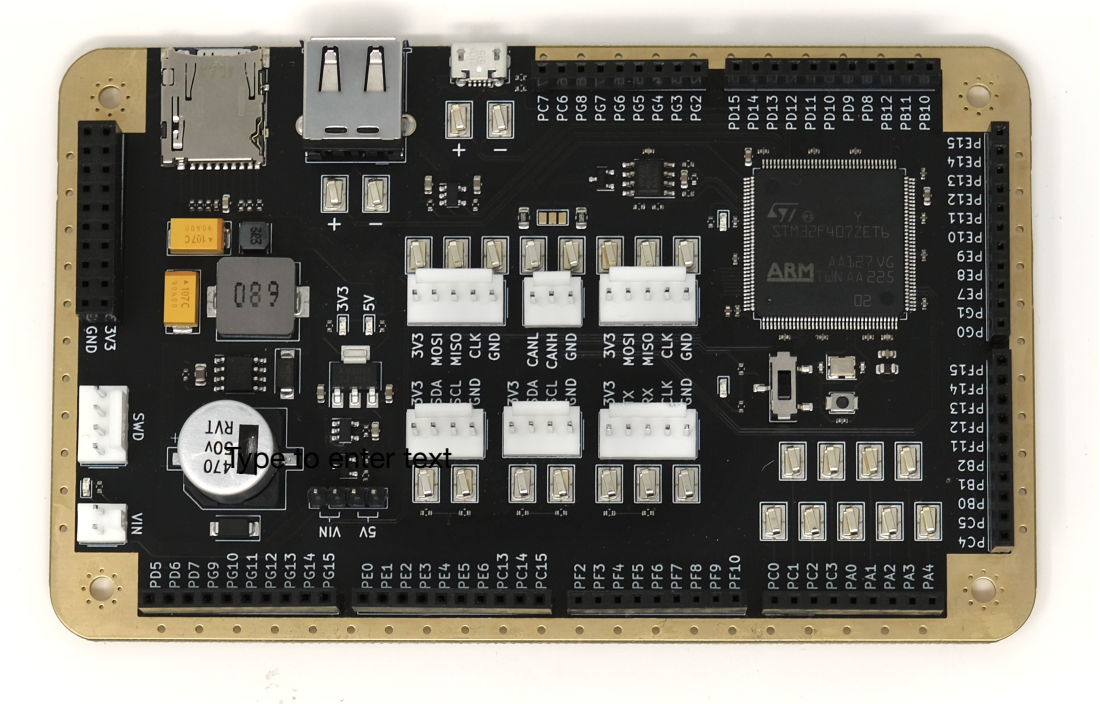
STM32 Skipper User Guide



Main Features:

- Processor - (STM32F407ZE)
- 79 dedicated GPIO (up to 94)
- 2 USB interfaces(1 device, 1 host)
- 4 bit microSD(Tf) card interface
- 2 dedicated SPI interfaces
- 2 dedicated I2C interfaces
- 1 dedicated CAN interface
- 1 dedicated U(S)ART interface
- 26 Oscilloscope probe clips
- Perimeter ground plane
- High-voltage SMPS
- Multiple power sources
- 7-40v input(VIN)
- 5v input(USB, 5V pin)
- 3.3v Input(3V3 pin)
- 5v 2A Output
- 3.3v 1A Output
- Arduino IDE compatible
- STM32Cube IDE compatible
- SWD interface

Table of Contents

Pin Descriptions

Register A

PA0	GPIO
PA1	GPIO
PA2	GPIO
PA3	GPIO
PA4	GPIO
PA5	SPI_1_CLK
PA6	SPI_1_MISO
PA7	SPI_1_MOSI
PA8	U(S)ART_1_CLK
PA9	USB_1_EN
PA10	USB_1_FLG
PA11	USB_1_D-
PA12	USB_1_D+
PA13	SWDIO
PA14	SWCLK
PA15	SD_DETECT

