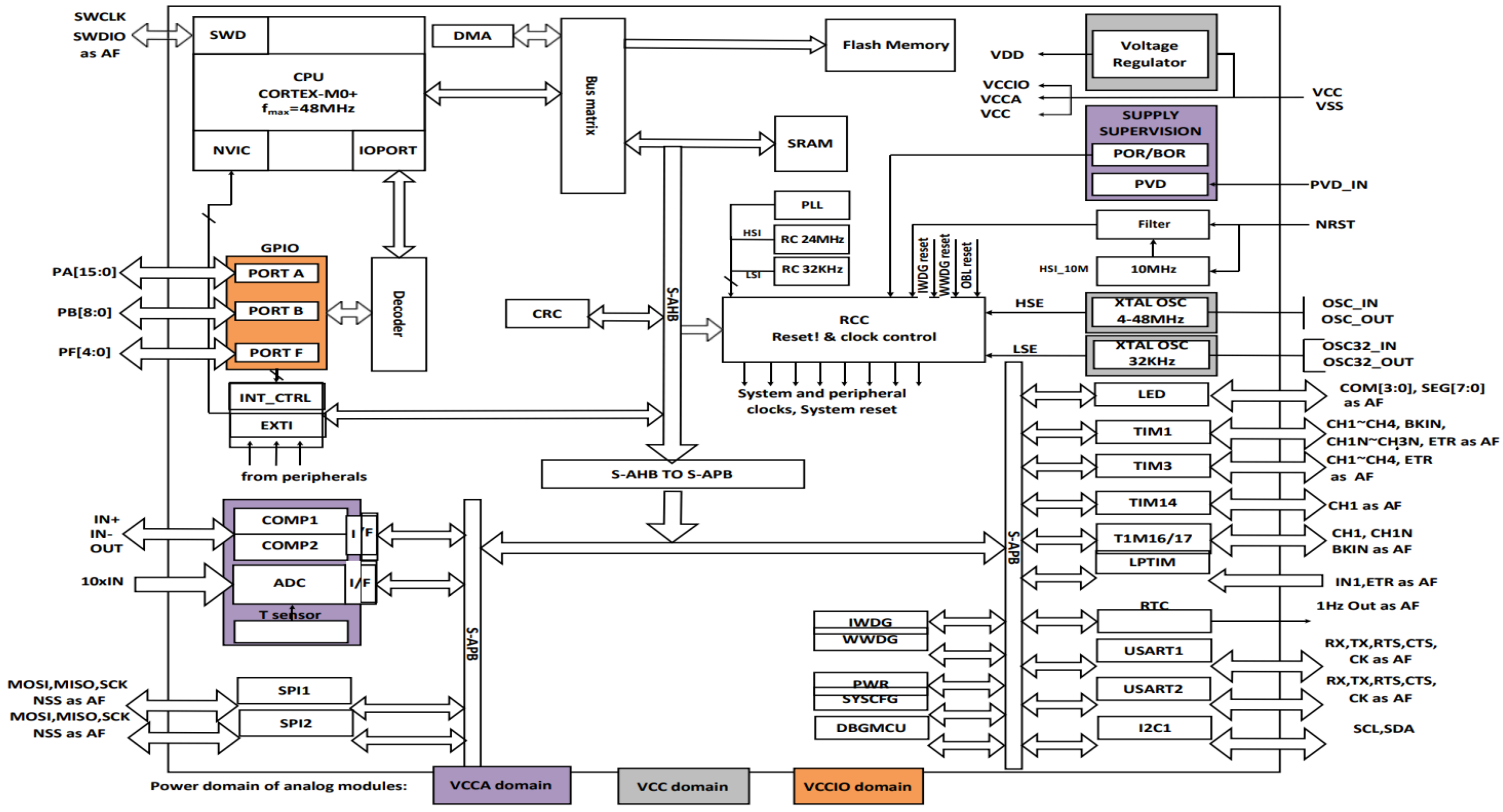
PUYA

PY32 MICROCONTROLLER.



Introduction

Puya manufactures the inexpensive 32-bit microcontrollers PY32F003 and PY32F030, which use ARM Cortex M0+. Puya produces an extremely affordable selection of Arm Cortex-M0+ components in the PY32 series. The series consists of three families: PY32F002A, PY32F003, and PY32F030. The families appear to be completely progressive, despite differences in memory, maximum speed, and peripheral inclusion (for example, firmware designed for the PY32F002A should function on the PY32F003 and PY32F030). Whatever the case, every component is a contemporary design with an operating voltage range of 1.7 to 5.5V and internal regulators.

PY32F003:

PUYA Semiconductor develops microcontrollers in the PY32 series, including the PY32F003. These microcontrollers are renowned for their excellent performance and low power consumption, and they are made for a wide range of uses. These are the PY32F003's salient characteristics:

- Processor: ARM Cortex-M0+ core.

- Maximum clock speed of 48 MHz.

- Memory: SRAM is used for data processing while Flash memory is usually used for storage.

- Peripherals: A number of built-in peripherals, including GPIOs, timers, SPI, I2C, ADC, and more, are included.

- Power: Use low power settings to prolong battery life in applications that are portable.

The PY32F003 is appropriate for use in consumer appliances, industry control, devices for smart homes, and other fields thanks to these qualities. Please feel free to inquire if you need more information on any specific PY32F003 features or if you have any special questions!

PY32F030:

Another microcontroller in the PY32 series from PUYA Semiconductor is the PY32F030. Here are a few of its salient attributes:

- Processor: ARM Cortex-M0 core.

- Maximum clock speed of 48 MHz.

- Memory: Usually consists of SRAM and Flash memory.

- Peripherals: A variety of built-in peripherals are available, including timers, GPIOs, UART, SPI, I2C, ADC, and DAC, among others.

- Power Consumption: It is appropriate for battery-powered applications due to its low power consumption design.

- Package Options: To accommodate diverse design requirements, available in a range of package sizes.

Numerous applications, such as consumer electronics, industrial control systems, smart home devices, and other embedded systems requiring dependable and effective performance, are appropriate for the PY32F030. Tell me if you require further information on any aspect of the PY32F030 or if you have any specific inquiries!

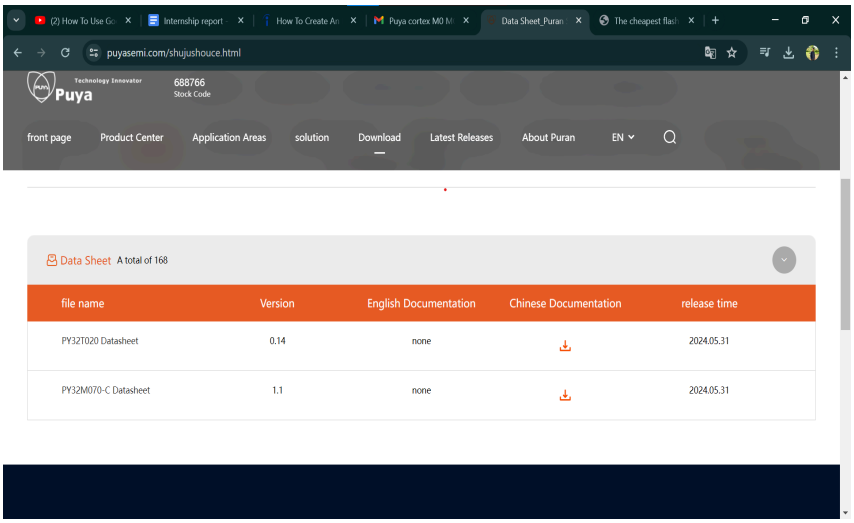
The top of the lineup is represented by the PY32F030. With the addition of a 2x PLL, the core may operate at 48 MHz using the internal oscillator set to 24 MHz. To provide you with up to 30 GPIO pins, the F030 additionally provides compatibility for 32-pin QFP/QFN packages. Apart from the above listed capacities, there is a higher-capacity 64K flash / 8K RAM variant of the F030 available. With the exception of the F030's integrated scanning LED matrix controller, which supports four 8-segment digits, the peripheral set of the F003 and F030 are the same.

Documentation and SDK

- You may download everything you need by visiting Puya's PY32 website, simply scrolling all the way to the right and clicking on the "Datasheet" download (Link: https://www.puyasemi.com/).
- Observe that instead of a datasheet, every single PY32 resource is contained in a ZIP file (yep, each product on the website connects to the same file).
- Datasheets, reference manuals, code samples, and a Keil DFP containing the flash algorithms required to burn your code onto these components using a SWD probe are all included.
- Although only an English reference manual was available for the PY32F030 at the time of writing, the documentation is generally good. Not too big of a deal because you can utilize this same approach for all parts.

Compiling

- Puya's SDK has prepared them for Keil MDK development, much like it has most Arm components from the East.
- Keil MDK may be downloaded for free and used right away, but I needed a solution based on VSCode, Cortex-Debug, and GCC, which required some further tinkering.
- I generated GCC linker config and startup files for each component with a memory map and vector table based on the Keil linker config and startup files, using other GCC Arm Cortex project code I had lying around as a reference. I was able to develop a project by including their STM32 HAL-like peripheral library with a little additional source-file manipulation.



PyOCD

PyOCD is a command-line tool and open-source Python library for programming and debugging Arm Cortex-M microcontrollers.

- It offers developers an easy-to-use interface for controlling and interacting with these devices, facilitating the uploading of firmware, debugging of code, and execution of numerous low-level activities.
- In order to install pyOCD make sure that your system has the latest python version installed.
- Use ‘pip install pyocd’ command on powershell or command prompt to install pyOCD in your system.
- To check whether pyOCD is installed on your desktop, Use ‘pyocd —version’ to check the version of pyOCD you have installed.

OpenOCD

An open-source project called OpenOCD (Open On-Chip Debugger) offers boundary-scan testing, in-system programming, and debugging for embedded systems.

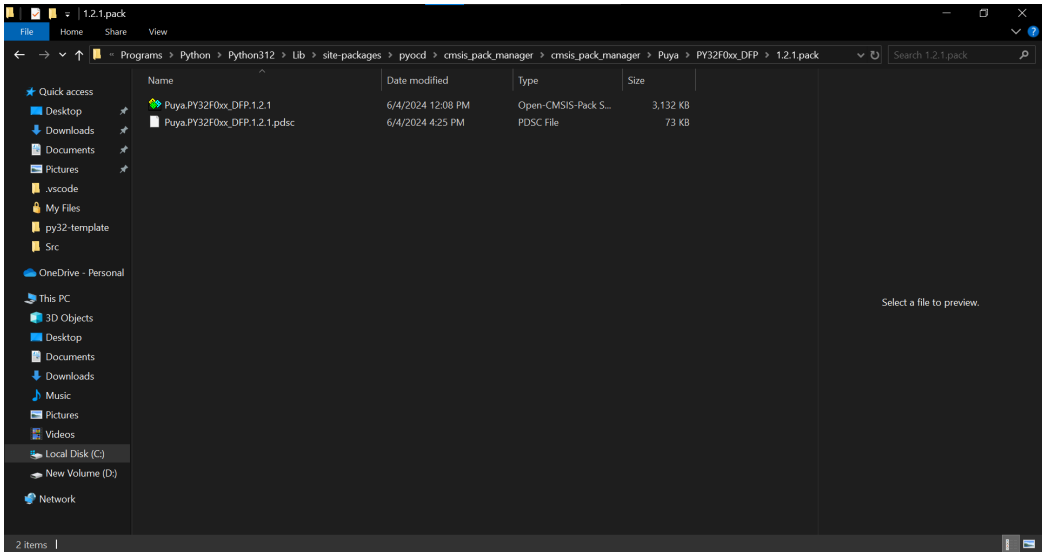
- It provides a flexible tool for embedded systems development by supporting a broad variety of microcontrollers and development boards.

You can download Openocd from xpack or the link given below:

https://xpack.github.io/dev-tools/openocd/releases/

Flashing

- These days, I use two debug servers: pyOCD (which I use with inexpensive CMSIS-DAP probes) and Segger's J-Link GDB Server (clearly for use with their Segger J-Link).



- I tried looking for the Puya pack using pyOCD's search function, but it wasn't found in the repositories, so I had to install it manually. I opened the PY32 DFP (a ZIP file), and extracted it.
- It’s important to rename the file to include the version number in it — Puya.PY32F0xx_DFP.1.2.1.pdsc.
- I then copied the original whole DFP file to the same directory, following the same convention as all the other packs use — copying it to Puya\PY32F0xx_DFP\1.2.1.pack.
- After that, to rebuild the index file to include the new pack,

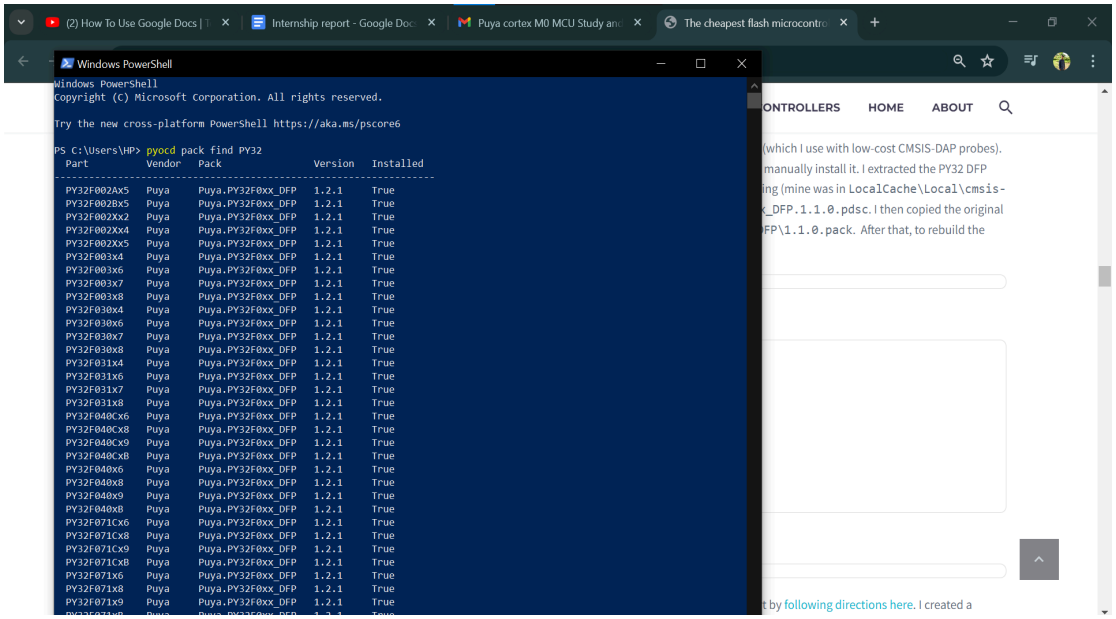
I ran:PS C:\> pack-manager add-packs Puya.PY32F0xx_DFP.1.2.1.pdsc

I checked that pyOCD could now find the pack:

and then installed it:

