 / 2
cpp_ssd1306/inc/ssd1306.hpp	 82.9 % 34 / 41	66.7 % 40 / 60
cpp_ssd1306/src/ssd1306.cpp	 98.6 % 68 / 69	57.5 % 46 / 80
cpp_tlc5955/src/tlc5955.cpp	 52.2 % 143 / 274	45.6 % 99 / 217
main_app/src/mainapp.cpp	 0.0 % 0 / 12	0.0 % 0 / 18

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

GCC Code Coverage Report

Directory: [source/](#)

File: [source/cpp_ssd1306/inc/font.hpp](#)

Date: 2021-11-27 04:38:01

	Exec	Total	Coverage
Lines:	8	8	100.0 %
Branches:	2	2	100.0 %

Line	Branch	Exec	Source
1			
2			
3			<code>#ifndef __FONT_HPP__</code>
4			<code>#define __FONT_HPP__</code>
5			
6			<code>#include <stdint.h></code>
7			<code>#include <array></code>
8			<code>//#include <variant></code>
9			<code>//#include <fontdata.hpp></code>
10			
11			
12			
13			<code>namespace ssd1306</code>
14			<code>{</code>
15			
16			<code>void thing();</code>
17			
18			<code>template<std::size_t FONT_SIZE></code>
19			<code>class Font</code>
20			<code>{</code>
21			
22			<code>public:</code>
23			
24			<code>// @brief Construct a new Font object</code>
25			<code>Font() = default;</code>
26			
27			<code>// @brief function to get a font pixel (16bit half-word).</code>
28			<code>// @param idx The position in the font data array to retrieve data</code>
29			<code>// @return uint16_t The halfword of data we retrieve</code>
30		654	<code>bool get_pixel(size_t idx, uint32_t &bit_line)</code>
31			<code>{</code>
32	✓✓	654	<code>if (idx > data.size())</code>
33			<code>{</code>
34		3	<code>return false;</code>
35			<code>}</code>
36			<code>else</code>
37			<code>{</code>
38		651	<code>bit_line = static_cast<uint32_t>(data.at(idx));</code>
39		651	<code>return true;</code>
40			<code>}</code>
41			<code>}</code>
42			
43			<code>// @brief get the width member variable</code>
44			<code>// @return uint8_t the width value</code>
45		10659	<code>uint8_t width() { return m_width; }</code>
46			
47			<code>// @brief get the height member variable</code>
48			<code>// @return uint8_t the height value</code>
49		2007	<code>uint8_t height() { return m_height; }</code>
50			
51			<code>// @brief helper function to get the size of the private font data array.</code>
52			<code>// @return size_t the array size</code>
53		30	<code>size_t size() { return data.size(); }</code>
54			
55			<code>private:</code>
56			
57			<code>// @brief The width of the font in pixels</code>
58			<code>static uint8_t m_width;</code>
59			
60			<code>// @brief The height of the font in pixels</code>
61			<code>static uint8_t m_height;</code>
62			
63			<code>// @brief the font data</code>
64			<code>static std::array<uint16_t, FONT_SIZE> data;</code>
65			
66			<code>};</code>
67			

```
68 // specializations
69 typedef Font<475> Font3x5;
70 typedef Font<680> Font5x7;
71 typedef Font<950> Font7x10;
72 typedef Font<1710> Font11x16;
73 typedef Font<2470> Font16x26;
74
75 } // namespace ssd1306
76
77 #endif // __FONT_HPP__
```

GCC Code Coverage Report

Directory:	Exec	Total	Coverage
source/			
File: source/cpp_ssd1306/inc/ssd1306.hpp	Lines: 34	41	82.9 %
Date: 2021-11-27 04:38:01	Branches: 40	60	66.7 %

Line	Branch	Exec	Source
1			/*
2			* Display.hpp
3			*
4			* Created on: 7 Nov 2021
5			* Author: chris
6			*/
7			
8			#ifndef Display_HPP_
9			#define Display_HPP_
10			
11			#include <variant>
12			#include <font.hpp>
13			#include <sstream>
14			#include <iostream>
15			#include <array>
16			#include <utility>
17			
18			
19			
20			#ifdef USE_HAL_DRIVER
21			#include "stm32g0xx.h"
22			#include "main.h"
23			#include "spi.h"
24			#endif
25			
26			
27			namespace ssd1306
28			{
29			
30			// @brief
31			enum class Colour: uint16_t
32			{
33			Black = 0x00,
34			White = 0x01
35			};
36			
37			// @brief
38			class Display
39			{
40			public:
41	3		Display() = default;
42			
43			// @brief
44			bool init(void);
45			
46			// @brief
47			// @param colour
48			void fill(Colour colour);
49			
50			// @brief
51			bool update_screen(void);
52			
53			// @brief
54			// @param x
55			// @param y
56			// @param colour
57			bool draw_pixel(uint8_t x, uint8_t y, Colour colour);
58			
59			// @brief
60			// @tparam FONT_SIZE
61			// @param msg
62			// @param font
63			// @param x
64			// @param y
65			// @param bg
66			// @param fg
67			// @param padding
68			// @param update
69			
70			// @return char
71			template<std::size_t FONT_SIZE>
72			char write(std::stringstream &msg, Font<FONT_SIZE> &font, uint8_t x, uint8_t y, Colour bg, Colour fg, int padding, bool update);
73			
74			// @brief
75			// @tparam FONT_SIZE
76			// @param ss
77			// @param font
78			// @param colour
79			// @param padding
80			// @return char
81			template<std::size_t FONT_SIZE>
82			char write_string(std::stringstream &ss, Font<FONT_SIZE> &font, Colour colour, int padding);
83			
84			// @brief
85			// @tparam FONT_SIZE
86			// @param ch
87			// @param font
88			// @param colour
89			// @param padding
90			// @return char
91			template<std::size_t FONT_SIZE>
92			char write_char(char ch, Font<FONT_SIZE> &font, Colour colour, int padding);
93			
94			// @brief Set the cursor object
			// @param x

```

95 // @param y
96 bool set_cursor(uint8_t x, uint8_t y);
97
98 // @brief
99 void print_buffer_stdout();
100
101
102 private:
103
104 // @brief
105 void reset(void);
106
107 // @brief
108 // @param cmd_byte
109 bool write_command(uint8_t cmd_byte);
110
111 // @brief
112 // @param data_buffer
113 // @param data_buffer_size
114 bool write_data(uint8_t* data_buffer, size_t data_buffer_size);
115
116 // @brief
117 uint16_t m_currentx {0};
118
119 // @brief
120 uint16_t m_currenty {0};
121
122 // @brief
123 uint8_t m_inverted {0};
124
125 // @brief
126 uint8_t m_initialized {0};
127
128 // @brief
129 static constexpr uint16_t m_width {128};
130
131 // @brief
132 static constexpr uint16_t m_height {64};
133
134 // @brief
135 std::array<uint8_t, (m_width*m_height)/8> m_buffer;
136
137 #ifdef USE_HAL_DRIVER
138
139 // @brief
140 SPI_HandleTypeDef m_spi_port {hspl1};
141 // @brief
142 uint16_t m_cs_port {0};
143 // @brief
144 uint16_t m_cs_pin {0};
145 // @brief
146
147 GPIO_TypeDef* m_dc_port {SPI1_DC_GPIO_Port};
148 // @brief
149 uint16_t m_dc_pin {SPI1_DC_Pin};
150 // @brief
151 GPIO_TypeDef* m_reset_port {SPI1_RESET_GPIO_Port};
152 // @brief
153 uint16_t m_reset_pin {SPI1_RESET_Pin};
154
155 #endif
156
157 };
158
159 // Out-of-class definitions of member function templates
160
161 template<std::size_t FONT_SIZE>
162 char Display::write(std::stringstream &msg, Font<FONT_SIZE> &font, uint8_t x, uint8_t y, Colour bg, Colour fg, int padding, bool update)
163 {
164     12 fill(bg);
165     12 if (!set_cursor(x, y))
166     {
167         6 return 0;
168     }
169     6 char res = write_string(msg, font, fg, padding);
170     6 if (update)
171     {
172         6 update_screen();
173     }
174     6 return res;
175 }
176
177 template<std::size_t FONT_SIZE>
178 char Display::write_char(char ch, Font<FONT_SIZE> &font, Colour color, int padding)
179 {
180
181     // Check remaining space on current line
182     ✓xx✓ 54 if (m_width <= (m_currentx + font.height()) ||
183     ✓x 27 m_width <= (m_currenty + font.height()))
184     {
185         // Not enough space on current line
186         return 0;
187     }
188
189     // add extra leading horizontal space
190     ✓✓ 27 if (padding == 1)
191     {
192         ✓✓ 648 for(size_t n = 0; n < font.height(); n++)
193         {
194             ✓xx✓ 624 if (!draw_pixel(m_currentx, (m_currenty + n), Colour::Black))
195         {

