Color Format Conversion Using Arm Helium

Example Code Introduction for 32-bit NuMicro® Family

Document Information

Application	This example code uses the Arm Helium to perform color format conversion and functional verification.
BSP Version	M55M1_Series_BSP_CMSIS_V3.00.001
Hardware	NuMaker-M55M1(BETA) V1.1

The information described in this document is the exclusive intellectual property of Nuvoton Technology Corporation and shall not be reproduced without permission from Nuvoton.

Nuvoton is providing this document only for reference purposes of NuMicro microcontroller and microprocessor based system design. Nuvoton assumes no responsibility for errors or omissions.

All data and specifications are subject to change without notice.

For additional information or questions, please contact: Nuvoton Technology Corporation.

www.nuvoton.com

1. Overview

This example code uses Arm Helium to implement color format conversion and functional verification. RGB565 to RGB888 color format conversion is done with the M-Profile Vector Extension (MVE) instruction provided by Arm Helium. The example also includes a non-vectorized conversion code, which is used for functional verification and used as a reference base of the performance improvement.

1.1 Principle

Helium vectorization allows CPU to operate on 128-bit data simultaneously, such as load, store, logical and vector arithmetic operations. According to user scenarios, 128-bit data can be configured as different width/lane combinations. For example, it could be sixteen 8-bit lanes, eight 16-bit lanes, and four 32-bit lanes, as shown in Figure 1-1.

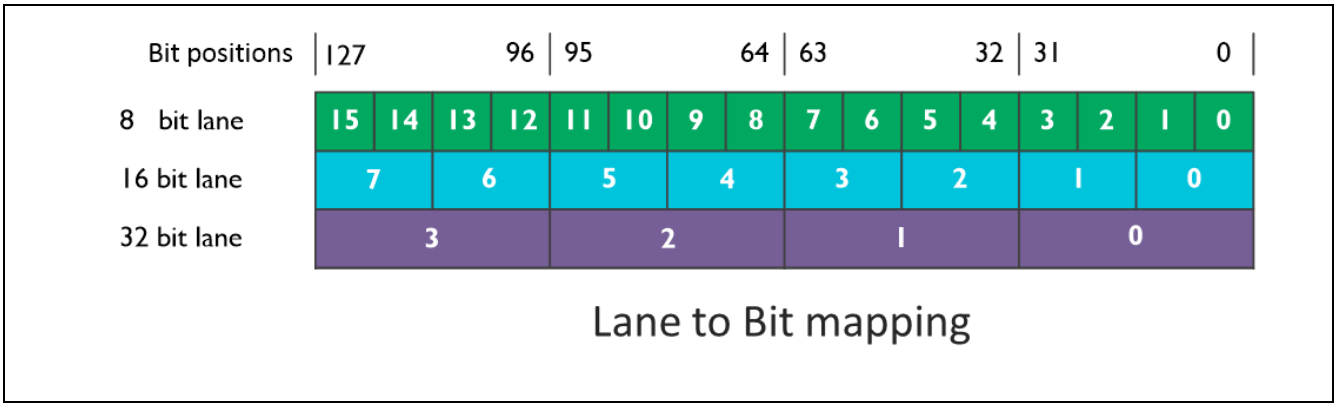

Figure 1-1 Helium Lanes and Data Width Definition

The processing flow of RGB565 to RGB888 color conversion in non-vectorized way is shown in Figure 1-2. One RGB565 pixel is converted at a time. Input data width is 16-bit, however, operations involved are byte-based. One converted RGB888 pixel is composed of three bytes. After conversion, output data width is 24-bit.

