



FOCUS LCDs

LCDs MADE SIMPLE®

Ph. 480-503-4295 | NOPP@FocusLCDs.com

TFT | OLED | GRAPHIC | CHARACTER | UWVD | SEGMENT | CUSTOM

Application Note FAN9208

Programming the GT911 Capacitive Touch Controller

This user guide provides detailed information on programming the GT911 capacitive touch controller used on the E43GB-I-MW405-C MIPI Display.

Contents

Introduction	4
GT911	4
Focus LCDs E43GB-I-MW405-C	5
STM32H747I-DISCO Dev Board I2C Specification	6
Purpose	6
I2C Communication.....	7
I2C Peripheral Initialization	7
I2C4 Hardware and Pin Mapping.....	7
GPIO for I2C4	7
Enabling the I2C4 Peripheral Clock	8
Initialize the I2C4 Peripheral	9
Establish and Validate I2C Communications	10
Handling Errors and Debugging.....	10
GT911 Capacitive Touch Controller Overview	10
GT911 Configuration and Registers.....	11
GT911 I2C4 Connections and Pin Mapping	11
GT911 Register Map and Overview.....	11
GT911 Register Configuration	11
GT911 Initialization.....	12
Touch Events	14
Data Structure of Touch Events.....	14
Touch Initialization	14
Reading Touch Data.....	15
Processing Touch Events	15
Optimization Considerations.....	17
Gesture Recognition.....	18
Configuring the GT911 for Gesture Recognition	18
Processing Gesture Events	18
Gesture Mode vs. Touch Mode	21
Optimizing Gesture Handling	22
Application Integration	22
Conclusion	24
Recommended Next Steps.....	24
Additional Information.....	24
LCD Handling Precautions	25
Disclaimer	25
Revision History.....	25

List of Figures

Figure 1: E43GB-I-MW405-C	5
Figure 2: GPIO Configuration	8
Figure 3: RCC I2C and GPIO Peripheral Clock Enable	9
Figure 4: Initializing the I2C4 Peripheral	9
Figure 5: GT911 Main Registers	11
Figure 6: Reset Sequence	12
Figure 7: GT911 Initialization	12
Figure 8: GT911 Configuration	13
Figure 9: Touch Data Structure	14
Figure 10: Touch Interface Initialization	14
Figure 11: Reading the Touch Events from Touch_Process().....	15
Figure 12: The Call to Process_TouchEvents()	15
Figure 13: Retrieved the Raw Touch Data.....	16
Figure 14: Mapping the Coordinates	16
Figure 15: Touch Event Type	16
Figure 16: Touch Lift Off Processing.....	17
Figure 17: Touch Event Processing in the Callback Function	17
Figure 18: Capturing the Gesture Event.....	18
Figure 19: Process_Gestures().....	19
Figure 20: Pinch.....	19
Figure 21: Rotate.....	20
Figure 22: Swipe	20
Figure 23: Gesture Handler Part 1.....	21
Figure 24: Gesture Handler Part 2.....	21
Figure 25: I2C Config in Main	22
Figure 26: Touch Initialization in Main.....	23
Figure 27: Event Callback Handlers.....	23
Figure 28: The Main While Loop	23

Programming the GT911 Capacitive Touch Controller

The GT911 is a capacitive touch controller used in embedded systems for its precision and versatility. It supports up to 5-point multi-touch with high accuracy and fast response times. The GT911 also supports several built-in gesture recognitions, including swipe, multi-press, and pinch.

The controller features adjustable resolution, configurable touch sensitivity, and advanced noise immunity, making it ideal for modern touch-based applications. With its I2C interface, the GT911 simplifies integration into a variety of embedded platforms.

Introduction

This application note focuses on using the GT911 touch controller in conjunction with the STM32H747I-DISCO development board and the Focus LCDs E43GB-I-MW405-C display. The E43GB-I-MW405-C is a 4.3-inch TFT LCD with a resolution of 480 x 800 pixels, offering bright and vivid visuals for user interfaces. The display includes the GT911 capacitive touch controller, enabling responsive and precise touch input across its surface.

The STM32H747I-DISCO development board, based on the STM32H747 dual-core microcontroller, provides powerful processing capabilities and a rich set of peripherals. Its I2C4 peripheral is utilized for communication with the GT911 touch controller. The I2C4 interface operates with flexible clock configurations, supports multiple addressing modes, and integrates robust error handling, making it well-suited for handling touch data efficiently.

The driver code being developed in this application note takes advantage of the ARM CMSIS header files and some of the low-level STM32 HAL code for peripheral initialization. The GT911, Touch, and higher-level code is of a bare-metal design to facilitate adapting to other microcontroller platforms. Only the low-level peripheral code is dependent on the MCU architecture.

This document covers:

1. Initializing the I2C4 peripheral on the STM32H747I-DISCO.
2. Configuring and initializing the GT911 touch controller.
3. Reading and writing data to the GT911 touch controller via I2C.
4. Developing application-level code to process touch events and report X/Y coordinates.
5. Programming gesture recognition.

GT911

The GT911 is a high-performance capacitive touch controller manufactured by Goodix Technology supporting multi-touch and gesture detection.

Key features include:

- Support for up to 5-point multi-touch detection
- Built-in touch key functionality
- 32-bit CPU core for efficient touch processing
- Automatic calibration and drift compensation
- Low power consumption modes:
 - Normal working current: 3.5mA (typical)
 - Sleep mode current: < 50µA
 - Hibernation current: < 10µA
- Operating voltage range: 2.8V to 3.3V
- Sampling rate up to 100Hz
- High noise immunity and interference resistance
- Built-in voltage regulator
- Support for various screen sizes up to 10 inches
- I2C communication interface up to 400 kHz
- Programmable interrupt trigger conditions

Focus LCDs E43GB-I-MW405-C

The Focus LCDs E43GB-I-MW405-C is a 4.3-inch TFT LCD module with integrated GT911 capacitive touch controller.