```

196		return false;	
197		}	
198		}	
199	24	m_currentx += 1;	
200		}	
201			
202			
203		// Use the font to write	
204		uint32_t b;	
205	✓✓	651 for(size_t i = 0; i < font.height(); i++) {	
206	✓✓✓	627 if (!font.get_pixel((ch - 32) * font.height() + i, b)) { return false; }	
207	✓✓	10608 for(size_t j = 0; j < font.width(); j++) {	
208	✓✓	9984 if((b << j) & 0x8000)	
209		{	
210	✓✓	2958 if (color == (Colour::White))	
211		{	
212	✓✓✓	1479 if (!draw_pixel(m_currentx + j, (m_currenty + i), Colour::White))	
213		{	
214		return false;	
215		}	
216		}	
217		else	
218		{	
219	✓✓✓	1479 if (!draw_pixel(m_currentx + j, (m_currenty + i), Colour::Black))	
220		{	
221		return false;	
222		}	
223		}	
224		else	
225		{	
226			
227	✓✓	7026 if (color == (Colour::White))	
228		{	
229	✓✓✓	3513 if (!draw_pixel(m_currentx + j, (m_currenty + i), Colour::Black))	
230		{	
231		return false;	
232		}	
233		}	
234		else	
235		{	
236	✓✓✓	3513 if (!draw_pixel(m_currentx + j, (m_currenty + i), Colour::White))	
237		{	
238		return false;	
239		}	
240		}	
241		}	
242		}	
243		}	
244		}	
245			
246		// The current space is now taken	
247		24 m_currentx += font.width();	
248		// add extra leading horizontal space	
249	✓✓	24 if (padding == 1)	
250		24 m_currentx += 1;	
251			
252		// Return written char for validation	
253		24 return ch;	
254		}	
255			
256		template<std::size_t FONT_SIZE>	
257		33 char Display::write_string(std::stringstream &ss, Font<FONT_SIZE> &font, Colour color, int padding)	
258		{	
259		// Write until null-byte	
260		char ch;	
261	✓✓✓	33 while (ss.get(ch))	
262	✓✓	{	
263	✓✓✓	24 if (write_char(ch, font, color, padding) != ch)	
264		{	
265		// Char could not be written	
266		return ch;	
267		}	
268		}	
269			
270		// Everything ok	
271		9 return ch;	
272		}	
273			
274		} // namespace ssd1306	
275			
276		#endif /* Display_HPP_ */	

GCC Code Coverage Report

Directory: [source/](#)

File: [source/cpp_ssd1306/src/ssd1306.cpp](#)