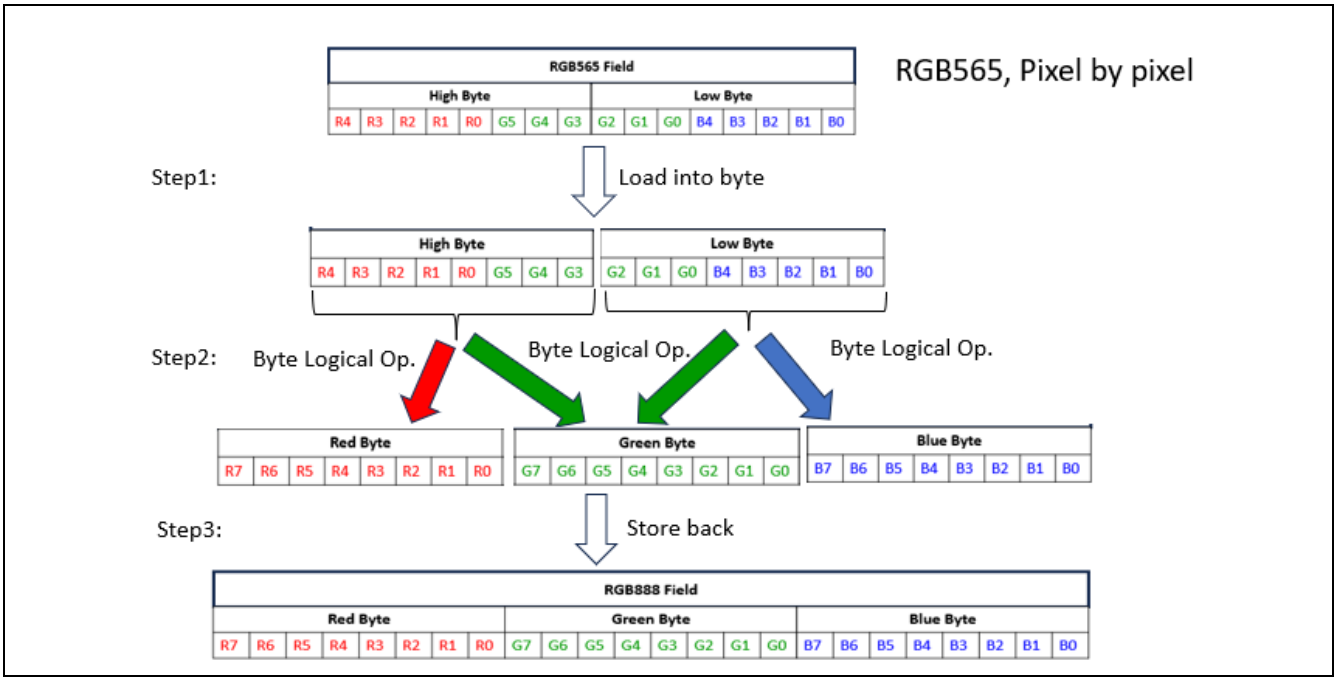

Figure 1-2 RGB565 to RGB888 Process Flow

If sixteen pixels are loaded into sixteen lanes of Helium at the same time, sixteen byte-based logical operations can be done all together in step2. The conversion is vectorized. The flow is shown in Figure 1-3.

