






GCC Code Coverage Report

Directory: ./	Exec	Total	Coverage
Date: 2021-11-27 15:42:49	Lines: 252	402	62.7 %
Legend: low: < 75.0 % medium: >= 75.0 % high: >= 90.0 %	Branches: 184	373	49.3 %

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	65.0 % 39 / 60
cpp_ssd1306/src/ssd1306.cpp	 100.0 % 67 / 67	57.9 % 44 / 76
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: ./

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

Date: 2021-11-27 15:42:49

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

Line	Branch	Exec	Source
1			
2			
3			#ifndef __FONT_HPP__
4			#define __FONT_HPP__
5			
6			#include <stdint.h>
7			#include <array>
8			// #include <variant>
9			// #include <fontdata.hpp>
10			
11			
12			
13			namespace ssd1306
14			{
15			
16			template<std::size_t FONT_SIZE>
17			class Font
18			{
19			
20			public:
21			
22			// @brief Construct a new Font object
23			Font() = default;
24			
25			// @brief function to get a font pixel (16bit half-word).
26			// @param idx The position in the font data array to retrieve data
27			// @return uint16_t The halfword of data we retrieve
28		218	bool get_pixel(size_t idx, uint32_t &bit_line)
29			{
30	✓✓	218	if (idx > data.size())
31			{
32		1	return false;
33			}
34			else
35			{
36		217	bit_line = static_cast<uint32_t>(data.at(idx));
37		217	return true;
38			}
39			}
40			
41			// @brief get the width member variable
42			// @return uint8_t the width value
43		3553	uint8_t width() { return m_width; }
44			
45			// @brief get the height member variable
46			// @return uint8_t the height value
47		696	uint8_t height() { return m_height; }
48			
49			// @brief helper function to get the size of the private font data array.
50			// @return size_t the array size
51		10	size_t size() { return data.size(); }
52			
53			private:
54			
55			// @brief The width of the font in pixels
56			static uint8_t m_width;
57			
58			// @brief The height of the font in pixels
59			static uint8_t m_height;
60			
61			// @brief the font data
62			static std::array<uint16_t, FONT_SIZE> data;
63			
64			};
65			
66			// specializations
67			typedef Font<475> Font3x5;