Figure 1: E43GB-I-MW405-C

Display Characteristics:

- 4.3-inch TFT LCD panel
- Resolution: 480 x 800 pixels (portrait orientation)
- 16.7M colors (24-bit RGB interface)
- LED backlight with 500 cd/m² typical brightness
- Viewing angle: 80°/80°/80°/80° (L/R/U/D)
- Operating temperature: -20°C to +70°C

Touch Panel Specifications:

- GT911 capacitive touch controller
- 5-point multi-touch capability
- I2C interface (address 0xBA)
- 2mm cover glass thickness
- >85% optical transparency
- Surface hardness: 6H
- Touch response time: <5ms
- Position reporting accuracy: ±2mm

STM32H747I-DISCO Dev Board I2C Specification

The STM32H747I-DISCO development board features multiple I2C interfaces with the following capabilities:

- Four I2C interfaces (I2C1, I2C2, I2C3, and I2C4)
- Support for Standard Mode (SM, up to 100 kHz)
- Support for Fast Mode (FM, up to 400 kHz)
- Fast Mode Plus (FM+, up to 1 MHz) on all I2C peripherals
- Programmable digital noise filters
- Programmable analog noise filters
- SMBus 2.0/PMBus compatible
- Programmable timings and duty cycles
- DMA support for efficient data transfer
- Dual addressing capability
- 7-bit and 10-bit addressing modes
- Hardware CRC calculation

Purpose

This application note will walk through the implementation process, providing detailed C code snippets and explanations for each critical step integrating the GT911 touch controller on the Focus LCDs E43GB-I-MW405-C display with the STM32H747I-DISCO board.

I2C Communication

The GT911 touch controller communicates via I2C, specifically using I2C4 on the STM32H747I-DISCO board with a device address of 0xBA and a maximum clock speed of 400kHz. In this application the standard clock speed of 100kHz will be used for reliable operation.

The driver being developed includes error handling, timeout management, and proper peripheral initialization. It's configured for 100 kHz standard mode operation but can be adjusted for fast mode (400 kHz) by modifying the timing parameter in I2C_Init.

I2C Peripheral Initialization

The process of initializing the I2C peripheral on the STM32H747I-DISCO includes configuring the GPIO pins, initializing the I2C4 peripheral, and finally establishing I2C communications.

I2C4 Hardware and Pin Mapping

I2C4 is another I2C peripheral available on the STM32H747. On the STM32H747I-DISCO development board, I2C4 is typically mapped to the following pins:

- PD12 for I2C4_SCL (Clock Line)
- PD13 for I2C4_SDA (Data Line)

Check the board schematics to confirm the pin assignments and ensure no conflicts with other peripherals or features using these pins.

GPIO for I2C4

The GPIO pins used for I2C4 must be configured as alternate functions to support I2C communication. Proper pin configuration ensures signal integrity and compatibility with the GT911 touch controller.

1. Set **PD12** and **PD13** as **GPIO_MODE_AF_OD** (alternate function open drain). This configuration is essential for bidirectional communication on the I2C bus.
2. Enable the internal pull-up resistors (**GPIO_PULLUP**) on both pins to keep them in a known state during idle conditions.
3. Set the GPIO speed to "Fast" or "High" for high-frequency I2C communication.
4. Assign the alternate function for I2C4:
 - a. **PD12 – AF4_I2C4**
 - b. **PD13 – AF4_I2C4**

Ensure external pull-up resistors (typically 2.2 kΩ up to 4.7 kΩ) are connected to the I2C bus lines if internal pull-ups are insufficient for reliable operation.

```

204  /* MSP Initialization and De-initialization */
205  static void I2C_MspInit(void)
206  {
207      GPIO_InitTypeDef GPIO_InitStructure = {0};
208      RCC_PeriphCLKInitTypeDef RCC_PeriphClkInit = {0};
209
210      /* Enable GPIO and I2C clocks */
211      __HAL_RCC_GPIOD_CLK_ENABLE();
212      __HAL_RCC_I2C4_CLK_ENABLE();
213
214      /* Configure I2C4 clock source */
215      RCC_PeriphClkInit.PeriphClockSelection = RCC_PERIPHCLK_I2C4;
216      RCC_PeriphClkInit.I2c4ClockSelection = RCC_I2C4CLKSOURCE_D3PCLK1;
217      HAL_RCCEx_PeriphCLKConfig(&RCC_PeriphClkInit);
218
219      /* I2C4 GPIO Configuration
220       * PD12 -> I2C4_SCL
221       * PD13 -> I2C4_SDA
222       */
223      GPIO_InitStructure.Pin = GPIO_PIN_12|GPIO_PIN_13;
224      GPIO_InitStructure.Mode = GPIO_MODE_AF_OD;
225      GPIO_InitStructure.Pull = GPIO_PULLUP;
226      GPIO_InitStructure.Speed = GPIO_SPEED_FREQ_HIGH;
227      GPIO_InitStructure.Alternate = GPIO_AF4_I2C4;
228      HAL_GPIO_Init(GPIOD, &GPIO_InitStructure);
229
230      /* Enable I2C4 interrupt */
231      HAL_NVIC_SetPriority(I2C4_EV_IRQn, 0, 0);
  
```

Figure 2: GPIO Configuration

The function `I2C_MspInit()` creates a GPIO structure for configuring the IO pins. Once that structure is initialized the RCC clock for the GPIO peripheral is enabled. The next step is to populate the GPIO structure with the configuration to enable I2C on pins 12 and 13.

Enabling the I2C4 Peripheral Clock

Before using the I2C4 peripheral, its clock source must be enabled through the RCC (Reset and Clock Control).

1. Use the appropriate macro to enable the I2C4 clock, such as `HAL_RCC_I2C4_CLK_ENABLE()` in the HAL library.
2. Verify that the clock for GPIO port D is also enabled, as I2C4 pins are connected to this port. For example:

a. `HAL_RCC_GPIOD_CLK_ENABLE()`

Enabling these clocks ensures the GPIO and I2C peripherals can operate correctly.


```

204 /* MSP Initialization and De-initialization */
205 static void I2C_MspInit(void)
206 {
207     GPIO_InitTypeDef GPIO_InitStructure = {0};
208     RCC_PeriphCLKInitTypeDef RCC_PeriphClkInit = {0};
209
210     /* Enable GPIO and I2C clocks */
211     __HAL_RCC_GPIOC_CLK_ENABLE();
212     __HAL_RCC_I2C4_CLK_ENABLE();
213
214     /* Configure I2C4 clock source */
215     RCC_PeriphClkInit.PeriphClockSelection = RCC_PERIPHCLK_I2C4;
216     RCC_PeriphClkInit.I2c4ClockSelection = RCC_I2C4CLKSOURCE_D3PCLK1;
217     RCC_PeriphClkInit.I2c4ClockSelection = RCC_I2C4CLKSOURCE_D3PCLK1;