Register B

PB0	GPIO
PB1	GPIO
PB2	GPIO
PB3	SPI_3_CLK
PB4	SPI_3_MISO
PB5	SPI_3_MOSI
PB6	U(S)ART_1_TX
PB7	U(S)ART_1_RX
PB8	I2C_1_SCL
PB9	I2C_1_SDA
PB10	GPIO
PB11	GPIO
PB12	GPIO
PB13	USB_2_DETECT
PB14	USB_2_D-
PB15	USB_2_D+

Register C

PC0	GPIO
PC1	GPIO
PC2	GPIO
PC3	GPIO
PC4	GPIO
PC5	GPIO
PC6	GPIO
PC7	GPIO
PC8	SD_DAT0
PC9	SD_DAT1
PC10	SD_DAT2
PC11	SD_DAT3
PC12	SD_CLK
PC13	GPIO
PC14	GPIO
PC15	GPIO

Register D

PD0	CAN_1_RX
PD1	CAN_1_TX
PD2	SD_CMD
PD3	CAN_1_RS
PD4	Built In LED
PD5	GPIO
PD6	GPIO
PD7	GPIO
PD8	GPIO
PD9	GPIO
PD10	GPIO
PD11	GPIO
PD12	GPIO
PD13	GPIO
PD14	GPIO
PD15	GPIO

Register E

PE0	GPIO
PE1	GPIO
PE2	GPIO
PE3	GPIO
PE4	GPIO
PE5	GPIO
PE6	GPIO
PE7	GPIO
PE8	GPIO
PE9	GPIO
PE10	GPIO
PE11	GPIO
PE12	GPIO
PE13	GPIO
PE14	GPIO
PE15	GPIO

Register F

PF0	I2C_2_SDA
PF1	I2C_2_SCL
PF2	GPIO
PF3	GPIO
PF4	GPIO
PF5	GPIO
PF6	GPIO
PF7	GPIO
PF8	GPIO
PF9	GPIO
PF10	GPIO
PF11	GPIO
PF12	GPIO
PF13	GPIO
PF14	GPIO
PF15	GPIO

Register G

PG0	GPIO
PG1	GPIO
PG2	GPIO
PG3	GPIO
PG4	GPIO
PG5	GPIO
PG6	GPIO
PG7	GPIO
PG8	GPIO
PG9	GPIO
PG10	GPIO
PG11	GPIO
PG12	GPIO
PG13	GPIO
PG14	GPIO
PG15	GPIO