Date: 2021-11-27 04:38:01

	Exec	Total	Coverage
Lines:	68	69	98.6 %
Branches:	46	80	57.5 %

Line	Branch	Exec	Source
1			/*
2			* Display.cpp
3			*
4			* Created on: 7 Nov 2021
5			* Author: chris
6			*/
7			
8			// https://cdn-shop.adafruit.com/datasheets/SSD1306.pdf
9			
10			#include "ssd1306.hpp"
11			#include <iomanip>
12			#include <bitset>
13			
14			namespace ssd1306
15			{
16			
17			18 bool Display::init(void)
18			{
19			18 bool res = true;
20			// Reset Display
21			18 reset();
22			
23			// Wait for the screen to boot
24			#ifdef USE_HAL_DRIVER
25			HAL_Delay(100);
26			#endif
27			// Init Display
28	X✓	18	if (!write_command(0xAE)) { return false; } //display off
29			
30	X✓	18	if (!write_command(0x20)) { return false; } //Set Memory Addressing Mode
31	X✓	18	if (!write_command(0x10)) { return false; } // 00,Horizontal Addressing Mode; 01,Vertical Addressing Mode;
32			// 10,Page Addressing Mode (RESET); 11,Invalid
33			
34	X✓	18	if (!write_command(0xB0)) { return false; } //Set Page Start Address for Page Addressing Mode,0-7
35			
36			
37	X✓	18	if (!write_command(0xC8)) { return false; } //Set COM Output Scan Direction
38			
39			
40	X✓	18	if (!write_command(0x00)) { return false; } //--set low column address
41	X✓	18	if (!write_command(0x10)) { return false; } //--set high column address
42			
43	X✓	18	if (!write_command(0x40)) { return false; } //--set start line address - CHECK
44			
45	X✓	18	if (!write_command(0x81)) { return false; } //--set contrast control register - CHECK
46	X✓	18	if (!write_command(0xFF)) { return false; }
47			
48			
49	X✓	18	if (!write_command(0xA1)) { return false; } //--set segment re-map 0 to 127 - CHECK
50			
51			
52			
53	X✓	18	if (!write_command(0xA6)) { return false; } //--set normal color
54			
55			
56	X✓	18	if (!write_command(0xA8)) { return false; } //--set multiplex ratio(1 to 64) - CHECK
57	X✓	18	if (!write_command(0x3F)) { return false; } //
58			
59	X✓	18	if (!write_command(0xA4)) { return false; } //0xa4,Output follows RAM content;0xa5,Output ignores RAM content
60			
61	X✓	18	if (!write_command(0xD3)) { return false; } //-set display offset - CHECK
62	X✓	18	if (!write_command(0x00)) { return false; } //-not offset
63			
64	X✓	18	if (!write_command(0xD5)) { return false; } //--set display clock divide ratio/oscillator frequency
65	X✓	18	if (!write_command(0xF0)) { return false; } //--set divide ratio
66			
67	X✓	18	if (!write_command(0xD9)) { return false; } //--set pre-charge period
68	X✓	18	if (!write_command(0x22)) { return false; } //
69			
70	X✓	18	if (!write_command(0xDA)) { return false; } //--set com pins hardware configuration - CHECK
71	X✓	18	if (!write_command(0x12)) { return false; }
72			
73	X✓	18	if (!write_command(0xDB)) { return false; } //--set vcomh
74	X✓	18	if (!write_command(0x20)) { return false; } //0x20,0.77xVcc
75			
76	X✓	18	if (!write_command(0x8D)) { return false; } //--set DC-DC enable
77	X✓	18	if (!write_command(0x14)) { return false; } //
78	X✓	18	if (!write_command(0xAF)) { return false; } //--turn on Display panel
79			
80			// Clear screen

```

81      18      fill(Colour::Black);
82
83      // Flush buffer to screen
84      18      update_screen();
85
86      // Set default values for screen object
87      18      m_currentx = 0;
88      18      m_currenty = 0;
89
90      18      m_initialized = 1;
91
92      18      return res;
93  }
94
95
96      30      void Display::fill(Colour color)
97      {
98  ✓✓ 30750      for(auto &pixel : m_buffer)
99      {
100  ✓✓ 30720      pixel = (color == Colour::Black) ? 0x00 : 0xFF;
101      }
102      30  }
103
104      24      bool Display::update_screen(void)
105      {
106  ✓✓ 216      for(uint8_t i = 0; i < 8; i++)
107      {
108  ✗✓ 192      if (!write_command(0xB0 + i)) { return false; }
109  ✗✓ 192      if (!write_command(0x00)) { return false; }
110  ✗✓ 192      if (!write_command(0x10)) { return false; }
111  ✗✓ 192      if (!write_data(&m_buffer[m_width * i], m_width)) { return false; }
112      }
113      24      return true;
114      }
115
116      10608      bool Display::draw_pixel(uint8_t x, uint8_t y, Colour color)
117      {
118  ✓✓✓ 10608      if(x >= m_width || y >= m_height) {
119      // Don't write outside the buffer
120      return false;
121      }
122
123      // Draw in the right color
124  ✓✓ 10608      if(color == Colour::White) {
125      4992      m_buffer[x + (y / 8) * m_width] |= 1 << (y % 8);
126      } else {
127      5616      m_buffer[x + (y / 8) * m_width] &= ~(1 << (y % 8));
128      }
129
130      10608      return true;
131      }
132
133      12      bool Display::set_cursor(uint8_t x, uint8_t y)
134      {
135  ✓✓✓✓ 12      if(x >= m_width || y >= m_height)
136      {
137      6      return false;
138      }
139      else
140      {
141      6      m_currentx = x;
142      6      m_currenty = y;
143      }
144      6      return true;
145      }
146
147      18      void Display::print_buffer_stdout()
148      {
149
150
151      18      }
152
153
154      18      void Display::reset(void)
155      {
156      // CS = High (not selected)
157      //HAL_GPIO_WritePin(Display_CS_Port, Display_CS_Pin, GPIO_PIN_SET);
158
159      // Reset the Display
160      #ifndef USE_HAL_DRIVER
161      HAL_GPIO_WritePin(m_reset_port, m_reset_pin, GPIO_PIN_RESET);
162      HAL_Delay(10);
163      HAL_GPIO_WritePin(m_reset_port, m_reset_pin, GPIO_PIN_SET);
164      HAL_Delay(10);
165      #endif
166      18      }
167
168      1080      bool Display::write_command(uint8_t cmd_byte __attribute__((unused)))
169      {
170      #ifndef USE_HAL_DRIVER
171      HAL_StatusTypeDef res = HAL_OK;

```



```

172 //HAL_GPIO_WritePin(m_cs_port, m_cs_pin, GPIO_PIN_RESET); // select Display
173 HAL_GPIO_WritePin(m_dc_port, m_dc_pin, GPIO_PIN_RESET); // command
174 res = HAL_SPI_Transmit(&m_spi_port, (uint8_t *) &cmd_byte, 1, HAL_MAX_DELAY);
175 if (res != HAL_OK)
176 {
177     return false;
178 }
179 return true;
180 //HAL_GPIO_WritePin(m_cs_port, m_cs_pin, GPIO_PIN_SET); // un-select Display
181 #else
182 1080 return true;
183 #endif
184 }
185
186 192 bool Display::write_data(uint8_t* data_buffer __attribute__((unused)), size_t data_buffer_size __attribute__((unused)))
187 {
188 #ifdef USE_HAL_DRIVER
189     HAL_StatusTypeDef res = HAL_OK;
190     //HAL_GPIO_WritePin(m_cs_port, m_cs_pin, GPIO_PIN_RESET); // select Display
191     HAL_GPIO_WritePin(m_dc_port, m_dc_pin, GPIO_PIN_SET); // data
192     res = HAL_SPI_Transmit(&m_spi_port, data_buffer, data_buffer_size, HAL_MAX_DELAY);
193     if (res != HAL_OK)
194     {
195         return false;
196     }
197     return true;
198     //HAL_GPIO_WritePin(m_cs_port, m_cs_pin, GPIO_PIN_SET); // un-select Display
199 #else
200 192 return true;
201 #endif
202
203
204
205 }
206
207 } // namespace ssd1306

```

GCC Code Coverage Report

Directory: [source/](#)

File: [source/cpp_tlc5955/src/tlc5955.cpp](#)