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

GCC Code Coverage Report

Directory: ./

File: cpp_ssd1306/inc/ssd1306.hpp

Date: 2021-11-27 15:42:49

	Exec	Total	Coverage
Lines:	34	41	82.9 %
Branches:	39	60	65.0 %

Line	Branch	Exec	Source
1			/*
2			* Display.hpp
3			*
4			* Created on: 7 Nov 2021
5			* Author: chris
6			*/
7			
8			// @note See datasheet
9			// https://cdn-shop.adafruit.com/datasheets/SSD1306.pdf
10			
11			#ifndef Display_HPP_
12			#define Display_HPP_
13			
14			#include <variant>
15			#include <font.hpp>
16			#include <sstream>
17			#include <iostream>
18			#include <array>
19			#include <utility>
20			
21			
22			
23			#ifdef USE_HAL_DRIVER
24			#include "stm32g0xx.h"
25			#include "main.h"
26			#include "spi.h"
27			#endif
28			
29			
30			namespace ssd1306
31			{
32			
33			// @brief
34			enum class Colour: uint16_t
35			{
36			Black = 0x00,
37			White = 0x01
38			};
39			
40			// @brief
41			class Display
42			{
43			public:
44	1		Display() = default;
45			
46			// @brief
47			bool init();
48			
49			// @brief
50			// @param colour
51			void fill(Colour colour);
52			
53			// @brief
54			bool update_screen();
55			
56			// @brief
57			// @param x
58			// @param y
59			// @param colour
60			bool draw_pixel(uint8_t x, uint8_t y, Colour colour);
61			
62			// @brief
63			// @tparam FONT_SIZE
64			// @param msg
65			// @param font
66			// @param x
67			// @param y
68			// @param bg
69			
70			// @param fg
71			// @param padding
72			// @param update
73			// @return char
74			template<std::size_t FONT_SIZE>
75			char write(std::stringstream &msg, Font<FONT_SIZE> &font, uint8_t x, uint8_t y, Colour bg, Colour fg, bool padding, bool update);
76			
77			// @brief
78			// @tparam FONT_SIZE
79			// @param ss
80			// @param font
81			// @param colour
82			// @param padding
83			// @return char
84			template<std::size_t FONT_SIZE>
85			char write_string(std::stringstream &ss, Font<FONT_SIZE> &font, Colour colour, bool padding);
86			
87			// @brief
88			// @tparam FONT_SIZE
89			// @param ch
90			// @param font
91			// @param colour
92			// @param padding
93			// @return char
94			template<std::size_t FONT_SIZE>
			char write_char(char ch, Font<FONT_SIZE> &font, Colour colour, bool padding);

```

95
96 // @brief Set the cursor object
97 // @param x
98 // @param y
99 bool set_cursor(uint8_t x, uint8_t y);
100
101 // @brief
102 void print_buffer_stdout();
103
104
105 private:
106
107 // @brief
108 void reset();
109
110 // @brief
111 // @param cmd_byte
112 bool write_command(uint8_t cmd_byte);
113
114 // @brief
115 // @param data_buffer
116 // @param data_buffer_size
117 bool write_data(uint8_t* data_buffer, size_t data_buffer_size);
118
119 // @brief
120 uint16_t m_currentx {0};
121
122 // @brief
123 uint16_t m_currenty {0};
124
125 // @brief
126 uint8_t m_inverted {0};
127
128 // @brief
129 uint8_t m_initialized {0};
130
131 // @brief
132 static constexpr uint16_t m_width {128};
133
134 // @brief
135 static constexpr uint16_t m_height {64};
136
137 // @brief byte buffer for ssd1306
138 std::array<uint8_t, (m_width*m_height)/8> m_buffer;
139
140 #ifdef USE_HAL_DRIVER
141
142 // @brief
143 SPI_HandleTypeDef m_spi_port {hspi1};
144 // @brief
145 uint16_t m_cs_port {0};
146
147 // @brief
148 uint16_t m_cs_pin {0};
149 // @brief
150 GPIO_TypeDef* m_dc_port {SPI1_DC_GPIO_Port};
151 // @brief
152 uint16_t m_dc_pin {SPI1_DC_Pin};
153 // @brief
154 GPIO_TypeDef* m_reset_port {SPI1_RESET_GPIO_Port};
155 // @brief
156 uint16_t m_reset_pin {SPI1_RESET_Pin};
157
158 #endif
159
160 };
161
162 // Out-of-class definitions of member function templates
163
164 template<std::size_t FONT_SIZE>
165 char Display::write(std::stringstream &msg, Font<FONT_SIZE> &font, uint8_t x, uint8_t y, Colour bg, Colour fg, bool padding, bool update)
166 {
167     4 fill(bg);
168     4 if (!set_cursor(x, y))
169     {
170         2 return 0;
171     }
172     2 char res = write_string(msg, font, fg, padding);
173     2 if (update)
174     {
175         2 update_screen();
176     }
177     2 return res;
178 }
179
180 template<std::size_t FONT_SIZE>
181 char Display::write_char(char ch, Font<FONT_SIZE> &font, Colour color, bool padding)
182 {
183
184     // Check remaining space on current line
185     18 if (m_width <= (m_currentx + font.height()) ||
186     9 m_width <= (m_currenty + font.height()))
187     {
188         // Not enough space on current line
189         return 0;
190     }
191
192     // add extra leading horizontal space
193     9 if (padding)
194     {
195         243 for(size_t n = 0; n < font.height(); n++)