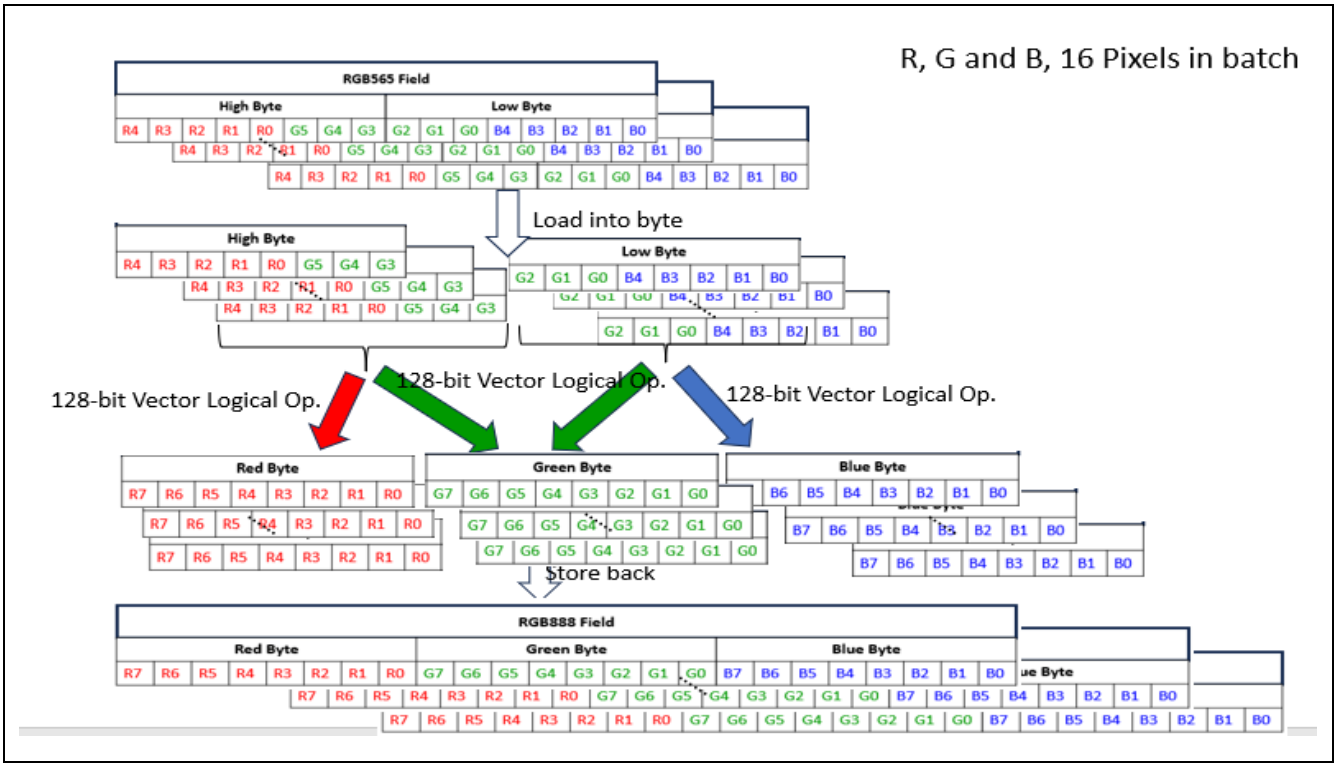

Figure 1-3 RGB565 to RGB888 Vectorized Process Flow

Generally, pixel data resides on memory of contiguous address. To load RGB565 high byte and low byte into two vector registers, Helium provides “scatter load” scheme. To execute scatter load, one should provide base address and offset. For example, to load RGB565 high byte, the corresponding offset should be specified as 0x*0e0c0a080604020*. For RGB565 low byte, the specified offset is 0x0*f0d0b0907050301*, each offset is represented by 1 byte.

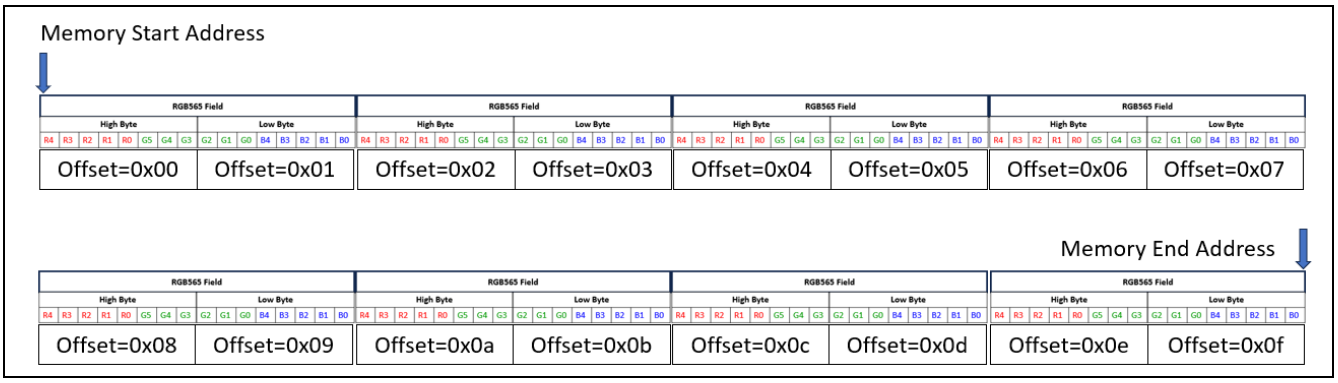

Figure 1-4 RGB565 Scatter Load with Offset

After conversion, the converted R, G, and B data are stored in three vector registers. These separated data can be placed on a continuous memory address through the Helium “scatter store” scheme. See Figure 1-5 as a simple case, to put first two RGB888 pixel back to memory in contiguous address, one should specify that offset of R is specify as 0x0300, G as 0x0401 and B as 0x0502 respectively.