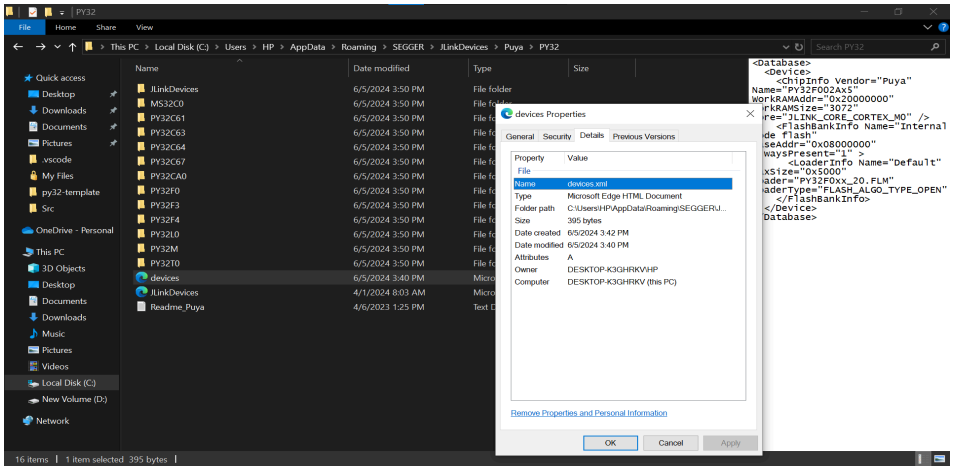
```
PS C:\> pyocd pack install PY32F003x8
```

Actually, obtaining J-Link support required less work. Although Puya PY32 is not supported by Segger, you can add custom device compatibility by following these instructions. After making a folder structure called JLinkDevices\Puya\PY32, I inserted a Devices.xml file.:




```
<Database>
  <Device>
    <ChipInfo Vendor="Puya" Name="PY32F002Ax5" WorkRAMAddr="0x20000000"
WorkRAMSize="3072" Core="JLINK_CORE_CORTEX_M0" />
    <FlashBankInfo Name="Internal code flash" BaseAddr="0x08000000"
AlwaysPresent="1" >
      <LoaderInfo Name="Default" MaxSize="0x5000" Loader="PY32F0xx_20.FLM"
LoaderType="FLASH_ALGO_TYPE_OPEN" />
    </FlashBankInfo>
  </Device>
</Database>
```

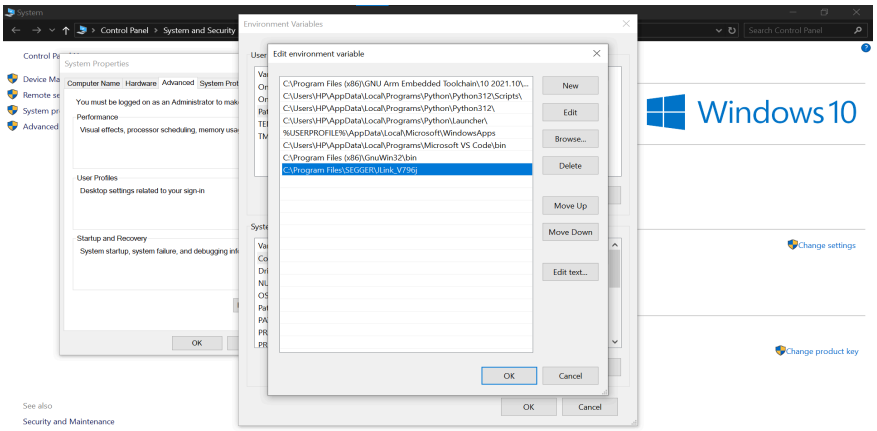
Next, I moved Puya's DFP's FLM flash programming algorithm files into this folder. I was able to see the device in J-Link's target selection box after doing so, and everything functioned well. It should be noted that this requires a relatively current version of J-Link, so confirm that you have updated and that the relevant version is truly installed on your PATH.



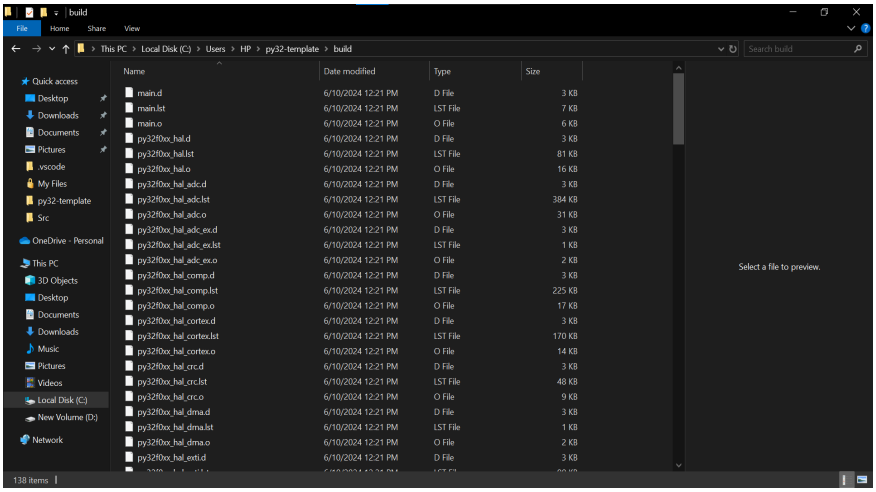
Debugging

Visual Studio Code and the j-link debugger are required for debugging. In order to debug the code, you must first produce the required ELF file.

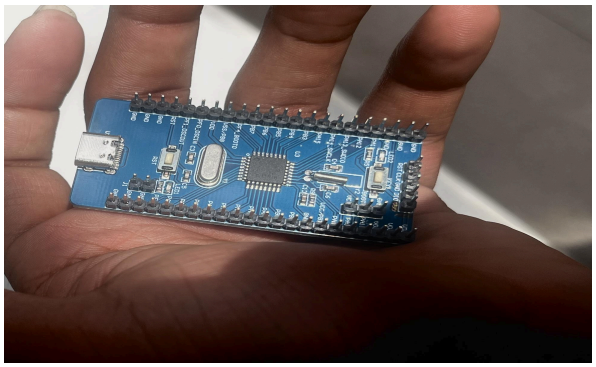
- Make use of the Makefile that comes with the template by creating one. Utilizing Windows Powershell and the make program, create the template. You must get a different “make” on Google if you're running Windows. Download the latest version of the “make” and build the template. After building you can find the ELF files on the build folder present on the template file.
- For connecting j-link with the target device you need to download J-Link on SEGGER website and if you get any error on PATH, you need to manually add the PATH in environment variables.
- Go to This PC>Right click and select properties. Go to Advanced system settings>environment variables and select add and paste your PATH manually on the environment variables and click ok.



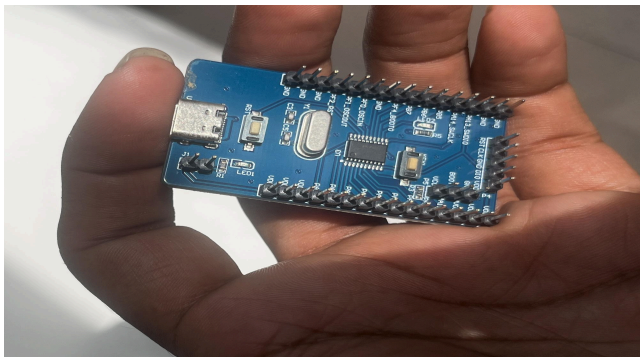
After adding the path open vscode and download necessary extensions and Cortex-debug.



For debugging we need the SWD protocol to connect the target device(PY32F030). We can use SWD on the J-LINK debugger.



PY32F030x8



PY32F003x8

To connect SWD(Serial Wire Debug) j link programmer to PUYA PY32F030(Target device), We need,

1.J-Link Programmer
2.Puya PY32F0 Microcontroller
3.SWD Cable
4.Development Board or Breadboard
5.Connection Wires: Jumper wires.

Steps to connect SWD j link to Puya PY32F0

1.Identify the SWD pins on PY32

SWDIO : Serial wire debug input/output
SWCLK: Serial Wire Clock
GND: Ground
VCC: Power supply (to match the voltage of your microcontroller, usually 3.3V or 5V)
Optionally, nRESET: Reset line (not always required but can be useful)

2. Refer to the Datasheet:

- Look at the Puya PY32 microcontroller datasheet or reference manual to find the exact pins for SWDIO, SWCLK, VCC, and GND.

3. Connect the J-Link to the Microcontroller:

- J-Link Pinout:
  - Pin 1 (VTref): Connect to VCC of the PY32
  - Pin 2 (SWDIO): Connect to SWDIO of the PY32
  - Pin 3 (GND): Connect to GND of the PY32
  - Pin 4 (SWCLK): Connect to SWCLK of the PY32
  - Optional: Connect the RESET pin of J-Link to the RESET pin of the PY32 if needed.

4. Power the Microcontroller:

- Ensure that the Puya PY32 is powered on. The J-Link typically detects the target voltage through the VTref pin.

5. Setup the Software:

- Download and install the SEGGER J-Link software package from the SEGGER website.
- Install the necessary development environment (e.g., SEGGER Embedded Studio, Keil, IAR, or OpenOCD if you are using open-source tools).

6. Configure the Debugger:
   - Open the J-Link Commander (a command-line tool) or the J-Link Configuration tool.
   - Connect to the J-Link device to ensure it's recognized.

7. Programming/Debugging:
   - Use your development environment to start a debug session or to program the microcontroller.
   - Ensure the SWD interface is selected in your toolchain settings.

Example Connection Diagram:
Assuming the standard 10-pin Cortex-M debug connector:

J-LINK	PY32
PIN 1 (VTref)	VCC
PIN 2 (SWDIO)	SWDIO
PIN 3 (GND)	GND
PIN 4 (SWCLK)	SWCLK
PIN 5 (GND)	GND(Optional)
PIN 10 (RESET)	RESET(Optional)

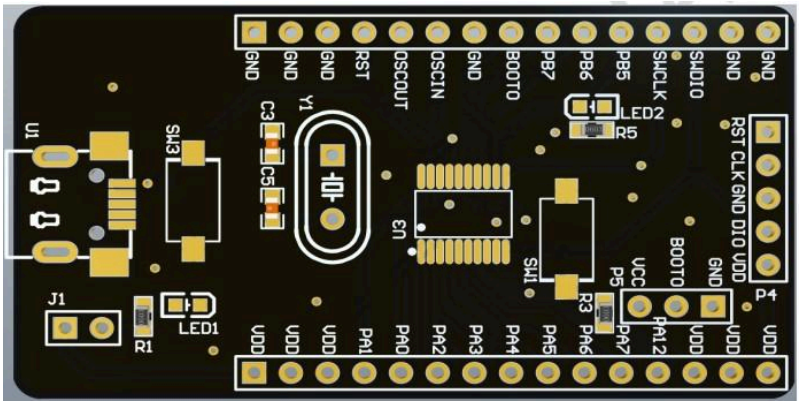
Tips:-
- Check Connections: Double-check all connections for proper placement and contact.
- Voltage Levels: Ensure voltage levels match between the J-Link and the PY32 microcontroller to avoid damage.
- Firmware Updates: Make sure the J-Link firmware is up to date.

Troubleshooting:
- No Connection: If the J-Link cannot detect the target, verify all connections and check the power supply.
- Incorrect Pins: Double-check the datasheet of the PY32 to ensure you're using the correct pins.

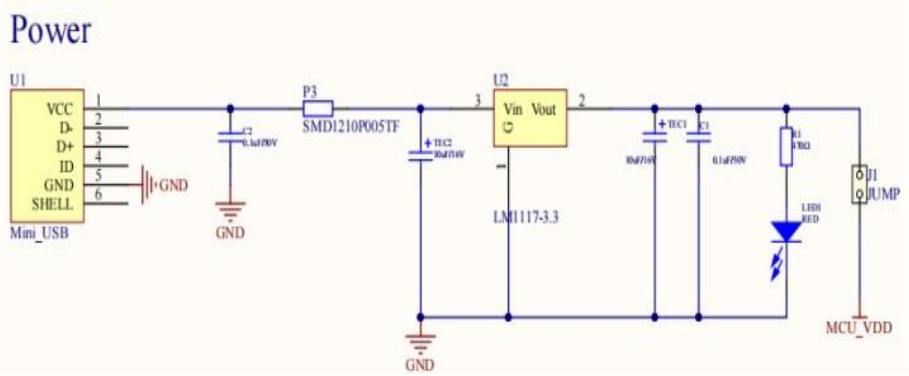
- Software Settings: Ensure the correct settings are selected in your development environment for the target microcontroller and SWD interface.

By following these steps, you should be able to successfully connect your J-Link programmer to the Puya PY32 microcontroller for programming and debugging.

PY32F003

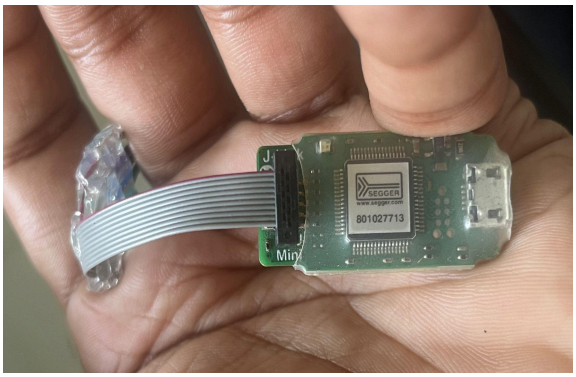


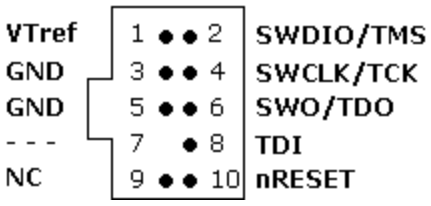
The target voltage can only be achieved only when the jump and J1 are connected shown below.



Power supply

To connect PY32F003 we used J Link EDU Mini (SWD) for debugging ,





PIN CONFIGURATION OF EDU MINI.

J-LINK EDU MINI	PY32F003
PIN 1 (VTref)	VCC
PIN 2 (SWDIO)	SWDIO
PIN 3 (GND)	GND
PIN 4 (SWCLK)	SWCLK
PIN 5 (GND)	GND(Optional)
PIN 10 (RESET)	RESET(Optional)

Download the code file using GIT or direct zip file using the link provided, (File name: PY32-template)

https://github.com/jaydcarlson/py32-template

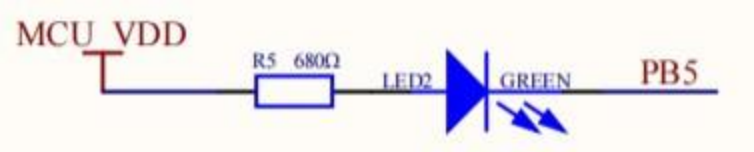
Build the project file using the “make all” command on Powershell or command prompt. Open the project folder on visual studio code and open all the json files. Check whether the json files are correct.

Check whether all the paths to elf files mentioned are correct. Open the main code inside the src file and debug it.