```

```

196 {
197 ✓xx 234 if (!draw_pixel(m_currentx, (m_currenty + n), Colour::Black))
198 {
199     return false;
200 }
201 }
202 9 m_currentx += 1;
203 }
204
205 // Use the font to write
206 uint32_t font_data_word;
207 ✓ 217 for(size_t font_height_idx = 0; font_height_idx < font.height(); font_height_idx++)
208 {
209 ✓xx 209 if (!font.get_pixel((ch - 32) * font.height() + font_height_idx, font_data_word)) { return false; }
210
211 ✓ 3536 for(size_t font_width_idx = 0; font_width_idx < font.width(); font_width_idx++)
212 {
213 ✓ 3328 if((font_data_word << font_width_idx) & 0x8000)
214 {
215 ✓ 986 if (color == (Colour::White))
216 {
217 ✓xx 493 if (!draw_pixel(m_currentx + font_width_idx, m_currenty + font_height_idx, Colour::White))
218 {
219     return false;
220 }
221
222     }
223     else
224     {
225 ✓xx 493 if (!draw_pixel(m_currentx + font_width_idx, m_currenty + font_height_idx, Colour::Black))
226 {
227     return false;
228 }
229     }
230     }
231     else
232     {
233 ✓ 2342 if (color == (Colour::White))
234 {
235 ✓xx 1171 if (!draw_pixel(m_currentx + font_width_idx, m_currenty + font_height_idx, Colour::Black))
236 {
237     return false;
238 }
239     }
240     else
241     {
242 ✓xx 1171 if (!draw_pixel(m_currentx + font_width_idx, m_currenty + font_height_idx, Colour::White))
243 {
244     return false;
245 }
246     }
247     }
248     }
249 }
250
251 // The current space is now taken
252 8 m_currentx += font.width();
253
254 // add extra leading horizontal space
255 ✓x 8 if (padding)
256 {
257     8 m_currentx += 1;
258 }
259
260 // Return written char for validation
261 8 return ch;
262 }
263
264 template<std::size_t FONT_SIZE>
265 char Display::write_string(std::stringstream &ss, Font<FONT_SIZE> &font, Colour color, bool padding)
266 {
267     // Write until null-byte
268     char ch;
269 ✓xx 11 while (ss.get(ch))
270 {
271     ✓xx 8 if (write_char(ch, font, color, padding) != ch)
272     {
273         // Char could not be written
274         return ch;
275     }
276 }
277
278 // Everything ok
279 3 return ch;
280 }
281
282 } // namespace ssd1306
283
284 #endif /* Display_HPP_ */
285

```

GCC Code Coverage Report

Directory: ./	Exec	Total	Coverage
File: cpp_ssd1306/src/ssd1306.cpp	Lines: 67	67	100.0 %
Date: 2021-11-27 15:42:49	Branches: 44	76	57.9 %