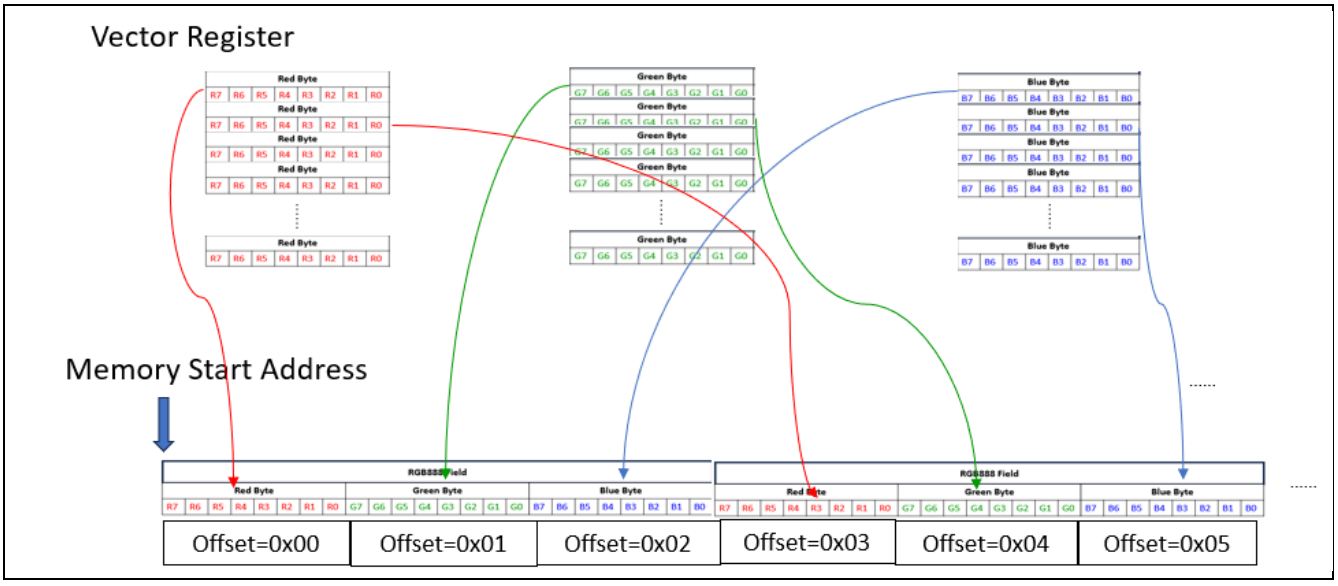

Figure 1-5 RGB565 Scatter Store with Offset

1.2 Execution Result

After running this example, the execution cycle counts of vectorized and non-vectorized conversions are provided as follows. The speed improvement factor is also available, and the conversion result is checked byte by byte.

```
+----------------------------------------+
|   Helium RGB565 to 888 Sample Code KEIL |
+----------------------------------------+

 Execute RGB565to888

 Tick Counters are:558!

 Execute VRGB565to888_TP

 Tick Counters are:382!

 Compare RGB565 Results
 Match: image_dst[0] = 0xf0
 Match: image_dst[1] = 0x58
 Match: image_dst[2] = 0xf8
 Match: image_dst[3] = 0x30
 Match: image_dst[4] = 0x94
 Match: image_dst[5] = 0x90
 Match: image_dst[6] = 0xa0
 Match: image_dst[7] = 0x94
 Match: image_dst[8] = 0x10
 Match: image_dst[9] = 0x38
 Match: image_dst[10] = 0x50
 Match: image_dst[11] = 0x90
 Match: image_dst[12] = 0xc0
 Match: image_dst[13] = 0x98
 Match: image_dst[14] = 0x10
 Match: image_dst[15] = 0xa0
 Match: image_dst[16] = 0x94
 Match: image_dst[17] = 0x10
 Match: image_dst[18] = 0xa0
 Match: image_dst[19] = 0x8c
 Match: image_dst[20] = 0x10
 Match: image_dst[21] = 0x30
 Match: image_dst[22] = 0x94
 Match: image_dst[23] = 0x10
 Match: image_dst[24] = 0x30
 Match: image_dst[25] = 0xb4
 Match: image_dst[26] = 0x10
 Match: image_dst[27] = 0x70
 Match: image_dst[28] = 0x80
 Match: image_dst[29] = 0xe0
 Match: image_dst[30] = 0x30
 Match: image_dst[31] = 0x9c
 Match: image_dst[32] = 0x90
 Match: image_dst[33] = 0x30
 Match: image_dst[34] = 0x84
```

```
Match: image_dst[35] = 0x90
Match: image_dst[36] = 0x30
Match: image_dst[37] = 0x88
Match: image_dst[38] = 0x10
Match: image_dst[39] = 0x30
Match: image_dst[40] = 0x88
Match: image_dst[41] = 0x90
Match: image_dst[42] = 0x30
Match: image_dst[43] = 0x8c
Match: image_dst[44] = 0x10
Match: image_dst[45] = 0x70
Match: image_dst[46] = 0xec
Match: image_dst[47] = 0x90

RGB565 Speedup 1.46 X

Test Done
```