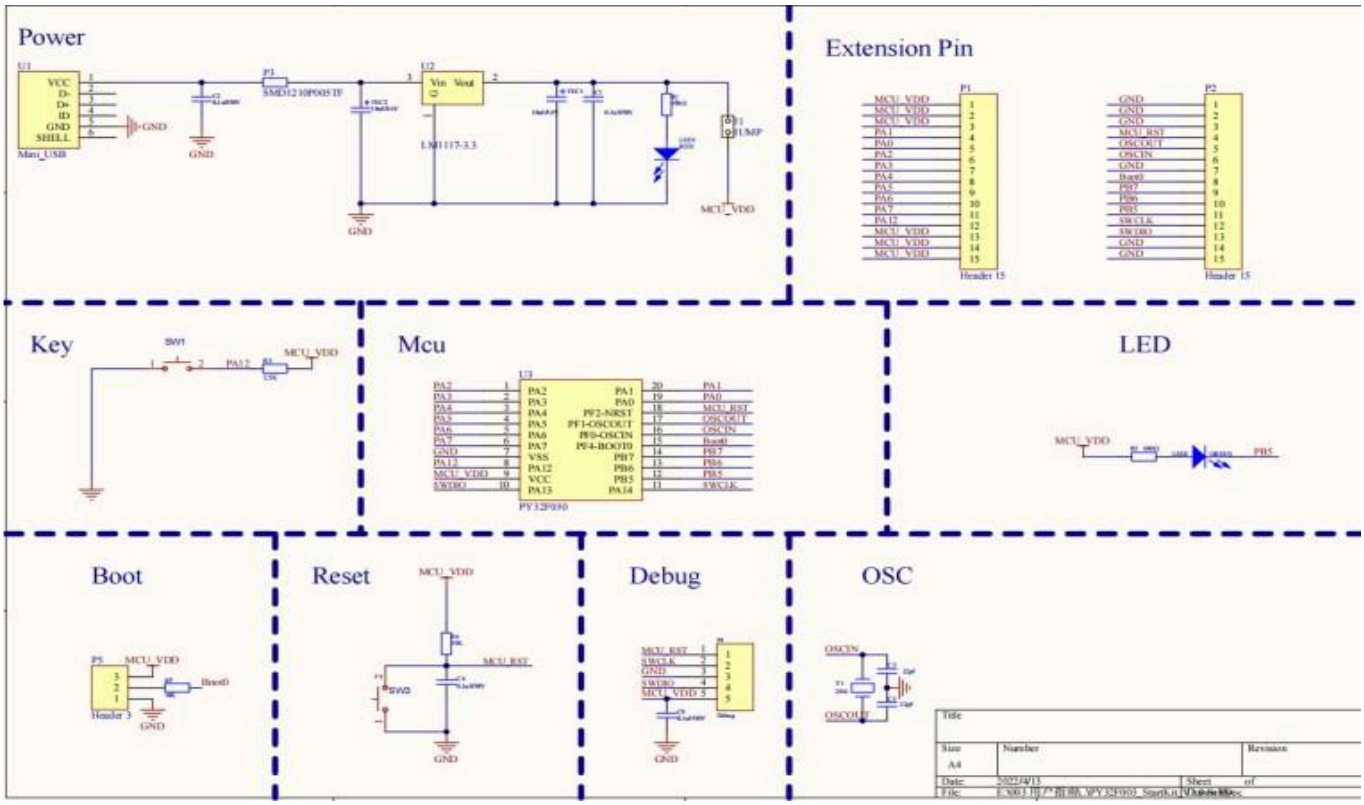
Connect the J Link edu mini with the target microcontroller. Make sure that the connections are perfect and do not forget to connect the J1 and jump in the microcontroller in order to pass the target voltage.

Connect one end of EDU mini to PC using an usb connector and connect the other end of target microcontroller to the PC. Now debug the code. VScode automatically connects to the J-LINK. After successful connection your target device use step over and you can see your led in the target microcontroller starts to blink.

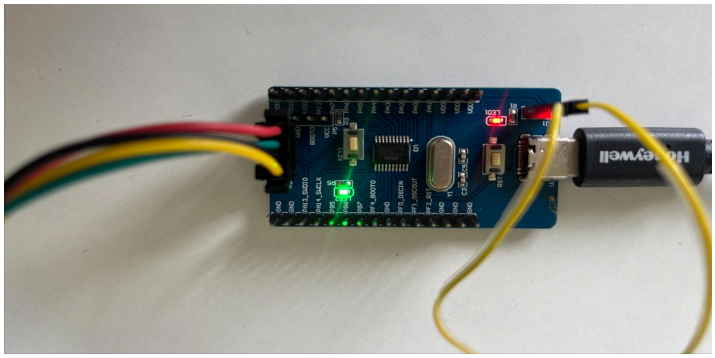
LED functional schematic diagram:



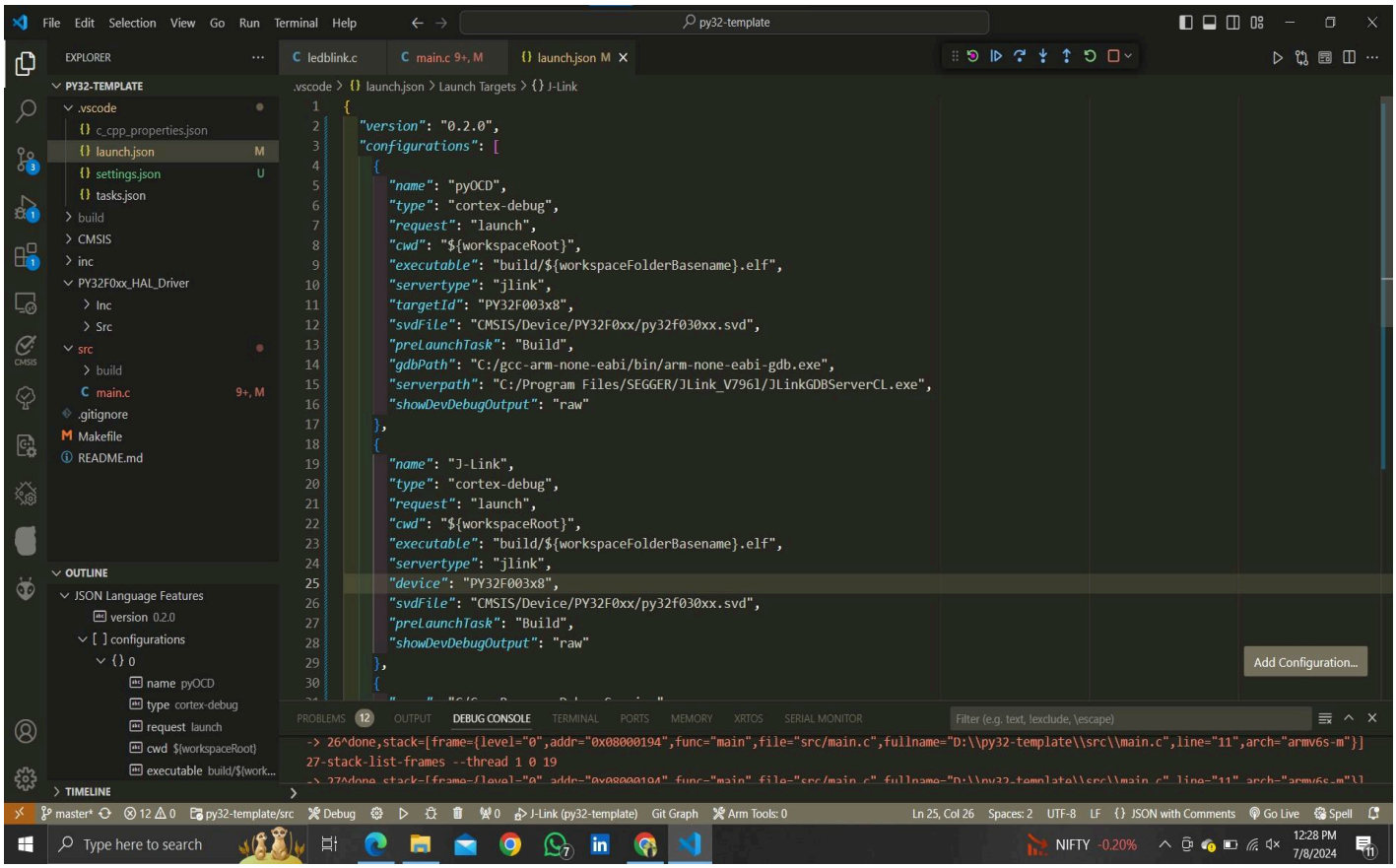
Schematics of PY32F003x8:



PY32F003X8:



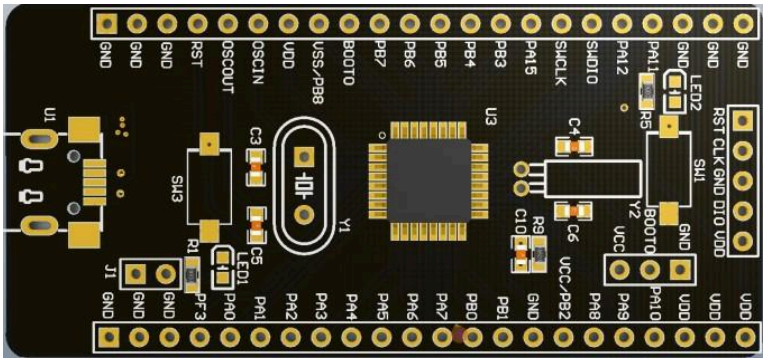
Launch.json file:



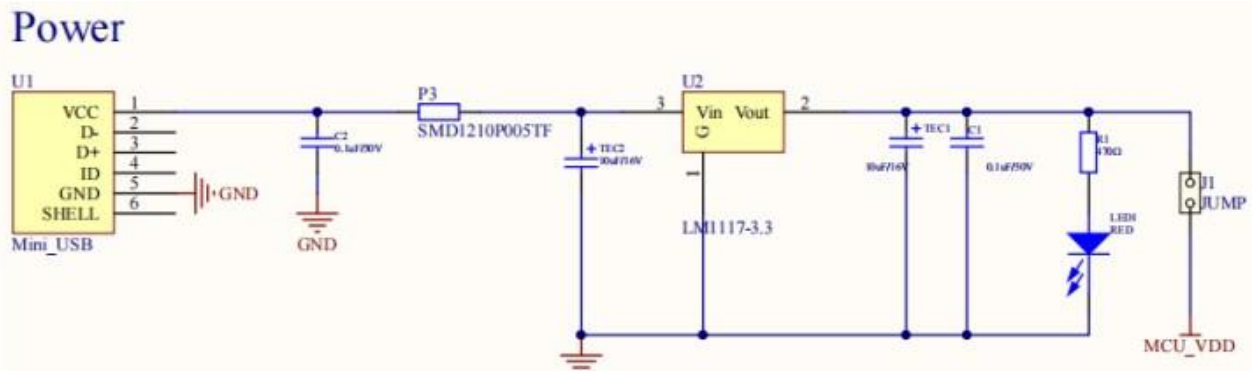
Led code:

```
#include <stdio.h>
#include "py32f0xx.h"

void SysTick_Handler()
{
    HAL_IncTick();
}

int main (void)
{
    HAL_Init();
    SystemCoreClockUpdate();
    __HAL_RCC_GPIOB_CLK_ENABLE();
    HAL_GPIO_Init(GPIOB, &(GPIO_InitTypeDef){.Mode = GPIO_MODE_OUTPUT_OD, .Pin =
GPIO_PIN_5});
    while(1) {
        HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_5);
        HAL_Delay(100);
    }
}
```

PY32F030x8:



The target voltage can only be achieved only when the jump and J1 are connected shown below.



Power supply

Same J - Link edu mini(SWD) is used for debugging,

J-LINK EDU MINI	PY32F030
PIN 1 (VTref)	VCC
PIN 2 (SWDIO)	SWDIO
PIN 3 (GND)	GND
PIN 4 (SWCLK)	SWCLK
PIN 5 (GND)	GND(Optional)
PIN 10 (RESET)	RESET(Optional)

Download the code file using GIT or direct zip file using the link provided, (File name:MCU-Templates)

https://github.com/wagiminator/MCU-Templates.git

Go to MCU-Templates/PY32F0xx/template. Build the project file using the “make all” command on Powershell or command prompt. Open the project folder on visual studio code and open all the json files. Check whether the json files are correct.

Check whether all the paths to elf files mentioned are correct. Open the main code inside the src file and debug it.

For PY32F030 Openocd is used as GNU Debugger so you need to install openocd on arm official page. Go to arm official page and download gcc-arm-none-eabi.

In launch.json select the correct path of Openocd elf file in order to access openocd for debugging

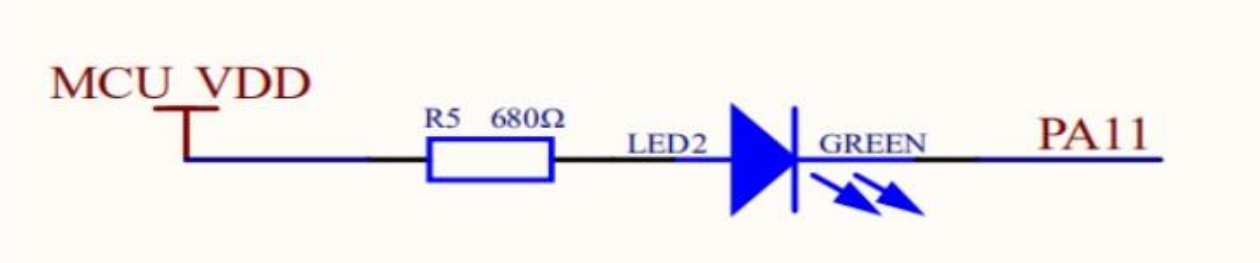
Connect the J Link edu mini with the target microcontroller. Make sure that the connections are perfect and do not forget to connect the J1 and jump in the microcontroller in order to pass the target voltage.

Connect one end of EDU mini to PC using an usb connector and connect the other end of target microcontroller to the PC.

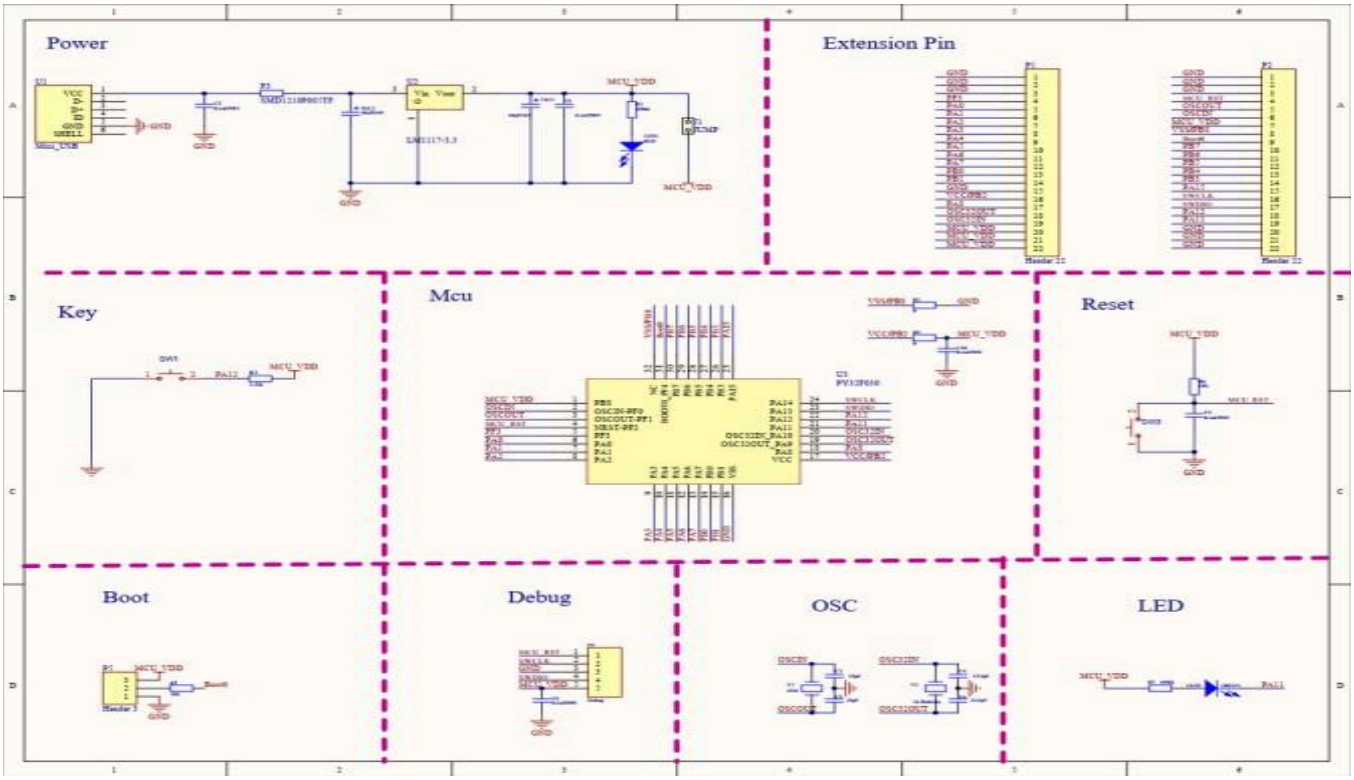
Now debug the code. VScode automatically connects to the J-LINK.