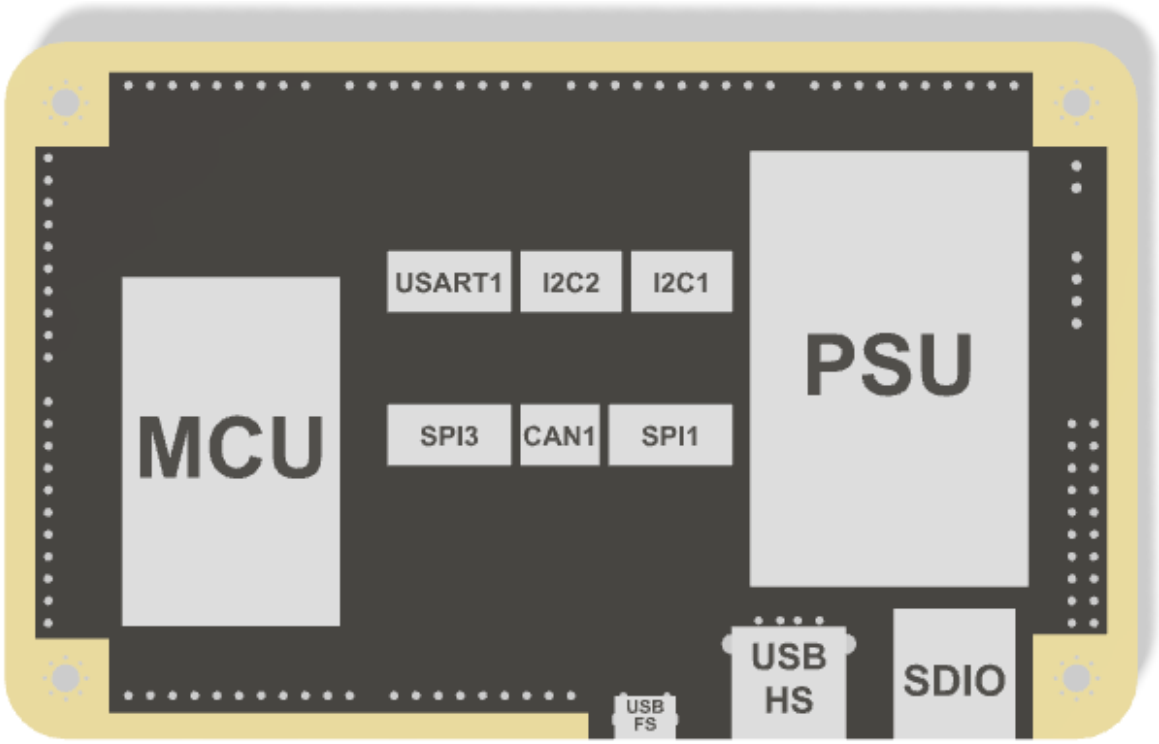
Skipper library

I have developed an STM32cube library specifically for the Skipper. The library includes various classes, and functions designed to make using the Skipper effortless. The Skipper library is available to download on the *Skipper GitHub page*. In addition there are various examples and templates to help get you started

I2C Module

The STM32 Skipper Library includes an easy way to get started with I2C. The I2C module includes all the configuration settings you need to activate, update, and control the two dedicated I2C ports on Skipper.

Fig.1



I2C1, and I2C2 ports can be seen in the above photo

The STM32 Skipper has two dedicated I2C ports. The ports are called I2C1, and I2C2. The Skipper library is able to control either one, or both of these ports independently. This way, you can communicate simultaneously on two buses, even with two different configurations.

Creating an I2C instance

In order to create, and start using an I2C peripheral, you must create an instance. In order to do this, you can use the

[I2C *instanceName*;]

replace *instanceName* with whatever you want that instance to be called. This name will be used every time you want to use the I2C peripheral.
eg-

[*instanceName*.write(0x43);]

Assigning a port

You must assign the physical port to be used for your I2C instance. You can either pick (I2C1), or (I2C2). The way you do this is with the assignPort function. The implementation looks like this.

[*instanceName.*assignPort(I2C1);]
Or,
[*instanceName.*assignPort(I2C2);]

To see the port placement on the board, look at fig.1.

If you do not select a port, it will default to I2C1.

Select mode

I2C can work in one of two modes, Master, or Slave.

In Master mode, Skipper provides the clock and slave address of the device to be accessed. This way Skipper is in full control of the device, choosing when communicate, and which device to communicate with.

In Slave mode, Skipper will wait until a master calls its address to initiate the data transfer. The slave address selection is described later in this document.

To select master mode,
[*instanceName.*mode = 1;]
To select slave mode,
[*instanceName.*mode = 0;]

If no mode is provided, it will default to master mode.

Select slave address (slave mode only)

In slave mode, Skippers address can be selected using the address variable. The address is an 8 bit number used to differentiate Skipper from other devices on the bus.