Date: 2021-11-27 04:38:01

	Exec	Total	Coverage
Lines:	143	274	52.2 %
Branches:	99	217	45.6 %

Line	Branch	Exec	Source
1			#include "tlc5955.hpp"
2			#include <sstream>
3			#include <cmath>
4			#include <cstring>
5			#ifdef USE_RTT
6			#include <SEGGER_RTT.h>
7			#endif
8			namespace tlc5955
9			{
10			
11			
12		3	uint16_t Driver::startup_tests()
13			{
14			// latch bit test
15	X	3	if (m_common_byte_register[0] != 0b00000000) built_in_test_fail++;
16	X	3	set_control_bit(true);
17	X	3	if (m_common_byte_register[0] != 0b10000000) built_in_test_fail++; // 128
18			
19			// control byte test
20			
21			// Ctrl1 10010110
22			// bits [=====]
23			// Bytes ===== [
24			// #0 #1
25			
26	X	3	set_ctrl_cmd_bits();
27	X	3	if (m_common_byte_register[0] != 0b11001011) built_in_test_fail++; // 203
28			
29			// padding bits test - bytes 1-48 should be empty
30	X	3	set_padding_bits();
31	✓	147	for (uint8_t idx = 1; idx < 49; idx++)
32			{
33	X	144	if (m_common_byte_register[idx] != 0) { built_in_test_fail++; }
34			}
35			
36			// function bits test
37			// bits [===]
38			// [==]
39			// Bytes #49 #50
40			
41	X	3	set_function_data(true, false, false, false, false);
42	X	3	if (m_common_byte_register[49] != 0b00000010) built_in_test_fail++; // 2
43	X	3	set_function_data(true, true, false, false, false);
44	X	3	if (m_common_byte_register[49] != 0b00000011) built_in_test_fail++; // 3
45	X	3	set_function_data(true, true, true, false, false);
46	X	3	if (m_common_byte_register[50] != 0b10000000) built_in_test_fail++; // 128
47	X	3	set_function_data(true, true, true, true, false);
48	X	3	if (m_common_byte_register[50] != 0b11000000) built_in_test_fail++; // 192
49	X	3	set_function_data(true, true, true, true, true);
50	X	3	if (m_common_byte_register[50] != 0b11100000) built_in_test_fail++; // 224
51			
52			// BC bits test
53			// BC blue green red
54			// bits [=====] [=====] [=====]
55			// bits ===== [=====] [=====]
56			// Bytes #50 #51 #52
57			
58		3	std::bitset<m_bc_data_resolution> bc_test_on {127};
59		3	std::bitset<m_bc_data_resolution> bc_test_off {0};
60			
61	X	3	if (m_common_byte_register[50] != 0b11100000) built_in_test_fail++; // 224
62	X	3	if (m_common_byte_register[51] != 0b00000000) built_in_test_fail++;
63	X	3	if (m_common_byte_register[52] != 0b00000000) built_in_test_fail++;
64			
65	X	3	set_bc_data(bc_test_on, bc_test_off, bc_test_off);
66	X	3	if (m_common_byte_register[50] != 0b11111111) built_in_test_fail++; // 255
67	X	3	if (m_common_byte_register[51] != 0b11000000) built_in_test_fail++; // 192
68	X	3	if (m_common_byte_register[52] != 0x00000000) built_in_test_fail++;
69			
70	X	3	set_bc_data(bc_test_off, bc_test_on, bc_test_off);
71	X	3	if (m_common_byte_register[50] != 0b11100000) built_in_test_fail++; // 224
72	X	3	if (m_common_byte_register[51] != 0b00111111) built_in_test_fail++; // 63
73	X	3	if (m_common_byte_register[52] != 0b10000000) built_in_test_fail++; // 128
74			
75	X	3	set_bc_data(bc_test_off, bc_test_off, bc_test_on);
76	X	3	if (m_common_byte_register[50] != 0b11100000) built_in_test_fail++; // 224
77	X	3	if (m_common_byte_register[51] != 0x00000000) built_in_test_fail++;
78	X	3	if (m_common_byte_register[52] != 0b01111111) built_in_test_fail++; // 127
79			
80	X	3	set_bc_data(bc_test_off, bc_test_off, bc_test_off);
81	X	3	if (m_common_byte_register[50] != 0b11100000) built_in_test_fail++; // 224
82	X	3	if (m_common_byte_register[51] != 0b00000000) built_in_test_fail++;
83	X	3	if (m_common_byte_register[52] != 0b00000000) built_in_test_fail++;
84			
85	X	3	set_bc_data(bc_test_on, bc_test_on, bc_test_on);
86	X	3	if (m_common_byte_register[50] != 0b11111111) built_in_test_fail++; // 255
87	X	3	if (m_common_byte_register[51] != 0b11111111) built_in_test_fail++; // 255
88	X	3	if (m_common_byte_register[52] != 0b11111111) built_in_test_fail++; // 255
89			
90			// MC B G R
91			// bits [=] [=] [=]
92			// bits [=====] [
93			// Bytes #53 #54
94			
95		3	std::bitset<m_mc_data_resolution> mc_test_on {7};
96		3	std::bitset<m_mc_data_resolution> mc_test_off {0};

```

97  ✓X 3      set_mc_data(mc_test_on, mc_test_off, mc_test_off);
98  ✓X 3      if (m_common_byte_register[53] != 0b11000000) built_in_test_fail++; // 224
99  ✓X 3      if (m_common_byte_register[54] != 0b00000000) built_in_test_fail++; // 0
100 ✓X 3      set_mc_data(mc_test_off, mc_test_on, mc_test_off);
101 ✓X 3      if (m_common_byte_register[53] != 0b00011100) built_in_test_fail++; // 28
102 ✓X 3      if (m_common_byte_register[54] != 0b00000000) built_in_test_fail++; // 0
103 ✓X 3      set_mc_data(mc_test_off, mc_test_off, mc_test_on);
104 ✓X 3      if (m_common_byte_register[53] != 0b00000011) built_in_test_fail++; // 3
105 ✓X 3      if (m_common_byte_register[54] != 0b10000000) built_in_test_fail++; // 128
106
107 ✓X 3      set_mc_data(mc_test_off, mc_test_off, mc_test_off);
108 ✓X 3      if (m_common_byte_register[53] != 0b00000000) built_in_test_fail++;
109 ✓X 3      if (m_common_byte_register[54] != 0b00000000) built_in_test_fail++;
110
111 3      std::bitset<m_mc_data_resolution> mc_test_blue {1};
112 3      std::bitset<m_mc_data_resolution> mc_test_green {1};
113 3      std::bitset<m_mc_data_resolution> mc_test_red {1};
114 ✓X 3      set_mc_data(mc_test_off, mc_test_off, mc_test_red);
115 ✓X 3      if (m_common_byte_register[53] != 0b00000000) built_in_test_fail++; // 0
116 ✓X 3      if (m_common_byte_register[54] != 0b10000000) built_in_test_fail++; // 128
117 ✓X 3      set_mc_data(mc_test_off, mc_test_off, mc_test_red <= 1);
118 ✓X 3      if (m_common_byte_register[53] != 0b00000001) built_in_test_fail++; // 1
119 ✓X 3      if (m_common_byte_register[54] != 0b00000000) built_in_test_fail++; // 0
120 ✓X 3      set_mc_data(mc_test_off, mc_test_off, mc_test_red <= 1);
121 ✓X 3      if (m_common_byte_register[53] != 0b00000010) built_in_test_fail++; // 2
122 ✓X 3      if (m_common_byte_register[54] != 0b00000000) built_in_test_fail++; // 0
123 ✓X 3      set_mc_data(mc_test_off, mc_test_green, mc_test_red <= 1);
124 ✓X 3      if (m_common_byte_register[53] != 0b00000100) built_in_test_fail++; // 4
125 ✓X 3      if (m_common_byte_register[54] != 0b00000000) built_in_test_fail++; // 0
126 ✓X 3      set_mc_data(mc_test_off, mc_test_green <= 1, mc_test_red);
127 ✓X 3      if (m_common_byte_register[53] != 0b00001000) built_in_test_fail++; // 8
128 ✓X 3      if (m_common_byte_register[54] != 0b00000000) built_in_test_fail++; // 0
129 ✓X 3      set_mc_data(mc_test_off, mc_test_green <= 1, mc_test_red);
130 ✓X 3      if (m_common_byte_register[53] != 0b00010000) built_in_test_fail++; // 16
131 ✓X 3      if (m_common_byte_register[54] != 0b00000000) built_in_test_fail++; // 0
132 ✓X 3      set_mc_data(mc_test_blue, mc_test_green <= 1, mc_test_red);
133 ✓X 3      if (m_common_byte_register[53] != 0b00100000) built_in_test_fail++; // 32
134 ✓X 3      if (m_common_byte_register[54] != 0b00000000) built_in_test_fail++; // 0
135 ✓X 3      set_mc_data(mc_test_blue <= 1, mc_test_green, mc_test_red);
136 ✓X 3      if (m_common_byte_register[53] != 0b01000000) built_in_test_fail++; // 64
137 ✓X 3      if (m_common_byte_register[54] != 0b00000000) built_in_test_fail++; // 0
138 ✓X 3      set_mc_data(mc_test_blue <= 1, mc_test_green, mc_test_red);
139 ✓X 3      if (m_common_byte_register[53] != 0b10000000) built_in_test_fail++; // 128
140 ✓X 3      if (m_common_byte_register[54] != 0b00000000) built_in_test_fail++; // 0
141
142
143
144 3      return built_in_test_fail;
145 }
146
147 1887 void Driver::set_value_nth_bit(uint8_t &target, bool value, uint16_t shift_idx)
148 {
149 ✓ 1887 if (value) { target |= (1U << shift_idx); }
150 1647 else { target &= ~(1U << shift_idx); }
151 1887 print_common_bits();
152 1887 }
153
154
155 3 void Driver::set_control_bit(bool ctrl_latch)
156 {
157     // Latch
158     // bits      =
159     // Bytes      [
160     //             #0
161
162     //m_common_bit_register.set(m_latch_offset, ctrl_latch);
163 3      set_value_nth_bit(m_common_byte_register[0], ctrl_latch, 7);
164 3  }
165
166 3 void Driver::set_ctrl_cmd_bits()
167 {
168
169     // Ctrl      10010110
170     // bits      [=====]
171     // Bytes      [=====] [
172     //             #0      #1
173
174     // 7 MSB bits of ctrl byte into 7 LSB of byte #0
175 ✓ 24 for (int8_t idx = m_ctrl_cmd_size_bits - 1; idx > 0; idx--)
176 {
177 21      set_value_nth_bit(m_common_byte_register[0], m_ctrl_cmd.test(idx), idx - 1 );
178
179 }
180
181     // the last m_ctrl_cmd bit in to MSB of byte #1
182 3      set_value_nth_bit(m_common_byte_register[1], m_ctrl_cmd.test(0), 7);
183
184 3  }
185
186 3 void Driver::set_padding_bits()
187 {
188
189     // Padding 0 ===== 79
190     // Bytes      [=====] [=====] [=====] [=====] [=====] [=====] [=====] [
191     //             #1      #2      #3      #4      #5      #6      #7      #8      #9      #10
192
193     // Padding 80 ===== 159
194     // Bytes      [=====] [=====] [=====] [=====] [=====] [=====] [=====] [
195     //             #11     #12     #13     #14     #15     #16     #17     #18     #19     #20
196
197     // Padding 160 ===== 239
198     // Bytes      [=====] [=====] [=====] [=====] [=====] [=====] [=====] [
199     //             #21     #22     #23     #24     #25     #26     #27     #28     #29     #30
200
201     // Padding 240 ===== 319