```

Figure 3: RCC I2C and GPIO Peripheral Clock Enable

Initialize the I2C4 Peripheral

The I2C4 peripheral must be initialized with settings compatible with the GT911 touch controller. This includes:

- **Clock Speed:** Set the I2C clock speed based on the GT911's requirements, such as 100 kHz (Standard Mode) or 400 kHz (Fast Mode).
- **Addressing Mode:** Configure I2C4 for 7-bit addressing, as the GT911 typically uses a 7-bit slave address (e.g., 0x5D [0xBA/0xBB] or 0x14 [0x28/0x29]).
- **Timing Configuration:** Calculate the timing register values using STM32CubeMX or manually determine them based on the reference manual. This configuration depends on the I2C clock source, peripheral clock frequency, and desired I2C speed.
- **Initialization:** Use HAL, LL (Low-Level) drivers, or direct register manipulation to initialize the I2C4 peripheral. Specify parameters such as the clock speed, addressing mode, and timing values.

```

84 I2C_Status I2C_Init(I2C_Config *config)
85 {
86     if (i2c_initialized)
87     {
88         return I2C_ERROR_BUSY;
89     }
90
91     /* Configure I2C handle */
92     hi2c4.Instance = I2C4;
93     hi2c4.Init.Timing = 0x10C0E0CF; /* 100 kHz standard mode with 54 MHz clock */
94     hi2c4.Init.OwnAddress1 = config->ownAddress << 1;
95     hi2c4.Init.AddressingMode = config->addressingMode == 0 ? I2C_ADDRESSINGMODE_7BIT : I2C_ADDRESSINGMODE_10BIT;
96     hi2c4.Init.DualAddressMode = config->dualAddressing == 0 ? I2C_DUALADDRESS_DISABLE : I2C_DUALADDRESS_ENABLE;
97     hi2c4.Init.GeneralCallMode = config->generalCall == 0 ? I2C_GENERALCALL_DISABLE : I2C_GENERALCALL_ENABLE;
98     hi2c4.Init.NoStretchMode = I2C_NOSTRETCH_DISABLE;
99
100     /* Initialize I2C peripheral */
101     I2C_MspInit();
102     if (HAL_I2C_Init(&hi2c4) != HAL_OK)
103     {
104         return I2C_ERROR_INIT;
105     }
106
107     /* Enable analog filter */
108     if (HAL_I2CEx_ConfigAnalogFilter(&hi2c4, I2C_ANALOGFILTER_ENABLE) != HAL_OK)
109     {
110         return I2C_ERROR_INIT;
111     }
112
113     i2c_initialized = 1;
114     return I2C_OK;
115 }

```

Figure 4: Initializing the I2C4 Peripheral

Establish and Validate I2C Communications

Once the I2C4 peripheral is initialized, you can communicate with the GT911 touch controller. The typical order of events is:

1. **Generate a Start Condition:** The master (STM32H747) generates a start condition by pulling the SDA line low while SCL remains high.
2. **Send the Slave Address:** Send the 7-bit GT911 address followed by the R/W bit (0 for write, 1 for read).
3. **Exchange Data:** Transmit or receive data in 8-bit chunks. For each byte sent, the slave must send an acknowledgment (ACK).
4. **Generate a Stop Condition:** The master releases the SDA line to high while SCL is high to signal the end of communication.

After configuring the I2C4 peripheral, verify the setup to ensure successful communication with the GT911. Use an oscilloscope or logic analyzer to monitor the SDA and SCL lines. Inspect the traffic for proper waveforms and timings on PD12 (SCL) and PD13 (SDA). Verify the presence of pull-up resistors (internal or external, typically between 2.2k Ω to 4.7k Ω , but could be as high as 10k Ω) on the I2C lines. Finally, test the basic read and write operations to ensure that the GT911 responds correctly to its I2C address.

Handling Errors and Debugging

I2C communication issues can occur due to electrical noise, incorrect configuration, or addressing errors. Implement robust error handling:

1. Check for acknowledgments (ACK/NACK) after every data frame.
2. Handle bus errors, arbitration losses, or timeouts using HAL or custom routines.
3. Implement retry mechanisms for transient failures.

GT911 Capacitive Touch Controller Overview

The GT911 is a capacitive touch controller that communicates via the I2C bus. It supports up to 5 touch points and offers flexible configuration for screen resolution and touch parameters.

Key features include: I2C slave mode with a default 7-bit address, configurable resolution and orientation, and registers for status, configuration, and touch data.

GT911 Configuration and Registers

GT911 I2C4 Connections and Pin Mapping

The GT911 is connected to the STM32H747I-DISCO's I2C4 peripheral as follows:

- I2C4 SCL (Clock Line): **PD12** – configured in the I2C code.
- I2C4 SDA (Data Line): **PD13** – configured in the I2C code
- Interrupt Line: **PK7** (used to detect touch events).
- Reset Line: **PG3** (used to reset the GT911).
- Configure **PK7** as an input pin for interrupt detection.
- Configure **PG3** as an output pin for resetting the GT911.

Verify these pin connections on the development board and ensure no pin conflicts.

GT911 Register Map and Overview

The GT911 touch controller features a comprehensive internal register map for configuration and control. Key register sections include:

Register Range	Description	Access
0x8040 – 0x8046	Command and Status	R/W
0x8047 – 0x80FF	Configuration	R/W
0x8100 – 0x813F	Coordinate Data	R
0x8140 – 0x814E	Product ID and Information	R
0x814F – 0x8156	Touch Point Data	R
0x8157 – 0x81FF	Reserved	-

GT911 Register Configuration

Key registers for GT911 configuration:

```

83  /* GT911 Register Map */
84  #define GT911_PRODUCT_ID      0x8140
85  #define GT911_FIRMWARE_VER   0x8144
86  #define GT911_CONFIG_VER     0x8047
87  #define GT911_CONFIG_FRESH   0x8100
88  #define GT911_READ_XY_REG    0x814E
89  #define GT911_POINT1_INFO    0x814F
90  #define GT911_POINT2_INFO    0x8157
91  #define GT911_POINT3_INFO    0x815F
92  #define GT911_POINT4_INFO    0x8167
93  #define GT911_POINT5_INFO    0x816F
94  #define GT911_STATUS_REG     0x814E
95  #define GT911_BUFFER_STATUS  0x814E
96
    
```

Figure 5: GT911 Main Registers

GT911 Initialization

The initialization of the GT911 begins with configuring the I2C peripheral on the STM32H747I-DISCO board. The I2C interface is enabled and set to operate at either standard (100kHz) or fast (400kHz) mode, depending on system requirements.