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

```

81 // Clear screen
82 6 fill(Colour::Black);
83
84 // Flush buffer to screen
85 6 update_screen();
86
87 // Set default values for screen object
88 6 m_currentx = 0;
89 6 m_currenty = 0;
90
91 6 m_initialized = 1;
92
93 6 return res;
94 }
95
96
97 10 void Display::fill(Colour color)
98 {
99 ✓✓ 10250 for(auto &pixel : m_buffer)
100 {
101 ✓✓ 10240 pixel = (color == Colour::Black) ? 0x00 : 0xFF;
102 }
103 10 }
104
105 8 bool Display::update_screen()
106 {
107 ✓✓ 72 for(uint8_t i = 0; i < 8; i++)
108 {
109 x✓ 64 if (!write_command(0xB0 + i)) { return false; }
110 x✓ 64 if (!write_command(0x00)) { return false; }
111 x✓ 64 if (!write_command(0x10)) { return false; }
112 x✓ 64 if (!write_data(&m_buffer[m_width * i], m_width)) { return false; }
113 }
114 8 return true;
115 }
116
117 3562 bool Display::draw_pixel(uint8_t x, uint8_t y, Colour color)
118 {
119 // Draw in the right color
120 ✓✓ 3562 if(color == Colour::White)
121 {
122 1664 m_buffer[x + (y / 8) * m_width] |= 1 << (y % 8);
123 }
124 else
125 {
126 1898 m_buffer[x + (y / 8) * m_width] &= ~(1 << (y % 8));
127 }
128
129 3562 return true;
130 }
131
132 4 bool Display::set_cursor(uint8_t x, uint8_t y)
133 {
134 ✓✓✓✓ 4 if(x >= m_width || y >= m_height)
135 {
136 2 return false;
137 }
138 else
139 {
140 2 m_currentx = x;
141 2 m_currenty = y;
142 }
143 2 return true;
144 }
145
146 6 void Display::print_buffer_stdout()
147 {
148
149
150 6 }
151
152
153 6 void Display::reset()
154 {
155 // CS = High (not selected)
156 //HAL_GPIO_WritePin(Display_CS_Port, Display_CS_Pin, GPIO_PIN_SET);
157
158 // Reset the Display
159 #ifdef USE_HAL_DRIVER
160 HAL_GPIO_WritePin(m_reset_port, m_reset_pin, GPIO_PIN_RESET);
161 HAL_Delay(10);
162 HAL_GPIO_WritePin(m_reset_port, m_reset_pin, GPIO_PIN_SET);
163 HAL_Delay(10);
164 #endif
165 6 }
166
167 360 bool Display::write_command(uint8_t cmd_byte __attribute__((unused)))
168 {
169 #ifdef USE_HAL_DRIVER
170 HAL_StatusTypeDef res = HAL_OK;
171 //HAL_GPIO_WritePin(m_cs_port, m_cs_pin, GPIO_PIN_RESET); // select Display

```



```

172 HAL_GPIO_WritePin(m_dc_port, m_dc_pin, GPIO_PIN_RESET); // command
173 res = HAL_SPI_Transmit(&m_spi_port, (uint8_t *) &cmd_byte, 1, HAL_MAX_DELAY);
174 if (res != HAL_OK)
175 {
176     return false;
177 }
178 return true;
179 //HAL_GPIO_WritePin(m_cs_port, m_cs_pin, GPIO_PIN_SET); // un-select Display
180 #else
181 360 return true;
182 #endif
183 }
184
185 64 bool Display::write_data(uint8_t* data_buffer __attribute__((unused)), size_t data_buffer_size __attribute__((unused)))
186 {
187 #ifdef USE_HAL_DRIVER
188     HAL_StatusTypeDef res = HAL_OK;
189     //HAL_GPIO_WritePin(m_cs_port, m_cs_pin, GPIO_PIN_RESET); // select Display
190     HAL_GPIO_WritePin(m_dc_port, m_dc_pin, GPIO_PIN_SET); // data
191     res = HAL_SPI_Transmit(&m_spi_port, data_buffer, data_buffer_size, HAL_MAX_DELAY);
192     if (res != HAL_OK)
193     {
194         return false;
195     }
196     return true;
197     //HAL_GPIO_WritePin(m_cs_port, m_cs_pin, GPIO_PIN_SET); // un-select Display
198 #else
199 64 return true;
200 #endif
201
202
203 }
204
205 } // namespace ssd1306
206

```

GCC Code Coverage Report

Directory: [J](#)