After successful connection your target device use step over and you can see your led in the target microcontroller starts to blink.

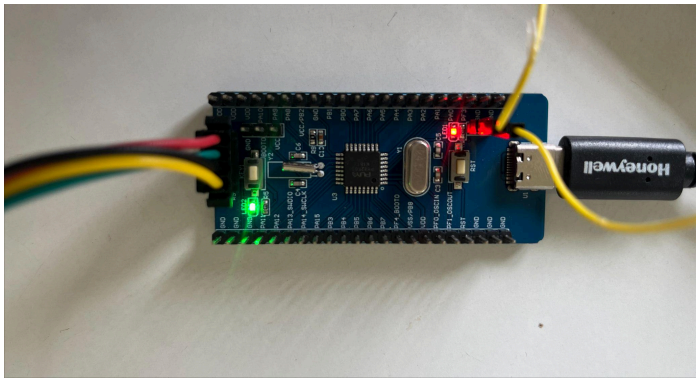
LED functional schematic diagram:



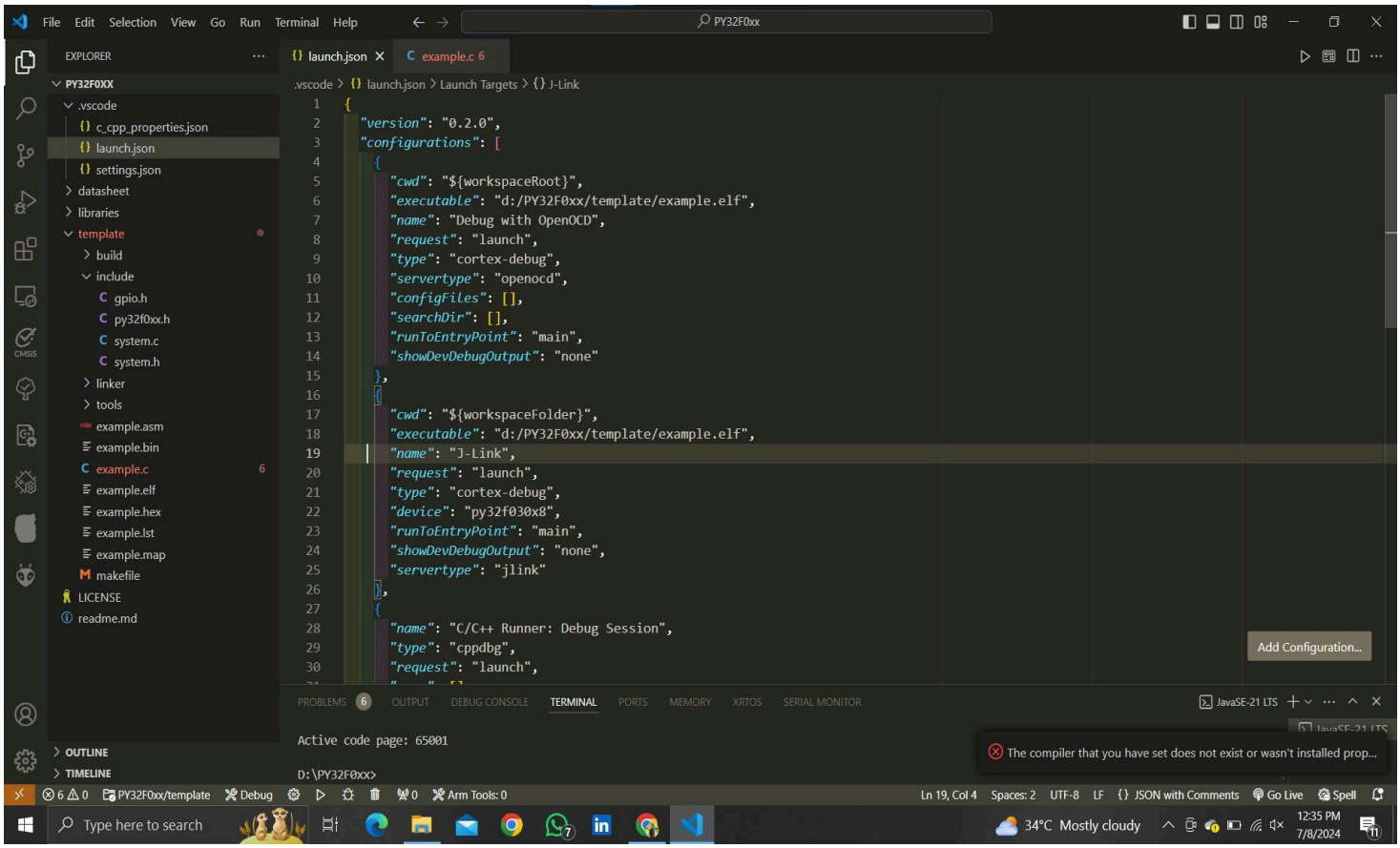
Schematics of PY32F030x8:



PY32F030X8:



Launch.json file:



Led code:

```
#include "system.h"
#include "gpio.h"

#define PIN_LED   PA11

//
================================================================================
// Main Function
//
================================================================================
int main (void) {
  // Setup
  PIN_output(PIN_LED);

  // Loop
  while(1) {
    DLY_ms(500);
    PIN_toggle(PIN_LED);
  }
}
```

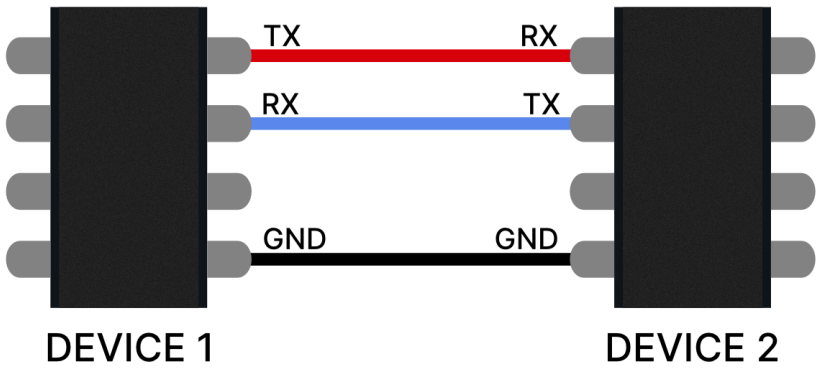
Interfacing UART with PY32F030

To connect a UART (Universal Asynchronous Receiver/Transmitter) interface with the PY32F030 microcontroller, follow these steps

There are some special pins on the PY32F030 microcontroller for UART connectivity.These are TX (transmit) and RX (receive) pins respectively. For information on these pins, Read the datasheet for the microcontroller.

Pin configuration :

TX Pin (Py32F030) to RX Pin (Peripheral Device)
RX Pin (Py32F030) to TX Pin (Peripheral Device)
Ground (GND) to Ground (Peripheral Device)



PIN CONFIGURATION OF PY32F030:

The Py32F030 microcontroller has several UART peripherals, and their pins vary depending on the specific package and variant. Below is a general overview of the UART pins for the Py32F030 series:

UART Peripherals and Pins

UART_ MAP	0	1	2	3	4	5	6	7
TX	PA 2	PA 7	PA 9	PA 14	PA14	PB 6	PF 1	No mapping
RX	PA 3	PA 8	PA 10	PA 13	PA 15	PA 15	PF 0	No mapping

Connect UART Pins:

1. Connect the **TX pin** of the Py32F030 to the **RX pin** of the external device.
2. Connect the **RX pin** of the Py32F030 to the **TX pin** of the external device.
3. Connect the **GND** of the Py32F030 to the **GND** of the external device.

CODE:

```
#include "uart.h"
#include <string.h>  // For strlen
void UART_sendString(const char* str) {
    while (*str) {
        UART_write(*str++);
    }
}
int main(void) {
    // Initialize the UART
    UART_init();
    const char* message = "RADINNOLABS\r\n";
    // Main loop
    while (1) { // Send the string periodically
        UART_sendString(message);
        // Small delay to avoid flooding the terminal
        for (volatile int i = 0; i < 1000000; ++i);
        // Check if data is available to read
        if (UART_available()) {
            // Read a character from UART
            char receivedChar = UART_read();
            // Echo the received character back via UART
            UART_write(receivedChar);
        }
    }
```

```
    return 0;

}
```

Code Explanation:

#include "uart.h"
#include <string.h> // For strlen

`#include "uart.h"`: This includes the header file for the UART (Universal Asynchronous Receiver/Transmitter) functions. This file contains declarations of functions like `UART_init()`, `UART_write()`, `UART_read()`, etc., which are used to initialize and manage UART communication.
`#include <string.h>`: This includes the standard C library for string handling functions, such as `strlen()`. However, in this code, `strlen()` is included but not actually used. You might see its usage in a larger program, but in this case, it doesn't affect the functionality.

void UART_sendString(const char* str) {
  while (*str) {
    UART_write(*str++);
  }
}

`void UART_sendString(const char* str)`: This is a function that sends a string of characters (`str`) via UART. It takes a pointer to a constant string (`const char*`), meaning the string it points to should not be modified.
`while (*str)`: This is the start of a loop that continues as long as `*str` is not the null terminator (`'\0'`). `*str` is the character pointed to by the pointer `str`, and this condition ensures the loop runs through each character in the string until it reaches the end.
`UART_write(*str++)`: This sends the current character (`*str`) over UART using the `UART_write()` function. After sending the character, `str++` increments the pointer to the next character in the string. This ensures that the next character will be processed in the next iteration of the loop.

int main(void) {
  // Initialize the UART
  UART_init();

`int main(void)`: This is the entry point of the program. It's where the execution of the program begins.
`UART_init()`: This function initializes the UART peripheral. It typically sets the baud rate, data format (number of bits, parity, stop bits), and other settings required to communicate over UART. This function should be defined elsewhere in the program, possibly in the `uart.c` file.

```
const char* message = "RADINNOLABS\r\n";
```

`const char* message = "RADINNOLABS\r\n";`: This line defines a pointer to a constant string `message` that contains the text `"RADINNOLABS\r\n"`. The `\r\n` at the end represents a carriage return (`\r`) followed by a newline (`\n`), which is often used in terminal communication to move the cursor to the next line.

```
while (1) {
```

`while (1)`: This starts an infinite loop. The `1` means `true` in C, so this loop will run indefinitely, which is typical for embedded systems that continuously perform tasks.

```
    // Send the string periodically
    UART_sendString(message);
```

`UART_sendString(message);`: This line calls the `UART_sendString()` function to send the defined message `"RADINNOLABS\r\n"` via UART. The message will be sent character by character until the null terminator is encountered.

```
    // Small delay to avoid flooding the terminal
    for (volatile int i = 0; i < 1000000; ++i);
```

`for (volatile int i = 0; i < 1000000; ++i);`: This is a simple delay loop. It iterates 1 million times to introduce a delay between sending messages, preventing the terminal from being flooded with too many characters too quickly. The `volatile` keyword ensures that the compiler does

not optimize the delay loop away. The actual delay depends on the system clock speed and the number of iterations.

```
    // Check if data is available to read
    if (UART_available()) {
```

`if (UART_available())`: This checks whether there is any data available in the UART receive buffer. The `UART_available()` function typically returns a non-zero value if there is data available to be read, and zero otherwise.

```
        // Read a character from UART
        char receivedChar = UART_read();
```

`char receivedChar = UART_read();`: If data is available, this line reads one byte (a character) from the UART receive buffer using the `UART_read()` function. The character is stored in the variable `receivedChar`.

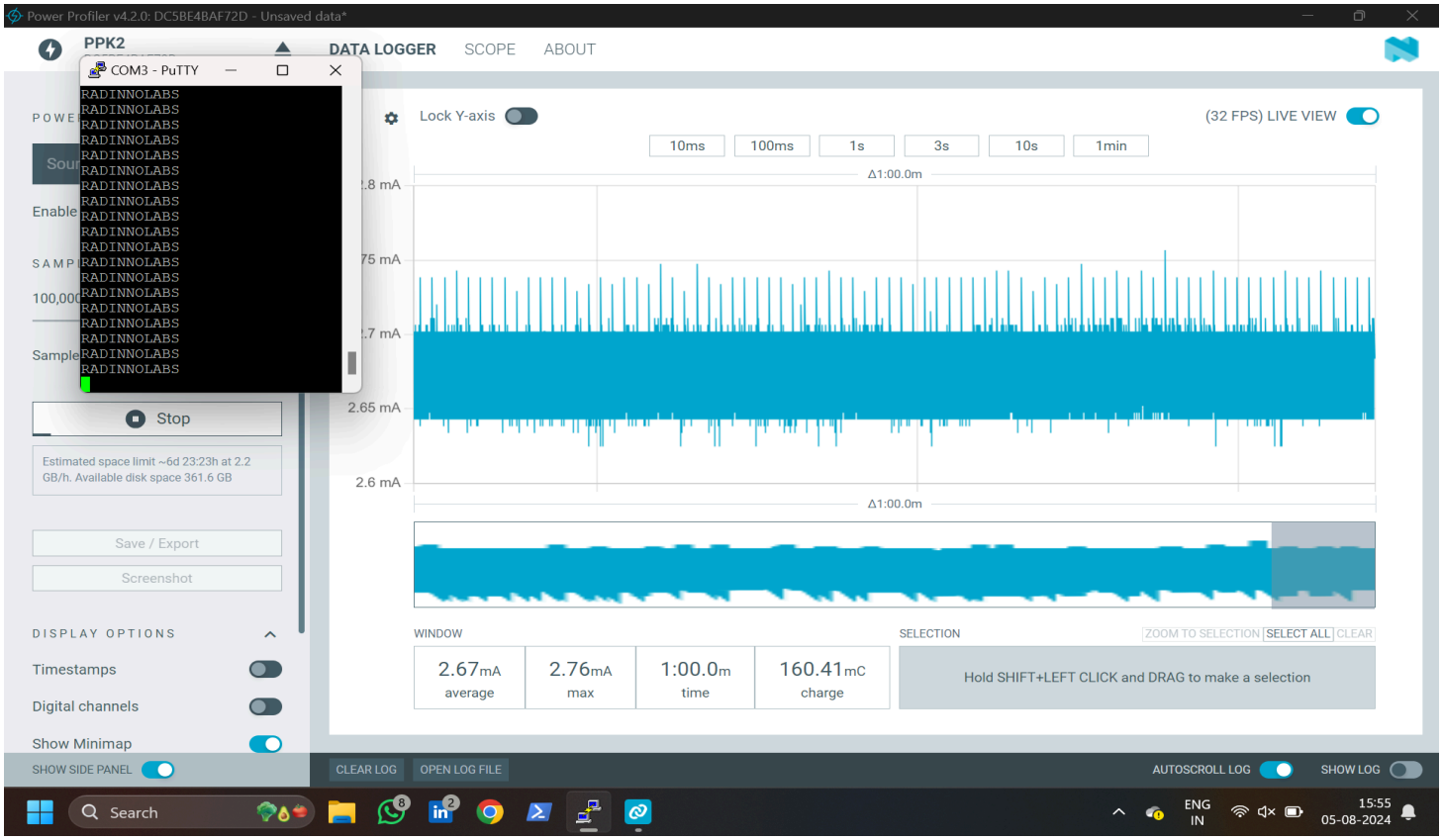
```
        // Echo the received character back via UART
        UART_write(receivedChar);
```

`UART_write(receivedChar);`: This line sends the character back to the terminal via UART. It is an "echo" feature: whatever character is received is immediately sent back to the sender. This is often used for debugging purposes to confirm that the system is receiving and sending data correctly.

```
    }
    return 0;
}
```

`return 0;`: The `main` function returns `0`, which traditionally indicates successful execution in a C program. However, in embedded systems, this is not always meaningful since the program typically runs indefinitely in a `while(1)` loop.
`}`: This closes the `while (1)` loop, and the `main()` function.