```

```

202 // Bytes          [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====] [
203 //                #31      #32      #33      #34      #35      #36      #37      #38      #39      #40
204
205 // Padding 320 [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====] 389
206 // Bytes          [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====]
207 //                #41      #42      #43      #44      #45      #46      #47      #48      #49
208
209 // first, we write 7 LSB bits of m_common_byte_register[1] = 0
210 ✓✓ 24 for (int8_t idx = 6; idx > -1; idx--)
211     {
212         21 set_value_nth_bit(m_common_byte_register[1], false, idx);
213     }
214
215 // The next 47 bytes are don't care padding = 0
216 const uint16_t padding_bytes_remaining = 470;
217 ✓✓ 138 for (uint16_t byte_idx = 2; byte_idx < padding_bytes_remaining; byte_idx++)
218     {
219         ✓✓ 1215 for (int8_t bit_idx = 7; bit_idx > -1; bit_idx--)
220             {
221                 1080 set_value_nth_bit(m_common_byte_register[byte_idx], false, bit_idx);
222             }
223     }
224
225 // lastly, we write 6 MSB bits of m_common_byte_register[49] = 0
226 ✓✓ 21 for (int8_t idx = 7; idx > 1; idx--)
227     {
228         18 set_value_nth_bit(m_common_byte_register[49], false, idx);
229     }
230
231 3 }
232
233 15 void Driver::set_function_data(bool DSPRPT, bool TMGRST, bool RFRESH, bool ESPWM, bool LSDVLT)
234 {
235
236     // Function
237     // bits      [===]
238     //           [=] [==
239     // Bytes      #49 #50
240
241     // if all are set to true, byte #49 = 3, byte #50 = 224
242     15 set_value_nth_bit(m_common_byte_register[49], DSPRPT, 1);
243     15 set_value_nth_bit(m_common_byte_register[49], TMGRST, 0);
244     15 set_value_nth_bit(m_common_byte_register[50], RFRESH, 7);
245     15 set_value_nth_bit(m_common_byte_register[50], ESPWM, 6);
246     15 set_value_nth_bit(m_common_byte_register[50], LSDVLT, 5);
247     15 }
248
249 15 void Driver::set_bc_data(std::bitset<m_bc_data_resolution> &blue_value,
250     std::bitset<m_bc_data_resolution> &green_value,
251     std::bitset<m_bc_data_resolution> &red_value)
252 {
253     // BC      blue green red
254     // bits      [=====] [=====] [=====]
255     // bits      [=====] [=====] [=====]
256     // Bytes      #50 #51 #52
257
258     // set 5 LSB of byte #50 to bits 6-2 of BC blue_value
259     ✓✓ 90 for (int8_t bit_idx = m_bc_data_resolution - 1; bit_idx > 1; bit_idx--)
260     {
261         // offset the bit position in byte #50 by 2 places.
262         75 set_value_nth_bit(m_common_byte_register[50], blue_value.test(bit_idx), bit_idx - 2);
263     }
264
265     // set the first 2 MSB bits of byte #51 to the last 2 LSB of blue_value
266     15 set_value_nth_bit(m_common_byte_register[51], blue_value.test(1), 7);
267     15 set_value_nth_bit(m_common_byte_register[51], blue_value.test(0), 6);
268
269     // set 5 LSB of byte #51 to bits 6-1 of BC green_value
270     ✓✓ 105 for (int8_t bit_idx = m_bc_data_resolution - 1; bit_idx > 0; bit_idx--)
271     {
272         // offset the bit position in byte #51 by 1 places.
273         90 set_value_nth_bit(m_common_byte_register[51], green_value.test(bit_idx), bit_idx - 1);
274     }
275
276     // set MSB of byte#52 to LSB of green_value
277     15 set_value_nth_bit(m_common_byte_register[52], green_value.test(0), 7);
278
279     // set 7 LSB of byte #50 to bits all 7 bits of BC red_value
280     ✓✓ 120 for (int8_t bit_idx = m_bc_data_resolution - 1; bit_idx > -1; bit_idx--)
281     {
282         // No offset for bit position in byte #52.
283         105 set_value_nth_bit(m_common_byte_register[52], red_value.test(bit_idx), bit_idx);
284     }
285
286     15 }
287
288 39 void Driver::set_mc_data(std::bitset<m_mc_data_resolution> &blue_value,
289     std::bitset<m_mc_data_resolution> green_value,
290     std::bitset<m_mc_data_resolution> &red_value)
291 {
292     // MC      B G R
293     // bits      [=] [=] [=]
294     // bits      [=====] [
295     // Bytes      #53 #54
296
297     // 3 bits of blue in 3 MSB of byte #51 == 128
298     39 set_value_nth_bit(m_common_byte_register[53], blue_value.test(m_mc_data_resolution - 1), 7);
299     39 set_value_nth_bit(m_common_byte_register[53], blue_value.test(m_mc_data_resolution - 2), 6);
300     39 set_value_nth_bit(m_common_byte_register[53], blue_value.test(m_mc_data_resolution - 3), 5);
301
302     // 3 bits of green in next 3 bits of byte #51 == 144
303     39 set_value_nth_bit(m_common_byte_register[53], green_value.test(m_mc_data_resolution - 1), 4);
304     39 set_value_nth_bit(m_common_byte_register[53], green_value.test(m_mc_data_resolution - 2), 3);
305     39 set_value_nth_bit(m_common_byte_register[53], green_value.test(m_mc_data_resolution - 3), 2);
306
307     // 3 bits of red in 2 LSB of byte #51 (== 146) and MSB of byte #52 (== 0)

