

GD32F130xx 快速开发指南

-基于技新 GD32F130G8U6 核心板



前言	5
GD32F130xx 开发平台介绍	6
1、 GD32F130xx 系列芯片介绍	6
2、 软件平台介绍	8
3、 硬件平台介绍	8
3.1 技新 GD32F130G8U6 核心板	8
3.2 下载&仿真器	9
GD32F1x0 开发环境搭建	11
GD32F130G8 新建工程	16
GD32F130G8 程序下载	23
1、 CMSIS-DAP 仿真器（以 GD-LINK 例）	23
2、 串口	25
第一章 GPIO 应用	29
1、 GPIO 简介	29
2、 点亮 LED	30
2.1 GPIO 的输出配置	30
2.2 例程介绍	31
3、 KEY 按键检测	32
3.1 GPIO 的输入配置	32
3.2 例程介绍	32
第二章 EXTI 应用	34
1、 EXTI 简介	34
2、 KEY 外部中断检测	37
2.1 KEY 的外部中断配置	37
2.2 例程介绍	37
第三章 CCTL 应用	40
1、 CCTL 简介	40
2、 时钟输出	41
2.1 系统时钟输出配置	41
2.2 例程介绍	41
第四章 USART 应用	45
1、 USART 简介	45
2、 print 输出	47
2.1 配置 printf 输出	47
2.2 例程介绍	47
3、 串口收发中断	50
3.1 串口中断配置	50
3.2 例程介绍	50
4、 串口 DMA 传输	53
4.1 串口 DMA 配置	53

4.2 例程介绍	53
第五章 TIMER 应用	57
1、高级定时器 TIMER0 简介	57
2.1 互补信号输出配置	58
2.2 例程介绍	59
3、TIMER0 的 DMA 更新事件输出 PWM	61
3.1 TIMER0 的 DMA 更新事件配置	61
3.2 例程介绍	62
4、通用定时器 TIMER1&TIMER2 简介	65
5、TIMER1 的 PWM 输出	66
5.1 TIMER1 的 PWM 配置	66
5.2 例程介绍	67
6、TIMER1 的单脉冲输出	69
6.1 TIMER1 的单脉冲配置	69
6.2 例程介绍	70
7、TIMER2 的输入捕获	72
7.1 TIMER2 的输入捕获配置	72
7.2 例程介绍	73
8、TIMER2 的 PWM 捕获	76
8.1 TIMER2 的 PWM 捕获配置	76
8.2 例程介绍	77
第六章 I2C 应用	81
1、I2C 简介	81
2、I2C 与 0.96 寸 OLED 模块通讯	82
2.1 I2C 的配置	82
2.2 例程介绍	84
第七章 SPI 应用	88
1、SPI 简介	88
2、SPI 与 0.96 寸模块通讯	89
2.1 SPI 的配置	89
2.2 例程介绍	89
第八章 ADC 应用	93
1、ADC 简介	93
2、ADC 规则组的连续转换功能	94
2.1 ADC 规则组的连续转换功能配置	94
2.2 例程介绍	95
3、规则组的 DMA 功能	97
3.1 规则组的 DMA 功能配置	97
3.2 例程介绍	98
第九章 FWDGT 应用	102

4、 FWDGT 简介.....	102
5、 FWDGT 应用.....	103
2.1 FWDGT 的配置.....	103
2.2 例程介绍.....	103
第十章 WWDGT 应用.....	105
1、 WWDGT 简介.....	105
2、 WWDGT 应用.....	106
2.1 WWDGT 的配置.....	106
2.2 例程介绍.....	107

前言

目前市场上 Cortex-M3 内核的芯片受到越来越多的电子工程师青睐，同时在水场上的应用也越来越广泛，GD 针对这一市场开发了一款基于 Cortex-M3 内核的超值型系列芯片 GD32F1x0，相对市场上其他的 Cortex-M3 内核 MCU，GD32F1x0 具有低成本、高性能等优势。

本教程结合官方的用户手册以及固件库例程，通过实际例程讲解以及实验现象来帮助读者理解和使用 GD32F130xx 这个系列的芯片。软件平台使用的是 MDK-ARM 和官方外设驱动库 GD32F1x0_Firmware_Library_v3.1.0（库函数开发），硬件使用技新 GD32F130G8U6 核心板 V1.0 和 GD-LINK 下载&调试器。

教程从开发平台介绍、开发环境搭建、建立工程等基础内容，到 GD13F130xx 外设应用，包括：GPIO 应用、EXTI 应用、CLK 应用、USART 应用、TIMER 应用、I2C 应用、SPI 应用、ADC 应用、FWDGT 应用和 WWDGT 应用等十大部分内容。外设应用部分的内容都配有源码，并配合硬件平台进行实验讲解。教程面对的对象是具有一定的 MCU 编程基础以及 C 语言基础的，主旨是帮助开发者快速入门和快速开发使用 GD32F130xx 系列产品。

发教程的所有资料，包括软件、硬件、开发工具、PDF 文档等等可以到[技新网](http://www.jixin.pro/)上下载。

技新网：<https://www.jixin.pro/>

GD 官网：<http://www.gigadevice.com/>

GD32 官网：<http://gd32mcu.21ic.com/site>

GD32F130xx 开发平台介绍

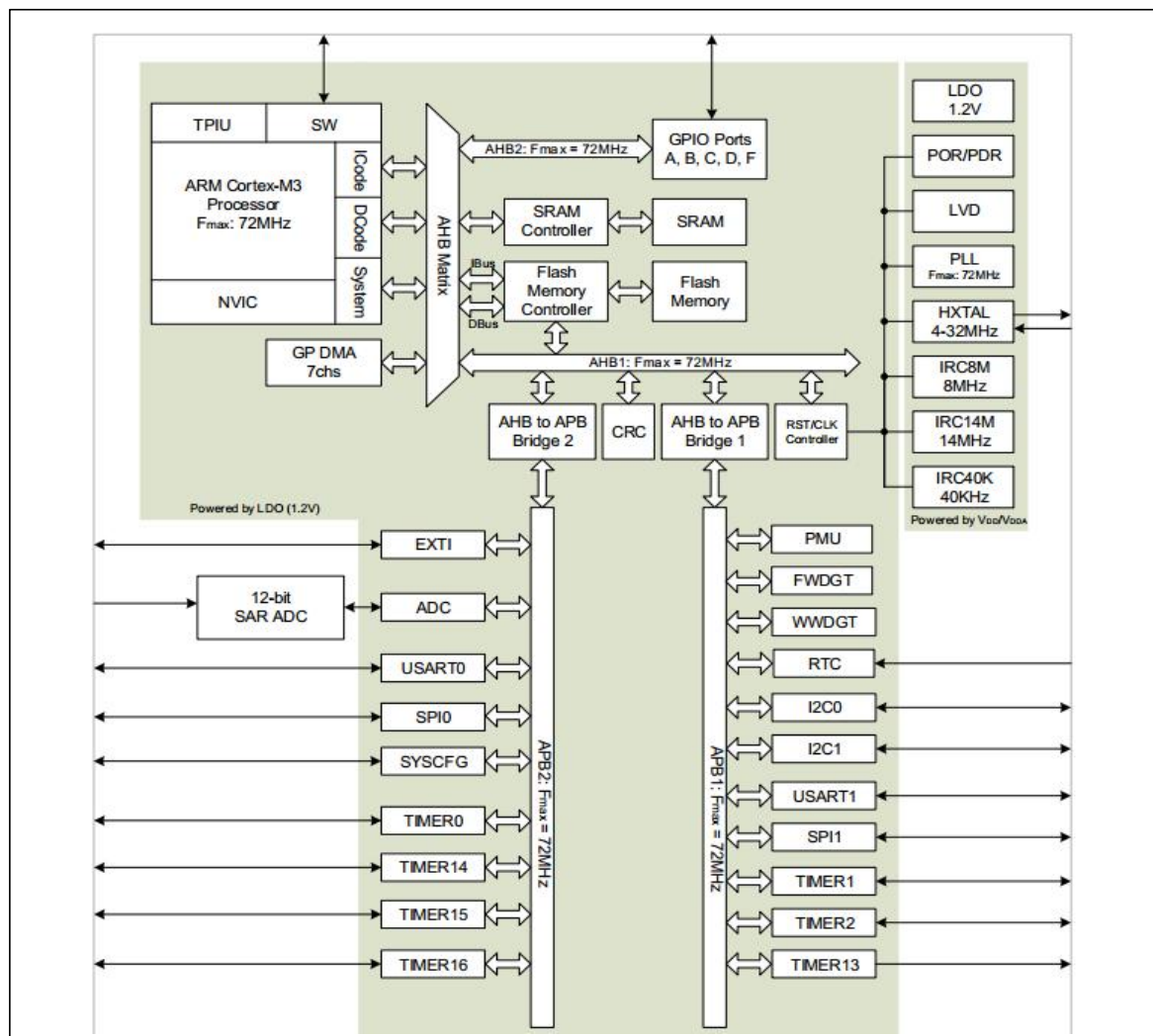
GD32F13xx 系列属于 GD32 Cortex-M3 超值型系列 MCU，它拥有高性价比、主频高至 72MHz、FLASH 访问零等待等特点。本节内容从以下几方面介绍 GD32F130xx 系列的开发平台：

- GD32F130xx 系列芯片介绍
- 软件平台介绍
- 硬件平台介绍

1、GD32F130xx 系列芯片介绍

GD32F13xx 系列属于 GD32 Cortex-M3 超值型系列 MCU 家族，它是一个基于 ARM[®] Cortex[®]-M3 RISC 内核的通用 MCU，运行在 72MHz 频率下，FLASH 访问零等待可获取最大效率，并提供片内最大 64KB 的 FLASH 存储器和 8KB 的 SRAM 存储器，I/O 口与外设挂在在两条 APB 总线，并提供有一个 12-bit ADC、最多可达 5 个通用的 16-bit 定时器、一个 32-bit 定时器、一个 PWM 高级定时器、可作为标准和高级通讯接口：最多可达两路 SPI，两路 I2C 和两路 USARTs。

GD32F13xx 系列工作电压在 2.6V 到 3.6V 之间，工作温度范围-40 到+85℃。GD32F130xx 板块图如下所示：



GD32F130xx 系列性能与外设列表如下：

Part Number		GD32F130xx												
		F4	F6	F8	G4	G6	G8	K4	K6	K8	C4	C6	C8	R8
Flash (KB)		16	32	64	16	32	64	16	32	64	16	32	64	64
SRAM (KB)		4	4	8	4	4	8	4	4	8	4	4	8	8
Timers	General timer(32-bit)	1 <small>(1)</small>	1 <small>(1)</small>	1 <small>(1)</small>	1 <small>(1)</small>	1 <small>(1)</small>	1 <small>(1)</small>	1 <small>(1)</small>	1 <small>(1)</small>	1 <small>(1)</small>	1 <small>(1)</small>	1 <small>(1)</small>	1 <small>(1)</small>	1 <small>(1)</small>
	General timer(16-bit)	4 <small>(2,13,15-16)</small>	4 <small>(2,13,15-16)</small>	4 <small>(2,13,15-16)</small>	4 <small>(2,13,15-16)</small>	4 <small>(2,13,15-16)</small>	5 <small>(2,13-16)</small>	4 <small>(2,13,15-16)</small>	4 <small>(2,13,15-16)</small>	5 <small>(2,13-16)</small>	4 <small>(2,13,15-16)</small>	4 <small>(2,13,15-16)</small>	5 <small>(2,13-16)</small>	5 <small>(2,13-16)</small>
	Advanced timer(16-bit)	1 <small>(0)</small>	1 <small>(0)</small>	1 <small>(0)</small>	1 <small>(0)</small>	1 <small>(0)</small>	1 <small>(0)</small>	1 <small>(0)</small>	1 <small>(0)</small>	1 <small>(0)</small>	1 <small>(0)</small>	1 <small>(0)</small>	1 <small>(0)</small>	1 <small>(0)</small>
	SysTick	1	1	1	1	1	1	1	1	1	1	1	1	1
	Watchdog	2	2	2	2	2	2	2	2	2	2	2	2	2
	RTC	1	1	1	1	1	1	1	1	1	1	1	1	1
Connectivity	USART	1 <small>(0)</small>	2 <small>(0-1)</small>	2 <small>(0-1)</small>	1 <small>(0)</small>	2 <small>(0-1)</small>	2 <small>(0-1)</small>	1 <small>(0)</small>	2 <small>(0-1)</small>	2 <small>(0-1)</small>	1 <small>(0)</small>	2 <small>(0-1)</small>	2 <small>(0-1)</small>	2 <small>(0-1)</small>
	I2C	1 <small>(0)</small>	1 <small>(0)</small>	2 <small>(0-1)</small>	1 <small>(0)</small>	1 <small>(0)</small>	2 <small>(0-1)</small>	1 <small>(0)</small>	1 <small>(0)</small>	2 <small>(0-1)</small>	1 <small>(0)</small>	1 <small>(0)</small>	2 <small>(0-1)</small>	2 <small>(0-1)</small>
	SPI	1 <small>(0)</small>	1 <small>(0)</small>	2 <small>(0-1)</small>	1 <small>(0)</small>	1 <small>(0)</small>	2 <small>(0-1)</small>	1 <small>(0)</small>	1 <small>(0)</small>	2 <small>(0-1)</small>	1 <small>(0)</small>	1 <small>(0)</small>	2 <small>(0-1)</small>	2 <small>(0-1)</small>
GPIO		15	15	15	23	23	23	27	27	27	39	39	39	55
EXTI		16	16	16	16	16	16	16	16	16	16	16	16	16
ADC	Units	1	1	1	1	1	1	1	1	1	1	1	1	1
	Channels (External)	9	9	9	10	10	10	10	10	10	10	10	10	16
	Channels (Internal)	3	3	3	3	3	3	3	3	3	3	3	3	3
Package		TSSOP20			QFN28			QFN32			LQFP48			LQFP 64

2、软件平台介绍

目前常用于 GD32 系列 MCU 的 IDE（集成开发环境）有两个：MDK-ARM 与 IAR For ARM，GD32 官方也发布了关于这两个环境的 GD32F1x0 器件支持包。

此外 GD32 官方在 2018-02-08 发布 GD32F1x0_Firmware_Library_v3.1.0，简称 GD32F1x0 外设驱动程序和通用例程，支持 GD32F GD32F130 / GD32F150 / GD32F170 / GD32F190 包含 USB Device 驱动程序（仅支持 GD32F150）和例程，并提供 Keil 和 IAR 两种工程。

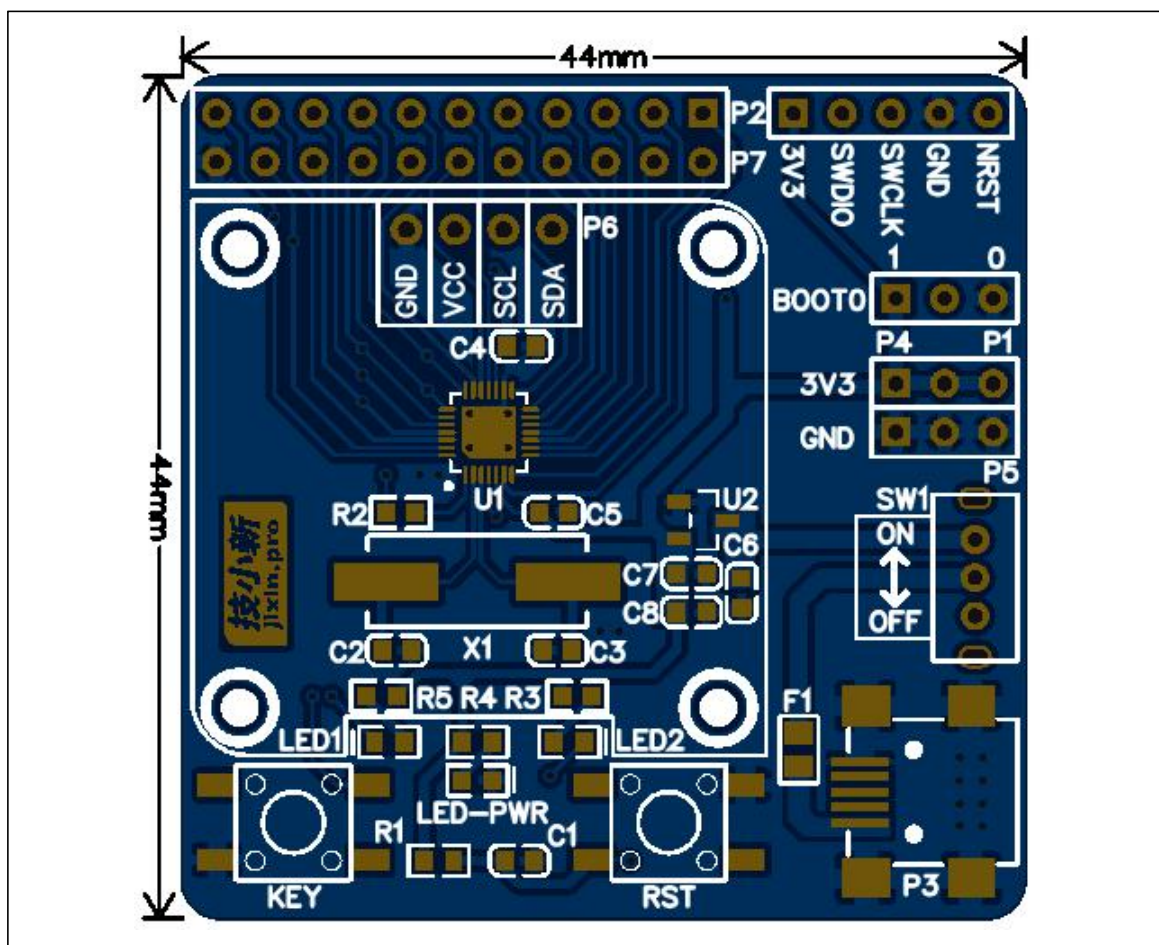
本系列课程使用的开发方式是 MDK + 固件库，软件平台如下：

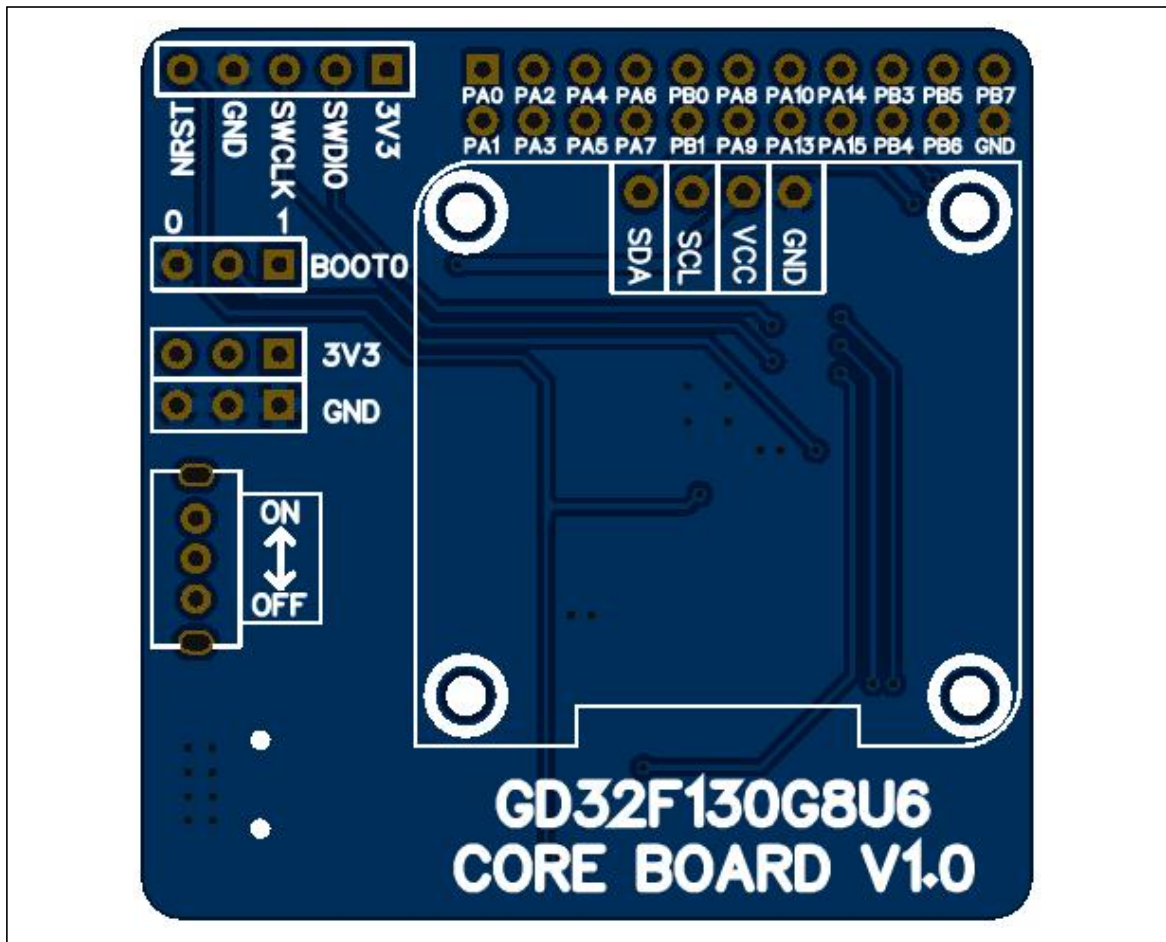
- 开发环境：MDK-ARM + GD32F1x0 器件支持包
- 固件库：GD32F1x0_Firmware_Library_v3.1.0

3、硬件平台介绍

3.1 技新 GD32F130G8U6 核心板

硬件平台使用的是技新 GD32F130G8U6 核心板 V1.0，核心板带有下载接口、BOOT0 接口、3.3V 电源接口、4Pin 的 0.96' OLED 接口、用户 LED 两个、用户按键一个、复位按键一个、MINI_USB 供电接口、引出剩余 21 个可用 GPIO、板子体积小方便调试测试，核心板的 PCB 预览图（正面&背面）如下：





技新 GD32F130G8U6 核心板 V1.0 使用的是 LCEDA（云端 PCB 设计工具），并在 LCEDA 上开源，可在网址 <https://lceda.cn/jixin> 上查看。

3.2 下载&仿真器

GD32 系列 MCU 可以采用 CMSIS-DAP 仿真器进行下载&调试，如 GD 官方的 GD-LINK，还有其他的 CMSIS-DAP 等仿真器：



使用技新 GD32F130G8U6 核心板 V1.0 可以采取两种方式进行下载程序：CMSIS-DAP 仿真器（如 GD-LINK）和串口。官方的 GD-LINK 可以在线编程（在编译器中配置并下载），也可以脱机编程（使用 GD-Link 编程调试工具下载）。使用串口，需要一个 USB 转 TTL 模块连接电脑与核心板的 USART1（PA9-TX，PA10-RX），配置 BOOT0 为高电平，然后使用 GD 的串口下载调试下载.HEX 文件，下载后把 BOOT0 配置为低电平重新上电。

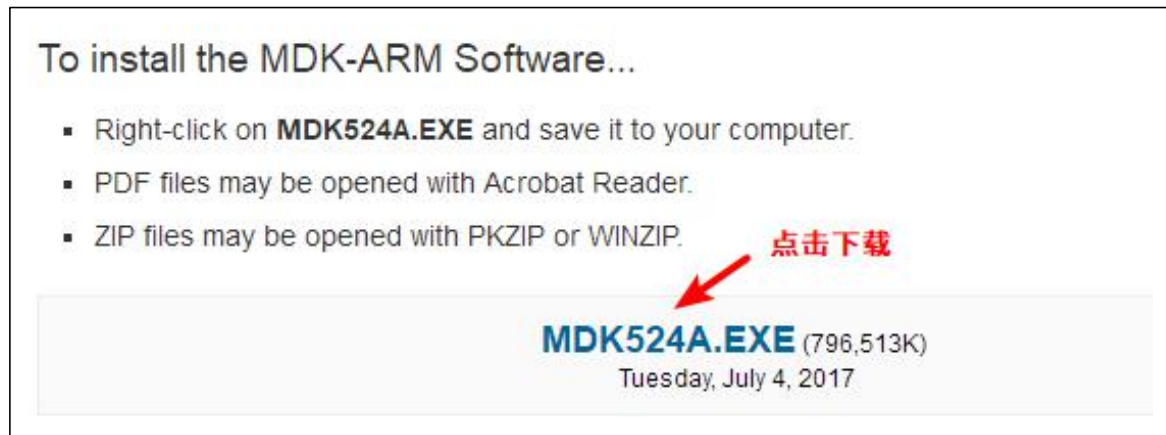
硬件平台如下：


- 实验平台：技新 GD32F130G8U6 核心板
- 下载&仿真器：CMSIS-DAP 仿真器（如 GD-LINK）

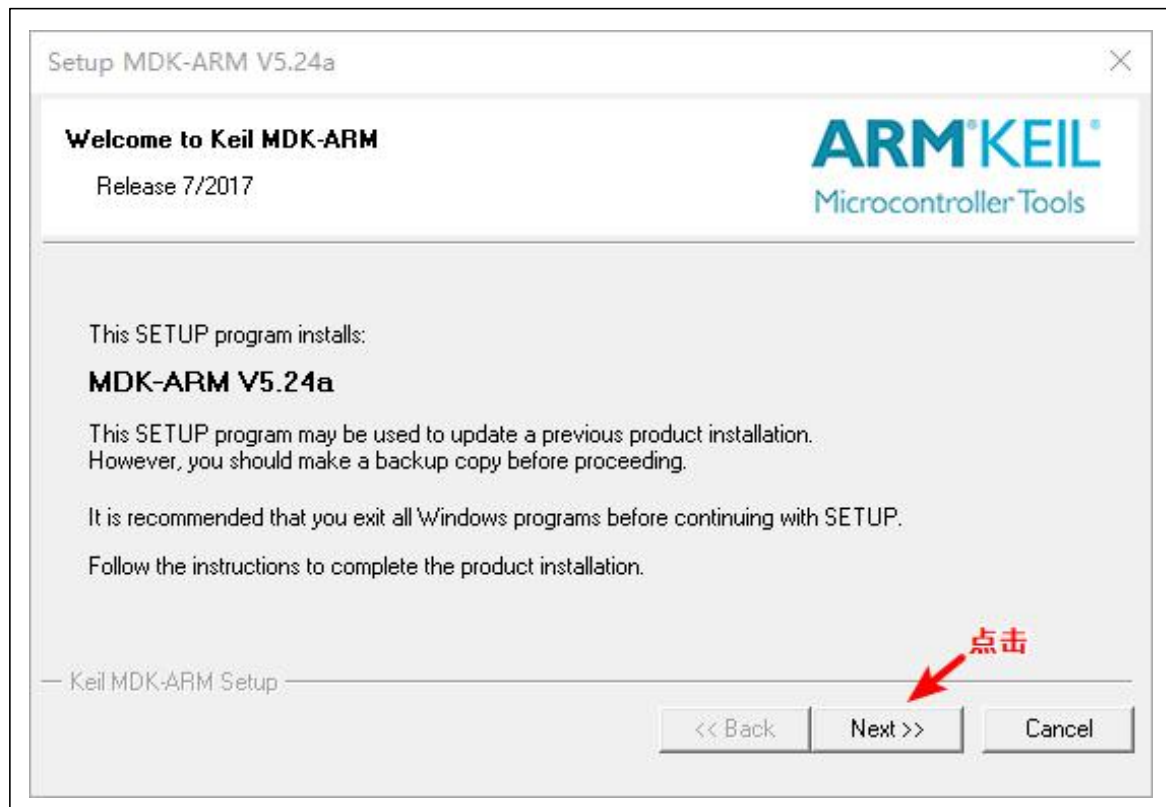
GD32F1x0 开发环境搭建

本节介绍基于 MDK-ARM（5.24a 版本）+ GD32F1x0 系列的器件支持包（用户也可根据需求安装其他的固件包）的 GD32 开发环境搭建，用户根据下面步骤进行开发环境搭建：

- 1) 打开网址 <https://www.keil.com/demo/eval/arm.htm>，进入 KEIL 官网的下载页面，点击下载软件（没有注册的用户需要填写信息才能下载）



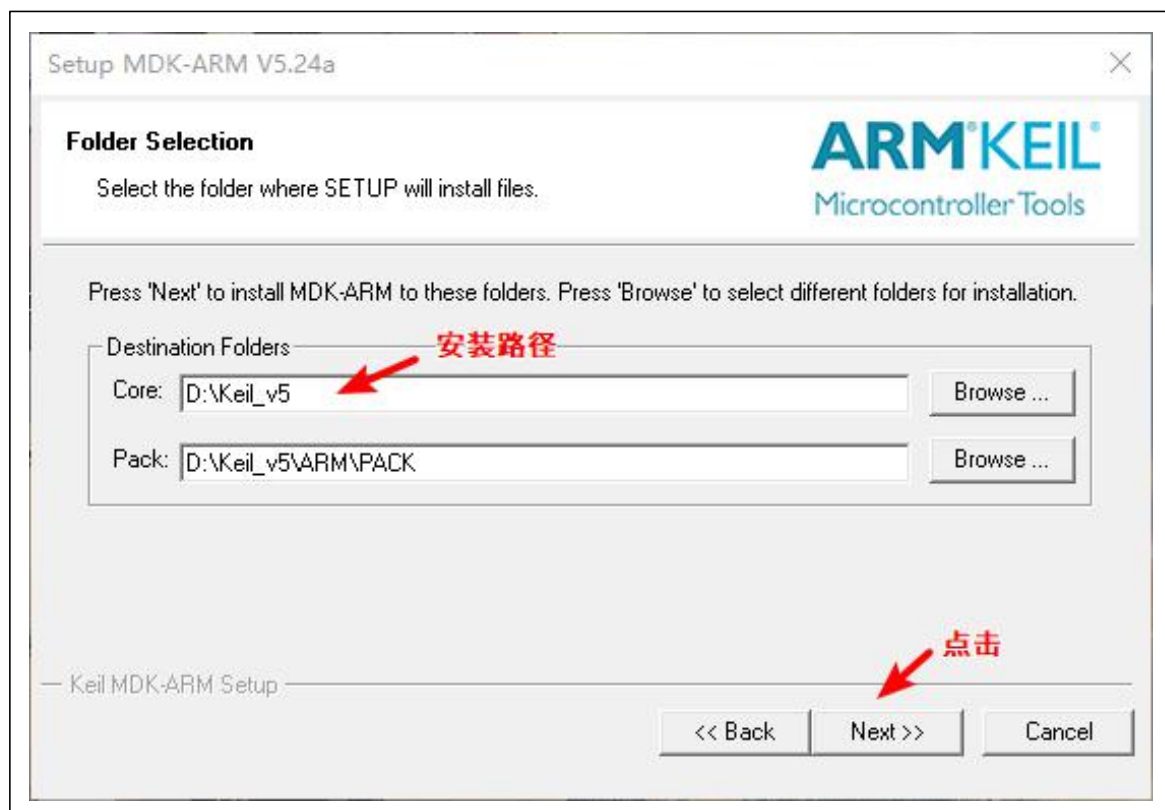
- 2) 下载完成后，双击  MDK524a.EXE 进行安装，在 Welcome 窗口点击 Next 进入下一步



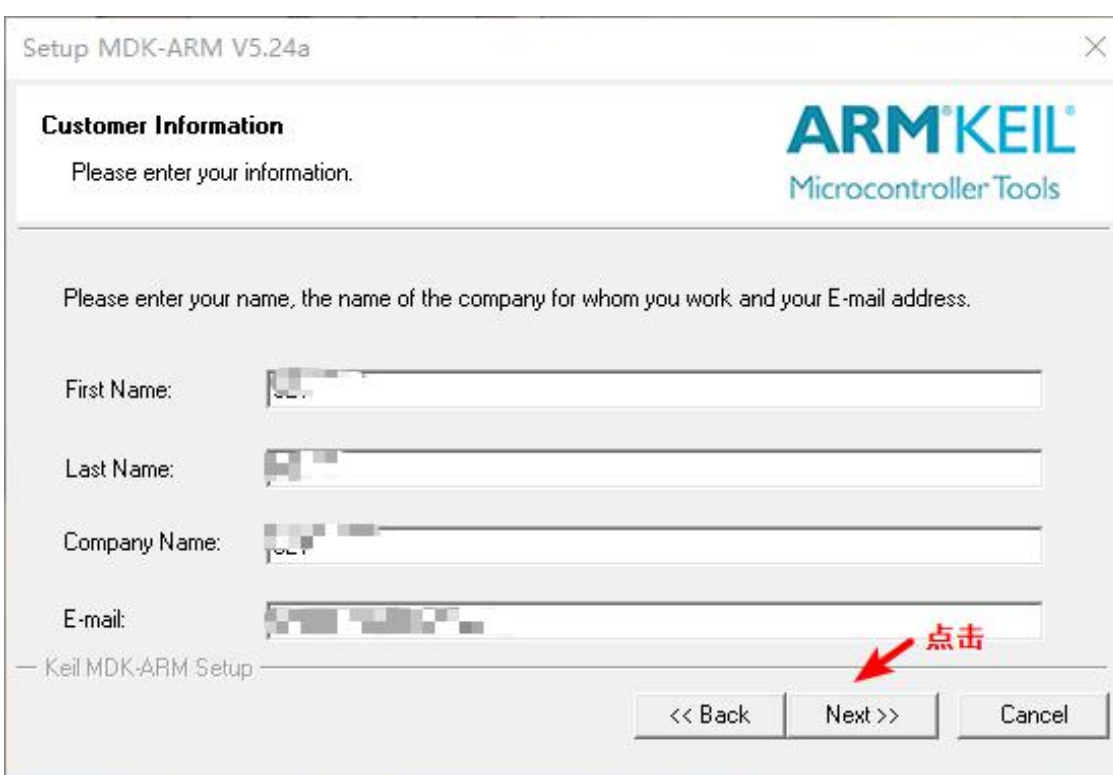
- 3) 在 License Agreement 窗口勾选 I agree..., 然后点击 Next



- 4) 选择安装路径，安装在 D:\Keil_v5（可根据个人习惯，安装到其他地方），点击 Next



- 5) 在 Customer Information 窗口填入相关信息（可随意填），然后点击 Next



Setup MDK-ARM V5.24a

Customer Information

Please enter your information.

ARM[®] KEIL[®]
Microcontroller Tools

Please enter your name, the name of the company for whom you work and your E-mail address.

First Name:

Last Name:

Company Name:

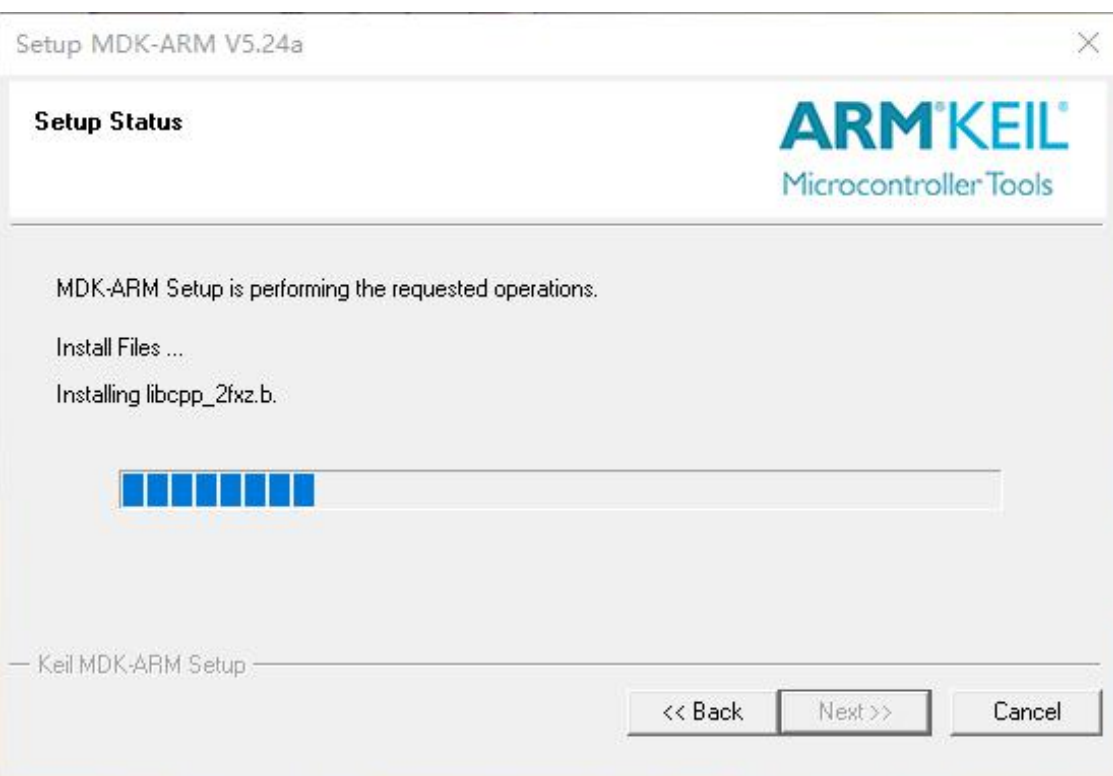
E-mail:

— Keil MDK-ARM Setup —

<< Back Next >> Cancel

点击

- 6) 软件开始安装，安装过程大概几分钟左右



Setup MDK-ARM V5.24a

Setup Status

ARM[®] KEIL[®]
Microcontroller Tools

MDK-ARM Setup is performing the requested operations.

Install Files ...

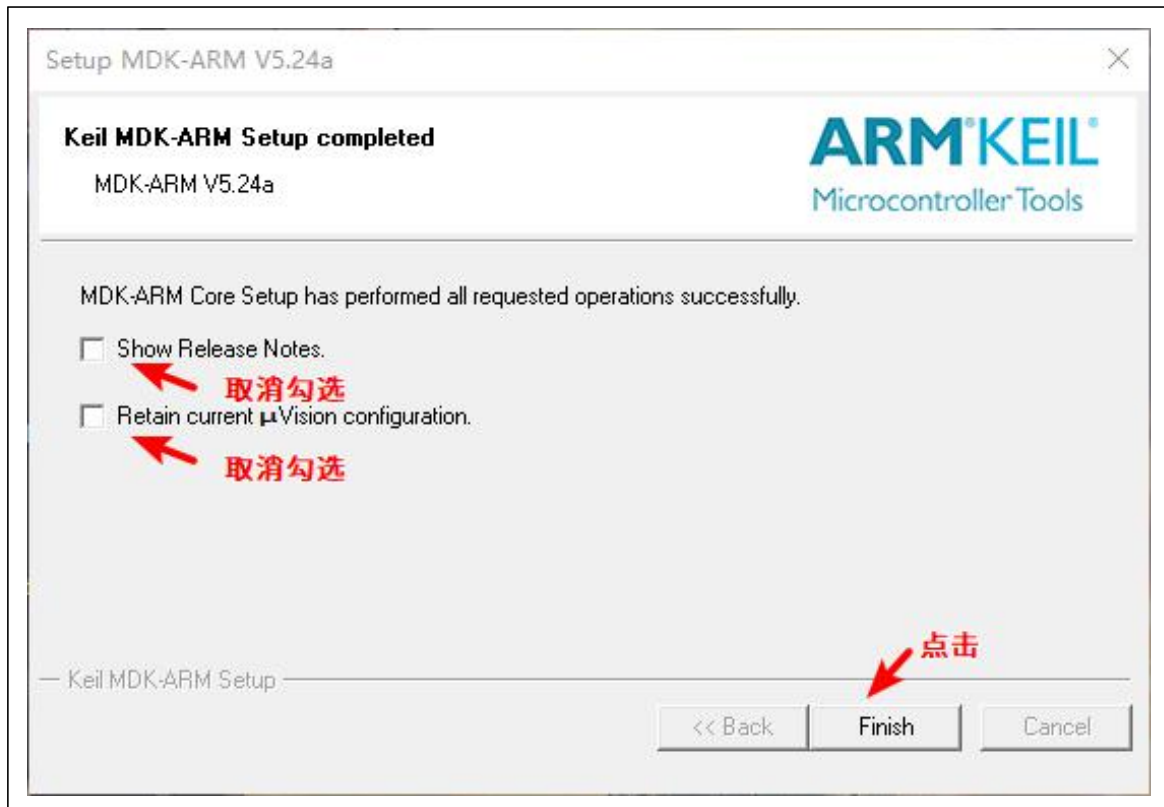
Installing libcpp_2fxz.b.

Progress bar: [|||||]

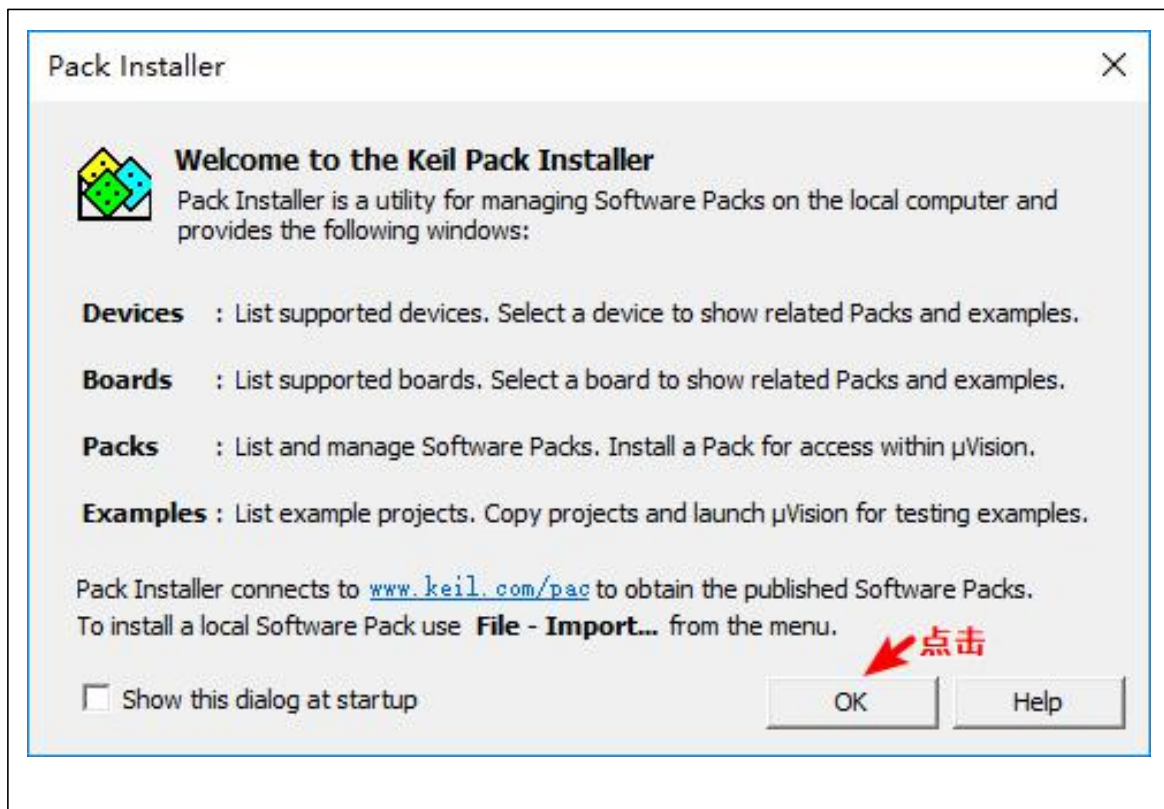
— Keil MDK-ARM Setup —


<< Back Next >> Cancel

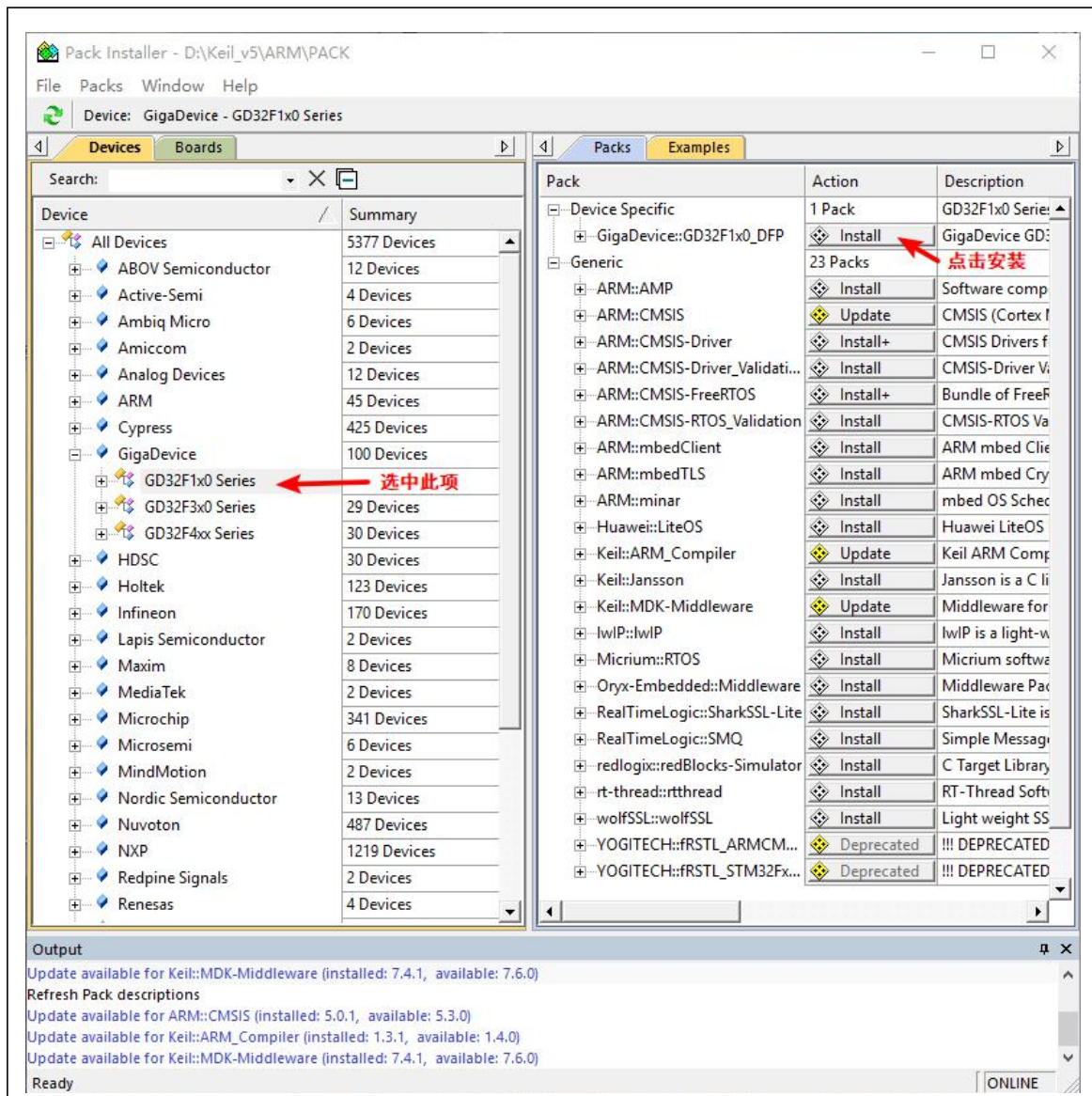
7) 安装完成后，取消 Show...与 Retain...前面的勾选，点击 Finish



8) 点击 Finish 后会弹出 Pack Installer 和 Pack Install 窗口，用于安装固件包，点击 OK



9) 在 Pack Installer 窗口点击刷新按钮 ，在 Device 下把 GigaDevice 展开，选中 GD32F1x0 Series（用户可根据需求安装其他的固件包），然后点击右上边对应 GigaDevice::GD32F1x0_DFP 的 Install 按钮开始下载并安装（在右下角会显示安装进度）



10) 安装完成后，右边的窗口对应 GigaDevice::GD32F1x0_DFP 的按钮会变为 Up to date



至此 GD32F1x0 的开发环境已经搭建完成（在桌面可找到 keil uVision5 的快捷方式），MDK 使用限制代码在 32KByte，购买注册码激活可无限制使用（激活方法可百度：KEIL 激活）。

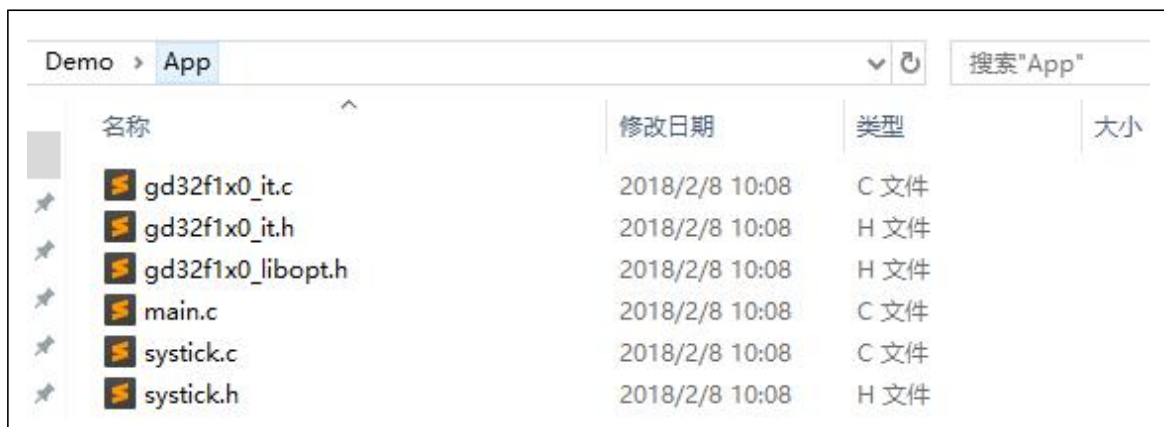
GD32F130G8 新建工程

本节介绍基于 MDK-ARM + GD32F1x0_Firmware_Library_v3.1.0（官方固件库）来新建 GD32F130G8 工程，GD32F1x0_Firmware_Library_v3.1.0 固件库可以在 GD32 官方网址下载，网址：http://gd32mcu.21ic.com/documents/index/classify_id/7：

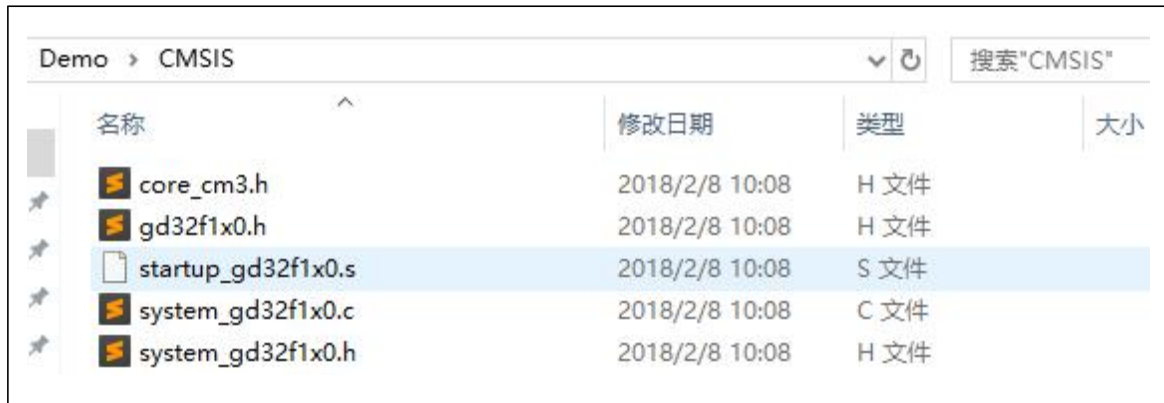
- 1) 在文件夹并命名 Demo，在 Demo 文件夹下分别建立 App、CMSIS、Doc、StdPeriph_Driver、Utilities 等 5 个文件夹，每个文件夹作用如下所示：



- 2) 把从官下载的固件库文件 GD32F1x0_Firmware_Library_v3.1.0.rar 解压，把 GD32F1x0_Firmware_Library_v3.1.0\Examples\GPIO\Running_led 目录下除了 readme.txt 的所有文件复制到 Demo\App 目录下，readme.txt 复制到 Demo\Doc 目录下：

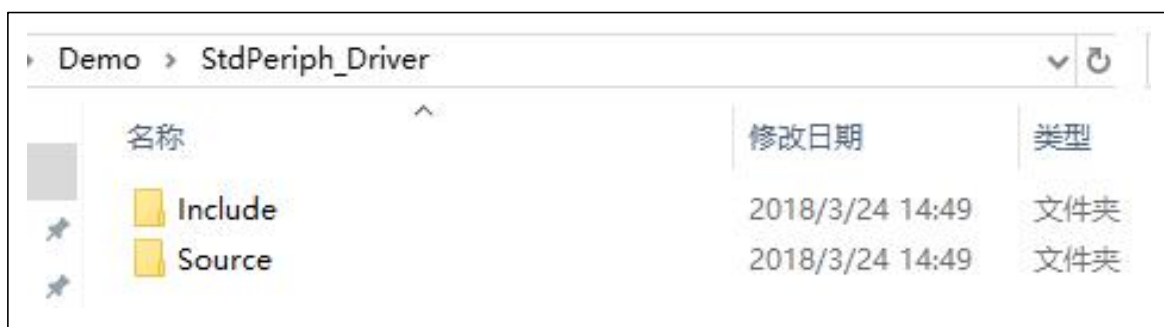


- 3) 把固件库文件 GD32F1x0_Firmware_Library_v3.1.0\Firmware\CMSIS 目录下的 core_cm3.h , ...Firmware\CMSIS\GD\GD32F1x0\Include 目录下的 gd32f1x0.h 、 system_gd32f1x0.h , ...Firmware\CMSIS\GD\GD32F1x0\Source 目录下的 system_gd32f1x0.c , 把 ...Firmware\CMSIS\GD\GD32F1x0\Source\ARM 目录下的 startup_gd32f1x0.s 等文件复制到 Demo\CMSIS 目录下：



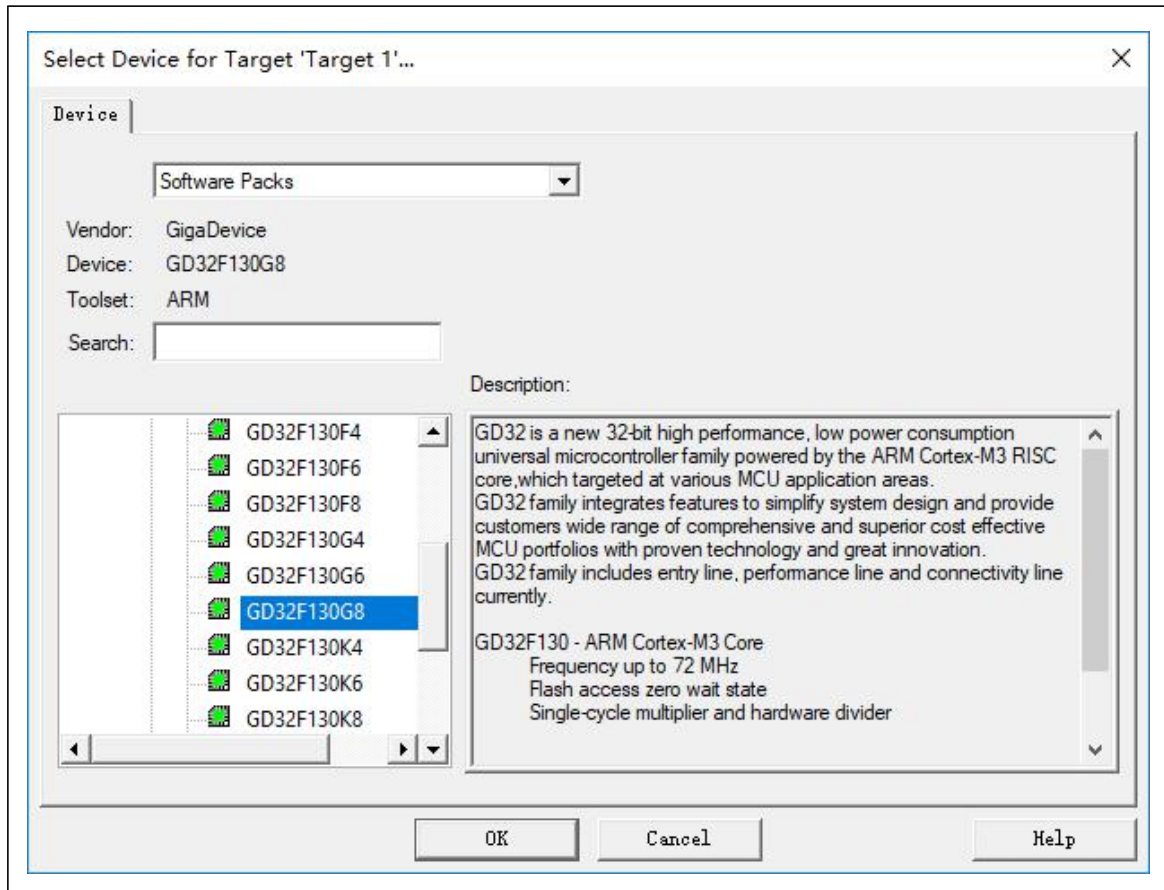
名称	修改日期	类型	大小
core_cm3.h	2018/2/8 10:08	H 文件	
gd32f1x0.h	2018/2/8 10:08	H 文件	
startup_gd32f1x0.s	2018/2/8 10:08	S 文件	
system_gd32f1x0.c	2018/2/8 10:08	C 文件	
system_gd32f1x0.h	2018/2/8 10:08	H 文件	

- 4) 把固件库 GD32F1x0_Firmware_Library_v3.1.0\Firmware\GD32F1x0_standard_peripheral 目录下的 Include、Source 文件夹复制到 Demo\StdPeriph_Driver 目录下：

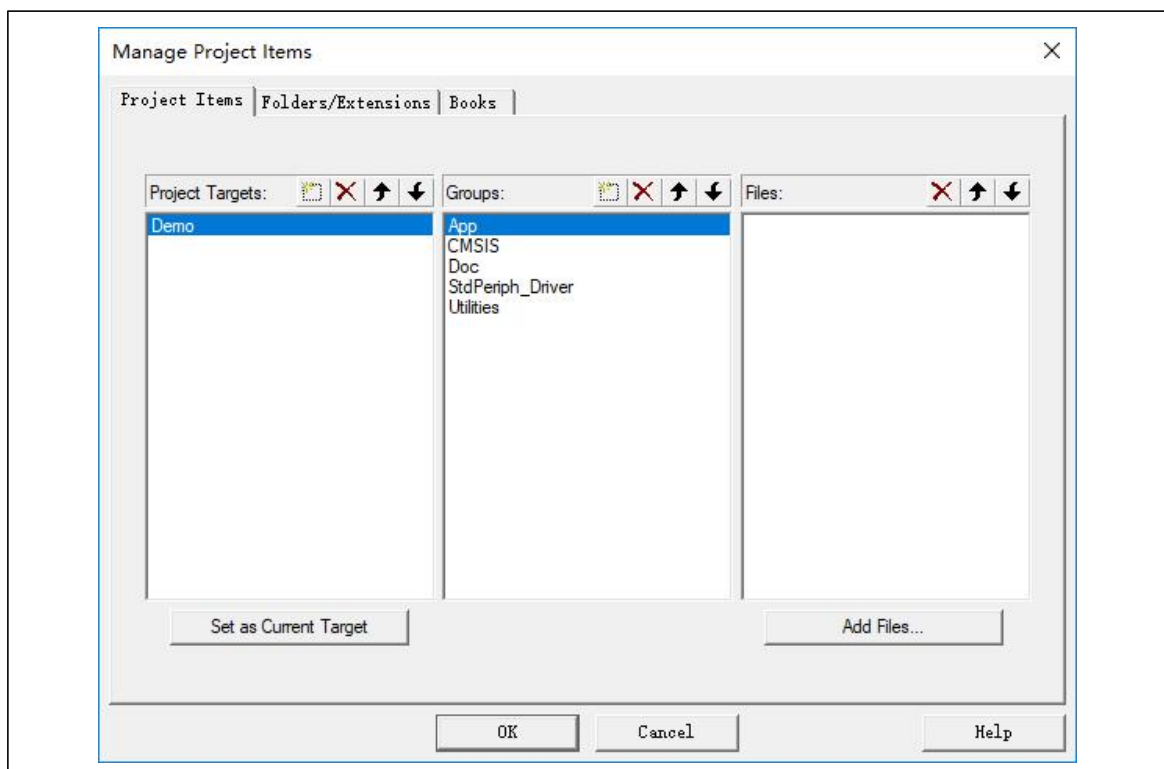


名称	修改日期	类型
Include	2018/3/24 14:49	文件夹
Source	2018/3/24 14:49	文件夹

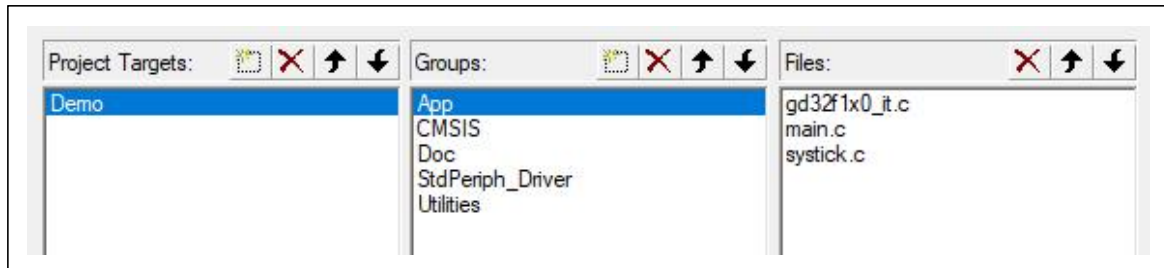
- 5) 打开MDK工具，点击 Project-->New µVision Project 新建工程，命名为 Demo 并保存到 Demo 文件夹下，然后在选择器件的窗口中选择 GD32F130G8，然后点击 OK



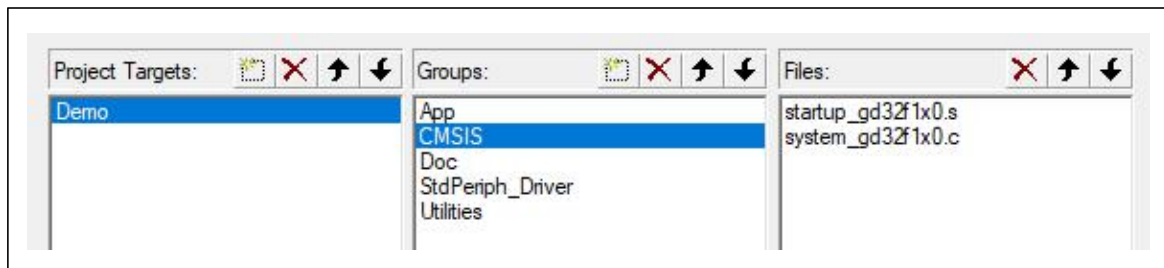
- 6) 弹出的 Manage Run-Time Environment 窗口直接点击 X 关掉；在左边的 Project 窗口选中 Target 1 右键选择 Manage Project Items...打开 Manage Project Items 窗口，Project Targets 下建立 Demo 目标，Groups 下建 App、CMSIS、Doc、StdPeriph_Driver、Utilities 等 5 个组：



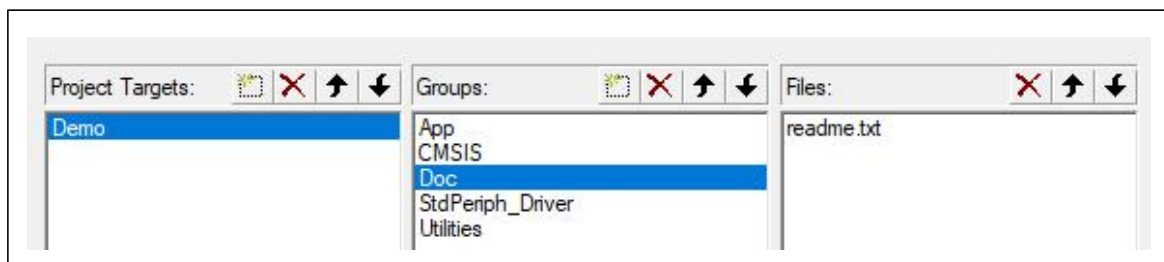
- 7) 往 Groups 下的 APP 组添加 Demo\App 目录下的 gd32f1x0_it.c、main.c、systick.c 三个文件：



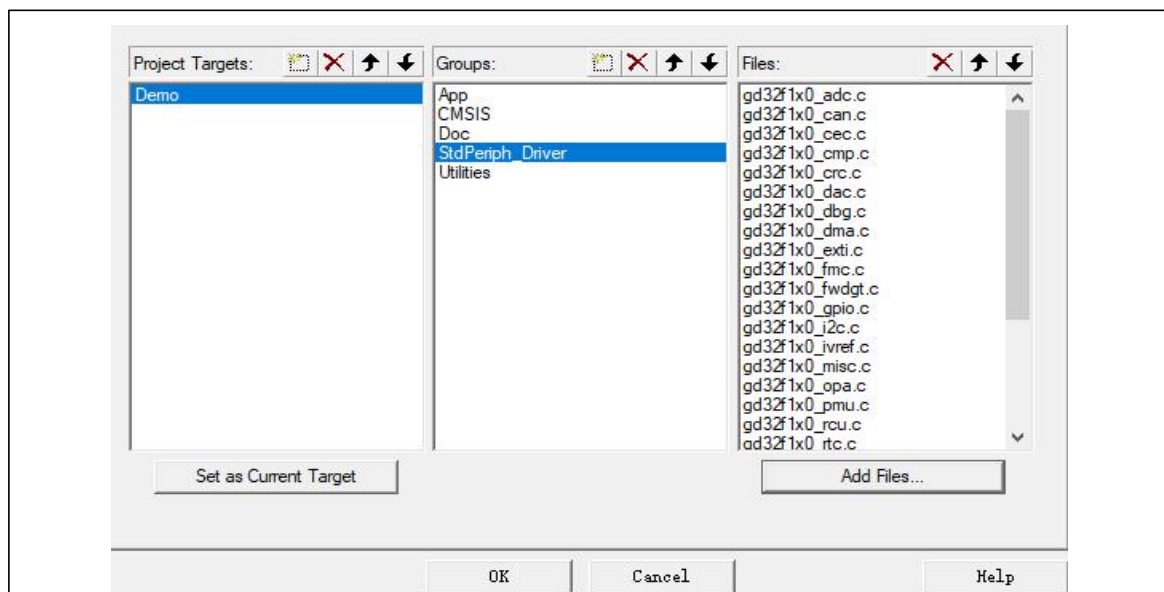
- 8) 往 Groups 下的 CMSIS 组添加 Demo\CMSIS 目录下的 startup_gd32f1x0.s、system_gd32f1x0.c 文件：



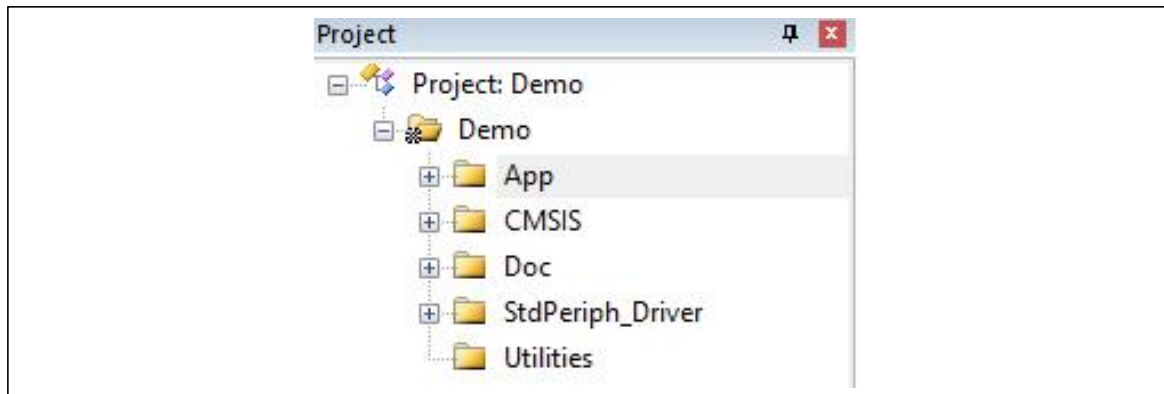
- 9) 往 Groups 下的 Doc 组添加 Demo\Doc 目录下的 readme.txt 文件：



- 10) 往 Groups 下 StdPeriph_Driver 组添加 Demo\StdPeriph_Driver\Source 目录下的所有.c 文件，之后点击 OK：



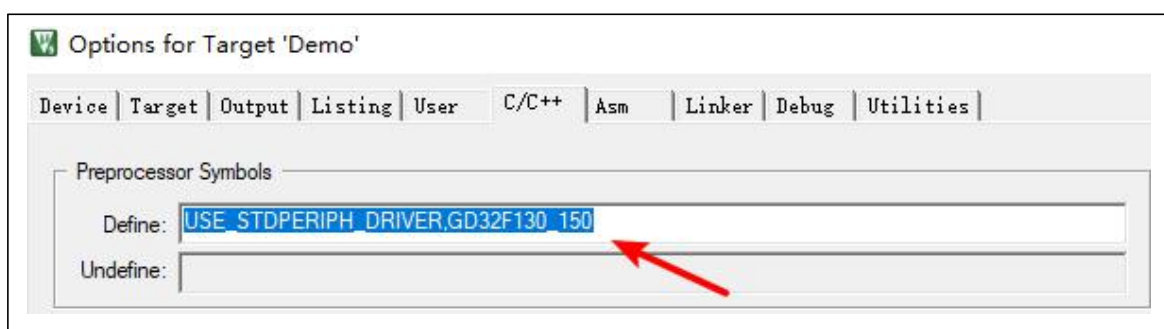
- 11) Utilities 文件是存放用户代码的，这里没有所以不需要添加，按上述添加完后 Project 窗口下的接口如下：



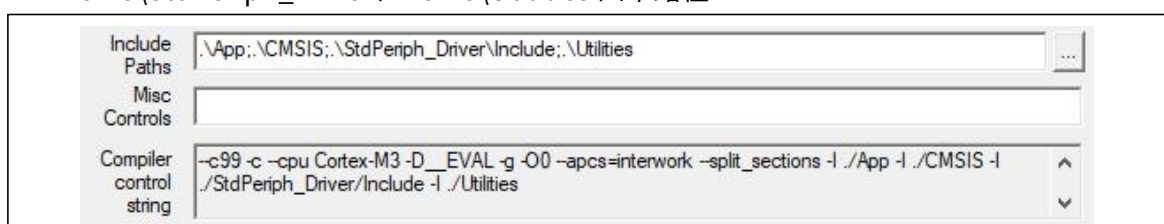
- 12) 接下来开始对工程进行配置，点击工具栏的  工具，进入 Options 窗口。在 Output 栏下把 Create HEX File 选项勾上：



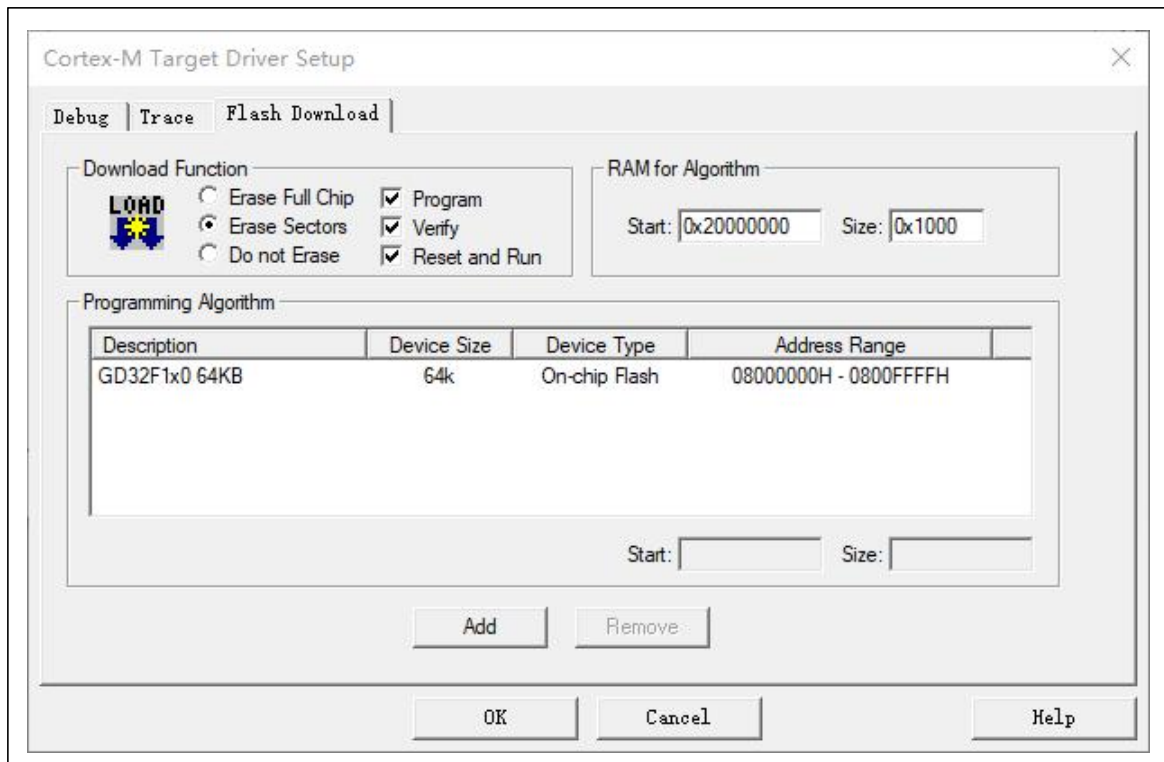
- 13) 在 C/C++ 栏下，Define: 小窗口填入 **USE_STDPERIPH_DRIVER, GD32F130_150**



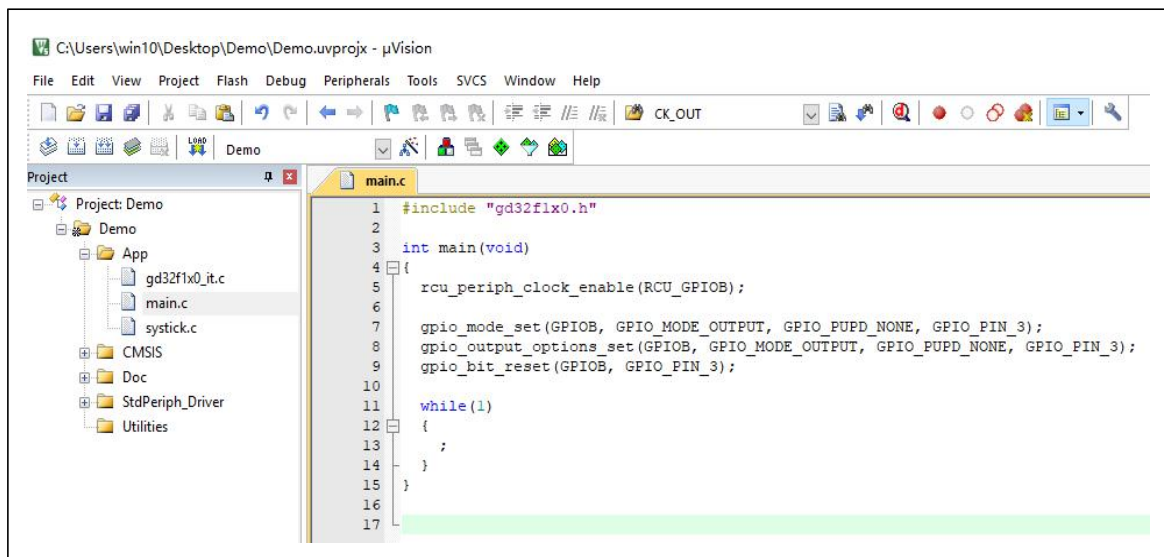
- 14) 在 C/C++ 栏下，Include Paths 小窗口分别添加 Demo\App、Demo\CMSIS、Demo\StdPeriph_Driver、Demo\Utilities 四个路径




- 15) 在 Utilities 栏下，点击 Settings 进入 Cortex-M Target Driver Setup 窗口，把 Reset and Run 选项勾上，然后点击 OK，回到 Options 窗口也点击 OK：



- 16) 回到工程，双击 APP 下的 main.c 文件，打开 main.c 文件，把文件内容修改为如下：



- 17) 点击  进行编译，发现会出现如下错误（这是因为有些电脑安装安装 MDK5.14a 以上的版本的时候会缺失一些文件导致的）：


```

.\CMSIS\core_cm3.h(147): error: #5: cannot open source input file "core_cmInstr.h": No such file or directory
#include <core_cmInstr.h> /* Core Instruction Access */
StdPeriph_Driver\Source\gd32fx0_usart.c: 0 warnings, 1 error
compiling gd32fx0_wdgt.c...
.\CMSIS\core_cm3.h(147): error: #5: cannot open source input file "core_cmInstr.h": No such file or directory
#include <core_cmInstr.h> /* Core Instruction Access */
StdPeriph_Driver\Source\gd32fx0_wdgt.c: 0 warnings, 1 error
".\Objects\Demo.axf" - 26 Error(s), 0 Warning(s).
Target not created.
Build Time Elapsed: 00:00:03

```

- 18) 从网上下载 core_cmInstr.h 、 core_cmFunc.h、 core_cm4_simd.h 三个文件（用户也可以从技新提供的例程中的 CMSIS 目录下复制），把这三个文件复制到工程目录 CMSIS 下：

Demo > CMSIS				搜索"CMSI	
名称	类型	大小			
core_cm3.h	H 文件	99 KB			
core_cm4_simd.h	H 文件	23 KB			
core_cmFunc.h	H 文件	17 KB			
core_cmInstr.h	H 文件	21 KB			
gd32fx0.h	H 文件	15 KB			
startup_gd32fx0.s	S 文件	15 KB			
system_gd32fx0.c	C 文件	11 KB			
system_gd32fx0.h	H 文件	3 KB			

- 19) 点击  从新编译，0 错误 0 警告，HEX 文件生成在 Demo\Objects 文件夹下（该文件夹是工程自动生成的），新建工程到此结束

```

compiling gd32fx0_wdgt.c...
linking...
Program Size: Code=936 RO-data=392 RW-data=4 ZI-data=2148
FromELF: creating hex file...
".\Objects\Demo.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed: 00:00:05

```


GD32F130G8 程序下载

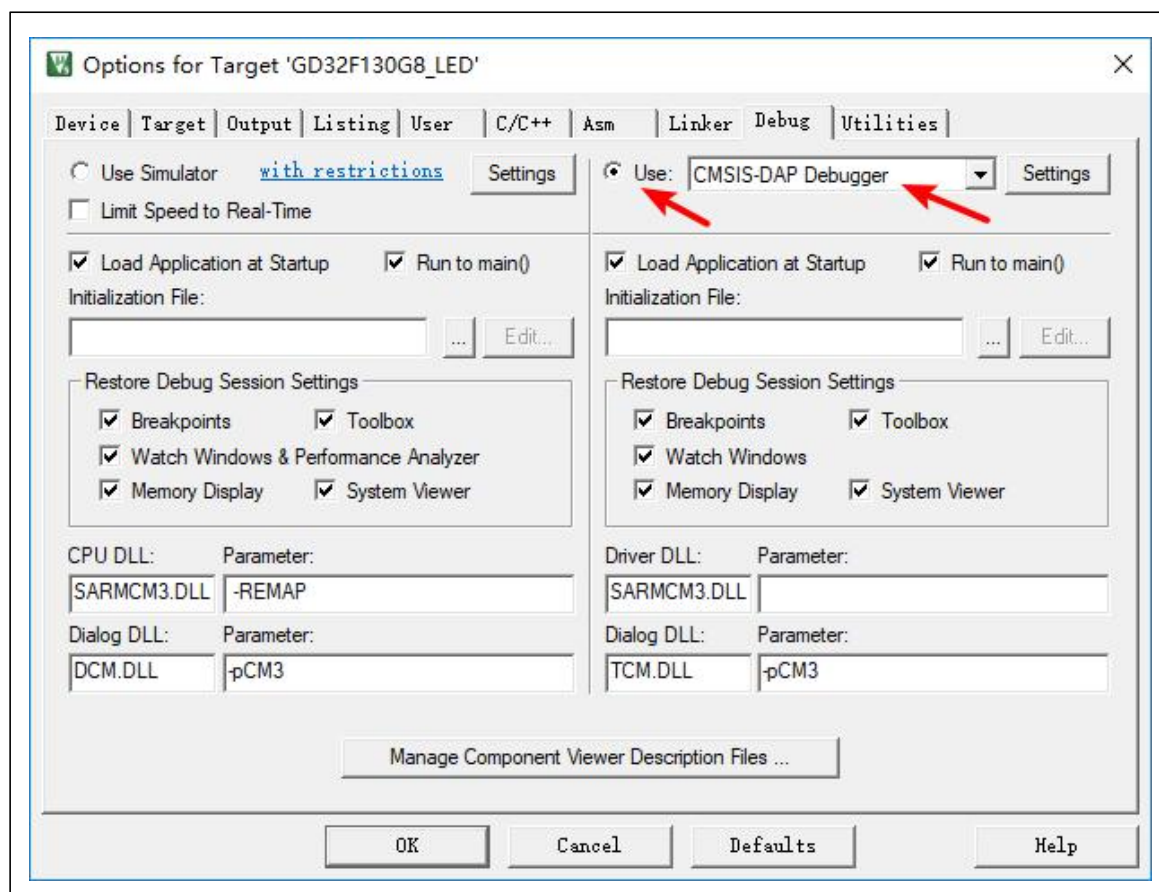
本节内容介绍三种方法下载 GD32F130G8 程序（也适用于其他 GD32 MCU）：

- CMSIS-DAP 仿真器（以 GD-LINK 为例）
- 串口

1、CMSIS-DAP 仿真器（以 GD-LINK 例）

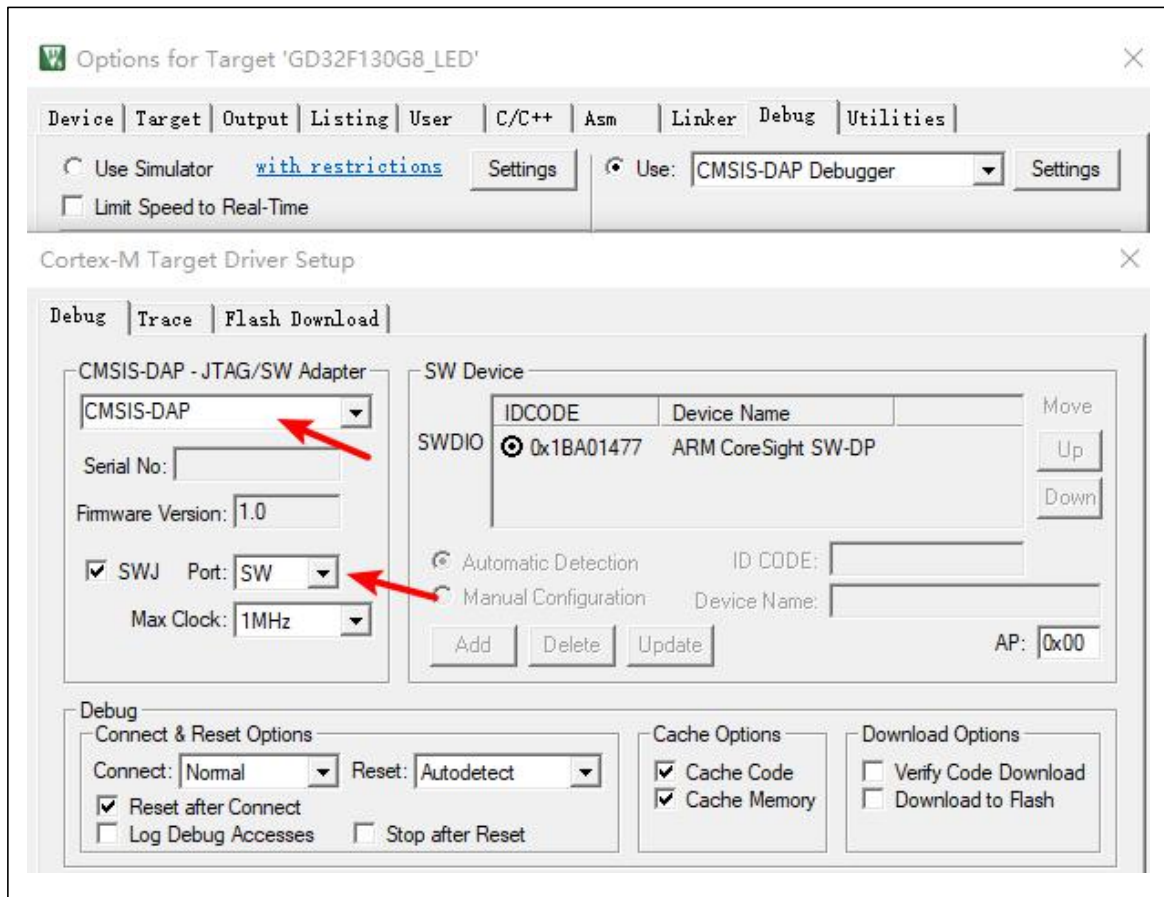
CMSIS-DAP 仿真器是开源的，支持所有的 Cortex 内核，支持在线调试和硬件仿真，属于 HID 设备，无需安装驱动，插上电脑即可使用。GD-LINK 是 GD 官方推出的一个全功能的仿真调试器和编译器，也属于 CMSIS-DAP 仿真器。下面以 GD-LINK 为例介绍如何在 MDK 环境中下载程序：

- 1) 把 GD-LINK 与 技新 GD32F130G8U6 核心板连接好：GND-->GND，3V3-->3V3，TMS/IO-->SWDIO，TCK/CK-->SWCLK，TReset-->NRST。核心板的 P1 口（BOOT0）使用跳线帽短到 0 端。
- 2) 打开 MDK 程序（以上一节建立的 Demo 工程为例），点击  图标进入 Options 选项，在

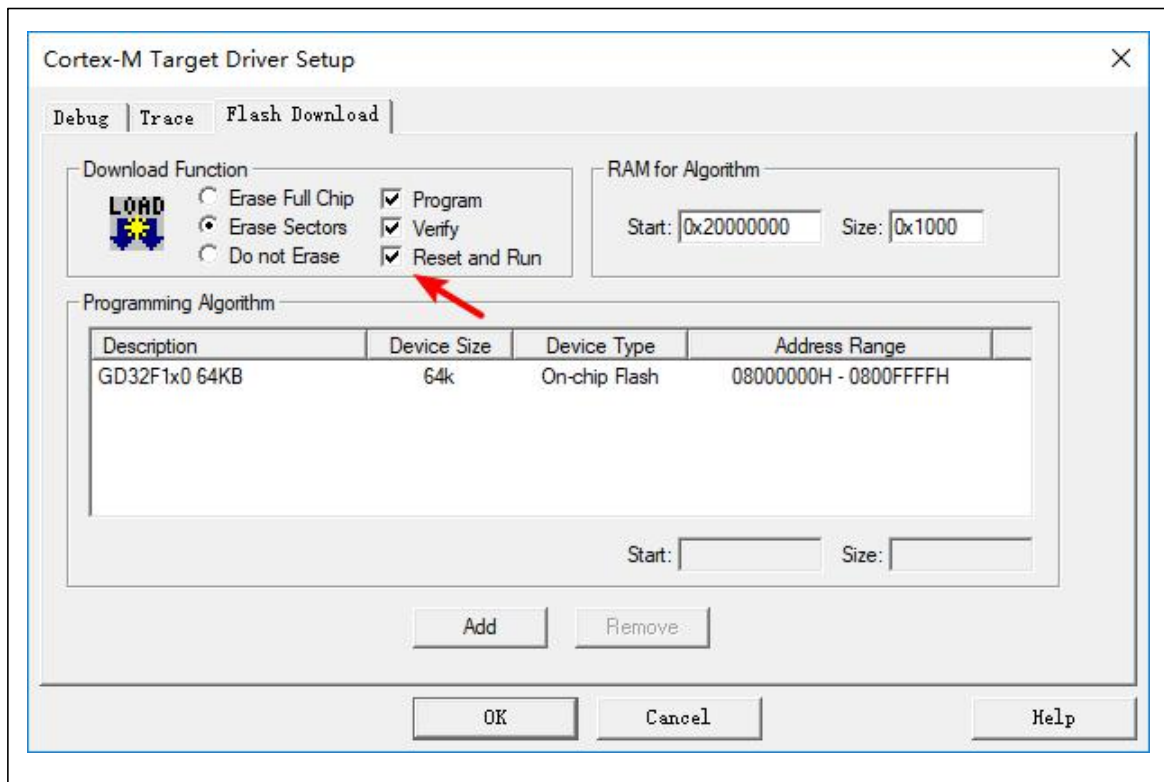




Debug 栏下选择 Use 以及 CMSIS-DAP Debugger

- 3) 点击 Settings 进入 Cortex-M Target Driver Setup 窗口，在 Debug 栏下选择 CMSIS-DAP，Port 选择 SW，如下：



4) 在 Flash Download 选项把 Reset and Run 选项勾上



点击 OK 退出 Cortex-M Target Driver Setup 窗口，再点击 OK 退出 Options 窗口，之后点击  下载按钮把程序下载到核心板中，可以看到 LED1 被点亮。点击  按钮可以进入仿真界面。

注：官方提供了 GD-LINK 的编程下载工具以及说明手册，可以实现在线编程以及脱机下载，下载地址：<http://gd32mcu.21ic.com/documents>：

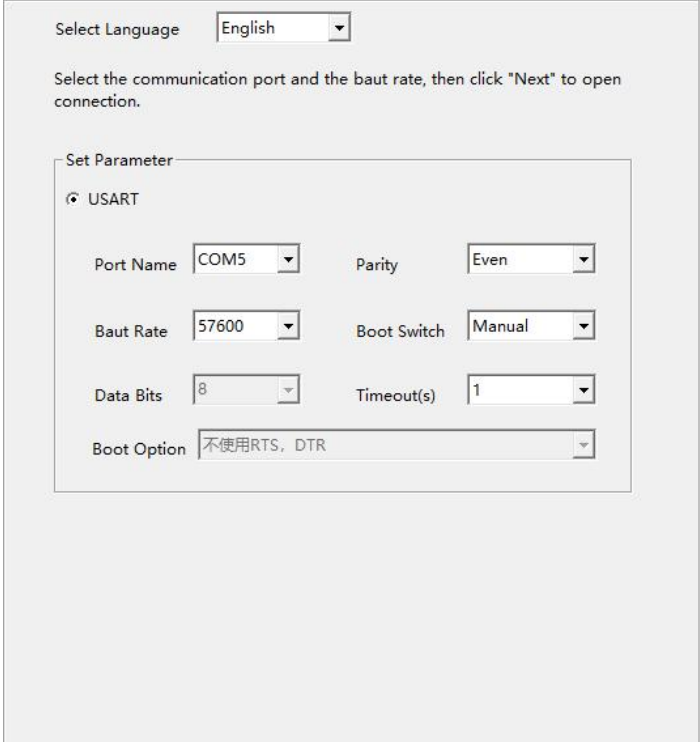
GigaDevice GD-Link Programmer	2.18.1.2542		无	2017-12-07
简介：GD-Link编程调试工具PC端软件				

2、串口

- 1) 核心板的 P1 口（BOOT0）使用跳线帽短到 1 端然后重新上电，使用一个 USB 转 TTL 模块（以技新的 USB 转 TTL CH340C 模块为例），然后连接电脑与核心板（GND-->GND，TXD-->PA10，RXD-->PA9），然后下载 GD 官方的串口下载调试工具（下载地址：<http://gd32mcu.21ic.com/documents>）：

GigaDevice MCU ISP Programmer	2.6.1.2134		无	2017-08-14
简介：串口下载调试工具				

- 2) 下载都打开工具并选择正确的串口号（不了解的可以百度一下：如何查看串口号）：



Select Language: English

Select the communication port and the baut rate, then click "Next" to open connection.

Set Parameter

USART

Port Name: COM5 Parity: Even

Baut Rate: 57600 Boot Switch: Manual


Data Bits: 8 Timeout(s): 1

Boot Option: 不使用RTS, DTR

3) 点击 Next，可以看到一些设备信息：



Flash Size	64 KB
SRAM Size	8 KB
UID	7E A3 31 42 33 31 30 02 4B 39 37 34





































4) 点击 Next，可以看到 FLASH 区域的状态：



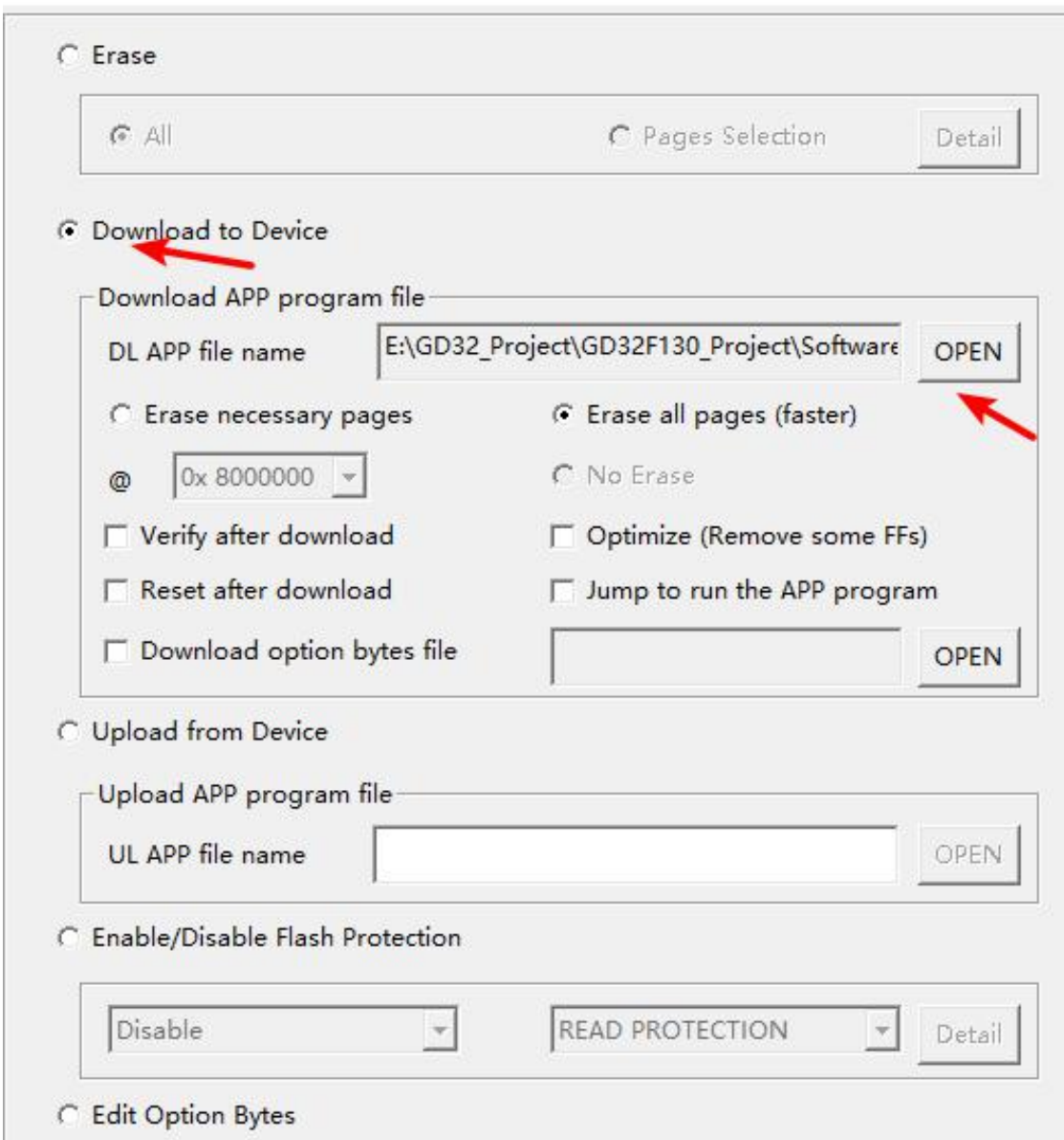
Device: GD32F130G8U6

Version: 2.2

Legend  Protected  Unprotected

Page Name	Start Address...	End Address	Page Size	R	W
Page0	0x 8000000	0x 80003FF	0x 400(1K)		
Page1	0x 8000400	0x 80007FF	0x 400(1K)		
Page2	0x 8000800	0x 8000BFF	0x 400(1K)		
Page3	0x 8000C00	0x 8000FFF	0x 400(1K)		
Page4	0x 8001000	0x 80013FF	0x 400(1K)		
Page5	0x 8001400	0x 80017FF	0x 400(1K)		
Page6	0x 8001800	0x 8001BFF	0x 400(1K)		
Page7	0x 8001C00	0x 8001FFF	0x 400(1K)		
Page8	0x 8002000	0x 80023FF	0x 400(1K)		
Page9	0x 8002400	0x 80027FF	0x 400(1K)		
Page10	0x 8002800	0x 8002BFF	0x 400(1K)		
Page11	0x 8002C00	0x 8002FFF	0x 400(1K)		
Page12	0x 8003000	0x 80033FF	0x 400(1K)		
Page13	0x 8003400	0x 80037FF	0x 400(1K)		
Page14	0x 8003800	0x 8003BFF	0x 400(1K)		
Page15	0x 8003C00	0x 8003FFF	0x 400(1K)		
Page16	0x 8004000	0x 80043FF	0x 400(1K)		
Page17	0x 8004400	0x 80047FF	0x 400(1K)		

- 5) 点击 Next，进入下载页面，选择 Download to Device，然后点击 OPEN 打开要下载的.hex 文件，MDK 编译生成的.hex 文件在 Objects 文件夹下，然后点击 Next 下载：



☐ Erase

☒ All ☐ Pages Selection

☒ Download to Device

Download APP program file

DL APP file name

☐ Erase necessary pages ☒ Erase all pages (faster)

@ ☐ No Erase

☐ Verify after download ☐ Optimize (Remove some FFs)

☐ Reset after download ☐ Jump to run the APP program

☐ Download option bytes file

☐ Upload from Device

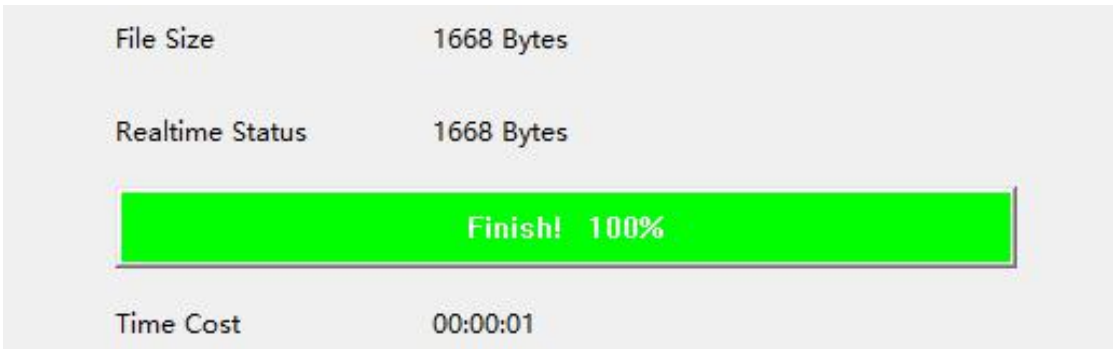
Upload APP program file

UL APP file name

☐ Enable/Disable Flash Protection

☐ Edit Option Bytes

- 6) 下载成功后如下：



File Size	1668 Bytes
Realtime Status	1668 Bytes
Finish! 100%	
Time Cost	00:00:01

- 7) 关闭软件，把 P1 口（BOOT0）使用跳线帽短到 0 端，重新上电，可以看到程序正常运行，LED1 被点亮。

第一章 GPIO 应用

GD32F130xx 系列最多可支持 55 个通用 I/O 管脚（GPIO），GDF130G8U6 有 23 个。每个 GPIO 端口有相关的控制和配置寄存器以满足特定应用的要求。本章分为以下 3 部分介绍 GPIO 相关内容：

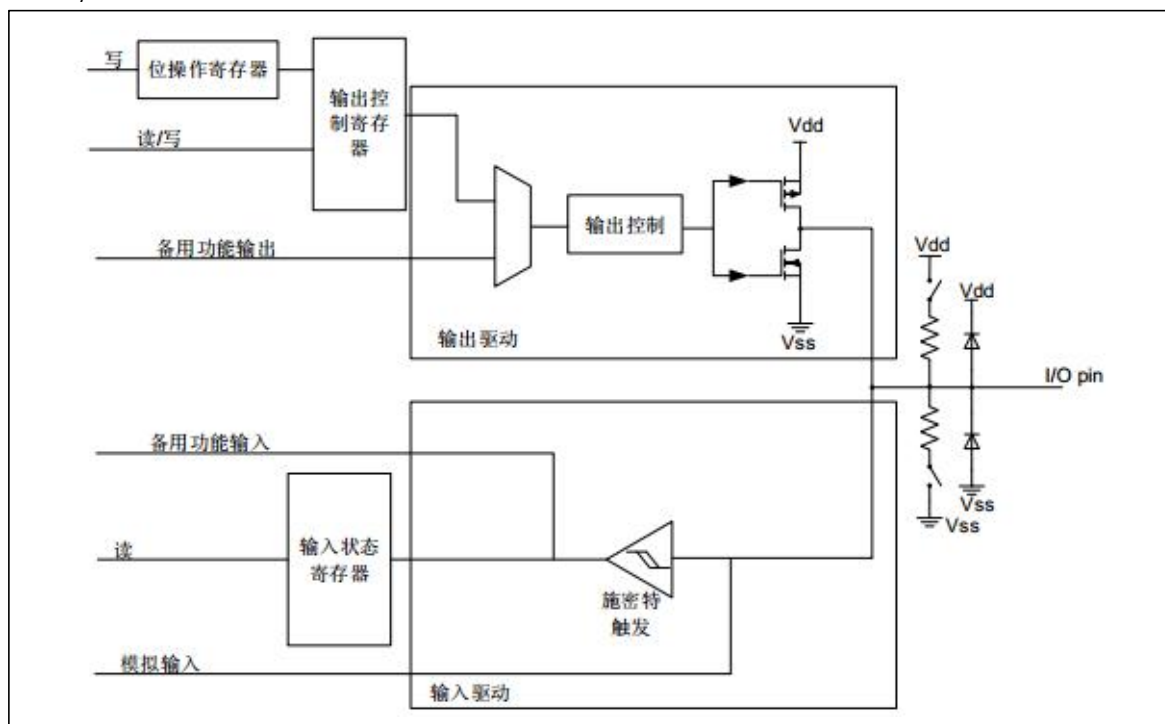
- GPIO 简介
- 点亮 LED
- KEY 按键检测

1、GPIO 简介

GD32F130G8U6 的 GPIO 端口和其他的备用功能（AFs）复用同一管脚，可以在特定的封装下获得最大的灵活性，GPIO 管脚通过配置相关的寄存器可以用作备用功能管脚。每一个 GPIO 可以软件配置为输出(推挽或开漏)，输入、外设备用功能或模拟模式。每一个 GPIO 管脚可以配置为上拉、下拉或无上拉/下拉。除了模拟模式外，所有的管脚 GPIOs 都具备大电流驱动能力。主要特征如下：

- ❖ 输入/输出方向控制
- ❖ 施密特触发器输入功能使能控制
- ❖ 每个管脚弱上拉/下拉功能
- ❖ 输出推挽/开漏使能控制
- ❖ 输出置位/复位控制
- ❖ 输出驱动速度选择
- ❖ 模拟输入/输出配置
- ❖ 备用功能输入/输出配置
- ❖ 端口配置锁定

标准 I/O 端口位的基本结构如下：



GPIO 配置表如下：

PAD TYPE			CTLn	OMn	PUDn
GPIO 输入	X	悬空	00	X	00
		上拉			01
		下拉			10
GPIO 输出	推挽	悬空	01	0	00
		上拉			01
		下拉			10
	开漏	悬空		1	00
		上拉			01
		下拉			10
AFIO 输入	X	悬空	10	X	00
		上拉			01
		下拉			10
AFIO 输出	推挽	悬空	10	0	00
		上拉			01
		下拉			10
	开漏	悬空		1	00
		上拉			01
		下拉			10
ANALOG	X	X	11	X	XX

在复位器件或复位之后，备用功能并未激活 GPIO 端口都被配置成输入悬空模式，这种输入禁用上拉(PU)/下拉(PD)电阻，但是除了 PA13（SWDIO 在输入上拉模式）和 PA14（SWCLK 在输入下拉模式）。GPIO 可以用作的功能有：输入、输出、备用、模拟输入和锁定。（不是所有 I/O 口都同时具备上述所有功能）

2、点亮 LED

2.1 GPIO 的输出配置

本小节介绍将 GPIO 配置为输出模式实现点亮 LED。使用的例程：GD32F130G8_LED，该例程位于 GD32F130G8_Examples\GD32F130G8_01_GPIO\目录下；实验板 GD32F130G8U6 核心板 V1.0 上的两个 LED 连接：LED1-->PB3，LED2-->PB4。

LED1&LED2 的 GPIO 输出模式配置步骤如下：

- 1) 使能对应的 GPIO 外设时钟
- 2) 设置 GPIO 模式为输出，以及有无上下拉
- 3) 设置输出类型，以及输出速度
- 4) 输出高/低电平

2.2 例程介绍

对应例程的 led.c 文件下的 LED_Init()函数：

```
void LED_Init(void)
{
    /* 使能 LED1 与 LED2 所在 GPIO 外设时钟 */
    rcu_periph_clock_enable(LED1_CLOCK);
    rcu_periph_clock_enable(LED2_CLOCK);

    /* 设置 LED1 与 LED2 的引脚模式为输出模式，无上下拉 */
    gpio_mode_set(LED1_PORT, GPIO_MODE_OUTPUT, GPIO_PUPD_NONE, LED1_PIN);
    gpio_mode_set(LED2_PORT, GPIO_MODE_OUTPUT, GPIO_PUPD_NONE, LED2_PIN);

    /* 设置 LED1 与 LED2 的输出类型：推挽输出；速度：最大 50MHz */
    gpio_output_options_set(LED1_PORT, GPIO_OTYPE_PP, GPIO_OSPEED_50MHZ, LED1_PIN);
    gpio_output_options_set(LED2_PORT, GPIO_OTYPE_PP, GPIO_OSPEED_50MHZ, LED2_PIN);

    /* 设置 LED1 与 LED2 的输出输出低（关闭 LED1&LED2） */
    gpio_bit_set(LED1_PORT, LED1_PIN);
    gpio_bit_set(LED2_PORT, LED2_PIN);
}
```

LED1&LED2 的 GPIO 端口、GPIO 时钟、GPIO 引脚以及相应的功能定义在 led.h 文件下：

```
/* 宏定义 -----*/
/* 定义 LED IO PORT 与 PIN */
#define LED1_CLOCK      RCU_GPIOB
#define LED1_PORT        GPIOB
#define LED1_PIN          GPIO_PIN_3

#define LED2_CLOCK      RCU_GPIOB
#define LED2_PORT        GPIOB
#define LED2_PIN          GPIO_PIN_4

/* 定义 LED 功能 */
#define LED1_ON()        gpio_bit_reset(LED1_PORT, LED1_PIN)
#define LED2_ON()        gpio_bit_reset(LED2_PORT, LED2_PIN)
#define LED1_OFF()       gpio_bit_set(LED1_PORT, LED1_PIN)
#define LED2_OFF()       gpio_bit_set(LED2_PORT, LED2_PIN)
#define LED1_Toggle()    (GPIO_OCTL(LED1_PORT) ^= LED1_PIN)
#define LED2_Toggle()    (GPIO_OCTL(LED2_PORT) ^= LED2_PIN)
```

主函数如下：

```
int main(void)
{
    systick_config(); //初始化滴答定时器
    LED_Init(); //LED 初始化
    while(1)
    {
        LED1_Toggle(); //对 LED1 的状态进行取反
        delay_1ms(500);
    }
}
```

主函数中使用到 systick_config()函数，该函数在 systick.c 与 systick.h 文件中实现以及定义，作用是使用 systick 滴答定时器实现 1ms 的延迟（使用中断方式，中断函数位于 gd32f1x0_it.c 文件下）。把例程下载到核心板后可以看到 LED1 闪烁。

3、KEY 按键检测

3.1 GPIO 的输入配置

本小节介绍将 GPIO 配置为输出模式实现点亮 LED。使用的例程：GD32F130G8_KEY，该例程位于 GD32F130G8_Examples\GD32F130G8_01_GPIO\目录下；实验板 GD32F130G8U6 核心板 V1.0 上的 KEY 连接：KEY-->PA15。

KEY 的输入配置步骤如下：

- 1) 使能对应的 GPIO 时钟
- 2) 配置 GPIO 模式为输入，带上拉电阻

3.2 例程介绍

KEY 的初始化在 key.c 文件中 KEY_Init()函数中实现：

```
void KEY_Init(void)
{
    /* 使能 KEY 所在 GPIO 外设时钟 */
    rcu_periph_clock_enable(KEY_CLOCK);
    /* 设置 KEY 的引脚模式为输入模式，带上拉电阻 */
    gpio_mode_set(KEY_PORT, GPIO_MODE_INPUT, GPIO_PUPD_PULLUP, KEY_PIN);
}
```

使用 KEY 按键功能时需要对按键进行扫描，按键扫描函数如下：


```
uint8_t KEY_Scan(void)
{
    if(KEY_READ == 0)
    {
        delay_1ms(10); //消抖
        if(KEY_READ == 0)
        {
            while(KEY_READ == 0); //松手检测
            return KEY_PRES;
        }
    }
    return 0;
}
```

上面的 KEY_Init() 与 KEY_Scan() 函数中使用的 KEY_CLOCK、KEY_READ 等变量在 key.h 文件中定义：

```
/* 定义 KEY IO PORT 与 PIN */
#define KEY_CLOCK          RCU_GPIOA
#define KEY_PORT            GPIOA
#define KEY_PIN             GPIO_PIN_15
/* 定义 KEY 按下时的返回值 */
#define KEY_PRES            0x01
/* 读取 KEY 值 */
#define KEY_READ            gpio_input_bit_get(KEY_PORT, KEY_PIN)
```

主函数如下：

```
int main(void)
{
    uint8_t temp; //定义变量用于读取按键值
    systick_config(); //初始化滴答定时器
    LED_Init(); //LED 初始化
    KEY_Init(); //KEY 初始化
    while(1)
    {
        temp = KEY_Scan(); //读取按键值
        if(temp == KEY_PRES) //如果按键按下
            LED1_Toggle(); //切换 LED1 状态
    }
}
```

把程序下载到核心板后，按一下 KEY，LED1 状态改变一次。

第二章 EXTI 应用

EXTI（中断和事件控制器），Cortex-M3 集成了嵌套式矢量型中断控制器(Nested Vectored Interrupt Controller (NVIC))来实现高效的异常和中断处理。NVIC 实现了低延迟的异常和中断处理，以及电源管理控制。它和内核是紧密耦合的。本章分为以下 2 部分介绍 EXTI 相关内容：

- EXTI 简介
- KEY 外部中断检测

1、EXTI 简介

GD32F130xx 提供了一个外部中断/事件控制器(EXTI)，包括 23 个相互独立的边沿检测电路并且能够向处理器内核产生中断请求或唤醒事件。EXTI 有三种触发类型：上升沿触发、下降沿触发和双边沿触发。EXTI 中的每一个边沿检测电路都可以独立配置和屏蔽。EXTI 主要特征如下：


- ❖ Cortex-M3 系统异常
- ❖ 52 种可屏蔽的外设中断
- ❖ 4 位中断优先级配置位—共提供 16 个中断优先等级
- ❖ 高效的中断处理
- ❖ 支持异常抢占和咬尾中断
- ❖ 将系统从省电模式唤醒
- ❖ 外中断中有 23 个相互独立的边沿检测电路
- ❖ 3 种触发类型：上升沿触发，下降沿触发和双边沿触发
- ❖ 软件中断或事件触发
- ❖ 可配置的触发源

ARM Cortex-M3 处理器和嵌套式矢量型中断控制器(NVIC)在处理(Handler)模式下对所有异常进行优先级区分以及处理。当异常发生时，系统自动将当前处理器工作状态压栈，在执行完中断服务子程序(ISR)后自动将其出栈。

取向量是和当前工作状态压栈并行进行的，从而提高了中断入口效率。处理器支持咬尾中断，可实现背靠背中断，大大削减了反复切换工作状态所带来的开销。Cortex-M3 中的 NVIC 异常类型如下：

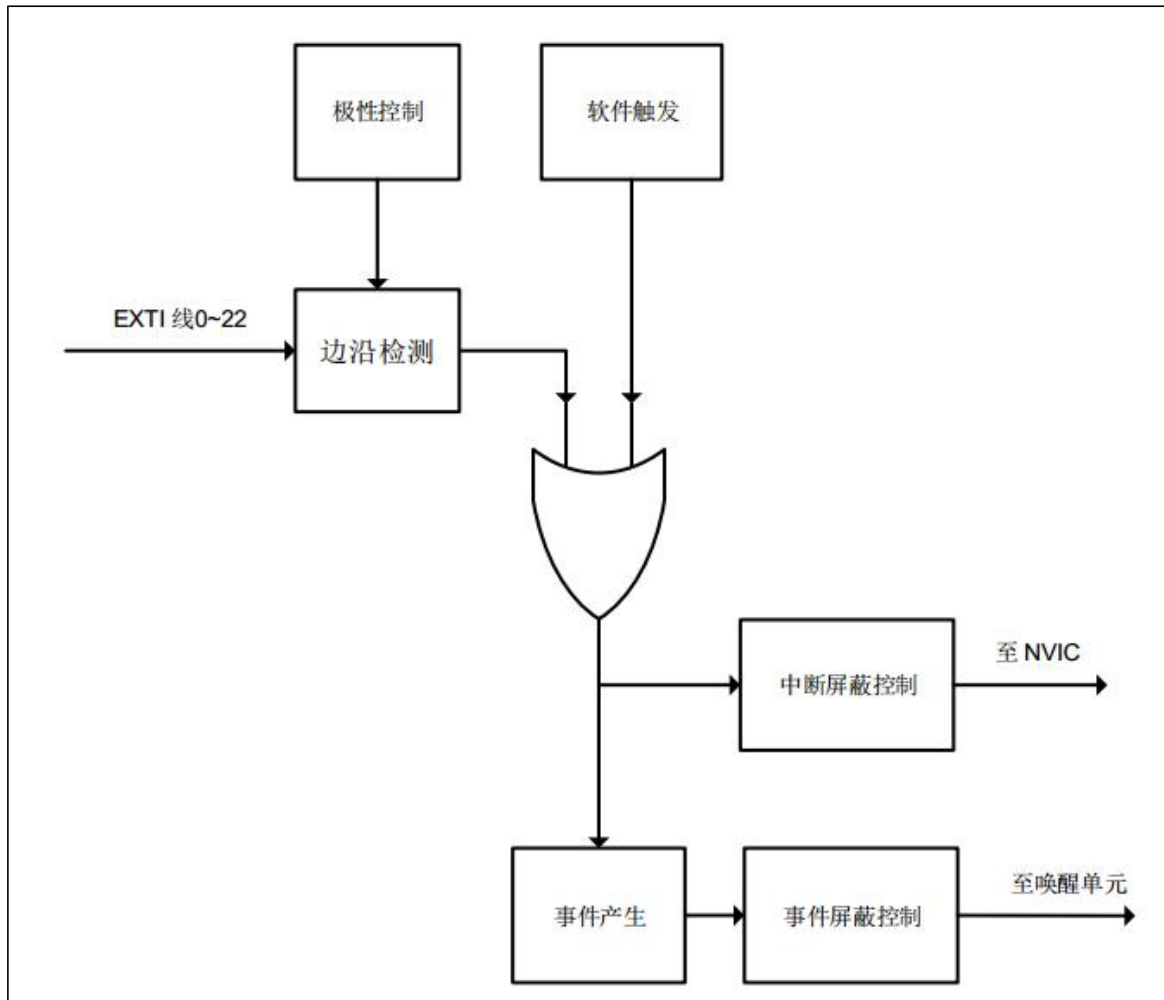
异常类型	向量编号	优先级(a)	向量地址	描述
-	0	-	0x0000_0000	保留
复位	1	-3	0x0000_0004	复位
NMI	2	-2	0x0000_0008	不可屏蔽中断
硬件故障	3	-1	0x0000_000C	各种硬件级别的故障
存储器管理	4	可编程设置	0x0000_0010	存储器管理
总线故障	5	可编程设置	0x0000_0014	预取指故障, 存储器访问故障
用法故障	6	可编程设置	0x0000_0018	未定义的指令或非法状态
-	7-10	-	0x0000_001C - 0x0000_002B	保留
服务调用	11	可编程设置	0x0000_002C	通过 SWI 指令实现系统的服务调用
调试监控	12	可编程设置	0x0000_0030	调试监控器
-	13	-	0x0000_0034	保留
挂起服务	14	可编程设置	0x0000_0038	可挂起的系统服务请求
系统节拍	15	可编程设置	0x0000_003C	系统节拍定时器
中断	16-89	可编程设置	0x0000_0040 - 0x0000_0164	外设中断 (详见下表)

本章主要介绍外部中断的应用，GD32F130xx 中的每一个 I/O 口都可以配置为外部中断模式，并映射在三个中断向量中：

中断编号	向量编号	外设中断描述	向量地址
IRQ 0	16	窗口式看门狗定时器中断	0x0000_0040
IRQ 1	17	通过 EXTI 线检测的 LVD 中断	0x0000_0044
IRQ 2	18	RTC 全局中断	0x0000_0048
IRQ 3	19	Flash 全局中断	0x0000_004C
IRQ 4	20	RCU 全局中断	0x0000_0050
IRQ 5	21	EXTI 线 0-1 中断 	0x0000_0054
IRQ 6	22	EXTI 线 2-3 中断 	0x0000_0058
IRQ 7	23	EXTI 线 4-15 中断 	0x0000_005C
IRQ 8	24	TSI 全局中断	0x0000_0060

GD32F130xx 的 EXTI 触发源包括来自 I/O 管脚的 16 根线以及来自内部模块的 6 根线。包括 LVD、RTC、USB、USART、CEC 和 CMP。除了中断，EXTI 还可以向处理器提供事件信号。Cortex-M3 内核完全支持等待中断(WFI)，等待事件(WFE)和发送事件(SEV)指令。芯片内部有一个唤醒中断控制器(WIC)，用户可以放心的让处理器和 NVIC 进入功耗极低的休眠模式，由 WIC 来识别中断和事件以及判断

优先级。当某些预期的事件发生时，EXTI 能唤醒处理器及整个系统，例如一个特定的 I/O 管脚电平翻转或者 RTC 闹钟动作。EXTI 的框图如下：



EXTI 可以配置为硬件触发或软件触发，通常使用的 KEY 外部中断检测就是硬件触发，EXTI 的触发源如下：

EXTI 线编号	触发源	归属
0	PA0 / PB0 / PC0 / PF0	外部
1	PA1 / PB1 / PC1 / PF1	外部
2	PA2 / PB2 / PC2 / PD2	外部
3	PA3 / PB3 / PC3	外部
4	PA4 / PB4 / PC4 / PF4	外部
5	PA5 / PB5 / PC5 / PF5	外部
6	PA6 / PB6 / PC6 / PF6	外部
7	PA7 / PB7 / PC7 / PF7	外部
8	PA8 / PB8 / PC8	外部
9	PA9 / PB9 / PC9	外部
10	PA10 / PB10 / PC10	外部

EXTI 线编号	触发源	归属
11	PA11 / PB11 / PC11	外部
12	PA12 / PB12 / PC12	外部
13	PA13 / PB13 / PC13	外部
14	PA14 / PB14 / PC14	外部
15	PA15 / PB15 / PC15	外部
16	LVD	外部
17	RTC 闹钟	外部
18	USBD 唤醒	外部
19	RTC 干预和时间戳	外部
20	保留	保留
21	比较器 0 输出	外部
22	比较器 1 输出	外部
23	保留	保留
24	保留	保留
25	USART0 唤醒	内部
26	保留	保留
27	CEC 唤醒	内部

2、KEY 外部中断检测

2.1 KEY 的外部中断配置

本小节介绍将 GPIO 配置为外部中断输入模式，并通过 KEY 来产生外部中断从而控制 LED1 状态。使用的例程：GD32F130G8_EXTI，该例程位于 GD32F130G8_Examples\GD32F130G8_02_EXTI\目录下。

KEY 的外部中断配置步骤如下：

- 1) 使能 KEY 对应的 GPIO 时钟，使能 CFGCMP 时钟
- 2) 设置 KEY 引脚为输入模式，带上拉电阻
- 3) 设置 KEY 外部中断优先级
- 4) 配置 KEY 的外部中断线
- 5) 初始化外部中断线，使能中断模式，下降沿触发
- 6) 清除中断标志位

2.2 例程介绍

EXTI 的初始化在 exti.c 文件中实现：

```
void EXTI_Init(void)
{
    /* 使能 KEY 所在 GPIO 外设时钟与 CFGCMP 时钟 */
```



```
rcu_periph_clock_enable(KEY_CLOCK);
rcu_periph_clock_enable(RCU_CFGCMP);
/* 设置 KEY 的引脚模式为输入模式，带上拉电阻 */
gpio_mode_set(KEY_PORT, GPIO_MODE_INPUT, GPIO_PUPD_PULLUP, KEY_PIN);
/* 设置 KEY 外部中断的优先级 */
nvic_irq_enable(KEY_EXTI_IRQn, 2U, 0U);
/* 配置 KEY 的外部中断线 */
syscfg_exti_line_config(KEY_EXTI_PORT_SOURCE, KEY_EXTI_PIN_SOURCE);
/* 初始化 KEY 外部中断线，中断模式，下降沿触发 */
exti_init(KEY_EXTI_LINE, EXTI_INTERRUPT, EXTI_TRIG_FALLING);
/* 清除中断标志位 */
exti_interrupt_flag_clear(KEY_EXTI_LINE);
}
```

相关的定义在 exti.h 文件中：

```
/* 定义 KEY IO CLOCK、PORT 与 PIN */
#define KEY_CLOCK          RCU_GPIOA
#define KEY_PORT           GPIOA
#define KEY_PIN            GPIO_PIN_15

/* 定义 KEY 与外部中断相关内容 */
#define KEY_EXTI_LINE      EXTI_15
#define KEY_EXTI_PORT_SOURCE EXTI_SOURCE_GPIOA
#define KEY_EXTI_PIN_SOURCE EXTI_SOURCE_PIN15
#define KEY_EXTI_IRQn     EXTI4_15_IRQn
```

主函数如下：

```
int main(void)
{
    systick_config(); //初始化滴答定时器
    LED_Init(); //LED 初始化
    EXTI_Init(); //EXTI 初始化
    while(1)
    {
        ;
    }
}
```

其中对 LED1 状态进行切换的功能在中断函数中实现，中断函数位于 gd32f1x0_it.c 文件下：

```
void EXTI4_15_IRQHandler(void)
```

```
{
    if (RESET != exti_interrupt_flag_get(KEY_EXTI_LINE)) //读取中断标志，判断是否 KEY 引起的中断
    {
        LED1_Toggle(); //切换 LED1 状态
    }
    exti_interrupt_flag_clear(KEY_EXTI_LINE); //清除中断标志位
}
```

把程序下载到核心板后，按一下 KEY，LED1 状态改变一次。

注：使用 KEY 会有抖动问题，EXTI 例程没有对 KEY 进行消抖处理。

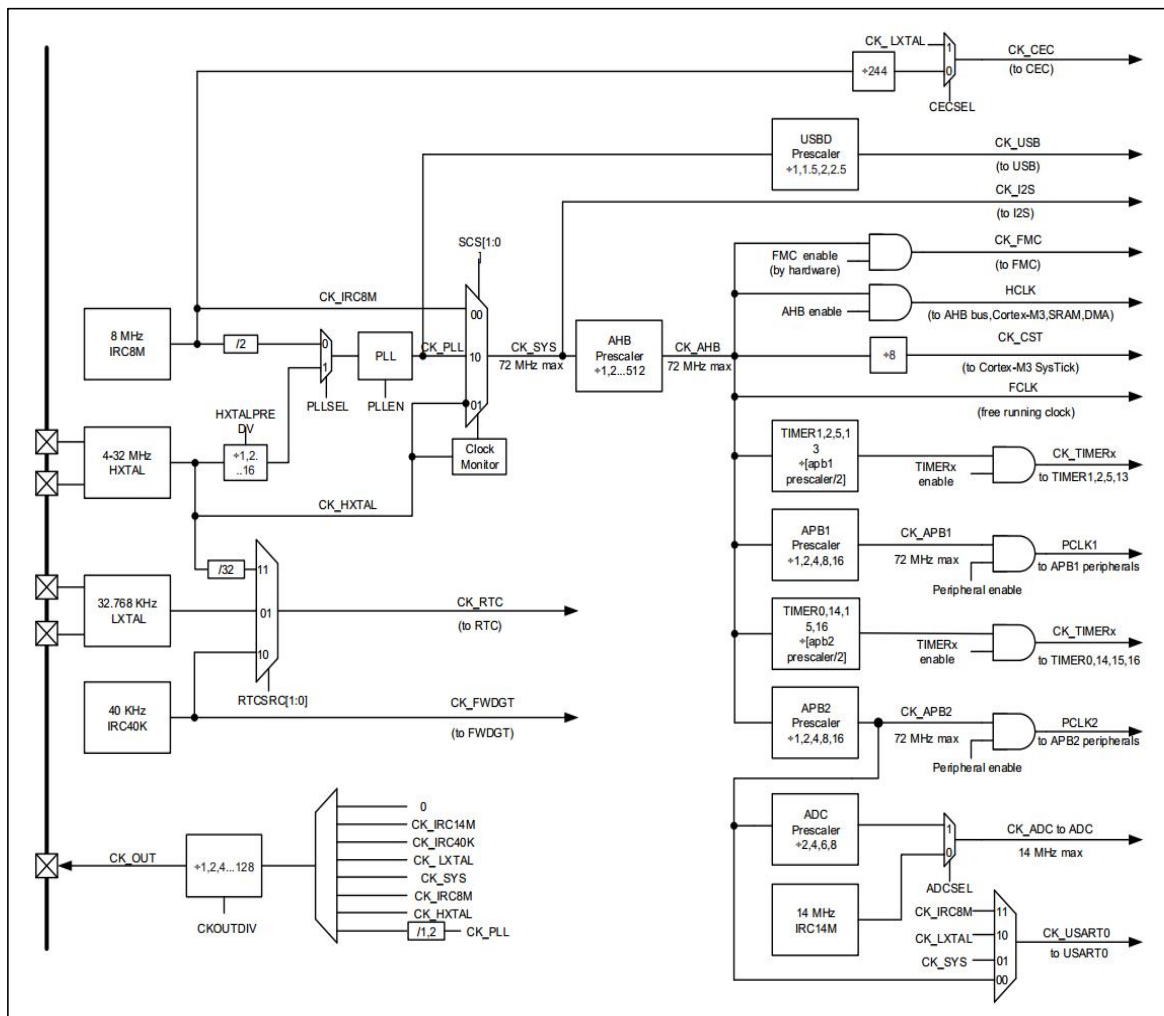
第三章 CCTL 应用

CCTL（时钟控制单元），时钟控制单元提供了一系列频率的时钟功能，包括一个内部 8M RC 振荡器时钟(IRC8M)、一个内部高速 14M RC 振荡器时钟(IRC14M)、一个外部高速晶体振荡器时钟(HXTAL)、一个内部低速 RC 振荡器时钟(IRC40K)、一个外部低速晶体振荡器时钟(LXTAL)、一个锁相环(PLL)、一个 HXTAL 时钟监视器、时钟预分频器、时钟多路复用器和时钟选通电路。本章分为以下 2 部分介绍 CCTL 相关内容：

- CCTL 简介
- 时钟输出

1、CCTL 简介

AHB、APB 和 Cortex™-M3 时钟都源自系统时钟(CK_SYS)，系统时钟的时钟源为 IRC8M、HXTAL 或 PLL。系统时钟的最大运行时钟频率可以达到 72MHz。独立看门狗定时器有独立的时钟源（IRC40K），实时时钟(RTC)使用 IRC40K、LXTAL 或 HXTAL/32 作为时钟源。GD32F130xx 系列产品的时钟树如下：



预分频器可以配置 AHB、APB2 和 APB1 域的时钟频率。AHB 和 APB2/APB1 域的最高时钟频率为 72MHz。但当使用 I2C 外设时，APB1 时钟需保证不大于 36MHz。

GD32F130xx 的 CCTL 主要特征如下：

- ❖ 4 到 32 MHz 外部高速晶体振荡器 (HXTAL)
- ❖ 8 MHz 内部高速 RC 振荡器 (IRC8M)
- ❖ 14 MHz 内部高速 RC 振荡器 (IRC14M)
- ❖ 32.768 Hz 外部低速晶体振荡器 (LXTAL)
- ❖ 40 kHz 内部低速 RC 振荡器 (IRC40K)
- ❖ PLL 时钟源可以是 HXTAL 或 IRC8M
- ❖ HXTAL 时钟可监视

系统复位后，IRC8M 时钟被选为系统时钟，改变时钟配置寄存器 RCU_CFG0 中的系统时钟变换位 SCS 可以切换系统时钟源为 HXTAL 或 PLL。当 SCS 的值改变，系统时钟将使用原来的时钟源继续运行直到转换的目标时钟源稳定。当一个时钟源被直接或通过 PLL 间接作为系统时钟时，它将不能被停止。使用官方固件库进行开发时，默认使用 HXTAL 时钟（一般外接 8MHz 晶振），系统频率为 72MHz。

2、时钟输出

2.1 系统时钟输出配置

本小节介绍如何配置在 CK_OUT 口输出指定时钟信号，并通过示波器来检测时钟信号。使用的例程：CK_OUT，该例程位于 GD32F130G8_Examples\GD32F130G8_03_CCTL\目录下。

时钟输出的配置步骤如下：

- 1) 使能 CK_OUT 引脚所在 GPIO 外设时钟
- 2) 配置 CK_OUT 引脚用作复用模式
- 3) 配置 CK_OUT 引脚为推挽输出
- 4) 配置 CK_OUT 引脚用作复用功能
- 5) 配置 CK_OUT 引脚输出指定时钟源与分配系数

2.2 例程介绍

CK_OUT 的配置在 ck_out.c 文件中实现：

```
void CK_OUT_Config(uint32_t ckout_src, uint32_t ckout_div)
{
    /* 使能 CK_OUT 引脚所在的 GPIO 外设时钟 */
    rcu_periph_clock_enable(CK_OUT_CLOCK);
    /* 配置 PA8(CK_OUT) 用作复模式 */
    gpio_mode_set(CK_OUT_PORT, GPIO_MODE_AF, GPIO_PUPD_NONE, CK_OUT_PIN);
    /* 配置 PA8(CK_OUT) 为推挽输出 */
    gpio_output_options_set(CK_OUT_PORT, GPIO_OTYPE_PP,
        GPIO_OSPEED_50MHZ, CK_OUT_PIN);
}
```

```
/* 配置 PA8(CK_OUT) 用作复用功能 */
gpio_af_set(CK_OUT_PORT, GPIO_AF_0, CK_OUT_PIN);
/* 配置 PA8(CK_OUT) 输出时钟源与分配系数 */
rcu_ckout_config(ckout_src, ckout_div);
}
```

CK_OUT 引脚的定义在 ck_out.h 文件中：

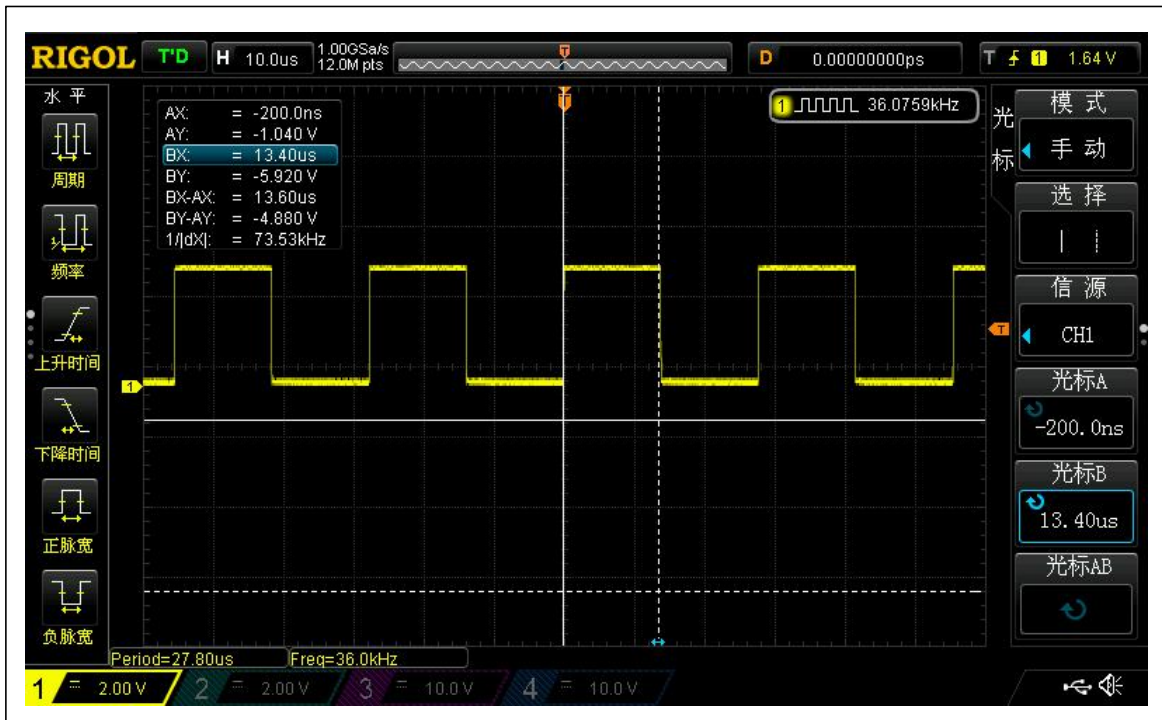
```
/* 定义 CK_OUT IO PORT 与 PIN */
#define CK_OUT_CLOCK          RCU_GPIOA
#define CK_OUT_PORT           GPIOA
#define CK_OUT_PIN             GPIO_PIN_8
```

主函数中输出的是内部 40KHz 时钟，如下：

```
int main(void)
{
    rcu_osci_on(RCU_IRC40K); //打开内部 40KHz 时钟
    while(rcu_osci_stab_wait(RCU_IRC40K) == ERROR); //等待时钟稳定
    CK_OUT_Config(RCU_CKOUTSRC_IRC40K, RCU_CKOUT_DIV1); //输出内部 40KHz、1 分频时钟
    systick_config(); //初始化滴答定时器
    LED_Init(); //LED 初始化
    while(1)
    {
        LED1_Toggle(); //改变 LED1 状态
        delay_1ms(500);
    }
}
```

把程序下载到核心板后，LED1 闪烁，PA8（CK_OUT）输出内部 40KHz 时钟频率。为了更直观的了解系统时钟，下面是使用示波器检测 GD32F130G8U6 输出不同的时钟信号。

内部 40KHz 时钟：在使用需要打开时钟并等待稳定，如例程中的主函数所示，然后再输出时钟信号：CK_OUT_Config(RCU_CKOUTSRC_IRC40K, RCU_CKOUT_DIV1)，实测 36.0579kHz，如下图：



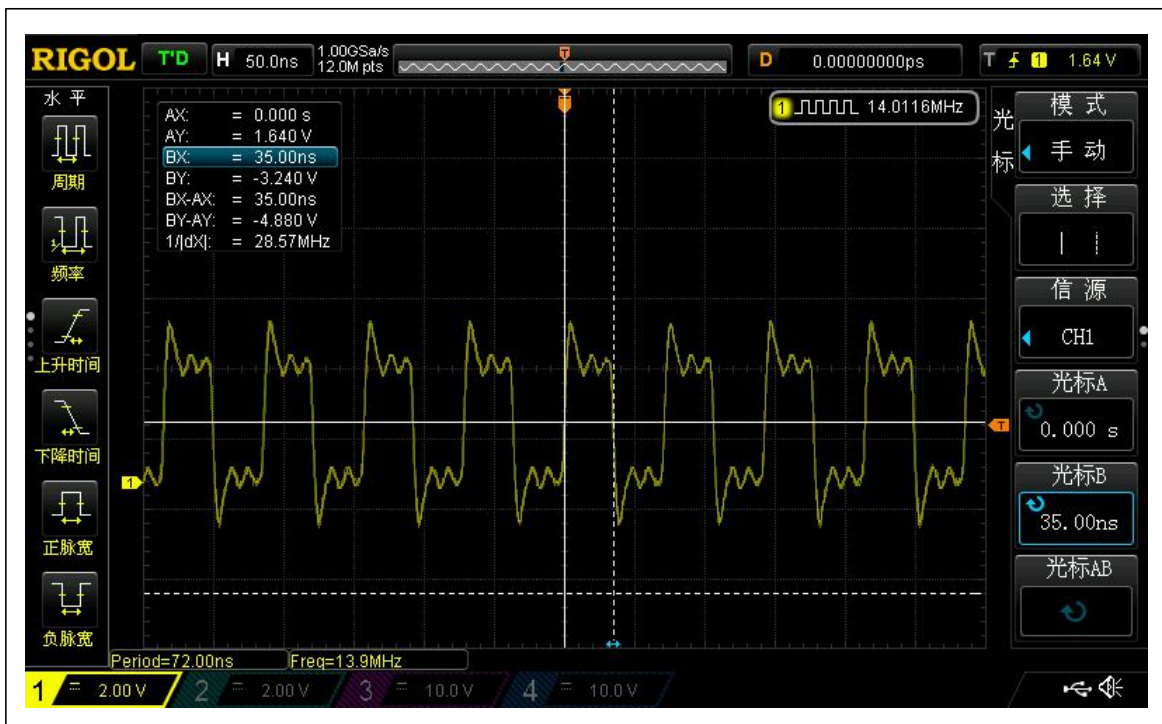
系统时钟内部 14MHz 时钟（可用作 ADC 时钟），需要先打开时钟并等待稳定：

```
rcu_osci_on(RCU_IRC14M); //打开内部 14MHz 时钟
```

```
while(rcu_osci_stab_wait(RCU_IRC14M) == ERROR); //等待时钟稳定
```

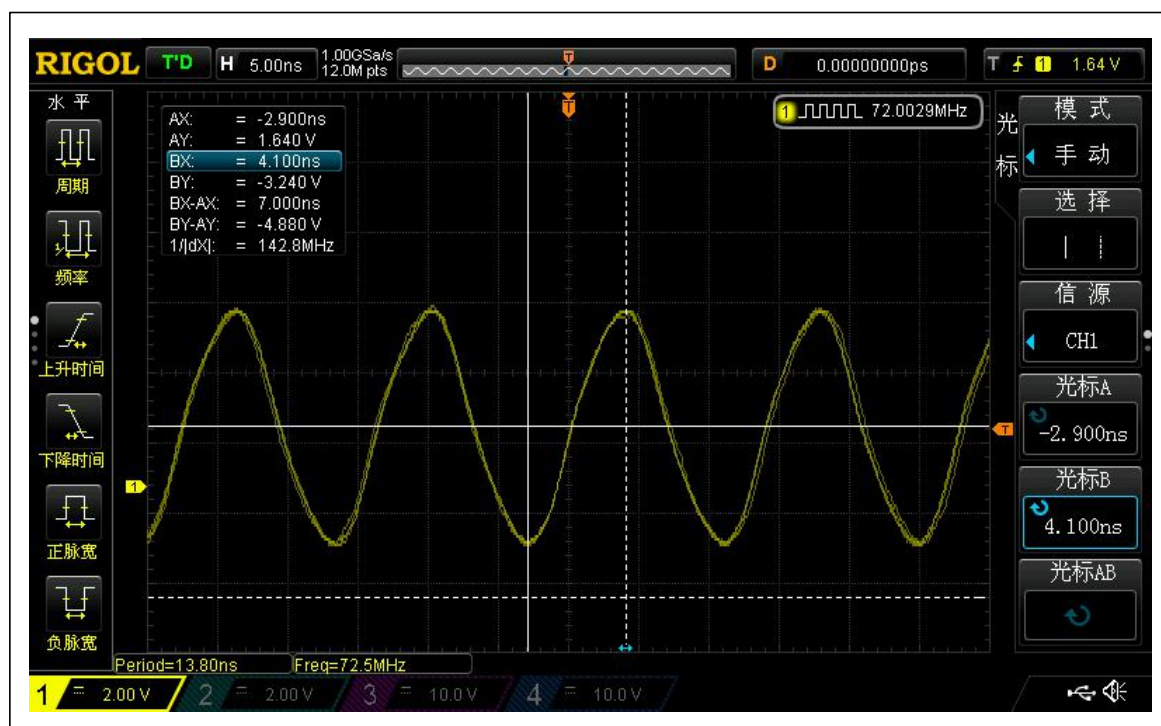
```
CK_OUT_Config(RCU_CKOUTSRC_IRC14M, RCU_CKOUT_DIV1); //输出内部 14MHz、1 分频时钟
```

实测 14.0116MHz，如下图：



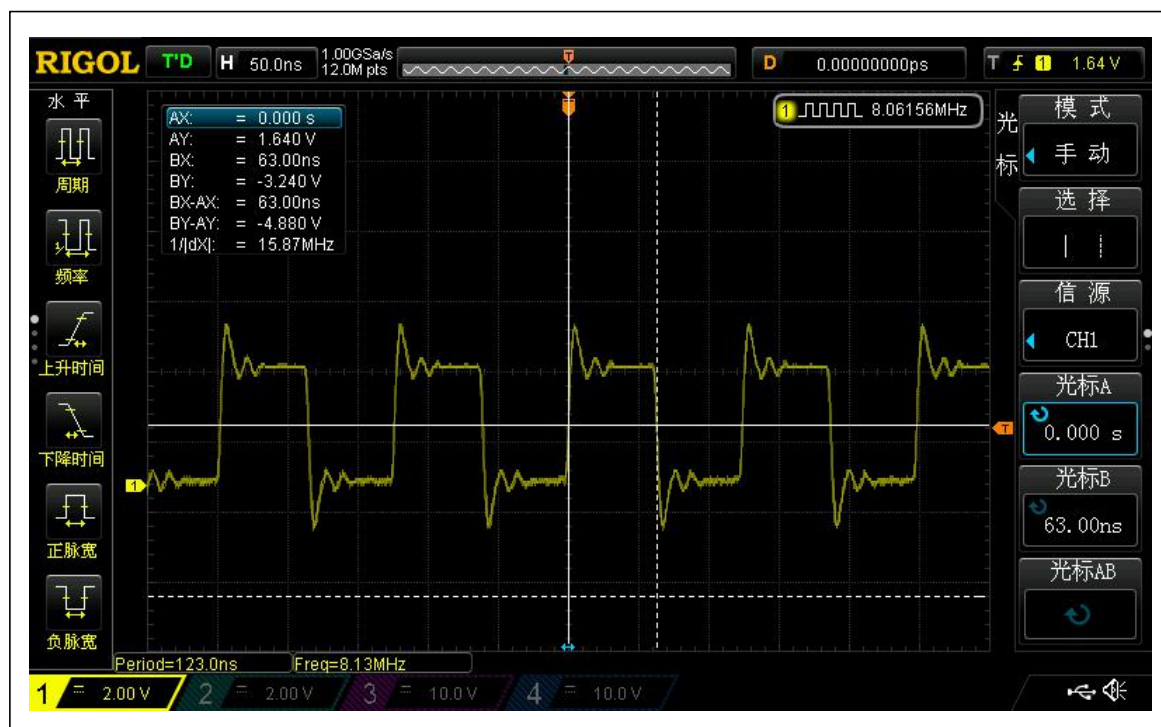
系统时钟（外部 8M 晶振倍频）：CK_OUT_Config(RCU_CKOUTSRC_CKSYS, RCU_CKOUT_DIV1)

实测 72.0029MHz，如下图：



内部8M时钟：CK_OUT_Config(RCU_CKOUTSRC_IRC8M, RCU_CKOUT_DIV1), 实测8.06156MHz

如下图：



第四章 USART 应用

USART（通用同步异步收发器），通用同步异步收发器(USART)提供了一个灵活方便的串行数据交换接口。数据帧可以通过全双工或半双工，同步或异步的方式进行传输。USART 提供了可编程的波特率发生器，能对系统时钟进行分频产生 USART 发送和接收所需的特定频率。本章从以下部分介绍 USART 相关内容：

- USART 简介
- printf 输出
- 串口收发中断
- 串口 DMA 传输

1、USART 简介

本章内容主要介绍使用 USART 用作串口功能的使用，USART 可以实现的功能还有许多，如红外编码规范，SIR，智能卡协议，LIN，以及同步单双工模式，还支持多处理器通信和 Modem 流控操作 (CTS/RTS)。USART 的数据帧支持从 LSB 或者 MSB 开始传输，数据位的极性和 TX/RX 引脚都可以灵活配置，并且都支持 DMA 功能来实现高速率的数据通信。UASRT 的主要特性如下：

- ❖ NRZ 标准格式(Mark/Space)
- ❖ 全双工异步通信
- ❖ 半双工单线通信
- ❖ 双时钟域：
 - 互为异步关系的 APB 时钟和 USART 时钟
 - 不依赖于 PCLK 设置的波特率设置
- ❖ 可编程的波特率产生器，当时钟频率为 72MHz，过采样为 8 时，最高速度可达 9 MBits/s
- ❖ 完全可编程的串口特性：
 - 数据位(8 或 9 位)低位或高位在前
 - 偶校验位，奇校验位，无校验位的生成/检测
 - 产生 1，1.5 或者 2 个停止位
- ❖ 可互换的 Tx/Rx 引脚
- ❖ 可配置的数据极性
- ❖ 自动检测波特率
- ❖ 支持硬件 Modem 流控操作(CTS/RTS)和 RS485 驱动使能
- ❖ 借助集中式 DMA，可实现可配置的多级缓存通信
- ❖ 发送器和接收器可分别使能
- ❖ 奇偶校验位控制：
 - 发送奇偶校验位
 - 检测接收的数据字节的奇偶校验位
- ❖ LIN 断开帧的产生和检测
- ❖ 支持红外数据协议(IrDA)
- ❖ 同步传输模式以及为同步传输输出发送时钟

- ❖ 支持兼容 ISO7816-3 的智能卡接口：
 - 字节模式(T=0)
 - 块模式(T=1)
 - 直接和反向转换
- ❖ 多处理器通信：
 - 如果地址不匹配，则进入静默模式
 - 通过线路空闲检测或者地址掩码检测从静默模式唤醒
- ❖ 支持 ModBus 通信：
 - 超时功能
 - CR/LF 字符识别
- ❖ 从深度睡眠模式唤醒：
 - 通过标准的 RBNE 中断
 - 通过 WUF 中断
- ❖ 多种状态标志：
 - 传输检测标志：接收缓冲区不为空(RBNE)，发送缓冲区为空(TBE)，传输完成(TC)
 - 错误检测标志：过载错误(ORERR)，噪声错误(NERR)，帧格式错误(FERR)，奇偶校验错误(PERR)
 - 硬件流控操作标志：CTS 变化(CTSF)
 - LIN 模式标志：LIN 断开检测(LBDF)
 - 多处理器通信模式标志：IDLE 帧检测(IDLEF)
 - ModBus 通信标志：地址/字符匹配(AMF)，接收超时(RTF)
 - 智能卡模式标志：块结束(EBF)和接收超时(RTF)
 - 从深度睡眠模式唤醒标志
 - 若相应的中断使能，这些事件发生将会触发中断

GD32F130G8U6 有两个 USART 模块：USART0 和 USART1，USART0 完全实现上述功能，而 USART1 值实现了上面所介绍功能的部分，下面的这些功能在 USART1 中没有实现：

- ❖ 自动波特率检测
- ❖ 智能卡模式
- ❖ IrDA SIR ENDEC 模块
- ❖ LIN 模式
- ❖ 双时钟域和从深度睡眠模式唤醒
- ❖ 接收超时中断
- ❖ Modbus 通信

USART 相关管脚：

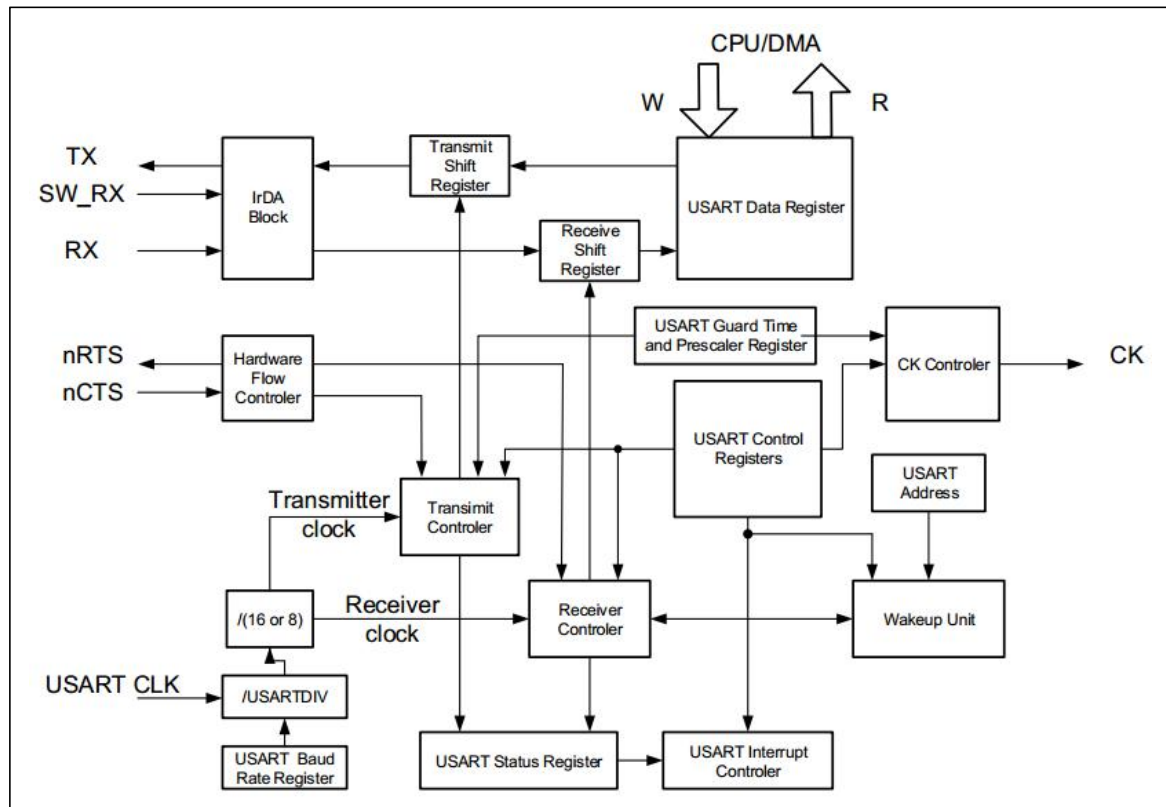
RX：类型--输入，接收数据

TX：类型--输出，发送数据，当 USART 使能后，若无数据发送，默认为高电平

CK：类型--输出，用于同步通讯的串行时钟信号

nCTS：类型--输入，硬件流控模式发送使能信号

USART 模块内部框图如下：



USART 数据帧开始与起始位，结束于停止位，数据长度 8/9 位，停止位可配置为 1、1.5 或 2，校验位可为奇校验、偶校验或无校验。波特率分频系数是一个 16 位的数字，包括 12 位整数和 4 位小数，可以使 USART 能够产生所有标准波特率。

2、print 输出

2.1 配置 printf 输出

本小节介绍如何使用 GD32F130G8U6 的 USART0 实现 print 输出。使用的例程：USART_Printf，该例程位于 GD32F130G8_Examples\GD32F130G8_04_USART\目录下。USART0 的配置步骤如下：

- 1) 使能 USART0 对应的 GPIO 时钟
- 2) 使能 USART0 外设时钟
- 3) 配置 USART0 的 TX 引脚作为复用功能、复用模式，带上拉电阻、推挽输出，速度 10MHz
- 4) 配置 USART0 的 RX 引脚作为复用功能、复用模式，带上拉电阻、推挽输出，速度 10MHz
- 5) 配置 USART0：波特率 115200、使能发送、使能接收（默认：停止位 1 位、无奇偶校验）
- 6) 使能 USART0

2.2 例程介绍

USART0 的配置在 usart.c 文件中实现：


```
void USART0_Init(void)
{
    /* 使能 GPIOA 外设时钟 */
    rcu_periph_clock_enable(RCU_GPIOA);
    /* 使能 USART0 外设时钟 */
    rcu_periph_clock_enable(USART0_CLOCK);
    /* 配置 USART0 的 TX 引脚作为复用功能、复用模式，带上拉电阻、推挽输出，速度 10MHz*/
    gpio_af_set(USART0_PORT, GPIO_AF_1, USART0_TX_PIN);
    gpio_mode_set(USART0_PORT, GPIO_MODE_AF, GPIO_PUPD_PULLUP, USART0_TX_PIN);
    gpio_output_options_set(USART0_PORT, GPIO_OTYPE_PP, GPIO_OSPEED_10MHZ,
                           USART0_TX_PIN);
    /* 配置 USART0 的 RX 引脚作为复用功能、复用模式，带上拉电阻、推挽输出，速度 10MHz*/
    gpio_af_set(USART0_PORT, GPIO_AF_1, USART0_RX_PIN);
    gpio_mode_set(USART0_PORT, GPIO_MODE_AF, GPIO_PUPD_PULLUP, USART0_RX_PIN);
    gpio_output_options_set(USART0_PORT, GPIO_OTYPE_PP, GPIO_OSPEED_10MHZ,
                           USART0_RX_PIN);
    /* 配置 USART0：波特率 115200、使能发送、使能接收（默认：数据位 8 位、停止位 1 位，无奇偶校验） */
    usart_deinit(USART0);    //端口号
    usart_baudrate_set(USART0, 115200U);    //波特率 115200
    usart_transmit_config(USART0, USART_TRANSMIT_ENABLE);    //使能发送
    usart_receive_config(USART0, USART_RECEIVE_ENABLE);    //使能接收
    usart_enable(USART0);    //使能 USART0
}
```

USART0 的相关 GPIO 时钟与端口在 usart.h 文件中定义：


```
/* 定义 USARTx IO PORT 与 PIN */
#define USART0_CLOCK            RCU_USART0
#define USART0_PORT             GPIOA
#define USART0_TX_PIN          GPIO_PIN_9
#define USART0_RX_PIN          GPIO_PIN_10

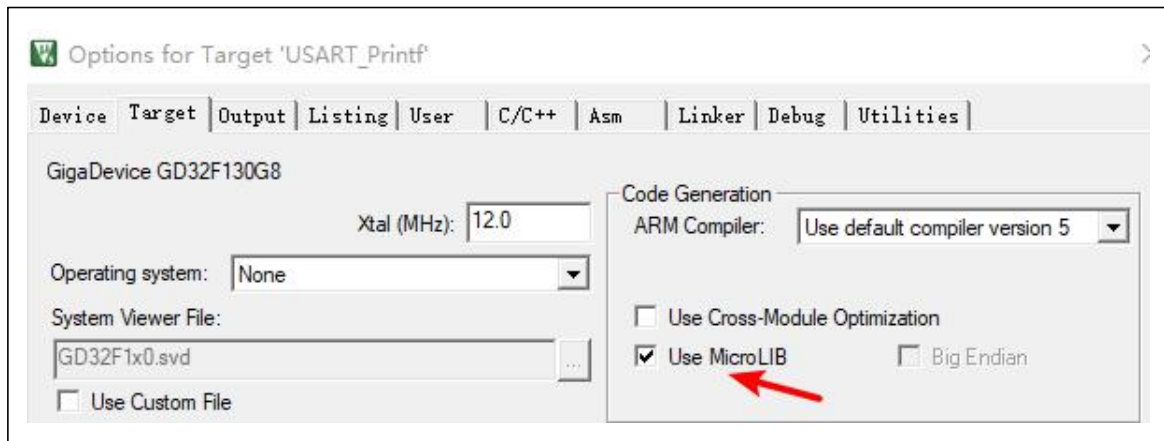
#define USART1_CLOCK            RCU_USART1
#define USART1_PORT             GPIOA
#define USART1_TX_PIN          GPIO_PIN_2
#define USART1_RX_PIN          GPIO_PIN_3
```

实现 USART0 的配置后，还需要使用 USART0 重定向 C 库中的 printf 函数，该函数在 main.c 文件中实现（main.c 文件中需要包含 stdio.h 头文件：#include <stdio.h>）：

```
/* 重定向 C 库中的 printf 函数 */
```

```
int fputc(int ch, FILE *f)
{
    usart_data_transmit(USART0, (uint8_t)ch);
    while(RESET == usart_flag_get(USART0, USART_FLAG_TBE));
    return ch;
}
```

重定向后还需要配置一下 MDK 的环境才能使用 printf 函数，点击  进入工程的 Options，在 Target 栏下把 Use Micro LIB 选项勾上：

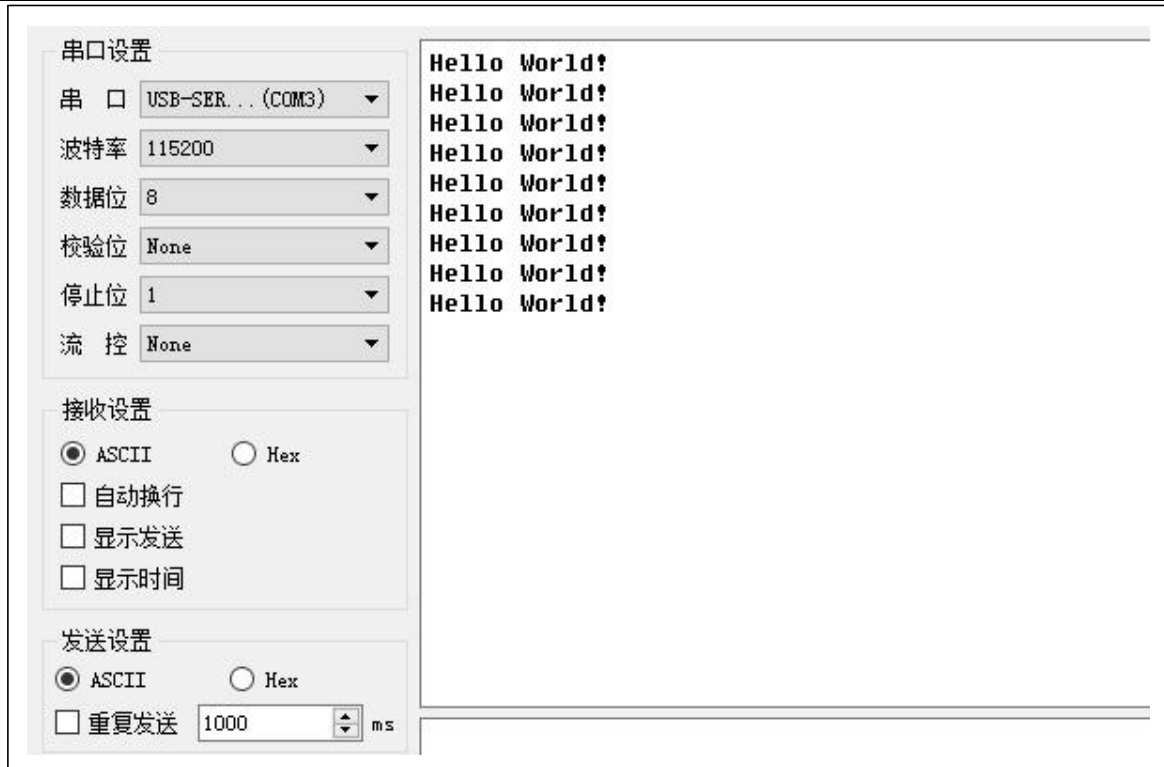


之后就可以使用 printf 函数向串口发送数据了（主函数），主函数如下：

```
int main(void)
{
    systick_config(); //systick 初始化
    LED_Init(); //LED 初始化
    USART0_Init(); //串口 0 初始化
    while(1)
    {
        printf("Hello World!\r\n"); //串口打印：Hello World!
        LED1_Toggle();
        delay_1ms(500);
    }
}
```

把程序下载到核心板后，LED1 闪烁；使用一个 USB 转 TTL 模块与核心板进行连接，连接线序为：模块的 TX-->PA10、模块的 RX-->PA9、模块的 GND-->GND。连接好后把 USB 转 TTL 模块接入电脑，打开电脑上位机（串口调试助手），上位机配置为：波特率 115200、数据位 8 位、校验位无、停止位 1 位、流控无。然后选择正确串口（不了解的可以百度一下：如何查看串口号），把串口打开就可以看到每 500ms 打印一次：Hello World!

程序效果如下：



3、串口收发中断

3.1 串口中断配置

本小节介绍如何使用串口中断的方式进行收发数据，使用的例程：USART_TxRx_Interrupt，该例程位于 GD32F130G8_Examples\GD32F130G8_04_USART\目录下。例程主要在 USART_Printf 例程上添加了中断部分的内容，添加的内容在主函数中，中断处理函数在 gd32f1x0_it.c 文件。主函数处理过程如下：

- 1) LED 初始化、串口初始化
- 2) USART0 的中断优先级配置
- 3) 使能串口发送中断
- 4) 等待串口发送数据（数据在中断处理函数中发送）
- 5) 等待数据发送完成（中断处理函数中发送完成后会关闭发送中断），发送完成点亮 LED1
- 6) 使能串口接收中断
- 7) 等待接收 32Byte 数据
- 8) 接收数据大于等于 32Byte 后（中断处理函数中接收大于等于 32Byte 后会关闭接收中断），串口打印 USART receive successfully!，点亮 LED2

3.2 例程介绍

数据发送、接收缓存和发送接收数据大小等变量定义在 main.c 中（gd32f1x0_it.c 中申明引用）：

```
#define ARRAYNUM(arr_naname)      (uint32_t)(sizeof(arr_naname) / sizeof(*(arr_naname)))
#define TRANSMIT_SIZE              (ARRAYNUM(transmitter_buffer) - 1)  //计算发送数据的大小
```

```
uint8_t transmitter_buffer[] = "USART interrupt test\n\r"; //要发送的数据
uint8_t receiver_buffer[32]; //定义接收数据的缓存
uint8_t transfersize = TRANSMIT_SIZE; //定义发送数据的大小
uint8_t receivesize = 32; //定义接收数据大小
__IO uint8_t txcount = 0; //定义发送数量变量
__IO uint16_t rxcount = 0; //定义接收数量变量
```

主函数如下：

```
int main(void)
{
    LED_Init(); //LED 初始化
    USART0_Init(); //串口 0 初始化
    /* USART0 的中断配置 */
    nvic_irq_enable(USART0_IRQn, 0, 0);
    /* 使能 USART0 TBE (串口发送) 中断 */
    usart_interrupt_enable(USART0, USART_INT_TBE);
    /* 等待串口中断发送 transmitter_buffer 数据 */
    while(txcount < transfersize);
    /* 等待串口发送完成 */
    while(RESET == usart_flag_get(USART0, USART_FLAG_TC));
    LED1_ON(); //LED1 亮
    /* 使能 RBNE (串口接收) 中断 */
    usart_interrupt_enable(USART0, USART_INT_RBNE);
    /* 等待接收数据, 接收数据大于等于 32Byte 时打印接收完成信息 */
    while(rxcount < receivesize);
    if(rxcount == receivesize)
        printf("USART receive successfully!\n\r");
    LED2_ON(); //LED2 亮
    while(1)
    {
        ;
    }
}
```

gd32f1x0_it.c 文件下的中断处理函数如下：

```
void USART0_IRQHandler(void)
{
    /* 串口接收中断 */
    if(RESET != usart_interrupt_flag_get(USART0, USART_INT_FLAG_RBNE))
    {
```

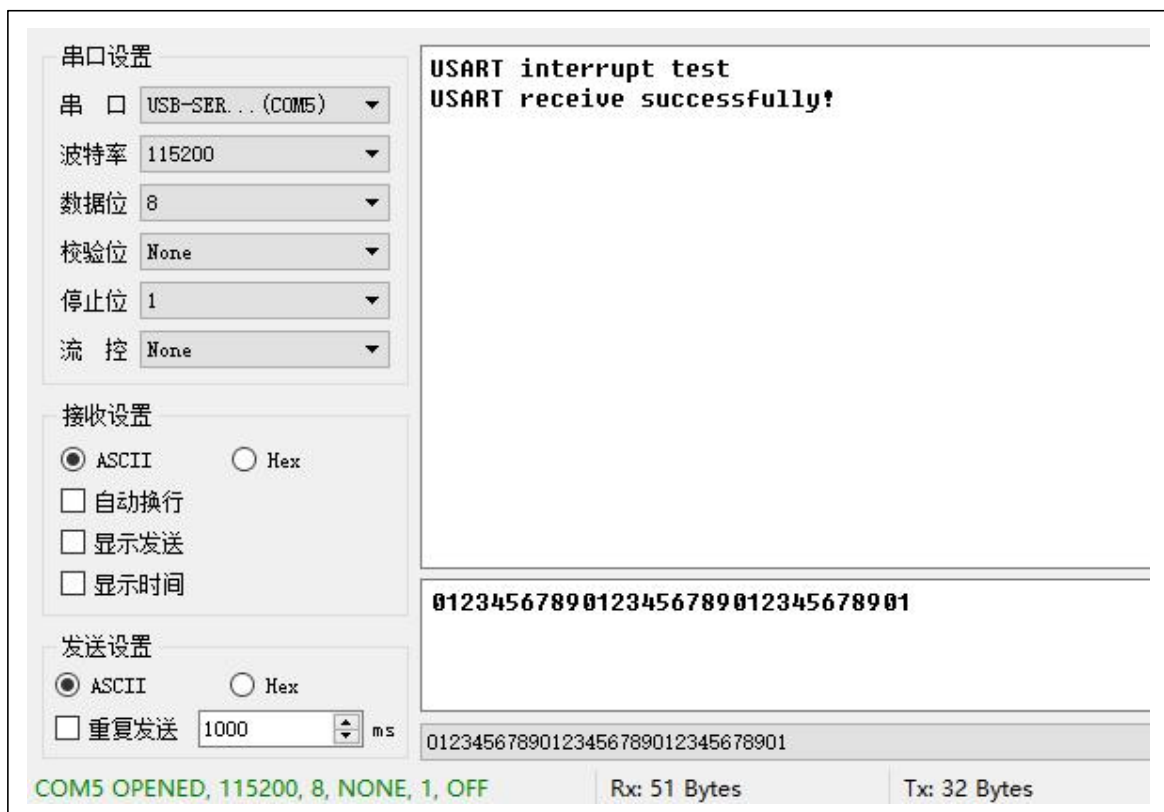
```

/* 保存接收的数据 */
receiver_buffer[rxcount++] = usart_data_receive(USART0);
if(rxcount == receivesize) //如果接收的数据量 = 32，关闭接收中断
    usart_interrupt_disable(USART0, USART_INT_RBNE);
}

/* 串口发送中断 */
if(RESET != usart_interrupt_flag_get(USART0, USART_INT_FLAG_TBE))
{
    /* 发送 transmitter_buffer 数据 */
    usart_data_transmit(USART0, transmitter_buffer[txcount++]);
    if(txcount == transfersize) //数据全部发送完毕，关闭发送中断
        usart_interrupt_disable(USART0, USART_INT_TBE);
}
}

```

把程序下载到核心板后，使用一个 USB 转 TTL 模块与核心板进行连接，连接线序为：模块的 TX-->PA10、模块的 RX-->PA9、模块的 GND-->GND。连接好后把 USB 转 TTL 模块接入电脑，打开电脑上位机（串口调试助手），上位机配置为：波特率 115200、数据位 8 位、校验位无、停止位 1 位、流控无。然后选择正确串口（不了解的可以百度一下：如何查看串口号），把串口打开，按一下板子 RST 键，可以看到串口打印出：USART interrupt test，LED1 亮，然后在上位机往串口发送 32Byte 数据，串口打印：USART receive successfully!，LED2 亮。



4、串口 DMA 传输

4.1 串口 DMA 配置

本小节介绍如何使用 DMA 的方式进行收发数据，使用的例程：USART_DMA，该例程位于 GD32F130G8_Examples\GD32F130G8_04_USART\目录下。例程主要在 USART_Printf 例程上添加了 DMA 部分的内容，添加的内容在主函数中。其中 DMA_CH1 用于 USART0_TX，DMA_CH2 用于 USART0_RX。主函数处理过程如下：

- 1) 定义 DMA 初始化结构体
- 2) 使能 DMA 外设时钟
- 3) 初始化 LED、初始化 USART0
- 4) 串口打印信息：USART DMA TEST!, LED1 亮
- 5) 初始化 DMA_CH1（用于串口发送）：
 - a. 内存到外设模式、内存地址、内存地址递增、内存数据宽度 8 位、数据量、外设地址、外设地址不递增、外设数据宽度 8 位、优先级超高
 - b. 禁能循环模式
 - c. 禁能内存到内存模式
 - d. 使能 DMA_CH1
 - e. 使能 USART0 的 DMA 发送
 - f. 等待 DMA 传输完成，完成后串口显示：rUSART DMA receive and transmit example, please input 10 bytes:
- 6) 初始化 DMA_CH2（用于串口接收）：
 - a. 外设到内存模式、内存地址、内存地址递增、内存数据宽度 8 位、数据量 10、外设地址、外设地址不递增、外设数据宽度 8 位、优先级超高
 - b. 禁能循环模式
 - c. 禁能内存到内存模式
 - d. 使能 DMA_CH2
 - e. 使能 USART0 的 DMA 接收
 - f. 等待 DMA 传输完成（DMA 等待接收 10Byte 数据），完成后串口打印串口发送过来的 10Byte 消息，LED2 状态改变

4.2 例程介绍

DMA 部分代码都在主函数中实现，主函数如下：

```
int main(void)
{
    /* 定义 DMA 初始化结构体 */
    dma_parameter_struct dma_init_struct;
    /* 使能 DMA 外设时钟 */
```

```
rcu_periph_clock_enable(RCU_DMA);
LED_Init(); //LED 初始化
USART0_Init(); //串口 0 初始化
/* 打印信息 */
printf("USART DMA TEST!\n\r");
LED1_ON(); //LED1 亮
/* 初始化 DMA_CH1 */
dma_deinit(DMA_CH1); //重置 DMA_CH1 相关寄存器
dma_init_struct.direction = DMA_MEMORY_TO_PERIPHERAL; //内存到外设模式
dma_init_struct.memory_addr = (uint32_t)txbuffer; //内存基地址
dma_init_struct.memory_inc = DMA_MEMORY_INCREASE_ENABLE; //内存地址递增
dma_init_struct.memory_width = DMA_MEMORY_WIDTH_8BIT; //数据宽度 8 位
dma_init_struct.number = ARRAYNUM(txbuffer); //数据量
dma_init_struct.periph_addr = USART0_TDATA_ADDRESS; //外设基地址
dma_init_struct.periph_inc = DMA_PERIPH_INCREASE_DISABLE; //外设地址不递增
dma_init_struct.periph_width = DMA_PERIPHERAL_WIDTH_8BIT; //数据宽度 8 位
dma_init_struct.priority = DMA_PRIORITY_ULTRA_HIGH; //优先级超高
dma_init(DMA_CH1, dma_init_struct); //初始化配置
/* 禁能循环模式 */
dma_circulation_disable(DMA_CH1);
/* 禁能内存到内存模式 */
dma_memory_to_memory_disable(DMA_CH1);
/* 使能 DMA_CH1 */
dma_channel_enable(DMA_CH1);
/* 使能 USART0 的 DMA 发送 */
usart_dma_transmit_config(USART0, USART_DENT_ENABLE);
/* 等待 DMA 传输完成 */
while (RESET == dma_flag_get(DMA_CH1, DMA_FLAG_FTF));
while(1){
    /* 初始化 DMA_CH2 */
    dma_deinit(DMA_CH2); //重置 DMA_CH2 相关寄存器
    dma_init_struct.direction = DMA_PERIPHERAL_TO_MEMORY; //外设到内存模式
    dma_init_struct.memory_addr = (uint32_t)rxbuffer; //内存基地址
    dma_init_struct.memory_inc = DMA_MEMORY_INCREASE_ENABLE; //内存地址递增
    dma_init_struct.memory_width = DMA_MEMORY_WIDTH_8BIT; //数据宽度 8 位
    dma_init_struct.number = 10; //数据量 10
    dma_init_struct.periph_addr = USART0_RDATA_ADDRESS; //外设基地址
    dma_init_struct.periph_inc = DMA_PERIPH_INCREASE_DISABLE; //外设地址不递增
    dma_init_struct.periph_width = DMA_PERIPHERAL_WIDTH_8BIT; //数据宽度 8 位
    dma_init_struct.priority = DMA_PRIORITY_ULTRA_HIGH; //优先级超高
```

```

    dma_init(DMA_CH2, dma_init_struct);    //初始化配置
    /* 禁能循环模式 */
    dma_circulation_disable(DMA_CH2);
    /* 禁能内存到内存模式 */
    dma_memory_to_memory_disable(DMA_CH2);
    /* 使能 DMA_CH2 */
    dma_channel_enable(DMA_CH2);
    /* 使能 USART 的 DMA 接收 */
    usart_dma_receive_config(USART0, USART_DENR_ENABLE);
    /* 等待 DMA 传输完成 */
    while (RESET == dma_flag_get(DMA_CH2, DMA_FLAG_FTF));
    /* 打印 DMA 接收的信息 */
    printf("\n\r%s\n\r", rxbuffer);
    LED2_Toggle(); //LED2 状态改变
}
}

```

地址的定义与发送缓存定义在 main.c 文件下：

```

uint8_t rxbuffer[10]; //接收数据缓存
/* 发送数据缓存 */
uint8_t txbuffer[]="\n\rUSART DMA receive and transmit example, please input 10 bytes:\n\r";

#define ARRAYNUM(arr_name) ((uint32_t)(sizeof(arr_name) / sizeof(*(arr_name))))
#define USART0_TDATA_ADDRESS ((uint32_t)0x40013828) //USART0 的发送数据寄存器地址
#define USART0_RDATA_ADDRESS ((uint32_t)0x40013824) //USART0 的接收数据寄存器地址

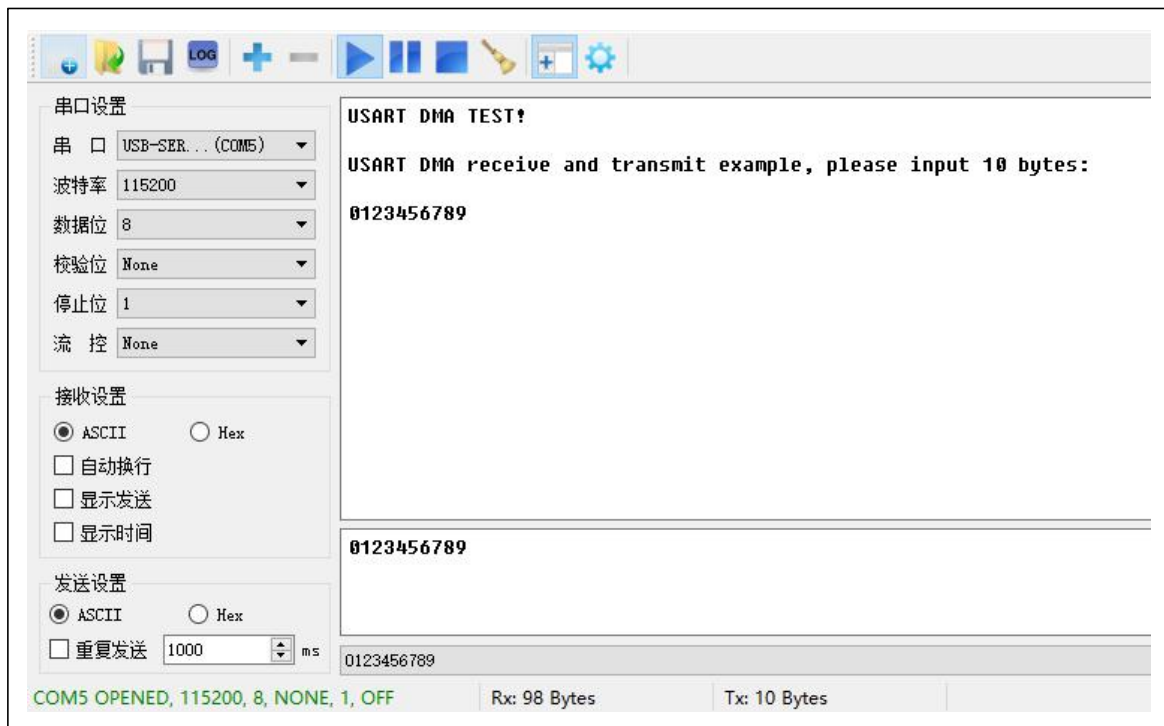
```

把程序下载到核心板后，使用一个 USB 转 TTL 模块与核心板进行连接，连接线序为：模块的 TX-->PA10、模块的 RX-->PA9、模块的 GND-->GND。连接好后把 USB 转 TTL 模块接入电脑，打开电脑上位机（串口调试助手），上位机配置为：波特率 115200、数据位 8 位、校验位无、停止位 1 位、流控无。然后选择正确串口（不了解的可以百度一下：如何查看串口号），把串口打开，按一下板子 RST 键，可以看到串口打印出：

USART DMA TEST!

USART DMA receive and transmit example, please input 10 bytes:

之后往串口发送 10 字节数据，串口会把接收到的 10 字节数据发送回来（超过 10 字节，只发送前 10 字节）效果如下：



第五章 TIMER 应用

TIMER（定时器），GD32F130G8U6 一共有 7 个定时器，包括 1 个 32 位通用定时器（TIMER1）、5 个 16 位通用定时器（TIMER2、13~16）和一个 16 位高级定时器（TIMER0）。本章将分别介绍三个定时器（TIMER0、TIMER1、TIMER2）以及在不同功能上的应用：

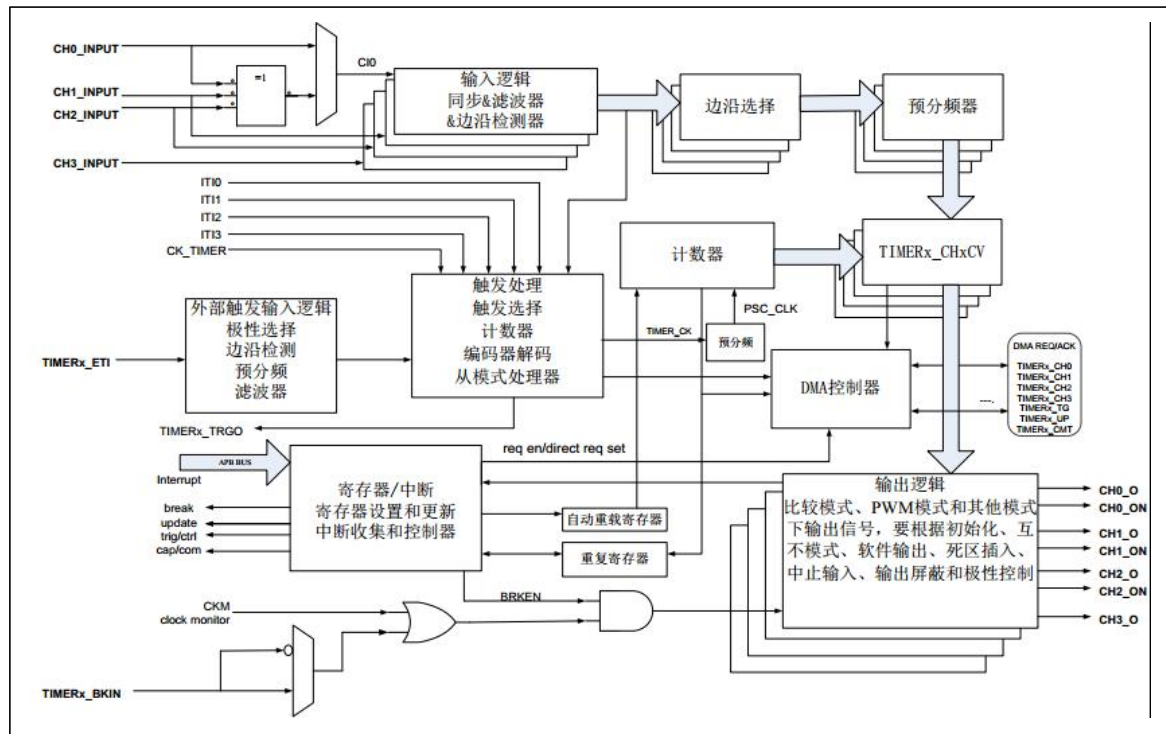
- 高级定时器 TIMER0 简介
 - TIMER0 的互补信号输出
 - TIMER0 的 DMA 更新事件输出 PWM
- 通用定时器 TIMER1&TIMER2 简介
 - TIMER1 的 PWM 输出
 - TIMER1 的单脉冲输出
 - TIMER2 的输入捕获
 - TIMER2 的 PWM 捕获

1、高级定时器 TIMER0 简介

TIMER0 是一个四通道的高级定时器，支持输入捕获和输出比较。可以产生 PWM 信号控制电机和电源管理。高级定时器含有一个 16 位无符号计数器。高级定时器是可编程的，可以被用来计数，其外部事件可以驱动其他定时器，它还包含了一个死区时间插入模块，非常适合电机控制。定时器和定时器之间是相互独立，但是他们可以被同步在一起形成一个更大的定时器，这些定时器的计数器一致地增加。TIMER0 的主要特征如下：

- ❖ 总通道数：4
- ❖ 计数器宽度：16 位
- ❖ 时钟源可选：内部时钟，内部触发，外部输入，外部触发
- ❖ 多种计数模式：向上计数，向下计数和中央计数
- ❖ 正交编码器接口：被用来追踪运动和分辨旋转方向和位置
- ❖ 霍尔传感器接口：用来做三相电机控制
- ❖ 可编程的预分频器：16 位，运行时可以被改变
- ❖ 每个通道可配置：输入捕获模式，输出比较模式，可编程的 PWM 模式，单脉冲模式
- ❖ 可编程的死区时间
- ❖ 自动重装载功能
- ❖ 可编程的计数器重复功能
- ❖ 中止输入功能
- ❖ 中断输出和 DMA 请求：更新事件，触发事件，比较/捕获事件和中止事件
- ❖ 多个定时器的菊链使得一个定时器可以同时启动多个定时器
- ❖ 定时器的同步允许被选择的定时器在同一个时钟周期开始计数
- ❖ 定时器主/从模式控制器

TIMER0 的结构框图如下：



2、TIMER0 的互补信号输出

2.1 互补信号输出配置

本小节介绍如何使用 TIMER0 实现互补信号输出。例程：TIMER0_Complementary_Signals，该例程位于 GD32F130G8_Examples\GD32F130G8_05_TIMER\GD32F130G8_TIMER0\目录下。例程使用 PA8 (TIMER0_CH0) 与 PA7 (TIMER0_CH0_ON) 输出一个频率 1000Hz，占空比 20 (80) %的互补脉冲信号。TIMER0 的配置步骤如下：

- 1) 使能 PA8 (TIMER0_CH0) 与 PA7 (TIMER0_CH0_ON) 所在 GPIO 时钟
- 2) 配置 PA8 端口用作复用功能
- 3) 配置 PA7 端口用作复用功能
- 4) 定义 TIMER0 初始化配置结构体
- 5) 使能 TIMER0 外设时钟
- 6) TIMER0 配置：72 分频（预分频）、边沿对齐、向上计数、周期 1000、时钟分频 1、重复计数 0
- 7) 使能捕获比较通道、使能捕获比较对立通道、使能捕获比较通道极性配置、使能捕获比较对立通道极性配置
- 8) TIMER0_CH0 输出配置
- 9) 配置 TIMER0_CH0 为 PWM0 模式
- 10) TIMER0 输出使能
- 11) TIMER0 重装载使能
- 12) TIMER0 使能

2.2 例程介绍

TIMER0 的配置分为两部分：TIMER0 相关的 GPIO 配置和 TIMER0 的模式配置，在 timer0 文件下实现：

```
void TIMER0_GPIO_Config(void)
```

```
{
    /* 使能 GPIOA 外设时钟 */
    rcu_periph_clock_enable(RCU_GPIOA);
    /* 配置 PA8(TIMER0_CH0) 用作复用功能 */
    gpio_mode_set(GPIOA, GPIO_MODE_AF, GPIO_PUPD_NONE, GPIO_PIN_8);
    gpio_output_options_set(GPIOA, GPIO_OTYPE_PP, GPIO_OSPEED_50MHZ, GPIO_PIN_8);
    gpio_af_set(GPIOA, GPIO_AF_2, GPIO_PIN_8);
    /* 配置 PA7(TIMER0_CH0_ON) 用作复用功能 */
    gpio_mode_set(GPIOA, GPIO_MODE_AF, GPIO_PUPD_NONE, GPIO_PIN_7);
    gpio_output_options_set(GPIOA, GPIO_OTYPE_PP, GPIO_OSPEED_50MHZ, GPIO_PIN_7);
    gpio_af_set(GPIOA, GPIO_AF_2, GPIO_PIN_7);
}
```

```
void TIMER0_Config(void)
```

```
{
    /* 定义 TIMER0 初始化配置结构体 */
    timer_oc_parameter_struct timer_ocintpara;
    timer_parameter_struct timer_initpara;
    /* 使能 TIMER0 外设时钟 */
    rcu_periph_clock_enable(RCU_TIMER0);
    /* TIMER0 配置 */
    timer_deinit(TIMER0);
    timer_initpara.prescaler      = 71;    //72 分频 = 72M/72 = 1M 时钟频率
    timer_initpara.alignedmode    = TIMER_COUNTER_EDGE; //边沿对齐
    timer_initpara.counterdirection = TIMER_COUNTER_UP; //向上计数
    timer_initpara.period         = 999;   //频率 1000Hz = 72MHz / (71+1) / (999+1)
    timer_initpara.clockdivision   = TIMER_CKDIV_DIV1; //时钟分频 1
    timer_initpara.repetitioncounter = 0; //重复计数 0
    timer_init(TIMER0, &timer_initpara); //初始化配置

    timer_ocintpara.outputstate = TIMER_CCX_ENABLE; //使能捕获比较通道
    timer_ocintpara.outputnstate = TIMER_CCXN_ENABLE; //禁能捕获比较对立通道
    timer_ocintpara.ocpolarity   = TIMER_OC_POLARITY_HIGH; //获比较通道极性高
    timer_ocintpara.ocnpolarity  = TIMER_OCN_POLARITY_HIGH; //捕获比较对立通道极性高
    timer_ocintpara.ocidlestate  = TIMER_OC_IDLE_STATE_LOW; //获比较通道空闲低
}
```

```

timer_ocintpara.ocnidlestate = TIMER_OCN_IDLE_STATE_LOW; //捕获比较对立通道空闲低
/* TIMER0_CH0 输出配置 */
timer_channel_output_config(TIMER0,TIMER_CH_0,&timer_ocintpara);
/* 配置 TIMER0_CH0 为 PWM0 模式 */
timer_channel_output_pulse_value_config(TIMER0,TIMER_CH_0,199); //初始脉冲值为 0
timer_channel_output_mode_config(TIMER0,TIMER_CH_0,TIMER_OC_MODE_PWM0);
//PWM0 输出模式
timer_channel_output_shadow_config(TIMER0,TIMER_CH_0,TIMER_OC_SHADOW_DISABLE);
//禁能影子寄存器

/* TIMER0 输出使能 */
timer_primary_output_config(TIMER0,ENABLE);
/* 重装载使能 */
timer_auto_reload_shadow_enable(TIMER0);
/* TIMER0 使能 */
timer_enable(TIMER0);
}

```

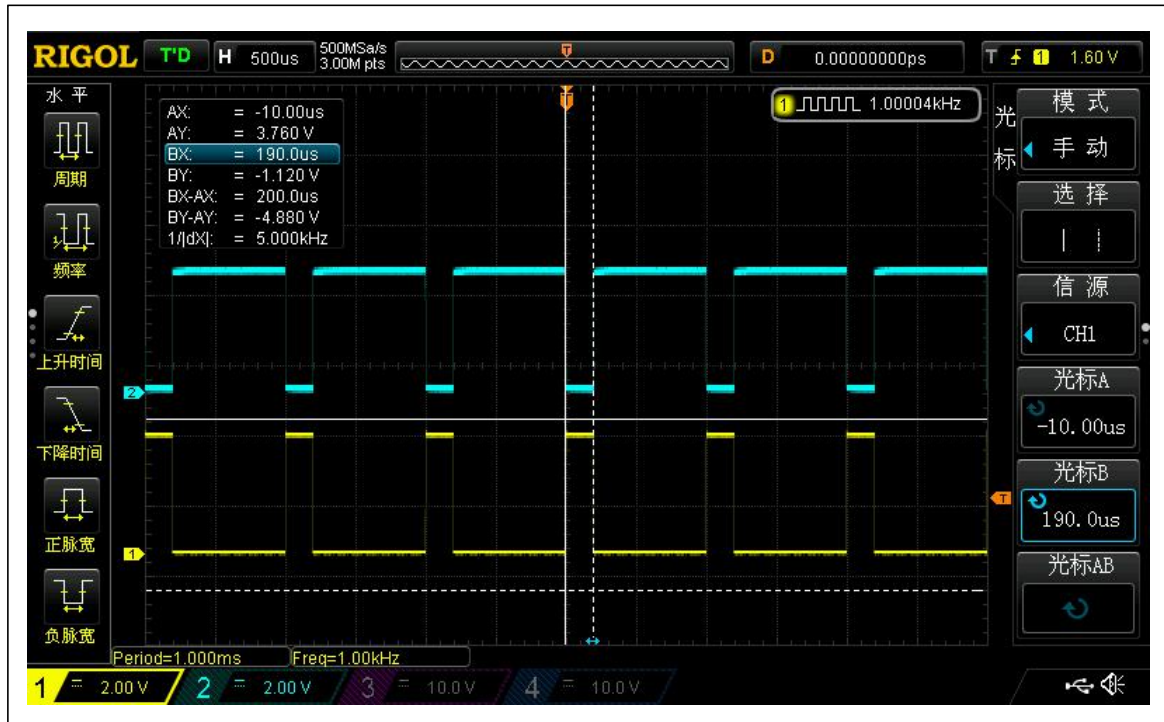
主函数如下所示：

```

int main(void)
{
    systick_config(); //初始化滴答定时器
    LED_Init(); //LED 初始化
    TIMER0_GPIO_Config(); //TIMER0 的 GPIO 配置
    TIMER0_Config(); //TIMER0 配置
    while(1)
    {
        LED1_Toggle(); //切换 LED1 状态
        delay_1ms(500);
    }
}

```

把程序下载到核心板后，LED1 闪烁。PA8（TIMER0_CH0）与 PA7（TIMER0_CH0_ON）输出一个频率 1000Hz，占空比 20（80）%的互补脉冲信号，使用示波器检测效果如下：



3、TIMER0 的 DMA 更新事件输出 PWM

3.1 TIMER0 的 DMA 更新事件配置

本小节介绍如何使用 TIMER0 实现 DMA 更新输出 PWM 信号。例程：TIMER0_DMA，该例程位于 GD32F130G8_Examples\GD32F130G8_05_TIMER\GD32F130G8_TIMER0\ 目录下。例程使用 PA8 (TIMER0_CH0) 与 PA9 (TIMER0_CH1) 分别输出一个频率 1000Hz，占空比 20%与频率 1000Hz，占空比 50%的 PWM 信号。TIMER0 的配置步骤如下：

- 1) PA8 (TIMER0_CH0) 与 PA9 (TIMER0_CH1) GPIO 配置：
 - a. 使能 GPIOA 时钟
 - b. 配置 PA8 为复用功能
 - c. 配置 PA9 为复用功能
- 2) TIMER0 的 DMA (DMA_CH4) 配置，使用的是 TIMER0 的 DMA 更新事件输出：
 - a. 定义 DMA 初始化配置结构体
 - b. 使能 DMA 外设时钟
 - c. 初始化 DMA_CH4：内存到外设模式、内存地址、使能内存地址递增、内存数据宽度 16bit、数据量 2、外设地址、使能外设地址递增、外设地址宽度 16bit、优先级高
 - d. 使能 DMA_CH4 循环
 - e. 禁止内存到内存模式
 - f. 使能 DMA_CH4
- 3) TIMER0 配置：
 - a. 定义 TIMER0 初始化结构体

- b. 使能 TIMER0 外设时钟
- c. TIMER0 配置：72 分频（预分频）、边沿对齐、向上计数、周期 1000、时钟分频 1、重复计数 0
- d. 使能捕获比较通道、禁能捕获比较对立通道、使能捕获比较通道极性配置、使能捕获比较对立通道极性配置
- e. TINNER0_CH0、TINNER0_CH1 输出配置
- f. 配置 TINNER0_CH0、TINNER0_CH1 为 PWM0 模式
- g. TIMER0 主输出使能
- h. TIMER0 更新 DMA 请求使能
- i. TIMER0 重装载使能
- j. TIMER0 使能

3.2 例程介绍

TIMER0 的 GPIO 配置、DMA 配置与 TIMER0 模式配置都在 timer0.c 文件下实现：

```
void TIMER0_GPIO_Config(void)
{
    /* 使能 GPIOA 外设时钟 */
    rcu_periph_clock_enable(RCU_GPIOA);
    /* 配置 PA8(TIMER0_CH0) 作为复用功能 */
    gpio_mode_set(GPIOA, GPIO_MODE_AF, GPIO_PUPD_NONE, GPIO_PIN_8);
    gpio_output_options_set(GPIOA, GPIO_OTYPE_PP, GPIO_OSPEED_50MHZ, GPIO_PIN_8);
    gpio_af_set(GPIOA, GPIO_AF_2, GPIO_PIN_8);
    /* 配置 PA9(TIMER0_CH1) 作为复用功能 */
    gpio_mode_set(GPIOA, GPIO_MODE_AF, GPIO_PUPD_NONE, GPIO_PIN_9);
    gpio_output_options_set(GPIOA, GPIO_OTYPE_PP, GPIO_OSPEED_50MHZ, GPIO_PIN_9);
    gpio_af_set(GPIOA, GPIO_AF_2, GPIO_PIN_9);
}

void TIMER0_DMA_Config(uint16_t* memory_addr)
{
    /* 定义 DMA 初始化配置结构体 */
    dma_parameter_struct dma_init_struct;
    /* 使能 DMA 外设时钟 */
    rcu_periph_clock_enable(RCU_DMA);
    /* 初始化 DMA channel4 */
    dma_deinit(DMA_CH4);
    dma_init_struct.direction    = DMA_MEMORY_TO_PERIPHERAL;    //内存到外设
    dma_init_struct.memory_addr = (uint32_t)memory_addr; //内存地址
}
```

```

dma_init_struct.memory_inc    = DMA_MEMORY_INCREASE_ENABLE; //使能内存地址递增
dma_init_struct.memory_width = DMA_MEMORY_WIDTH_16BIT; //内存数据宽度 16bit
dma_init_struct.number        = 2; //数据量 2
dma_init_struct.periph_addr   = (uint32_t)TIMER0_DMATB; //外设地址
dma_init_struct.periph_inc    = DMA_PERIPH_INCREASE_ENABLE; //使能外设地址递增
dma_init_struct.periph_width = DMA_PERIPHERAL_WIDTH_16BIT; //外设数据宽度 16bit
dma_init_struct.priority      = DMA_PRIORITY_HIGH; //优先级高
dma_init(DMA_CH4,dma_init_struct); //初始化配置
dma_circulation_enable(DMA_CH4); //使能 DMA_CH4 循环
dma_memory_to_memory_disable(DMA_CH4); //禁止内存到内存模式
/* 使能 DMA channel4 */
dma_channel_enable(DMA_CH4);
}

void TIMER0_Config(void)
{
    /* 定义 TIMER 初始化配置结构体 */
    timer_oc_parameter_struct timer_ocintpara;
    timer_parameter_struct timer_initpara;
    /* 使能 TIMER0 外设时钟 */
    rcu_periph_clock_enable(RCU_TIMER0);
    /* TIMER0 配置 */
    timer_deinit(TIMER0);
    timer_initpara.prescaler      = 71; //72 分频 = 72M/72 = 1M 时钟频率
    timer_initpara.alignedmode    = TIMER_COUNTER_EDGE; //边沿对齐
    timer_initpara.counterdirection = TIMER_COUNTER_UP; //向上计数
    timer_initpara.period         = 999; //频率 1000Hz = 72MHz / (71+1) / (999+1)
    timer_initpara.clockdivision  = TIMER_CKDIV_DIV1; //时钟分频 1
    timer_initpara.repetitioncounter = 0; //重复计数 0
    timer_init(TIMER0,&timer_initpara); //初始化配置

    timer_ocintpara.outputstate = TIMER_CCX_ENABLE; //使能捕获比较通道
    timer_ocintpara.outputnstate = TIMER_CCXN_DISABLE; //禁能捕获比较对立通道
    timer_ocintpara.ocpolarity  = TIMER_OC_POLARITY_HIGH; //获比较通道极性高
    timer_ocintpara.ocnpolarity = TIMER_OCN_POLARITY_HIGH; //捕获比较对立通道极性高
    timer_ocintpara.ocidlestate = TIMER_OC_IDLE_STATE_HIGH; //获比较通道空闲高
    timer_ocintpara.ocnidlestate = TIMER_OCN_IDLE_STATE_LOW; //捕获比较对立通道空闲低

    /* TINERO_CH0、TINERO_CH1 输出配置 */
    timer_channel_output_config(TIMER0,TIMER_CH_0,&timer_ocintpara);

```



```

timer_channel_output_config(TIMERO,TIMER_CH_1,&timer_ocintpara);
/* 配置 TINTER0_CH0、TINTER0_CH1 为 PWM0 模式 */
timer_channel_output_pulse_value_config(TIMERO,TIMER_CH_0,0); //初始脉冲值为 0
timer_channel_output_mode_config(TIMERO,TIMER_CH_0,TIMER_OC_MODE_PWM0);
//PWM0 输出模式
timer_channel_output_shadow_config(TIMERO,TIMER_CH_0,TIMER_OC_SHADOW_DISABLE);
//禁能影子寄存器
timer_channel_output_pulse_value_config(TIMERO,TIMER_CH_1,0); //初始脉冲值为 0
timer_channel_output_mode_config(TIMERO,TIMER_CH_1,TIMER_OC_MODE_PWM0);
//PWM0 输出模式
timer_channel_output_shadow_config(TIMERO,TIMER_CH_1,TIMER_OC_SHADOW_DISABLE);
//禁能影子寄存器
timer_primary_output_config(TIMERO,ENABLE); /* TIMERO 主输出使能 */
timer_dma_transfer_config(TIMERO,TIMER_DMACFG_DMATA_CH0CV,TIMER_DMACFG_DMATC
_4TRANSFER); /* TIMERO 更新 DMA 请求使能 */
timer_dma_enable(TIMERO,TIMER_DMA_UPD);
/* TIMERO 重装载使能 */
timer_auto_reload_shadow_enable(TIMERO);
/* TIMERO 使能 */
timer_enable(TIMERO);
}

```

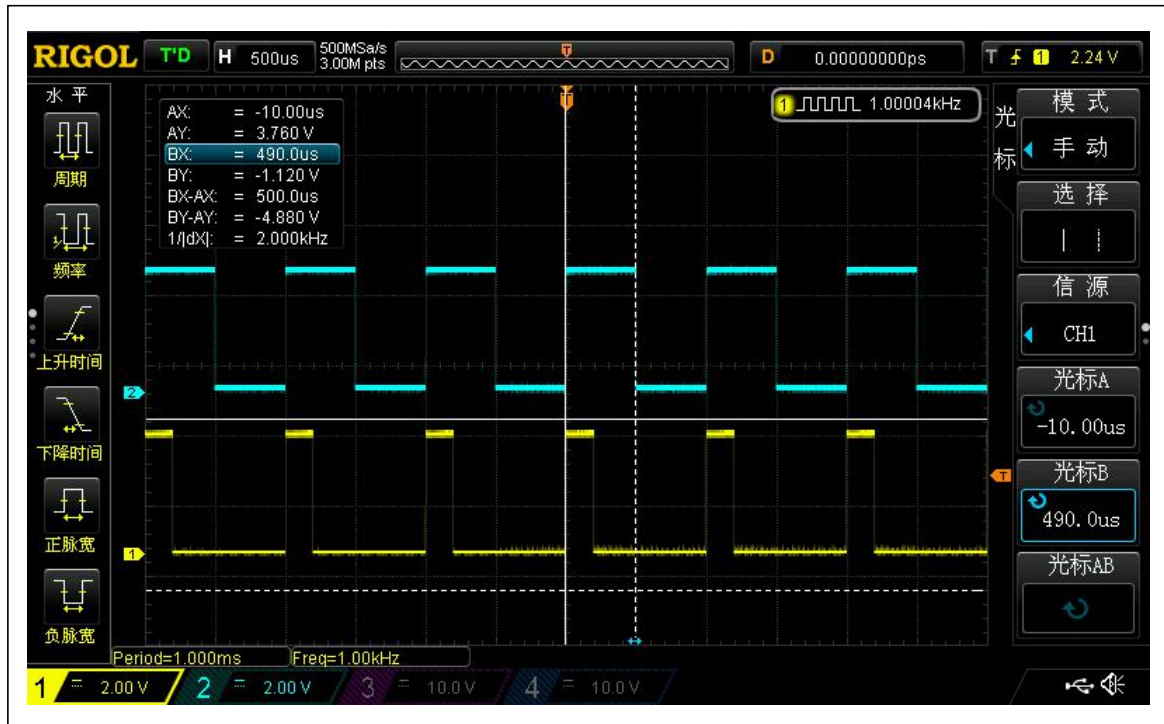
主函数如下：

```

int main(void)
{
    systick_config(); //初始化滴答定时器
    LED_Init(); //LED 初始化
    TIMERO_GPIO_Config(); //TIMERO 的 GPIO 配置
    TIMERO_DMA_Config(buffer); //TIMERO 的 DMA_CH4 配置
    TIMERO_Config(); //TIMERO 配置
    while(1)
    {
        LED1_Toggle();
        delay_1ms(500);
    }
}

```

把程序下载到核心板后，LED1 闪烁。PA8（TIMERO_CH0）与 PA9（TIMERO_CH1）分别输出一个频率 1000Hz，占空比 20%和频率 1000Hz，占空比 50%的脉冲信号，使用示波器检测效果如下：

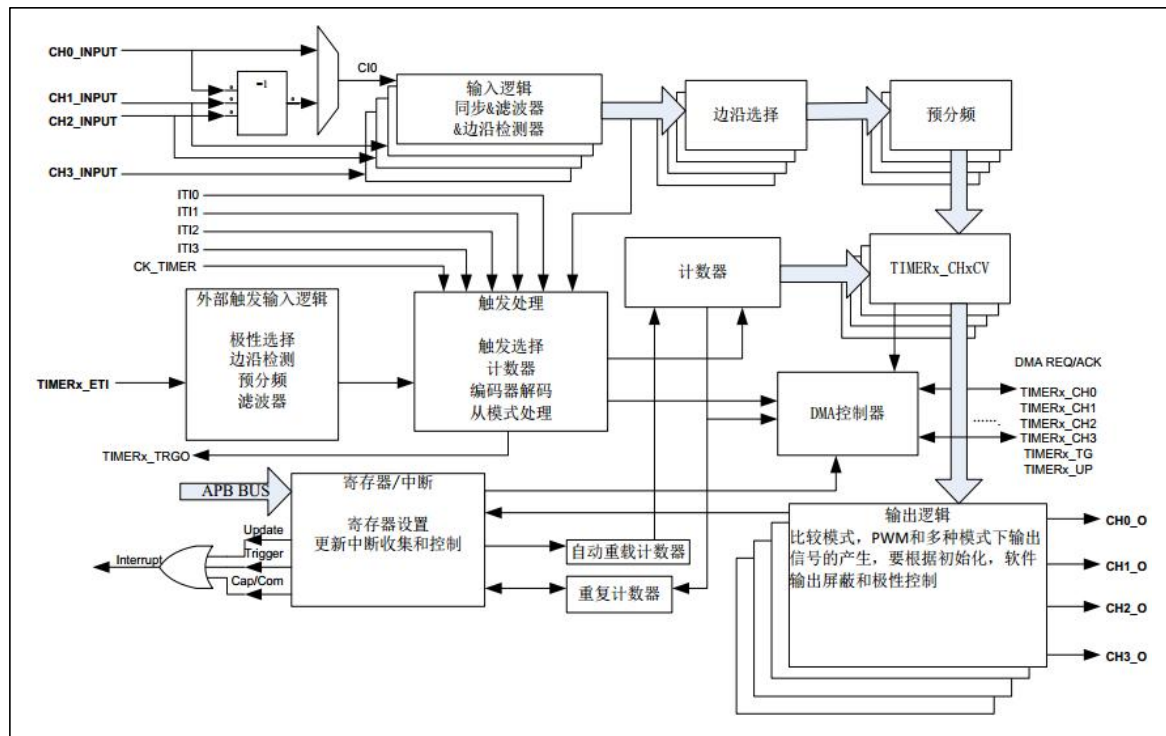


4、通用定时器 TIMER1&TIMER2 简介

TIMER1&TIMER2 属于通用定时器，是 4 通道定时器，支持输入捕获，输出比较，产生 PWM 信号控制电机和电源管理。通用定时器是可编程的，可以被用来计数，其外部事件可以驱动其他定时器。他们的主要特性如下：

- ❖ 总通道数：4
- ❖ 计数器宽度：16 位(TIMER2)，32 位(TIMER1)
- ❖ 时钟源可选：内部时钟，内部触发，外部输入，外部触发
- ❖ 多种计数模式：向上计数，向下计数和中央计数
- ❖ 正交编码器接口：被用来追踪运动和分辨旋转方向和位置
- ❖ 霍尔传感器接口：用来做三相电机控制
- ❖ 可编程的预分频器：16 位，运行时可以被改变
- ❖ 每个通道可配置：输入捕获模式，输出比较模式，可编程的 PWM 模式，单脉冲模式
- ❖ 自动重装载功能
- ❖ 中断输出和 DMA 请求：更新事件，触发事件，比较/捕获事件
- ❖ 多个定时器的级链使得一个定时器可以同时启动多个定时器
- ❖ 定时器的同步允许被选择的定时器在同一个时钟周期开始计数
- ❖ 定时器主/从模式控制器

通用定时器结构框图如下：



5、TIMER1 的 PWM 输出

5.1 TIMER1 的 PWM 配置

本小节介绍如何使用 TIMER1 实现输出 PWM 信号。例程：TIMER1_PWM，该例程位于 GD32F130G8_Examples\GD32F130G8_05_TIMER\GD32F130G8_TIMER1\ 目录下。例程使用 PA0 (TIMER1_CH0) 与 PA1 (TIMER1_CH1) 分别输出一个频率 1000Hz，占空比 20% 与频率 1000Hz，占空比 50% 的 PWM 信号。TIMER1 的配置步骤如下：

- 1) PA0 (TIMER1_CH0) 与 PA1 (TIMER1_CH1) GPIO 配置：
 - a. 使能 GPIOA 时钟
 - b. 配置 PA0 为复用功能
 - c. 配置 PA1 为复用功能
- 2) TIMER1 配置：
 - a. 定义 TIMER1 初始化结构体
 - b. 使能 TIMER1 外设时钟
 - c. TIMER1 配置：72 分频、边沿对齐、向上计数、周期 1000、时钟分频 1、重复计数 0
 - d. TIMER1_CH0、TIMER1_CH1 通道配置：使能输出比较通道、输出极性高
 - e. TIMER1_CH0、TIMER1_CH1 配置
 - f. 配置 TIMER1_CH0 的 PWM：20% 占空比、PWM0 模式、禁能影子寄存器
 - g. 配置 TIMER1_CH1 的 PWM：50% 占空比、PWM0 模式、禁能影子寄存器
 - h. 自动重载使能
 - i. 使能 TIMER1

5.2 例程介绍

TIMER1 的 GPIO 配置与 TIMER1 的模式配置在 timer1.c 文件中实现：

```
void TIMER1_GPIO_Config(void)
{
    /* 使能 GPIOA 外设时钟 */
    rcu_periph_clock_enable(RCU_GPIOA);
    /* 配置 PA0(TIMER1 CH0) 用作复用功能 */
    gpio_mode_set(GPIOA, GPIO_MODE_AF, GPIO_PUPD_NONE, GPIO_PIN_0);
    gpio_output_options_set(GPIOA, GPIO_OTYPE_PP, GPIO_OSPEED_50MHZ, GPIO_PIN_0);
    gpio_af_set(GPIOA, GPIO_AF_2, GPIO_PIN_0);
    /* 配置 PA1(TIMER1 CH1) 用作复用功能 */
    gpio_mode_set(GPIOA, GPIO_MODE_AF, GPIO_PUPD_NONE, GPIO_PIN_1);
    gpio_output_options_set(GPIOA, GPIO_OTYPE_PP, GPIO_OSPEED_50MHZ, GPIO_PIN_1);
    gpio_af_set(GPIOA, GPIO_AF_2, GPIO_PIN_1);
}

void TIMER1_Config(void)
{
    /* 定义 TIMER 初始化配置结构体 */
    timer_oc_parameter_struct timer_ocintpara;
    timer_parameter_struct timer_initpara;
    /* 使能 TIMER1 外设时钟 */
    rcu_periph_clock_enable(RCU_TIMER1);

    /* TIMER1 配置 */
    timer_deinit(TIMER1);
    timer_initpara.prescaler      = 71;    //72 分频 = 72M/72 = 1KHz 时钟频率
    timer_initpara.alignedmode    = TIMER_COUNTER_EDGE; //边沿对齐
    timer_initpara.counterdirection = TIMER_COUNTER_UP; //向上计数
    timer_initpara.period         = 999;   //频率 1kHz = 72MHz / (71+1) / (999+1)
    timer_initpara.clockdivision   = TIMER_CKDIV_DIV1; //时钟分频 1
    timer_initpara.repetitioncounter = 0; //重复计数 0
    timer_init(TIMER1, &timer_initpara); //初始化配置

    /* TIMER1_CH0、TIMER1_CH1 通道配置 */
    timer_ocintpara.outputstate = TIMER_CCX_ENABLE; //使能输出比较通道
    timer_ocintpara.ocpolarity  = TIMER_OC_POLARITY_HIGH; //输出极性高

    /* TIMER1_CH0、TIMER1_CH1 配置 */
}
```

```

timer_channel_output_config(TIMER1, TIMER_CH_0, &timer_ocintpara);
timer_channel_output_config(TIMER1, TIMER_CH_1, &timer_ocintpara);

/* 配置 TIMER1_CH0 的 PWM */
timer_channel_output_pulse_value_config(TIMER1, TIMER_CH_0, 199); //占空比 20%
timer_channel_output_mode_config(TIMER1, TIMER_CH_0, TIMER_OC_MODE_PWM0);
//PWM0 输出模式
timer_channel_output_shadow_config(TIMER1, TIMER_CH_0, TIMER_OC_SHADOW_DISABLE);
//禁能影子寄存器

/* 配置 TIMER1_CH1 的 PWM */
timer_channel_output_pulse_value_config(TIMER1, TIMER_CH_1, 499); //占空比 50%
timer_channel_output_mode_config(TIMER1, TIMER_CH_1, TIMER_OC_MODE_PWM0);
//PWM0 输出模式
timer_channel_output_shadow_config(TIMER1, TIMER_CH_1, TIMER_OC_SHADOW_DISABLE);
//禁能影子寄存器

/* 自动重装载使能 */
timer_auto_reload_shadow_enable(TIMER1);
/* 使能 TIMER1 */
timer_enable(TIMER1);
}

```

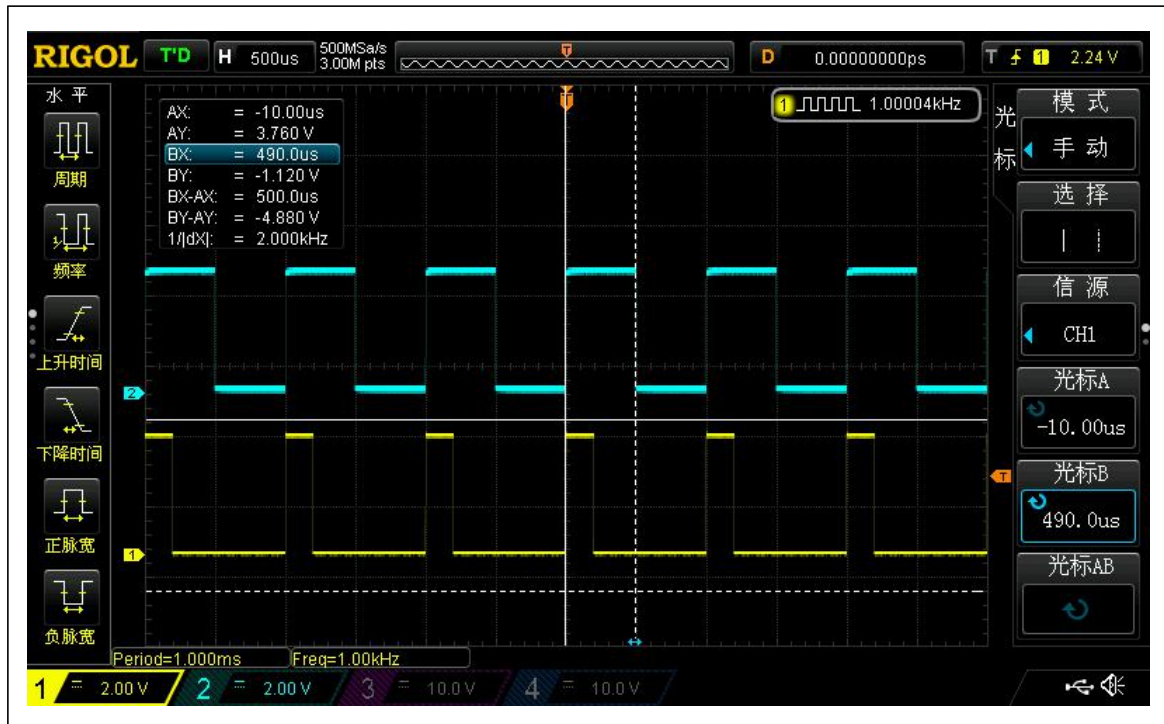
主函数如下：

```

int main(void)
{
    systick_config();
    LED_Init(); //初始化 LED
    TIMER1_GPIO_Config(); //TIMER1 的 GPIO 配置
    TIMER1_Config(); //TIMER1 配置
    while(1)
    {
        LED1_Toggle();
        delay_1ms(500);
    }
}

```

把程序下载到核心板后，LED1 闪烁。PA0（TIMER1_CH0）与 PA1（TIMER1_CH1）分别输出一个频率 1000Hz，占空比 20%和频率 1000Hz，占空比 50%的脉冲信号，使用示波器检测效果如下：



6、TIMER1 的单脉冲输出

6.1 TIMER1 的单脉冲配置

本小节介绍如何使用 TIMER1 实现输出单脉冲信号。例程：TIMER1_Single_Pulse，该例程位于 GD32F130G8_Examples\GD32F130G8_05_TIMER\GD32F130G8_TIMER1\ 目录下。例程使用 PA0（TIMER1_CH0）用作输入捕获功能与 PA1（TIMER1_CH1）PWM 输出功能，然后通过 KEY 按键产生一个触发信号给 PA0（把 PA0 与 PA15 短接到一起），然后 PA1 输出一个 3ms 的脉冲信号。TIMER1 的配置步骤如下：

- 1) PA0（TIMER1_CH0）与 PA1（TIMER1_CH1）GPIO 配置：
 - a. 使能 GPIOA 时钟
 - b. 配置 PA0 为复用功能
 - c. 配置 PA1 为复用功能
- 2) TIMER1 配置：
 - a. 定义 TIMER1 初始化结构体
 - b. 使能 TIMER1 外设时钟
 - c. TIMER1 配置：72 分频、边沿对齐、向上计数、周期 1000、时钟分频 1、重复计数 0
 - d. 使能 TIMER1 自动重载
 - e. TIMER1_CH1 通道配置：使能比较输出通道、输出极性高、空闲状态低
 - f. TIMER1_CH1 的 PWM 配置：脉冲时间 3ms、PWM1 模式、禁能影子寄存器
 - g. TIMER1_CH0 输入捕获配置：下降沿捕获、DIRECTTI 模式、分频 1、滤波系数 0
 - h. TIMER1 单脉冲模式

- i. 输入触发，CIO（TIMER1_CH0）外部触发
- j. TIMER1 从模式

6.2 例程介绍

本小节的例程中还使用 KEY 按键功能，这里不作介绍。TIMER1 的 GPIO 与模式配置在 timer1.c 文件下实现：

```
void TIMER1_GPIO_Config(void)
{
    /* 使能 GPIOA 外设时钟 */
    rcu_periph_clock_enable(RCU_GPIOA);
    /* 配置 PA0(TIMER1_CH0) 用作复用功能 */
    gpio_mode_set(GPIOA, GPIO_MODE_AF, GPIO_PUPD_NONE, GPIO_PIN_0);
    gpio_output_options_set(GPIOA, GPIO_OTYPE_PP, GPIO_OSPEED_50MHZ, GPIO_PIN_0);
    gpio_af_set(GPIOA, GPIO_AF_2, GPIO_PIN_0);
    /* 配置 PA1(TIMER1_CH1) 用作复用功能 */
    gpio_mode_set(GPIOA, GPIO_MODE_AF, GPIO_PUPD_NONE, GPIO_PIN_1);
    gpio_output_options_set(GPIOA, GPIO_OTYPE_PP, GPIO_OSPEED_50MHZ, GPIO_PIN_1);
    gpio_af_set(GPIOA, GPIO_AF_2, GPIO_PIN_1);
}

void TIMER1_Config(void)
{
    /* 定义 TIMER 初始化配置结构体 */
    timer_oc_parameter_struct timer_ocinitpara;
    timer_parameter_struct timer_initpara;
    timer_ic_parameter_struct timer_icinitpara;

    /* 使能 TIMER1 外设时钟 */
    rcu_periph_clock_enable(RCU_TIMER1);

    /* TIMER1 配置 */
    timer_deinit(TIMER1);
    timer_initpara.prescaler      = 3; //4 分频 = 72M/4 = 18MHz 时钟频率
    timer_initpara.alignedmode    = TIMER_COUNTER_EDGE; //边沿对齐
    timer_initpara.counterdirection = TIMER_COUNTER_UP; //向上计数
    timer_initpara.period         = 65535; //周期 65536
    timer_initpara.clockdivision   = TIMER_CKDIV_DIV1; //时钟分频 1
    timer_initpara.repetitioncounter = 0; //重复计数 0
}
```

```

timer_init(TIMER1,&timer_initpara); //初始化配置

/* 自动重装载禁能 */
timer_auto_reload_shadow_disable(TIMER1);

/* TIMER1_CH1 通道配置 */
timer_ocinitpara.outputstate = TIMER_CCX_ENABLE; //使能比较输出通道
timer_ocinitpara.ocpolarity = TIMER_OC_POLARITY_HIGH; //输出极性高
timer_ocinitpara.ocidlestate = TIMER_OC_IDLE_STATE_LOW; //空闲低
timer_channel_output_config(TIMER1, TIMER_CH_1, &timer_ocinitpara);

/* TIMER1_CH1 PWM 配置 */
timer_channel_output_pulse_value_config(TIMER1, TIMER_CH_1, 11535); //单脉冲时间 =
                                                                    (65535-11535)/18MHz = 3ms
timer_channel_output_mode_config(TIMER1, TIMER_CH_1,
                                TIMER_OC_MODE_PWM1); //PWM1 模式
timer_channel_output_shadow_config(TIMER1, TIMER_CH_1, TIMER_OC_SHADOW_DISABLE);
                                                                    //禁能影子寄存器

/* TIMER1_CH0 输入捕获配置 */
timer_icinitpara.icpolarity = TIMER_IC_POLARITY_FALLING; //下降沿捕获
timer_icinitpara.icselection = TIMER_IC_SELECTION_DIRECTTI; //DIRECTTI 模式
timer_icinitpara.icprescaler = TIMER_IC_PSC_DIV1; //分频 1
timer_icinitpara.icfilter = 0x00; //滤波系数 0
timer_input_capture_config(TIMER1, TIMER_CH_0, &timer_icinitpara);

/* TIMER1 单脉冲模式 */
timer_single_pulse_mode_config(TIMER1, TIMER_SP_MODE_SINGLE);
/* TIMER1 输入触发 : CIO 外部触发 */
timer_input_trigger_source_select(TIMER1, TIMER_SMCFG_TRGSEL_CIOFE0);
/* 从模式选择 : TIMER1 */
timer_slave_mode_select(TIMER1, TIMER_SLAVE_MODE_EVENT);

}

```

主函数如下：

```

int main(void)
{
    KEY_Init(); //初始化 KEY
    TIMER1_GPIO_Config(); //TIMER1 的 GPIO 配置
}

```

```
TIMER1_Config();    //TIMER1 配置
```

```
while(1)
```

```
{
```

```
    ;
```

```
}
```

```
}
```

把程序下载到核心板后，将 PA0（TIMER1_CH0）与 PA15（KEY）短接。当 KEY 按下时，PA1（TIMER1_CH1）就会产生一个 3ms 的高电平脉冲，使用示波器检测效果如下：



7、TIMER2 的输入捕获

7.1 TIMER2 的输入捕获配置

本小节介绍如何使用 TIMER2 实现输入捕获功能。例程：TIMER2_Input_Capture，该例程位于 GD32F130G8_Examples\GD32F130G8_05_TIMER\GD32F130G8_TIMER2\ 目录下。例程使用 PA6（TIMER2_CH0）用作输入捕获功能，然后通过 PA8（CK_OUT）输出内部 40KHz 的时钟脉冲，将 PA6 与 PA8 短接实现使用 PA6（TIMER2_CH0）来测量脉冲信号，然后使用串口打印出来（在 TIMER2 的中断中计算并打印）。TIMER2 的配置步骤如下：

- 1) PA6（TIMER2_CH0）GPIO 配置：
 - a. 使能 GPIOA 时钟
 - b. 配置 PA6 为复用功能

- 2) TIMER2 配置：
 - a. 定义 TIMER2 初始化结构体
 - b. TIMER2 中断向量配置
 - c. 使能 TIMER2 外设时钟
 - d. TIMER2 配置：72 分频、边沿对齐、向上计数、周期 65536、时钟分频 1、重复计数 0
 - e. TIMER2_CH0 的输入捕获配置：上升沿捕获、DIRECTTI 模式、分频 1、滤波系数 0
 - f. 自动重装载使能
 - g. 清除 TIMER2_CH0 中断标志
 - h. 使能 TIMER2_CH0 中断
 - i. 使能 TIMER2

7.2 例程介绍

本例程使用到了 USART0、CK_OUT、TIMER2 等外设功能，这里主要讲解 TIME2 相关。TIMER2 的 GPIO 与模式配置在 timer2.c 文件中实现：

```
void TIMER2_GPIO_Config(void)
{
    /* 使能 GPIOA 外设时钟 */
    rcu_periph_clock_enable(RCU_GPIOA);

    /* 配置 PA6(TIMER2 CH0) 用作复用功能 */
    gpio_mode_set(GPIOA, GPIO_MODE_AF, GPIO_PUPD_NONE, GPIO_PIN_6);
    gpio_output_options_set(GPIOA, GPIO_OTYPE_PP, GPIO_OSPEED_50MHZ, GPIO_PIN_6);
    gpio_af_set(GPIOA, GPIO_AF_1, GPIO_PIN_6);
}

void TIMER2_Config(void)
{
    /* 定义 TIMER 初始化配置结构体 */
    timer_ic_parameter_struct timer_icinitpara;
    timer_parameter_struct timer_initpara;

    /* 中断向量配置 */
    nvic_priority_group_set(NVIC_PRIGROUP_PRE1_SUB3);
    nvic_irq_enable(TIMER2_IRQn, 1, 1);

    /* 使能 TIMER2 外设时钟 */
    rcu_periph_clock_enable(RCU_TIMER2);
```

```

/* TIMER2 配置 */
timer_deinit(TIMER2);
timer_initpara.prescaler      = 71;    //72 分频 = 72M/72 = 1mHz 时钟频率
timer_initpara.alignedmode    = TIMER_COUNTER_EDGE; //边沿对齐
timer_initpara.counterdirection = TIMER_COUNTER_UP; //向上计数
timer_initpara.period         = 65535; //周期 65536
timer_initpara.clockdivision   = TIMER_CKDIV_DIV1; //时钟分频 1
timer_initpara.repetitioncounter = 0; //重复计数 0
timer_init(TIMER2, &timer_initpara); //初始化配置

/* TIMER2_CH0 配置为输入捕获 */
timer_icinitpara.icpolarity = TIMER_IC_POLARITY_RISING; //上升沿捕获
timer_icinitpara.icselection = TIMER_IC_SELECTION_DIRECTTI; //DIRECTTI 模式
timer_icinitpara.icprescaler = TIMER_IC_PSC_DIV1; //分频 1
timer_icinitpara.icfilter    = 0x0; //滤波系数 0
timer_input_capture_config(TIMER2, TIMER_CH_0, &timer_icinitpara);

/* 自动重载使能 */
timer_auto_reload_shadow_enable(TIMER2);
/* 清除 TIMER2_CH0 的中断标志位 */
timer_interrupt_flag_clear(TIMER2, TIMER_INT_CH0);
/* 使能 TIMER2_CH0 中断 */
timer_interrupt_enable(TIMER2, TIMER_INT_CH0);
/* 使能 TIMER2 */
timer_enable(TIMER2);
}

```

捕获的处理在 gd32f1x0_it.c 文件下的中断处理函数中实现，相关变量的定义也在 gd32f1x0_it.c 文件下：

```

__IO uint16_t readvalue1 = 0, readvalue2 = 0; //定义第一次捕获与第二次捕获变量
__IO uint16_t ccnumber = 0; //定义捕获标志位
__IO uint32_t count = 0; //定义计数器值变量
__IO float fre = 0; //定义频率变量

void TIMER2_IRQHandler(void)
{
    /* TIMER2_CH0 产生中断 */
    if((SET == timer_interrupt_flag_get(TIMER2, TIMER_INT_FLAG_CH0))
    {
        timer_interrupt_flag_clear(TIMER2, TIMER_INT_FLAG_CH0); //清除中断标志位
    }
}

```



```

        if(0 == ccnumber)    //读取第一次捕获值
        {
            readvalue1 = timer_channel_capture_value_register_read(TIMER2,
                                                                    TIMER_CH_0);

            ccnumber = 1;
        }
        else if(1 == ccnumber)    //读取第二次捕获值
        {
            /* read channel 0 capture value */
            readvalue2 = timer_channel_capture_value_register_read(TIMER2,
                                                                    TIMER_CH_0);

            if(readvalue2 > readvalue1)    //计算两次捕获的计数器差值（计数器没有溢出）
                count = (readvalue2 - readvalue1);
            else //计算两次捕获的计数器差值（计数器溢出）
                count = ((0xFFFF - readvalue1) + readvalue2);

            fre = (float)1000000 / count;    //计算频率
            /* 打印捕获的两次值 */
            printf("\r\nthe value1 is %d, the value2 is %d\n", readvalue1, readvalue2);
            printf("\r\nthe count is %d\n", count); //打印两次捕获的计数器差值
            printf("\r\nthe frequency is %f\r\n", fre);    //打印频率
            ccnumber = 0;    //标志位清 0
        }
    }
}

```

主函数如下：

```

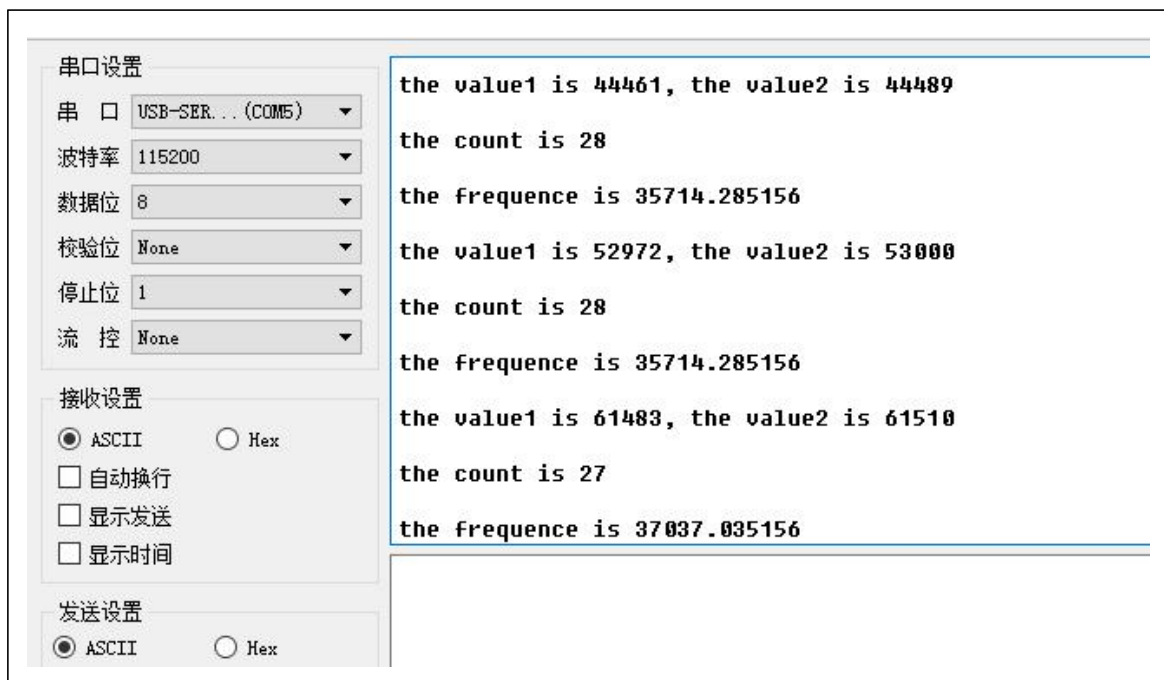
int main(void)
{
    rcu_osci_on(RCU_IRC40K);    //打开内部 40KHz 时钟
    while(rcu_osci_stab_wait(RCU_IRC40K) == ERROR); //等待时钟稳定
    CK_OUT_Config(RCU_CKOUTSRC_IRC40K, RCU_CKOUT_DIV1); //输出内部 40KHz、1 分频时钟

    LED_Init();    //初始化 LED
    USART0_Init();    //初始化 USART0
    TIMER2_GPIO_Config();    //TIMER2 的 GPIO 配置
    TIMER2_Config();    //TIMER2 配置
    LED1_ON();    //LED1 亮
}

```

```
while(1)
{
    ;
}
}
```

把程序下载到核心板后，LED1 亮，将 PA6（TIMER2_CH0）与 PA8（CK_OUT）短接。使用一个 USB 转 TTL 模块与核心板进行连接，连接线序为：模块的 TX-->PA10、模块的 RX-->PA9、模块的 GND-->GND。连接好后把 USB 转 TTL 模块接入电脑，打开电脑上位机（串口调试助手），上位机配置为：波特率 115200、数据位 8 位、校验位无、停止位 1 位、流控无。然后选择正确串口（不了解的可以百度一下：如何查看串口号），把串口打开，可以看到串口打印出捕获的数据、频率等：



8、TIMER2 的 PWM 捕获

8.1 TIMER2 的 PWM 捕获配置

本小节介绍如何使用 TIMER2 实现 PWM 捕获功能。例程：TIMER2_PWM_Input_Capture，该例程位于 GD32F130G8_Examples\GD32F130G8_05_TIMER\GD32F130G8_TIMER2\目录下。例程使用 PA6（TIMER2_CH0）用作输入捕获功能，然后通过 PA8（CK_OUT）输出内部 40KHz 的时钟脉冲，将 PA6 与 PA8 短接实现使用 PA6（TIMER2_CH0）来测量 PWM 信号的频率与周期，然后使用串口打印出来（在 TIMER2 的中断中计算并打印），其中使用了 PA6（TIMER2_CH0）和 PA7（TIMER2_CH0）两个通道的 CH0CV 与 CH1CV 寄存器来计算频率和周期。TIMER2 的配置步骤如下：

- 1) PA6(TIMER2_CH0)和 PA7(TIMER2_CH1)GPIO 配置：
 - a. 使能 GPIOA 时钟

- b. 配置 PA6 为复用功能
- c. 配置 PA7 为复用功能
- 2) TIMER2 配置：
 - a. 定义 TIMER2 初始化结构体
 - b. TIMER2 中断向量配置
 - c. 使能 TIMER2 外设时钟
 - d. TIMER2 配置：72 分频、边沿对其、向上计数、周期 65536、时钟分频 1、重复计数 0
 - e. TIMER2_CH0 的 PWM 捕获配置：上升沿捕获、//DIRECTTI 模式、//分频 1、滤波系数 0
 - f. TIMER2 从模式配置
 - g. TIMER2 从模式使能
 - h. 自动重装载使能
 - i. 清除 TIMER2_CH0 中断标志位
 - j. 使能 TIMER2_CH0 中断
 - k. 使能 TIMER2

8.2 例程介绍

本例程使用到了 USART0、CK_OUT、TIMER2 等外设功能，这里主要讲解 TIME2 相关。TIMER2 的 GPIO 与模式配置在 timer2.c 文件中实现：

void TIMER2_GPIO_Config(void)

```
{
    /* 使能 GPIOA 外设时钟 */
    rcu_periph_clock_enable(RCU_GPIOA);
    /* 配置 PA6(TIMER2_CH0) 用作复用功能 */
    gpio_mode_set(GPIOA, GPIO_MODE_AF, GPIO_PUPD_NONE, GPIO_PIN_6);
    gpio_output_options_set(GPIOA, GPIO_OTYPE_PP, GPIO_OSPEED_50MHZ, GPIO_PIN_6);
    gpio_af_set(GPIOA, GPIO_AF_1, GPIO_PIN_6);
    /* 配置 PA7(TIMER2_CH1) 用作复用功能 */
    gpio_mode_set(GPIOA, GPIO_MODE_AF, GPIO_PUPD_NONE, GPIO_PIN_7);
    gpio_output_options_set(GPIOA, GPIO_OTYPE_PP, GPIO_OSPEED_50MHZ, GPIO_PIN_7);
    gpio_af_set(GPIOA, GPIO_AF_1, GPIO_PIN_7);
}
```

void TIMER2_Config(void)

```
{
    /* 定义 TIMER2 初始化配置结构体 */
    timer_ic_parameter_struct timer_icinitpara;
    timer_parameter_struct timer_initpara;
```

```

/* 中断向量配置 */
nvic_priority_group_set(NVIC_PRIGROUP_PRE1_SUB3);
nvic_irq_enable(TIMER2_IRQn, 1, 1);

/* 使能 TIMER2 外设时钟 */
rcu_periph_clock_enable(RCU_TIMER2);

/* TIMER2 配置 */
timer_deinit(TIMER2);
timer_initpara.prescaler      = 71;    //72 分频 = 72M/72 = 1mHz 时钟频率
timer_initpara.alignedmode    = TIMER_COUNTER_EDGE; //边沿对齐
timer_initpara.counterdirection = TIMER_COUNTER_UP; //向上计数
timer_initpara.period         = 65535; //周期 65536
timer_initpara.clockdivision   = TIMER_CKDIV_DIV1; //时钟分频 1
timer_initpara.repetitioncounter = 0; //重复计数 0
timer_init(TIMER2, &timer_initpara); //初始化配置

/* TIMER2_CH0 配置为 PWM 输入捕获 */
timer_icinitpara.icpolarity = TIMER_IC_POLARITY_RISING; //上升沿捕获
timer_icinitpara.icselection = TIMER_IC_SELECTION_DIRECTTI; //DIRECTTI 模式
timer_icinitpara.icprescaler = TIMER_IC_PSC_DIV1; //分频 1
timer_icinitpara.icfilter    = 0x0; //滤波系数 0
timer_input_pwm_capture_config(TIMER2, TIMER_CH_0, &timer_icinitpara);

/* TIMER2 从模式配置 */
timer_input_trigger_source_select(TIMER2, TIMER_SMCFG_TRGSEL_CIOFE0);
timer_slave_mode_select(TIMER2, TIMER_SLAVE_MODE_RESTART);

/* 从模式使能 */
timer_master_slave_mode_config(TIMER2, TIMER_MASTER_SLAVE_MODE_ENABLE);
/* 自动重装载使能 */
timer_auto_reload_shadow_enable(TIMER2);
/* 清除 TIMER2_CH0 的中断标志位 */
timer_interrupt_flag_clear(TIMER2, TIMER_INT_CH0);
/* 使能 TIMER2_CH0 中断 */
timer_interrupt_enable(TIMER2, TIMER_INT_CH0);
/* 使能 TIMER2 */
timer_enable(TIMER2);
}

```

捕获的处理在 gd32f1x0_it.c 文件下的中断处理函数中实现，相关变量的定义也在 gd32f1x0_it.c 文件下：

```
__IO uint32_t ic1value = 0, ic2value = 0; // 定义存储捕获值变量
__IO uint16_t dutycycle = 0; // 定义存储周期变量
__IO float frequency = 0; // 定义存储频率变量

void TIMER2_IRQHandler(void)
{
    if (SET == timer_interrupt_flag_get(TIMER2, TIMER_INT_FLAG_CH0))
    {
        /* 清楚 TIMER2_CH0 中断标志位 */
        timer_interrupt_flag_clear(TIMER2, TIMER_INT_FLAG_CH0);

        /* 读取 TIMER2_CH0 捕获值 (CH0CV) */
        ic1value = timer_channel_capture_value_register_read(TIMER2, TIMER_CH_0) + 1;

        if (0 != ic1value)
        {
            /* 读取 TIMER2_CH0 捕获值 (CH1CV) */
            ic2value = timer_channel_capture_value_register_read(TIMER2, TIMER_CH_1) + 1;

            dutycycle = (ic2value * 100) / ic1value; // 计算周期
            frequency = (float)1000000 / ic1value; // 计算频率

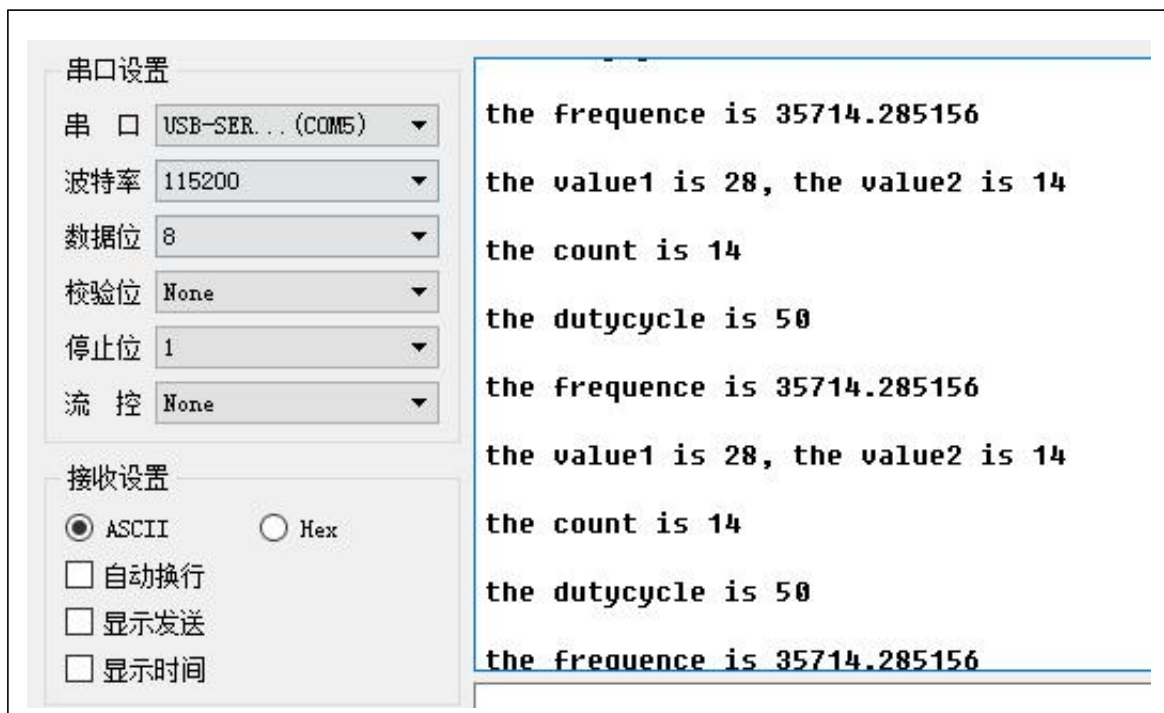
            /* 打印数据 */
            printf("\r\nthe value1 is %d, the value2 is %d\n", ic1value, ic2value);
            printf("\r\nthe count is %d\n", (ic1value - ic2value));
            printf("\r\nthe dutycycle is %d\n", dutycycle);
            printf("\r\nthe frequency is %f\r\n", frequency);
        }
        else
        {
            dutycycle = 0; // 清零
            frequency = 0; // 清零
        }
    }
}
```

主函数如下：

```
int main(void)
{
    rcu_osci_on(RCU_IRC40K);    //打开内部 40KHz 时钟
    while(rcu_osci_stab_wait(RCU_IRC40K) == ERROR); //等待时钟稳定
    CK_OUT_Config(RCU_CKOUTSRC_IRC40K, RCU_CKOUT_DIV1); //输出内部 40KHz、1 分频时钟

    LED_Init();    //初始化 LED
    USART0_Init(); //初始化 USART0
    TIMER2_GPIO_Config(); //TIMER2 的 GPIO 配置
    TIMER2_Config();    //TIMER2 配置
    LED1_ON(); //LED1 亮
    while(1)
    {
        ;
    }
}
```

把程序下载到核心板后，LED1 亮，将 PA6（TIMER2_CH0）与 PA8（CK_OUT）短接。使用一个 USB 转 TTL 模块与核心板进行连接，连接线序为：模块的 TX-->PA10、模块的 RX-->PA9、模块的 GND-->GND。连接好后把 USB 转 TTL 模块接入电脑，打开电脑上位机（串口调试助手），上位机配置为：波特率 115200、数据位 8 位、校验位无、停止位 1 位、流控无。然后选择正确串口（不了解的可以百度一下：如何查看串口号），把串口打开，可以看到串口打印出捕获的数据、频率、周期等：



第六章 I2C 应用

I2C（内部集成电路总线接口），I2C 模块提供了符合工业标准的两线串行接口，可用于 MCU 和外部 I2C 设备的通讯。I2C 总线使用两条串行线：串行数据线 SDA 和串行时钟线 SCL。本章从以下部分介绍 I2C 相关内容：

- I2C 简介
- I2C 与 0.96 寸 OLED 模块通讯

1、I2C 简介

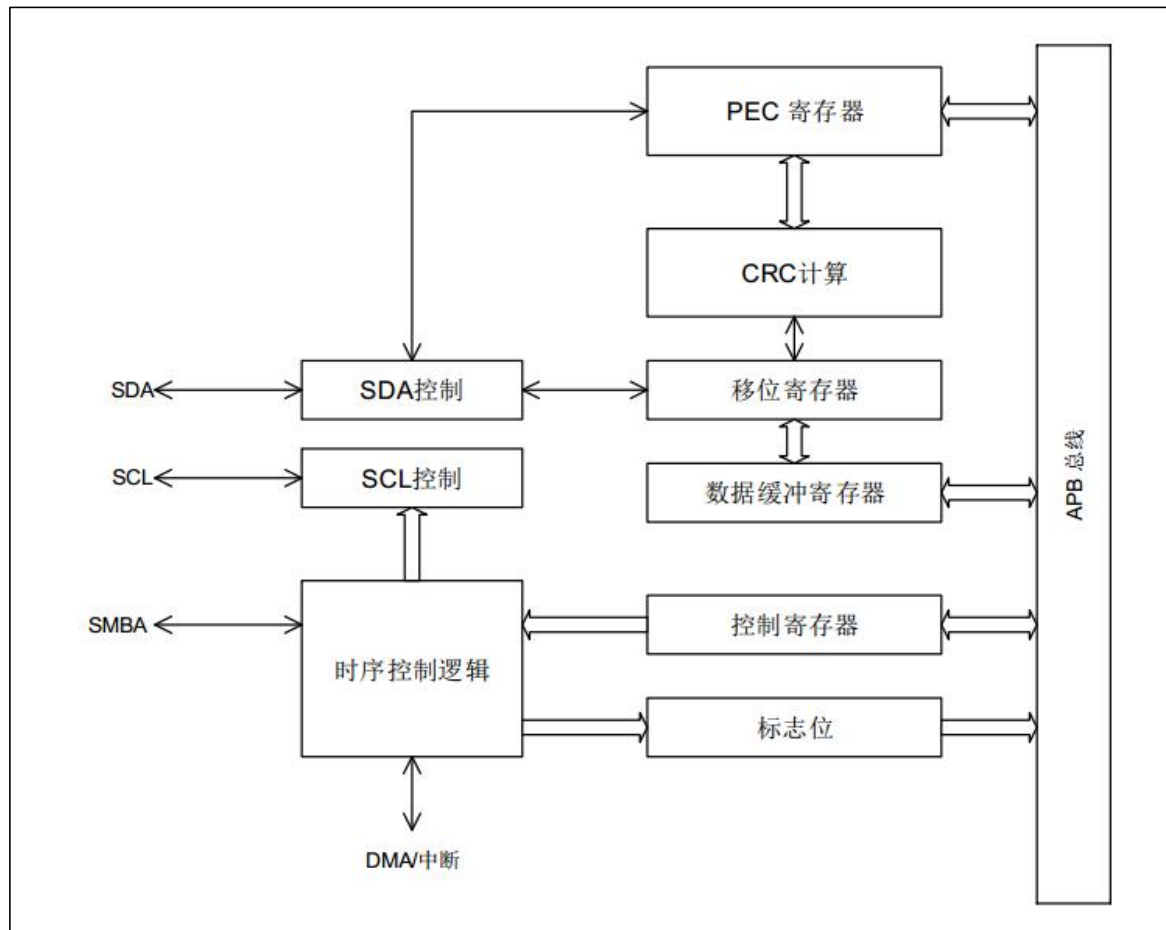
I2C 接口模块实现了 I2C 协议的标速模式和快速模式，具备 CRC 计算和校验功能、支持 SMBus(系统管理总线)和 PMBus (电源管理总线)，此外还支持多主机 I2C 总线架构。I2C 接口模块也支持 DMA 模式，可有效减轻 CPU 的负担。I2C 的主要特征如下：

- ❖ 并行总线至 I2C 总线协议的转换及接口
- ❖ 同一接口既可实现主机功能又可实现从机功能
- ❖ 主从机之间的双向数据传输
- ❖ 支持 7 位和 10 位的地址模式和广播寻址
- ❖ 支持 I2C 多主机模式
- ❖ 支持标速(最高 100 kHz)和快速(最高 400 kHz)
- ❖ 从机模式下可配置的 SCL 主动拉低
- ❖ 支持 DMA 模式
- ❖ 兼容 SMBus 2.0 和 PMBus
- ❖ 两个中断：字节成功发送中断和错误事件中断
- ❖ 可选择的 PEC (报文错误校验)生成和校验

I2C 模块有两条接口线：串行数据 SDA 线和串行时钟 SCL 线。连接到总线上的设备通过这两根线互相传递信息。SDA 和 SCL 都是双向线，通过一个电流源或者上拉电阻接到电源正极。当总线空闲时，两条线都是高电平。连接到总线的设备输出极必须是开漏或者开集，以提供线与功能。I2C 总线上的数据在标准模式下可以达到 100Kbit/s，在快速模式下可以达到 400Kbit/s。由于 I2C 总线上可能会连接不同工艺的设备，逻辑‘0’和逻辑‘1’的电平并不是固定的，取决于 VDD 的实际电平。

I2C 模块在时钟信号的高电平期间 SDA 线上的数据必须稳定。只有在时钟信号 SCL 变低的时候数据线 SDA 的电平状态才能跳变，每个数据比特传输需要一个时钟脉冲。

I2C 的模块框图如下：



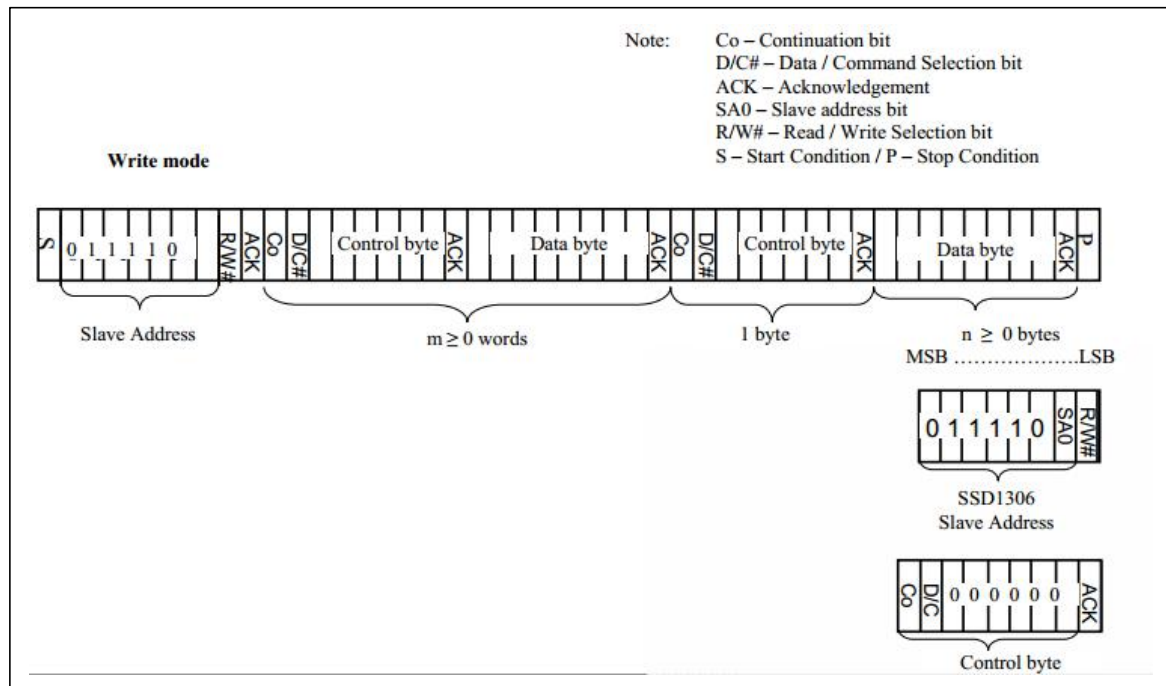
2、I2C 与 0.96 寸 OLED 模块通讯

2.1 I2C 的配置

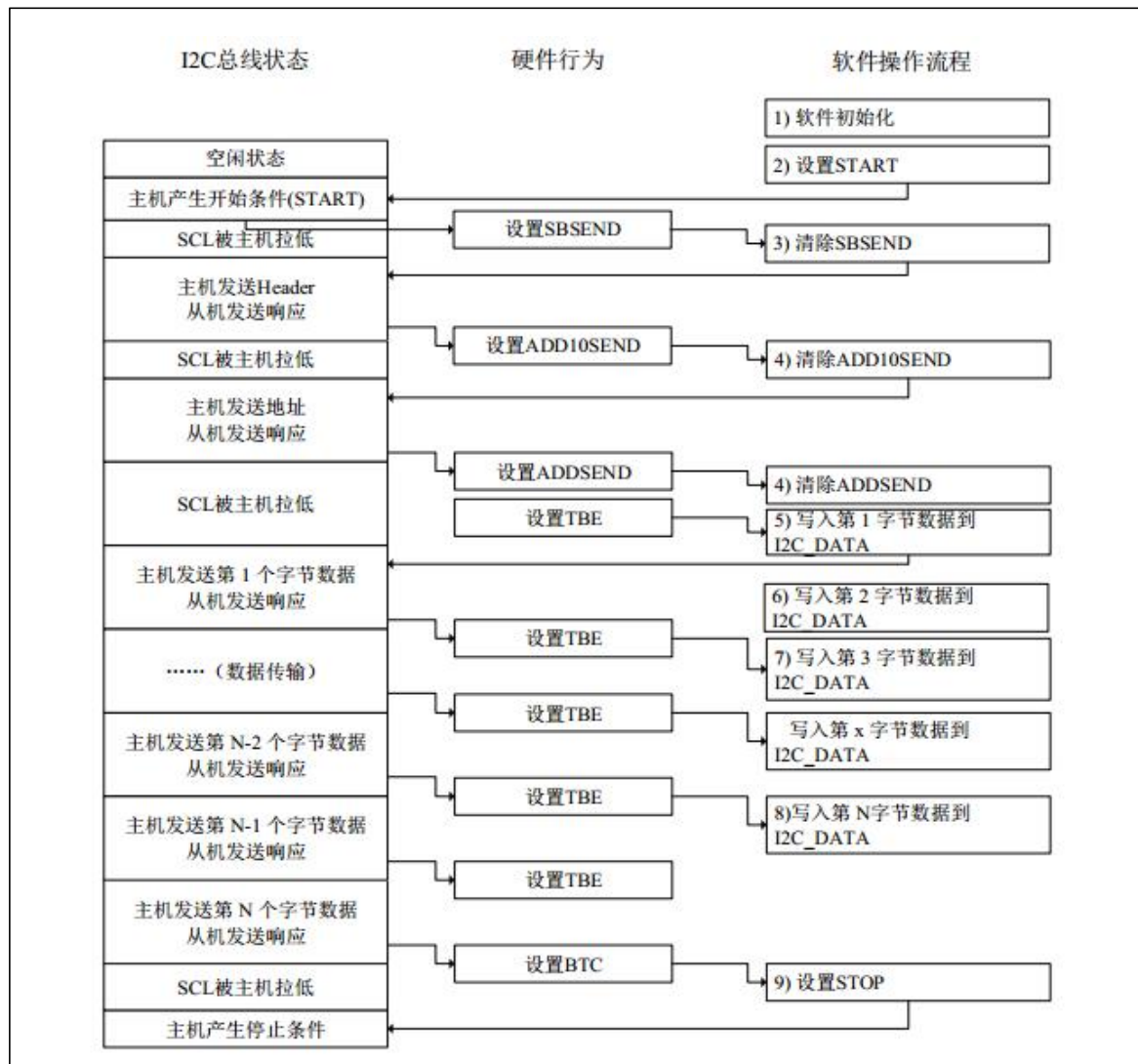
GD32F130G8U6 有两个 I2C 模块，本小节介绍如何使用 I2C0 实现与 0.96 寸 OLED 模块（默认使用技新的 4Pin-0.96 寸 OLED 模块）通讯并显示字符串。例程：I2C_OLED，该例程位于 GD32F130G8_Examples\GD32F130G8_06_I2C\目录下。例程中 I2C0 部分可分为 I2C 的初始化、I2C 与 0.96 寸 OLED 模块通讯时序两部分，I2C0 初始化步骤如下：

- 1) 使能 I2C0 相关的 GPIO（GPIOB）时钟、使能 I2C0 外设时钟
- 2) 配置 PB6（I2C0_SCL）与 PB7（I2C0_SDA）引脚为复用功能
- 3) I2C0 的时钟配置
- 4) I2C0 的地址配置
- 5) 使能 I2C0
- 6) 使能 I2C0 应答

I2C0 与 OLED 通讯的时序部分，通讯中 GD32F130G8U6 的 I2C0 模块充当主机，0.96 寸 OLED 模块充当从机，并且通讯过程只有主机向从机发送这一个流程（OLED 模块 IIC 通讯不支持读）。0.96 寸 OLED 模块的 IIC 通讯时序如下：



GD32F130G8U6 的 I2C0 模块作为主机发送模式的流程如下：



2.2 例程介绍

I2C0 的初始化在 i2c.c 文件中实现：

```
void I2C0_Init(void)
{
    /* 使能 I2C0 所在 GPIO 外设时钟与 I2C0 外设时钟 */
    rcu_periph_clock_enable(RCU_GPIOB);
    rcu_periph_clock_enable(I2C0_CLOCK);

    /* 配置 I2C0_SCL 与 I2C0_SDA 引脚为复用功能 */
    gpio_af_set(I2C0_SCL_PORT, GPIO_AF_1, I2C0_SCL_PIN);
    gpio_af_set(I2C0_SDA_PORT, GPIO_AF_1, I2C0_SDA_PIN);

    gpio_mode_set(I2C0_SCL_PORT, GPIO_MODE_AF, GPIO_PUPD_PULLUP, I2C0_SCL_PIN);
    gpio_output_options_set(I2C0_SCL_PORT, GPIO_OTYPE_OD,
                            GPIO_OSPEED_50MHZ, I2C0_SCL_PIN);
    gpio_mode_set(I2C0_SDA_PORT, GPIO_MODE_AF, GPIO_PUPD_PULLUP, I2C0_SDA_PIN);
    gpio_output_options_set(I2C0_SDA_PORT, GPIO_OTYPE_OD,
                            GPIO_OSPEED_50MHZ, I2C0_SDA_PIN);

    /* I2C0 时钟配置 */
    i2c_clock_config(I2C0, 100000, I2C_DTCY_2);
    /* I2C0 地址配置 */
    i2c_mode_addr_config(I2C0, I2C_I2CMODE_ENABLE, I2C_ADDFORMAT_7BITS, 0x12);
    /* 使能 I2C0 */
    i2c_enable(I2C0);
    /* 使能应答 */
    i2c_ack_config(I2C0, I2C_ACK_ENABLE);
}
```

相关的时钟以及端口在 i2c.h 文件中定义：

```
#define I2C_SLAVE_ADDRESS7      0x78    //从机地址
#define I2C0_CLOCK              RCU_I2C0

#define I2C0_SCL_PORT           GPIOB
#define I2C0_SCL_PIN            GPIO_PIN_6
#define I2C0_SDA_PORT           GPIOB
#define I2C0_SDA_PIN            GPIO_PIN_7
```

I2C0 与 OLED 模块的通讯部分代码在 oled.c 文件中实现，其中 OLED 模块的从机地址是 0x78，该地址在 i2c.h 中也有定义。oled.c 文件中的 i2c 通讯代码如下：

```
void Write_OLED_Command(unsigned char IIC_Command)
{
    /* 等待总线空闲 */
    while(i2c_flag_get(I2C0, I2C_FLAG_I2CBSY));
    /* 发送起始条件 */
    i2c_start_on_bus(I2C0);
    /* 等待 SBSEND 位置位 */
    while(!i2c_flag_get(I2C0, I2C_FLAG_SBSEND));
    /* 发送从机地址 */
    i2c_master_addressing(I2C0, 0x78, I2C_TRANSMITTER);
    /* 等待 ADDSEND 位置位 */
    while(!i2c_flag_get(I2C0, I2C_FLAG_ADDSEND));
    /* 清除 ADDSEND 位 */
    i2c_flag_clear(I2C0, I2C_FLAG_ADDSEND);
    /* 发送数据 */
    i2c_data_transmit(I2C0, 0x00);
    /* 等待发送数据缓存器空 */
    while(!i2c_flag_get(I2C0, I2C_FLAG_TBE));
    /* 发送数据 */
    i2c_data_transmit(I2C0, IIC_Command);
    /* 等待发送数据缓存器空 */
    while(!i2c_flag_get(I2C0, I2C_FLAG_TBE));
    /* 发送停止条件 */
    i2c_stop_on_bus(I2C0);
    while(I2C_CTL0(I2C0)&0x0200);
}
```

还有一个函数 Write_OLED_Data () 也是上面一样通讯协议，只是发送的第一个数据变为 0x40（上面发送的第一个数据是 0x00）：

```
void Write_OLED_Data(unsigned char IIC_Data)
{
    /* 等待总线空闲 */
    while(i2c_flag_get(I2C0, I2C_FLAG_I2CBSY));
    /* 发送起始条件 */
    i2c_start_on_bus(I2C0);
    /* 等待 SBSEND 位置位 */
    while(!i2c_flag_get(I2C0, I2C_FLAG_SBSEND));
    /* 发送从机地址 */
```

```

i2c_master_addressing(I2C0, 0x78, I2C_TRANSMITTER);
/* 等待 ADDSEND 位置位 */
while(!i2c_flag_get(I2C0, I2C_FLAG_ADDSEND));
/* 清除 ADDSEND 位 */
i2c_flag_clear(I2C0, I2C_FLAG_ADDSEND);
/* 发送数据 */
i2c_data_transmit(I2C0, 0x40);
/* 等待发送数据缓存器空 */
while(!i2c_flag_get(I2C0, I2C_FLAG_TBE));
/* 发送数据 */
i2c_data_transmit(I2C0, IIC_Data);
/* 等待发送数据缓存器空 */
while(!i2c_flag_get(I2C0, I2C_FLAG_TBE));
/* 发送停止条件 */
i2c_stop_on_bus(I2C0);
while(I2C_CTL0(I2C0)&0x0200);
}

```

主函数如下：

```

int main(void)
{
    systick_config(); //初始化滴答定时器
    LED_Init(); //LED 初始化
    I2C0_Init(); //I2C0 初始化

    OLED_Init(); //OLED 初始化
    OLED_Clear(); //OLED 清屏
    OLED_ShowString(30,0,"OLED TEST"); //显示 OELD TEST
    while(1)
    {
        LED1_Toggle();
        delay_1ms(500);
    }
}

```

把技新的 OLED 模块插入 P6 接口（或者：GND-->GND、VCC--3V3、SCL--PB6、SDA-->PB7），下载程序，可以看到 LED1 闪烁，OLED 显示：OELD TEST。效果如下：



第七章 SPI 应用

SPI（串行外设接口），SPI 模块提供了基于 SPI 协议的数据发送和接收功能，可以工作于主机或从机模式。SPI 接口支持具有硬件 CRC 计算和校验的全双工和单工模式。GD32F130xx 的 SPI 接口还支持单线配置下的主机和从机模式。本章从以下部分介绍 SPI 相关内容：

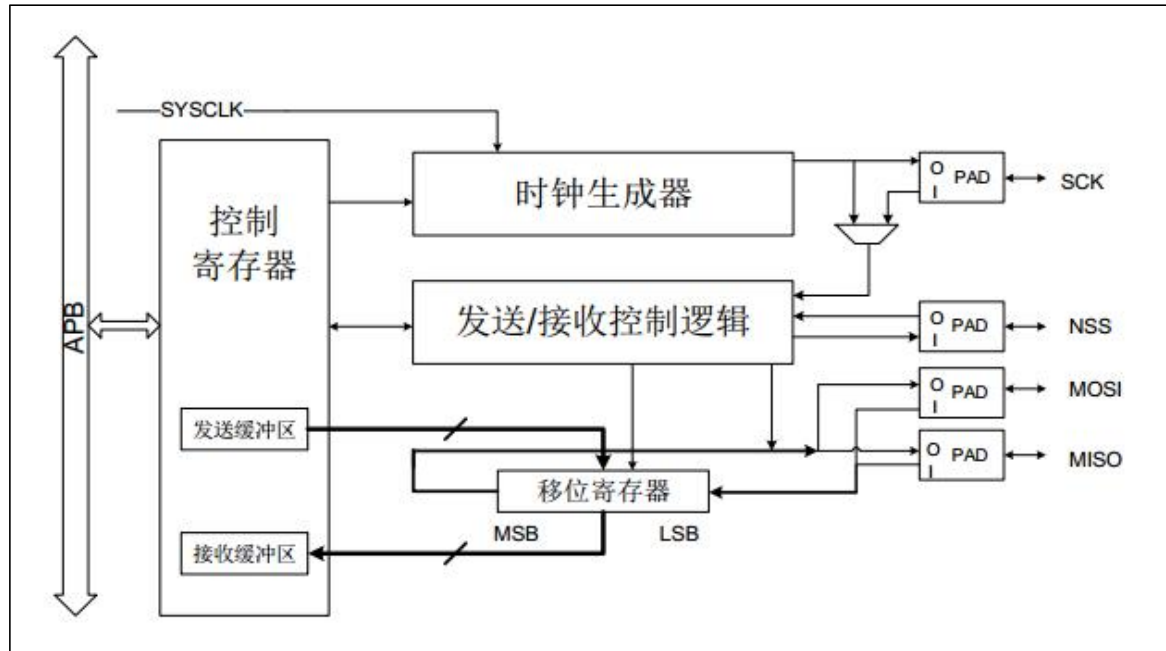
- SPI 简介
- SPI 与 0.96 寸 OLED 模块通讯

1、SPI 简介

SPI 的主要特征如下：

- ❖ 具有全双工和单工模式的主从操作
- ❖ 16 位宽度，独立的发送和接收缓冲区
- ❖ 8 位或 16 位数据帧格式
- ❖ 低位在前或高位在前的数据位顺序
- ❖ 软件和硬件 NSS 管理
- ❖ 硬件 CRC 计算、发送和校验
- ❖ 发送和接收支持 DMA 模式

SPI 的结构框图如下所示：



2、SPI 与 0.96 寸模块通讯

2.1 SPI 的配置

GD32F130G8U6 有两个 SPI 模块，本小节介绍如何使用 SPI0 实现与 0.96 寸 OLED 模块（默认使用技新的 7Pin-0.96 寸 OLED 模块）通讯并显示字符串。例程：SPI_OLED，该例程位于 GD32F130G8_Examples\GD32F130G8_07_SPI\目录下。SPI0 初始化步骤如下：

- 1) 定义 SPI 初始化配置结构体
- 2) 打开 SPI0 相关 GDIO 端口时钟（GPIOA）、打开 SPI0 外设时钟
- 3) 配置 SPI0 相关 IO 口为复用功能：SCK-->PA5、MISO-->PA6、MOSI-->PA7
- 4) SPI0 参数配置：
 - a. 全双工模式
 - b. SPI 主模式
 - c. 8 位数据结构
 - d. 时钟极性配置
 - e. NSS 软件控制
 - f. SPI 时钟 8 分频
 - g. 高位在前
- 5) 使能 SPI0

SPI0 读写一字节数据步骤如下：

- 1) 等待发送缓存器空
- 2) 发送数据
- 3) 等待接收缓存器非空
- 4) 接收数据
- 5) 返回接收的数据

2.2 例程介绍

SPI0 的初始化以及读写函数在 spi.c 文件中实现：

```
void SPI0_Init(void)
{
    /* 定义 spi 配置参数结构体 */
    spi_parameter_struct spi_init_struct;

    /* 打开 SPI0 的端口时钟与外设时钟 */
    rcu_periph_clock_enable(RCU_GPIOA);
    rcu_periph_clock_enable(RCU_SPI0);
```

```
/* SPI0 GPIO 配置: SCK/PA5, MISO/PA6, MOSI/PA7 */
gpio_af_set(GPIOA, GPIO_AF_0, GPIO_PIN_5 | GPIO_PIN_6 | GPIO_PIN_7);
gpio_mode_set(GPIOA, GPIO_MODE_AF, GPIO_PUPD_NONE, GPIO_PIN_5 | GPIO_PIN_6 |
              GPIO_PIN_7);

gpio_output_options_set(GPIOA, GPIO_OTYPE_PP, GPIO_OSPEED_50MHZ, GPIO_PIN_5 |
                        GPIO_PIN_6 | GPIO_PIN_7);

/* SPI0 参数配置 */
spi_init_struct.trans_mode          = SPI_TRANSMODE_FULLDUPLEX; //全双工模式
spi_init_struct.device_mode         = SPI_MASTER; //SPI 主模式
spi_init_struct.frame_size          = SPI_FRAME_SIZE_8BIT; //8 位数据结构
spi_init_struct.clock_polarity_phase = SPI_CLK_PL_HIGH_PH_2EDGE; //时钟极性配置
spi_init_struct.nss                 = SPI_NSS_SOFT; //NSS 软件控制
spi_init_struct.prescale            = SPI_PSC_8; //8 分频
spi_init_struct.endian              = SPI_ENDIAN_MSB; //高位在前
spi_init(SPI0, &spi_init_struct); //初始化配置

spi_enable(SPI0); //使能 SPI0
}
```

```
uint8_t SPI_RW_Byte(uint8_t data)
{
    uint8_t temp; //定义变量读取数据
    /* 等待发送缓存器空 */
    while(RESET == spi_i2s_flag_get(SPI0, SPI_FLAG_TBE));
    /* 发送数据 */
    spi_i2s_data_transmit(SPI0, data);
    /* 等待接收缓存器非空 */
    while(RESET == spi_i2s_flag_get(SPI0, SPI_FLAG_RBNE));
    /* 接收一字节数据 */
    temp = spi_i2s_data_receive(SPI0);
    return temp;
}
```

与 OLED 模块的通讯还需要 3 条信号线对 OLED 进行控制才能实现通讯，分别是 PA2-->RES、PA3-->DC、PA4-->CS（SPI 与 OLED 接的线序：PA5-->D0、PA7-->D1），在 oled.c 中初始化以及在 oled.h 中定义：

```
static void OLED_GPIO_Init(void)
{
```

```

/* 打开 GPIOA 时钟 */
rcu_periph_clock_enable(RCU_GPIOA);
/* 配置 RES、DC、CS 引脚为推挽输出 */
gpio_mode_set(GPIOA, GPIO_MODE_OUTPUT, GPIO_PUPD_NONE, GPIO_PIN_2 | GPIO_PIN_3 |
               GPIO_PIN_4);
gpio_output_options_set(GPIOA, GPIO_OTYPE_PP, GPIO_OSPEED_50MHZ, GPIO_PIN_2 |
                        GPIO_PIN_3 | GPIO_PIN_4);
/* 配置 RES、DC、CS 引脚输出高 */
gpio_bit_set(GPIOA, GPIO_PIN_2 | GPIO_PIN_3 | GPIO_PIN_4);
}

#define OLED_RES_GPIO_PORT      (GPIOA)
#define OLED_RES_GPIO_PINS     (GPIO_PIN_2)

#define OLED_DC_GPIO_PORT      (GPIOA)
#define OLED_DC_GPIO_PINS     (GPIO_PIN_3)

#define OLED_CS_GPIO_PORT      (GPIOA)
#define OLED_CS_GPIO_PINS     (GPIO_PIN_4)

#define OLED_RES_Set()          gpio_bit_set(OLED_RES_GPIO_PORT, OLED_RES_GPIO_PINS)
#define OLED_RES_Clr()          gpio_bit_reset(OLED_RES_GPIO_PORT, OLED_RES_GPIO_PINS)

#define OLED_DC_Set()           gpio_bit_set(OLED_DC_GPIO_PORT, OLED_DC_GPIO_PINS)
#define OLED_DC_Clr()           gpio_bit_reset(OLED_DC_GPIO_PORT, OLED_DC_GPIO_PINS)

#define SPI_NSS_HIGH()          gpio_bit_set(OLED_CS_GPIO_PORT, OLED_CS_GPIO_PINS)
#define SPI_NSS_LOW()           gpio_bit_reset(OLED_CS_GPIO_PORT, OLED_CS_GPIO_PINS)

```

主函数如下：

```

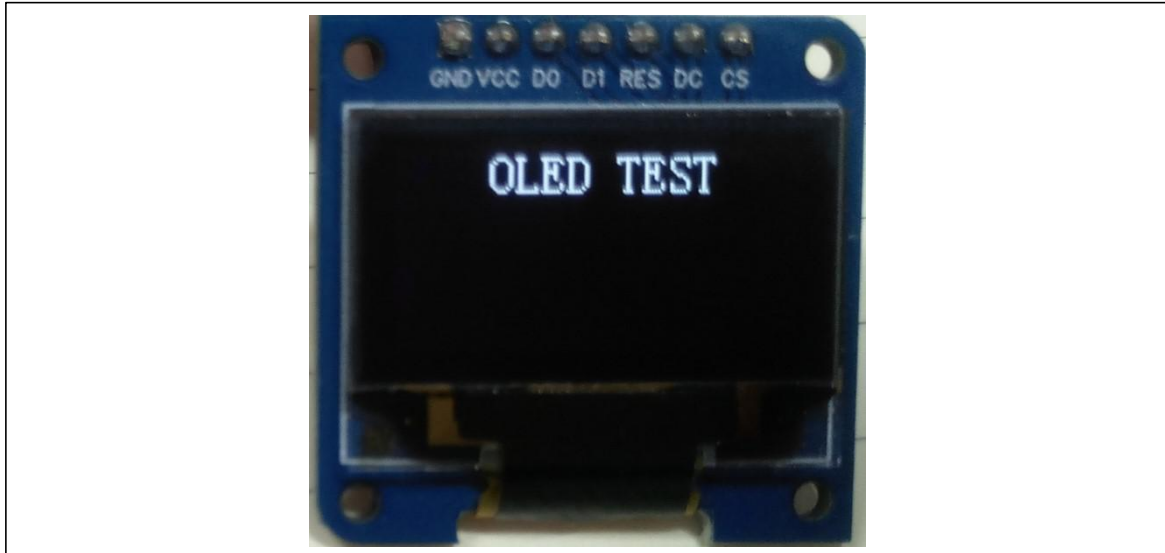
int main(void)
{
    systick_config(); //初始化滴答定时器
    LED_Init(); //LED 初始化
    SPI0_Init(); //SPI0 初始化
    OLED_Init(); //OLED 初始化
    OLED_Clear(); //OLED 清屏

    OLED_ShowString(30,0,"OLED TEST");//OELD 显示：OLED TEST
    while(1)

```

```
{  
    LED1_Toggle();  
    delay_1ms(500);  
}  
}
```

将技新的 OLED 模块与 GD32F130G8U6 核心板连接：GND-->GND、VCC--3V3、D0--PA5、D1-->PA7、RES-->PA2、DC-->PA3、CS-->PA4），下载程序，可以看到 LED1 闪烁，OLED 显示：OELD TEST。效果如下：



第八章 ADC 应用

ADC（模拟数字转换器），12 位 ADC 是一种采用逐次逼近方式的模拟数字转换器。它有 19 个多路复用通道（GD32F130G8U6 有 10 路 ADC 通道：ADC_IN0~ADC_IN9），可以测量来自 16 个外部通道，2 个内部通道和电池电压（VBAT）通道的模拟信号。模拟看门狗允许应用程序来检测输入电压是否超出用户设定的高低阈值。每个通道的 A/D 转换可以配置成单次、连续、扫描或间断转换模式。ADC 转换的结果可以按照左对齐或右对齐的方式存储在 16 位数据寄存器中。本章从以下部分介绍 ADC 相关内容：

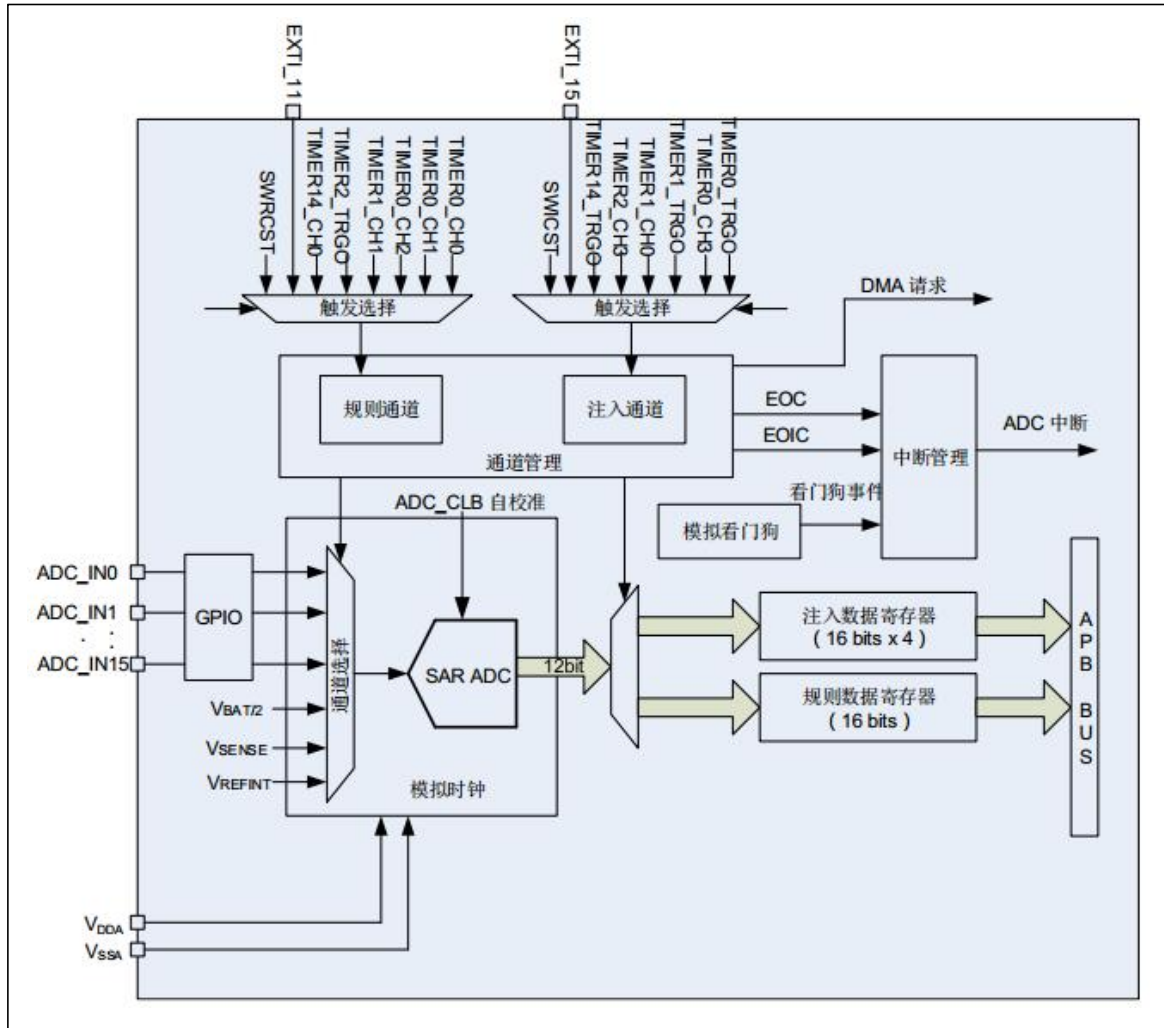
- ADC 简介
- ADC 规则组的连续转换功能
- ADC 规则组的 DMA 功能

1、ADC 简介

ADC 的主要特性如下：

- ❖ 高性能：
 - 12 位分辨率
 - ADC 转换时间：1.0us(1MS/s)
 - 自校准时间：83 个 ADC 时钟周期
 - 可编程的采样时间
 - 数据寄存器可配置数据对齐方式
 - 支持规则通道数据转换的 DMA 请求
- ❖ 双时钟域架构(APB 时钟和 ADC 时钟)
- ❖ 模拟输入通道：
 - 16 个外部模拟输入通道（GD32F130G8U6 有 10 个）
 - 1 个内部温度传感通道（VSENSE）
 - 1 个内部参考电压输入通道（VREFINT）
 - 1 个监测外部 VBAT 供电引脚的内部输入通道
- ❖ 可配置的转换触发：
 - 软件
 - 配置极性进行硬件触发（来自 TIMER0、TIMER1、TIMER2 和 TIMER14 内部定时器事件）
 - 规则通道和注入通道外部触发
- ❖ 转换模式：
 - 转换单个通道，或者扫描一序列的通道
 - 单次模式，每次触发转换一个选择的输入通道
 - 连续模式，连续转换所选择的输入通道
 - 间断模式
- ❖ 规则转换和注入组转换完成，模拟看门狗都可以产生中断
- ❖ 模拟看门狗
- ❖ ADC 供电要求：2.6V 到 3.6V，一般电源电压为 3.3V
- ❖ ADC 输入范围： $VSSA \leq V_{IN} \leq VDDA$

ADC 的模块框图如下：



2、ADC 规则组的连续转换功能

2.1 ADC 规则组的连续转换功能配置

本小节介绍如何使用 ADC 规则组的连续转换功能，例程：Regular_Continuous，该例程位于：GD32F130G8_Examples\GD32F130G8_08_ADC\目录下。例程主要是配置 ADC 规则组的连续转换功能来读取电压值（使用的是 ADC 通道 8：ADC_IN8），GPIO 口是 PB0），然后通过串口把数据打印出来，配置步骤如下：

- 1) 使能 ADC_IN8 所在 GPIO 时钟 (GPIOB)
- 2) 使能 ADC 外设时钟
- 3) 配置 ADC 时钟
- 4) 配置 ADC_IN8 所在 I/O 口为模拟输入模式
- 5) ADC 连续模式使能
- 6) ADC 触发器配置为软件触发
- 7) ADC 数据右对齐

- 8) 通道长度 1
- 9) 规则通道配置：规则组 0、ADC 通道 8、采样时间 55.5 个 ADC 时钟、使能规则通道
- 10) 使能 ADC
- 11) 使能 ADC 校准

在主函数中使能 ADC 转换并读取 ADC 值在串口上打印出来，步骤如下：

- 1) 使能软件触发
- 2) 清除 EOC 位（转换结束标志位）
- 3) 等待 ADC 转换结束
- 4) 读取 ADC 值并转换
- 5) 串口打印 ADC 值

2.2 例程介绍

ADC 的配置在 adc.c 文件中实现：

```
void ADC_Config(void)
{
    /* 使能 ADC 通道所在 GPIO 外设时钟 */
    rcu_periph_clock_enable(RCU_GPIOB);
    /* 使能 ADC 时钟 */
    rcu_periph_clock_enable(RCU_ADC);
    /* 配置 ADC 时钟 */
    rcu_adc_clock_config(RCU_ADCCCK_APB2_DIV6);
    /* 配置 ADC 通道所在 GPIO 为模拟模式 */
    gpio_mode_set(ADC_GPIO_PORT, GPIO_MODE_ANALOG, GPIO_PUPD_NONE,
                  ADC_GPIO_PIN);
    /* ADC 连续模式使能 */
    adc_special_function_config(ADC_CONTINUOUS_MODE, ENABLE);
    /* ADC 触发器配置：软件触发 */
    adc_external_trigger_source_config(ADC_REGULAR_CHANNEL,
                                      ADC_EXTTRIG_REGULAR_SWRCST);
    /* ADC 数据对齐配置：右对齐 */
    adc_data_alignment_config(ADC_DATAALIGN_RIGHT);
    /* ADC 通道长度配置：1 */
    adc_channel_length_config(ADC_REGULAR_CHANNEL, 1);

    /* ADC 规则通道配置：规则组 0、ADC 通道 8、采样时间 55.5ADC 时钟周期、使能规则通道 */
    adc_regular_channel_config(0, ADC_CHANNEL_8, ADC_SAMPLETIME_55POINT5);
    adc_external_trigger_config(ADC_REGULAR_CHANNEL, ENABLE);
}
```

```
adc_enable();    //使能 ADC
adc_calibration_enable();    //使能 ADC 校准
}
```

ADC 通道的端口定义在 adc.h 文件下：

```
#define ADC_CHANNEL            ADC_CHANNEL_8
#define ADC_GPIO_PORT         GPIOB
#define ADC_GPIO_PIN          GPIO_PIN_0
```

主函数如下：

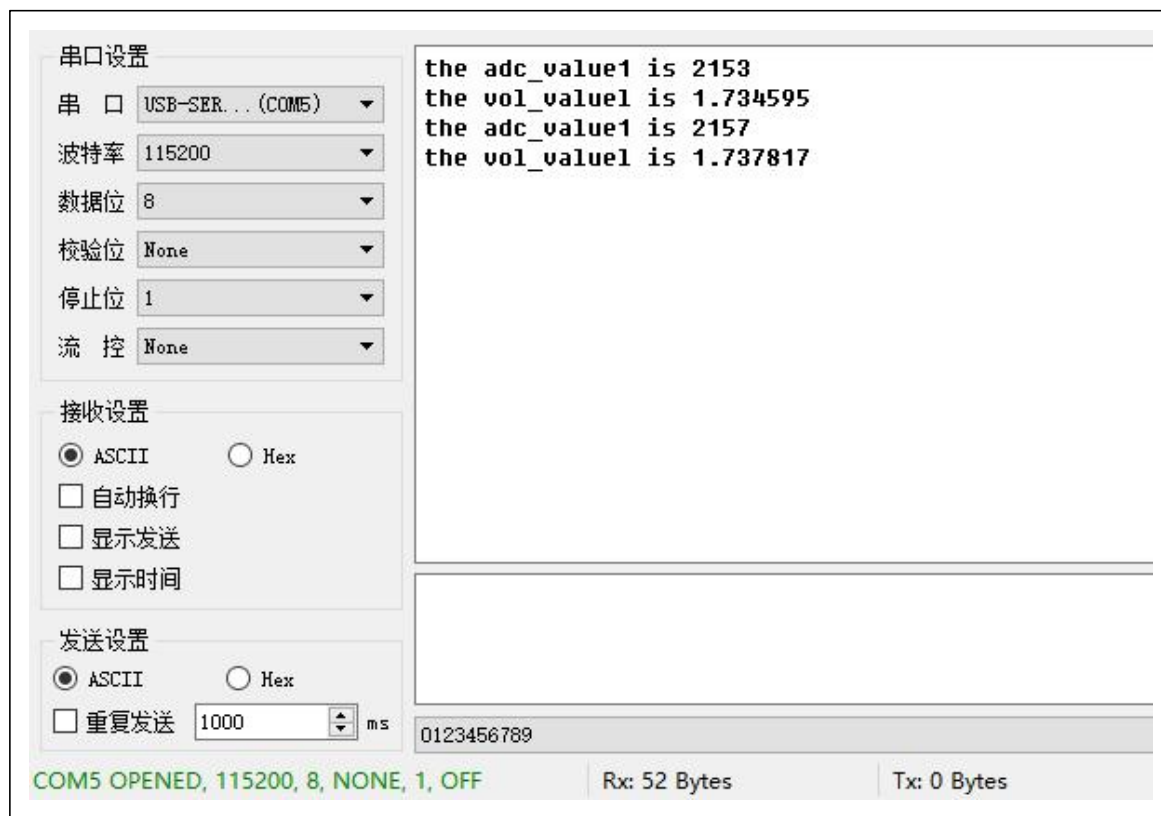
```
int main(void)
{
    systick_config(); //初始化滴答定时器
    LED_Init();    //LED 初始化
    USART0_Init(); //USART0 初始化
    ADC_Config();  //ADC 初始化

    /* 软件触发使能 */
    adc_software_trigger_enable(ADC_REGULAR_CHANNEL);
    while(1)
    {
        adc_flag_clear(ADC_FLAG_EOC); //清除 EOC 标志位
        while(SET != adc_flag_get(ADC_FLAG_EOC)); //等待 ADC 转换完成

        adc_value = ADC_RDATA; //读取 ADC 值
        vol_value = (float)adc_value/4096 * 3.3; //转换为电压值

        printf("the adc_value1 is %d\r\n", adc_value); //打印 ADC 值
        printf("the vol_valuel is %f\r\n", vol_value); //打印电压值
        LED1_Toggle();
        delay_1ms(500);
    }
}
```

把程序下载到核心板后，LED1 闪烁。将一个点位器模块的输出口接到 PB0，然后使用一个 USB 转 TTL 模块与核心板进行连接，连接线序为：模块的 TX-->PA10、模块的 RX-->PA9、模块的 GND-->GND。连接好后把 USB 转 TTL 模块接入电脑，打开电脑上位机（串口调试助手），上位机配置为：波特率 115200、数据位 8 位、校验位无、停止位 1 位、流控无。然后选择正确串口（不了解的可以百度一下：如何查看串口号），把串口打开，按一下板子 RST 键，可以看到串口打印出 ADC 值以及电压值（也可以把 PB0 短接到 GND 或 VCC，此时 ADC 值约为 0 或 4096，电压值约为 0 或 3300mV）：



3、规则组的 DMA 功能

3.1 规则组的 DMA 功能配置

本小节介绍如何使用 ADC 规则组的 DMA 功能，例程：Regular_DMA，该例程位于：GD32F130G8_Examples\GD32F130G8_08_ADC\目录下。例程主要是配置 ADC 规则组的连续转换功能和 DMA 功能来读取电压值（使用的是 ADC 通道 8：ADC_IN8），GPIO 口是 PB0），然后通过串口把数据打印出来，例程的效果是跟第二小节一样的，只是数据的读取使用 DMA（DMA_CH0）功能。ADC 配置步骤如下：

- 1) 使能 ADC_IN8 所在 GPIO 时钟（GPIOB）
- 2) 使能 ADC 外设时钟
- 3) 配置 ADC 时钟
- 4) 配置 ADC_IN8 所在 I/O 口为模拟输入模式
- 5) ADC 连续模式使能
- 6) ADC 触发器配置为软件触发
- 7) ADC 数据右对齐
- 8) 通道长度 1
- 9) 规则通道配置：规则组 0、ADC 通道 8、采样时间 55.5 个 ADC 时钟、使能规则通道
- 10) 使能 ADC
- 11) 使能 ADC 校准

ADC 的 DMA 配置步骤如下：

- 1) 使能 DMA 时钟
- 2) DMA_CH0 通道配置：
 - a. 外设地址
 - b. 内存地址
 - c. 外设到内存模式
 - d. 内存数据宽度 16 位
 - e. 外设数据宽度 16 位
 - f. 优先级高
 - g. 数据量 1
 - h. 外设地址递增禁能
 - i. 内存地址递增禁能
 - j. 使能循环模式
 - k. 使能 DMA_CH0
 - l. 使能 ADC 的 DMA 模式

主函数中读取 ADC 数据步骤如下：

- 1) 软件触发使能
- 2) 等待 DMA 传输完成
- 3) 转换电压值
- 4) 串口打印

3.2 例程介绍

adc 的配置和 DMA 配置在 adc.c 文件中实现：

```
void ADC_Config(void)
{
    /* 使能 ADC 通道所在 GPIO 外设时钟 */
    rcu_periph_clock_enable(RCU_GPIOB);
    /* 使能 ADC 时钟 */
    rcu_periph_clock_enable(RCU_ADC);
    /* 配置 ADC 时钟 */
    rcu_adc_clock_config(RCU_ADCK_APB2_DIV6);
    /* 配置 ADC 通道所在 GPIO 为模拟模式 */
    gpio_mode_set(ADC_GPIO_PORT, GPIO_MODE_ANALOG, GPIO_PUPD_NONE,
                  ADC_GPIO_PIN);
    /* ADC 连续模式使能 */
    adc_special_function_config(ADC_CONTINUOUS_MODE, ENABLE);
    /* ADC 触发器配置：软件触发 */
}
```



```

adc_external_trigger_source_config(ADC_REGULAR_CHANNEL,
                                   ADC_EXTTRIG_REGULAR_SWRCST);

/* ADC 数据对齐配置：右对齐 */
adc_data_alignment_config(ADC_DATAALIGN_RIGHT);
/* ADC 通道长度配置：1 */
adc_channel_length_config(ADC_REGULAR_CHANNEL, 1);
/* ADC 规则通道配置：规则组 0、ADC 通道 8、采样时间 55.5ADC 时钟周期、使能规则通道 */
adc_regular_channel_config(0, ADC_CHANNEL_8, ADC_SAMPLETIME_55POINT5);
adc_external_trigger_config(ADC_REGULAR_CHANNEL, ENABLE);

adc_enable();    //使能 ADC
adc_calibration_enable();    //使能 ADC 校准
}

void ADC_DMA_Config(void)
{
    /* 使能 DMA 外设时钟 */
    rcu_periph_clock_enable(RCU_DMA);

    /* ADC_DMA_channel 配置 */
    dma_deinit(DMA_CH0);    //重置 DMA_CH0 相关寄存器
    dma_periph_address_config(DMA_CH0, (uint32_t)&ADC_RDATA);    //外设地址
    dma_memory_address_config(DMA_CH0, (uint32_t)&adc_value);    //内存地址
    dma_transfer_direction_config(DMA_CH0, DMA_PERIPHERAL_TO_MEMORY);    //外设到内存
    dma_memory_width_config(DMA_CH0, DMA_MEMORY_WIDTH_16BIT);    //内存数据宽度 16Bit
    dma_periph_width_config(DMA_CH0, DMA_PERIPHERAL_WIDTH_16BIT);    //外设数据宽度 16Bit
    dma_priority_config(DMA_CH0, DMA_PRIORITY_HIGH);    //优先级高
    dma_transfer_number_config(DMA_CH0, 1);    //数据量 1
    dma_periph_increase_disable(DMA_CH0);    //外设地址递增禁能
    dma_memory_increase_disable(DMA_CH0);    //内存地址递增禁能
    dma_circulation_enable(DMA_CH0);    //使能循环模式
    dma_channel_enable(DMA_CH0);    //使能 DMA_CH0

    adc_dma_mode_enable();    //使能 ADC 的 DMA 模式
}

```

ADC 通道的端口定义在 adc.h 文件下：

```

#define ADC_CHANNEL            ADC_CHANNEL_8
#define ADC_GPIO_PORT         GPIOB
#define ADC_GPIO_PIN          GPIO_PIN_0

```

主函数如下：

```
int main(void)
{
    systick_config(); //初始化滴答定时器
    LED_Init(); //LED 初始化
    USART0_Init(); //USART0 初始化
    ADC_Config(); //ADC 初始化
    ADC_DMA_Config(); //ADC 的 DMA 配置初始化

    /* 软件触发使能 */
    adc_software_trigger_enable(ADC_REGULAR_CHANNEL);
    while(1)
    {
        /* 等待 DMA 发送完成 */
        while (RESET == dma_flag_get(DMA_CH0, DMA_FLAG_FTF));

        vol_value = (float)adc_value/4096 * 3.3; //转换为电压值
        printf("the adc_value1 is %d\r\n", adc_value); //打印 ADC 值
        printf("the vol_valuel is %f\r\n", vol_value); //打印电压值

        LED1_Toggle();
        delay_1ms(500);
    }
}
```

注：内存地址在 main.c 文件中定义，adc.c 文件引用

把程序下载到核心板后，LED1 闪烁。将一个点位器模块的输出口接到 PB0，然后使用一个 USB 转 TTL 模块与核心板进行连接，连接线序为：模块的 TX-->PA10、模块的 RX-->PA9、模块的 GND-->GND。连接好后把 USB 转 TTL 模块接入电脑，打开电脑上位机（串口调试助手），上位机配置为：波特率 115200、数据位 8 位、校验位无、停止位 1 位、流控无。然后选择正确串口（不了解的可以百度一下：如何查看串口号），把串口打开，按一下板子 RST 键，可以看到串口打印出 ADC 值以及电压值（也可以把 PB0 短接到 GND 或 VCC，此时 ADC 值约为 0 或 4096，电压值约为 0 或 3300mV）：

串口设置

串 口

USB-SER... (COM5)

波特率

115200

数据位

8

校验位

None

停止位

1

流 控

None

接收设置

☒ ASCII

☐ Hex

☐ 自动换行

☐ 显示发送

☐ 显示时间

发送设置

☒ ASCII

☐ Hex

☐ 重复发送

1000

ms

```

the adc_value1 is 2153
the vol_value1 is 1.734595
the adc_value1 is 2157
the vol_value1 is 1.737817

```

0123456789

COM5 OPENED, 115200, 8, NONE, 1, OFF

Rx: 52 Bytes

Tx: 0 Bytes

第九章 FWDGT 应用

看门狗定时器（WDGT）是一个硬件计时电路，用来监测由软件故障导致的系统故障。片上有两个看门狗定时器外设，自由看门狗定时器（FWDGT）和窗口看门狗定时器（WWDGT）。它们使用灵活，并提供了很高的安全水平和精准的时间控制。两个看门狗定时器都是用来解决软件故障问题的。看门狗定时器在内部计数值达到了预设的门限的时候，会触发一个复位（对于窗口看门狗定时器来说，会产生一个中断）。当处理器工作在调试模式的时候看门狗定时器定时计数器可以停止计数。本章从以下部分介绍 FWDGT 相关内容：

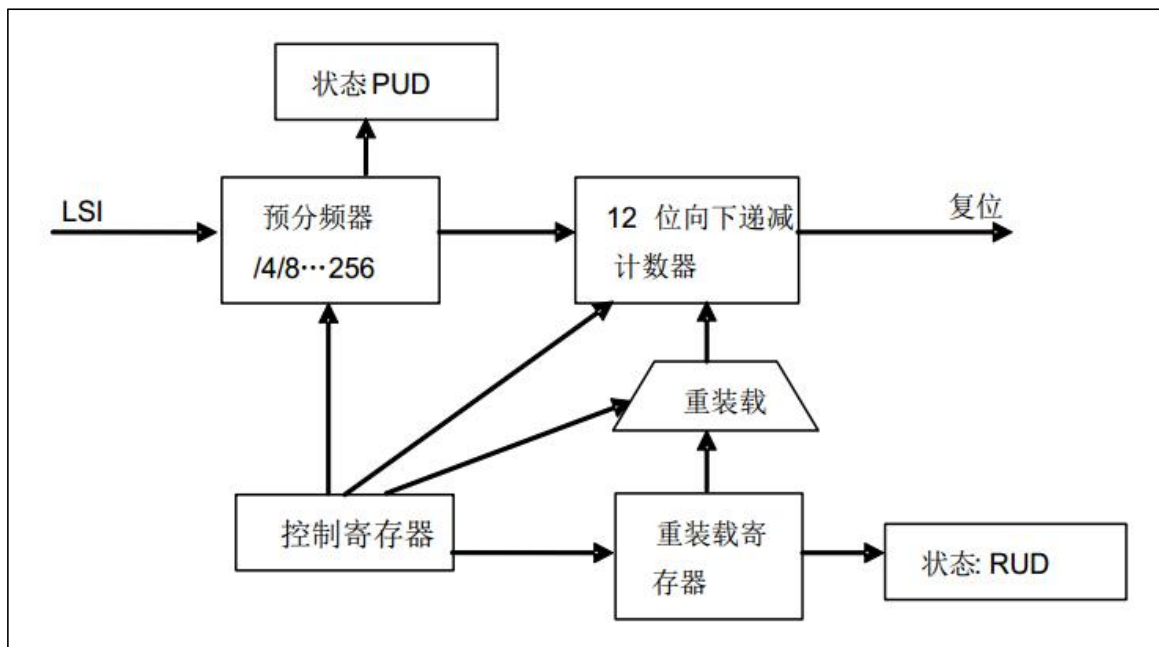
- FWDGT 简介
- FWDGT 应用

4、FWDGT 简介

FWDGT 的主要特性如下：

- ❖ 自由运行的 12 位向下计数器
- ❖ 如果看门狗定时器被使能，那么当向下计数器的值达到 0 时产生系统复位
- ❖ 独立时钟源，自由看门狗定时器在主时钟故障（例如待机和深度睡眠模式下）时仍能工作
- ❖ 自由看门狗定时器硬件控制位，可以用来控制是否在上电时自动启动自由看门狗定时器
- ❖ 可以配置自由看门狗定时器在调试模式下选择停止还是继续工作

独立看门狗定时器带有一个 8 级预分频器和一个 12 位的向下递减计数器。独立看门狗使用的是系统内部的 40kHz（IRC40K）时钟。独立看门狗模块框图如下：



独立看门狗定时器在 40kHz（IRC40K）时的最小/最大计数时间：

预分频系数	PSC[2:0]位	最小计数时间 (ms) RL[11:0]=0x000	最大计数时间 (ms) RL[11:0]=0xffff
1/4	000	0.1	409.6
1/8	001	0.2	819.2
1/16	010	0.4	1638.4
1/32	011	0.8	3276.8
1/64	100	1.6	6553.6
1/128	101	3.2	13107.2
1/256	110 或 111	6.4	26214.4

5、FWDGT 应用

2.1 FWDGT 的配置

本小节介绍 FWDGT 的应用，设置 FWDGT 计数器产生 1S 的计时时间，在主函数中每隔 500ms 喂狗一次（刷新一次计数器）LED1 状态切换一次，当 KEY 按下时，触发外部中断死循环，导致无法喂狗而产生 FWDGT 复位，程序重新运行，如果检测到有 FWDGT 复位产生则 LED2 亮。例程：FWDGT，该例程位于 GD32F130G8_Examples\GD32F130G8_09_FWDGT\目录下。FWDGT 的配置与检测都在主函数中实现，FWDGT 配置步骤如下：

- 1) 使能 IRC40K 时钟
- 2) 等待时钟稳定
- 3) 使能 FWDGT，并设置计数时间约为 1S

2.2 例程介绍

FWDGT 的配置与复位标志位检测都在主函数中实现，主函数如下：

```
int main(void)
{
    systick_config(); //初始化滴答定时器
    /* 使能 IRC40K 时钟 */
    rcu_osci_on(RCU_IRC40K);
    /* 等待 IRC40K 时钟稳定 */
    rcu_osci_stab_wait(RCU_IRC40K);
    LED_Init(); //LED 初始化
    EXTI_Init(); //EXTI 初始化

    /* 使能 FWDGT，时钟频率约为 40K/64 = 625Hz，装载值 625，计数时间约为 1S */
```

```
fwdgt_config(625,FWDGT_PSC_DIV64);
fwdgt_enable();

/* 检测是否有 FWDGT 复位产生 */
if(RESET != rcu_flag_get(RCU_FLAG_FWDGTRST))
{
    rcu_all_reset_flag_clear(); //清除标志位
    LED2_ON(); //点亮 LED2
}
else
    LED2_OFF(); //关闭 LED2

while(1)
{
    fwdgt_counter_reload(); //看门狗喂狗
    LED1_Toggle();
    delay_1ms(500);
}
}
```

外部中断函数在 gd32f1x0_it.c 文件下：

```
void EXTI4_15_IRQHandler(void)
{
    if (RESET != exti_interrupt_flag_get(KEY_EXTI_LINE)) //读取中断标志，判断是否 KEY
                                                         引起的外部中断
    {
        exti_interrupt_flag_clear(KEY_EXTI_LINE); //清除中断标志位
        while(1);
    }
}
```

把程序下载到核心板后，LED1 闪烁；按下 KEY，LED2 亮，LED1 闪烁。

第十章 WWDGT 应用

窗口看门狗定时器(WWDGT)用来监测由软件故障导致的系统故障。窗口看门狗定时器开启后，向下递减计数器值逐渐减小。计数值达到 0x3F 时会产生复位(CNT[6]位被清 0)。在计数器计数值达到窗口寄存器值之前，计数器的更新也会产生复位。因此软件需要在给定的区间内更新计数器。窗口看门狗定时器在计数器计数值达到 0x40 或者在计数值达到窗口寄存器之前更新计数器，都会产生一个提前唤醒标志，如果使能中断也将会产生中断。本章从以下部分介绍 WWDGT 相关内容：

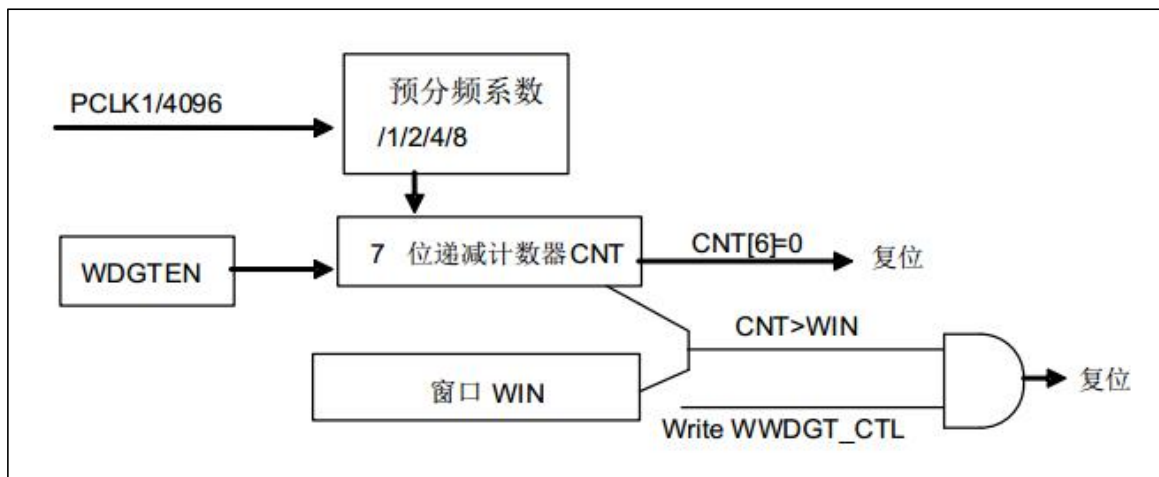
- WWDGT 简介
- WWDGT 应用

1、WWDGT 简介

WWDGT 的主要特性如下：

- ❖ 可编程的 7 位自由运行向下递减计数器
- ❖ 当窗口看门狗使能后，有以下两种情况会产生复位：
 - 当计数器达到 0x3F 时产生复位
 - 当计数器的值大于窗口寄存器的值时，更新计数器会产生复位
- ❖ 提前唤醒中断(EWI)：如果看门狗定时器打开，中断使能，计数值达到 0x40 或者在计数值达到窗口寄存器之前更新计数器的时候会产生中断
- ❖ 可以配置窗口看门狗定时器在调试模式下选择停止还是继续工作

WWDGT 模块框图如下：



窗口看门狗定时器超时的计算公式如下：

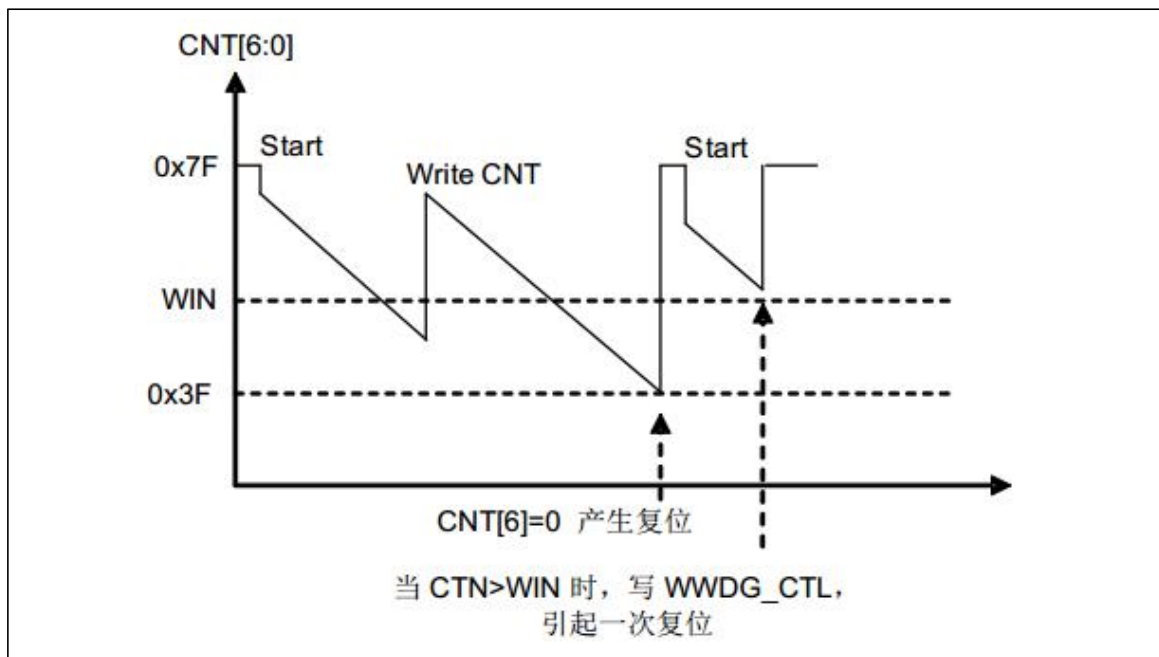
$$t_{\text{wwdgt}} = t_{\text{pclk1}} \times 4096 \times 2^{\text{psc}} \times (\text{CNT}[5:0] + 1) \text{ (ms)}$$

其中：

T_{wwdgt} ：窗口看门狗定时器的超时时间

t_{pclk1} ：APB1 以 ms 为单位的时钟周期

窗口看门狗定时器时序图如下：



36MHz (fPCLK1) 下的最小/最大超时时间如下表：

预分频系数	PSC[1:0]	最小超时 CNT[6:0]=0x40	最大超时 CNT[6:0]=0x7f
1/1	00	113μs	7.28ms
1/2	01	227μs	14.56ms
1/4	10	455μs	29.12ms
1/8	11	901μs	58.25ms

2、WWDGT 应用

2.1 WWDGT 的配置

本小节介绍 WWDGT 的应用，设置 WWDGT 的窗口时间为 $21.4\text{ms} < x < 29.2\text{ms}$ ，在主函数中每隔 25ms 喂狗一次（刷新窗口值），当 KEY 按下时，触发外部中断死循环，导致无法喂狗而产生 WWDGT 复位，程序重新运行，如果检测到有 WWDGT 复位产生则 LED2 亮。例程：WWDGT，该例程位于 GD32F130G8_Examples\GD32F130G8_10_WWDGT\目录下。WWDGT 的配置与检测都在主函数中实现，WWDGT 配置步骤如下：

- 1) 使能 WWDGT 外设时钟
- 2) 配置 WWDGT 窗口值
- 3) 使能 WWDGT

2.2 例程介绍

WWDGT 的配置与复位标志位检测都在主函数中实现，主函数如下：

```
int main(void)
{
    systick_config(); //滴答定时器配置
    LED_Init(); //LED 初始化
    EXTI_Init(); //EXTI 初始化

    /* 检测是否有 WWDGT 复位产生 */
    if(RESET != rcu_flag_get(RCU_FLAG_WWDGTRST))
    {
        rcu_all_reset_flag_clear(); //清除标志位
        LED2_ON(); //点亮 LED2
    }
    else
        LED2_OFF(); //关闭 LED2

    /* 使能 WWDGT 外设时钟 */
    rcu_periph_clock_enable(RCU_WWDGT);

    /*
        看门狗时钟：72MHz/4096/8 = 2200Hz
        溢出时间：(127-63) / 2200 = 29.2ms
        窗口刷新时间：29.2ms > x > 21.4ms      ((127-80)/2200) = 21.4ms
    */
    wwdgt_config(127,80,WWDGT_CFG_PSC_DIV8);
    wwdgt_enable(); //使能 WWDGT
    LED1_ON(); //LED1 亮
    while(1)
    {
        delay_1ms(25);
        /* WWDGT 喂狗 */
        wwdgt_counter_update(127);
    }
}
```

外部中断函数在 gd32f1x0_it.c 文件下：

```
void EXTI4_15_IRQHandler(void)
{

```

```
if (RESET != exti_interrupt_flag_get(KEY_EXTI_LINE)) //读取中断标志，判断是否 KEY
                                                    引起的中断
{
    exti_interrupt_flag_clear(KEY_EXTI_LINE); //清除中断标志位
    while(1);
}
}
```

把程序下载到核心板后，LED1 亮；按下 KEY，LED2 亮。