UART Image:


Power Profiler v4.2.0: DC5BE4BAF72D - Unsaved data*
PPK2
DATA LOGGER
SCOPE
ABOUT
COM3 - PuTTY
RADINNOLABS
Lock Y-axis
(32 FPS) LIVE VIEW
10ms
100ms
1s
3s
10s
1min
Δ1:00.0m
2.65 mA
2.6 mA
Stop
Estimated space limit ~6d 23:23h at 2.2 GB/h. Available disk space 361.6 GB
Save / Export
Screenshot
DISPLAY OPTIONS
Timestamps
Digital channels
Show Minimap
WINDOW
2.67mA average
2.76mA max
1:00.0m time
160.41mC charge
SELECTION
ZOOM TO SELECTION
SELECT ALL
CLEAR
Hold SHIFT+LEFT CLICK and DRAG to make a selection
SHOW SIDE PANEL
CLEAR LOG
OPEN LOG FILE
AUTOSCROLL LOG
SHOW LOG
Search
ENG
IN
15:55
05-08-2024

PUYA PY32F030 - AHT21B INTERFACE USING I2C.

What is AHT21B and how does it work?

The AHT21B is a high-precision, fully calibrated temperature and humidity sensor. It uses digital outputs and communicates over I²C, making it suitable for a variety of applications that require environmental monitoring. Here's a detailed breakdown of how it works:

1. Basic Operation

The AHT21B sensor integrates a capacitive humidity sensing element, a temperature measuring device, an analog-to-digital converter (ADC), and a calibration unit, all in a small package. It operates in two key phases: initialization and data acquisition.

2. Power-On and Initialization

1.When powered on, the AHT21B sensor requires initialization to prepare it for data measurement. This involves sending specific commands to the sensor via I²C to set it into a working state.

2.The most common initialization command is `0xE1`, which configures the sensor into normal mode. It can also be put into a different mode using other initialization commands, but `0xE1` is often used as a default.

3.The command `0xBE` that you're using with the AHT21B sensor refers to an alternate configuration or mode of operation that you're setting in the sensor. Let's break it down:

1. Command Structure in AHT21B

The AHT21B sensor responds to a set of predefined commands. These commands instruct the sensor to either initialize, enter a certain mode, or trigger a measurement. The most common initialization command is `0xE1`, which puts the sensor into **normal mode**. However, when using `0xBE`, you're likely placing the sensor into a different mode or configuration.

2. Understanding 0xBE

The hexadecimal value `0xBE` is an 8-bit value (in binary: `10111110`), and it's likely a control byte that sets specific operational settings in the AHT21B sensor. Unfortunately, specific details about `0xBE` are not widely documented in public datasheets or manuals for the AHT21B, as manufacturers often reserve some command codes for internal purposes or undocumented features.

However, we can hypothesize a few things based on common practices with sensors like the AHT21B.

3. Possible Function of 0xBE

`0xBE` could trigger one of the following:

a) Alternate Initialization Command

Just like `0xE1` is used to initialize the sensor into normal working mode, `0xBE` might be a variant that initializes the sensor differently. It might configure the sensor into:

- **Low-power mode**: Some sensors offer a low-power mode, where power consumption is reduced at the cost of slower or less frequent measurements.
- **Test or Debug mode**: Sometimes manufacturers include undocumented commands to enable internal test features for debugging or factory calibration.

If `0xBE` puts the sensor into low-power mode or a similar mode, the sensor could still function similarly to the normal mode (`0xE1`), but with altered performance characteristics, such as slower response time or reduced precision.

b) Calibration Mode

Since many humidity and temperature sensors have built-in calibration routines, `0xBE` could be related to starting or adjusting internal calibration. In this case, the sensor might adjust its calibration coefficients based on environmental conditions or reset to factory-calibrated values.

c) Status Check or Special Configuration

In some cases, control commands like `0xBE` might query or configure sensor-specific settings:

- **Status Query**: This could mean the command is used to read internal status registers (e.g., error status, mode flags, etc.).
- **Special Configuration**: The command might adjust non-standard settings such as operational ranges, resolution, or specific internal parameters that control how the sensor performs measurements.

4. Comparison with 0xE1

While `0xE1` is well-documented as the typical initialization command that puts the AHT21B sensor into **normal mode**, where it can operate as intended (performing temperature and humidity measurements), `0xBE` is less commonly used and might modify the sensor's behavior in the following ways:

- **Reduced Measurement Frequency**: If `0xBE` puts the sensor in a low-power mode, it might take longer for the sensor to perform each measurement.

- **Different Accuracy Settings**: The accuracy of the sensor's readings might change, with reduced power usage or faster response times.
- **Reserved or Custom Use**: Some commands like `0xBE` could be reserved for specific use cases or custom applications. The manufacturer might use this for factory-level testing or for operations that aren't relevant to typical end-users.

5. How It Might Work in Practice

If you're using `0xBE`, you might observe some different behaviors from the sensor compared to when using `0xE1`. Here's what could happen:

- The sensor might still perform humidity and temperature measurements, but the results could be less frequent or less accurate, depending on the mode it is in.
- The sensor might require a longer delay before it responds after the measurement command (`0xAC`).
- The sensor could enter a state that uses less power, which would be ideal for battery-operated systems but could result in slower operation.

6. Testing 0xBE

If you're using `0xBE` in your code, you should:

1. **Monitor the Power Consumption**: If `0xBE` activates a low-power mode, you might observe reduced power usage during operation.
2. **Measure the Sensor's Response Time**: Check whether it takes longer to get valid data from the sensor after triggering measurements. The delay might indicate that the sensor is operating in a slower mode.
3. **Compare Accuracy**: Compare the sensor's data output between the `0xBE` and `0xE1` commands. If `0xBE` puts the sensor into a different operating mode, there could be a noticeable difference in precision or stability of the readings.

7. Practical Example

Let's assume the AHT21B is in a low-power mode when initialized with `0xBE`:

- After sending the initialization command `0xBE`, the sensor might take longer to respond to measurement requests because it is conserving energy.
- When you send the measurement command `0xAC`, the sensor processes the request, but it might operate with lower precision or slower update rates.
- The data format (humidity and temperature readings) would still follow the same structure (6 bytes: 3 for humidity, 3 for temperature), but the values could show slight deviations due to the low-power mode affecting performance.

8. Manufacturer Intent

It's important to note that not all commands are meant for public use. Manufacturers often design sensors with undocumented commands like `0xBE` for internal purposes, and they may not be recommended for typical users. If you're not seeing any major difference in operation with `0xBE` versus `0xE1`, it could simply be an alternate initialization mode that isn't designed for normal use but doesn’t harm performance.

3. I²C Communication

- **I²C Address**: The AHT21B has a fixed 7-bit address `0x38`. Communication starts by addressing this sensor with the I²C address.
- The master device (in your case, the microcontroller) communicates with the sensor using two lines:

SCL (Serial Clock Line): For clocking the data transfer.

SDA (Serial Data Line): For sending and receiving data.

Steps in I²C Communication:

- **Start Condition**: The microcontroller sends a START condition followed by the AHT21B's address.
- **Command Phase**: The microcontroller sends commands to the AHT21B to either initialize it or to trigger a measurement.
- **Data Phase**: After a measurement is triggered, the sensor performs the necessary internal processing and then returns data back to the microcontroller.

4. Measurement Modes

Once initialized, the AHT21B sensor can measure both temperature and humidity.

- **Triggering a Measurement**: A common command to trigger a measurement is `0xAC`. This command prompts the sensor to take both temperature and humidity readings.
- **Measurement Time**: It takes about 80 milliseconds (ms) for the sensor to complete its measurement. During this time, it processes the data internally and prepares it for transmission.

5. Data Retrieval

After the measurement is completed, the sensor returns 6 bytes of data:

- **1st and 2nd bytes**: Humidity data (20-bit value).
- **3rd byte**: A combination of the least significant humidity bits and most significant temperature bits.
- **4th and 5th bytes**: Temperature data (20-bit value).
- **6th byte**: A cyclic redundancy check (CRC) byte for error checking.

Each of these values needs to be extracted and converted into meaningful temperature and humidity readings.

Data Format:

- **Humidity Data**: 20-bit value stored in the first three bytes.
- **Temperature Data**: 20-bit value stored in the last three bytes.

6. Conversion to Physical Units

Once the raw data is received, the sensor's output must be converted to standard units (percentage for humidity, degrees Celsius for temperature).

- **Humidity Conversion**:
  - The 20-bit humidity value (RH_RAW) is first extracted from the raw data bytes.
  - This raw value is then converted to relative humidity using the formula:
  - RH=RH_RAW220×100RH = \frac{{RH_RAW}}{{2^{20}}} \times 100RH=220RH_RAW×100
  - This gives the relative humidity in percentage.
- **Temperature Conversion**:
  - Similarly, the 20-bit temperature value (T_RAW) is extracted.
  - The temperature in degrees Celsius is calculated using the formula:
  - T=T_RAW220×200−50T = \frac{{T_RAW}}{{2^{20}}} \times 200 - 50T=220T_RAW×200−50
  - This formula converts the raw temperature value into a meaningful temperature in degrees Celsius.

7. Cyclic Redundancy Check (CRC)

The sensor appends a CRC byte to ensure that the data received by the microcontroller is error-free. The microcontroller can optionally verify this CRC to detect transmission errors.

8. Power Management

The AHT21B sensor has low power consumption and operates between 2.0V and 5.5V. It can be used in battery-powered applications, and it also supports power-down modes for energy saving.

9. Internal Calibration

The sensor is factory-calibrated for both humidity and temperature. This means that the output data is already corrected, and no further calibration is necessary by the user.

10. Accuracy and Precision

- **Temperature Accuracy**: ±0.3°C
- **Humidity Accuracy**: ±2% RH (relative humidity) These accuracy levels make the sensor suitable for applications where precise environmental monitoring is critical.

11. Periodic Measurements

The sensor can be polled periodically (every few seconds, as per your preference). After each measurement, the microcontroller can retrieve the temperature and humidity values and print them over UART, or store them for further processing.

12. Command Summary

- **0xE1**: Initialization command (Normal mode).
- **0xAC**: Trigger measurement (read both humidity and temperature).
- **Data bytes**: 6 bytes received for temperature, humidity, and CRC check.

Key Points:

1. **High precision**: The AHT21B is highly accurate for both humidity and temperature.
2. **Digital communication**: Uses the I²C protocol, which simplifies wiring and communication with microcontrollers
3. **Factory calibration**: No need for user calibration.
4. **Low power**: Suitable for battery-operated devices.
5. **Small package**: Ideal for space-constrained designs.

Hardware steps:

To interface the AHT21B sensor with your PY32F030x8 microcontroller, follow these hardware steps:

1. Connect I2C Pins:

- SCL (Clock Line) to PF0 (I2C1_SCL)**: Connect the I2C clock (SCL) pin of the AHT21B to PF0 on your PY32F030x8.

- SDA (Data Line) to PF1 (I2C1_SDA)**: Connect the I2C data (SDA) pin of the AHT21B to PF1 on your PY32F030x8.

Pull-up Resistors:

- Connect 4.7kΩ pull-up resistors between SCL (PF0) and 3.3V, and between SDA (PF1) and 3.3V. These resistors are required to ensure proper signal levels for I2C communication.

2. Connect UART Pins:

- TX (Transmit) to PA2 (USART2_TX):

Connect the TX pin of the AHT21B to PA2 on the PY32F030x8 for UART transmission.

- RX (Receive) to PA3 (USART2_RX):

Connect the RX pin of the AHT21B to PA3 on the PY32F030x8 for receiving data.

3. Power the AHT21B Sensor:

- Connect the VCC pin of the AHT21B to the 3.3V supply (ensure it is within the operating voltage range of the sensor).

- Connect the GND pin of the AHT21B to the ground (GND) of the PY32F030x8.

4. Power Supply for the Microcontroller:

- Make sure the PY32F030x8 microcontroller is powered with an appropriate voltage supply (e.g., 3.3V or 5V based on your configuration).

5. Optional: Connect Debugging Tools:

- You may want to use a debugger (e.g., ST-Link or J-Link) for debugging and programming the microcontroller.

This completes the hardware setup to interface the AHT21B sensor with your PY32F030x8 microcontroller.