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

Date: 2021-11-27 15:42:49

	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			#ifndef USE_RTT
6			#include <SEGGER_RTT.h>
7			#endif
8			namespace tlc5955
9			{
10			
11			
12			uint16_t Driver::startup_tests()
13			{
14			// latch bit test
15	X	1	if (m_common_byte_register[0] != 0b00000000) built_in_test_fail++;
16	X	1	set_control_bit(true);
17	X	1	if (m_common_byte_register[0] != 0b10000000) built_in_test_fail++; // 128
18			
19			// control byte test
20			
21			// Ctrl 10010110
22			// bits [=====]
23			// Bytes [=====]
24			// #0 #1
25			
26	X	1	set_ctrl_cmd_bits();
27	X	1	if (m_common_byte_register[0] != 0b1001011) built_in_test_fail++; // 203
28			
29			// padding bits test - bytes 1-48 should be empty
30	X	1	set_padding_bits();
31	X	49	for (uint8_t idx = 1; idx < 49; idx++)
32			{
33	X	48	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	1	set_function_data(true, false, false, false, false);
42	X	1	if (m_common_byte_register[49] != 0b00000010) built_in_test_fail++; // 2
43	X	1	set_function_data(true, true, false, false, false);
44	X	1	if (m_common_byte_register[49] != 0b00000011) built_in_test_fail++; // 3
45	X	1	set_function_data(true, true, true, false, false);
46	X	1	if (m_common_byte_register[50] != 0b10000000) built_in_test_fail++; // 128
47	X	1	set_function_data(true, true, true, true, false);
48	X	1	if (m_common_byte_register[50] != 0b10000000) built_in_test_fail++; // 192
49	X	1	set_function_data(true, true, true, true, true);
50	X	1	if (m_common_byte_register[50] != 0b11000000) 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		1	std::bitset<m_bc_data_resolution> bc_test_on {127};
59		1	std::bitset<m_bc_data_resolution> bc_test_off {0};
60			
61	X	1	if (m_common_byte_register[50] != 0b11000000) built_in_test_fail++; // 224
62	X	1	if (m_common_byte_register[51] != 0b00000000) built_in_test_fail++;
63	X	1	if (m_common_byte_register[52] != 0b00000000) built_in_test_fail++;
64			
65	X	1	set_bc_data(bc_test_on, bc_test_off, bc_test_off);
66	X	1	if (m_common_byte_register[50] != 0b11111111) built_in_test_fail++; // 255
67	X	1	if (m_common_byte_register[51] != 0b10000000) built_in_test_fail++; // 192
68	X	1	if (m_common_byte_register[52] != 0x00000000) built_in_test_fail++;
69			
70	X	1	set_bc_data(bc_test_off, bc_test_on, bc_test_off);
71	X	1	if (m_common_byte_register[50] != 0b11000000) built_in_test_fail++; // 224
72	X	1	if (m_common_byte_register[51] != 0b00111111) built_in_test_fail++; // 63
73	X	1	if (m_common_byte_register[52] != 0b10000000) built_in_test_fail++; // 128
74			
75	X	1	set_bc_data(bc_test_off, bc_test_off, bc_test_on);
76	X	1	if (m_common_byte_register[50] != 0b11000000) built_in_test_fail++; // 224
77	X	1	if (m_common_byte_register[51] != 0x00000000) built_in_test_fail++;
78	X	1	if (m_common_byte_register[52] != 0b01111111) built_in_test_fail++; // 127
79			
80	X	1	set_bc_data(bc_test_off, bc_test_off, bc_test_off);
81	X	1	if (m_common_byte_register[50] != 0b11000000) built_in_test_fail++; // 224
82	X	1	if (m_common_byte_register[51] != 0b00000000) built_in_test_fail++;
83	X	1	if (m_common_byte_register[52] != 0b00000000) built_in_test_fail++;
84			
85	X	1	set_bc_data(bc_test_on, bc_test_on, bc_test_on);
86	X	1	if (m_common_byte_register[50] != 0b11111111) built_in_test_fail++; // 255
87	X	1	if (m_common_byte_register[51] != 0b11111111) built_in_test_fail++; // 255
88	X	1	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		1	std::bitset<m_mc_data_resolution> mc_test_on {7};
96		1	std::bitset<m_mc_data_resolution> mc_test_off {0};