2. Code Description

The program that calculates RGB565 to RGB888 is located in *main.c*.

```
……
#define BYTENUMBER_RGB565  2
……
/**
  * @brief      Convert RGB565 to RGB888 Color Format
  * @param[in]  src is the source data(RGB565) address
  * @param[in]  dst is the destination data(RGB88) address
  * @param[in]  len is the source data length in packets
  * @return     void
  * @details    This function converts RGB565 to RGB888 color format in non vector manner.
  */

void RGB565to888(uint8_t *src, uint8_t *dst, uint8_t len)
{
    uint8_t buf888_high, buf888_low;
    uint8_t dst888_r, dst888_g, dst888_b;
    int count = len;

    while (count >= 0)
    {
        //RGB565_IN_UINT8

        //Sequential Load
        buf888_high = *src;
        buf888_low = *(src + 1);

        //Separate RGB
        dst888_r  = (buf888_high & 0xF8) ;
        dst888_g = ((buf888_high & 0x07) << 5) | ((buf888_low & 0xE0) >> 3) ;
        dst888_b  = (buf888_low << 3) ;

        //Sequential Store
        *(dst++) = dst888_r;
        *(dst++) = dst888_g;
        *(dst++) = dst888_b;

        src += BYTENUMBER_RGB565;
        count --;
    }
}
```

The above function does not utilize vectorized processing. It converts data pixel by pixel. On the other hand, the code of vectorized processing for color conversion is as follows:

```
….
#define OFFSET_R_LOW     0x15120F0C09060300
#define OFFSET_R_HIGH    0x2D2A2724211E1B18
#define OFFSET_G_LOW     0x1613100D0A070401
#define OFFSET_G_HIGH    0x2E2B2825221F1C19
#define OFFSET_B_LOW     0x1714110E0B080502
#define OFFSET_B_HIGH    0x2F2C292623201D1A
……
#define OFFSET_565HIGHBYTE_LOW     0x0E0C0A0806040200
#define OFFSET_565HIGHBYTE_HIGH    0x1E1C1A1816141210
```

```
#define OFFSET_565LOWBYTE_LOW    0x0F0D0B0907050301
#define OFFSET_565LOWBYTE_HIGH   0x1F1D1B1917151311

#define BYTENUMBER_RGB888  3
#define PIXELS_LOOP       16
#define PIXELS_CONVERT_EACHLOOP       16
…
/**
  * @brief      Convert RGB565 to RGB888 Color Format
  * @param[in]  src is the source data(RGB565) address
  * @param[in]  dst is the destination data(RGB88) address
  * @param[in]  size_h is the height of source image data
  * @param[in]  size_w is the width of source image data
  * @return     void
  * @details    This function converts RGB565 to RGB888 color format in vector manner.
  */

void VRGB565to888_TP(uint8_t *src, uint8_t *dst, uint16_t size_h, uint16_t size_w)
{
    int8_t count;
    uint8x16_t vsrc_8_16_lo;
    uint8x16_t vsrc_8_16_hi;
    uint8x16_t vdst_8_16_r;
    uint8x16_t vdst_8_16_g;
    uint8x16_t vdst_8_16_b;
    uint8x16_t offset_8_16;

    count = size_h * size_w;

    while (count > 0)
    {
        //RGB565_IN_UINT8

        //Include tail predication to handle improper tailing.
        mve_pred16_t p = vctp8q(count);

        //Scatter Load//
        offset_8_16 = vcreateq_u8(OFFSET_565LOWBYTE_LOW, OFFSET_565LOWBYTE_HIGH);

        vsrc_8_16_lo = vldrbq_gather_offset_z_u8(src, offset_8_16, p);

        offset_8_16 = vcreateq_u8(OFFSET_565HIGHBYTE_LOW, OFFSET_565HIGHBYTE_HIGH);

        vsrc_8_16_hi = vldrbq_gather_offset_z_u8(src, offset_8_16, p);

        //Separate RGB//
        vdst_8_16_r  =  vrshlq(vshrq(vsrc_8_16_hi, 3), 3);//vrshrq results in rounded.
        vdst_8_16_g =  vorrq_u8(vrshlq(vsrc_8_16_hi, 5),
                                vrshlq(vshrq(vsrc_8_16_lo, 5), 2));

        vdst_8_16_b =  vrshlq(vsrc_8_16_lo, 3);
        //Scatter Store//
        offset_8_16 = vcreateq_u8(OFFSET_R_LOW, OFFSET_R_HIGH);

        vstrbq_scatter_offset_p(dst, offset_8_16, vdst_8_16_r, p);

        offset_8_16 = vcreateq_u8(OFFSET_G_LOW, OFFSET_G_HIGH);

        vstrbq_scatter_offset_p(dst, offset_8_16, vdst_8_16_g, p);
```

```
        offset_8_16 = vcreateq_u8(OFFSET_B_LOW, OFFSET_B_HIGH);

        vstrbq_scatter_offset_p(dst, offset_8_16, vdst_8_16_b, p);

        //Update data source and destination pointer

        //Source: shift one RGB566 pixel size(2 byte)
        src += BYTENUMBER_RGB565 * PIXELS_CONVERT_EACHLOOP;

        //Destination: shift one RGB888 pixel size(3 byte)
        dst += BYTENUMBER_RGB888 * PIXELS_CONVERT_EACHLOOP;

        count -= PIXELS_CONVERT_EACHLOOP;

    }
}
```