[I used J-Link edu mini debugger for this project]

CODE:

```
#include "i2c.h"       // Include your I2C driver

#include "uart.h"      // Include your UART driver

#include <stdint.h>

#include <stdio.h>     // Include stdio for snprintf

#include "gpio.h"

#include "py32f0xx.h"



// AHT21B sensor I2C address

#define AHT21B_ADDRESS  0x38



// AHT21B commands

#define AHT21B_CMD_INIT  0xBE

#define AHT21B_CMD_MEAS  0xAC

#define AHT21B_CMD_RESET 0xBA
```

```
void delay_ms(uint32_t ms);
void AHT21B_Init(void);
void AHT21B_TriggerMeasurement(void);
void AHT21B_ReadData(float* temperature, float* humidity);
void AHT21B_Reset(void);


// UART_SendString function declaration and definition
void UART_SendString(const char *str) {
    while (*str) {
        UART_write(*str++);  // Replace this with your actual UART sending function
    }
}


int main(void) {
    // Initialize system and peripherals
    UART_init();  // Initialize UART for communication
    I2C_init();   // Initialize I2C
    UART_SendString("Initializing AHT21B...\r\n");
    delay_ms(500);
    // Initialize AHT21B sensor
    AHT21B_Init();
    while (1) {
        float temperature = 0;
        float humidity = 0;
        delay_ms(500);
        // Trigger a new measurement
```

```
        AHT21B_TriggerMeasurement();
        // Read temperature and humidity data
        AHT21B_ReadData(&temperature, &humidity);
        // Variables for integer and decimal parts
        int tempInt = (int)(temperature);
        int tempDec = (int)((temperature - tempInt) * 100);  // Get 2 decimal places
        int humInt = (int)(humidity);
        int humDec = (int)((humidity - humInt) * 100);  // Get 2 decimal places
        // Display the temperature and humidity
        char buffer[64];
        snprintf(buffer, sizeof(buffer), "Temperature: %d.%02d C\r\nHumidity: %d.%02d %%\r\n",
                 tempInt, tempDec, humInt, humDec);
        UART_SendString(buffer);  // Send the updated values via UART
        delay_ms(10000);  // 10 seconds delay before the next measurement
    }
}


void AHT21B_Init(void) {
    uint8_t initCmd[] = {AHT21B_CMD_INIT, 0x08, 0x00};
    // Send initialization command
    I2C_start(AHT21B_ADDRESS << 1);  // Send the address with the write bit
    for (int i = 0; i < sizeof(initCmd); i++) {
        I2C_write(initCmd[i]);
    }
    I2C_stop();
    delay_ms(500);  // Wait for sensor stabilization
    // Verify initialization
```

```
    uint8_t status = 0xFF;
    I2C_start((AHT21B_ADDRESS << 1) | 1);  // Read status
    status = I2C_read(0);
    I2C_stop();
    if (!(status & 0x08)) {
        UART_SendString("AHT21B initialization failed!\r\n");
    } else {
        UART_SendString("AHT21B initialized successfully.\r\n");
    }
}

void AHT21B_TriggerMeasurement(void) {
    uint8_t measureCmd[] = {AHT21B_CMD_MEAS, 0x33, 0x00};
    UART_SendString("Triggering measurement...\r\n");
    // Send measurement command to AHT21B
    I2C_start(AHT21B_ADDRESS << 1);  // Send the address with the write bit
    for (int i = 0; i < sizeof(measureCmd); i++) {
        I2C_write(measureCmd[i]);  // Send the measurement command
    }
    I2C_stop();

    UART_SendString("Measurement command sent. Waiting for sensor readiness...\r\n");
    // Wait for readiness (max 1 second)
    uint8_t status = 0xFF;
    for (int i = 0; i < 20; i++) {  // Check status 20 times (20 * 50ms = 1s)
        I2C_start((AHT21B_ADDRESS << 1) | 1);
        status = I2C_read(0);  // Read status byte
```

```
        I2C_stop();
        if (!(status & 0x80)) {
            UART_SendString("Sensor ready.\r\n");
            return;
        }
        delay_ms(50);
    }
    UART_SendString("Sensor not ready after 1 second.\r\n");
}

void AHT21B_ReadData(float* temperature, float* humidity) {
    uint8_t data[6];
    int attempts = 5;  // Retry up to 5 times

    while (attempts--) {
        // Request data from AHT21B
        I2C_start((AHT21B_ADDRESS << 1) | 1);  // Send the address with the read bit
        for (int i = 0; i < 6; i++) {
            data[i] = I2C_read(i < 5);  // Acknowledge all but the last byte
        }
        I2C_stop();


        // Check if the status byte indicates valid data
        if (!(data[0] & 0x80)) {
            break;  // Data is valid
        }
```

```
        delay_ms(50);  // Wait and retry
    }

    if (data[0] & 0x80) {
        UART_SendString("Invalid data after retries, skipping...\r\n");
        *temperature = 0;
        *humidity = 0;
        return;
    }
    // Extract and validate data
    uint32_t rawHumidity = ((data[1] << 12) | (data[2] << 4) | (data[3] >> 4)) & 0xFFFFF;
    uint32_t rawTemperature = ((data[3] & 0x0F) << 16) | (data[4] << 8) | data[5];

    *humidity = ((float)rawHumidity / 1048576.0) * 100.0;
    *temperature = ((float)rawTemperature / 1048576.0) * 200.0 - 50.0;
    // Clamp and validate
    if (*temperature < -40.0 || *temperature > 85.0) {
        UART_SendString("Temperature out of range, retrying...\r\n");
        *temperature = 0;
    }
    if (*humidity < 0.0 || *humidity > 100.0) {
        UART_SendString("Humidity out of range, retrying...\r\n");
        *humidity = 0;
    }
}
```

```
void AHT21B_Reset(void) {

    uint8_t resetCmd = AHT21B_CMD_RESET;



    UART_SendString("Sending reset command...\r\n");



    // Send reset command

    I2C_start(AHT21B_ADDRESS << 1);  // Send the address with the write bit

    I2C_write(resetCmd);             // Send the reset command

    I2C_stop();



    delay_ms(1000);  // Wait for the reset to complete

}



// Delay function (simple blocking)

void delay_ms(uint32_t ms) {

    for (uint32_t i = 0; i < ms; i++) {

        for (uint32_t j = 0; j < 1000; j++) {

            __asm__("nop");  // No-operation for precise timing

        }

    }

}
```

Code Explanation:

```
#include "i2c.h"       // Include your I2C driver
#include "uart.h"      // Include your UART driver
#include <stdint.h>
#include <stdio.h>     // Include stdio for snprintf
#include "gpio.h"
#include "py32f0xx.h"
```

These headers are included to enable the use of various peripherals and functions. `i2c.h`, `uart.h`, and `gpio.h` are custom drivers for I2C, UART, and GPIO operations, respectively.

`stdint.h` provides definitions for fixed-width integers.

`stdio.h` is included to use `snprintf()` for formatting strings.

`py32f0xx.h` is likely a microcontroller-specific header for the **PY32F030x8** (based on the context) which includes register definitions for the hardware.

```
#define AHT21B_ADDRESS  0x38
```

Defines the I2C address of the AHT21B sensor. The address is `0x38` for the AHT21B in its default state.

```
#define AHT21B_CMD_INIT  0xBE
#define AHT21B_CMD_MEAS  0xAC
#define AHT21B_CMD_RESET 0xBA
```

These define the commands for the AHT21B sensor:

- `AHT21B_CMD_INIT`: Command to initialize the sensor.
- `AHT21B_CMD_MEAS`: Command to trigger a measurement.
- `AHT21B_CMD_RESET`: Command to reset the sensor.

```
void delay_ms(uint32_t ms);

void AHT21B_Init(void);

void AHT21B_TriggerMeasurement(void);

void AHT21B_ReadData(float* temperature, float* humidity);

void AHT21B_Reset(void);
```

Declares functions to initialize the sensor, trigger measurements, read data, reset the sensor, and perform delays in milliseconds.

```
void UART_SendString(const char *str) {
    while (*str) {
        UART_write(*str++);  // Replace this with your actual UART sending function
    }
}
```

`UART_SendString` sends a string over UART (a serial communication interface).

- It loops through each character of the string `str` and sends it one by one using `UART_write()`.
- `UART_write()` is assumed to be a low-level function that sends one character through UART.

```
int main(void) {
    // Initialize system and peripherals
```

```
    UART_init();  // Initialize UART for communication
    I2C_init();   // Initialize I2C
```

The `main()` function starts by initializing the UART and I2C peripherals.

`UART_init()` and `I2C_init()` are assumed to be custom functions that set up the respective communication protocols.

```
    UART_SendString("Initializing AHT21B...\r\n");
    delay_ms(500);

    // Initialize AHT21B sensor
    AHT21B_Init();
```

A message is sent via UART to indicate that the sensor initialization is starting, followed by a 500ms delay.

`AHT21B_Init()` is called to initialize the sensor.

```
    while (1) {
        float temperature = 0;
        float humidity = 0;

        delay_ms(500);

        // Trigger a new measurement
        AHT21B_TriggerMeasurement();
```

```
    // Read temperature and humidity data
    AHT21B_ReadData(&temperature, &humidity);
```

An infinite loop starts, where every 500ms, it triggers a new measurement and then reads the temperature and humidity values into the variables `temperature` and `humidity`.

```
    // Variables for integer and decimal parts
    int tempInt = (int)(temperature);
    int tempDec = (int)((temperature - tempInt) * 100);  // Get 2 decimal places
    int humInt = (int)(humidity);
    int humDec = (int)((humidity - humInt) * 100);  // Get 2 decimal places
```

The temperature and humidity values are split into integer and decimal parts (2 decimal places) for easier display.

```
    // Display the temperature and humidity
    char buffer[64];
    snprintf(buffer, sizeof(buffer), "Temperature: %d.%02d C\r\nHumidity: %d.%02d %%\r\n",
             tempInt, tempDec, humInt, humDec);

    UART_SendString(buffer);  // Send the updated values via UART
```

A formatted string containing the temperature and humidity is created using `snprintf` and then sent via UART using `UART_SenStrding()`.

```
        delay_ms(10000);  // 10 seconds delay before the next measurement
    }
}
```

The program waits for 10 seconds before starting the next measurement.

```
void AHT21B_Init(void) {
    uint8_t initCmd[] = {AHT21B_CMD_INIT, 0x08, 0x00};

    // Send initialization command
    I2C_start(AHT21B_ADDRESS << 1);  // Send the address with the write bit
    for (int i = 0; i < sizeof(initCmd); i++) {
        I2C_write(initCmd[i]);
    }
    I2C_stop();
```

The `AHT21B_Init()` function sends the initialization command to the sensor.

- `AHT21B_CMD_INIT` followed by `0x08` and `0x00` is the initialization command sequence.
- `I2C_start` sends a start condition for I2C communication, `I2C_write` sends the data, and `I2C_stop` ends the I2C transaction.

```
    delay_ms(500);  // Wait for sensor stabilization

    // Verify initialization
    uint8_t status = 0xFF;
```

```
    I2C_start((AHT21B_ADDRESS << 1) | 1);  // Read status
    status = I2C_read(0);
    I2C_stop();
```

After sending the initialization command, a delay is introduced to allow the sensor to stabilize.

The code then reads the sensor status to verify initialization. If the sensor is ready, it should respond with a specific value.

```
    if (!(status & 0x08)) {
        UART_SendString("AHT21B initialization failed!\r\n");
    } else {
        UART_SendString("AHT21B initialized successfully.\r\n");
    }
}
```

If the sensor initialization fails (status check fails), an error message is displayed. Otherwise, a success message is sent via UART.

```
void AHT21B_TriggerMeasurement(void) {
    uint8_t measureCmd[] = {AHT21B_CMD_MEAS, 0x33, 0x00};

    UART_SendString("Triggering measurement...\r\n");

    // Send measurement command to AHT21B
    I2C_start(AHT21B_ADDRESS << 1);  // Send the address with the write bit
    for (int i = 0; i < sizeof(measureCmd); i++) {
```

```
        I2C_write(measureCmd[i]);  // Send the measurement command
    }
    I2C_stop();
```

This function sends the command to trigger a new measurement.

The command sequence is `AHT21B_CMD_MEAS`, followed by `0x33` and `0x00`, which tells the sensor to begin measurement.

```
    UART_SendString("Measurement command sent. Waiting for sensor readiness...\r\n");

    // Wait for readiness (max 1 second)
    uint8_t status = 0xFF;
    for (int i = 0; i < 20; i++) {
  // Check status 20 times (20 * 50ms = 1s)
        I2C_start((AHT21B_ADDRESS << 1) | 1);
        status = I2C_read(0);  // Read status byte
        I2C_stop();

        if (!(status & 0x80)) {
            UART_SendString("Sensor ready.\r\n");
            return;
        }
        delay_ms(50);
    }
```

```
    UART_SendString("Sensor not ready after 1 second.\r\n");
}
```

After sending the measurement command, it waits for the sensor to be ready.

It checks the sensor's status byte (20 times, every 50ms). If the sensor is ready, it proceeds; otherwise, it reports a timeout.

```
void AHT21B_ReadData(float* temperature, float* humidity) {
    uint8_t data[6];
    int attempts = 5;  // Retry up to 5 times
```

This function reads the temperature and humidity data from the sensor. It retries up to 5 times if the data is invalid.

```
    while (attempts--) {
        // Request data from AHT21B
        I2C_start((AHT21B_ADDRESS << 1) | 1);  // Send the address with the read bit
        for (int i = 0; i < 6; i++) {
            data[i] = I2C_read(i < 5);  // Acknowledge all but the last byte
        }
        I2C_stop();
```

It reads 6 bytes of data from the sensor. The data includes raw temperature and humidity values.

```
        // Check if the status byte indicates valid data
        if (!(data[0] & 0x80)) {
            break;  // Data is valid
        }
```

```
        delay_ms(50);  // Wait and retry
    }
```

The first byte of the response contains a status flag. If the status byte indicates invalid data (bit 7 is set), it retries up to 5 times.

```
    if (data[0] & 0x80) {
        UART_SendString("Invalid data after retries, skipping...\r\n");
        *temperature = 0;
        *humidity = 0;
        return;
    }
```

If valid data is not received after retries, it sets the temperature and humidity to `0` and returns.

```
    // Extract and validate data
    uint32_t rawHumidity = ((data[1] << 12) | (data[2] << 4) | (data[3] >> 4)) & 0xFFFFF;
    uint32_t rawTemperature = ((data[3] & 0x0F) << 16) | (data[4] << 8) | data[5];
```

The raw humidity and temperature values are extracted from the sensor’s data buffer and packed into 20-bit values.

```
    *humidity = ((float)rawHumidity / 1048576.0) * 100.0;
    *temperature = ((float)rawTemperature / 1048576.0) * 200.0 - 50.0;
```

The raw values are converted to real temperature and humidity values using the sensor's scaling factors.

```
    // Clamp and validate
    if (*temperature < -40.0 || *temperature > 85.0) {
        UART_SendString("Temperature out of range, retrying...\r\n");
        *temperature = 0;
    }
    if (*humidity < 0.0 || *humidity > 100.0) {
        UART_SendString("Humidity out of range, retrying...\r\n");
        *humidity = 0;
    }
}
```

Temperature and humidity values are clamped within valid ranges (-40°C to 85°C for temperature, 0% to 100% for humidity). If any values fall outside these ranges, the sensor is retried.

```
void AHT21B_Reset(void) {
    uint8_t resetCmd = AHT21B_CMD_RESET;

    UART_SendString("Sending reset command...\r\n");

    // Send reset command
    I2C_start(AHT21B_ADDRESS << 1);  // Send the address with the write bit
    I2C_write(resetCmd);             // Send the reset command
    I2C_stop();
```

```
    delay_ms(1000);  // Wait for the reset to complete
}
```

This function sends a reset command to the AHT21B to reset it, and then waits for 1 second for the sensor to restart.

```
void delay_ms(uint32_t ms) {
    for (uint32_t i = 0; i < ms; i++) {
        for (uint32_t j = 0; j < 1000; j++) {
            __asm__("nop");  // No-operation for precise timing
        }
    }
}
```

This is a simple delay function that creates a delay of `ms` milliseconds by executing "no operation" assembly instructions in a loop.

