Application Note

BRT_AN_090

EVE Working with Capacitive Touch Screens

Version 1.0

Issue Date: 01-11-2023

Bridgetek Pte Ltd (BRTChip)
1 Tai Seng Avenue, Tower A, #03-05, Singapore 536464
Tel: +65 6547 4827
Web Site: http://www.brtchip.com

Table of Contents

1 Introduction

A capacitive touch screen controller IC is often used to process the touch commands on the Capacitive Touch Panel Module (CTPM) and sends information to the graphics processor or MCU through a serial interface, usually the I^2C interface. Although different controller IC manufacturers may use the same I^2C interface, the programming sequence and register mappings are usually different.

This document covers an introduction of the Capacitive Touch Screen Engine (CTSE) of the BT815/7 and provides information on working with different Capacitive Touch Panel modules for BT815/7.

The information provide in this document is correct at the time of writing and is provided for guidance only. Bridgetek recommends that customers test the intended touch controller at their prototyping stage as specifications of third-party ICs are subject to change by the manufacturer beyond the control of Bridgetek.

2 Capacitive Touch Screen Engine (CTSE)

The Capacitive Touch Screen Engine (CTSE) of the BT815/7 communicates with the external Capacitive Touch Panel Module (CTPM) through an I^2C interface where BT815/7 is the I^2C master.

The CTPM asserts its interrupt line whenever a touch is detected. Upon detecting this interrupt, the BT815/7 reads the touch data through the I^2C interface. Up to 5 touch points can be detected and stored in the BT815/7 register at the same time.

The BT815/7 has a built-in ROM to support a list of touch controllers. When the touch controller of the selected CTPM is not in the direct support list, the CTPM can be supported by either creating a custom firmware using the Custom Touch feature in the EVE Asset Builder (EAB) program or using the Host Driven Multi-Touch if the MCU can provide touch inputs.

The MCU controls the CTSE operation mode by writing to the register REG_CTOUCH_MODE. This register and all subsequent register definitions can be found in Application Note BRT_AN_033.

REG_CTOUCH_MODE	Mode	Description
0	Off	Acquisition stopped
1	Reserved	Reserved
2	Reserved	Reserved
3	On	Acquisition Started

Table 1 - Capacitive Touch Controller Operating Modes

The BT815/7 supports compatibility mode and extended mode. After reset or boot up, the CTSE operates in compatibility mode and only one touch point is detected. In extended mode, it can detect up to five touch points simultaneously.

2.1 Compatibility Mode

The CTSE is in compatibility mode when the REG_CTOUCH_EXTENDED is set to 1b'1. The CTSE reads the X and Y coordinates from the CTPM and writes to register REG_CTOUCH_RAW_XY. If the screen is not being touched, both the register fields in REG_CTOUCH_RAW_XY read 65535 (FFFFh). These touch values are transformed into the screen coordinates using the matrix in registers REG_TOUCH_TRANFORM_A~F.

The post-transform coordinates are available in register REG_TOUCH_SCREEN_XY. If the screen is not being touched, both the register fields in REG_TOUCH_SCREEN_XY read -32768 (8000h). The values for REG_TOUCH_TRANSFORM_A~F may be computed using an on-screen calibration process.

If the screen is being touched, the screen coordinates are looked up in the screen's tag buffer, delivering a final 8-bit tag value, in register REG_TOUCH_TAG. Because the tag lookup takes a full frame, and touch coordinates change continuously, the original (X, Y) used for the tag lookup is also available in register REG_TOUCH_TAG_XY.

2.2 Extended Mode

Setting REG_CTOUCH_EXTENDED to 1b'0 enables extended mode. In extended mode, a new set of readout registers are available, allowing gestures and up to five touches to be read simultaneously. There are two classes of registers: Control Registers and Status Registers. Control registers are written by the MCU and status registers can be read out by the MCU and the BT815/7's hardware tag system.

The five touch coordinates are packed in REG_CTOUCH_TOUCH_XY, REG_CTOUCH_TOUCH1_XY, REG_CTOUCH_TOUCH2_XY, REG_CTOUCH_TOUCH3_XY, REG_CTOUCH_TOUCH4_X, and REG_CTOUCH_TOUCH4_Y.

Coordinates stored in these registers are signed 16-bit values, and therefore have a range of -32768 (8000h) to 32767 (7FFFh). The no-touch condition is indicated when both register fields in these registers read -32768 (8000h). These coordinates are already transformed into screen coordinates based on the raw data read from the CTPM, using the matrix in registers REG_TOUCH_TRANSFORM_A~F. To obtain raw (X, Y) coordinates read from the CTPM, the user sets the REG_TOUCH_TRANSFORM_A~F registers to the identity matrix.

The BT815/7 tag mechanism is implemented by hardware. A tag is a value assigned by the user that is attached to the following graphics object drawn on the screen. When the graphics object attached to the tag value is being touched, the tag mechanism looks up the tag value in the screen's tag buffer based on the screen coordinates of the graphics object. Up to 5 tags can be looked up.

In touch extended mode, the INT_TOUCH bit in REG_INT_FLAG register will not be set upon touch down event. It is recommended to use INT_CONV_COMPLETE instead.

2.3 Capacitive Touch Configuration

On a capacitive touch system some users may need to adjust the CTPM default values, such as the registers affecting touch sensitivity. To do this the following sequence shall be executed once after chip reset:

- Hold the touch engine in reset (set REG_CPURESET = 2)
- Write the CTPM configure register address and value to the BT815/7 designated memory location
- up to 10 register address/value can be added
- Release the touch engine reset (set REG_CPURESET = 0)

Adjusting the CTPM default values can also be done by writing a custom touch firmware. Bridgetek recommends customers to use the custom touch firmware as describe in Section 5.2.

3 Hardware Interface

3.1 Voltage levels

Some of the supported CTPM are intended for low-voltage application, where their I/O voltage is either 2.5V or 1.8V. The BT815/7 can support these CTPM by connecting the VCCIO2 to the same I/O voltage as these CTPM. As the VCCIO2 are also supplied to the RGB signals, the I/O voltage for the RGB signals are also affected. Hence, the LCD selected must be compatible to this I/O voltage.

In the event where the I/O voltage of the LCD module and the touch controller is not the same, a level shifter would be required to translate the levels between the touch controller and BT815/7.

3.2 I²C Interface

3.2.1 I²C Target Address

The BT815/7 supports I2C target address that is seven bits long. The default target address is 0x38 that is for the FocalTech controller IC. This target address can be changed via the REG_TOUCH_CONFIG register to support the other manufacturer in the direct support list. Some of the touch controllers in the direct support list have a configurable/programmable I²C addresses, the address in REG_TOUCH_CONFIG can be set to match the address configured in the touch controller.

3.2.2 I2C Bus Speed

The I²C interface of the BT815/7 is a I²C master bus where the BT815/7 supplies the clock to the touch controller. On the BT815, the I²C Serial Clock (SCL) is about 380KHz when using the touch controller from the direct support list and is about 300KHz when using the custom firmware. On BT817, the SCL is about 300KHz in both cases.

3.2.3 I²C Clock Stretching

The BT815 does not support clock stretching with native touch firmware. Only BT817 natively supports clock stretching where the touch controller pauses a transaction by holding the SCL line LOW. The transaction cannot continue until the line is released HIGH again. This allows the touch controller more time to prepare the data to be transmit. The custom touch firmware compiled with EVE Asset Builder (EAB) supports clock stretching for both BT815/7.

3.3 Hardware Connection

Figure 1 shows a typical connection of the Capacitive Touch Panel Module connection to the BT815/7. The CTP_SCL and CTP_SDA lines are connected to the SCL and SDA lines of the CTPM respectively. As these lines are open drain, they require to connect to the VCCIO2 via two pull-up resistors (Rp). The recommended pull-up resistor (Rp) values depend on the system implementation, but a value between 1KΩ and 3KΩ can be used for prototyping. The wake signal, if present, is connected to the VCCIO2 via a pull-up resistor (Rwake). In some cases, the interrupt line of the CTPM is open drain and require a pull-up resistor (Rint). Please refer to the datasheet of the CTPM to confirm the pin type and recommendations. The Rs series resistors, typically between 10Ω to 33Ω, are present to improve the EMI performance.

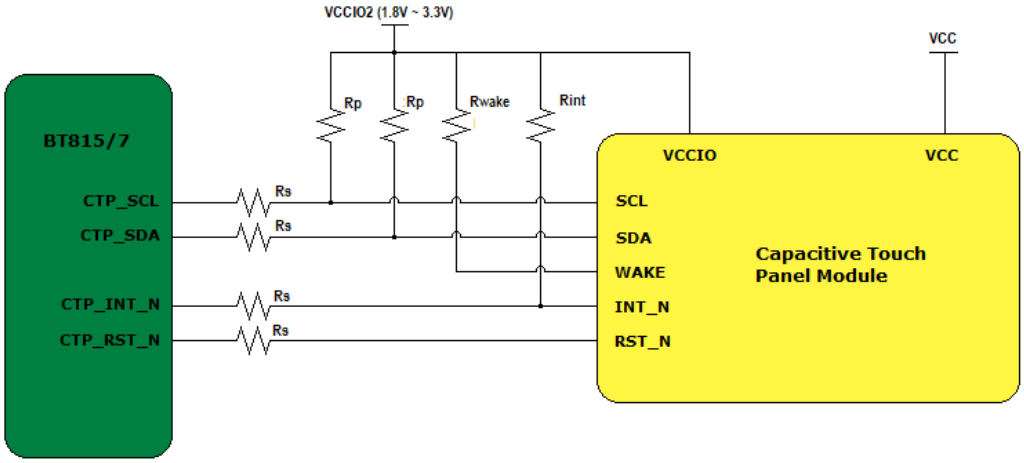

Figure 1 - Capacitive Touch Panel Module Connection

The touch controller IC is often integrated into either the display panel itself or within the separate ribbon cable attached to the panel. The number of connections in the ribbon cable and the order of the signal may vary. Please consult the datasheet of the CTPM to confirm the pinout.

4 Compatible touch controller (built-in support)

The built-in ROM holds the programming sequence and register mappings of the touch controller in the direct support list.

Table 2 shows the direct support list for BT815/7

Brand	Models
FocalTech	FT5x06 series: FT5206, FT5306, FT5406
	FT5x16 series: FT5316
	FT5x26 series: FT5426
	FT5x36 series: FT5336, FT 5436
	FT5x46 series: FT5346, FT5446
	FT6xx6 series: FT6306, FT6236, FT6336U
Goodix	GT9xx series: GT911, GT9271, GT928
HYCON	HY46xx series*: HY4614B, HY4635

Table 2 – Direct Support List for B815/7

Key: Asterisk *: The HYCON HY46xx touch controller is similar to the FocalTech touch controller. As such, the native touch firmware (built-in support) for BT817 can support the HYCON HY46xx touch controller. However, as BT815 does not support clock stretching with the native touch firmware, the BT815 can only support the HYCON HY46xx touch controller with the custom firmware as described in Section 5.

Note: The particular models listed in this table indicates samples for each family tested compatible with BT815/7. Other models not listed are expected to work as long as the protocol is compatible.

The BT815/7 uses the I^2C address value set in the register REG_TOUCH_CONFIG to differentiate between FocalTech and Goodix touch controllers. For FocalTech touch controller IC, the I^2C address must be set in the range of 0x38 ~ 0x3F, while the I^2C address for the Goodix touch controller IC must be set as 0x5D. The touch engine of the BT815/7 is required to perform a reset after the I^2C address is changed.

Below is the programming sequence to change the I^2C address:

```
Write REG_CPURESET = 2
Write REG_TOUCH_CONFIG = 0x5D0
Write REG_CPURESET = 0
```

5 Compatible Touch Controller (Custom Touch)

To support touch controller that is not in the direct support list, user can write a custom firmware and use the Custom Touch Compiler in the EVE Asset Builder (EAB) program to create a custom firmware. The custom firmware is a piece of code that can be execute by the EVE co-processor to access the touch controller.

The custom touch compiler is a small program in tiny C-like language. Resembles C in syntax, it allows:

- functions of "void" and "int" type
- 'if…else' constructs, 'while' and 'do…while' loops
- up to 3 local variables of "int" type only
- integer arithmetic. ('/' and '%' not supported)

There are three user-defined functions:

- i2c_addr() this function returns the I^2C address of the CTPM
- setup() this function performs a one-time initialization of the CTPM
- loop() this function continuously polls the CTPM and update the touch sensing registers.

In addition, a number of built-in functions are available for I/O in the user's code.

- void delay_ms(n) delay in n milliseconds
- void delay_us(n) delay in n microseconds
- void i2c_regwr(u, v) write value 'v' to register 'u'
- void i2c_startread(u) start a read transaction from register 'u'
- int i2c_read16le() read 16-bits from I^2C, return value in little endian
- int i2c_read16be() read 16-bits from I^2C, return value in big endian
- int i2c_read8() read 8-bits from I^2C
- void i2c_stop() end the I^2C transaction
- void report_touch(id, x, y) set the touch sensing registers, id: 0 – 4, (x, y): 15-bits values
- int getINT() get the interrupt status
- void setINT(u) When 'u'='0': sets CTP_INT_N pin to drive low; When 'u'='1': sets CTP_INT_N pin to Hi-Z (input)
- void setWAKE(u) When 'u' = '0': sets CTP_RST_N pin to low; When 'u' = '1': sets CTP_RST_N pin to high
- int getCYA() get the register REG_TOUCH_CONFIG value

5.1 Sample Codes

Below are some sample codes for different controllers. These sample codes are available in Eve Asset Builder version 2.10 or later.

Sample Code to support Sitronix ST1633i touch controller

```
/***
Custom Touch Firmware for ST1633i Touch Controller
Please use the EVE Asset Builder – Custom Touch to compile the below source code
***/

// slave address of ST1633i Touch Controller
int i2c_addr()
{
      return 0x55;
}

// set the reset pin to low, wait for 5ms
// set the reset pin to high.
// Wait for another 65ms before starting any I2C transaction
void setup()
{
```

```
	setWAKE(0);
	delay_ms(5);
	setWAKE(1);
	delay_ms(65);
}

void loop()
{
	// Wait for the interrupt to go low
	while (getINT() == 1)
		;

	// Start I2C read transaction with offset 0x12
	i2c_startread(0x12);

	// Declare local variables
	int event_xy = 0
	int xy = 0;
	int id = 0;

	// Support up to 5 touches
	while (id < 5)
	{
		// Read in the data from the touch controller
		event_xy = i2c_read8();
		xy = i2c_read16be();
		i2c_read8();

		// Check if the event bit is set
		if ((event_xy & 0x80) == 0x80)
		{
			// Report the touch point
			report_touch(id,
				(((event_xy<<4)&0x700)|((xy>>8)&0x00ff)),
				(((event_xy<< 8) & 0x700)|(xy&0xff)));
		}
		id = id + 1;
	}
	// Stop I2C transaction
	i2c_stop();

	// Wait for the interrupt to go high
	while (getINT() == 0)
		;
}
```

Sample code to support EETI EXC80W46 touch controller.
For Multi-Touch Report Format (without width and height, 16K X/Y resolution)

```
/***
Custom Touch Firmware for EETI EXC80W46 Touch Controller
For Multi-Touch Report Format (without width and height, 16K X/Y resolution)
Please use the EVE Asset Builder – Custom Touch to compile the below source code
***/

//slave address of EETI EXC80W46
int i2c_addr()
{
	return 0x2A;
}

// set the reset pin to low, wait for 10ms
// set the reset pin to high.
// Wait for another 300ms before starting any I2C transaction
void setup()
{
	setWAKE(0);
	delay_ms(10);
```

```
	setWAKE(1);
	delay_ms(300);
}

// additional sub-routine to report touch points
void rtouch()
{
	// Read in the data from the touch controller
	int state_ID = i2c_read16be();
	int x = i2c_read16le();
	int y = i2c_read16le();

	// state ID is set and touch point less than 5
	if (((state_ID & 0x0100) == 0x0100) && ((state_ID & 0x00ff) < 5))
	{
		// Report the touch point
		report_touch( (state_ID& 0xff), x, y);
	}

	i2c_read16be();
	i2c_read16be();
}

void loop()
{
	// Wait for interrupt pin to go low
	while (getINT() == 1)
		;

	// While the interrupt is low. The interrupt will go goes high when there is no data in controller
	while (getINT() == 0)
	{
		// Start I2C read transaction at offset 0x00
		i2c_startread(0x00);

		// Read in th length of the report
		int len = i2c_read16le();

		len = len - 2;

		// Report ID = 0x0018
		if (i2c_read8() == 0x0018)
		{
			// Read in number of touches
			int n_touches = i2c_read8();

			// Calculate the remaining data
			len = len - (2 + (n_touches * 10));

			// Report Touch points
			while (n_touches> 0)
			{
				rtouch();
				n_touches = n_touches - 1;
			}

			// len > 0 indicates that there are still data in the touch controller
			while (len> 0)
			{
				// dummy read to clear the data in touch controller
				i2c_read8();
				len = len – 1;
			}
		}
		// Other Report ID. Still need to clear the data in controller
		else
		{
			while (len> 0)
```

```
                {
                    i2c_read8();
                    len = len - 1;
                }
            }
        }
        // Interrupt has gone high Stop I2C transaction
        i2c_stop();
}
```

Sample code to support HYCON HY46xx touch controller

```
/***
Custom Touch Firmware for HYCON HY46xx Touch Controller
Please use the EVE Asset Builder – Custom Touch to compile the below source code
***/

// slave address for HYCON HY46xx
int i2c_addr()
{
    return 0x38;
}

// set the reset pin to low, wait for 30ms
// set the reset pin to high.
// Wait for another 300ms before starting any I2C transaction
void setup()
{
    setWAKE(0);
    delay_ms(30);
    setWAKE(1);
    delay_ms(300);
}

void loop()
{
    // Declare local variables
    int x, id_y;

    // Wait for interrupt pin to go low
    while (getINT() == 1)
        ;
    // Start I2C read transaction with offset 0x02
    i2c_startread(0x02);

    // Read in the number of touches
    int n_touches = i2c_read8() & 0x0F;

    // Touch points detected. Report the touch points
    while (n_touches != 0)
    {
        x = i2c_read16be();
        id_y = i2c_read16be();

        report_touch(((id_y >> 12) & 0x000F), (x & 0x0FFF), (id_y & 0x0FFF));

        i2c_read8();
        i2c_read8();

        n_touches = n_touches - 1;
    }

    // Stop I2C transaction
    i2c_stop();

    // Wait for the interrupt pin to go high
    while (getINT() == 0)
        ;
```

```
}
```

Sample code to support ILItek ILI2511 touch controller

```
/***
Custom Touch Firmware for ILITek ILI2511 Touch Controller
Please use the EVE Asset Builder – Custom Touch to compile the below source code
***/

// slave address of ILI2511
int i2c_addr()
{
	return 0x41;
}

// set the reset pin to low, wait for 30ms
// set the reset pin to high.
// Wait for another 300ms before starting any I2C transaction
void setup()
{
	setWAKE(0);
	delay_ms(30);
	setWAKE(1);
	delay_ms(300);
}

void loop()
{
	// wait for INT to go low
	while (getINT() == 1)
		;
	// Start I2C read transaction at offset 0x10
	i2c_startread(0x10);
	i2c_read8();

	// Declare local variables
	int id = 0;
	int event_x;
	int y;

	// support up to 5 touches
	while (id < 5)
	{
		event_x = i2c_read16be();
		y = i2c_read16be();
		i2c_read8();

		// Touch detected. Report the touch points
		if ((event_x & 0x8000) == 0x8000)
		{
			report_touch(id, event_x & 0x7fff, y & 0x7fff);
		}
		id = id + 1;
	}

	y = 0;
	// read the remaining bytes of a report
	while (y < (64-id*5-1))
	{
		i2c_read8();
```

```
            y = y + 1;
        }

        // Stop I2C transaction
        i2c_stop();

        // wait for INT to go high
        while (getINT() == 0)
            ;
}
```

Sample code to support FocalTech touch controller

```
int i2c_addr()
{
    return 0x38;
}

void setup()
{
    setWAKE(0);
    delay_ms(10);
    setWAKE(1);
    delay_ms(300);
}

void rtouch()
{
    int x = i2c_read16be() & 0xfff;
    int id_y = i2c_read16be();
    i2c_read8();
    i2c_read8();

   report_touch(id_y >> 12, x, id_y & 0xfff);
}

void loop()
{
   while (getINT() == 1) // wait for INT to go low
      ;
   i2c_startread(0x02);

   int n_touches = i2c_read8();
   while (n_touches != 0)
```

```
  {
    rtouch();
    n_touches = n_touches - 1;
  }
  i2c_stop();
  while (getINT() == 0) // wait for INT to go high
    ;
}
```

5.2 Adjusting the CTPM default values using Custom Firmware

The default values of the CTPM can be adjusted using the custom touch firmware. This is done by using the i2c_regwr(u, v) command to adjust the values in the CTPM.

Example

To change the register to reduce the touch sensitivity in FocalTech Touch Panel, the command, i2c_regwr(0x80, 70) is written in the void setup() function.

```
int i2c_addr()
{
  return 0x38;
}

void setup()
{
  setWAKE(0);
  delay_ms(10);
  setWAKE(1);
  delay_ms(300);
  i2c_regwr(0x80, 70);
}

void rtouch()
{
  int x = i2c_read16be() & 0xfff;
  int id_y = i2c_read16be();
  i2c_read8();
  i2c_read8();
  report_touch(id_y >> 12, x, id_y & 0xfff);
}
```

```
void loop()
{
   while (getINT() == 1) // wait for INT to go low
      ;
   i2c_startread(0x02);

   int n_touches = i2c_read8();
   while (n_touches != 0)
   {
      rtouch();
      n_touches = n_touches - 1;
   }
   i2c_stop();
   while (getINT() == 0) // wait for INT to go high
      ;
}
```

5.3 Compiling the code

To compile the code into a loadable firmware image for the touch functionality, install and launch the EVE Asset Builder (EAB) v2.10.0 or later.

1. Click on the Custom Touch button to go to the Custom Touch page as shown in Figure 2.

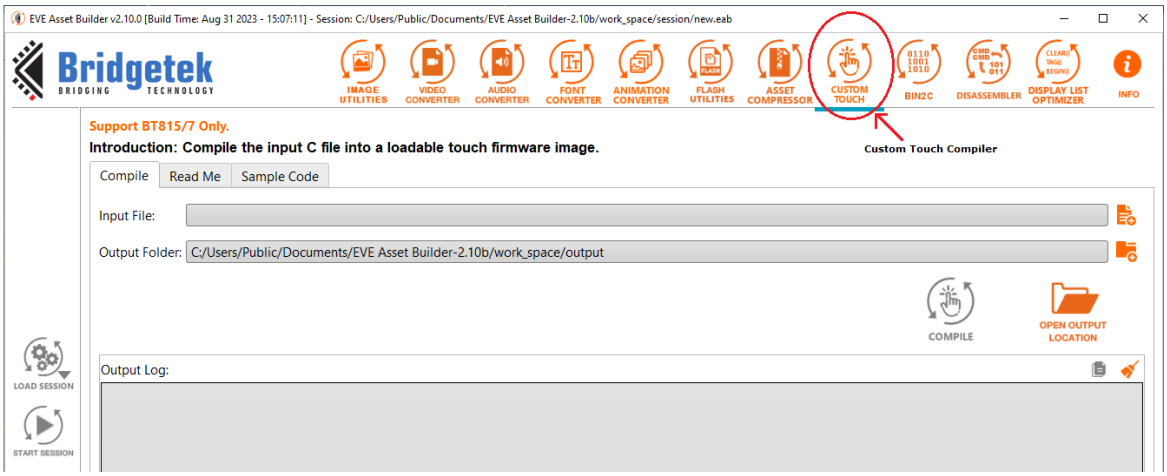

Figure 2 - EVE Asset Builder - Custom Touch Page

2. Click on the Add input file icon to add the C file of your custom touch source code as shown in Figure 3.

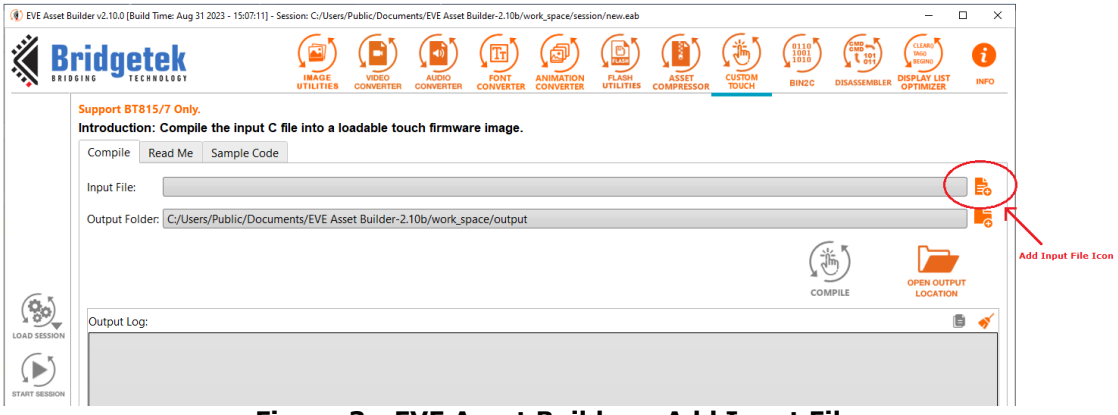

Figure 3 - EVE Asset Builder – Add Input File

3. A pop-up menu will appear. Select the desired source file and press Open. In this example, the ft.c is selected.

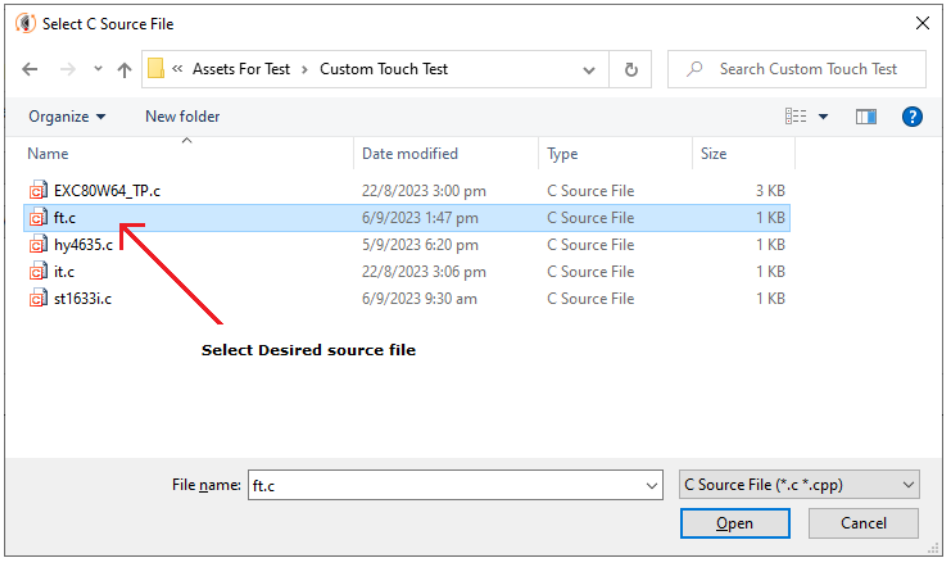

Figure 4 - EVE Asset Builder – Select C Source File

4. Click on the Add output Folder icon if user want to change to a preferred output folder.

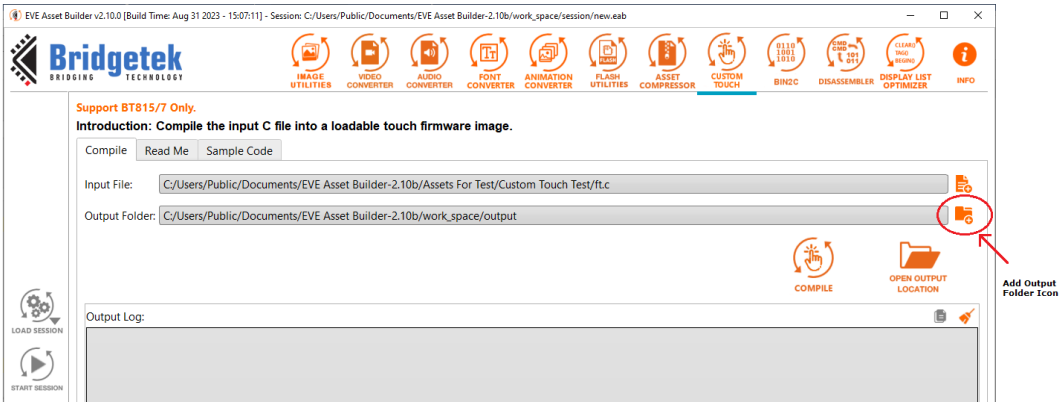

Figure 5 - EVE Asset Builder – Add Output Folder Icon

5. Once the desired C file is added to the compiler, the compile icon will change from Grey to Orange. Click on the Compile icon to start the compilation.

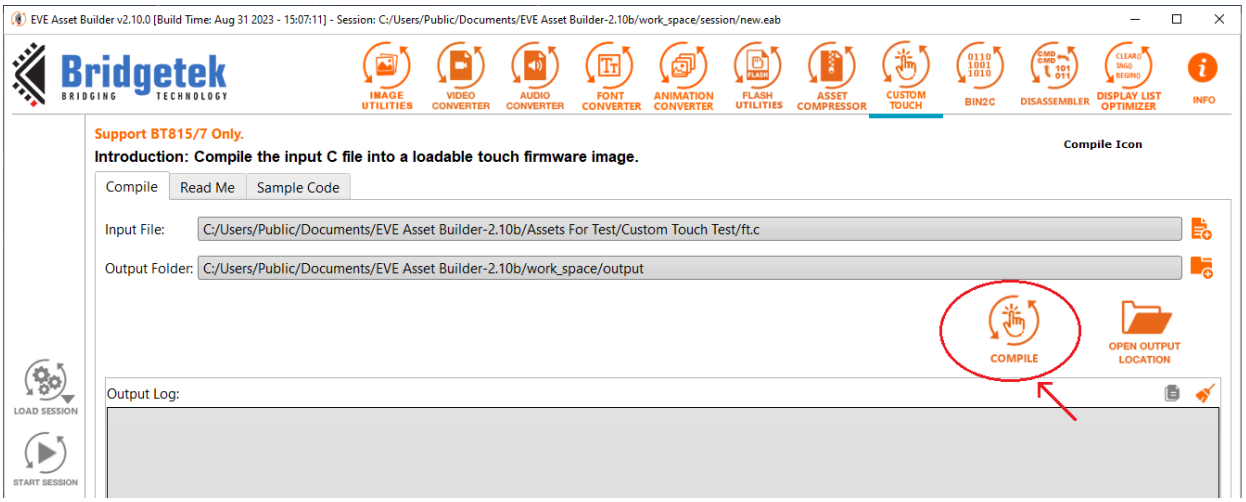

Figure 6 - EVE Asset Builder – Compile Icon

6. A message is pop-up after the compiler icon is pressed to indicate the compiling progress.

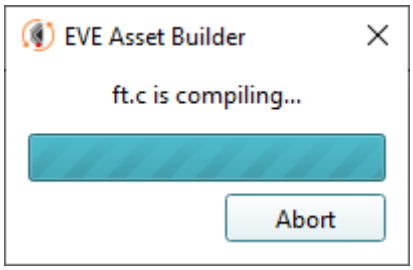

Figure 7 - EVE Asset Builder – Compiling Progress

7. After compilation, the output log message window will show the results of the compilation. When the compilation is successful, the output log shows the number of new bytes in firmware and the two files.

Figure 8 - EVE Asset Builder – Output Log

8. The Open Output Location icon allows the user to open the output location with just one click.

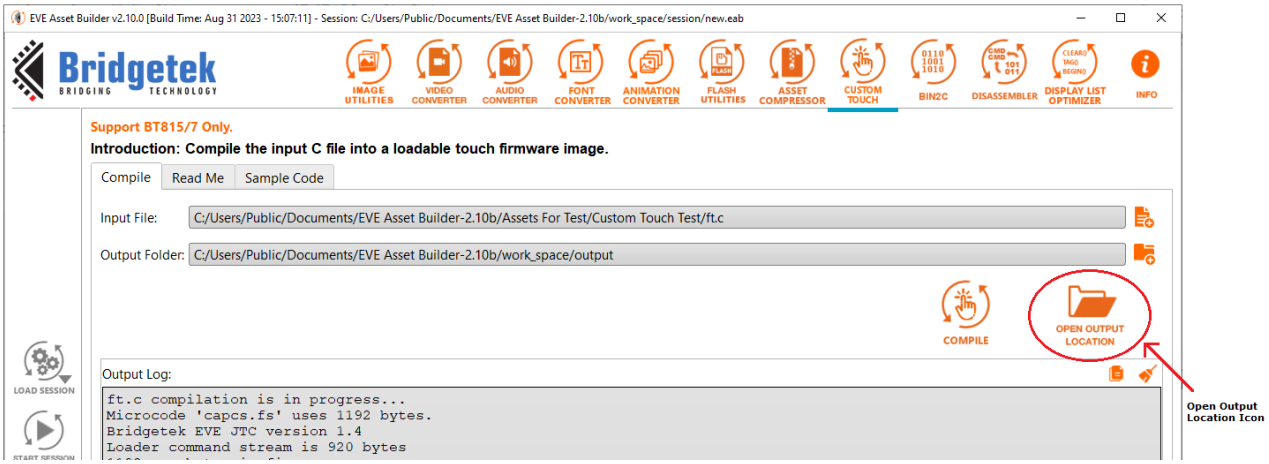

Figure 9 - EVE Asset Builder – Open Output Location

9. The output folder contains the compiled output files. In this case, 'ft.load.bin' and 'ft.load.h'. The ft.load.bin and ft.load.h is the co-processor loadable touch firmware.

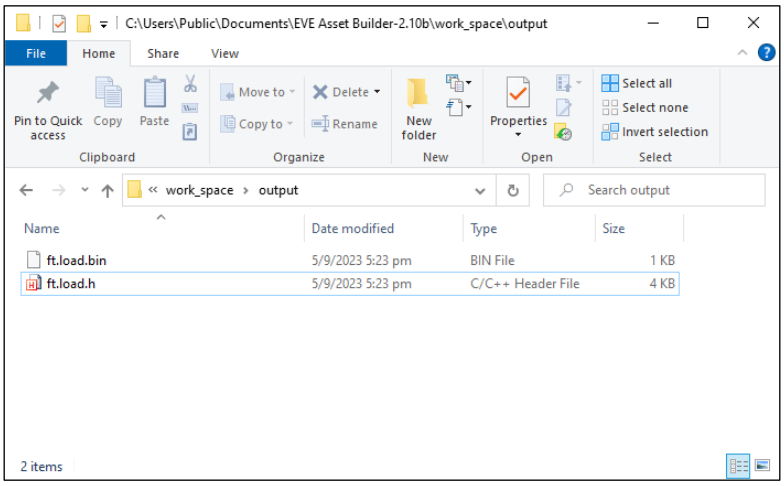

Figure 10 - EVE Asset Builder – Output Folder

5.4 Loading the code

When a reset is applied to the BT815/7 or when the power is lost, the information for the custom firmware is lost. As such, the custom firmware is required to be loaded each time during power-up or after the chip is reset. The custom firmware can be loaded in two ways.

The output.load.bin file can be read directly by the MCU and write to the RAM_CMD buffer. The MCU then updates the write pointer register to start the read transaction by the co-processor of the BT815/7. It then waits for the co-processor the complete all the transactions where the value of the read pointer is equal to the value of the write pointer.

Below shows a Pseudo-code on how to use the output.load.bin custom firmware.

```
/***
Assume that EVE boot-up sequence is properly done before this routine.
The custom firmware is a piece of code that can be executed by Eve coprocessor,
so the routine will do:
1) Read the firmware into local memory or use the .load.h file
2) Write the firmware into RAM_CMD
```

```
3) Update the register REG_CMD_WRITE to start execution
4) Wait till the execution is done (REG_CMD_READ==REG_CMD_WRITE)
***/
// Read the custom touch firmware 'output.load.bin' into local memory
custom_touch_content = read("output.load.bin", "b");

// Write the custom touch firmware into RAM_CMD
EVE_Hal_wrMem(RAM_CMD, custom_touch_content, sizeof(custom_touch_content));

// Update the write pointer register to start the execution
EVE_Hal_wr32(REG_CMD_WRITE, sizeof(custom_touch_content));

// Wait till Eve completes all the commands
while (EVE_Hal_rd32(REG_CMD_READ) == EVE_Hal_rd32(REG_CMD_WRITE));

// DONE
```

The custom firmware can also be integrated into the MCU firmware by using the output.load.h file. The output.load.h file is the binary file in text format. The MCU can integrate the data in this file to an array buffer and loads the array after the BT815/7 boot-up sequence is properly done.

Below Pseudo-code shows how to use the output.load.h custom firmware.

```
// This is the firmware compiled from "C:/Users/Public/Documents/EVE Asset Builder-2.10b/Assets For Test/Custom Touch Test/ft.c"
// To load the firmware, send the following
// bytes to the command FIFO (RAM_CMD)

uint8_t data[] = {
26,255,255,255,32,32,48,0,4,0,0,0,2,0,0,0,26,255,255,255,0,176,48,0,4,0,0,0,79,2,0,0,34,255,255,255,0,176,48,0,
120,218,93,19,77,104,156,85,112,222,247,54,171,217,148,117,63,41,69,36,11,251,173,27,151,118,83,40,221,182,
86,84,152,73,218,154,38,132,84,122,104,14,133,190,183,27,119,179,63,18,130,23,145,104,167,61,244,244,193,13
5,23,171,96,138,7,87,145,128,39,131,72,47,130,53,72,105,201,65,40,165,16,232,161,72,79,230,210,34,168,176,20
6,108,86,17,121,204,155,121,243,63,111,102,22,2,0,224,184,97,56,169,149,125,206,231,59,159,121,224,164,153,
243,235,62,20,40,251,216,149,133,82,126,217,43,165,116,190,163,186,123,250,249,206,63,84,236,108,51,110,10
0,48,110,136,135,229,156,79,150,237,114,220,16,188,2,237,184,81,197,164,150,69,187,252,122,193,250,55,32,11
3,177,203,226,97,124,21,66,151,212,36,122,11,218,146,71,179,204,134,115,190,234,18,201,98,22,194,110,206,16
7,48,238,166,113,140,231,112,14,22,80,99,217,90,206,135,157,193,45,81,237,202,113,225,167,177,44,22,192,249
,78,232,222,43,148,156,97,91,95,45,28,192,69,28,199,236,224,206,119,166,177,5,214,143,163,245,137,207,226,1
26,174,163,86,244,188,96,173,98,17,15,240,5,180,174,136,47,14,37,47,252,43,25,23,73,226,52,179,81,6,201,201,
182,146,250,62,161,146,38,240,58,66,227,243,66,236,128,51,28,118,191,42,104,238,161,251,182,128,215,122,12
0,26,191,145,106,161,189,1,137,239,97,15,38,249,44,84,120,30,13,31,226,170,179,254,144,88,100,240,167,66,15,
127,16,13,195,21,86,235,10,159,197,155,3,61,24,232,237,113,182,97,91,60,220,198,77,252,94,224,46,252,162,24
7,240,117,27,182,176,135,55,81,253,61,150,136,21,222,68,232,222,19,139,202,32,142,202,54,241,30,92,230,20,1
27,34,253,254,29,9,237,210,125,220,22,110,232,158,97,219,218,21,156,30,226,145,33,78,13,177,29,226,96,136,2
05,0,239,226,35,140,34,128,216,145,80,73,237,17,22,233,63,199,24,254,19,39,8,120,23,2,220,133,9,58,98,142,81
,241,127,231,136,153,160,163,100,253,14,20,249,60,20,4,74,2,145,244,250,20,101,241,12,189,36,175,0,3,236,247
,161,17,186,126,63,118,210,239,154,206,164,114,214,221,69,74,99,138,95,193,69,58,101,206,152,5,51,99,212,11
8,130,231,96,132,67,7,237,75,116,26,223,137,98,55,168,105,101,45,210,14,218,85,187,244,129,73,188,114,43,12
0,53,170,194,30,149,198,146,251,144,222,150,233,200,119,46,155,151,197,71,232,166,209,214,66,247,113,212,6
7,104,127,36,54,147,194,61,204,113,119,84,32,45,0,13,157,51,249,83,115,146,117,2,178,124,3,190,164,239,162,5
3,212,137,133,110,232,223,165,178,247,148,239,232,252,254,104,148,151,146,217,202,226,26,254,204,179,67,19
0,230,242,5,253,5,70,126,232,24,61,203,118,53,169,1,223,162,45,218,145,73,185,101,148,206,249,45,50,180,3,99
,60,47,17,126,139,212,19,200,158,229,100,106,160,254,107,20,250,75,52,141,186,45,35,12,205,135,209,251,248,
68,236,198,248,6,62,137,222,162,89,154,1,253,49,187,116,80,226,61,39,182,25,252,35,58,40,19,166,52,20,229,63
,165,142,167,6,174,60,37,233,251,148,228,208,74,5,70,94,163,65,48,184,129,175,83,134,55,36,159,235,52,121,10
1,3,78,80,63,177,245,19,146,175,245,10,37,23,202,198,148,164,63,251,196,86,183,70,229,95,147,238,218,125,25
0,148,228,159,154,39,139,213,32,224,59,98,163,122,192,101,124,179,56,62,165,180,225,227,178,121,175,5,40,15
4,32,154,243,197,153,192,250,7,102,255,212,209,41,35,81,207,5,127,3,25,211,104,171,0,0,0,26,255,255,255,32,3
2,48,0,4,0,0,0,0,0,0,0
};
```

```
// Write the custom touch firmware into RAM_CMD
EVE_Hal_wrMem(RAM_CMD, data, sizeof(data));

// Update the write pointer register to start the execution
EVE_Hal_wr32(REG_CMD_WRITE, sizeof(data));

// Wait till Eve completes all the commands
while (EVE_Hal_rd32(REG_CMD_READ) == EVE_Hal_rd32(REG_CMD_WRITE));

// DONE
```

5.5 Debugging the Code

When debugging the custom firmware, it is possible to add printf() functions to the code and print debug messages onto the attached LCD display. The code is compiled in the same way as described in Section 5.2.

The printf() function resembles C in syntax with the following limitations:

- Cannot have more than 3 variables in one printf() function. This causes the incorrect data to be printed.
- Only two format specifier is allowed: '%x' to display hexadecimal value and '%d' to display decimal value.
- For hexadecimal, it always shows data in 16-bits. That is, if the data is one byte long, zeros are added in front.

The debugger firmware for BT815 and BT817 is a different firmware. They are stored in <EAB install>/Asset For Test/Custom Touch Test/debug/BT815 and <EAB install>/Asset For Test/Custom Touch Test/debug/BT817 respectively.

To launch the BT815/7 into debugger mode, the custom firmware needs to be loaded to BT815/7 first followed by the respective debugger firmware. The debugger firmware is loaded to BT815/7 in the same way as described in Section 5.4.

After the debugger firmware is successfully loaded, the debugger screen is shown on the display panel as shown in Figure 11.

Figure 11 – Debugger Screen

Example

Taking the following custom firmware from FocalTech as an example, the printf() statements are added in the rtouch() function to print the values of the id, x, and y.

```
int i2c_addr()
{
    return 0x38;
}

void setup()
```

```
{
    setWAKE(0);
    delay_ms(10);
    setWAKE(1);
    delay_ms(300);
}

void rtouch()
{
    int x = i2c_read16be() & 0xfff;
    int id_y = i2c_read16be();
    i2c_read8();
    i2c_read8();

    printf("ID: %d, x: %x, y: %x\n", id_y >> 12, x, id_y & 0xfff);

   report_touch(id_y >> 12, x, id_y & 0xfff);
}

void loop()
{
   while (getINT() == 1) // wait for INT to go low
      ;
   i2c_startread(0x02);

   int n_touches = i2c_read8();
   while (n_touches != 0)
   {
      rtouch();
      n_touches = n_touches - 1;
   }
   i2c_stop();
   while (getINT() == 0) // wait for INT to go high
      ;
}
```

The custom touch firmware above is compiled and loaded to BT815/7 followed by the debugger firmware. When the screen is touch, the printed messages are shown on the screen as shown in Figure 12 – Printed Message in Debugger Mode

. More than one message may be shown on the screen as shown in this example depending on the sensitivity of the screen.

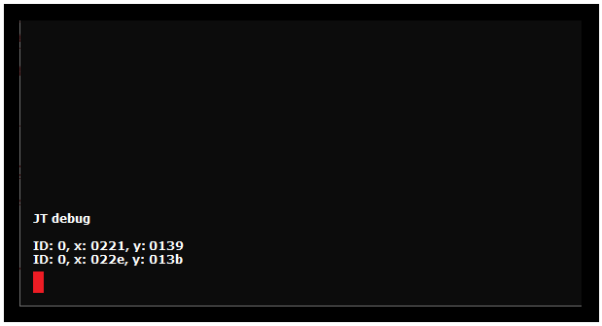

Figure 12 – Printed Message in Debugger Mode

Note that the printf() functions must be commented out as shown in the sample code below for normal operation without debug feature to avoid unexpected behaviour during normal operation.

```
int i2c_addr()
{
    return 0x38;
}

void setup()
{
    setWAKE(0);
    delay_ms(10);
    setWAKE(1);
    delay_ms(300);
}

void rtouch()
{
    int x = i2c_read16be() & 0xfff;
    int id_y = i2c_read16be();
    i2c_read8();
    i2c_read8();

    // printf("ID: %d, x: %x, y: %x\n", id_y >> 12, x, id_y & 0xfff);

    report_touch(id_y >> 12, x, id_y & 0xfff);
}

void loop()
{
    while (getINT() == 1) // wait for INT to go low
        ;
    i2c_startread(0x02);

    int n_touches = i2c_read8();
    while (n_touches != 0)
    {
        rtouch();
        n_touches = n_touches - 1;
    }
    i2c_stop();
    while (getINT() == 0) // wait for INT to go high
        ;
}
```

6 Host Driven Multi-Touch

If the MCU can provide touch inputs, it can supply them directly to the BT815/7 using Host Driven Multi-Touch. By using this mode, an application can choose to select a touch controller that is not in the BT815/7 direct support list.

To use the host driven multi-touch, the MCU shall be connected to the touch panel directly. The four touch related pins of the BT815/7 can be left unconnected on the PCB. The MCU is responsible for communicating with the touch controller, fetching the touch data when reported, and writing the touch data to the BT815/7 for touch TAG lookup and reporting.

The touch host mode can be entered by setting bit 14 in register REG_TOUCH_CONFIG and resetting the touch engine:

- Hold the touch engine in reset (set REG_CPURESET = 2)
- Write 1 to bit 14 in REG_TOUCH_CONFIG (set REG_TOUCH_CONFIG = 0x4000)
- Release the touch engine reset (set REG_CPURESET = 0)

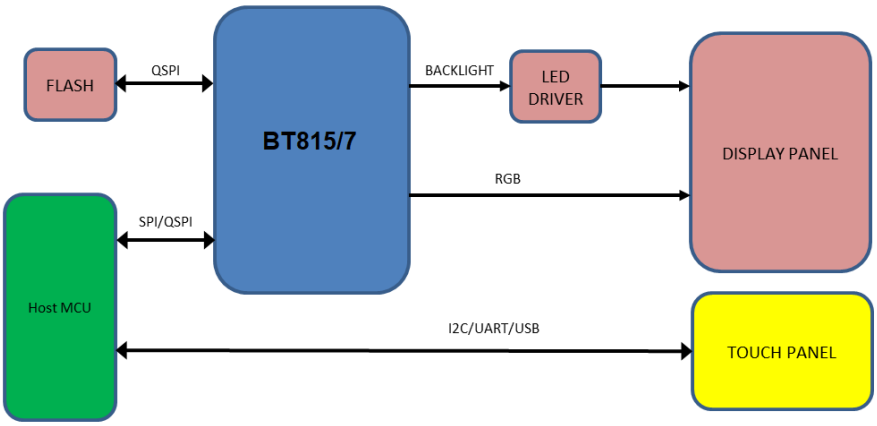

Figure 13 – Touch Host Mode Connection

In touch host mode, the host supplies touch information via four registers:

BT817 address	Register Name	Bits	Description
0x30210c	REG_EHOST_TOUCH_X	unsigned 16-bit	Touch x coordinates
0x302118	REG_EHOST_TOUCH_Y	unsigned 16-bit	Touch Y coordinates
0x302114	REG_EHOST_TOUCH_ID	4-bit	Touch ID / phase
0x302170	REG_EHOST_TOUCH_ACK	4-bit	Acknowledgement

The MCU writes raw (X, Y) coordinates and IDs to the above registers. Up to 5 touches can be set, using touch IDs 0 ~ 4. The MCU indicates no touch by supplying coordinates (0x8000, 0x8000). When the MCU writes 0xf to the ID register, BT815/7 sets the ACK register to 0, transforms all the raw coordinates, and writes the results to the regular touch registers.

Below shows a Pseudo-code on how to use the update the coordinates in Touch Host Mode.

```
wait until REG_EHOST_TOUCH_ACK is 1
        for each touch:
                write x coordinate to REG_EHOST_TOUCH_X
                write y coordinate to REG_EHOST_TOUCH_Y
                write id to REG_EHOST_TOUCH_ID
                write 0xf to REG_EHOST_TOUCH_ID
```

As soon as BT815/7 has converted the coordinates, it writes '1' to the ACK register and sets the INT_CONV_COMPLETE interrupt flag.

The ID should be zero in touch compatibility mode. The MCU should indicate no touch at all by writing (0x8000, 0x8000) with ID 0.

In extended mode, the multiple touches may be sent in any order. Any IDs not assigned are assumed to be not pressed. Again, the host should indicate no touch at all by writing (0x8000, 0x8000) with ID 0.

The MCU can use three methods to ensure that the BT815/7 is ready to accept touch inputs:

1. Poll the ACK register until it is '1'
2. Check the status of the INT_CONV_COMPLETE interrupt flag.
3. Supply touches slower that 1000Hz, since BT815/7 guarantees to process the touches in under 1ms. Note that report rates from capacitive touch panels are about 100Hz.

Like the direct capacitive driver, this host driven multi-touch works when REG_CTOUCH_EXTENDED is both CTOUCH_MODE_EXTENDED and CTOUCH_MODE_COMPATIBILTIY. CTOUCH_MODE_COMPATIBILITY should be used for the calibration procedure, just as when using native capacitive support. After changing mode, the BT815/7 touch engine must be reset.

7 Touch Screen Calibration

The calibration feature of the BT815/7 allows it to determine the alignment of the touch panel relative to the display screen. When the application runs the calibration command on the BT815/7, the user is requested to tap three dots on the screen. The position of the three dots are as follows (X, Y): (10%w, 10%h), (90%w, 50%h), and (50%w, 90%h) where 'w' is the width of the screen resolution and 'h' is the height of the screen resolution. In addition, the BT817 also supports sub-windows calibration by using the CMD_CALIBRATESUB(). Instead of using the whole screen area as specified above, user can use a smaller sub-window specified by the CMD_CALIBRATESUB(). This is useful for TFT modules with special shape such as round display and bar-type display. The BT815/7 calculates six transform values during the calibration routine which then allow it to make adjustments so that the users touch is aligned to the graphics underneath. It is normally required that the screen is calibrated for any application which will use the touch to ensure touch accuracy. However, the calibration process can be skipped for those capacitive touch TFT modules that are already calibrated at factory where the touch screen's resolution and orientation is the same as the display's resolution and orientation.

Note that the calibration is carried out in compatibility mode and the transforms still apply when switching to extended mode.

When power to the BT815/7 is lost or a reset is applied to the BT815/7, these transform values are also lost. In most of the demo applications, this requires the calibration routine to be run after each power-up. To provide a better end-user experience, it is possible to run the calibration once, store the values on the MCU's non-volatile memory or BT815/7 attached flash memory and then restore the values after each power on.

7.1 Sub-Window Calibration

The sub-window calibration is done by running the sub-window calibration command, CMD_CALIBRATESUB(). This command is used to execute the touch screen calibration routine in a sub-window defined by the coordinates supplied to the command.

C prototype

```
void cmd_calibratesub(uint16_t x,
                      uint16_t y,
                      uint16_t w,
                      uint16_t h,
                      uint32_t result);
```

Parameters

x	x-coordinate of top-left of sub-window, in pixels
y	y-coordinate of top-left of sub-window, in pixels
w	width of sub-window, in pixels
h	height of sub-window, in pixels
results	output parameter; written with 0 on failure

7.1.1 Round Display

Round displays, like square displays, uses Cartesian coordinates. When running the calibration command, CMD_CALIBRATE(), the position of the three dots may displayed off-screen causing the calibration process not able to complete as the user is not able to tap the dots. To illustrate further, consider the diagram shown in Figure 14. The circle with the origin 'O' represents the round display, while the width of the display is represented by AB and the height of the display is represent by BC. The coordinates that are in the areas in Grey are outside the round display. Hence, when using the CMD_CALIBRATE() to calibrate this display, the first dot (10%w, 10%h) will be outside the visible display.

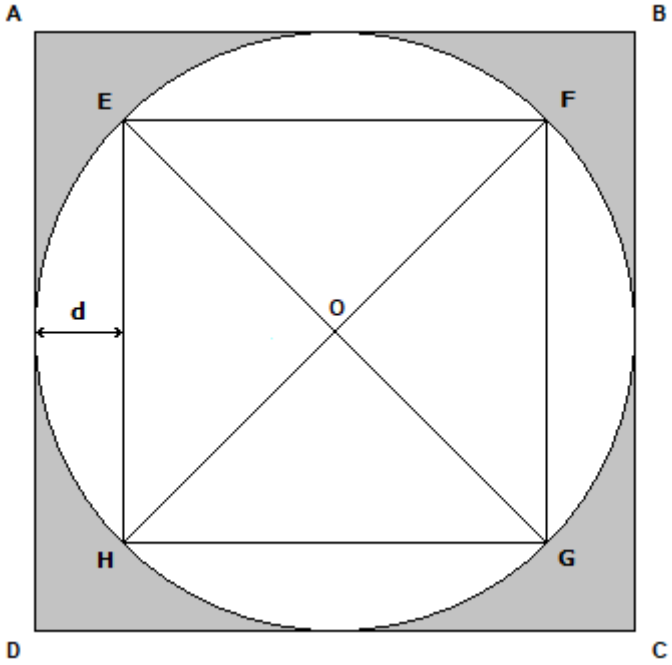

Figure 14 – Largest Square to fit in a Round Display

To effectively calibrate the touch screen of the round display, the largest square that can fit into the round display is first determined shown in Figure 14 as square EFGH. The coordinates at point E and the width and height of the square can then be used as coordinates for the sub-window calibration.

To determine the coordinates at point E, the length of the square is first determined. This can be done by using the Pythagoras theorem as EFG is a right-angle triangle.

$$(EG)^2 = (EF)^2 + (FG)^2 \qquad \text{——————} \quad (1)$$

As the length EF and FG is the same and the length of EG is the same as AB (Diameter of the circle). The equation (1) can be rewritten as

$$(AB)^2 = 2 \cdot (EF)^2 \qquad \text{——————} \quad (2)$$

Therefore, length EF is determined as

$$EF = \frac{AB}{\sqrt{2}} \qquad \text{——————} \quad (3)$$

The distance from the round display to the square, d, can be determined as

$$d = \frac{AB - EF}{2} \qquad \text{——————} \quad (4)$$

Replacing EF with equation (3)

$$d = \frac{\left(AB - \frac{AB}{\sqrt{2}}\right)}{2}$$

$$= \frac{(\sqrt{2}) \cdot (AB) - (AB)}{2(\sqrt{2})}$$

$$= \left(\frac{(\sqrt{2}) - 1}{2(\sqrt{2})}\right) \cdot (AB)$$

$$d \approx 0.146 \cdot (AB) \qquad \text{——————} \quad (5)$$

Hence, taking into consideration that the coordinates at A is (0, 0), the coordinates at point E is (d, d) = ((0.146 x AB), (0.146 x AB)). The width of the sub-window is EF = (0.707 x AB) and the height of the sub-window is FG = (0.707 x AB).

By applying these values to the command CMD_CALIBRATESUB(), the three dots will be displayed within the window and the calibration can be performed successfully.

Example

Consider a round display with 480x480 resolution, the coordinates of top-left of sub-window, in pixels would be ((0.146 x 480), (0.146 x 480)) = (70, 70) and the width of the sub-windows, in pixels would be (0.707 x 480) = 339 and the height of sub-windows, in pixels would be (0.707 x 480) = 339.

```
cmd_dlstart();
cmd(CLEAR(1,1,1));
cmd_text(240, 140, 31, OPT_CENTER,” Please tap on the dot”);
cmd_calibratesub(70, 70, 339, 339, 0);
```

7.1.2 Bar-type display

In some bar-type display, the visual display may not start from coordinates (0, 0). As illustrated in Figure 15 – Digram to illustrate the visible coordinates not at (0,0)

, point A is coordinates (0, 0) but the visible display is from point F onwards. In such cases, if the CMD_CALIBRATE() is used to calibrate the screen, the first dot at (10%w, 10%h) will be outside the visible screen.

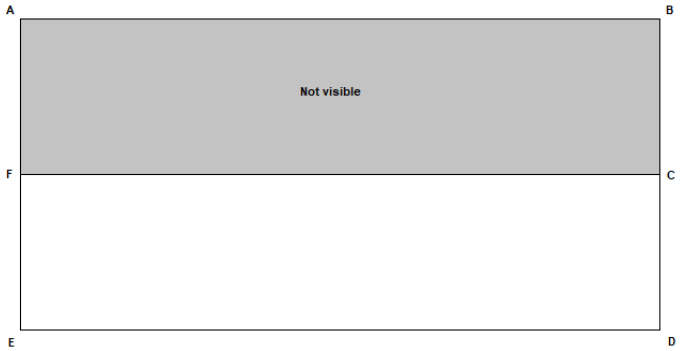

Figure 15 – Digram to illustrate the visible coordinates not at (0,0)

To calibrate this type of screen, user is required to know the coordinates at point F and create a sub-windows at the visible part of the screen using the CMD_CALIBRATESUB().

7.2 Storing Calibration Values

In order to store the values, the calibration could, for example, be run during a factory test of the finish product. The MCU would do the following:

During factory test:

- Run a co-processor list containing the Calibrate command
- Wait for REG_CMD_READ == REG_CMD_WRITE (this indicates the completion of the co-processor command list and will only occur once the user has tapped the three dots)
- The command will have populated the registers REG_TOUCH_TRANSFORM_A to REG_TOUCH_TRANSFORM_F with the transform values.
- The MCU can now read these six registers with standard 32-bit register read commands and can store the values in the MCU or BT815/7 attached flash for example.

On Power up:

- The MCU can now write the six values back to their respective REG_TOUCH_TRANSFORM registers with standard 32-bit register writes, instead of running the calibration command.

Re-calibrate Option:

- It may also be desirable to provide a menu option in the application or another way in which the user can re-run the calibration if required (e.g., during maintenance of a machine or if the screen had been damaged and replaced with a new LCD panel). The new values are then read and replace the old values in the MCU or BT815/7 attached flash.

Example

Pseudo-code example below demonstrates one possible application. In this scenario, on power-up, the MCU would initialize the BT815/7 and enters the calibration routine only if the calibration has never been run before or if the user is pressing (touching) and holding the screen during power-up. Otherwise, it is considered that the calibration had been carried out already (and so suitable transform values are available in the MCU's EEPROM) and the operator does not wish to re-calibrate. The values are loaded, in this case, from the MCU's EEPROM and the operator does not need to carry out the tapping of the calibration dots.

The code below could be called each time the MCU and BT815/7 are powered up and before the main application starts. If the calibration dots are not already stored in the MCU's EEPROM or if the user touches and holds the screen during power-up, the calibration routine is run and the resulting values are read by the MCU and stored in its EEPROM. Otherwise, the values from the MCU's EEPROM from the previous calibration are written to the BT815/7 REG_TOUCH_TRANSFORM_A to REG_TOUCH_TRANSFORM_F registers.

This code uses byte 0 of the MCU's EEPROM to indicate whether the calibration data has already been stored in its EEPROM (0x7C was used to indicated 'values stored') and uses EEPROM bytes 1 – 24 to store the actual calibration data copied from the BT815/7 REG_TOUCH_TRANSFORM_A ~ F.

```
// Check if calibration data exists already or if the user is touching the screen
If ((IsTouch()) || (EEPROM.Read(0) != 0x7C))
{
	// Blank the screen
	Blank();

	// Wait for user to release touch
	while (IsTouch())
		;

	// Ensure the display PWM is at 100%
	write(REG_PWM_DUTY, 128);
```

```
	// Start a new co-processor command list
	BeginCoProList();

	CMD DLSTART		// DL Start command
	CMD CLEAR(1,1,1)	// Clear the buffers
	CMD COLOR_RGB(255,255,255)		// Set colour for the subsequent text
	CMD TEXT(screen.w/2, screen.h/2, 28, OPT_CENTERX|OPT_CENTERY, "please tap on the dot");
	CMD CALIBRATE		// Run the actual calibration
	CMD DISPLAY			// Display command
	CMD SWAP		// Swap command

	FlushCoProBuffer();	// Send above co-processor commands to the BT815/7
	WaitCmdFifoIdle();	// Wait until co-processor finishes execution

	// i.e. until REG_CMD_READ == REG_CMD_WRITE
	// The FT8xx's registers REG_TOUCH_TRANSFORM_A to REG_TOUCH_TRANSFORM_F now have
	// their calibrated values. The MCU can read the six 32-bit values. Here, we read
	// them a byte at a time since the EEPROM is programmed on a byte-by-byte basis.

	for (int i = 0; i< 24; i++)
	{
		EEPROM.write(1 + i, rd32(REG_TOUCH_TRANSFORM_A + i));
	}

	EEPROM.write(0, 0x7c); // Write loc 0 to 0x7C to show the data is stored

	// Now EEPROM(0) has value 0x7C and EEPROM(1) to (24) have the values of the
	// six 32-bit REG_TOUCH_TRANSFORM registers
}
else
{
	// If the calibration values were already in EEPROM and the user had not touched
	// the screen on power-up then we read the existing EEPROM values and write them
	// to the FT8xx's REG_TOUCH_TRANSFORM registers

	for (int i = 0; i< 24; i++)
		wr32(REG_TOUCH_TRANSFORM_A + i, EEPROM.read(1 + i));
}
// Now, the main application can begin
```

8 Conclusion

The BT815/7 can support a wide range of CTPM by using the custom firmware or Touch Host Mode when the touch controller is not in the direct support list. This application note provides information and guidance for users when choosing and configuring different CTPM for their applications.

9 Contact Information

Refer to https://brtchip.com/contact-us/ for contact information

Distributor and Sales Representatives

Please visit the Sales Network page of the Bridgetek Web site for the contact details of our distributor(s) and sales representative(s) in your country.

Appendix A – References

Document References

DS_BT81X (BT81X Advanced Embedded Video Engine)
DS_BT817_8 (BT817/8 Advanced Embedded Video Engine)
AN_336 FT8xx – Selecting an LCD Display
BRT_AN_033 BT81X Series Programming Guide Version
EVE Asset Builder (EAB) 2.10.0 or later

Acronyms and Abbreviations

Terms	Description
ACK	Acknowledge
CTP_SCL	Capacitive Touch Panel Serial Clock
CTP_SDA	Capacitive Touch Panel Serial Data
CTPM	Capacitive Touch Panel Module
CTSE	Capacitive Touch Screen Engine
EAB	EVE Asset Builder
EVE	Embedded Video Engine
EEPROM	Electrical Erasable Programmable Read-Only Memory
ID	Identity
I^2C	Inter-Integrated Circuit
LCD	Liquid Crystal Display
MCU	Microcontroller Unit
PCB	Printed Circuit Board
ROM	Read-Only Memory
SCL	Serial Clock
SDA	Serial Data

Appenix B – List of Figures and Tables

List of Figures

List of Tables

Appendix C – Revision History

Document Title: BRT_AN_090 EVE Working with Capacitive Touch Screens
Document Reference No: BRT_000422
Clearance No: BRT#204
Product Page: http://brtchip.com/product
Document Feedback: Send Feedback

Revision	Changes	Date
1.0	Initial Release	01-11-2023