The functions used are summarized in following tables. Table 2-1 covers logical operation parts:

Intrinsic Function	Parameter	Return Value	Function Description
vshrq	(uint8x16_t a, const int imm)	uint8x16_t	Shift each element of the first vector register *a* in the parameter to the right by a constant value *imm*, and the result is written to the return value vector register.
vrshlq	(uint8x16_t a, int32_t b)	uint8x16_t	Each element of the first vector register *a* in the parameter is shifted to the left by the value of the lowest byte of the second variable *b*, and the result is written to the return value vector register.
vorrq_u8	(uint8x16_t a, uint8x16_t b)	uint8x16_t	The bitwise logic OR results of the two vector registers *a* and *b* in the parameters are calculated and written to the return value vector register.

Table 2-1 Functions of Helium Logical Operations

Note that logical operation of 16 lanes is executed simultaneously. Take the color of blue as example, in non-vectorized case:

```
dst888_b = (buf888_low << 3);
```

vectorized case:

```
vdst_8_16_b = vrshlq(vsrc_8_16_lo, 3);
```

Table 2-2 covers data load/store parts:

Intrinsic Function	Parameter	Return Value	Function Description
vcreateq_u8	(uint64_t a, uint64_t b)	uint8x16_t	Returns a vector with the same representation as the concatenation of the two 64-bit values *a* and *b* specified by the parameter.
vldrbq_gather_offset_z_u8	(uint8_t const * base, uint8x16_t offset, mve_pred16_t p)	uint8x16_t	Loads contiguous elements from memory address *base* to the return vector register. The target address is calculated based on the base address and offset. With reference p, if data length is smaller than 16, zero will be padded by Helium.
vstrbq_scatter_offset_p	(uint8_t * base, uint8x16_t offset, uint8x16_t value, mve_pred16_t p)	void	Store successive elements from the vector register *value* to the memory base location. The target address is calculated according to the base address and offset. With reference p, if data length is smaller than 16, zero will be padded by Helium.

Table 2-2 Functions of Helium Data Load/Store Operations

Refer to Figure 1-4 and Figure 1-5 for detailed description about scatter load/store. In non-vectorized case, one pixel is loaded at a time:

```
//Sequential Load
buf888_high = *src;
buf888_low = *(src + 1);
```

In vectorized case, sixteen pixels are loaded at a time and high/low bytes are assigned in two different vector registers. Since low byte addresses of adjacent pixels are not contiguous, base address and offset of the pixel are required for scatter load.

```
……
#define OFFSET_565LOWBYTE_LOW    0x0F0D0B0907050301
#define OFFSET_565LOWBYTE_HIGH   0x1F1D1B1917151311

//Scatter Load//
offset_8_16 = vcreateq_u8(OFFSET_565LOWBYTE_LOW, OFFSET_565LOWBYTE_HIGH);

vsrc_8_16_lo = vldrbq_gather_offset_z_u8(src, offset_8_16, p);
```

OFFSET_565LOWBYTE_LOW and OFFSET_565LOWBYTE_HIGH are 64-bit in width. According to Table 2-2, after executing the above code, content of vector register offset_8_16 should be:

