Embedded System Final Report

Topic: Computer Knob with FOC Controlled Gimbal Motor

Embedded System Lab final

https://www.canva.com/design/DAGHLpvsHCQ/UREpAiY853gU0Op0jrSfrw/edit?utm_content=DAGHLpvsHCQ&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton

Motivation

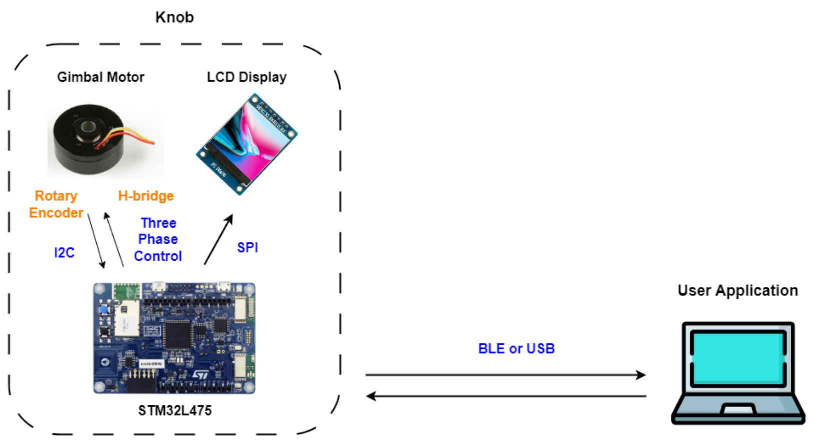
Our project is motivated from the following video:

https://www.youtube.com/watch?v=ip641WmY4pA

However, instead of being only a knob, we design it to be a computer knob which can be used to control some settings on user's computer.

Project Structure

The following figure is our system diagram:



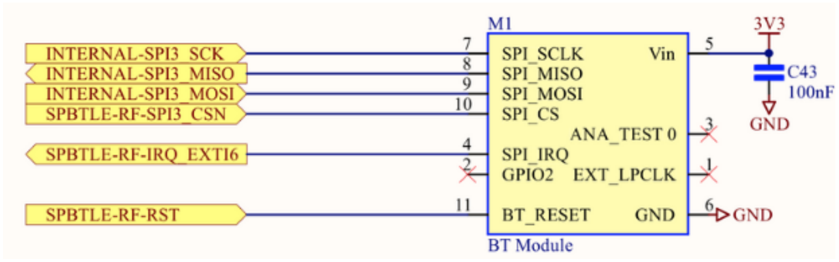
System diagram of our project

We utilize STM B-L475E-IOT01A as our main MCU. For controlling the knob, which is a gimbal motor, the rotary encoder on the the motor will read the position of the motor, and send the data to the STM board using I2C. The STM board will then process the data and send three-phase control signal to control the motor. As for connecting with user application(i.e. computer), we can use two methods, one is using USB HID(Human Interface Device) for wired connection, and the other is using BLE(Bluetooth Low Energy) for wireless communication. Additionally, we also have a LCD display to show what we are controlling currently. The STM board draws the pixels on the LCD display through SPI.

GATT Server Implementation on STM32

HCI command

We used SPI to send HCI command to interface with the BLE on board module.



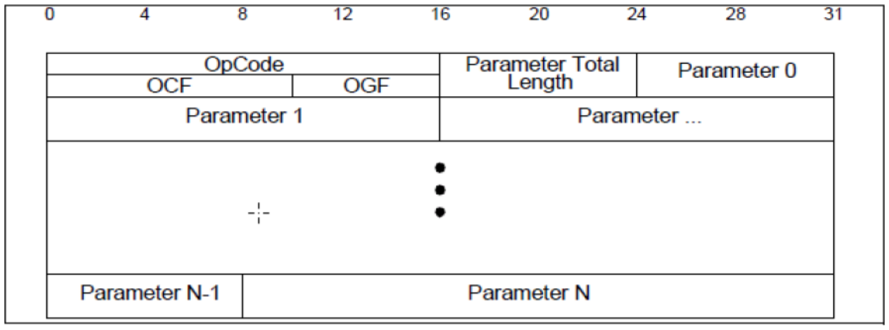
SPI connection to BT Module.

According to [1], the HCI is a standardized Bluetooth interface to bridge the host and controller devices. It is used for sending commands, receiving events, and for sending and receiving data.