```

97  ✓X 1 set_mc_data(mc_test_on, mc_test_off, mc_test_off);
98  ✓X 1 if (m_common_byte_register[53] != 0b11000000) built_in_test_fail++; // 224
99  ✓X 1 if (m_common_byte_register[54] != 0b00000000) built_in_test_fail++; // 0
100 ✓X 1 set_mc_data(mc_test_off, mc_test_on, mc_test_off);
101 ✓X 1 if (m_common_byte_register[53] != 0b00011100) built_in_test_fail++; // 28
102 ✓X 1 if (m_common_byte_register[54] != 0b00000000) built_in_test_fail++; // 0
103 ✓X 1 set_mc_data(mc_test_off, mc_test_off, mc_test_on);
104 ✓X 1 if (m_common_byte_register[53] != 0b00000011) built_in_test_fail++; // 3
105 ✓X 1 if (m_common_byte_register[54] != 0b10000000) built_in_test_fail++; // 128
106
107 ✓X 1 set_mc_data(mc_test_off, mc_test_off, mc_test_off);
108 ✓X 1 if (m_common_byte_register[53] != 0b00000000) built_in_test_fail++;
109 ✓X 1 if (m_common_byte_register[54] != 0b00000000) built_in_test_fail++;
110
111 1 std::bitset<m_mc_data_resolution> mc_test_blue {1};
112 1 std::bitset<m_mc_data_resolution> mc_test_green {1};
113 1 std::bitset<m_mc_data_resolution> mc_test_red {1};
114 ✓X 1 set_mc_data(mc_test_off, mc_test_off, mc_test_red);
115 ✓X 1 if (m_common_byte_register[53] != 0b00000000) built_in_test_fail++; // 0
116 ✓X 1 if (m_common_byte_register[54] != 0b10000000) built_in_test_fail++; // 128
117 ✓X 1 set_mc_data(mc_test_off, mc_test_off, mc_test_red <= 1);
118 ✓X 1 if (m_common_byte_register[53] != 0b00000001) built_in_test_fail++; // 1
119 ✓X 1 if (m_common_byte_register[54] != 0b00000000) built_in_test_fail++; // 0
120 ✓X 1 set_mc_data(mc_test_off, mc_test_off, mc_test_red <= 1);
121 ✓X 1 if (m_common_byte_register[53] != 0b00000010) built_in_test_fail++; // 2
122 ✓X 1 if (m_common_byte_register[54] != 0b00000000) built_in_test_fail++; // 0
123 ✓X 1 set_mc_data(mc_test_off, mc_test_green, mc_test_red <= 1);
124 ✓X 1 if (m_common_byte_register[53] != 0b00000100) built_in_test_fail++; // 4
125 ✓X 1 if (m_common_byte_register[54] != 0b00000000) built_in_test_fail++; // 0
126 ✓X 1 set_mc_data(mc_test_off, mc_test_green <= 1, mc_test_red);
127 ✓X 1 if (m_common_byte_register[53] != 0b000001000) built_in_test_fail++; // 8
128 ✓X 1 if (m_common_byte_register[54] != 0b00000000) built_in_test_fail++; // 0
129 ✓X 1 set_mc_data(mc_test_off, mc_test_green <= 1, mc_test_red);
130 ✓X 1 if (m_common_byte_register[53] != 0b00010000) built_in_test_fail++; // 16
131 ✓X 1 if (m_common_byte_register[54] != 0b00000000) built_in_test_fail++; // 0
132 ✓X 1 set_mc_data(mc_test_blue, mc_test_green <= 1, mc_test_red);
133 ✓X 1 if (m_common_byte_register[53] != 0b00100000) built_in_test_fail++; // 32
134 ✓X 1 if (m_common_byte_register[54] != 0b00000000) built_in_test_fail++; // 0
135 ✓X 1 set_mc_data(mc_test_blue <= 1, mc_test_green, mc_test_red);
136 ✓X 1 if (m_common_byte_register[53] != 0b01000000) built_in_test_fail++; // 64
137 ✓X 1 if (m_common_byte_register[54] != 0b00000000) built_in_test_fail++; // 0
138 ✓X 1 set_mc_data(mc_test_blue <= 1, mc_test_green, mc_test_red);
139 ✓X 1 if (m_common_byte_register[53] != 0b10000000) built_in_test_fail++; // 128
140 ✓X 1 if (m_common_byte_register[54] != 0b00000000) built_in_test_fail++; // 0
141
142
143
144 1 return built_in_test_fail;
145 }
146
147 629 void Driver::set_value_nth_bit(uint8_t &target, bool value, uint16_t shift_idx)
148 {
149 ✓ 629 if (value) { target |= (1U << shift_idx); }
150 549 else { target &= ~(1U << shift_idx); }
151 629 print_common_bits();
152 629 }
153
154
155 1 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 1 set_value_nth_bit(m_common_byte_register[0], ctrl_latch, 7);
