



Technology Innovator

Puya

Touch Application Development Guide Based on the PY32 Software Library



Puya Semiconductor (Shanghai) Co., Ltd.

Content

1. Introduction	4
2. Software Library Structure	5
3. Touch Library Introduction and Usage	8
3.1. Introduction to Touch Function Categories	8
3.2. MCU Resource Requirements	8
3.2.1. MCU Hardware Resources	8
3.2.2. MCU Software Resources	8
3.3. Touch Library Interface Analysis	10
3.3.1. Data Interface	10
3.3.2. Function Interface	11
3.4. Touch Library Function Options and Parameter Configuration	12
3.4.1. Touch Keys User Configuration	12
3.4.2. Slider/Wheel User Configuration	13
3.4.3. Touch Function and Parameter Configuration	15
3.5. User-Modifiable Touch Library Callback Functions	16
3.6. Touch Application Configuration Steps and Process	17
3.6.1. Conventional Keys Application	17
3.6.2. Slider/Wheel Application	23
3.6.3. Touch Low-Power Application	25
3.6.4. Floating Keys Application	26
3.6.5. Waterproof Keys Application	27
3.6.6. Water Detection Application Using Touch	29
4. System Configuration and Peripheral Application Analysis	31
4.1. System Configuration	31
4.1.1. Clock Configuration	31
4.1.2. Option Bytes	31
4.2. Peripheral Application Interfaces and Usage	31
4.2.1. UART	31
4.2.2. Timer	33
4.2.3. PWM	35
4.2.4. ADC	36
4.2.5. I2C (Slave Mode)	37
4.2.6. RTC	38
4.2.7. Watchdog	39
4.3. Typical Application Drivers	40
4.3.1. Digital Tube/LED Driver	40
4.3.2. Dedicated Driver IC for Digital Tubes/LEDs	42
4.3.3. Infrared Remote Control Reception	43
4.3.4. User Data Power-Down Storage	43

4.3.5.	W2812B-like LED Driver	45
4.3.6.	Buzzer Control	46
5.	User Application Program Interface	47
6.	Version history	48

Puya Confidential

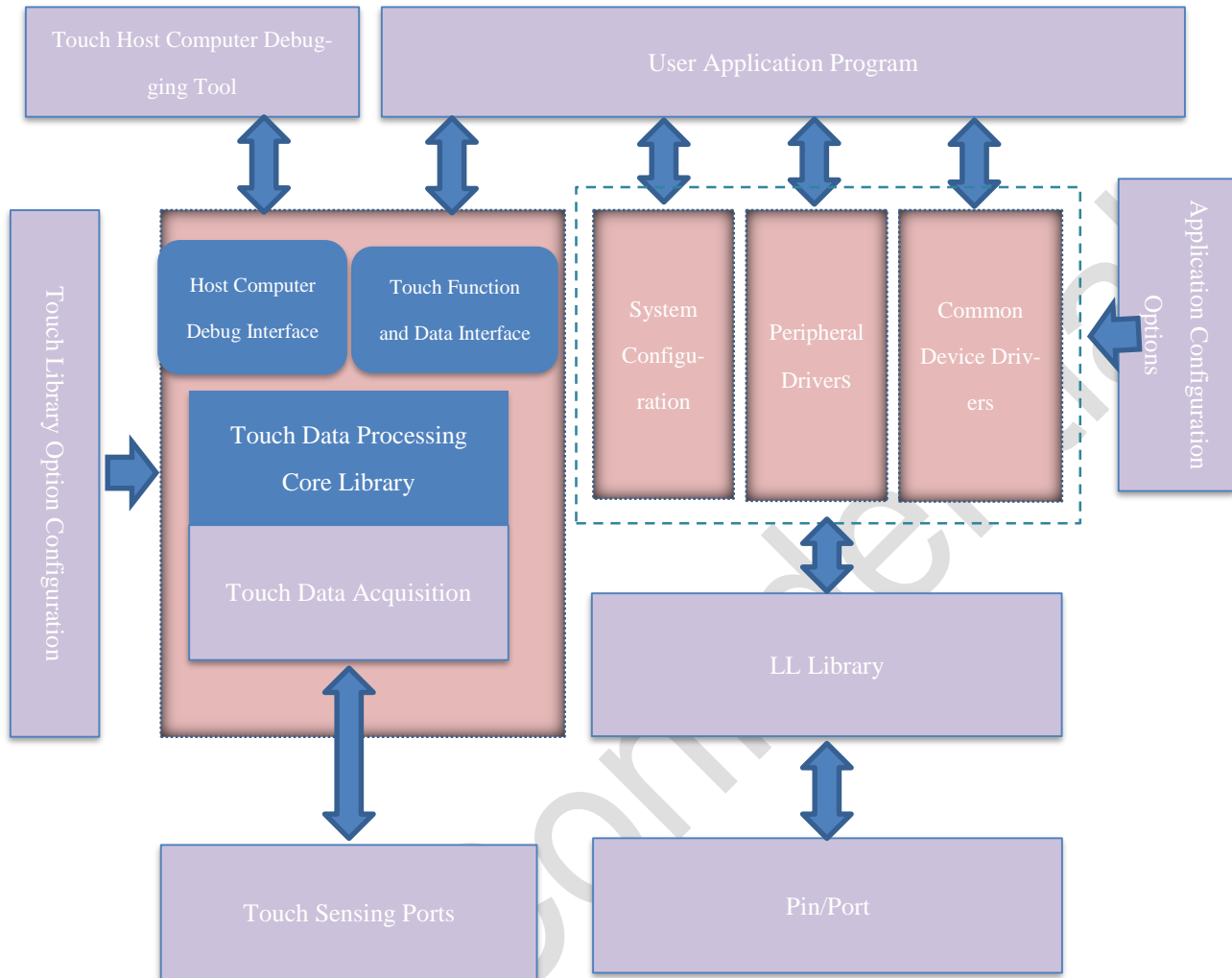
1. Introduction

The PY32 standard software library (PY32_Touch_Library_Vxx) integrates system configuration, touch library, peripheral library (LL Library), application examples based on the LL Library, and drivers for commonly used external components. This project employs a visual configuration interface, allowing initialization and parameter configuration for the system, touch, peripherals, and external devices by checking options and filling in parameters. Based on this project for application development, users do not need to concern themselves with the initialization processes at the chip level for the system, peripherals, and TK (Touch Keys). They only need to call the function and data interfaces of the corresponding modules to utilize the chip's functionalities. This enables developers to focus solely on product feature development, making the application development process more intuitive and efficient. Even developers unfamiliar with Puya MCUs can easily get started and become proficient.

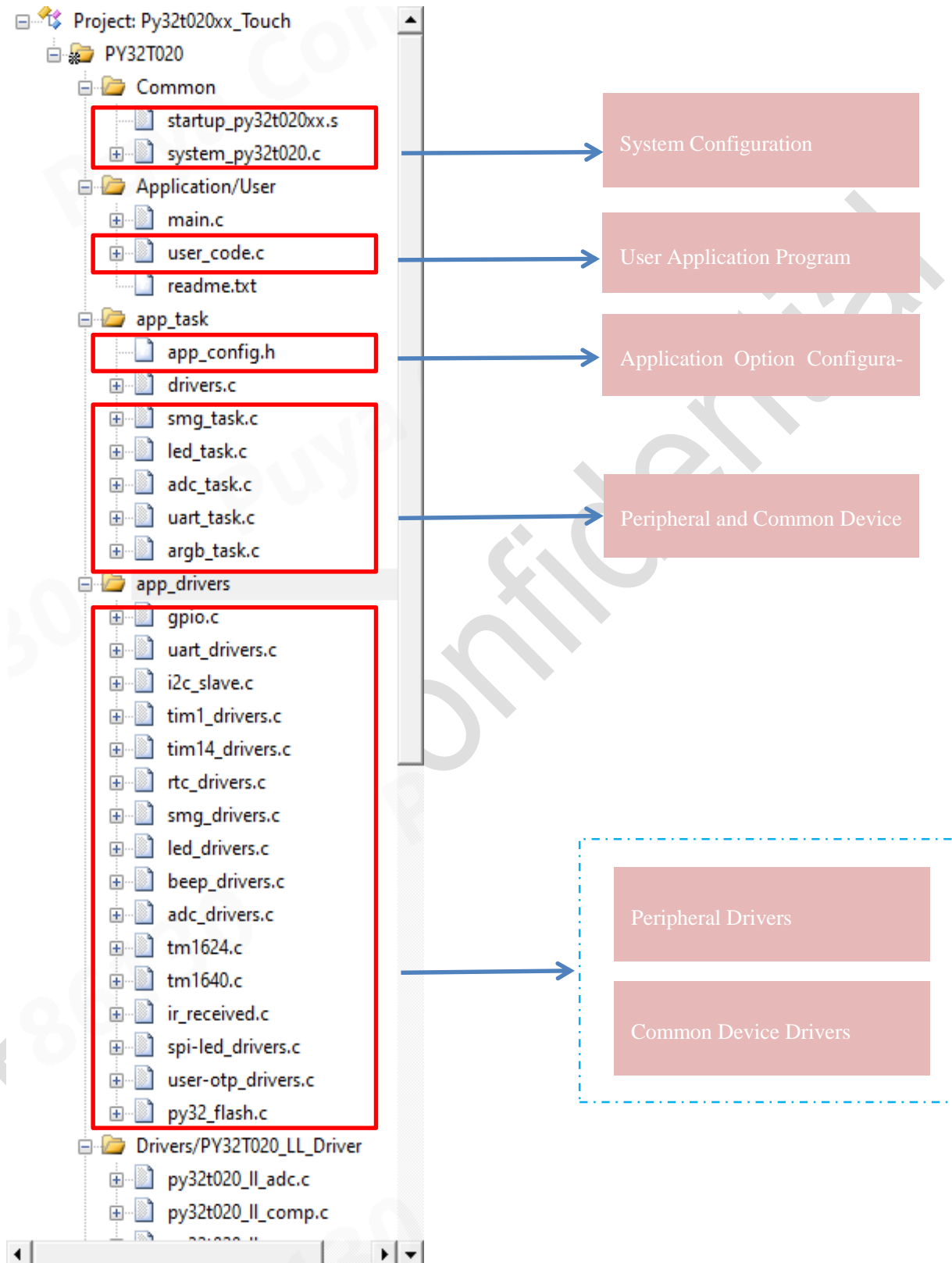
This document will provide a detailed introduction to the definitions and usage methods of each functional module in this project, giving developers a comprehensive understanding of the application development process based on the standard project.

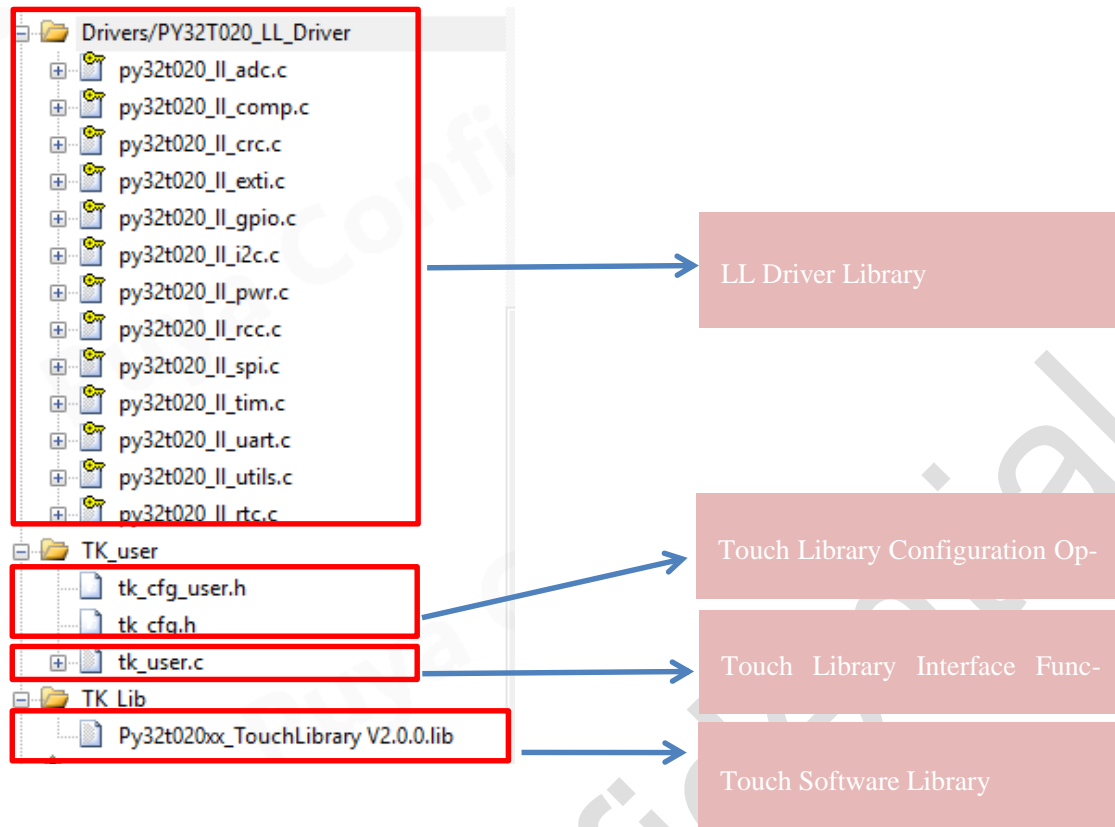
2. Software Library Structure

The basic structure of the software library is shown in the following figure:



The correspondence between the main modules in the above software structure diagram and the software project directory is as follows:





3. Touch Library Introduction and Usage

3.1. Introduction to Touch Function Categories

The touch library supports the following applications:

- Conventional Keys (Sensing components: springs, pads, conductive foam, etc.)
- Waterproof Keys
- Slider/Wheel
- Floating Keys
- Touch Water Detection
- Low-Power Applications

3.2. MCU Resource Requirements

3.2.1. MCU Hardware Resources

The PY32 touch library requires the following hardware resources:

- SYSTICK timer (non-exclusive)
- One general-purpose IO per touch channel
- Waterproof Shield channel requires one channel IO
- RTC required in low-power mode

3.2.2. MCU Software Resources

The FLASH and SRAM resources occupied by the PY32 touch library are shown in the table below:

Touch Function	FLASH	SRAM	Channels	Remarks
Conventional Keys	5.83K	1.15K	8(Key)	SRAM increases by approx. 30 bytes per additional key
Floating Keys	6.47K	1.27K	6(Key)	
Waterproof Keys	7.08K	1.38K	9(Key)+1(Shield)	
Wheel + Slider + Keys	8.00K	1.29K	2(Key)+4(Wheel)+4(Slider)	
Slider + Wheel	6.96K	1.23K	4(Wheel)+4(Slider)	
Touch Water Detection	5.79K	1.06K	1(Ref)+1(Detect)	

Note: The SRAM usage in the above table includes a 512-byte stack.