The HCI supports four types of packets:

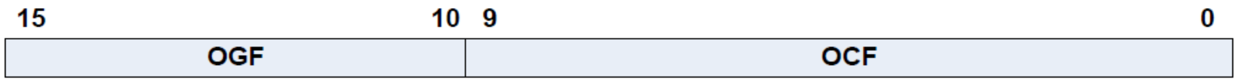
Packet	**Packet Type**
Command	0x01
Asynchronous Data	0x02
Synchronous Data	0x03
Event	0x04
Extended Command	0x09

Each packet have different contents, for example, the structure of a command packet is shown in the figure below:



Structure of a HCI command packet[1]

The OpCode is subdivided into two parts: OpCode Group Field (OGF) and OpCode Command Field (OCF).



Structure of HCI command OpCode[1]

For BLE, the OGF values are:

Command Group	Value
Link Control Commands	1
Link Policy Commands	2
Controller and Baseband Commands	3
Informational Parameters	4
Status Parameters	5
Testing Commands	6

LE Only Commands	8
Vendor Specific Commands	63

Implementation

HCI Commands

The OpCode and parameters for HCI commands are defined in `/BLE/ble_commands.h`.

For example, adding a custom service is defined as:

```
uint8_t ADD_CUSTOM_SERVICE[]={0x01,0x02,0xFD,0x13,0x02,0x00,0x00,0x00,0x00,0x00,0x00,0x01,0x00,0
```

The parameters for the properties of GATT characteristics are defined by:

```
#define READABLE 0x02
#define NOTIFIBLE 0x10
#define WRITABLE  0x04
```

The function `BLE_command(uint8_t *command, int size, uint8_t *result, int sizeRes, int returnHandles)` , defined in `/BLE/enable.c` , will send the HCI command to the Bluetooth module via SPI.

Services and Characteristics

We defined the UUID for our GATT service to be:

```
uint8_t UUID_ANGLE_SERVICE[] = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x00
```

The UUID of our characteristic is defined as:

```
uint8_t UUID_CHAR_ENCODER[] = {0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x03, 0x00, 0x00, 0x00, 0x00,
```

During the initialization phase of STM32, function `ble_init()` adds the service and characteristic using the self-defined functions `addService` and `addCharacteristic`:

```
addService(UUID_ANGLE_SERVICE, ANGLE_SERVICE_HANDLE, SET_ATTRIBUTES(1 + 2 + 3 * 1));
addCharacteristic(UUID_CHAR_ENCODER, ENCODER_CHAR_HANDLE, ANGLE_SERVICE_HANDLE, SET_CONTENT_LENG
```

The functions combine the information into a HCI command and send it using the function `BLE_command` mentioned above.

Update Value

In `main.c`, when the value (`Partition_Inx`) of the knob changes, we call the function `updateSignedMillesimal` to update the value of GATT characteristic.

```
updateSignedMillesimal(ANGLE_SERVICE_HANDLE, ENCODER_CHAR_HANDLE, ENCODER_VALUE, 10, Partition_I
```

The function transform a signed integer into the format of: `{"Angle":"±0000"}` and send a HCI command to update the value.

GATT Client Implementation on Windows11

Windows APIs

Windows APIs are a set of functions, procedures, and protocols provided by the Microsoft Windows operating system to allow software developers to create and interact with applications that run on the Windows platform.

While the programming language for Windows APIs can be C or C#, most documentation and tools for the Bluetooth APIs are for C#. Therefore, we used C# to build our application.

Requirements and Setup using .NET CLI

.NET is a framework that supports building and running Windows apps and web services. To start a C# project with .NET command line interface (CLI), we can follow the steps:

1. For starters, we need to install .NET8.0 software develop kit (SDK) on the website: https://dotnet.microsoft.com/en-us/download. To check whether the installation is successful, we can use the command: `dotnet --list-sdks`.
2. To generate a C# project, ew can use the command `dotnet new console --framework net8.0 --use-program-main`
3. We are writing a Windows11 application, we need to specify the target framework to be `net8.0-windows10.0.22000.0` in the `.csproj` file.

   💡 Target frameworks for different versions of Windows

   - **windows10.0.17763.0**: Windows 10, version 1809
   - **windows10.0.18362.0**: Windows 10, version 1903
   - **windows10.0.19041.0**: Windows 10, version 2004
   - **windows10.0.22000.0**: Windows 11

BLE GATT Client

Setup

The API we are using is Windows.Devices.Bluetooth, which is in the Windows Runtime (WinRT) projection support package. To install the package, we can use the command `dotnet add package Microsoft.Windows.CsWinRT --version 2.0.7`.