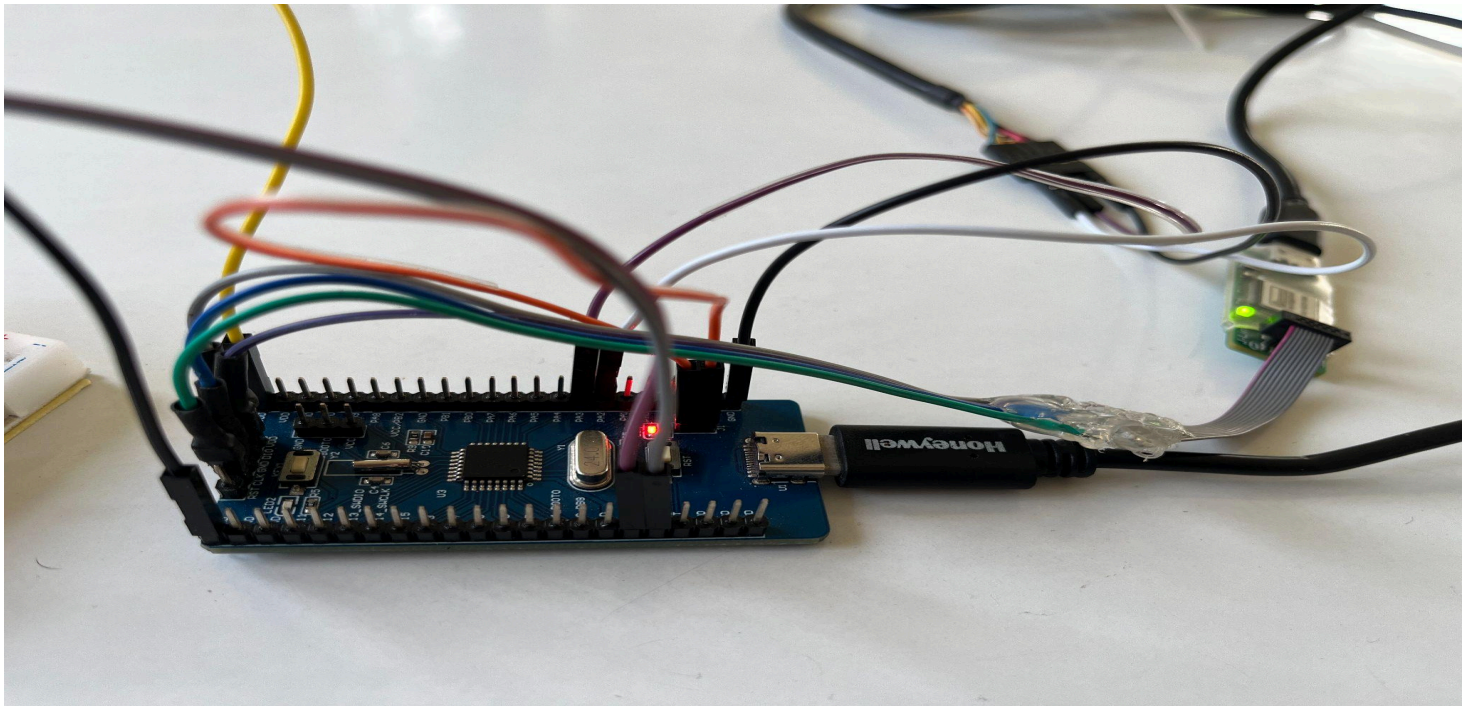
The code is designed to interface with the **AHT21B** sensor to measure temperature and humidity via I2C communication.

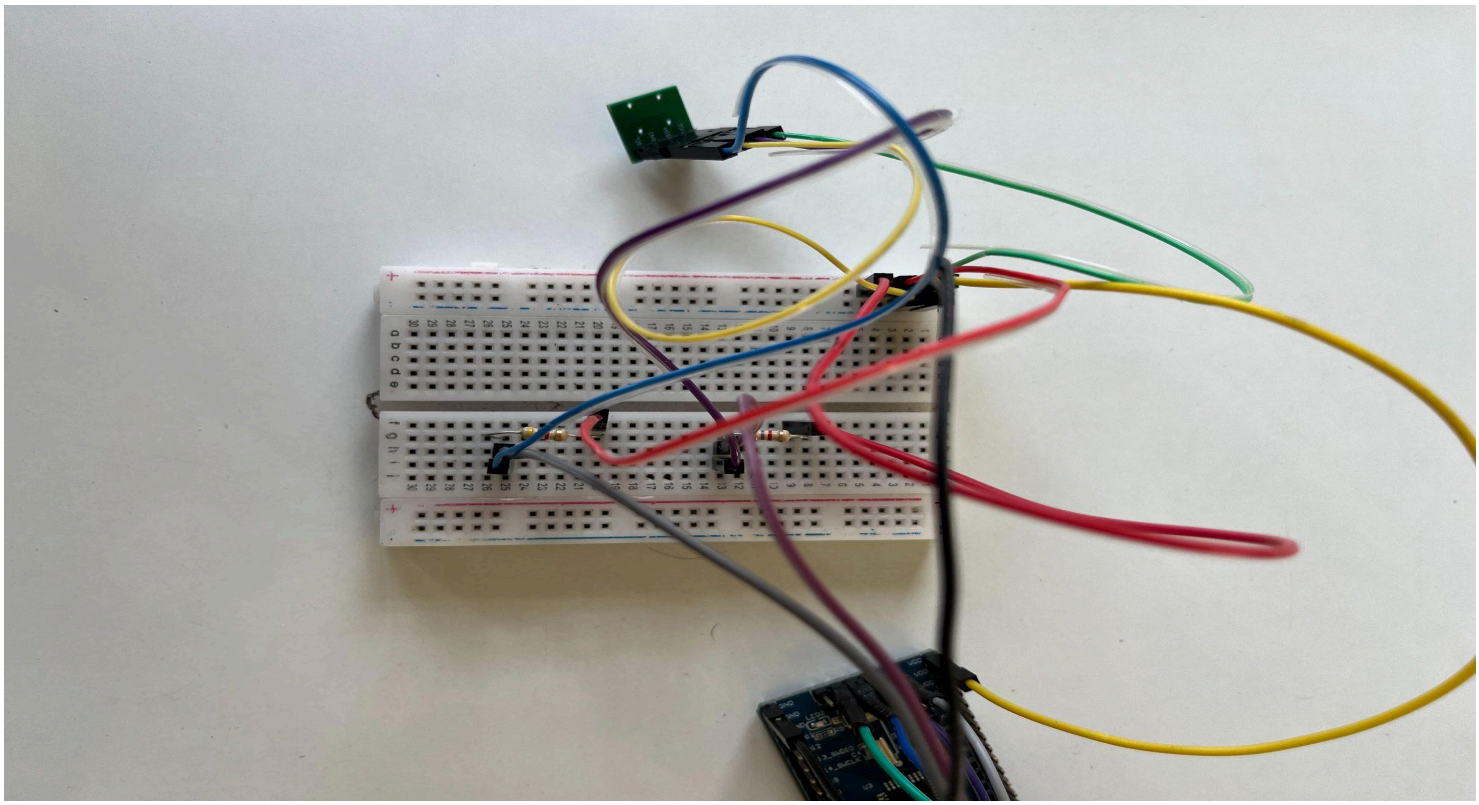
The sensor is initialized, measurements are triggered, and the results are read and displayed via UART.

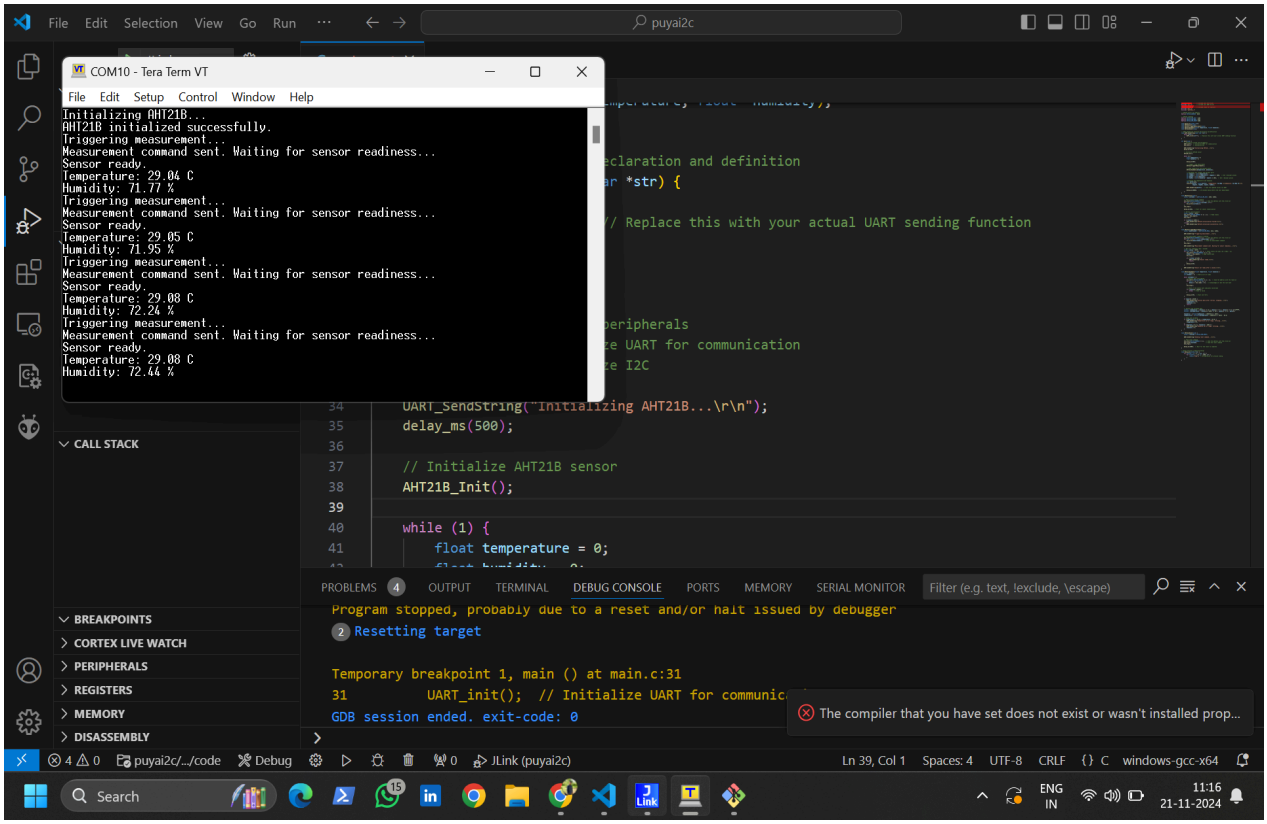
Data is formatted and sent as readable strings over UART.

There are functions to handle initialization, data retrieval, sensor resetting, and timing.

Images:

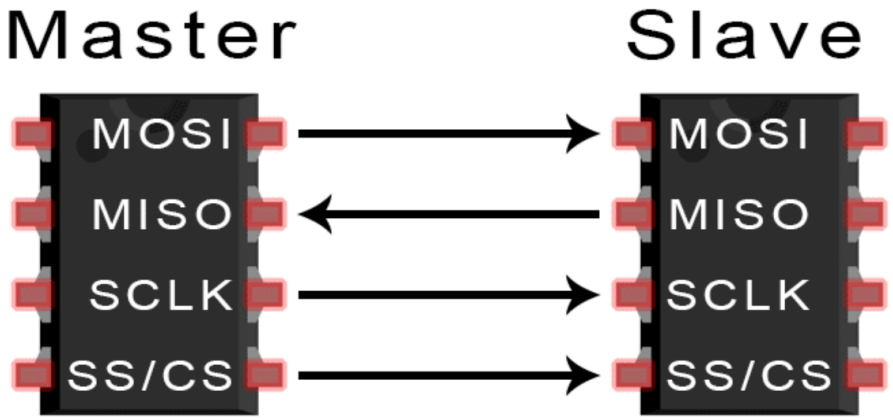

Honeywell



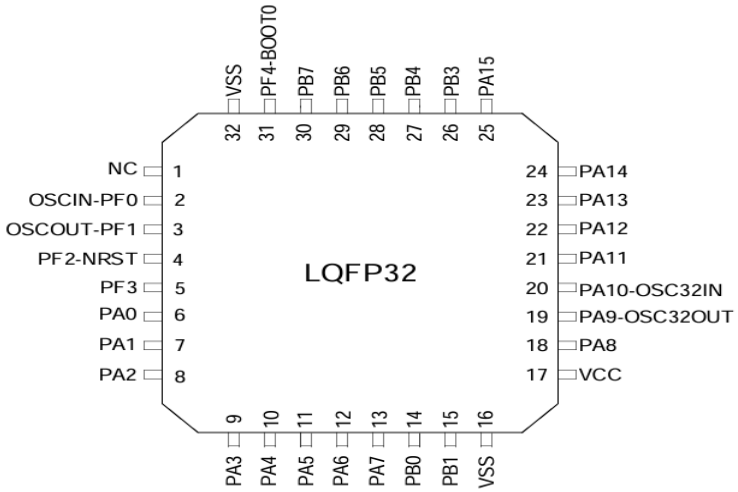

COM10 - Tera Term VT
File Edit Setup Control Window Help
Initializing AHT21B...
AHT21B initialized successfully.
Triggering measurement...
Measurement command sent. Waiting for sensor readiness...
Sensor ready.
Temperature: 29.04 C
Humidity: 71.77 %
Triggering measurement...
Measurement command sent. Waiting for sensor readiness...
Sensor ready.
Temperature: 29.05 C
Humidity: 71.95 %
Triggering measurement...
Measurement command sent. Waiting for sensor readiness...
Sensor ready.
Temperature: 29.08 C
Humidity: 72.24 %
Triggering measurement...
Measurement command sent. Waiting for sensor readiness...
Sensor ready.
Temperature: 29.08 C
Humidity: 72.44 %
UART_SendString("Initializing AHT21B...\r\n");
delay_ms(500);
// Initialize AHT21B sensor
AHT21B_Init();
while (1) {
float temperature = 0;
PROBLEMS 4 OUTPUT TERMINAL DEBUG CONSOLE PORTS MEMORY SERIAL MONITOR
Program stopped, probably due to a reset and/or halt issued by debugger
Resetting target
Temporary breakpoint 1, main () at main.c:31
GDB session ended. exit-code: 0
The compiler that you have set does not exist or wasn't installed prop...
CALL STACK
BREAKPOINTS
CORTEX LIVE WATCH
PERIPHERALS
REGISTERS
MEMORY
DISASSEMBLY
Ln 39, Col 1 Spaces: 4 UTF-8 CRLF C windows-gcc-x64
Search
ENG IN
11:16 21-11-2024

SERIAL PERIPHERAL INTERFACE (SPI) IN PY32F030 MCU

PY32F030 has two SPIs in it. SPIs offers half-duplex, full-duplex, and simplex synchronous serial communication among the chip and external devices. This interface provides the communication clock (SCK) for external slave devices and can be set up in master mode. Additionally, the interface can function with many masters set up.



PIN CONFIGURATION



LQFP32 Pinout1 PY32F030K1xT

7	7	7	7	PA1	I/O	COM		SPI1_SCK USART1_RTS USART2_RTS LED_DATA_C EVENTOUT	COMP1_INP ADC_IN1
6	6	6	6	PA0	I/O	COM		SPI2_SCK USART1_CTS LED_DATA_B USART2_CTS COMP1_OUT TIM1_CH3 TIM1_CH1N SPI1_MISO USART2_TX IR_OUT	ADC_IN0 COMP1_INM
13	13	13	13	PA7	I/O	COM		SPI1_MOSI TIM3_CH2 TIM1_CH1N TIM14_CH1 TIM17_CH1 EVENTOUT COMP2_OUT USART1_TX USART2_TX I2C_SDA SPI1_MISO	ADC_IN7

Four I/O pins are designated for SPI communication with external devices.

MISO: Master In/Slave Out data. Typically, this pin is utilized for slave mode data transmission and master mode data reception.