```

```

308     set_value_nth_bit(m_common_byte_register[53], red_value.test(m_mc_data_resolution - 1), 1);
309     set_value_nth_bit(m_common_byte_register[53], red_value.test(m_mc_data_resolution - 2), 0);
310     set_value_nth_bit(m_common_byte_register[54], red_value.test(m_mc_data_resolution - 3), 7);
311
312 }
313
314 void Driver::set_dc_data(const uint8_t led_idx, std::bitset<m_dc_data_resolution> &blue_value,
315 std::bitset<m_dc_data_resolution> &green_value,
316 std::bitset<m_dc_data_resolution> &red_value)
317 {
318
319     // DC      B15  G15  R15  B14  G14  R14  B13  G13  R13  B12  G12  R12
320     // bits    [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====]
321     // Bytes    [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====]
322     //          #54    #55    #56    #57    #58    #59    #60    #61    #62    #63    #64
323
324     // DC      B11  G11  R11  B10  G10  R10  B9  G9  R9  B8  G8  R8
325     // bits    [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====]
326     // Bytes    [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====]
327     //          #64    #65    #66    #67    #68    #69    #70    #71    #72    #73    #74
328
329     // DC      B7  G7  R7  B6  G6  R6  B5  G5  R5  B4  G4  R4
330     // bits    [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====]
331     // Bytes    [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====]
332     //          #75    #76    #77    #78    #79    #80    #81    #82    #83    #84    #85
333
334
335
336
337     switch(led_idx)
338     {
339     case 0:
340
341         set_value_nth_bit(m_common_byte_register[93], blue_value.test(6), 3);
342         set_value_nth_bit(m_common_byte_register[93], blue_value.test(5), 2);
343         set_value_nth_bit(m_common_byte_register[93], blue_value.test(4), 1);
344         set_value_nth_bit(m_common_byte_register[93], blue_value.test(3), 0);
345         set_value_nth_bit(m_common_byte_register[94], blue_value.test(2), 7);
346         set_value_nth_bit(m_common_byte_register[94], blue_value.test(1), 6);
347         set_value_nth_bit(m_common_byte_register[94], blue_value.test(0), 5);
348
349         set_value_nth_bit(m_common_byte_register[94], green_value.test(6), 4);
350         set_value_nth_bit(m_common_byte_register[94], green_value.test(5), 3);
351         set_value_nth_bit(m_common_byte_register[94], green_value.test(4), 2);
352         set_value_nth_bit(m_common_byte_register[94], green_value.test(3), 1);
353         set_value_nth_bit(m_common_byte_register[94], green_value.test(2), 0);
354         set_value_nth_bit(m_common_byte_register[95], green_value.test(1), 7);
355         set_value_nth_bit(m_common_byte_register[95], green_value.test(0), 6);
356
357         set_value_nth_bit(m_common_byte_register[95], red_value.test(6), 5);
358         set_value_nth_bit(m_common_byte_register[95], red_value.test(5), 4);
359         set_value_nth_bit(m_common_byte_register[95], red_value.test(4), 3);
360         set_value_nth_bit(m_common_byte_register[95], red_value.test(3), 2);
361         set_value_nth_bit(m_common_byte_register[95], red_value.test(2), 1);
362         set_value_nth_bit(m_common_byte_register[95], red_value.test(1), 0);
363         set_value_nth_bit(m_common_byte_register[96], red_value.test(0), 7);
364
365         break;
366
367
368
369     case 1:
370         set_value_nth_bit(m_common_byte_register[90], blue_value.test(6), 0);
371         set_value_nth_bit(m_common_byte_register[91], blue_value.test(5), 7);
372         set_value_nth_bit(m_common_byte_register[91], blue_value.test(4), 6);
373         set_value_nth_bit(m_common_byte_register[91], blue_value.test(3), 5);
374         set_value_nth_bit(m_common_byte_register[91], blue_value.test(2), 4);
375         set_value_nth_bit(m_common_byte_register[91], blue_value.test(1), 3);
376         set_value_nth_bit(m_common_byte_register[91], blue_value.test(0), 2);
377
378         set_value_nth_bit(m_common_byte_register[91], green_value.test(6), 1);
379         set_value_nth_bit(m_common_byte_register[91], green_value.test(5), 0);
380         set_value_nth_bit(m_common_byte_register[92], green_value.test(4), 7);
381         set_value_nth_bit(m_common_byte_register[92], green_value.test(3), 6);
382         set_value_nth_bit(m_common_byte_register[92], green_value.test(2), 5);
383         set_value_nth_bit(m_common_byte_register[92], green_value.test(1), 4);
384         set_value_nth_bit(m_common_byte_register[92], green_value.test(0), 3);
385
386         set_value_nth_bit(m_common_byte_register[92], red_value.test(6), 2);
387         set_value_nth_bit(m_common_byte_register[92], red_value.test(5), 1);
388         set_value_nth_bit(m_common_byte_register[92], red_value.test(4), 0);
389         set_value_nth_bit(m_common_byte_register[93], red_value.test(3), 7);
390         set_value_nth_bit(m_common_byte_register[93], red_value.test(2), 6);
391         set_value_nth_bit(m_common_byte_register[93], red_value.test(1), 5);
392         set_value_nth_bit(m_common_byte_register[93], red_value.test(0), 4);
393
394         break;
395     case 2:
396
397         set_value_nth_bit(m_common_byte_register[88], blue_value.test(6), 5);
398         set_value_nth_bit(m_common_byte_register[88], blue_value.test(5), 4);
399         set_value_nth_bit(m_common_byte_register[88], blue_value.test(4), 3);
400         set_value_nth_bit(m_common_byte_register[88], blue_value.test(3), 2);
401         set_value_nth_bit(m_common_byte_register[88], blue_value.test(2), 1);
402         set_value_nth_bit(m_common_byte_register[88], blue_value.test(1), 0);
403         set_value_nth_bit(m_common_byte_register[89], blue_value.test(0), 7);
404
405         set_value_nth_bit(m_common_byte_register[89], green_value.test(6), 6);
406         set_value_nth_bit(m_common_byte_register[89], green_value.test(5), 5);
407         set_value_nth_bit(m_common_byte_register[89], green_value.test(4), 4);
408         set_value_nth_bit(m_common_byte_register[89], green_value.test(3), 3);
409         set_value_nth_bit(m_common_byte_register[89], green_value.test(2), 2);
410         set_value_nth_bit(m_common_byte_register[89], green_value.test(1), 1);
411         set_value_nth_bit(m_common_byte_register[89], green_value.test(0), 0);
412
413         set_value_nth_bit(m_common_byte_register[90], red_value.test(6), 7);
414         set_value_nth_bit(m_common_byte_register[90], red_value.test(5), 6);
415         set_value_nth_bit(m_common_byte_register[90], red_value.test(4), 5);