Scanning BLE advertisements (code in `/utility/AdvertisementScanner.cs`)

Using the `BluetoothLEAdvertisementWatcher` class provided by the `Windows.Devices.Bluetooth.Advertisement` API, we can add a handler function `OnAdvertisementReceived` to be called whenever a Bluetooth device is scanned.

```
BluetoothLEAdvertisementWatcher watcher = new BluetoothLEAdvertisementWatcher();
watcher.Received += OnAdvertisementReceived;
watcher.Start();
```

The `Received` event passes the arguments of the scanned Bluetooth advertisement. The arguments includes the complete local name, address, signal strength ... etc. We can use the arguments to identify the Bluetooth devise we want to connect. In our project, we scan for the devise with complete local name being `STM32`. Once we connect to the devise, we remove the handler function to stop the scanning process.

```
var localName = eventArgs.Advertisement.LocalName;
var address = eventArgs.BluetoothAddress;
var rssi = eventArgs.RawSignalStrengthInDBm;
// Print the result
Console.WriteLine($"Device: {localName}, Address: {address}, RSSI: {rssi} dBm");
if (localName == "STM32")
{
    BluetoothLEDevice bluetoothLeDevice = await BluetoothLEDevice.FromBluetoothAddressAsync(addr
    watcher.Received -= OnAdvertisementReceived;
}
```

Acquiring GATT characteristic values (code in `/utility/GetBLEData.cs`)

After connecting to STM32, we can get the desired GATT characteristic using `Windows.Devices.Bluetooth` and `Windows.Devices.Bluetooth.GenericAttributeProfile` APIs.

The UUID we set in the STM32 is shown below:

```
Guid serviceUuid = new Guid("00055005-0000-0000-0001-000000000000");
Guid characteristicUuid = new Guid("00055000-0000-0000-0003-000000000001");
```

We can get all the GATT services and select the desired service by its UUID.

```
GattDeviceServicesResult serviceResult = await bluetoothLeDevice.GetGattServicesAsync();
var services = serviceResult.Services;
foreach (var service in services)
{
    if (service.Uuid != serviceUuid) { continue; }
    gattService = service;
```

We can get the GATT characteristics in a similar manner;

```
GattCharacteristicsResult characterResult = await gattService.GetCharacteristicsAsync();
var characteristics = characterResult.Characteristics;
foreach (var characteristic in characteristics)
{
    if (characteristic.Uuid != characteristicUuid) { continue; }
    gattCharacteristic = characteristic;
```

After getting the characteristic we want, we can act based on its property. For instance, if the characteristic is notifiable, we can subscribe to it by writing the CCCD of the characteristic and adding a handler function.

```
GattCharacteristicProperties properties = gattCharacteristic.CharacteristicProperties;
if (properties.HasFlag(GattCharacteristicProperties.Notify))
{
    GattCommunicationStatus status = await characteristic.WriteClientCharacteristicConfiguration
        GattClientCharacteristicConfigurationDescriptorValue.Notify);
    if (status == GattCommunicationStatus.Success)
    {
        characteristic.ValueChanged += Characteristic_ValueChanged;
    }
}
```

The handler function reads the bytes from the buffer and translate them into a string, then extract the value we want.

```
var reader = DataReader.FromBuffer(args.CharacteristicValue);
byte[] input = new byte[reader.UnconsumedBufferLength];
reader.ReadBytes(input);
string str = Encoding.ASCII.GetString(input);
int angleValue = ExtractValue(str);
```

Volume Control Utility

Apart from the Bluetooth APIs, Windows APIs allow us to program various utilities for our PC. In this project, we used the `AudioSwitcher.AudioApi.CoreAudio` API, which can be installed by command `dotnet add package AudioSwitcher.AudioApi.CoreAudio --version 3.0.3`, to control the volume of our speaker.