The time required for PY32 touch library sampling and processing is shown in the table below:

Touch Function	Scan Mode	Main Frequency (MHz)	Channels	Window Setting	Single Channel Scan Time (us)	TK Interrupt Response Time (us)	TK Data Processing Time (us)
Conventional Keys	Conventional Scan	24	8	12000	1000	195	150
		48			1000	135	105
	Synchronous Scan	24	6	2400*20	4000	3148	143
		48			4000	2245	85
Waterproof Keys	Conventional Scan	24	8+1	6000	750	395	295
		48				277	205
Slider/Wheel	Conventional Scan	24	4	6000	500	92	102
		48				63	71
Water Detection	Conventional Scan	24	2	12000	500	15	73
		48				10	52

Explanation:

1. TK sampling time is proportional to the window setting value and independent of other factors.
2. With a determined main frequency, the TK interrupt time and TK data processing time are basically proportional to the number of channels.
3. Taking conventional keys scanning with a main frequency of 24M as an example, the total sampling time for 8 channels is: $8 * 1000\text{us} = 8000\text{us}$. The total interrupt response + data processing time is: $195 + 150 = 345\text{us}$. Since TK sampling does not occupy CPU time, the proportion of CPU time occupied by the TK task is: $345 / 8000 = 4.3\%$.

3.3. Touch Library Interface Analysis

3.3.1. Data Interface

Application Category	Variable Name	Variable Type	Description
Conventional Keys	TKCtr.KeyFlags	unsigned int	TKCtr.KeyFlags is a 32-bit unsigned integer. Each bit represents the trigger status of a touch key. Bits 0 to 25 correspond to KEY0 to KEY25, respectively. The user program can check the value of the corresponding bit to obtain the key status and execute the corresponding operation.
Waterproof Keys			
Floating Keys			
Slider/Wheel	TKCtr.SliderOrWheelPosition[n]	signed short int	TKCtr.SliderOrWheelPosition[n] represents the position information of the n-th Slider/Wheel. When the Slider/Wheel is triggered, the position range is from 0 to the set resolution. A value of -1 for TKTcr.SliderOrWheelPosition[n] indicates no trigger for the Slider/Wheel.
Touch Water Detection	TKCtr.DetectOut	unsigned char	TKCtr.DetectOut is an 8-bit unsigned integer. Each bit represents the water status of a water detection channel. A value of 1 indicates the presence of water in the corresponding channel.

3.3.2. Function Interface

Function Name	Input Parameters	Return Value	Description
TK_Init	None	None	Touch function initialization API function. Must be called before entering the main loop.
TK_MainFsm	None	None	Touch function main state machine API function. Should be called within the main loop.
TK_Timer-Handler	uint8_t ms ms indicates the time interval for calling this function, in milliseconds.	None	This function is called within the timer interrupt service routine (ISR) to provide a time base for the touch functionality.

```

int main(void)
{
    SystemClockConfig();
    /* System configuration initialization code start */
    app_drivers_init();
    /* System configuration initialization code end */
    /* Touch initialization start */
    #if APP_TK_ENABLE
    TK_Init();
    #endif
    /* Touch initialization end */
    /* User initialization code start */
    user_init();
    /* User initialization code end */
    log_printf("start loop\r\n");
    while (1)
    {
        #if APP_TK_ENABLE
        /* Touch state machine processing */
        TK_MainFsm();
        /* Touch key processing */
        TK_Loop();
        #endif
        app_drivers_loop();
        user_loop();
    }
}

void SysTick_Handler(void)
{
    static uint16_t Prescaler;
    #if (SYSTICK_DEBUG_GPIO != NO_PIN && SYSTICK_DEBUG)
    GPIO_ToggleBit(SYSTICK_DEBUG_GPIO);
    #endif
    Prescaler++;
    if (Prescaler >= (1000 / SYSTICK_TIMING_TIME))
    {
        Prescaler = 0;
        app_drivers_timer();
        #if APP_TK_ENABLE
        TK_TimerHandler(1);
        #endif
    }
    #if APP_BEEP_ENABLE
    BEEP_Toggle();
    #endif
    #if (APP_IR_RECEIVED_ENABLE == 1 && D_IR_TIM == 0)
    IR_Received_Scan();
    #endif
    #if APP_LED_ENABLE
    LED_Scan();
    #endif
    user_timer();
}

```

3.4. Touch Library Function Options and Parameter Configuration

3.4.1. Touch Keys User Configuration

For touch keys applications, the touch channels and key trigger threshold values should be configured first. These two configurations are implemented in `tk_cfg_user.h`, as shown in the following figure:

Puya Confidential

```

#ifndef _TK_CFG_USER_H
#define _TK_CFG_USER_H

// Key touch channel definition, fill in according to the KEY order. Unused keys must be defined as TK_CH_NONE
#define CH_KEY0    TK_CH17
#define CH_KEY1    TK_CH15
#define CH_KEY2    TK_CH11
#define CH_KEY3    TK_CH5
#define CH_KEY4    TK_CH16
#define CH_KEY5    TK_CH10
#define CH_KEY6    TK_CH7
#define CH_KEY7    TK_CH6
#define CH_KEY8    TK_CH_NONE
#define CH_KEY9    TK_CH_NONE
#define CH_KEY10   TK_CH_NONE
#define CH_KEY11   TK_CH_NONE
#define CH_KEY12   TK_CH_NONE
#define CH_KEY13   TK_CH_NONE
#define CH_KEY14   TK_CH_NONE
#define CH_KEY15   TK_CH_NONE
#define CH_KEY16   TK_CH_NONE
#define CH_KEY17   TK_CH_NONE
#define CH_KEY18   TK_CH_NONE
#define CH_KEY19   TK_CH_NONE
#define CH_KEY20   TK_CH_NONE
#define CH_KEY21   TK_CH_NONE
#define CH_KEY22   TK_CH_NONE
#define CH_KEY23   TK_CH_NONE
#define CH_KEY24   TK_CH_NONE
#define CH_KEY25   TK_CH_NONE

//-----

//-----
// Key trigger threshold definition
#define KEY0_THD    100
#define KEY1_THD    100
#define KEY2_THD    100
#define KEY3_THD    100
#define KEY4_THD    100
#define KEY5_THD    100
#define KEY6_THD    100
#define KEY7_THD    100
#define KEY8_THD    40
#define KEY9_THD    40
#define KEY10_THD   40
#define KEY11_THD   40
#define KEY12_THD   40
#define KEY13_THD   40
#define KEY14_THD   40
#define KEY15_THD   40
#define KEY16_THD   40
#define KEY17_THD   40
#define KEY18_THD   40
#define KEY19_THD   40
#define KEY20_THD   40
#define KEY21_THD   40
#define KEY22_THD   40
#define KEY23_THD   40
#define KEY24_THD   40
#define KEY25_THD   40

//=====

```

CH_KEYn corresponds to KEYn_THD. CH_KEYn defines the channel assignment for KEYn, while KEYn_THD represents the trigger threshold value for KEYn. When the data variation of the channel defined by CH_KEYn exceeds KEYn_THD, KEYn will be triggered.

3.4.2. Slider/Wheel User Configuration

Slider/Wheel applications require configuration of application type, channels, resolution, and trigger threshold values, defined in `tk_cfg_user.h`, as shown below:

```

//=====
#define SLIDER_OR_WHEEL0_TYPE          TK_APP_WHEEL          // Slider type
#define SLIDER_OR_WHEEL0_RESOLUTION    360                  // Slider resolution
#define SLIDER_OR_WHEEL0_THD           101                  // Slider threshold
#define SLIDER_OR_WHEEL0_CH0           TK_CH4               // Slider channels, fill in sequentially
#define SLIDER_OR_WHEEL0_CH1           TK_CH8
#define SLIDER_OR_WHEEL0_CH2           TK_CH5
#define SLIDER_OR_WHEEL0_CH3           TK_CH6
#define SLIDER_OR_WHEEL0_CH4           TK_CH7
#define SLIDER_OR_WHEEL0_CH5           TK_CH_NONE
#define SLIDER_OR_WHEEL0_CH6           TK_CH_NONE
#define SLIDER_OR_WHEEL0_CH7           TK_CH_NONE

#define SLIDER_OR_WHEEL1_TYPE          TK_APP_NONE
#define SLIDER_OR_WHEEL1_RESOLUTION    255
#define SLIDER_OR_WHEEL1_THD           450
#define SLIDER_OR_WHEEL1_CH0           TK_CH19
#define SLIDER_OR_WHEEL1_CH1           TK_CH23
#define SLIDER_OR_WHEEL1_CH2           TK_CH24
#define SLIDER_OR_WHEEL1_CH3           TK_CH25
#define SLIDER_OR_WHEEL1_CH4           TK_CH_NONE
#define SLIDER_OR_WHEEL1_CH5           TK_CH_NONE
#define SLIDER_OR_WHEEL1_CH6           TK_CH_NONE
#define SLIDER_OR_WHEEL1_CH7           TK_CH_NONE

#define SLIDER_OR_WHEEL2_TYPE          TK_APP_NONE
#define SLIDER_OR_WHEEL2_RESOLUTION    255
#define SLIDER_OR_WHEEL2_THD           80
#define SLIDER_OR_WHEEL2_CH0           TK_CH_NONE
#define SLIDER_OR_WHEEL2_CH1           TK_CH_NONE
#define SLIDER_OR_WHEEL2_CH2           TK_CH_NONE
#define SLIDER_OR_WHEEL2_CH3           TK_CH_NONE
#define SLIDER_OR_WHEEL2_CH4           TK_CH_NONE
#define SLIDER_OR_WHEEL2_CH5           TK_CH_NONE
#define SLIDER_OR_WHEEL2_CH6           TK_CH_NONE
#define SLIDER_OR_WHEEL2_CH7           TK_CH_NONE

#define SLIDER_OR_WHEEL3_TYPE          TK_APP_NONE
#define SLIDER_OR_WHEEL3_RESOLUTION    255
#define SLIDER_OR_WHEEL3_THD           80
#define SLIDER_OR_WHEEL3_CH0           TK_CH_NONE
#define SLIDER_OR_WHEEL3_CH1           TK_CH_NONE
#define SLIDER_OR_WHEEL3_CH2           TK_CH_NONE
#define SLIDER_OR_WHEEL3_CH3           TK_CH_NONE
#define SLIDER_OR_WHEEL3_CH4           TK_CH_NONE
#define SLIDER_OR_WHEEL3_CH5           TK_CH_NONE
#define SLIDER_OR_WHEEL3_CH6           TK_CH_NONE
#define SLIDER_OR_WHEEL3_CH7           TK_CH_NONE
//=====

```

Slider/Wheel Type Definition,
The slider is defined as TK_APP_SLIDER, and the wheel is defined as TK_APP_WHEEL. If defined as TK_APP_NONE, it indicates that the current slider/wheel is disabled. This touch software library supports up to 4 sliders/wheels. Note: Sliders/wheels must be defined sequentially from 0 to 3. For example, if two sliders/wheels are enabled, only SLIDER_OR_WHEEL0 and SLIDER_OR_WHEEL1 can be defined, whereas defining SLIDER_OR_WHEEL0 and SLIDER_OR_WHEEL2 (skipping index 1) is not allowed.

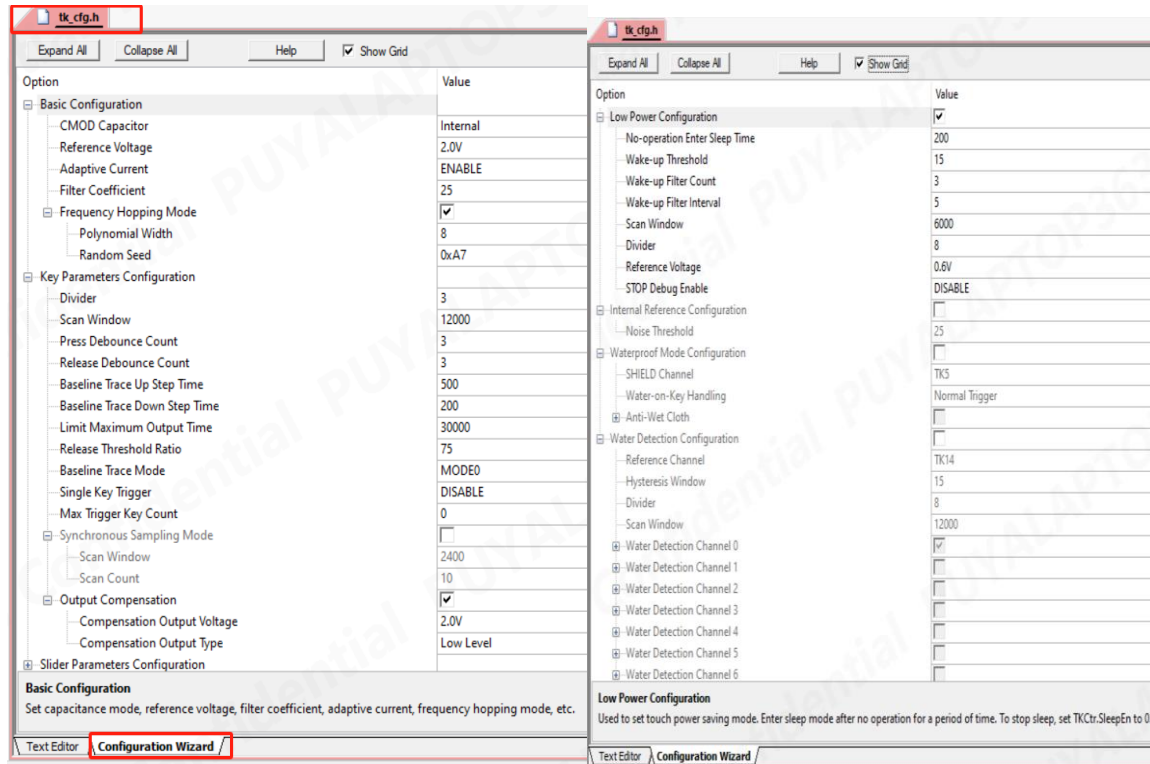
Slider/Wheel Resolution Definition,
If the resolution is defined as 360, the output position range will be: 0 to 359.