```

```

416         set_value_nth_bit(m_common_byte_register[90], red_value.test(3), 4);
417         set_value_nth_bit(m_common_byte_register[90], red_value.test(2), 3);
418         set_value_nth_bit(m_common_byte_register[90], red_value.test(1), 2);
419         set_value_nth_bit(m_common_byte_register[90], red_value.test(0), 1);
420
421         break;
422     case 3:
423
424         // DC      B3      G3      R3      B2      G2      R2      B1      G1      R1      B0      G0      R0
425         // bits    [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====]
426         // Bytes    == [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====] [=====]
427         //          #85    #86    #87    #88    #89    #90    #91    #92    #93    #94    #95    #96
428
429         set_value_nth_bit(m_common_byte_register[85], blue_value.test(6), 2);
430         set_value_nth_bit(m_common_byte_register[85], blue_value.test(5), 1);
431         set_value_nth_bit(m_common_byte_register[85], blue_value.test(4), 0);
432         set_value_nth_bit(m_common_byte_register[86], blue_value.test(3), 7);
433         set_value_nth_bit(m_common_byte_register[85], blue_value.test(2), 6);
434         set_value_nth_bit(m_common_byte_register[85], blue_value.test(1), 5);
435         set_value_nth_bit(m_common_byte_register[85], blue_value.test(0), 4);
436
437
438
439
440
441         break;
442     case 4:
443         break;
444     case 5:
445         break;
446     case 6:
447         break;
448     case 7:
449         break;
450     case 8:
451         break;
452     case 9:
453         break;
454     case 10:
455         break;
456     case 11:
457         break;
458     case 12:
459         break;
460     case 13:
461         break;
462     case 14:
463         break;
464     case 15:
465         break;
466
467     }
468 }
469
470
471 void Driver::set_all_dc_data(std::bitset<m_dc_data_resolution> &blue_value,
472                             std::bitset<m_dc_data_resolution> &green_value,
473                             std::bitset<m_dc_data_resolution> &red_value)
474 {
475     for (uint8_t led_idx = 0; led_idx < m_num_leds_per_chip; led_idx++)
476     {
477         set_dc_data(led_idx, blue_value, green_value, red_value);
478     }
479 }
480
481 void Driver::set_gs_data(uint8_t led_pos, std::bitset<16> &blue_value, std::bitset<16> &green_value, std::bitset<16> &red_value)
482 {
483     // offset for the current LED position
484     const uint16_t led_offset = m_gs_data_one_led_size_bits * led_pos;
485
486     // the current bit position within the GS section of the common register, starting at the section offset + LED offset
487     uint16_t gs_common_pos = m_gs_data_offset + led_offset;
488
489     // add each blue_value bit into the BC section of the common register
490     for (uint8_t idx = 0; idx < blue_value.size(); idx++)
491     {
492         // make sure we stay within bounds of the common register
493         if (gs_common_pos < m_common_reg_size_bits)
494         {
495             m_common_bit_register.set(gs_common_pos, blue_value[idx]);
496             gs_common_pos++;
497         }
498     }
499
500     // add each green_value bit into the GS section of the common register
501     for (uint8_t idx = 0; idx < green_value.size(); idx++)
502     {
503         // make sure we stay within bounds of the common register
504         if (gs_common_pos < m_common_reg_size_bits)
505         {
506             m_common_bit_register.set(gs_common_pos, green_value[idx]);
507             gs_common_pos++;
508         }
509     }
510
511     // add each red_value bit into the GS section of the common register
512     for (uint8_t idx = 0; idx < red_value.size(); idx++)
513     {
514         // make sure we stay within bounds of the common register
515         if (gs_common_pos < m_common_reg_size_bits)
516         {
517             m_common_bit_register.set(gs_common_pos, red_value[idx]);
518             gs_common_pos++;
519         }
520     }
521 }
522
523 void Driver::set_all_gs_data(std::bitset<m_gs_data_resolution> &blue_value,

```

```

524         std::bitset<m_gs_data_resolution> &green_value,
525         std::bitset<m_gs_data_resolution> &red_value)
526     {
527         for (uint8_t led_idx = 0; led_idx < m_num_leds_per_chip; led_idx++)
528         {
529             set_gs_data(led_idx, blue_value, green_value, red_value);
530         }
531     }
532
533
534
535 void Driver::send_data()
536 {
537     // clock the data through and latch
538     #ifdef USE_HAL_DRIVER
539         HAL_StatusTypeDef res = HAL_SPI_Transmit(&m_spi_interface, (uint8_t*)m_common_byte_register.data(), m_common_reg_size_bytes, HAL_MAX_DELAY);
540         UNUSED(res);
541     #endif
542     toggle_latch();
543 }
544
545 void Driver::toggle_latch()
546 {
547     #ifdef USE_HAL_DRIVER
548         HAL_Delay(m_latch_delay_ms);
549         HAL_GPIO_WritePin(m_lat_port, m_lat_pin, GPIO_PIN_SET);
550         HAL_Delay(m_latch_delay_ms);
551         HAL_GPIO_WritePin(m_lat_port, m_lat_pin, GPIO_PIN_RESET);
552         HAL_Delay(m_latch_delay_ms);
553     #endif
554 }
555
556 void Driver::flush_common_register()
557 {
558     m_common_bit_register.reset();
559     send_data();
560 }
561
562 1887 void Driver::print_common_bits()
563 {
564     #ifdef USE_RTT
565         SEGGER_RTT_printf(0, "\r\n");
566         for (uint16_t idx = 45; idx < 53; idx++)
567         {
568             SEGGER_RTT_printf(0, "%u ", +m_common_byte_register[idx]);
569         }
570     #endif
571 1887 }
572
573 // void Driver::flush_common_register()
574 // {
575 //     // reset the latch
576 //     HAL_GPIO_WritePin(m_lat_port, m_lat_pin, GPIO_PIN_RESET);
577 //
578 //     // clock-in the entire common shift register per daisy-chained chip before pulsing the latch
579 //     for (uint8_t shift_entire_reg = 0; shift_entire_reg < m_num_driver_ics; shift_entire_reg++)
580 //     {
581 //         // write the MSB bit low to signal greyscale data
582 //         HAL_GPIO_WritePin(m_sck_port, m_sck_pin, GPIO_PIN_RESET);
583 //         HAL_GPIO_WritePin(m_mosi_port, m_mosi_pin, GPIO_PIN_RESET);
584 //         HAL_GPIO_WritePin(m_sck_port, m_sck_pin, GPIO_PIN_SET);
585 //         HAL_GPIO_WritePin(m_sck_port, m_sck_pin, GPIO_PIN_RESET);
586 //
587 //         // Set all 16-bit colours to 0 greyscale
588 //         uint8_t grayscale_data[2] = {0x00, 0x00};
589 //         for (uint8_t idx = 0; idx < 16; idx++)
590 //         {
591 //             HAL_SPI_Transmit(&m_spi_interface, grayscale_data, 2, HAL_MAX_DELAY);
592 //             HAL_SPI_Transmit(&m_spi_interface, grayscale_data, 2, HAL_MAX_DELAY);
593 //             HAL_SPI_Transmit(&m_spi_interface, grayscale_data, 2, HAL_MAX_DELAY);
594 //         }
595 //     }
596 //
597 //     toggle_latch();
598 // }
599
600 // void Driver::enable_spi()
601 // {
602 //     HAL_GPIO_DeInit(GPIOB, TLC5955_SPI2_MOSI_Pin|TLC5955_SPI2_SCK_Pin);
603 //
604 //     m_spi_interface.Instance = SPI2;
605 //     m_spi_interface.Init.Mode = SPI_MODE_MASTER;
606 //     m_spi_interface.Init.Direction = SPI_DIRECTION_1LINE;
607 //     m_spi_interface.Init.DataSize = SPI_DATASIZE_8BIT;
608 //     m_spi_interface.Init.CLKPolarity = SPI_POLARITY_LOW;
609 //     m_spi_interface.Init.CLKPhase = SPI_PHASE_1EDGE;
610 //     m_spi_interface.Init.NSS = SPI_NSS_SOFT;
611 //     m_spi_interface.Init.BaudRatePrescaler = SPI_BAUDRATEPRESCALER_8;
612 //     m_spi_interface.Init.FirstBit = SPI_FIRSTBIT_MSB;
613 //     m_spi_interface.Init.TIMode = SPI_TIMODE_DISABLE;
614 //     m_spi_interface.Init.CRCCalculation = SPI_CRCCALCULATION_DISABLE;
615 //     m_spi_interface.Init.CRCPolynomial = 7;
616 //     m_spi_interface.Init.CRCLength = SPI_CRC_LENGTH_DATASIZE;
617 //     m_spi_interface.Init.NSSPMode = SPI_NSS_PULSE_DISABLE;
618 //
619 //     if (HAL_SPI_Init(&m_spi_interface) != HAL_OK) { Error_Handler(); }
620 //
621 //     __HAL_RCC_SPI2_CLK_ENABLE();
622 //     __HAL_RCC_GPIOB_CLK_ENABLE();
623 //
624 //     GPIO_InitTypeDef GPIO_InitStructure = {
625 //         TLC5955_SPI2_MOSI_Pin|TLC5955_SPI2_SCK_Pin,
626 //         GPIO_MODE_AF_PP,
627 //         GPIO_PULLDOWN,
628 //         GPIO_SPEED_FREQ_VERY_HIGH,
629 //         GPIO_AF1_SPI2,
630 //     };
631