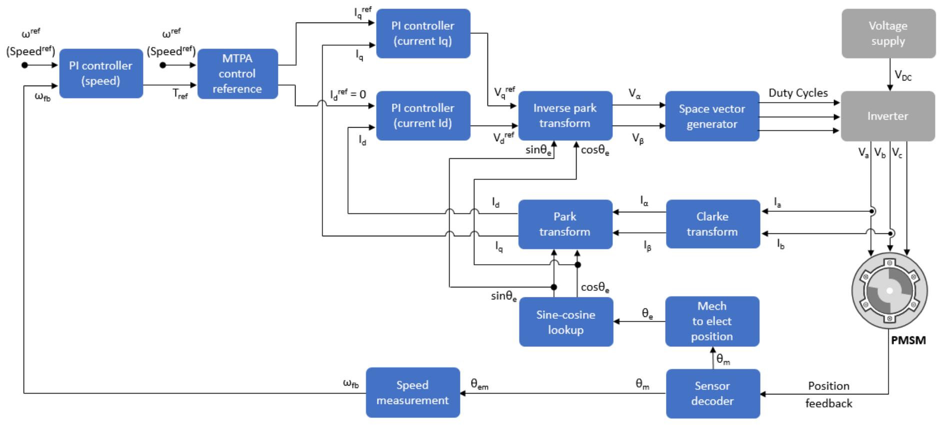
The implementation is quite simple. By getting the `DefaultPlaybackDevice` of the computer, we can easily set the volume of the computer.

```
private static CoreAudioController audioController = new CoreAudioController();
private static CoreAudioDevice playbackDevice = audioController.DefaultPlaybackDevice;

public static void SetVolume(int value)
{
    playbackDevice.Volume = value;
}
```

FOC(Field Oriented Control) motor control

Here is a block diagram of FOC three phase motor control scheme



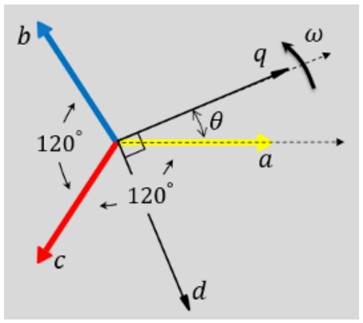
We have implemented the core blocks (Park, Clark, position feedback) but not the current control loop because it requires three channel high precision ADC, which would cost a lot of money but you won't get much performance improved.

Park transform

The Park Transform block converts the time-domain components of a three-phase system in an *abc* reference frame to direct, quadrature, and zero components in a rotating reference frame. The block can preserve the active and reactive powers with the powers of the system in the *abc* reference frame by implementing an invariant version of the Park transform. For a balanced system, the zero component is equal to zero.

The Park Transform block implements the transform for an *a*-phase to *q*-axis alignment as

$$\begin{bmatrix} d \\ q \\ 0 \end{bmatrix} = \frac{2}{3} \begin{bmatrix} \sin(\theta) & \sin(\theta - \frac{2\pi}{3}) & \sin(\theta + \frac{2\pi}{3}) \\ \cos(\theta) & \cos(\theta - \frac{2\pi}{3}) & \cos(\theta + \frac{2\pi}{3}) \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

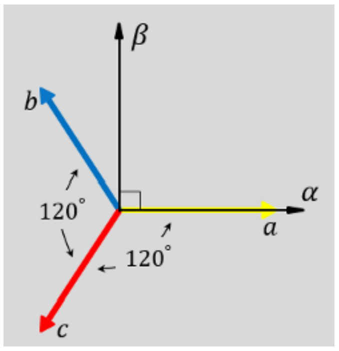


Clarke transform

The Clarke Transform block computes the Clarke transformation of balanced three-phase components in the *abc* reference frame and outputs the balanced two-phase orthogonal components in the stationary *αβ* reference frame. Alternatively, the block can compute Clarke transformation of three-phase components *a*, *b*, and *c* and output the components *α*, *β*, and 0.

The following equation describes the Clarke transform computation:

$$\begin{bmatrix} f_\alpha \\ f_\beta \\ f_0 \end{bmatrix} = \left(\frac{2}{3}\right) \times \begin{bmatrix} 1 & -\frac{1}{2} & -\frac{1}{2} \\ 0 & \frac{\sqrt{3}}{2} & -\frac{\sqrt{3}}{2} \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \end{bmatrix} \begin{bmatrix} f_a \\ f_b \\ f_c \end{bmatrix}$$



PID control

Continuous time PID control formula can be written as

$$U(t) = k_p(err(t) + \frac{1}{T_I}\int err(t)dt + \frac{T_D derr(t)}{dt})$$

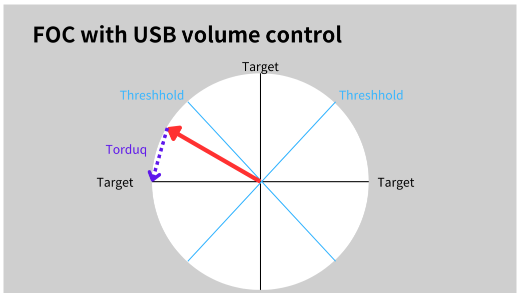
In discrete time is

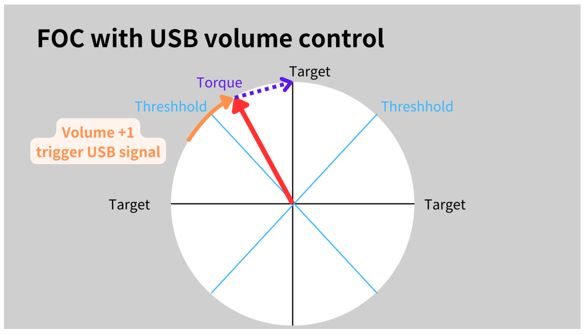
$$u(k) = K_p e(t) + K_i \sum_{n=0}^{k} e(n) + K_d(e(k) - e(k-1))$$

This is how we implement it in STM32, and with that, simulate a segmented dial.

```
void ClosedLoopPosition(float _Kp, float _Ki, float _Kd, float TargetAngle)
{
    Error = TargetAngle - bsp_as5600GetWrappedAngle();
    integral += Error;
    derivative = Error - Error_last;
    float Uq = _constrain(DIR * (_Kp * Error +
                                 _Ki * integral +
                                 _Kd * derivative),
                          -voltage_power_supply / 2, voltage_power_supply / 2);

    setPhaseVoltage(Uq, 0, _electricalAngle());
}
```





Here is the implementation of the inverse transforms

```
void setPhaseVoltage(float Uq, float Ud, float phase_angle)
{
    Ualpha = -Uq * sin(phase_angle);
    Ubeta = Uq * cos(phase_angle);

    Ua = Ualpha + voltage_power_supply / 2;
    Ub = (sqrt(3) * Ubeta - Ualpha) / 2 + voltage_power_supply / 2;
    Uc = (-Ualpha - sqrt(3) * Ubeta) / 2 + voltage_power_supply / 2;
    setPwm(Ua, Ub, Uc);
}
```

The way we use high frequency PWM (4k Hz) to emulate sine signal is by directly assign value to timer autoreload registor, changing the duty cycle.

```
void setPwm(float Ua, float Ub, float Uc)
{
    float dc_a = _constrain(Ua / voltage_power_supply, 0.0f, 1.0f);
    float dc_b = _constrain(Ub / voltage_power_supply, 0.0f, 1.0f);
    float dc_c = _constrain(Uc / voltage_power_supply, 0.0f, 1.0f);
    TIM3->CCR1 = round(dc_a * 10000);
    TIM3->CCR3 = round(dc_b * 10000);
    TIM3->CCR4 = round(dc_c * 10000);
}
```

Position feedback

It is done by the AS5600 magnetic rotary encoder in I2C. Note that full_rotation_offset records how many full rotations have been make. The way to determine full rotation is to set a threshold at 0.8pi over which we add 1 to full_rotation_offset.

```
int16_t bsp_as5600GetWrappedRawAngle(void)
{
    int16_t angle_data = bsp_as5600GetRawAngle();
    int16_t d_angle = angle_data - angle_data_prev;
    if ((float)abs(d_angle) > (float)(0.8 * AS5600_RESOLUTION))
    {
        full_rotation_offset += (d_angle > 0 ? -1 : 1);
    }
    angle_data_prev = angle_data;
    return (int16_t)(full_rotation_offset * AS5600_RESOLUTION + angle_data);
}
```