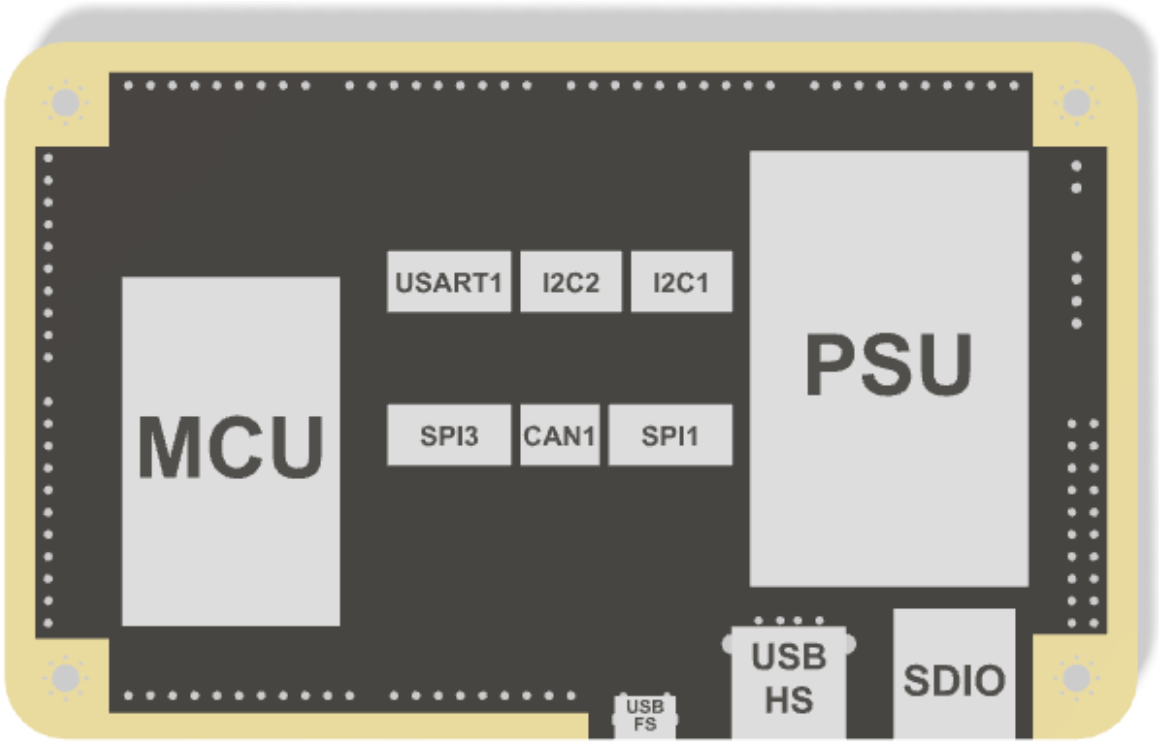
To set the address to, example (0b01010101),
[*instanceName.*slaveAddress = 0b01010101;]

If no slave address is provided, it will default to (0b10011001)

SPI Module

The STM32 Skipper Library includes an easy way to get started with SPI. The SPI module includes all the configuration settings you need to activate, update, and control the two dedicated SPI ports on Skipper.

Fig.1



I2C1, and I2C2 ports can be seen in the above photo

The STM32 Skipper has two dedicated SPI ports. The ports are called SPI1, and SPI3. The Skipper library is able to control either one, or both of these ports independently. This way, you can communicate simultaneously on two buses, even with two different configurations.

Creating an SPI instance

In order to create, and start using an SPI peripheral, you must create an instance. In order to do this, you can use the

[SPI *instanceName*;]

replace *instanceName* with whatever you want that instance to be called. This name will be used every time you want to use the SPI peripheral.
eg-

[*instanceName*.transmit(0x43, 2);]

Assigning a port

You must assign the physical port to be used for your SPI instance. You can either pick (SPI1), or (SPI3). The way you do this is with the assignPort function. The implementation looks like this.

[*instanceName.*assignPort(SPI1);]
Or,
[*instanceName.*assignPort(SPI3);]

To see the port placement on the board, look at fig.1.

If you do not select a port, it will default to SPI1.

Select speed (master mode only)

The SPI protocol is pretty flexible in terms of the frequency with the limiting factor being the max speed of the slave. The best way to find the max speed of the slave is to read the data sheet of the slave IC. If you are communicating with multiple different slaves, choose the max frequency of the slowest one.