GPIO pins for the INT and RESET lines must also be configured. The RESET pin is set as an output, while the INT pin is configured as an open-drain output. The default state for these pins is High for RESET and Low for INT.

The GT911 is then initialized through a specific reset sequence. The sequence involves toggling the RESET pin, holding the INT pin low, and finally releasing it after bringing the RESET pin high. A wait time of 100 milliseconds ensures the controller has enough time to initialize.

```

180 GT911_Status GT911_Reset(void)
181 {
182     /* Hardware reset sequence would go here */
183     /* This typically involves toggling Reset and INT pins */
184     /* Reset sequence */
185     HAL_GPIO_WritePin(GPIOD, GPIO_PIN_3, GPIO_PIN_RESET);
186     HAL_Delay(10);
187     HAL_GPIO_WritePin(GPIOD, GPIO_PIN_3, GPIO_PIN_SET);
188     HAL_Delay(50);
189 }
    
```

Figure 6: Reset Sequence

Communication is verified by reading the Product ID (1) from the GT911's registers using its I2C address. A successful read operation (2 and 3) confirms that the device is properly initialized.

```

93 GT911_Status GT911_Init(GT911_Config *config)
94 {
95     uint8_t temp_buf[4];
96
97     /* Reset the device first */
98     if (GT911_Reset() != GT911_OK)
99     {
100         return GT911_ERROR_INIT;
101     }
102
103     /* Wait for device to stabilize */
104     HAL_Delay(100);
105
106     /* Read product ID to verify communication */
107     if (GT911_GetProductID(temp_buf) != GT911_OK) 1
108     {
109         return GT911_ERROR_INIT;
110     }
111
112     /* Verify product ID (should be "911" or similar) */
113     if (temp_buf[0] != '9' || temp_buf[1] != '1' || temp_buf[2] != '1') 2
114     {
115         return GT911_ERROR_INIT;
116     }
117
118     /* Configure the device */
119     return GT911_ConfigureDevice(config); 3
120 }
    
```

Figure 7: GT911 Initialization

The GT911’s configuration registers, starting at address 0x8047, allow fine-tuning of the touch controller’s behavior. The configuration process begins with preparing the required settings. Key parameters include the number of touch points (set to five), touch threshold, X and Y resolution (480x800), interrupt trigger mode, and refresh rate (set to 5 milliseconds). These values are written to the configuration registers sequentially.

```

238 static GT911_Status GT911_ConfigureDevice(GT911_Config *config)
239 {
240     uint8_t cfg_buf[256];
241     memcpy(cfg_buf, GT911_DEFAULT_CONFIG, sizeof(GT911_DEFAULT_CONFIG));
242
243     /* Modify configuration based on input parameters */
244     cfg_buf[0] = config->num_touch;
245     cfg_buf[4] = (config->x_resolution >> 8) & 0xFF;
246     cfg_buf[5] = config->x_resolution & 0xFF;
247     cfg_buf[6] = (config->y_resolution >> 8) & 0xFF;
248     cfg_buf[7] = config->y_resolution & 0xFF;
249
250     /* Write configuration */
251     if (GT911_WriteReg(GT911_CONFIG_VER, cfg_buf, sizeof(GT911_DEFAULT_CONFIG)) != G
252     {
253         return GT911_ERROR_CONFIG;
254     }
255
256     /* Trigger a config refresh */
257     uint8_t refresh_cmd = 0x01;
258     if (GT911_WriteReg(GT911_CONFIG_FRESH, &refresh_cmd, 1) != GT911_OK)
259     {
260         return GT911_ERROR_CONFIG;
261     }
262
263     /* Wait for refresh to complete */
264     HAL_Delay(100);
265
266     return GT911_OK;
267 }
    
```

Figure 8: GT911 Configuration

Several registers must be configured at initialization to function properly. Some of the key parameters are:

Register	Name	Value	Description
0x8047	X Output Max	0x01E0	Horizontal Resolution (480)
0x8049	Y Output Max	0x0320	Vertical Resolution (800)
0x804B	Touch Points	0x05	Maximum Number of Touch Points
0x8057	Module Switch 1	0x14	Enable Interrupt Trigger on Touch
0x805D	Refresh Rate	0x05	Touch Data Refresh Rate (5ms)

Once the configuration data is written, it is saved to non-volatile memory by writing to the specific control register located at 0x8040. Writing a value of 0x01 to this register ensures that the configuration is stored and will persist across power cycles and resets. After saving the configuration, it is necessary to write 0x80 to the same control register at 0x8040. This step transitions the GT911 from configuration mode to application (operation) mode, enabling it to process touch data.

Touch data is continuously updated in the GT911’s memory space, beginning at address 0x814E. The system can either poll these addresses periodically or rely on the interrupt mechanism to read the data only when a touch event occurs. The data includes details about the number of touch points, their coordinates, and other relevant attributes.

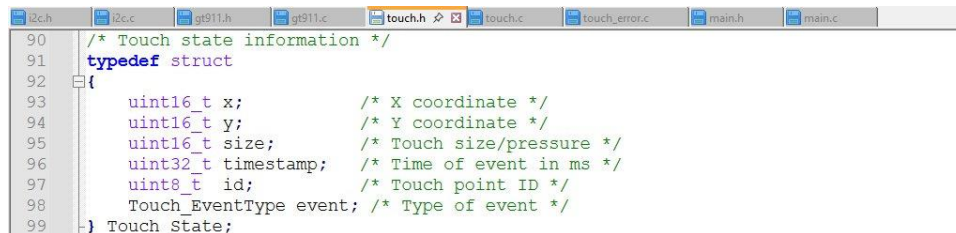
Touch Events

Data Structure of Touch Events

The GT911 stores touch event data in its memory, starting at address 0x814E. The structure of the touch data includes the following key fields:

- **Status Byte:** This byte indicates the number of touch points detected and the status of the touch data.
- **Touch Point Data:** For each detected touch point, the data includes:
 - **X-coordinate (2 bytes):** The horizontal position of the touch point.
 - **Y-coordinate (2 bytes):** The vertical position of the touch point.
 - **Touch ID:** An identifier for the touch point, allowing tracking of individual touches.
 - **Touch Event:** The type of event, such as touch down, lift off, or move.