164 1 }
165
166 1 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 ✓ 8 for (int8_t idx = m_ctrl_cmd_size_bits - 1; idx > 0; idx--)
176 {
177 7 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 1 set_value_nth_bit(m_common_byte_register[1], m_ctrl_cmd.test(0), 7);
183
184 1 }
185
186 1 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 ✓✓ 8 for (int8_t idx = 6; idx > -1; idx--)
211     {
212     7 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 ✓✓ 46 for (uint16_t byte_idx = 2; byte_idx < padding_bytes_remaining; byte_idx++)
218     {
219     ✓✓ 405 for (int8_t bit_idx = 7; bit_idx > -1; bit_idx--)
220         {
221         360 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 ✓✓ 7 for (int8_t idx = 7; idx > 1; idx--)
227     {
228     6 set_value_nth_bit(m_common_byte_register[49], false, idx);
229     }
230
231 1 }
232
233 5 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 5 set_value_nth_bit(m_common_byte_register[49], DSPRPT, 1);
243 5 set_value_nth_bit(m_common_byte_register[49], TMGRST, 0);
244 5 set_value_nth_bit(m_common_byte_register[50], RFRESH, 7);
245 5 set_value_nth_bit(m_common_byte_register[50], ESPWM, 6);
246 5 set_value_nth_bit(m_common_byte_register[50], LSDVLT, 5);
247 5 }
248
249 5 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 ✓✓ 30 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     25 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 5 set_value_nth_bit(m_common_byte_register[51], blue_value.test(1), 7);
267 5 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 ✓✓ 35 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     30 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 5 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 ✓✓ 40 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     35 set_value_nth_bit(m_common_byte_register[52], red_value.test(bit_idx), bit_idx);
284     }
285
286 5 }
287
288 13 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 13 set_value_nth_bit(m_common_byte_register[53], blue_value.test(m_mc_data_resolution - 1), 7);
299 13 set_value_nth_bit(m_common_byte_register[53], blue_value.test(m_mc_data_resolution - 2), 6);
300 13 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 13 set_value_nth_bit(m_common_byte_register[53], green_value.test(m_mc_data_resolution - 1), 4);
304 13 set_value_nth_bit(m_common_byte_register[53], green_value.test(m_mc_data_resolution - 2), 3);
305 13 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 13     set_value_nth_bit(m_common_byte_register[53], red_value.test(m_mc_data_resolution - 1), 1);
309 13     set_value_nth_bit(m_common_byte_register[53], red_value.test(m_mc_data_resolution - 2), 0);
310 13     set_value_nth_bit(m_common_byte_register[54], red_value.test(m_mc_data_resolution - 3), 7);
311
312 13 }
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
562 629 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
571 629 }
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_InitStruct = {
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: ./	Exec	Total	Coverage
Date: 2021-11-27 15:42:49	Lines: 252	402	62.7 %
Legend: low: < 75.0 % medium: >= 75.0 % high: >= 90.0 %	Branches: 184	373	49.3 %

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	65.0 % 39 / 60
cpp_ssd1306/src/ssd1306.cpp	 100.0 % 67 / 67	57.9 % 44 / 76
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: ./	Exec	Total	Coverage
File: main_app/src/mainapp.cpp	Lines: 0	12	0.0 %
Date: 2021-11-27 15:42:49	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
```