The SPI peripheral derives the frequency from the clock source used for that specific peripheral(PCLK). For SPI1, the clock is the APB2 peripheral clock, and for SPI3, the clock is the APB1 peripheral clock.

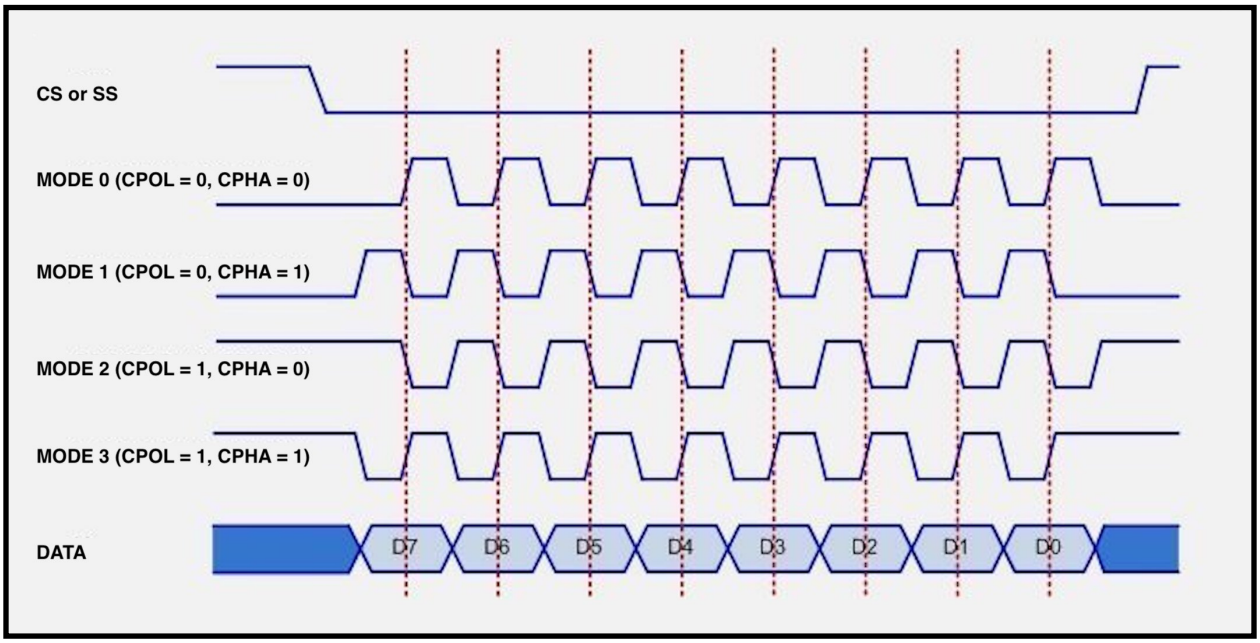
The frequency is further divided by 2, 4, 8, 16, 32, 64, 128, or 256. The value of this divider can be set using the setClockDiv function, the parameter should be set to:
0 for /2,
1 for /4,
2 for /8 etc.

Example: [*instanceName*.setClockDiv(2);] Sets the divider to 8.

The formula for the SPI frequency is, (PCLK / divider)

If PCLK = 16000000, and divider = 8, the SPI clock frequency will be 2Mhz.

Select mode



The SPI protocol describes four possible clock modes. These modes define the polarity of the clock, and at which point the data line is sampled. CPOL, or clock polarity sets weather the clock is HIGH, or LOW in the idle state. CPHA, or clock phase describes whether the data line is sampled on the rising, or falling edge. You can find the SPI mode of the slave, or host device by referencing the devices data sheet.

You can select the mode for the SPI module by using the setMode function.