Each touch point's data is arranged sequentially in the memory block.



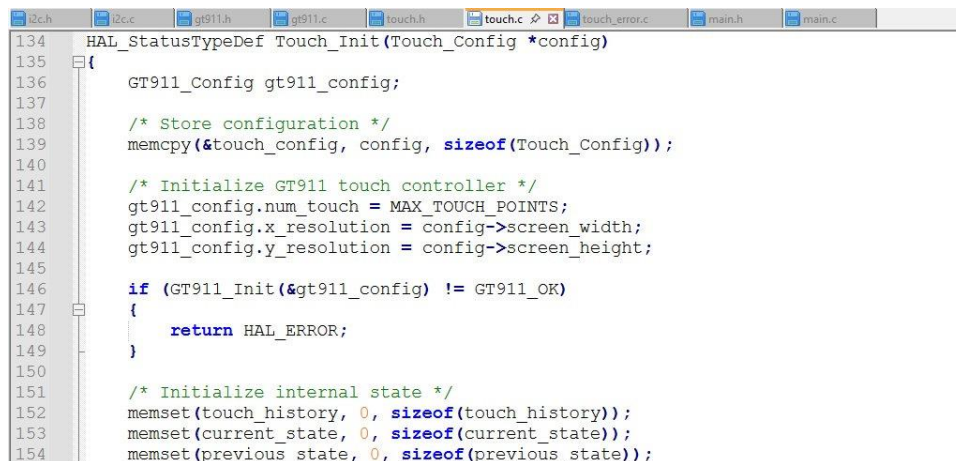
```

90  /* Touch state information */
91  typedef struct
92  {
93      uint16_t x;          /* X coordinate */
94      uint16_t y;          /* Y coordinate */
95      uint16_t size;       /* Touch size/pressure */
96      uint32_t timestamp; /* Time of event in ms */
97      uint8_t id;          /* Touch point ID */
98      Touch_EventType event; /* Type of event */
99  } Touch_State;
  
```

Figure 9: Touch Data Structure

Touch Initialization

Initialization of the touch interface provides some of the configuration data to be stored in the GT911 configuration structure. The maximum number of touch points and the resolution of the touch panel are the data stored in the config structure. Then the history, current state, and previous state arrays are set to zero.



```

134 HAL_StatusTypeDef Touch_Init(Touch_Config *config)
135 {
136     GT911_Config gt911_config;
137
138     /* Store configuration */
139     memcpy(&touch_config, config, sizeof(Touch_Config));
140
141     /* Initialize GT911 touch controller */
142     gt911_config.num_touch = MAX_TOUCH_POINTS;
143     gt911_config.x_resolution = config->screen_width;
144     gt911_config.y_resolution = config->screen_height;
145
146     if (GT911_Init(&gt911_config) != GT911_OK)
147     {
148         return HAL_ERROR;
149     }
150
151     /* Initialize internal state */
152     memset(touch_history, 0, sizeof(touch_history));
153     memset(current_state, 0, sizeof(current_state));
154     memset(previous_state, 0, sizeof(previous_state));
  
```

Figure 10: Touch Interface Initialization

Reading Touch Data

Check for interrupts if the INT pin is configured, as the interrupt signal indicates new touch data is available. To simplify the code for this application note, polling is used to check for updates. Read the status byte at address 0x814E first to determine if new touch data is available. This byte also specifies the number of touch points (ranging from 0 to 5).

```

159 HAL_StatusTypeDef Touch_Process(void)
160 {
161     /* Process power management first */
162     if (Touch_ProcessPower() != HAL_OK)
163     {
164         Touch_HandleError(TOUCH_ERR_POWER_FAIL);
165         return HAL_ERROR;
166     }
167
168     /* Only process touch if we're in active or idle state */
169     if (current_power_state > TOUCH_POWER_IDLE)
170     {
171         return HAL_OK;
172     }
173
174     GT911_TouchPoint points[MAX_TOUCH_POINTS];
175     uint8_t touch_count;
176
177     /* Read touch points from GT911 */
178     if (GT911_ReadTouchPoints(points, &touch_count) != GT911_OK)
179     {
180         Touch_HandleError(TOUCH_ERR_I2C_FAIL);
181         return HAL_ERROR;
182     }
183
184     /* Save previous state */
185     memcpy(previous_state, current_state, sizeof(current_state));
  
```

Figure 11: Reading the Touch Events from Touch_Process()

If the status byte indicates new data, the subsequent touch point information is read from the GT911's memory. Each touch point occupies a fixed number of bytes, and the data is extracted for all active touch points. After reading the touch data, the status byte must be cleared by writing 0x00 back to address 0x814E. This step notifies the controller that the data has been processed and prevents duplicate readings. This is handled from **Touch_Process()** by calling **Process_TouchEvents()**.

```

192
193     /* Process touch events */
194     Process_TouchEvents(points, touch_count);
195
  
```

Figure 12: The Call to Process_TouchEvents()

Processing Touch Events

The touch data is then retrieved, and it must be processed to get the information for the application. As a first step the touch ID should be retrieved. The touch ID helps differentiate between multiple touch points, allowing the system to track individual touches as they move or lift off. Then the touch event must be interpreted.

```

431 static void Process_TouchEvents(GT911_TouchPoint *points, uint8_t count)
432 {
433     uint32_t current_time = HAL_GetTick();
434
435     /* Process each touch point */
436     for (int i = 0; i < count; i++)
437     {
438         Touch_State *state = &current_state[i];
439
440         /* Get raw coordinates */
441         uint16_t raw_x = points[i].x;
442         uint16_t raw_y = points[i].y;
443

```

Figure 13: Retrieved the Raw Touch Data

Mapping the coordinates is the next step in the process. The raw X and Y coordinates are scaled to the resolution of the touch panel (e.g., 480x800) to align with the application's display requirements.

```

444     /* Apply calibration */
445     Apply_Calibration(&raw_x, &raw_y);
446
447     /* Update state with calibrated coordinates */
448     state->x = raw_x;
449     state->y = raw_y;
450
451     state->size = points[i].size;
452     state->id = points[i].track_id;
453     state->timestamp = current_time;

```

Figure 14: Mapping the Coordinates

The event type for each touch point is analyzed to determine the action. For example:

- Touch Down: A new touch point has been detected.
- Touch Move: An existing touch point has changed its position.
- Touch Lift Off: A touch point has been released.

```

455     /* Determine event type */
456     if (previous_state[i].event == TOUCH_EVENT_NONE)
457     {
458         state->event = TOUCH_EVENT_PRESS;
459     }
460     else if (state->x != previous_state[i].x || state->y != previous_state[i].y)
461     {
462         state->event = TOUCH_EVENT_MOVE;
463     }
464     else if (current_time - previous_state[i].timestamp >= touch_config.long_pre
465     {
466         state->event = TOUCH_EVENT_LONG_PRESS;
467     }

```

Figure 15: Touch Event Type

The release or lift off event is processed separately.


```

480     /* Check for released touches */
481     for (int i = count; i < MAX_TOUCH_POINTS; i++)
482     {
483         if (previous_state[i].event != TOUCH_EVENT_NONE)
484         {
485             Touch_State *state = &current_state[i];
486             state->event = TOUCH_EVENT_RELEASE;
487             state->timestamp = current_time;
488
489             if (event_callback != NULL)
490             {
491                 event_callback(state);
492             }

```

Figure 16: Touch Lift Off Processing

Based on the touch data, application-specific actions are executed. This might include updating a graphical user interface, triggering events, or controlling devices. The processing of the touch events is handled in the callback function located in the main.c file.

```

175     /* Touch event callback */
176     static void Touch_EventHandler(Touch_State *state)
177     {
178         switch (state->event)
179         {
180             case TOUCH_EVENT_PRESS:
181                 printf("Touch Press: ID=%d, X=%d, Y=%d, Size=%d\n",
182                     state->id, state->x, state->y, state->size);
183                 break;
184
185             case TOUCH_EVENT_RELEASE:
186                 printf("Touch Release: ID=%d\n", state->id);
187                 break;
188
189             case TOUCH_EVENT_MOVE:
190                 printf("Touch Move: ID=%d, X=%d, Y=%d\n",
191                     state->id, state->x, state->y);
192                 break;
193
194             case TOUCH_EVENT_LONG_PRESS:
195                 printf("Long Press: ID=%d, X=%d, Y=%d\n",
196                     state->id, state->x, state->y);
197                 break;
198
199             default:
200                 break;
201         }
202     }
203

```

Figure 17: Touch Event Processing in the Callback Function

Optimization Considerations

Efficient handling of touch events is critical to maintaining responsiveness in the application. Using the INT pin to signal new data reduces the need for constant polling, conserving processing resources. Then retrieve all touch point data in a single I2C transaction to minimize communication overhead. Implement debouncing logic to filter out noise and spurious touch events, improving accuracy. There is some debouncing logic already part of the GT911 controller. This is where the filtering registers are utilized. When possible, ensure the touch data is processed within the required refresh period (e.g., 5 milliseconds) to maintain a smooth user experience.

Gesture Recognition

The GT911 uses its integrated gesture engine to analyze touch data patterns in real-time. By tracking the movement and position of one or more touch points, the controller identifies predefined gestures. These gestures are mapped to specific codes, which are then stored in a dedicated register for retrieval by the host system.

The commonly supported gestures include swipe gestures (up, down, left, and right), zoom gestures (pinch-in and pinch-out), long press, and double tap. The gesture recognition process reduces the computational burden on the host system by offloading pattern analysis to the GT911.

Configuring the GT911 for Gesture Recognition

Gesture detection is enabled by writing to the gesture enable register, typically located in the configuration block of the GT911's memory. The specific address and value depend on the desired gesture set. Adjust gesture-specific parameters such as sensitivity, speed thresholds, and movement ranges. These parameters ensure accurate recognition based on the application's requirements and display size.

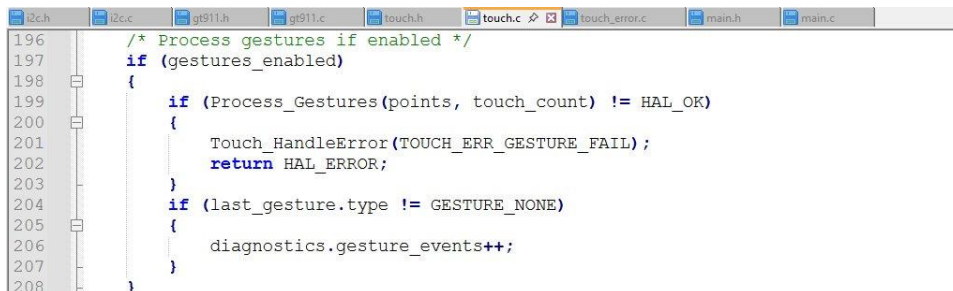
After updating the gesture settings, write 0x01 to the control register at address 0x8040 to save the configuration. Transition the GT911 to application mode by writing 0x80 to the same register.

Verify that gestures are detected correctly by reading gesture event data from the dedicated register after performing gestures on the touch panel.

Processing Gesture Events

The GT911 stores gesture event information at address 0x814F. This register contains the gesture code, which corresponds to the detected gesture.

The code breaks up the gesture processing into several functions. If the gesture feature of the GT911 is enabled in the firmware, then the initial processing is performed in the **Touch_Process()** function.



```

196  /* Process gestures if enabled */
197  if (gestures_enabled)
198  {
199      if (Process_Gestures(points, touch_count) != HAL_OK)
200      {
201          Touch_HandleError(TOUCH_ERR_GESTURE_FAIL);
202          return HAL_ERROR;
203      }
204      if (last_gesture.type != GESTURE_NONE)
205      {
206          diagnostics.gesture_events++;
207      }
208  }
  
```

Figure 18: Capturing the Gesture Event

Once the gesture event has been captured, the event is processed in the `Process_Gestures()` function. The gesture code is retrieved from the gesture register. This code uniquely identifies the type of gesture detected (e.g., swipe left, zoom in).

```

502 static void Process_Gestures(GT911_TouchPoint *points, uint8_t count)
503 {
504     /* Reset last gesture */
505     last_gesture.type = GESTURE_NONE;
506
507     if (count == 1)
508     {
509         /* Check for swipe gestures */
510         Touch_State *start = &touch_history[(history_index - 1) % TOUCH_HISTORY_SIZE];
511         Touch_State *end = &current_state[0];
512         Detect_Swipe(start, end);
513     }
514     else if (count == 2)
515     {
516         /* Check for pinch and rotation gestures */
517         Detect_Pinch(points, count);
518         Detect_Rotation(points, count);
519     }
520
521     /* Call gesture callback if a gesture was detected */
522     if (last_gesture.type != GESTURE_NONE && gesture_callback != NULL)
523     {
524         gesture_callback(&last_gesture);
525     }
526 }