Slider/Wheel Trigger Threshold Definition,
The slider/wheel will be triggered if the sum of the data variations from any two of its channels exceeds this configured threshold value.

Slider/Wheel Channel Definition,
Each slider/wheel supports a maximum of 8 channels. The channels must be defined sequentially in the configuration.

3.4.3. Touch Function and Parameter Configuration

Touch parameters and function selections can be defined in tk_cfg.h. In practice, it is not necessary to edit tk_cfg.h directly. Instead, configuration is done through its Configuration Wizard. The Configuration Wizard is a graphical configuration menu that allows for configuration by checking boxes, selecting from dropdown menus, or entering parameters.



3.5. User-Modifiable Touch Library Callback Functions

The touch software library exposes some function interfaces in the form of callback functions for users to modify to meet more application requirements. These callback functions are stored in the tk_user.c file. Commonly used callback functions are listed in the table below:

Function Name	Input Parameters	Return Value	Description
EnterStop_Callback	None	None	This function is called before the touch system enters low-power mode. Users can add custom code here for any necessary operations required prior to entering the low-power state.
ExitStop_Callback	None	None	This function is called after the touch system exits low-power mode. Users can add custom code here for any necessary operations required after waking up from the low-power state.
LoopStop_Callback	None	None	This function is called after data acquisition completes and the system wakes up within the low-power mode. Users can add code for periodic processing tasks that need to be executed during the low-power mode.
TK_RawDataCompensate	Parameter Name: chs Data Type: uint8_t Parameter Meaning: Channel Index Parameter Name: raw Data Type: uint16_t Parameter Meaning: Touch Data	Compensated touch data	Used to compensate raw touch data. For example, if touch data fluctuates synchronously with LED state changes (on/off), users can implement compensation logic within this function based on the current LED status.
FilterExDeal	Parameter Name: chs Data Type: uint8_t Parameter Meaning: Channel Index	None	This function is called as callback during the touch data filtering process. Users can adjust filtering parameters inside this function based on real-world measurements and conditions.
APP_TouchKeyFlags-Mask	None	None	This function is used for judging and handling abnormal states. For instance, after a key press is detected, it can check for the presence of interference. If

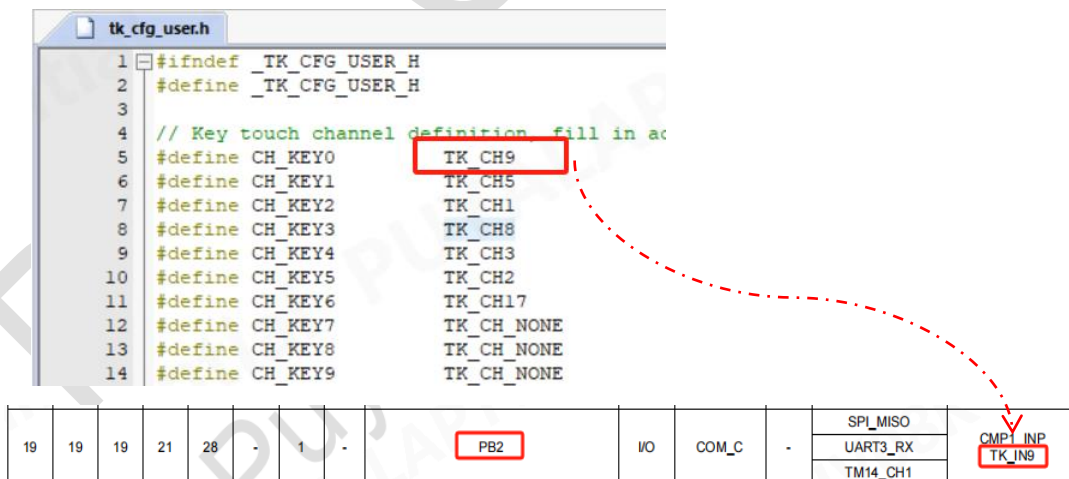
			interference is detected, it can discard (mask) the generated key press flag.
TK_ScanDone	None	Indicates whether a touch mode switch and re-initialization is required.	This function is called after touch data acquisition is complete. It is used to determine if the system needs to switch its touch detection mode and perform a re-initialization.
PrintfTask	None	None	Print interface function. This function is called after touch data acquisition and subsequent data processing are complete. Users can add relevant code inside this function to output touch data and status information via UART if printing is required.

3.6. Touch Application Configuration Steps and Process

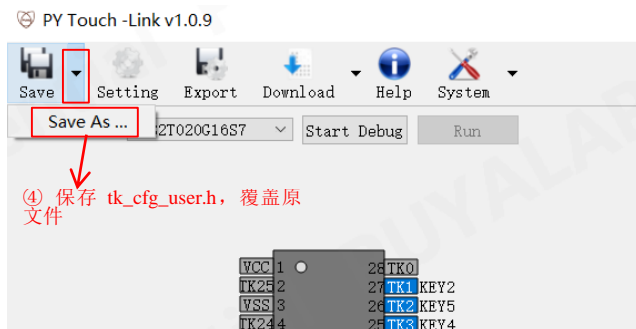
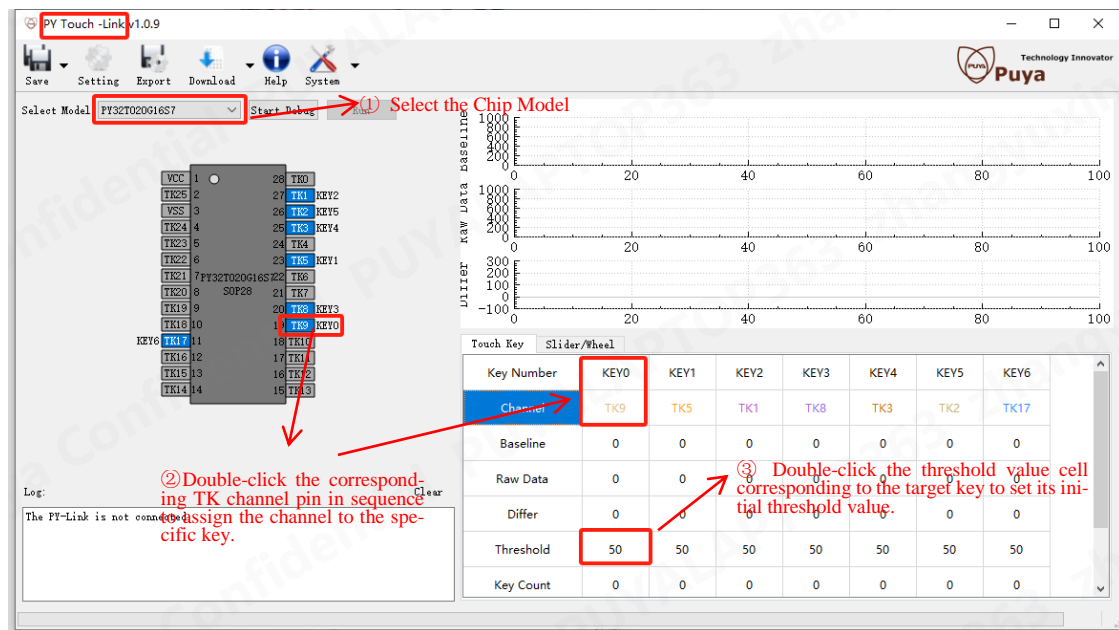
3.6.1. Conventional Keys Application

Step 1: Set the TK channel corresponding to the key. There are two methods:

Method 1: Fill in the TK channels sequentially directly in tk_cfg_user.h, as shown below:

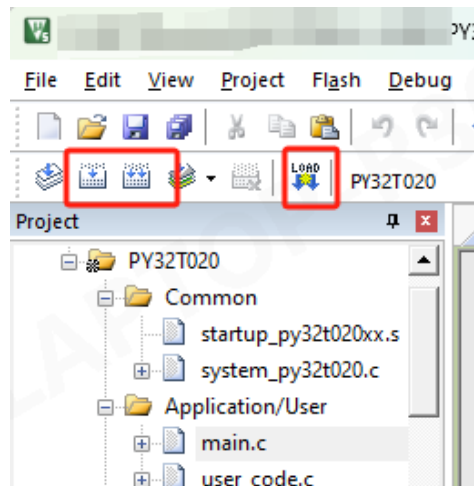


Method 2: Configure the TK channels in the upper computer debugging tool PY Touch-Link.exe, then export `tk_cfg_user.h` to overwrite the original file.

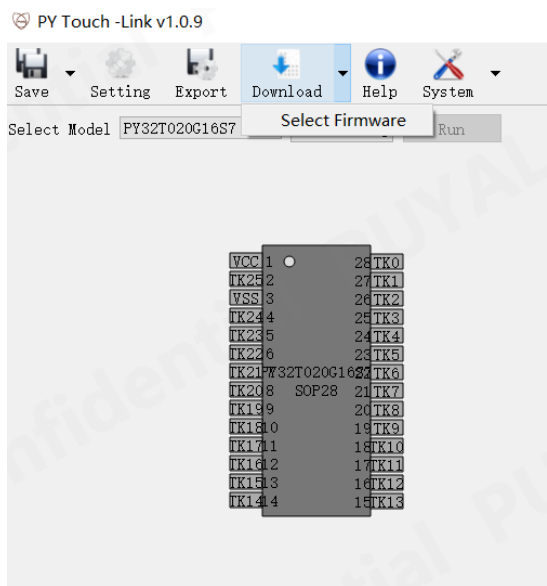


Step2: Compile the program. After compilation, download it to the target board. There are two methods for downloading the program:

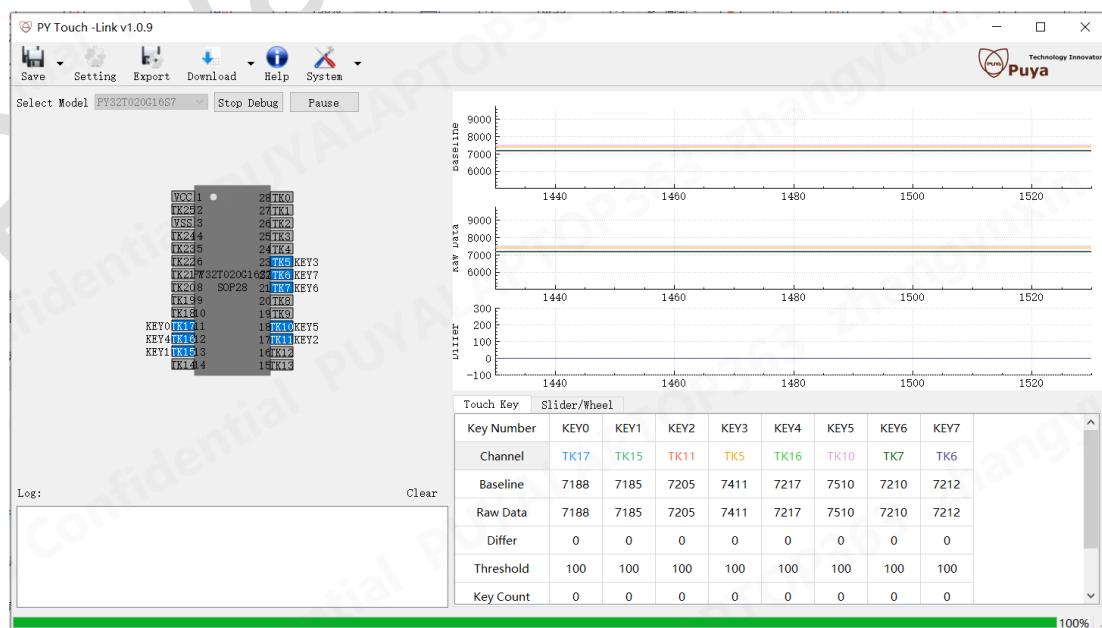
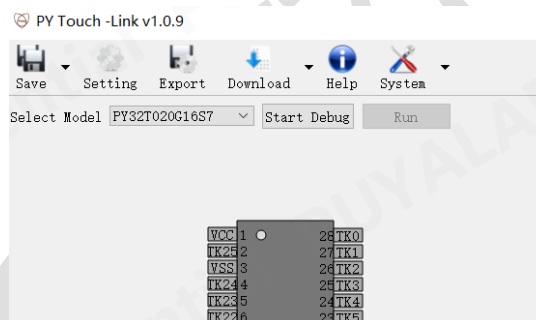
Method 1: Download directly within the KEIL environment, as shown below (Note: This download method requires the download pins to remain as SWD port function):



Method 2: Use PY Touch-Link.exe to download, as shown below:



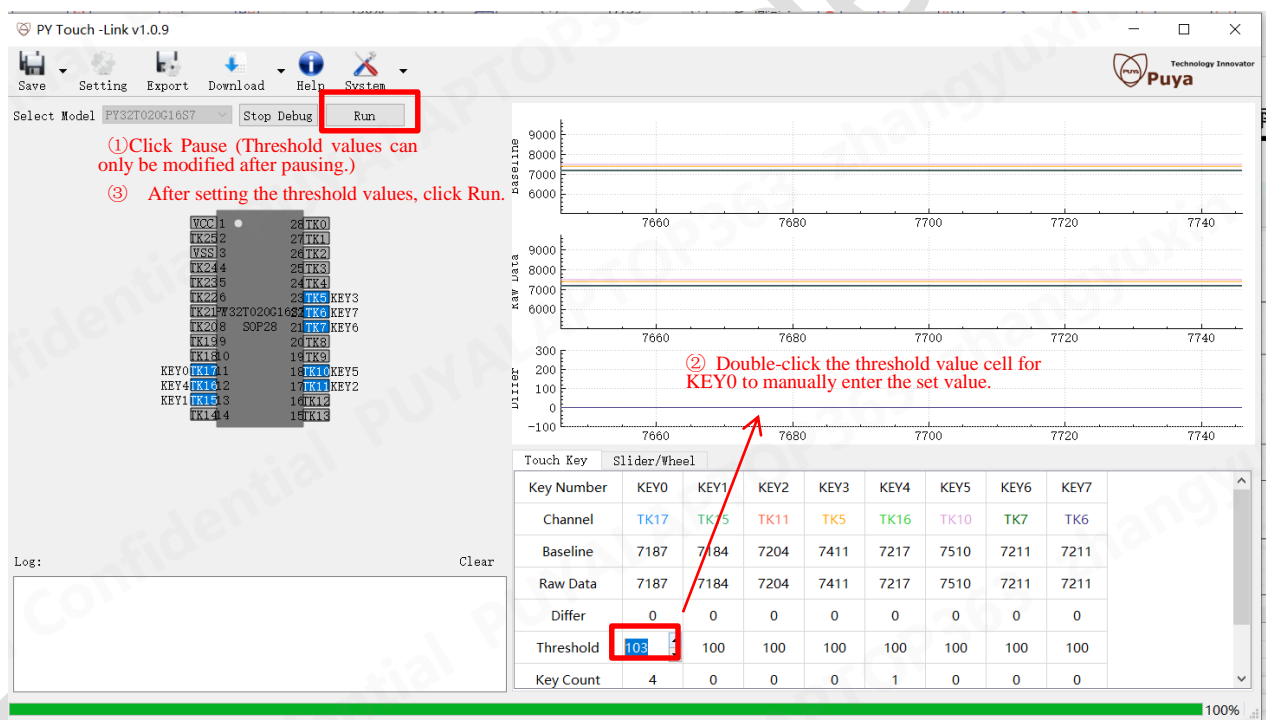
Step3: Connect to the upper computer debugging tool PY Touch-Link and enter touch debugging mode.



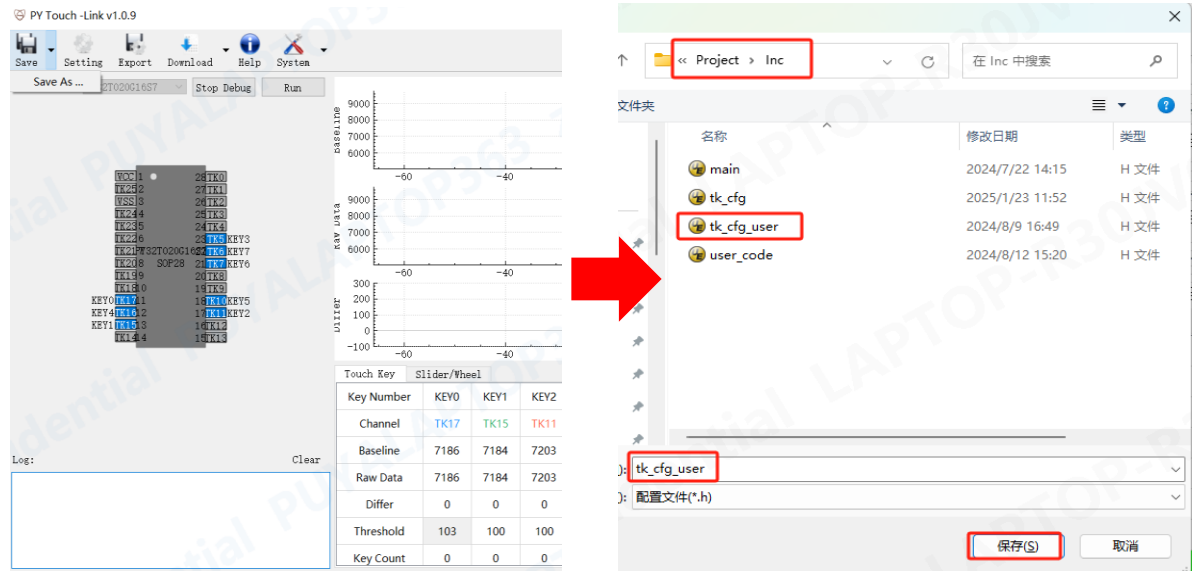
Step4: Normally touch the key with a finger (or copper pillar), observe the change in touch data, and set the trigger threshold to 60%–70% of the change value.

Touch Key	Slider/Wheel							
Key Number	KEY0	KEY1	KEY2	KEY3	KEY4	KEY5	KEY6	KEY7
Channel	TK17	TK15	TK11	TK5	TK16	TK10	TK7	TK6
Baseline	7194	7190	7206	7414	7224	7519	7216	7217
Raw Data	7342	7191	7206	7414	7249	7519	7217	7217
Differ	148	1	0	0	25	0	1	0
Threshold	100	100	100	100	100	100	100	100
Key Count	4	0	0	0	1	0	0	0

Example: As shown in the figure above, when a finger normally touches KEY0, the change value is about 184. The threshold can be set to: $148 * 70\% = 103$. The setting method is as follows:



Step5: After setting all keys according to Step4, export tk_cfg_user.h to overwrite the original file.



Step6 (Optional): Modify configuration parameters in tk_cfg.h. As shown in the figure, the parameters within the red box can be appropriately adjusted according to the actual situation. Other parameters can generally remain at their default settings.

Option

Option	Value
Basic Configuration	
CMOD Capacitor	Internal
Reference Voltage	2.0V
Adaptive Current	ENABLE
Filter Coefficient	25
Frequency Hopping Mode	ENABLE
Polynomial Width	8
Random Seed	0xA7
Key Parameters Configuration	
Divider	3
Scan Window	12000
Press Debounce Count	3
Release Debounce Count	3
Baseline Trace Up Step Time	500
Baseline Trace Down Step Time	200
Limit Maximum Output Time	30000
Release Threshold Ratio	75
Baseline Trace Mode	MODE0
Single Key Trigger	DISABLE
Max Trigger Key Count	0
Synchronous Sampling Mode	<input checked="" type="checkbox"/>
Scan Window	2400
Scan Count	10
Output Compensation	<input checked="" type="checkbox"/>
Basic Configuration	
Set capacitance mode, reference voltage, filter coefficient, adaptive current, frequency hopping mode, etc.	
Low Power Configuration	<input checked="" type="checkbox"/>
Internal Reference Configuration	<input checked="" type="checkbox"/>
Noise Threshold	25
Waterproof Mode Configuration	
SHIELD Channel	TK5
Water-on-Key Handling	Normal Trigger
Anti-Wet Cloth	<input checked="" type="checkbox"/>

Filter Coefficient Value Range: 5 to 50.
A smaller value results in stronger filtering capability but slower key response speed. In practical applications, while ensuring satisfactory key operation feel, the filter coefficient should be set as small as possible.

In scenarios where the touch keys exhibits large parasitic capacitance and consequently low sensitivity, the frequency division coefficient can be appropriately increased. During practical debugging, adjust this parameter incrementally while simultaneously monitoring the key's response variation. The optimal value is typically the coefficient that achieves approximately 90% of the maximum variation value.

A larger window size results in a longer scan time and a larger touch data variation when a key is pressed. If the key response speed is slow, appropriately reducing the window size can improve the response speed.

For specific application scenarios, it may be necessary to enable single-key mode or set a maximum limit for simultaneous multi-key detections. These parameters should be configured based on practical implementation needs.

The internal reference channel can be utilized to detect power supply interference. Enabling this channel and configuring an appropriate threshold value allows filtering of falsely triggered keys caused by abnormal power fluctuations, such as in scenarios involving walkie-talkie interference. Important: A substantial margin must be reserved for this threshold value; otherwise, it may impair normal key operation or cause key masking during EMC certification.

Step7: At this point, the configuration of conventional keys is complete. When a key is pressed, the corresponding bit in TKCtr.KeyFlags will be set. The user program can check the flag bit corresponding to the key and execute the corresponding operation.

3.6.2. Slider/Wheel Application

Step1:Configure the Slider/Wheel channels, type, and resolution. There are two methods to achieve this:

Method 1:In PY Touch-Link, select channels, set the Slider/Wheel type and resolution. After configuration, export tk_cfg_user.h to overwrite the original file (export method refers to the key application section), as shown below:

PY Touch -Link v1.0.9

Save Setting Export Download Help System

Select Model PY32T020G16S7 Start Debug Run

Key
Slider/Wheel

Slider/Wheel0
Slider/Wheel1
Slider/Wheel2
Slider/Wheel3

① Follow the sequential order of the slider/wheel channels, and right-click on the pin terminals to assign them.

Log: Clear

Touch library version 2.0.2.2

Touch Key	Slider/Wheel			
Index	0:Slider			
Channel	CH0	CH6	CH2	CH1
Baseline	0	0	0	0
Raw Data	0	0	0	0
Differ	0	0	0	0
Position	0			
Threshold	255			

Touch Key Slider/Wheel

Index Slider

Channel Slider

Baseline 0 0 0 0

Raw Data 0 0 0 0

Differ 0 0 0 0

Position 0

Threshold 80

② Activate the selection menu by double-clicking with the left mouse button, and choose either "Slider" or "Wheel" from the available options.

Touch Key Slider/Wheel

Index 0:Slider

Channel CH0 CH6 CH2 CH1

Baseline 0 0 0 0

Raw Data 0 0 0 0

Differ 0 0 0 0

Position 0

Threshold 250

③ Double-click the resolution cell with the left mouse button to enter the setup interface, then input the resolution value.

Method 2: Modify directly in tk_cfg_user.h, as shown below:

```
//=====
#define SLIDER_OR_WHEEL0_TYPE      TK_APP_WHEEL    // Slider type
#define SLIDER_OR_WHEEL0_RESOLUTION 360           // Slider resolution
#define SLIDER_OR_WHEEL0_THD      101             // Slider threshold
#define SLIDER_OR_WHEEL0_CH0      TK_CH0          // Slider channels, fill in sequentially
#define SLIDER_OR_WHEEL0_CH1      TK_CH6
#define SLIDER_OR_WHEEL0_CH2      TK_CH2
#define SLIDER_OR_WHEEL0_CH3      TK_CH1
#define SLIDER_OR_WHEEL0_CH4      TK_CH_NONE
#define SLIDER_OR_WHEEL0_CH5      TK_CH_NONE
#define SLIDER_OR_WHEEL0_CH6      TK_CH_NONE
#define SLIDER_OR_WHEEL0_CH7      TK_CH_NONE
```

Step2: After completing the basic Slider/Wheel configuration, compile the program, download it to the target board, connect to PY Touch-Link for debugging, and set the appropriate trigger threshold.

Touch Key		Slider/Wheel						Touch Key		Slider/Wheel								
Index	0:Wheel				1:Slider				Channel	CH0	CH6	CH2	CH1	CH19	CH23	CH24	CH25	
Channel	CH0	CH6	CH2	CH1	CH19	CH23	CH24	CH25	Baseline	3607	3599	3591	3526	3601	3593	3580	3582	
Baseline	3609	3598	3593	3528	3601	3595	3583	3581	Raw Data	3607	3599	3591	3526	3601	3593	3580	3582	
Raw Data	3709	3604	3600	3533	3605	3595	3584	3581	Differ	0	0	0	0	0	0	0	0	
Differ	100	6	7	5	4	0	1	0	Position	-1				-1				
Position	359				-1				Threshold	80					100			
Threshold	80				100				Resolution	360				255				

① Slide your finger normally on the slider/wheel and observe the variation of each channel.

② Access the threshold configuration interface by double-clicking, then iteratively refine the threshold value through testing to achieve the best user interaction quality.

```
//=====
#define SLIDER_OR_WHEEL0_TYPE      TK_APP_WHEEL    // Slider type
#define SLIDER_OR_WHEEL0_RESOLUTION 360           // Slider resolution
#define SLIDER_OR_WHEEL0_THD      100             // Slider threshold
#define SLIDER_OR_WHEEL0_CH0      TK_CH0          // Slider channels, fill in sequentially
#define SLIDER_OR_WHEEL0_CH1      TK_CH6
#define SLIDER_OR_WHEEL0_CH2      TK_CH2
#define SLIDER_OR_WHEEL0_CH3      TK_CH1
#define SLIDER_OR_WHEEL0_CH4      TK_CH_NONE
#define SLIDER_OR_WHEEL0_CH5      TK_CH_NONE
#define SLIDER_OR_WHEEL0_CH6      TK_CH_NONE
#define SLIDER_OR_WHEEL0_CH7      TK_CH_NONE
```

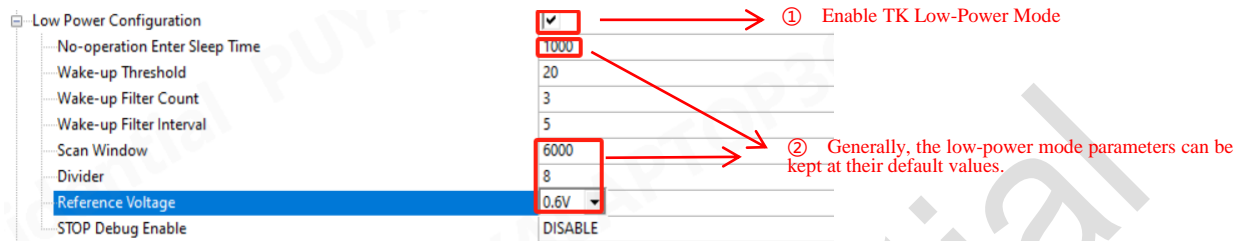
③ The determined final threshold value should be entered into the tk_cfg_user.h configuration file.

Step3: At this point, the Slider/Wheel configuration is complete. The generated position information is stored in TKCtr.SliderOrWheelPosition[n]. The user program can read this value and execute the corresponding operation.

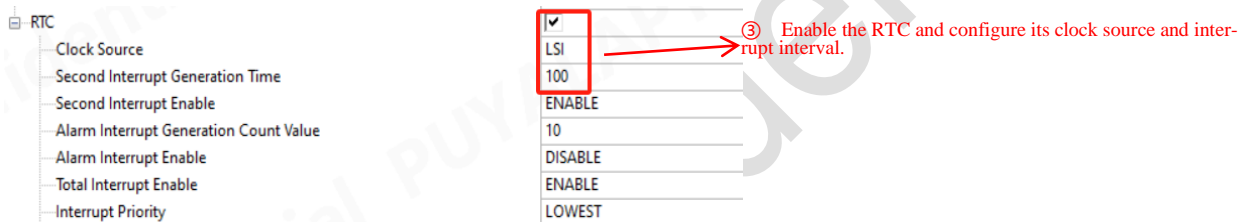
```
if (TKCtr.SliderOrWheelPosition[0] != -1)
{
    // User Application Program
}
if (TKCtr.SliderOrWheelPosition[1] != -1)
{
    // User Application Program
}
```


3.6.3. Touch Low-Power Application

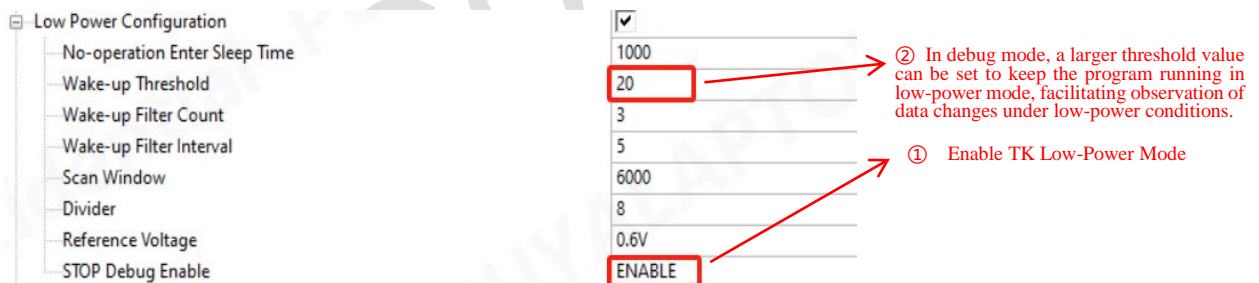
Step1: Enable touch low-power mode in tk_cfg.h and configure the basic parameters.



Since touch low-power uses RTC as the time base, RTC must also be configured and enabled first. RTC is configured in app_config.h.



Step2: Set the low-power mode trigger threshold.



Step3: Download the TK low-power mode program to the target board. Normally touch the key or slider with a finger and observe the data change to determine the threshold.

Touch Key			
Slider/Wheel			
Key Number	KEY0	KEY1	
Channel	TK14	TK15	TK_CO...
Baseline	3601	3607	5465
Raw Data	3602	3608	5513
Differ	1	1	48
Threshold	180	180	15
Key Count	2	1	0

In low-power debug mode, the threshold value should be set based on the data variation observed during finger touches. Typically, the threshold is set at approximately 50% of the variation value.

Note: The variation value may differ across keys. In practice, always reference the key with the smallest variation to ensure reliable triggering under low-power conditions.

Step4:Set the threshold based on the debug data, turn off debug mode, re-download the program for testing, and fine-tune the threshold based on the actual measured effect until the best experience is achieved.

Low Power Configuration	<input checked="" type="checkbox"/>
No-operation Enter Sleep Time	1000
Wake-up Threshold	20
Wake-up Filter Count	3
Wake-up Filter Interval	5
Scan Window	6000
Divider	8
Reference Voltage	0.6V
STOP Debug Enable	DISABLE

3.6.4. Floating Keys Application

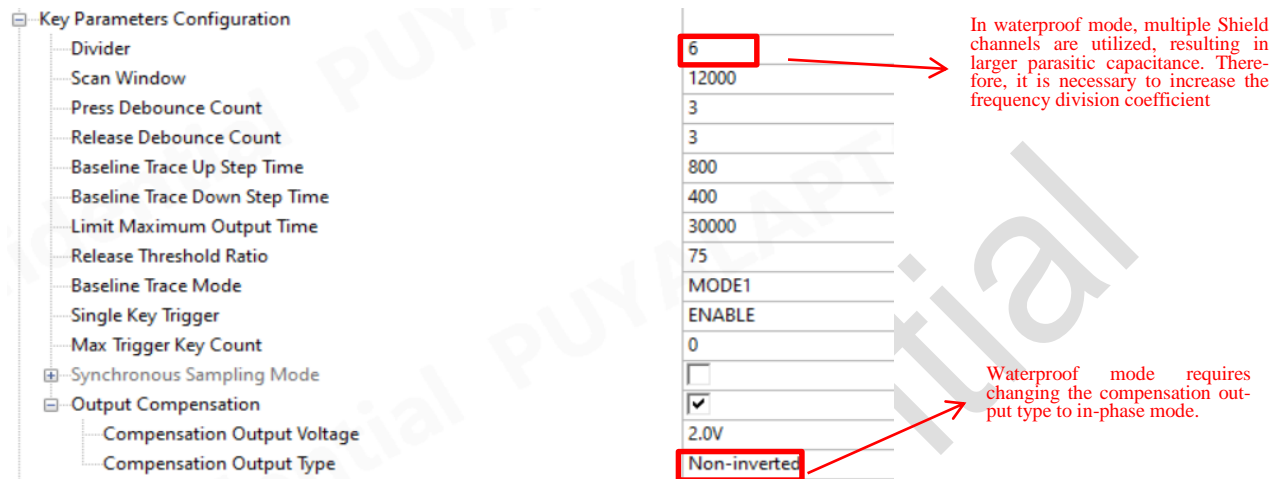
The setup method for floating keys applications is similar to conventional keys. The difference is that floating keys need to use synchronous scan mode, as shown below:

Key Parameters Configuration		
Divider	3	
Scan Window	12000	
Press Debounce Count	3	
Release Debounce Count	3	
Baseline Trace Up Step Time	800	The baseline update speed can be appropriately slowed down.
Baseline Trace Down Step Time	400	
Limit Maximum Output Time	30000	
Release Threshold Ratio	75	
Baseline Trace Mode	MODE1	If the key spacing is relatively close, crosstalk may occur between adjacent keys.
Single Key Trigger	ENABLE	1、 In this case, set the baseline update mode to MODE1, where all keys stop updating their baselines when any key is triggered. 2、 Alternatively, enable single-key mode to prevent key crosstalk.
Max Trigger Key Count	0	
Synchronous Sampling Mode	<input checked="" type="checkbox"/>	
Scan Window	2400	
Scan Count	15	Enable synchronous scan mode and configure appropriate window values and scan counts. During debugging, adjust the scan count based on the actual key data variation, ensuring the variation reaches at least 50.
Output Compensation	<input checked="" type="checkbox"/>	
Compensation Output Voltage	2.0V	
Compensation Output Type	Low Level	

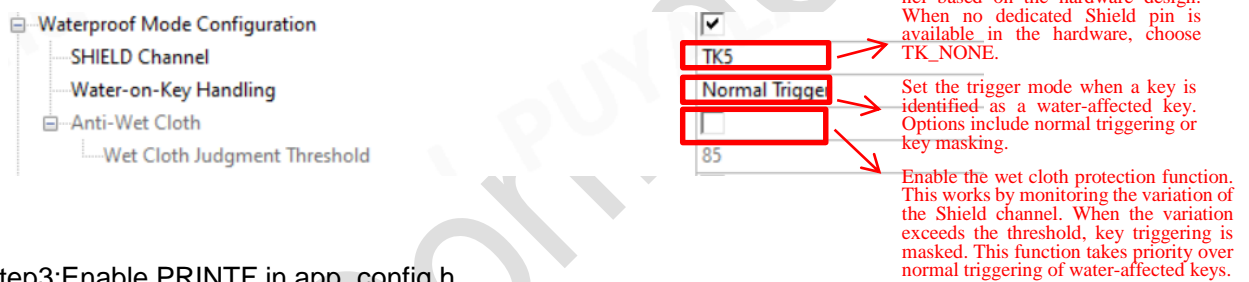
3.6.5. Waterproof Keys Application

The setup method for waterproof keys applications is similar to conventional keys. The difference is that waterproof keys need to set the output compensation to in-phase mode, as shown below:

Step1: Increase the frequency division coefficient and modify the compensation output type.



Step 2: Enable waterproofing and set the water detection button response mode.



Step3: Enable PRINTF in app_config.h.



Step4: After assembling the housing, download the program to the target board and connect to PY Touch-Link. You can see an additional TK_COM channel in the touch keys column; this channel represents the Shield channel data. When the wet cloth wiping prevention function is needed, observe the delta value of this channel during the wet cloth wiping action, enable the wet cloth wiping prevention, and fill the value into the wet cloth judgment threshold.

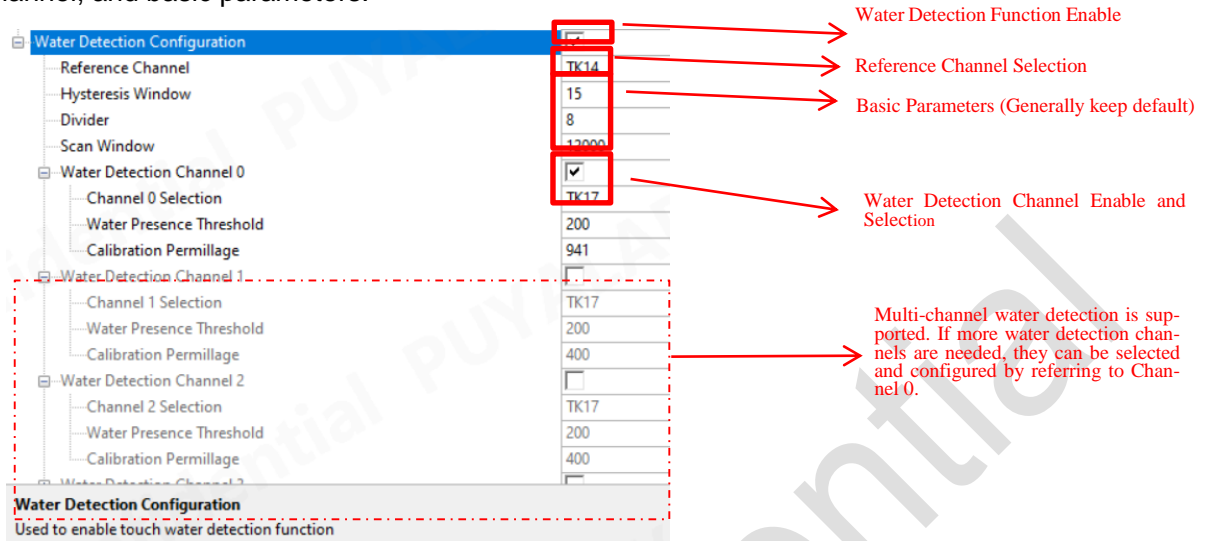
Touch Key	Slider/Wheel									
Key Number	KEY0	KEY1	KEY2	KEY3	KEY4	KEY5	KEY6	KEY7	KEY8	KEY9
Channel	TK0	TK1	TK12	TK6	TK7	TK11	TK8	TK9	TK10	TK_COM-SHIELD
Baseline	3610	3620	3600	3608	3607	3591	3588	3583	3598	3529
Raw Data	3610	3620	3600	3608	3607	3591	3588	3583	3598	3529
Differ	0	0	0	0	0	0	0	0	0	0
Threshold	50	50	50	50	50	50	50	50	50	85
Key Count	0	0	0	0	0	0	0	0	0	0

Step6: Modify the program according to the ratio obtained from actual measurement.

In a dry state, he printed ratio when the key is pressed is approximately 68. Normally, subtract 20 from this value to set the lower threshold, which serves as the demarcation point for water-affected keys. Then, add 20 to this value to set the upper threshold, which serves as the detection point for water splashing. After setting these values, perform multiple tests to verify the accuracy of the settings.

3.6.6. Water Detection Application Using Touch

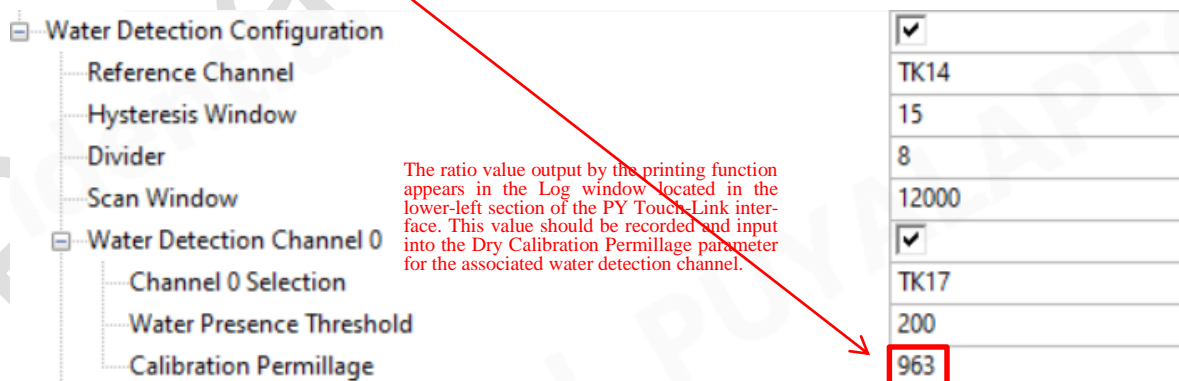
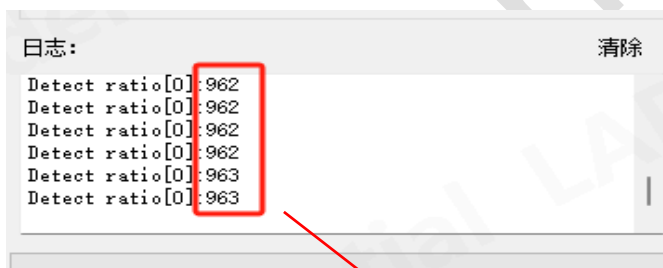
Step1: Enable the water detection function, configure the reference channel, water detection channel, and basic parameters.



Step2: Enable PRINTF in app_config.h.



Step3: Assemble the housing and container, keeping the container in a waterless state. Download the program to the target board and connect to PY Touch-Link.

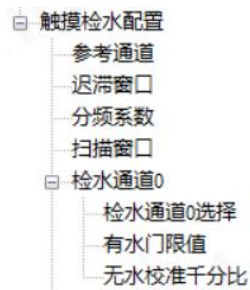


Step4: After filling in the reference value for the waterless state, recompile and download the program.

Touch Key	Slider/Wheel	
Channel	TK14	TK17
Baseline	5658	5658
Raw Data	5658	6012
Differ	0	354
Threshold	0	200
Key Count	0	7
Status		

Pour water into the container until the water level exceeds the position of the water detection PAD. At this point, the variation detected by the water detection channel is the water-present variation. Generally, approximately 50% of this water-present variation is taken as the water detection threshold value.

Step5: Fill in about 50% of the water-present change value into the "Water Present Threshold".



<input checked="" type="checkbox"/>	TK14
<input type="checkbox"/>	15
<input type="checkbox"/>	24
<input type="checkbox"/>	12000
<input checked="" type="checkbox"/>	TK17
<input type="checkbox"/>	177
<input type="checkbox"/>	963

Fill in the water threshold value: $354 \times 0.5 = 177$

Step6: Re-download the program, test repeatedly and fine-tune the threshold until the best water detection effect is achieved. At this point, the water detection function debugging is complete. The user program can execute corresponding operations based on the water detection status.

```

for (uint8_t n = 0; n < (WATER_DETECT_CNT - 1); n++)
{
    /* Water detected */
    if(TKCtrl.DetectOut & (1 << n))
    {
        // User Application Program
    }
    /* No water */
    else
    {
        // User Application Program
    }
}

```

4. System Configuration and Peripheral Application Analysis

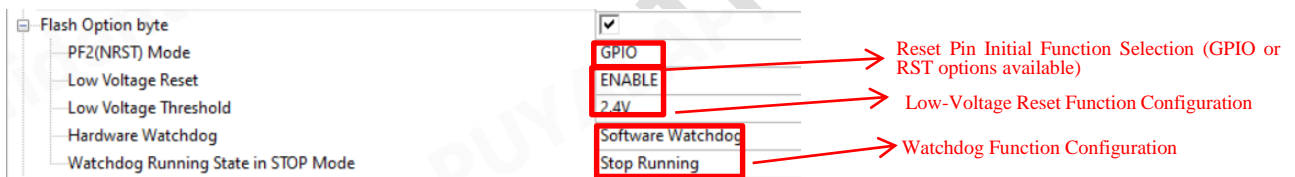
4.1. System Configuration

4.1.1. Clock Configuration

Configure in app_config.h, as shown in the figure:



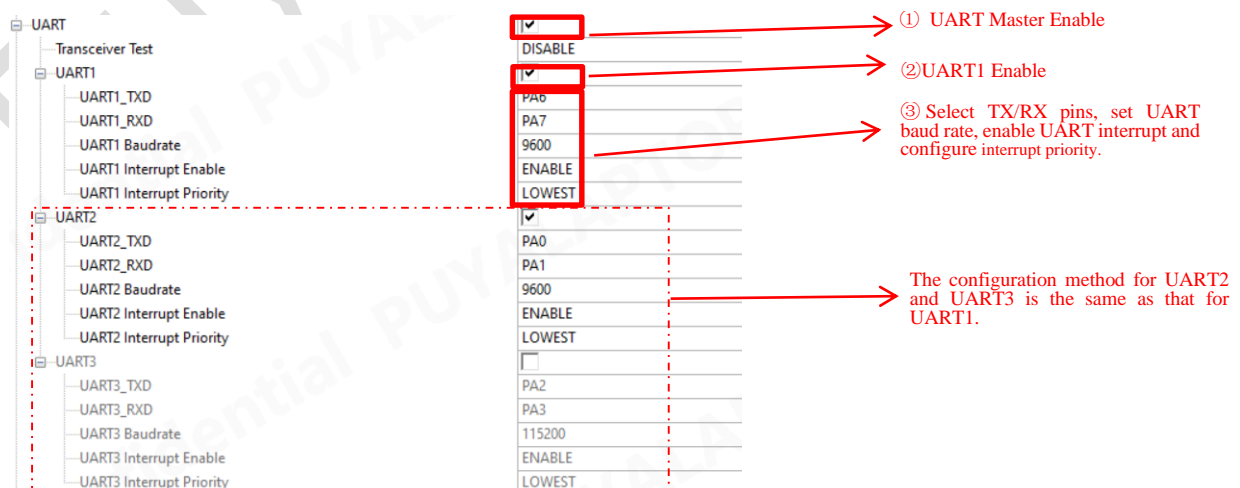
4.1.2. Option Bytes



4.2. Peripheral Application Interfaces and Usage

4.2.1. UART

4.2.1.1. UART Setup Interface



4.2.1.2. UART Function Interface Description

Function Name	Input Parameters			Return Value	Description
	Parameter Name	Data Type	Parameter Description		
UART_QueueSend	*UARTx	UART_TypeDef	Pointer to the UARTx register bank	Transmission Result Flag	Call this function to implement transmission of a single byte of data via UARTx.
	*Queue	SqQueue	Pointer to the transmit data queue		
	data	uint8_t	Single byte of data to be transmitted		
UART_QueueRead	*Queue	SqQueue	UART receive data queue pointer	Receive Queue Empty Flag	If the receive queue is not empty, calling this function reads one byte of data from the receive queue.
	*data	uint8_t	Data pointer; the byte read from the receive queue is assigned to the variable pointed to by this pointer		

Note: The above UART function interfaces are stored in uart_drivers.c.

4.2.1.3. UART Application Example

The image shows a screenshot of a development environment. At the top, there is a configuration section with a checkbox labeled "ENABLE" which is checked. A red arrow points from this checkbox to a text box that says: "UART Test Enable: After enabling, the chip can connect to a PC and communicate with a serial port assistant."

Below the configuration, there is a code editor showing the implementation of the UART_QueueSend function. The code is as follows:

```

void UART_Loop(void)
{
    #if ((UART1_RX_PIN != NO_PIN) && UART1_ENABLE)
        static uint8_t rx1_data[64];
        static uint8_t rx1_count = 0;
    #endif
    #if ((UART2_RX_PIN != NO_PIN) && UART2_ENABLE)
        static uint8_t rx2_data[64];
        static uint8_t rx2_count = 0;
    #endif
    #if ((UART3_RX_PIN != NO_PIN) && UART3_ENABLE)
        static uint8_t rx3_data[64];
        static uint8_t rx3_count = 0;
    #endif
    #if ((UART1_RX_PIN != NO_PIN) && UART1_ENABLE)
        while (UART1_QueueRead(&rx1_data[rx1_count]))
        {
            rx1_count++;
            rx1_timeout = 5;
            if (rx1_count == 64)
                break;
        }
        if ((rx1_count && rx1_timeout == 0) || rx1_count == 64)
        {
            #if (UART1_TX_PIN != NO_PIN)
                /* 串口接收完成 */
                for (uint8_t n = 0; n < rx1_count; n++)
                {
                    UART1_QueueSend(rx1_data[n]);
                }
            #endif
            rx1_count = 0;
        }
    #endif
}

```

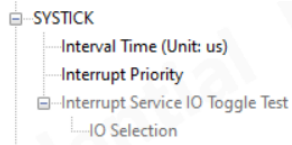
Two red boxes highlight specific parts of the code with red arrows pointing to explanatory text:

- The first box highlights the `while (UART1_QueueRead(&rx1_data[rx1_count]))` loop. A red arrow points to the text: "Data can be sent using a serial port assistant. The chip receives 64 bytes of data and stores it in the rx1_data array."
- The second box highlights the `if ((rx1_count && rx1_timeout == 0) || rx1_count == 64)` block. A red arrow points to the text: "The received 64-byte data is then sent back. If the data received by the serial port assistant matches the sent data, it confirms that UART transmission and reception are functioning normally."

4.2.2. Timer

4.2.2.1. SysTick

The SYSTICK setup interface is as follows:



- ① SYSTICK Enable
- ② Set Timing Interval and Interrupt Priority
Note: The timing interval must not exceed 5ms
- ③ To test the timing functionality, check this option and select a test pin. When checked, the selected IO will toggle within the SYSTICK interrupt service routine. An external oscilloscope or logic analyzer can monitor this IO waveform to verify SYSTICK interrupt timing accuracy.

The SYSTICK interrupt service routine is located in main.c, as shown below:

```

/*****
** Function: void SysTick_Handler(void)
** Description: System tick timer, currently set to 1ms timing
** Input: None
** Return: None
*****/
void SysTick_Handler(void)
{
    static uint16_t Prescaler;
    #if (SYSTICK_DEBUG_GPIO != NO_PIN && SYSTICK_DEBUG)
        GPIO_ToggleBit(SYSTICK_DEBUG_GPIO);
    #endif
    Prescaler++;
    if (Prescaler >= (1000 / SYSTICK_TIMING_TIME))
    {
        Prescaler = 0;
        app_drivers_timer();
    }
    #if APP_TK_ENABLE
        TK_TimerHandler(1);
    #endif
    #if TK_TICK_USAGE && DEBUG_ENABLE
        log_time++;
    #endif
    #if APP_BEEP_ENABLE
        BEEP_Toggle();
    #endif
    #if (APP_IR_RECEIVED_ENABLE == 1 && D_IR_TIM == 0)
        IR_Received_Scan();
    #endif
    #if APP_LED_ENABLE
        LED_Scan();
    #endif
    user_timer();
}

```

If the user program requires code execution within the SYSTICK timer interrupt service routine, such programs can be placed in user_timer

4.2.2.2. TIM1, TIM14

The TIMx (x=1, 14) timer setup interface is as follows:

The screenshot shows a configuration interface for two timers, TIM14 and TIM1. Each timer has a tree view on the left and a corresponding configuration panel on the right.

Timer	Timing Time (Unit: us)	Interrupt Enable	Interrupt Priority
TIM14	125	ENABLE	HIGHEST
TIM1	125	ENABLE	HIGHEST

Below the main configuration, there are checkboxes for 'Interrupt Service IO Toggle Test' and 'TIM14-PWM' (which is disabled).

Note: TIMx timer setup is basically the same as SYSTICK, refer to the relevant description of SYSTICK.

The TIMx timer interrupt service routines are located in tim1_drivers.c and tim14_drivers.c respectively, as shown below:

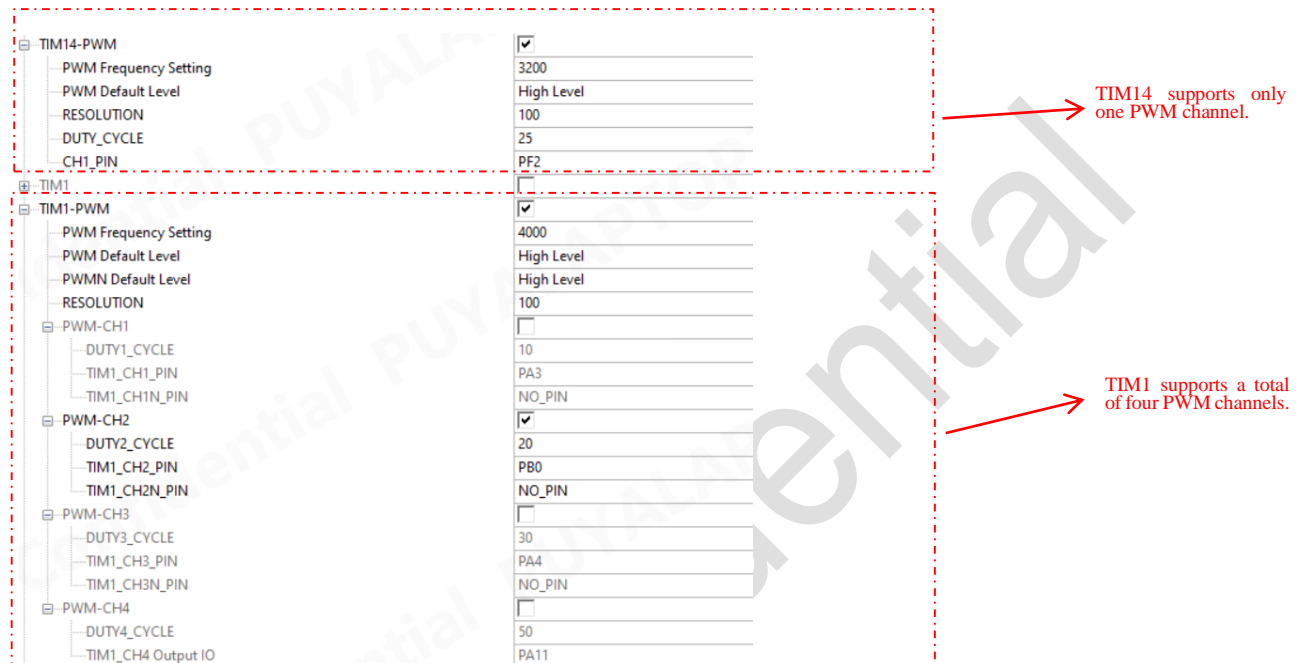
```
void TIM1_BRK_UP_TRG_COM_IRQHandler(void)
{
    /* TIM Update event */
    if (LL_TIM_IsActiveFlag_UPDATE(TIM1) != 0)
    {
        LL_TIM_ClearFlag_UPDATE(TIM1);
        /* Test GPIO toggle */
#ifdef TIM1_DEBUG_GPIO != NO_PIN && TIM1_DEBUG
        GPIO_ToggleBit(TIM1_DEBUG_GPIO);
#endif
        TIM1_PeriodElapsedCallback();
    }
}
```

```
void TIM14_IRQHandler(void)
{
    /* TIM Update event */
    if (LL_TIM_IsActiveFlag_UPDATE(TIM14) != 0)
    {
        LL_TIM_ClearFlag_UPDATE(TIM14);
        /* GPIO Toggle Test */
#ifdef TIM14_DEBUG_GPIO != NO_PIN && TIM14_DEBUG
        GPIO_ToggleBit(TIM14_DEBUG_GPIO);
#endif
        TIM14_PeriodElapsedCallback();
    }
}
```

4.2.3. PWM

The PWM function is integrated into TIMx and is an application mode of TIMx. Note: For TIMx, the PWM function and timer function cannot be used simultaneously.

The PWM setup interface is shown below:



When adjustment of the PWM duty cycle is needed in the application, the following interface functions can be called to set it.

```
void TIM14_PWM_Pulse(uint32_t CHx, uint16_t percent)
{
    /* CH1 compare value:250 */
    TIM_OC_Initstruct.CompareValue = TIM14_Autoreload * percent / TIM14_RESOLUTION;
    switch(CHx)
    {
        case LL_TIM_CHANNEL_CH1:
            TIM14->CCR1 = TIM_OC_Initstruct.CompareValue;
            break;
    }
}

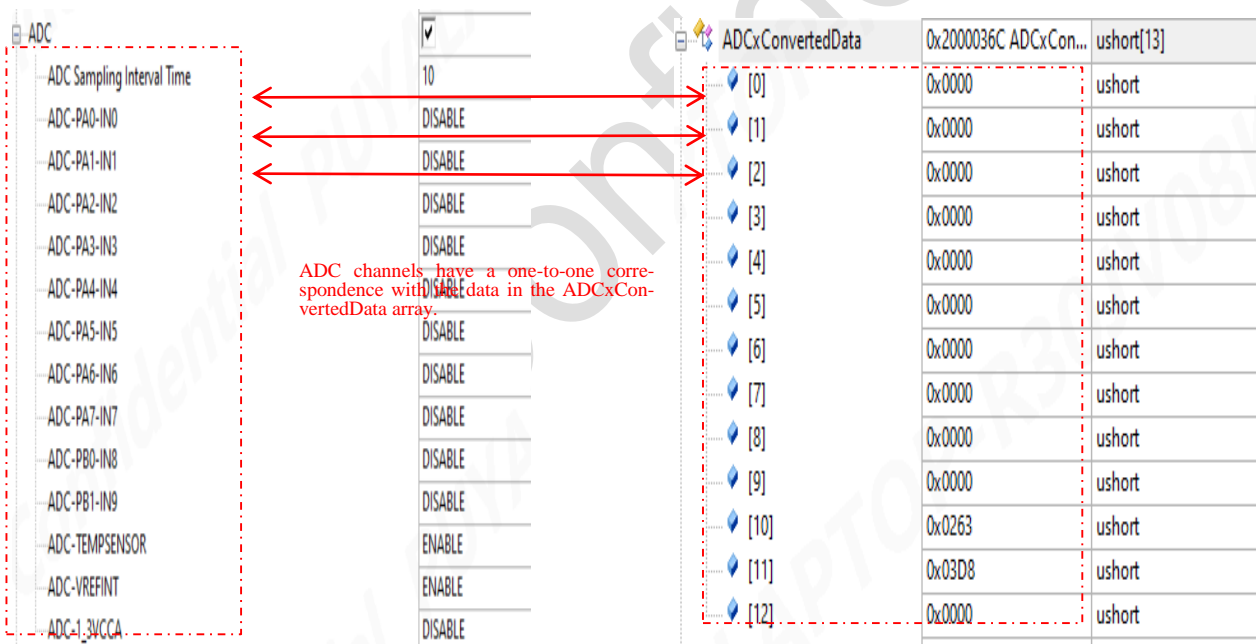
void TIM1_PWM_Pulse(uint32_t CHx, uint16_t percent)
{
    /* CH1 compare value:250 */
    TIM_OC_Initstruct.CompareValue = TIM1_Autoreload * percent / TIM1_RESOLUTION;
    switch (CHx)
    {
        case LL_TIM_CHANNEL_CH1:
            TIM1->CCR1 = TIM_OC_Initstruct.CompareValue;
            break;
        case LL_TIM_CHANNEL_CH2:
            TIM1->CCR2 = TIM_OC_Initstruct.CompareValue;
            break;
        case LL_TIM_CHANNEL_CH3:
            TIM1->CCR3 = TIM_OC_Initstruct.CompareValue;
            break;
        case LL_TIM_CHANNEL_CH4:
            TIM1->CCR4 = TIM_OC_Initstruct.CompareValue;
            break;
    }
}
```

4.2.4. ADC

The ADC setup interface is as follows:



After the ADC initialization is set, the program will perform ADC conversions on the enabled ADC channels sequentially at the set time intervals. The collected ADC data is stored in the array `ADCxConvertedData`. The user program can directly obtain ADC data from the `ADCxConvertedData` array and perform corresponding operations.



4.2.5. I2C (Slave Mode)

The I2C slave mode setup interface is shown below:

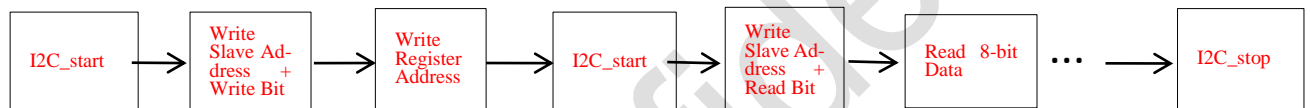


In slave mode, the master can read and write data to the slave. The slave's data is stored in the I2C_SRAMarray. I2C master communication should follow the timing below:

Master Write Operation:



Master Read Operation:



Note: The "Register Address" is the index of the I2C_SRAMarray, e.g., the address of I2C_SRAM[0] is 0.

4.2.6. RTC

The RTC setup interface is as follows:

The screenshot shows the RTC configuration window with the following settings and annotations:

- RTC Enable:** ☒ (Annotated: RTC Enable)
- Clock Source:** LSI (Annotated: Clock source can be selected as LSI or LSE)
- Second Interrupt Generation Time:** 100 (Annotated: Second interrupt generation time, unit is milliseconds (ms))
- Second Interrupt Enable:** DISABLE (Annotated: RTC alarm interrupt generation time is: second interrupt time multiplied by count value)
- Alarm Interrupt Generation Count Value:** 10
- Alarm Interrupt Enable:** DISABLE (Annotated: RTC global interrupt enable: both second interrupt and alarm interrupt must first enable the RTC global interrupt)
- Total Interrupt Enable:** DISABLE
- Interrupt Priority:** LOWEST
- Interrupt Service IO Toggle Test:** ☐ (Annotated: RTC second interrupt test: can select an IO to toggle within the interrupt)

The RTC interrupt callback functions are as follows (driver.c):

```

/*****
** Function: void RTC_SecCallback(void)
** Description: RTC second interrupt callback function
** Input: None
** Return: None
*****/
void RTC_SecCallback(void)
{
}

/*****
** Function: void RTC_AlrCallback(void)
** Description: RTC alarm interrupt callback function
** Input: None
** Return: None
*****/
void RTC_AlrCallback(void)
{
}

```

4.2.7. Watchdog

The watchdog setup interface is as follows:



The program defaults to feeds the watchdog within the main loop.

```

/**
 * Function: void app_drivers_loop(void)
 * Description: Polling in main function while(1), used to handle tasks of each module
 * Input: None
 * Return: None
 */
void app_drivers_loop(void)
{
    #if APP_IR_RECEIVED_ENABLE
        Ir_TypeDef remote;
        if (IR_Press(&remote))
        {
            log_printf("address:0X%X ", remote.ir_address); // Received address
            log_printf("command:0X%X ", remote.ir_command); // Received command
            log_printf("count:%d\r\n", remote.ir_count); // Received count
        }
    #endif

    /* External interrupt flag */
    if (EXTI_Flag != 0)
    {
        log_printf("EXTI_Flag:0X%X\r\n", EXTI_Flag);
        EXTI_Flag = 0;
    }

    #if APP_IWDG_ENABLE
        /* Feed watchdog */
        LL_IWDG_ReloadCounter(IWDG);
    #endif

    #if APP_ADC_ENABLE
        ADC_Loop();
    #endif

    #if APP_UART_ENABLE
        UART_Loop();
    #endif

    #if (APP_TM1624_ENABLE && TM1624_DISP_TEST)
        if (TM1624_Time > 500)
        {
            // ...
        }
    #endif
}

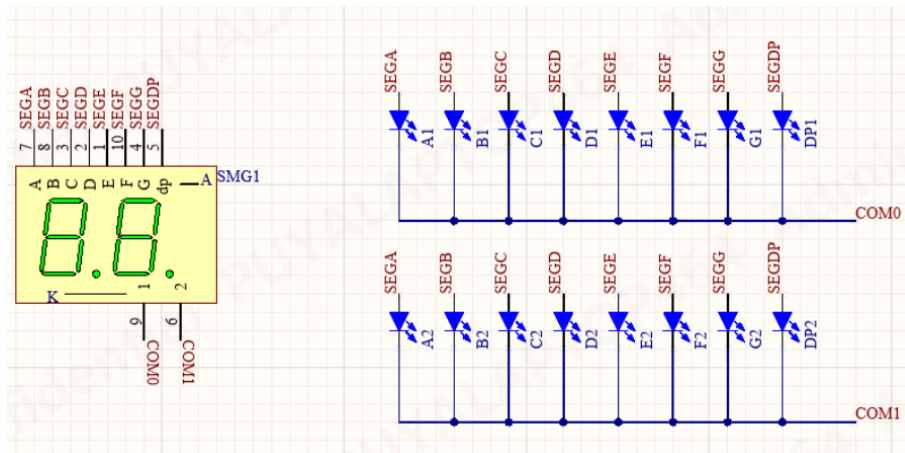
```

4.3. Typical Application Drivers

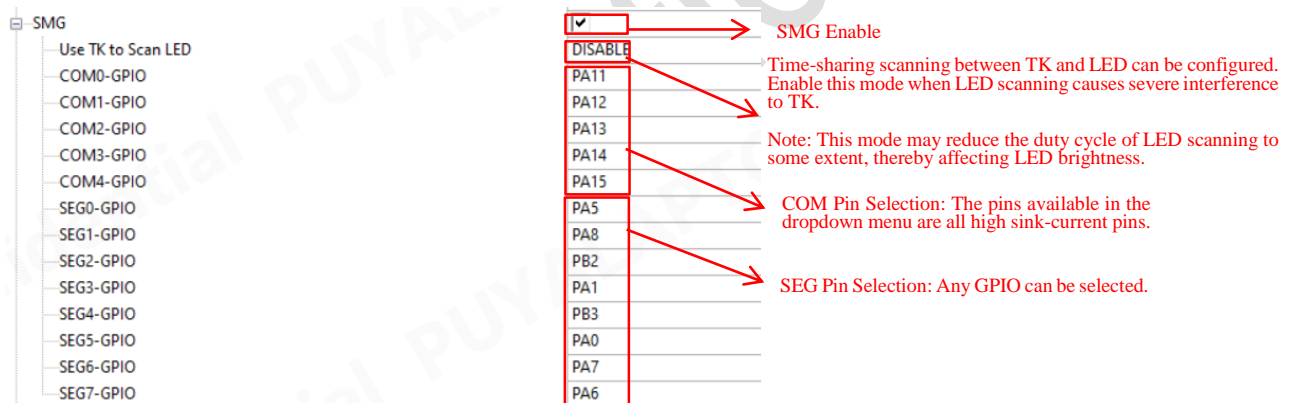
4.3.1. Digital Tube/LED Driver

4.3.1.1. Common Cathode Digital Tube/LED Driver

The typical circuit for a common cathode digital tube/LED is as follows:



The setup interface is as shown below:



Application example (smg_task.c) is as follows:


```

/*****
** Function name: void SMG_Show(uint8_t num)
** Description: Display decimal number using digital tube
** Input: num - number to display
** Return: None
*****/
void SMG_Show(uint16_t num)
{
    smg_data[0] = num_tbl[num / 10];
    smg_data[1] = num_tbl[num % 10];
}

/*****
** Function name: void LED_Show(uint8_t led_bit)
** Description: Button LED indicator
** Input: led_bit - bit position to display,
           bit0 indicates LED for TK0,
           bit1 indicates LED for TK1,
           .....
** Return: None
*****/
void LED_Show(uint16_t led_bit)
{
    uint8_t i;
    uint8_t bit = 0;
    for (i = 0; i < SEG_COUNT; i++)
    {
        if (led_bit & 0X01)
        {
            bit |= TK_LED[i];
            led_bit >>= 1;
        }
        smg_data[2] = bit;
    }
}

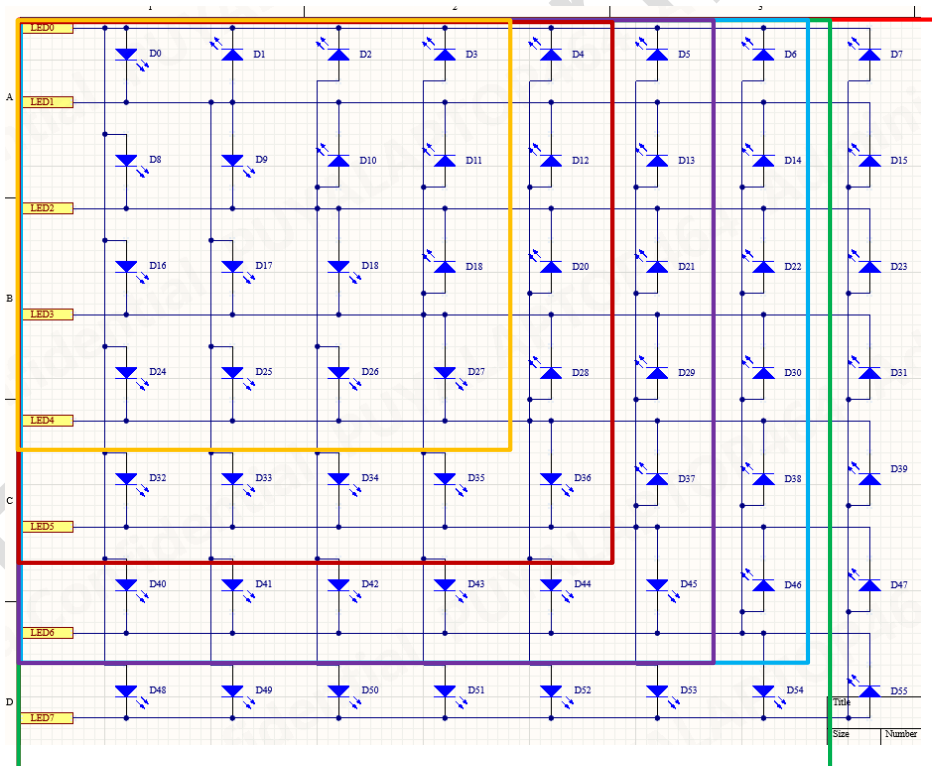
```

The data in the array smg_data corresponds to the display content of the digital tube/LED. In application, you only need to write data to smg_data.

4.3.1.2. Matrix LED Driver

The typical circuit for a matrix LED is as follows; the address corresponding to the LED is consistent for different matrices:

7X8 matrix, 7X7 matrix, 6X7 matrix, 6X6 matrix, 5X5 matrix, 4X4 matrix

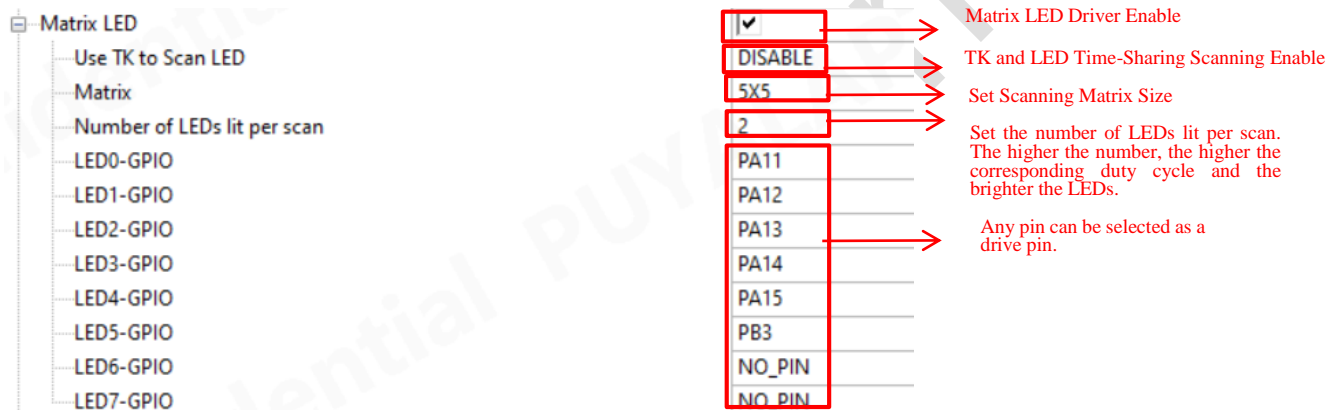


LED Display Configuration Buffer:

Control whether the light is on or off by controlling the led_data cache. 1: ON, 0: OFF.

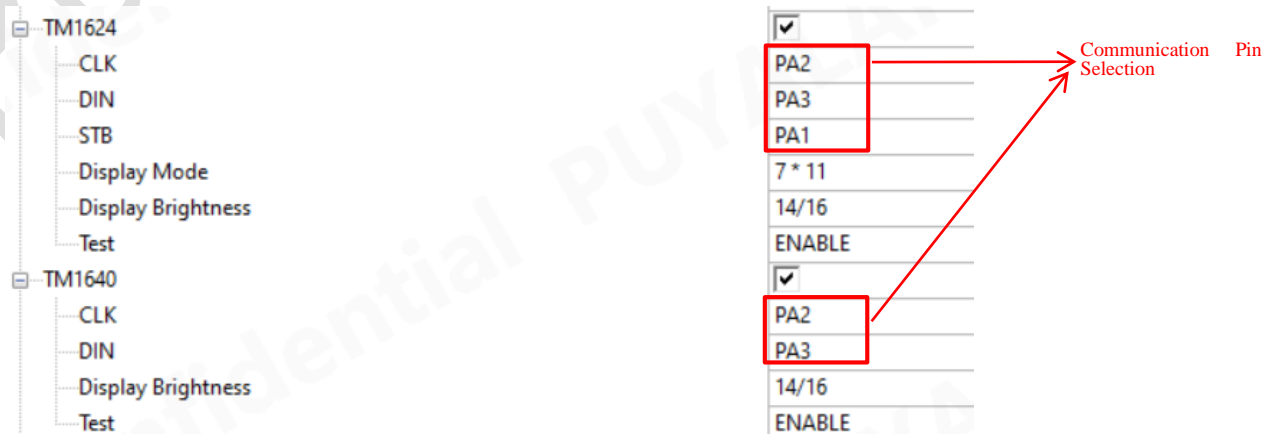
缓存	BIT0	BIT1	BIT2	BIT3	BIT4	BIT5	BIT6	BIT7
led_data[0]	D0	D1	D2	D3	D4	D5	D6	D7
led_data[1]	D8	D9	D10	D11	D12	D13	D14	D15
led_data[2]	D16	D17	D18	D19	D20	D21	D22	D23
led_data[3]	D24	D25	D26	D27	D28	D29	D30	D31
led_data[4]	D32	D33	D34	D35	D36	D37	D38	D39
led_data[5]	D40	D41	D42	D43	D44	D45	D46	D47
led_data[6]	D48	D49	D50	D51	D52	D53	D54	D55

The setup interface is as follows:



4.3.2. Dedicated Driver IC for Digital Tubes/LEDs

TM1624 and TM1640 are commonly used digital tubes/LEDs driver ICs. When using a driver IC to drive digital tubes/LEDs, it is only necessary to implement communication between the MCU and the driver IC and set the corresponding configuration commands and transmission data. Their setup interface is as follows:



In application, you only need to call the function TM1624_Display_Update/ TM1640_Display_Update to update the display content, as shown below:

```

}#if (APP_TM1624_ENABLE && TM1624_DISP_TEST)
    if (TM1624_Time > 500)
    {
        uint8_t i;
        TM1624_Time = 0;
        for (i = 0; i < 14; i++)
        {
            TM1624_Show[i] = ~TM1624_Show[i];
        }
        TM1624_Display_Update(TM1624_Show, 14, TM1624_ON | TM1624_DISP_DIM);
    }
#endif
}#if (APP_TM1640_ENABLE && TM1640_DISP_TEST)
    if (TM1640_Time > 500)
    {
        uint8_t i;
        TM1640_Time = 0;
        for (i = 0; i < 16; i++)
        {
            TM1640_Show[i] = ~TM1640_Show[i];
        }
        TM1640_Display_Update(TM1640_Show, 16, TM1640_ON | TM1640_DISP_DIM);
    }
#endif
}

```

4.3.3. Infrared Remote Control Reception

The infrared remote control reception function uses a timer to scan the IO port status for infrared signal decoding. The setup interface is as follows:



The application can call the function IR_Press to read the received infrared data, as shown below:

```

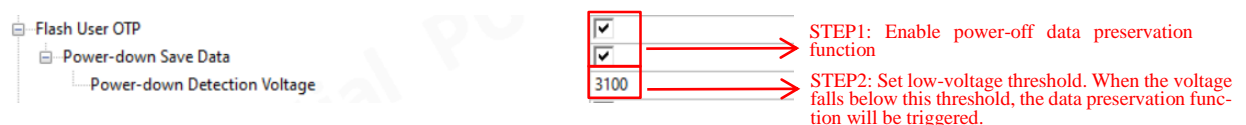
void app_drivers_loop(void)
{
    #if APP_IR_RECEIVED_ENABLE
        Ir_TypeDef remote;
        if (IR_Press(&remote))
        {
            log_printf("address:0X%X ", remote.ir_address); // Received address
            log_printf("command:0X%X ", remote.ir_command); // Received command
            log_printf("count:%d\r\n", remote.ir_count); // Received count
        }
    #endif
}

```

4.3.4. User Data Power-Down Storage

This function uses the ADC to periodically detect the chip's supply voltage. When the supply voltage falls below the set threshold, the program stores the user data that needs to be saved into FLASH.

The setup interface and steps are as follows:



ADC
ADC Sampling Interval Time
ADC-PA0-IN0
ADC-PA1-IN1
ADC-PA2-IN2
ADC-PA3-IN3
ADC-PA4-IN4
ADC-PA5-IN5
ADC-PA6-IN6
ADC-PA7-IN7
ADC-PB0-IN8
ADC-PB1-IN9
ADC-TEMPSENSOR
ADC-VREFINT
ADC-1_3VCCA

<input checked="" type="checkbox"/>
10
DISABLE
DISABLE
DISABLE
DISABLE
DISABLE
DISABLE
DISABLE
DISABLE
DISABLE
DISABLE
ENABLE
ENABLE
DISABLE

STEP3: Enable ADC and set the sampling interval time.

Note: The sampling time should be configured based on the actual power-down duration, ensuring that data storage is completed during the power-down process.

STEP4: Enable the ADC function to monitor the chip's power supply voltage.

Application example:

```
#if LVD_WRITE_USER_DATA
/*****
** Function name: void ADC_LVD_ISR(void)
** Description: Low voltage callback function
** Input: None
** Return: None
*****/
void ADC_LVD_ISR(void)
{
    Vcc_Power = 1200 * 4095 / ADCxConvertedData[ADC_CHANNEL_VREFINT];
    if (power_on == 1 && power_off == 0 && Vcc_Power > 2500 && Vcc_Power < low_voltage)
    {
        uint8_t sta;
        power_off = 1;
        power_count++;
        User_Cache_Write(0, &power_count, 1);
        sta = User_Flash_Write();
        log_printf("write_otp:%d\r\n", sta);
    }
}
#endif
```

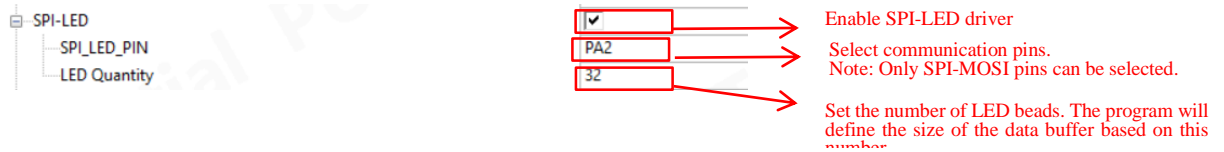
In this routine, power_count represents the number of power-down events. Each time a power-down occurs, the value increments. When the chip voltage is detected to be below the set threshold, power_count will be stored in the OTP area (Note: OTP is a dedicated FLASH sector with a capacity of 128 bytes). In practical applications, users can replace this with their own data as needed.

```
#if APP_USER_OTP_ENABLE
/* User data reading */
#endif
#if LVD_WRITE_USER_DATA
if (User_Cache_Read(0, &power_count, 1))
{
    log_printf("power_count:%d\r\n", power_count);
}
else
{
    log_printf("first power\r\n");
    power_count = 0;
}
#endif
#endif
```

After power-on, read the power_count value and print it out for verification.

4.3.5. W2812B-like LED Driver

This example uses SPI to simulate the communication timing of the magic color LED beads. The setup interface is as follows:



In the application, the application program first loads the data to be sent into the cache array `rgb_data`, and then calls the function `SPI_LED_Transmit` to send the data. The function `SPI_LED_Transmit` is shown below:

```

/*****
void SPI_LED_Transmit(void)
{
    for (uint16_t i = 0; i < SPI_LED_CNT * 6; i++)
    {
        while (LL_SPI_IsActiveFlag_TXE(SPI1) == 0)
        {
            ;
        }
        LL_SPI_TransmitData16(SPI1, rgb_data[i]);
    }
}

```

4.3.6. Buzzer Control

The buzzer setup interface is as follows:

BEEP	<input checked="" type="checkbox"/>
Buzzer Control Pin	PA15
Buzzer Valid Level	High
Buzzer Type	Active
Buzzer and ADC Pin Multiplexing	Not Multiplexed

In the application, the function BEEP_On can be invoked to control the buzzer, with the input parameter specifying the buzzer's activation duration, as shown below:

```

/*****
** Function name: void BEEP_On(uint8_t timeout)
** Description: Turn on the buzzer
** Input: timeout: turn-on time in ms
** Return: None
*****/
void BEEP_On(uint8_t timeout)
{
    beep_delay = timeout;
#ifdef BEEP_ADC
#ifdef APP_ADC_ENABLE
    while(adc_state == 1)    // In multiplexed state, if ADC is collecting, wait for collection to complete
    {
        __WFI();
    }
#endif
#endif
    ntc_delay = 5;
    GPIO_Init(BEEP_GPIO, OUTPUT | PUSH_PULL);
#ifdef BEEP_TYPE
    GPIO_Init(BEEP_GPIO, OUTPUT | PUSH_PULL);
#endif
#ifdef BEEP_GPIO_TYPE
    GPIO_SetBit(BEEP_GPIO);
#else
    GPIO_ClearBit(BEEP_GPIO);
#endif
}

```

应用实例:

```

for (i = 0; i < TKCtr.TouchKeyChCnt; i++)
{
    if (temp & 0X01)
    {
        if (KeyFlag & ((0X01) << i)) // Key pressed
        {
#ifdef APP_BEEP_ENABLE
            BEEP_On(100);
#endif

```

5. User Application Program Interface

The user_code.c file contains the interface functions for the user application program. The interface functions are listed in the table below:

Function Name	Functional Description
user_init	User application initialization function. Users can place their initialization programs within this function. It is typically called once during system startup before the main loop begins.
user_loop	Main loop task function. This function is called cyclically within the main loop. Users can place programs that need to run continuously inside this function.
user_timer	System tick timer interrupt callback function. This function is called within the SYS_TICK interrupt service routine. Users can place time-sensitive programs that require precise timing inside this function.
ADC_Callback	ADC conversion completion callback function. This function is called after the ADC conversion is finished. Users can read the ADC data and add corresponding processing programs inside this function.

6. Version history

Version	Date	Description
V1.0	2024.03.20	1. Initial version release



Puya Semiconductor Co., Ltd.

IMPORTANT NOTICE

Puya reserve the right to make changes, corrections, enhancements, modifications to Puya products and/or to this document at any time without notice. Purchasers should obtain the latest relevant information of Puya products before placing orders.

Puya products are sold pursuant to terms and conditions of sale in place at the time of order acknowledgement.

Purchasers are solely responsible for the choice and use of Puya products. Puya does not provide service support and assumes no responsibility when products that are used on its own or designated third party products.

Puya hereby disclaims any license to any intellectual property rights, express or implied.

Resale of Puya products with provisions inconsistent with the information set forth herein shall void any warranty granted by Puya.

Any with Puya or Puya logo are trademarks of Puya. All other product or service names are the property of their respective owners.

The information in this document supersedes and replaces the information in the previous version.

Puya Semiconductor Co., Ltd. – All rights reserved