```

```

632 // HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
633
634 // __HAL_SYSCFG_FASTMODEPLUS_ENABLE(SYSCFG_FASTMODEPLUS_PB8);
635
636 // }
637
638 // void Driver::disable_spi()
639 // {
640
641 // }
642
643 // void Driver::enable_gpio_output_only()
644 // {
645 // // disable SPI config
646 // __HAL_RCC_SPI2_CLK_DISABLE();
647 // HAL_GPIO_DeInit(GPIOB, TLC5955_SPI2_MOSI_Pin|TLC5955_SPI2_SCK_Pin);
648
649 // // GPIO Ports Clock Enable
650 // __HAL_RCC_GPIOB_CLK_ENABLE();
651
652 // // Configure GPIO pin Output Level
653 // HAL_GPIO_WritePin(GPIOB, TLC5955_SPI2_LAT_Pin|TLC5955_SPI2_GSCLK_Pin|TLC5955_SPI2_MOSI_Pin|TLC5955_SPI2_SCK_Pin, GPIO_PIN_RESET);
654
655 // // Configure GPIO pins
656 // GPIO_InitTypeDef GPIO_InitStruct = {
657 //     TLC5955_SPI2_LAT_Pin|TLC5955_SPI2_GSCLK_Pin|TLC5955_SPI2_MOSI_Pin|TLC5955_SPI2_SCK_Pin,
658 //     GPIO_MODE_OUTPUT_PP,
659 //     GPIO_PULLDOWN,
660 //     GPIO_SPEED_FREQ_VERY_HIGH,
661 //     0
662 // };
663
664 // HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
665
666 // __HAL_SYSCFG_FASTMODEPLUS_ENABLE(SYSCFG_FASTMODEPLUS_PB9);
667 // __HAL_SYSCFG_FASTMODEPLUS_ENABLE(SYSCFG_FASTMODEPLUS_PB6);
668 // __HAL_SYSCFG_FASTMODEPLUS_ENABLE(SYSCFG_FASTMODEPLUS_PB7);
669 // __HAL_SYSCFG_FASTMODEPLUS_ENABLE(SYSCFG_FASTMODEPLUS_PB8);
670
671 // }
672
673 } // namespace tlc5955

```


GCC Code Coverage Report

Directory: source/	Exec	Total	Coverage
File: source/main_app/src/mainapp.cpp	Lines: 0	12	0.0 %
Date: 2021-11-27 04:38:01	Branches: 0	18	0.0 %

Line	Branch	Exec	Source
1			/*
2			* mainapp.cpp
3			*
4			* Created on: 7 Nov 2021
5			* Author: chris
6			*/
7			
8			#include "mainapp.hpp"
9			#include <ssd1306.hpp>
10			#include <tlc5955.hpp>
11			#include <chrono>
12			#include <thread>
13			
14			#include <sstream>
15			
16			#ifdef __cplusplus
17			extern "C"
18			{
19			#endif
20			
21			
22			
23			void mainapp()
24			{
25			
26			static ssd1306::Font16x26 font;
27			static ssd1306::Display oled;
28			oled.init();
29			
30			// oled.fill(ssd1306::Colour::Black);
31			// oled.set_cursor(2, 0);
32			// std::stringstream text("Init LEDS");
33			// oled.write_string(text, small_font, ssd1306::Colour::White, 3);
34			// oled.update_screen();
35			
36			// std::bitset<tlc5955::Driver::m_bc_data_resolution> led_bc {127};
37			// std::bitset<tlc5955::Driver::m_mc_data_resolution> led_mc {4};
38			// std::bitset<tlc5955::Driver::m_dc_data_resolution> led_dc {127};
39			// std::bitset<tlc5955::Driver::m_gs_data_resolution> led_gs {32767};
40			// tlc5955::Driver leds;
41			
42			// leds.startup_tests();
43			
44			// leds.set_control_bit(true);
45			// leds.set_ctrl_cmd_bits();
46			// leds.set_padding_bits();
47			// leds.set_function_data(true, true, true, true, true);
48			
49			// leds.set_bc_data(led_bc, led_bc, led_bc);
50			// leds.set_mc_data(led_mc, led_mc, led_mc);
51			// // leds.set_all_dc_data(led_dc, led_dc, led_dc);
52			// leds.send_data();
53			//leds.flush_common_register();
54			
55			//leds.send_control_data();
56			uint8_t count = 0;
57			
58			while(true)
59			{
60			std::array<char, 10> digit_ascii {'0','1','2','3','4','5','6','7','8','9'};
61			std::stringstream msg;
62			msg << digit_ascii[count];
63			oled.write(msg, font, 2, 2, ssd1306::Colour::Black, ssd1306::Colour::White, 3, true);
64			if (count < digit_ascii.size() - 1) { count++; }
65			else { count = 0; }
66			
67			//leds.set_control_bit(false);

```
68 //leds.set_all_gs_data(led_gs, led_gs, led_gs);
69 // leds.send_data();
70 //leds.flush_common_register();
71 #ifdef USE_HAL_DRIVER
72     HAL_Delay(100);
73 #else
74     std::this_thread::sleep_for(std::chrono::milliseconds(100));
75 #endif
76 // leds.flush_common_register();
77 //HAL_Delay(1);
78 //HAL_GPIO_WritePin(TLC5955_SPI2_LAT_GPIO_Port, TLC5955_SPI2_LAT_Pin, GPIO_PIN_RESET);
79 }
80 }
81
82
83 #ifndef __cplusplus
84 }
85 #endif
```