```

Figure 19: `Process_Gestures()`

Additional processing is called to handle the pinch, rotation, and swipe gestures.

```

550 static void Detect_Pinch(GT911_TouchPoint *points, uint8_t count)
551 {
552     if (count != 2) return;
553
554     float current_distance = Calculate_Distance(points[0].x, points[0].y,
555                                                points[1].x, points[1].y);
556     float previous_distance = Calculate_Distance(previous_state[0].x, previous_state[0].y,
557                                                previous_state[1].x, previous_state[1].y);
558
559     float diff = current_distance - previous_distance;
560     if (fabs(diff) >= touch_config.swipe_threshold)
561     {
562         last_gesture.type = (diff > 0) ? GESTURE_PINCH_OUT : GESTURE_PINCH_IN;
563         last_gesture.magnitude = fabs(diff);
564     }
565 }

```

Figure 20: `Pinch`

```

567 static void Detect_Rotation(GT911_TouchPoint *points, uint8_t count)
568 {
569     if (count != 2) return;
570
571     float current_angle = Calculate_Angle(points[0].x, points[0].y,
572                                           points[1].x, points[1].y);
573     float previous_angle = Calculate_Angle(previous_state[0].x, previous_state[0].y,
574                                           previous_state[1].x, previous_state[1].y);
575
576     float angle_diff = current_angle - previous_angle;
577     if (fabs(angle_diff) >= touch_config.rotate_threshold)
578     {
579         last_gesture.type = (angle_diff > 0) ? GESTURE_ROTATE_CW : GESTURE_ROTATE_CCW;
580         last_gesture.angle = fabs(angle_diff);
581     }
582 }

```

Figure 21: Rotate

```

528 static void Detect_Swipe(Touch_State *start, Touch_State *end)
529 {
530     float dx = end->x - start->x;
531     float dy = end->y - start->y;
532     float distance = sqrtf(dx * dx + dy * dy);
533
534     if (distance >= touch_config.swipe_threshold)
535     {
536         /* Determine swipe direction */
537         if (fabs(dx) > fabs(dy))
538         {
539             last_gesture.type = (dx > 0) ? GESTURE_SWIPE_RIGHT : GESTURE_SWIPE_LEFT;
540         }
541         else
542         {
543             last_gesture.type = (dy > 0) ? GESTURE_SWIPE_DOWN : GESTURE_SWIPE_UP;
544         }
545         last_gesture.magnitude = distance;
546         last_gesture.duration = end->timestamp - start->timestamp;
547     }
548 }

```

Figure 22: Swipe

After reading the gesture code, clear the register by writing 0x00 to prevent duplicate event processing.

Now the gesture codes need to be mapped to actions. Use a lookup table or conditional logic in the application firmware to map gesture codes to specific actions. In the code presented, a gesture callback is used to call the handler function. This handler function is in main.c at the application level. It currently uses `printf()` to output a message indicating the gesture. It is left to the end user to adapt the code to their application.

```

205 static void Touch_GestureHandler(Touch_Gesture *gesture)
206 {
207     switch (gesture->type)
208     {
209         case GESTURE_SWIPE_LEFT:
210             printf("Swipe Left: Distance=%.1f, Duration=%lu ms\n",
211                 gesture->magnitude, gesture->duration);
212             break;
213
214         case GESTURE_SWIPE_RIGHT:
215             printf("Swipe Right: Distance=%.1f, Duration=%lu ms\n",
216                 gesture->magnitude, gesture->duration);
217             break;
218
219         case GESTURE_SWIPE_UP:
220             printf("Swipe Up: Distance=%.1f, Duration=%lu ms\n",
221                 gesture->magnitude, gesture->duration);
222             break;
223
224         case GESTURE_SWIPE_DOWN:
225             printf("Swipe Down: Distance=%.1f, Duration=%lu ms\n",
226                 gesture->magnitude, gesture->duration);
227             break;
228     }

```

Figure 23: Gesture Handler Part 1

```

228
229
230         case GESTURE_PINCH_IN:
231             printf("Pinch In: Magnitude=%.1f\n", gesture->magnitude);
232             break;
233
234         case GESTURE_PINCH_OUT:
235             printf("Pinch Out: Magnitude=%.1f\n", gesture->magnitude);
236             break;
237
238         case GESTURE_ROTATE_CW:
239             printf("Rotate Clockwise: Angle=%.1f degrees\n", gesture->angle);
240             break;
241
242         case GESTURE_ROTATE_CCW:
243             printf("Rotate Counter-Clockwise: Angle=%.1f degrees\n", gesture->angle);
244             break;
245
246         default:
247             break;
248     }

```

Figure 24: Gesture Handler Part 2

Gesture Mode vs. Touch Mode

The GT911 does not support gesture mode and touch mode simultaneously. When gesture mode is enabled, the touch controller focuses on detecting predefined gesture patterns, and regular touch point data (such as X and Y coordinates or multi-touch information) is not actively processed or reported. Conversely, in touch mode, the GT911 operates as a multi-touch controller, providing detailed touch data but not performing gesture recognition.

The GT911 operates either in gesture mode or touch mode based on the configuration settings. You need to decide which mode is more critical for your application.

To switch between modes, you need to reconfigure the controller and restart it appropriately. This involves writing the relevant configuration values to enable or disable gesture mode and saving the settings using the control register. Switching modes during runtime is possible but may introduce latency due to the reconfiguration process.

If your application requires both functionalities, consider designing it to toggle between modes based

on user context. For instance, gesture mode could be used for specific interfaces, while touch mode is active during detailed user interactions. Another approach is to offload gesture recognition to the application software using raw touch data from touch mode, although this requires additional processing and development effort.