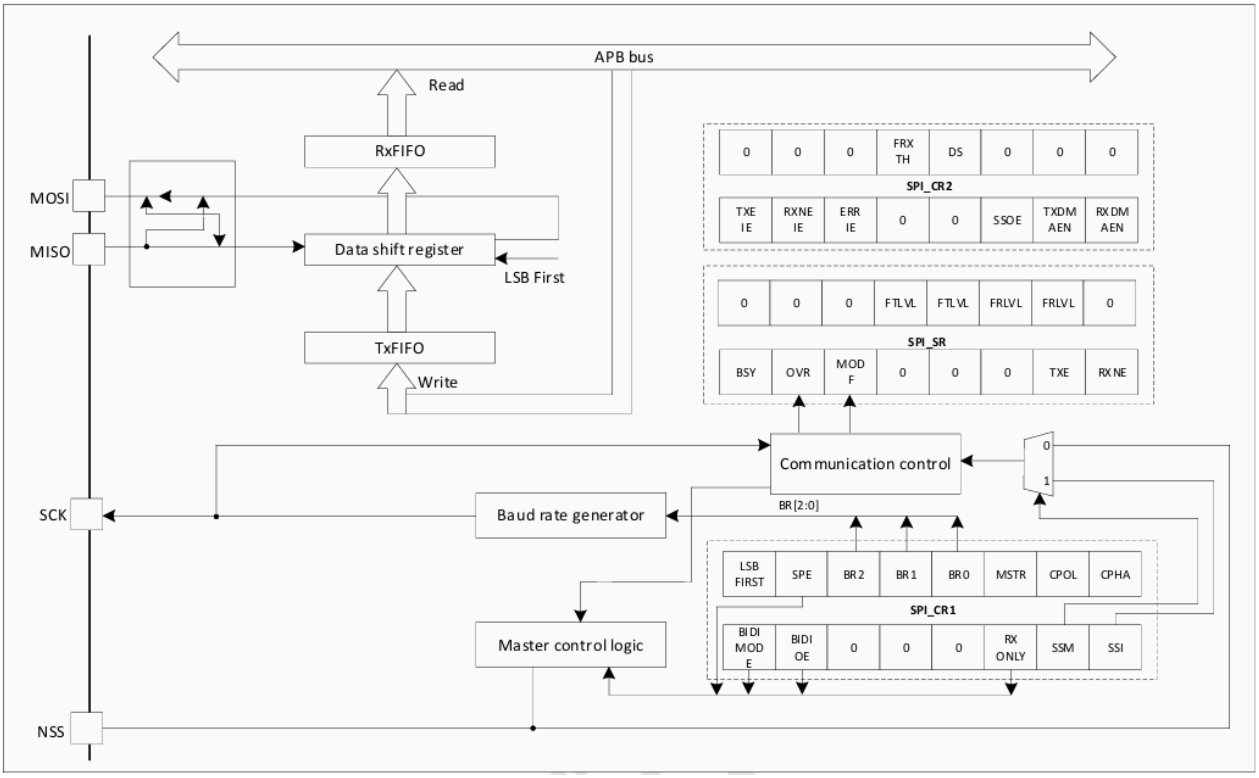
MOSI: Master Out/Slave In data. This pin typically transmits and receives data in master and slave modes.

SCK: Serial Clock output pin for SPI masters; input pin for SPI slaves.

NSS: Slave select pin. Depending on the SPI and NSS settings, this pin can be used for

- Select a single slave device for communication.
- Synchronize the data frame.
- Identify a dispute among many masters.

The SPI bus facilitates communication between a master device and one or more slave devices. The bus has at least two wires: one for clock signal and another for synchronous data transfer. Additional signals may be introduced based on data flow between SPI nodes and their slave select signal management.



SPI BLOCK DIAGRAM

CODE FOR SPI (PY32F030X8)
SPI.C:

```
#include"spi.h"

#define SPI_DR_8BIT *((volatile uint8_t*) &(SPI1->DR))  // for 8-bit data transfer

void SPI_init(void) {
  // Set GPIO pins
  #if SPI_MAP == 0
    // Setup GPIO pin PA1 (SCK), PA0 (MISO), PA7 (MOSI)
    RCC->IOPENR   |= RCC_IOPENR_GPIOAEN;
```

```
    GPIOA->MODER   = (GPIOA->MODER  & ~(((uint32_t)0b11<<(1<<1)) |
((uint32_t)0b11<<(0<<1)) | ((uint32_t)0b11<<(7<<1))))
                                    |  (((uint32_t)0b10<<(1<<1)) |
((uint32_t)0b10<<(0<<1)) | ((uint32_t)0b10<<(7<<1)));
    GPIOA->OTYPER &=                  ~(((uint32_t)0b1 <<(1<<0)) |
((uint32_t)0b1 <<(7<<0)));
    GPIOA->PUPDR   = (GPIOA->PUPDR  & ~(
((uint32_t)0b11<<(0<<1))                            ))
                                    |  (
((uint32_t)0b01<<(0<<1))                            );
    GPIOA->AFR[0]  = (GPIOA->AFR[0] & ~(((uint32_t)0xf <<(1<<2)) | ((uint32_t)0xf
<<(0<<2)) | ((uint32_t)0xf <<(7<<2))))
                                    |  (((uint32_t)0   <<(1<<2)) | ((uint32_t)10
<<(0<<2)) | ((uint32_t)0   <<(7<<2)));
  #elif SPI_MAP == 1
    // Setup GPIO pin PA2 (SCK), PA0 (MISO), PA1 (MOSI)
    RCC->IOPENR   |= RCC_IOPENR_GPIOAEN;
    GPIOA->MODER   = (GPIOA->MODER  & ~(((uint32_t)0b11<<(2<<1)) |
((uint32_t)0b11<<(0<<1)) | ((uint32_t)0b11<<(1<<1))))
                                    |  (((uint32_t)0b10<<(2<<1)) |
((uint32_t)0b10<<(0<<1)) | ((uint32_t)0b10<<(1<<1)));
    GPIOA->OTYPER &=                  ~(((uint32_t)0b1 <<(2<<0)) |
((uint32_t)0b1 <<(1<<0)));
    GPIOA->PUPDR   = (GPIOA->PUPDR  & ~(
((uint32_t)0b11<<(0<<1))                            ))
                                    |  (
((uint32_t)0b01<<(0<<1))                            );
    GPIOA->AFR[0]  = (GPIOA->AFR[0] & ~(((uint32_t)0xf <<(2<<2)) | ((uint32_t)0xf
<<(0<<2)) | ((uint32_t)0xf <<(1<<2))))
                                    |  (((uint32_t)10  <<(2<<2)) | ((uint32_t)10
<<(0<<2)) | ((uint32_t)10  <<(1<<2)));
  #elif SPI_MAP == 2
    // Setup GPIO pin PA1 (SCK), PA0 (MISO), PA2 (MOSI)
    RCC->IOPENR   |= RCC_IOPENR_GPIOAEN;
    GPIOA->MODER   = (GPIOA->MODER  & ~(((uint32_t)0b11<<(1<<1)) |
((uint32_t)0b11<<(0<<1)) | ((uint32_t)0b11<<(2<<1))))
```

```
                                    |  (((uint32_t)0b10<<(1<<1)) |
((uint32_t)0b10<<(0<<1)) | ((uint32_t)0b10<<(2<<1)));
    GPIOA->OTYPER &=                  ~(((uint32_t)0b1 <<(1<<0)) |
((uint32_t)0b1 <<(2<<0)));
    GPIOA->PUPDR   = (GPIOA->PUPDR  & ~(
((uint32_t)0b11<<(0<<1))                              ))
                                    |  (
((uint32_t)0b01<<(0<<1))                              );
    GPIOA->AFR[0]  = (GPIOA->AFR[0] & ~(((uint32_t)0xf <<(1<<2)) | ((uint32_t)0xf
<<(0<<2)) | ((uint32_t)0xf <<(2<<2))))
                                    |  (((uint32_t)0   <<(1<<2)) | ((uint32_t)10
<<(0<<2)) | ((uint32_t)0   <<(2<<2)));
  #elif SPI_MAP == 3
    // Setup GPIO pin PA5 (SCK), PA6 (MISO), PA7 (MOSI)
    RCC->IOPENR   |= RCC_IOPENR_GPIOAEN;
    GPIOA->MODER   = (GPIOA->MODER  & ~(((uint32_t)0b11<<(5<<1)) |
((uint32_t)0b11<<(6<<1)) | ((uint32_t)0b11<<(7<<1))))
                                    |  (((uint32_t)0b10<<(5<<1)) |
((uint32_t)0b10<<(6<<1)) | ((uint32_t)0b10<<(7<<1)));
    GPIOA->OTYPER &=                  ~(((uint32_t)0b1 <<(5<<0)) |
((uint32_t)0b1 <<(7<<0)));
    GPIOA->PUPDR   = (GPIOA->PUPDR  & ~(
((uint32_t)0b11<<(6<<1))                              ))
                                    |  (
((uint32_t)0b01<<(6<<1))                              );
    GPIOA->AFR[0] &=                  ~(((uint32_t)0xf <<(5<<2)) | ((uint32_t)0xf
<<(6<<2)) | ((uint32_t)0xf <<(7<<2)));
  #elif SPI_MAP == 4
    // Setup GPIO pin PA9 (SCK), PA13 (MISO), PA8 (MOSI)
    RCC->IOPENR   |= RCC_IOPENR_GPIOAEN;
    GPIOA->MODER   = (GPIOA->MODER  & ~(((uint32_t)0b11<<(9<<1)) |
((uint32_t)0b11<<(13<<1)) | ((uint32_t)0b11<<(8<<1))))
                                    |  (((uint32_t)0b10<<(9<<1)) |
((uint32_t)0b10<<(13<<1)) | ((uint32_t)0b10<<(8<<1)));
    GPIOA->OTYPER &=                  ~(((uint32_t)0b1 <<(9<<0)) |
((uint32_t)0b1 <<(8<<0)));
```

```
    GPIOA->PUPDR   = (GPIOA->PUPDR  & ~(
((uint32_t)0b11<<(13<<1))                          ))
                                    |  (
((uint32_t)0b01<<(13<<1))                          );
    GPIOA->AFR[1]  = (GPIOA->AFR[1] & ~(((uint32_t)0xf <<(1<<2)) | ((uint32_t)0xf <<(
5<<2)) | ((uint32_t)0xf <<(0<<2))))
                                    |  (((uint32_t)10  <<(1<<2)) | ((uint32_t)10  <<(
5<<2)) | ((uint32_t)10  <<(0<<2)));
  #elif SPI_MAP == 5
    // Setup GPIO pin PA9 (SCK), PA13 (MISO), PA12 (MOSI)
    RCC->IOPENR   |= RCC_IOPENR_GPIOAEN;
    GPIOA->MODER   = (GPIOA->MODER  & ~(((uint32_t)0b11<<(9<<1)) |
((uint32_t)0b11<<(13<<1)) | ((uint32_t)0b11<<(12<<1))))
                                    |  (((uint32_t)0b10<<(9<<1)) |
((uint32_t)0b10<<(13<<1)) | ((uint32_t)0b10<<(12<<1)));
    GPIOA->OTYPER &=                  ~(((uint32_t)0b1 <<(9<<0)) |
((uint32_t)0b1 <<(12<<0)));
    GPIOA->PUPDR   = (GPIOA->PUPDR  & ~(
((uint32_t)0b11<<(13<<1))                          ))
                                    |  (
((uint32_t)0b01<<(13<<1))                          );
    GPIOA->AFR[1]  = (GPIOA->AFR[1] & ~(((uint32_t)0xf <<(1<<2)) | ((uint32_t)0xf <<(
5<<2)) | ((uint32_t)0xf <<( 4<<2))))
                                    |  (((uint32_t)10  <<(1<<2)) | ((uint32_t)10  <<(
5<<2)) | ((uint32_t)0   <<( 4<<2)));
  #else
    #warning No automatic pin mapping for SPI
  #endif

  // Setup and enable SPI master, standard configuration
  RCC->APBENR2 |= RCC_APBENR2_SPI1EN;
  SPI1->CR1     = (SPI_PRESC << 3)    // set prescaler
                | SPI_CR1_MSTR        // master configuration
                | SPI_CR1_SSM         // software control of NSS
                | SPI_CR1_SSI         // set internal NSS high
                | SPI_CR1_SPE;        // enable SPI
  SPI1->CR2     = SPI_CR2_FRXTH;      // set RXNE flag after 8-bit
```

```
}

// Transfer one data byte (read and write)
uint8_t SPI_transfer(uint8_t data) {
  SPI_DR_8BIT = data;                  // send data byte
  while(!SPI_available());             // wait for data byte received
  return(SPI_DR_8BIT);                 // return received data byte
}
```

SPI.H

```
// ================================================================================
// Basic SPI Master Functions for PY32F0xx                                 * v1.0 *
// ================================================================================
//
// Functions available:
// --------------------
// SPI_init()               Init SPI with defined clock rate (see below)
// SPI_transfer(d)          Transmit and receive one data byte
//
// SPI_busy()               Check if SPI bus is busy
// SPI_ready()              Check if SPI is ready to write
// SPI_available()          Check if something was received
// SPI_enable()             Enable SPI module
// SPI_disable()            Disable SPI module
// SPI_setBAUD(n)           Set BAUD rate (see below)
// SPI_setCPOL(n)           0: SCK low in idle, 1: SCK high in idle
// SPI_setCPHA(n)           Start sampling from 0: first clock edge, 1: second clock
edge
//
// SPI pin mapping (set below in SPI parameters):
// ----------------------------------------------
// SPI_MAP    0     1     2     3     4     5     6
// SCK-pin   PA1   PA2   PA1   PA5   PA9   PA9   no mapping
// MISO-pin  PA0   PA0   PA0   PA6   PA13  PA13  no mapping
// MOSI-pin  PA7   PA1   PA2   PA7   PA8   PA12  no mapping
//
```

```
// Slave select pins (NSS) must be defined and controlled by the application.
// SPI clock rate must be defined below.
//
// 2023 by Stefan Wagner:   https://github.com/wagiminator

#ifndef SPI_H
#define SPI_H

#ifdef __cplusplus
extern "C" {
#endif
#include "system.h"

// SPI Parameters
#define SPI_PRESC           5     // SPI_CLKRATE = F_CPU / (2 << SPI_PRESC)
#define SPI_MAP             0     // SPI pin mapping (see above)

// I2C Functions and Macros
#define SPI_busy()          (SPI1->SR & SPI_SR_BSY)
#define SPI_ready()         (SPI1->SR & SPI_SR_TXE)
#define SPI_available()     (SPI1->SR & SPI_SR_RXNE)

#define SPI_enable()        SPI1->CR1 |=  SPI_CR1_SPE
#define SPI_disable()       SPI1->CR1 &= ~SPI_CR1_SPE
#define SPI_setCPOL(n)      (n)?(SPI1->CR1|=SPI_CR1_CPOL):(SPI1->CR1&=~SPI_CR1_CPOL)
#define SPI_setCPHA(n)      (n)?(SPI1->CR1|=SPI_CR1_CPHA):(SPI1->CR1&=~SPI_CR1_CPHA)
#define SPI_setBAUD(n)      SPI1->CR1 = (SPI1->CR1&~SPI_CR1_BR) | (((n)&7)<<3)

void SPI_init(void);
uint8_t SPI_transfer(uint8_t data);

#ifdef __cplusplus
}
#endif

#endif // SPI_H
```