Offset_8_16(uint8x16t)

0x1F	0x1D	0x1B	0x19	0x17	0x15	0x13	0x11	0xF	0xD	0xB	0x9	0x7	0x5	0x3	0x1

vsrc_8_16_lo(uint8x16t)

src [0x1F]	src [0x1D]	src [0x1B]	src [0x19]	src [0x17]	src [0x15]	src [0x13]	src [0x11]	src [0xF]	src [0xD]	src [0xB]	src [0x9]	src [0x7]	src [0x5]	src [0x3]	src [0x1]

For data storage, one RGB888 pixel (3 bytes) is stored at a time in non-vector case.

```
//Sequential Store
*(dst++) = dst888_r;
*(dst++) = dst888_g;
*(dst++) = dst888_b;
```

In vector case, the contents of the vector register (16 lanes) will be stored back to memory at the same time. To store in continuous manner, the base address and offset are required.

Take color of R for example,

```
….
#define OFFSET_R_LOW     0x15120F0C09060300
#define OFFSET_R_HIGH    0x2D2A2724211E1B18
……
//Scatter Store//
offset_8_16 = vcreateq_u8(OFFSET_R_LOW, OFFSET_R_HIGH);
```

Offset_8_16(uint8x16t)

0x2D	0x2A	0x27	0x24	0x21	0x1E	0x1B	0x18	0x15	0x12	0xF	0xC	0x9	0x6	0x3	0x0

```
vstrbq_scatter_offset_p(dst, offset_8_16, vdst_8_16_r, p);
```

After the above code is executed, the data of the in-memory *dst* is as follows:

address	0x2F	0x2E	0x2D	……	……	……	……	……	……	……
content			*vdst_8_16_r[15]*	……	……	……	……	……	……	……
address	……	……	0x9	……	0x5	0x4	0x3	0x2	0x1	0x0
content	……	……	*vdst_8_16_r[3]*	……			*vdst_8_16_r[1]*			*vdst_8_16_r[0]*

Colors of G and B are stored in a similar way as R, the process flow of R can be a reference for G and B.

3. Software and Hardware Requirements

3.1 Software Requirements

- BSP version
  - M55M1_Series_BSP_CMSIS_V3.00.001
- IDE version
  - Keil uVersion 5.28

3.2 Hardware Requirements

- Circuit components
  - NuMaker-M55M1(Beta V1.1)
- Schematic diagram of the route
  - as shown in Figure 3-1, UART0 TX (PB.13) pin is connected to Nu_Link Me. It is then connected to your PC via VCOM. Set baud-rate to 115200bps, and check the execution result of the sample code.

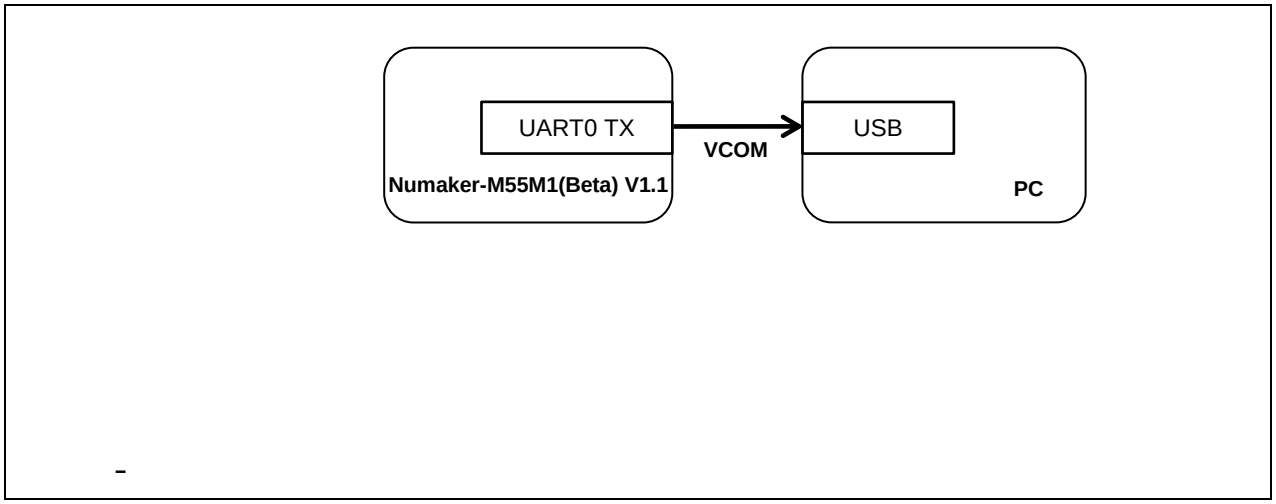

Figure 3-1 Pin Connection Diagram

4. Directory Information

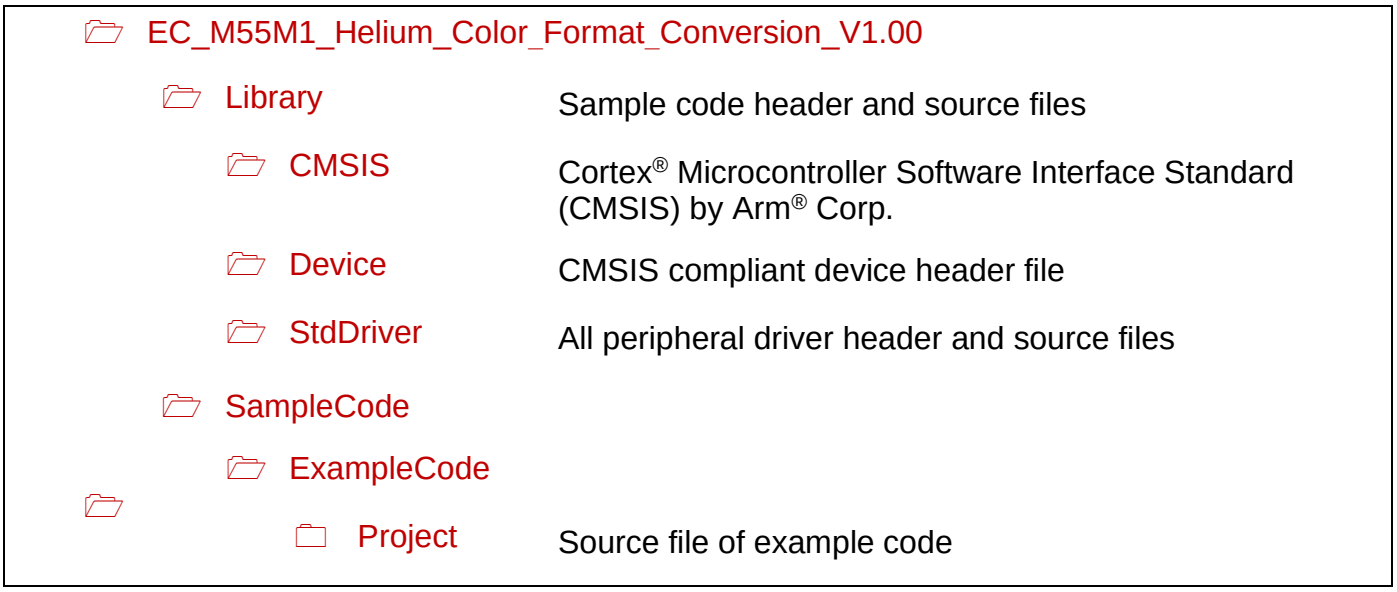

Figure 4-1 Directory Structure

5. Example Code Execution

1. Browse the sample code folder as described in the Directory Information section and double-click *M55M1_Helium_Color_Format_Conversion.uvprojx*.
2. Enter Keil compile mode.
   - Build
   - Download
   - Start/Stop debug session
3. Enter debug mode.
   - Run

6. Revision History

Date	Revision	Description
2024.06.21	1.00	Initial version.

Important Notice

Nuvoton Products are neither intended nor warranted for usage in systems or equipment, any malfunction or failure of which may cause loss of human life, bodily injury or severe property damage. Such applications are deemed, "Insecure Usage".

Insecure usage includes, but is not limited to: equipment for surgical implementation, atomic energy control instruments, airplane or spaceship instruments, the control or operation of dynamic, brake or safety systems designed for vehicular use, traffic signal instruments, all types of safety devices, and other applications intended to support or sustain life.

All Insecure Usage shall be made at customer's risk, and in the event that third parties lay claims to Nuvoton as a result of customer's Insecure Usage, customer shall indemnify the damages and liabilities thus incurred by Nuvoton.

Please note that all data and specifications are subject to change without notice.
All the trademarks of products and companies mentioned in this datasheet belong to their respective owners.