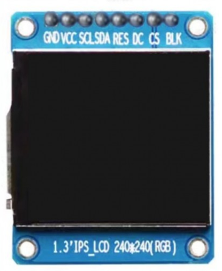
USB volume control

We wrote a custom HID report descriptor as if the deviec is a consumer comtroller with volume control capability. Note that report descriptor is an established protocol so it is done by look up.

```
        0x05, 0x0C, // Usage Page (Consumer)
        0x09, 0x01, // Usage (Consumer Control)
        0xA1, 0x01, // Collection (Application)
        0xA1, 0x00, //   Collection (Physical)
        0x09, 0xE9, //     Usage (Volume Increment)
        0x09, 0xEA, //     Usage (Volume Decrement)
        0x09, 0xE2, //     Usage (Mute)
        0x09, 0xCD, //     Usage (Play/Pause)
        0x35, 0x00, //     Physical Minimum (0)
        0x45, 0x07, //     Physical Maximum (7)
        0x15, 0x00, //     Logical Minimum (0)
        0x25, 0x01, //     Logical Maximum (1)
        0x75, 0x01, //     Report Size (1)
        0x95, 0x04, //     Report Count (4)
        0x81, 0x02, //     Input (Data,Var,Abs,No Wrap,Linear,Preferred State,No Null Position)
        0x75, 0x01, //     Report Size (1)
        0x95, 0x04, //     Report Count (4)
        0x81, 0x01, //     Input (Const,Array,Abs,No Wrap,Linear,Preferred State,No Null Positio
        0xC0,       //   End Collection
        0xC0,       // End Collection
```

LCD Display

We use ST7789 IPS LCD as our LCD display. The function of the pins are described below:



Pin	Function
GND	ground
VCC	power(3.3V)
SCL	SPI clock
SDA	SPI data
RES	reset when low
DC	low: control mode high: data mode

CS	low: activate mode high: idle mode
BLK	backlight control

The connection between ST7789 and STM board is shown in the following table:

ST7789	STM32
GND	GND
VCC	3.3V
SCL	ARD D13(SPI1_SCLK)
SDA	ARD D11(SPI1_MOSI)
RES	ARD D8
DC	ARD D9
CS	ARD D10
BLK	3.3V

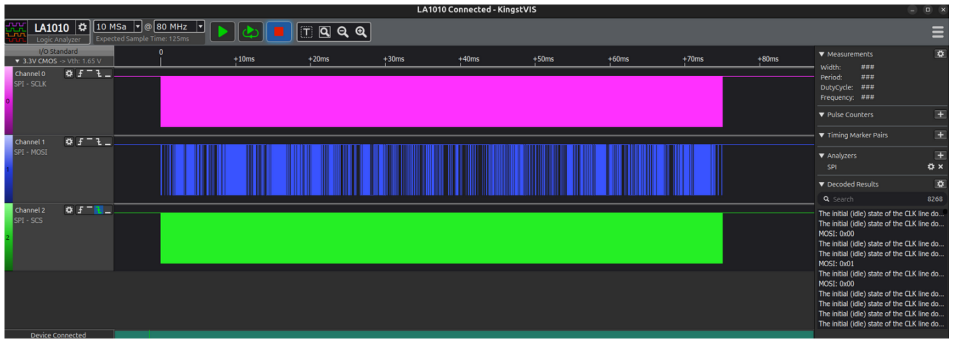
We use the library[2] to control the ST7789 LCD display. The display uses SPI to transmit data. For drawing string to the display, we use the following code:

```
if (VOL_FLAG != 0)
{
    sprintf(msg, "Vol: -", Partition_Inx);
    ST7789_WriteString(10, 100, msg, Font_16x26, WHITE, BLACK);  // write string
}
```

Also, since drawing a large amount of data is quite slow in the original library, we modify the library to enable frame buffer DMA.

```
if (DMA_MIN_SIZE <= buff_size)
{
    HAL_SPI_Transmit_DMA(&ST7789_SPI_PORT, buff, chunk_size);
    while (ST7789_SPI_PORT.hdmatx->State != HAL_DMA_STATE_READY)
    {
        // transmit data
    }
}
```

With DMA, the transmit time can be reduced about 70ms, without CPU participating in the process.



```
Width:      ###
Period:     ###
DutyCycle:  ###
Frequency:  ###
8268
The initial (idle) state of the CLK line do...
The initial (idle) state of the CLK line do...
MOSI: 0x00
The initial (idle) state of the CLK line do...
The initial (idle) state of the CLK line do...
MOSI: 0x01
The initial (idle) state of the CLK line do...
MOSI: 0x00
The initial (idle) state of the CLK line do...
The initial (idle) state of the CLK line do...
The initial (idle) state of the CLK line do...
The initial (idle) state of the CLK line do...
```

One frame data transfer

Reference

[1] https://software-dl.ti.com/simplelink/esd/simplelink_cc13×2_sdk/1.60.00.29_new/exports/docs/ble5stack/vendor_specific_guide/BLE_Vendor

[2] https://github.com/Floyd-Fish/ST7789-STM32