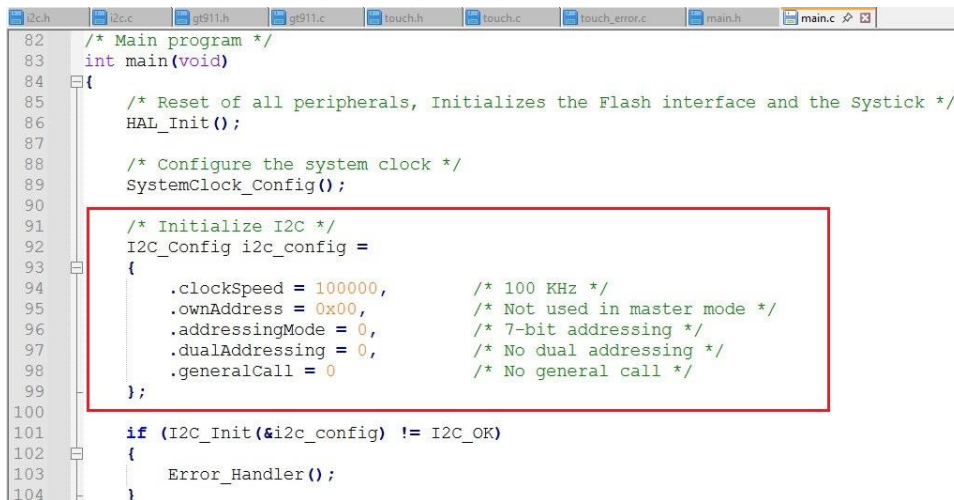
Optimizing Gesture Handling

To enhance responsiveness and accuracy, several steps can be taken. Fine-tune the sensitivity parameters during configuration to match the display size and user interaction patterns. Implement debouncing logic to avoid false positives or unintentional gesture detection. Ensure gesture data is processed promptly to deliver a seamless user experience. During development, log the gesture codes to validate detection accuracy and refine configuration settings.

Application Integration

The touch event handling can be integrated into the main application loop or through interrupt-driven mechanisms. The `Touch_EventHandler()` and `Touch_GestureHandler()` functions are implemented to process the touch events but only use `printf()` to output a message about the event. The end user will need to add functional code according to the specific application requirements.

The main application configures the I2C peripheral at the beginning of `main()`.



```

82  /* Main program */
83  int main(void)
84  {
85      /* Reset of all peripherals, Initializes the Flash interface and the Systick */
86      HAL_Init();
87
88      /* Configure the system clock */
89      SystemClock_Config();
90
91      /* Initialize I2C */
92      I2C_Config i2c_config =
93      {
94          .clockSpeed = 100000,      /* 100 KHz */
95          .ownAddress = 0x00,      /* Not used in master mode */
96          .addressingMode = 0,     /* 7-bit addressing */
97          .dualAddressing = 0,     /* No dual addressing */
98          .generalCall = 0        /* No general call */
99      };
100
101      if (I2C_Init(&i2c_config) != I2C_OK)
102      {
103          Error_Handler();
104      }
  
```

Figure 25: I2C Config in Main

Then the touch interface can be initialized.

```

106  /* Initialize touch interface */
107  Touch_Config touch_config =
108  {
109      .screen_width = 800,          /* Set your screen width */
110      .screen_height = 480,        /* Set your screen height */
111      .long_press_time = 1000,     /* 1 second for long press */
112      .swipe_threshold = 50,       /* 50 pixels minimum for swipe */
113      .rotate_threshold = 30      /* 30 degrees minimum for rotation */
114  };
115
116  if (Touch_Init(&touch_config) != HAL_OK)
117  {
118      Error_Handler();
119  }

```

Figure 26: Touch Initialization in Main

Next, the callback events need to register the event handlers.

```

135  /* Register callbacks */
136  Touch_RegisterEventCallback(Touch_EventHandler);
137  Touch_RegisterGestureCallback(Touch_GestureHandler);
138  Touch_RegisterErrorCallback(Touch_ErrorHandler);
139

```

Figure 27: Event Callback Handlers

The last step in getting the touch interface application code running is the main while loop. All the other code developed for the application makes the final while loop simple to implement. Here is what it looks like:

```

158  /* Main loop */
159  while (1)
160  {
161      /* Process touch input */
162      if (Touch_Process() != HAL_OK)
163      {
164          /* Handle error - maybe reset touch controller */
165          printf("Touch processing error\n");
166          HAL_Delay(100);
167          continue;
168      }
169
170      /* Add other application processing here */
171      HAL_Delay(10); /* Small delay to prevent overwhelming the touch controller */
172  }
173

```

Figure 28: The Main While Loop

Conclusion

Using the I2C4 peripheral on the STM32H747I-DISCO, the GT911 touch controller can be successfully configured to detect touches on the Focus LCDs E43GB-I-MW405-C display. This application note provided the steps required to program the STM32H747I-DISCO to control the GT911 touch controller. It walked through the configuration of the I2C peripheral, the low-level device driver for the GT911, and the higher-level touch interface code used by the application code.

Recommended Next Steps

In the source code that can be provided by Focus LCDs, power management, error handling, and diagnostic functions have been included. This code is currently limited to basic functionality and more robust functions are left to the end user. It is included as a template on how to implement the functions.

Basic touch panel calibration routines have been implemented in code but are not used in the application layer. It is up to the end user to add this functionality into the main application

Additional Information

The source code for the GT911 Capacitive Touch Controller can be acquired by contacting Focus LCDs.

LCD Handling Precautions

- Do not store the TFT-LCD module in direct sunlight, best stored in a dark place
- Do not leave it exposed to high temperature and high humidity for a long period of time
- Recommended temperature range is 0 to 35 °C, relative humidity should be less than 70%
- Stored modules away from condensation as formation of dewdrops may cause an abnormal operation or failure of the module.
- Protect the module from static discharge
- Do not press or scratch the surface and protect it from physical shock or any force

Disclaimer

Buyers and others who are developing systems that incorporate FocusLCDs products (collectively, “Designers”) understand and agree that Designers remain responsible for using their independent analysis, evaluation, and judgment in designing their applications and that Designers have full and exclusive responsibility to assure the safety of Designers’ applications and compliance of their applications (and of all FocusLCDs products used in or for Designers’ applications) with all applicable regulations, laws, and other applicable requirements.

Designer represents that, with respect to their applications, Designer has all the necessary expertise to create and implement safeguards that:

- (1) anticipate dangerous consequences of failures
- (2) monitor failures and their consequences, and
- (3) lessen the likelihood of failures that might cause harm and take appropriate actions.

The designer agrees that prior to using or distributing any applications that include FocusLCDs products, the Designer will thoroughly test such applications and the functionality of such FocusLCDs products as used in such applications.

Revision History

Revision	Notes	Date
1.0.0	Initial Version	11/5/2024