Example: [*instanceName*.setMode(2);] Sets the port to mode 2

Set up CS pins

the SPI module and the Skipper library do not control any chip select(CS) pins on Skipper, the user must provide, and control the CS pins to use. A typical setup with one device would look like this.

```
void init_CS(void){                  // sets up an idle HIGH chip select pin on PC0
      RCC->AHB1ENR |= (1<<2);        // enable GPIOC
      GPIOC->MODER |= (1<<0);        // set PC0 to output
      GPIOC->ODR |= (1<<0);          // set PC0 HIGH
}

void CS_enable(void){                // sets CS PC0 LOW
      GPIOC->ODR &= ~(1<<0);         // set PC0 LOW
}

void CS_disable(void){               // sets CS PC0 HIGH
      GPIOC->ODR |= (1<<0);          // set PC0 HIGH
}
```

Using these three functions, you can setup PC0 to be used as CS, enable it, and disable it. You will need one pin for each device.

SPI initialization/de-initialization

Once all the above steps are completed, you can call the init function. The init function takes all the values provided earlier, and enters them into the appropriate peripheral registers. The init function has no parameters.

Example: [*instanceName*.init();] initializes the SPI instance.

If the program would like to change the SPI parameters or is finished with the SPI peripheral, it can call the deInit function. The deInit function clears all the SPI registers, and disables the SPI clock. This function has no parameters.

Example: [*instanceName*.deInit();] de-initializes the SPI instance.

Write to the SPI bus

The transmit function is used to write to the SPI data bus. It has to parameters, the first parameter expects a pointer to an array of uint8_t elements. The second expects the size (in bytes) of the given array. The function will then loop through the array, writing all the bytes into one SPI transfer.

MASTER MODE:

In master mode, the transmit function is used to directly transfer data to the slave, a typical transfer will look something like this.

```
CS_enable();                                // enable chip select by setting it LOW
uint8_t address = 0b00011000;               // create a variable to hold the address
                                            of the device register
uint8_t data[5] = {address, 1, 2, 3, 4};    // create array (address, 1, 2, 3, 4)
uint8_t dataPointer = data;                 // create a pointer to the data array
instanceName.transmit(dataPointer, 5);      // call the transmit function
CS_disable();                               // disable chip select by setting it HIGH
```

In the above code, a data array of length 5 is created, the array includes four bytes of data, and the address(Note that the address is treated the exact same way as the data). the transfer function will then write the data to the SPI data register, and the SPI hardware will handle the physical transfer.

SLAVE MODE:

In slave mode, the write action will typically be preformed within the data interrupt routine. It may look like this.

```
// interrupt routine {
if(retrievedData == 0b00011000){

      instanceName.transmit(dataPointer, 5);    // call the transmit function
}

// }
```

receive from the SPI bus

MASTER MODE:

The receive function is exactly the same as the transmit function, except that it reads each byte and returns a pointer to the received data, It does so by sending dummy bytes (0's) on the MOSI line, and reads the data register to receive each byte on the MISO line.

```
void receive(uint8_t* ptr, uint8_t address, uint8_t size);
```

The receive function takes a pointer to a byte array, an 8 bit address, and a 8 bit size variable, the size refers to the size of the provided array pointer in bytes.

Here is an example of data reception.

```
CS_enable();                                // enable chip select by setting it LOW
uint8_t address = 0b00011000;               // create a variable to hold the address
                                            of the device register
uint8_t data[4];                            // create array to store data
uint8_t dataPointer = data;                 // create a pointer to the data array
instanceName.receive((address+128), dataPointer, 4);  // call the transmit function
CS_disable();                               // disable chip select by setting it HIGH
```

the above code receives four bytes of data from a selected address. note the +128 next to the address variable is used as an example for setting the R/W bit of the address, you should reference the slave devices data sheet for more information. It then will create a four byte array to save the data, a pointer to that array, and sends it to the receive function. The receive function will then send the address followed by (size) bytes of zero's after each byte is sent, it will capture the data from the data register, and save it to the supplied pointer. The data can then be accessed though the created array.

SLAVE MODE: