

Porting Guide

megawin

MG32F10x
移植 STM32F10x 手册

版本 1.01

日期 2022/3/03

目录

1. 前言	5
1.1. 文档须知	5
2. 硬件差异对比	6
2.1. 引脚差异对比	6
2.2. 内部资源对比	6
3. 开发环境搭建	7
3.1. 开发 MG32F10x 使用的 IDE	7
3.2. 开发包的安装	7
3.3. 建立一个工程	8
3.4. 固件库配置	14
3.5. 仿真器配置	16
3.5.1. 使用 ST-Link 仿真	16
3.5.2. 使用 J-Link 仿真	19
4. 硬件布线建议	22
4.1. 印制电路板	22
4.2. 器件位置	22
4.3. 接地和供电 (VSS/VDD)	22
4.4. 去耦合	22
4.5. 供电方案	23
4.6. 其他信号	23
4.7. 未用到的 IO 及其性	23
4.8. 时钟	24
4.9. 模拟信号	24
4.10. EMI	24
5. 外设移植	26
5.1. 移植前的准备	26
5.2. ADC	27
5.3. ANCTL (模拟控制器)	29
5.3.1. CMP	29
5.3.2. DCSS	29
5.4. BKP	30
5.5. CRC	30
5.6. DMAC	30
5.7. EXTI	32
5.8. FMC (闪存器控制模块)	33
5.9. GPIO	33
5.10. I2C	34
5.11. I2S	36
5.12. IWDG (独立看门狗)	36
5.13. LED	38
5.14. NVIC	38
5.15. PWR (电源模式控制)	38
5.16. RCC	40
5.17. RNG (随机数发生器)	43
5.18. RTC	43
5.19. SFM (特殊功能宏)	45
5.20. SPI	45

5.21. SYSTICK.....	46
5.22. TIM.....	48
5.23. UART	51
5.24. USB.....	52
5.25. WWDG（窗口看门狗）	54
6. 版本历史.....	55

1. 前言

1.1. 文档须知

MG32F10x 系列是笙泉于 2021 年推出的 Cortex-M3 系列的产品。该系列有着性能高、软硬件兼容性强等优点。本文档旨在帮助使用 STM32F10x 系列芯片的用户移植到 MG32F10x 系列。

总体来看，在硬件移植上是比较简单的，可以原脚位直接从 STM32F10x 系列芯片替换成 MG32F10x 系列，不需要另外再修改电路。但是软件上仍有些差异，因此在软件里需要另做修改，本文档也会介绍关于软件移植的细节，若有错漏，敬请谅解。

2. 硬件差异对比

2.1. 引脚差异对比

下表是 MG32F10x 系列引脚与 STM32F10x 系列的引脚差异。

封装	芯片	
	MG32F10x	STM32F10x
LQFP48	引脚位置、功能全兼容	
LQFP64	引脚位置、功能全兼容	

2.2. 内部资源对比

下表是 MG32F10x 系列与 STM32F10x 系列的硬件资源差异。

	MG32F103	MG32F104	STM32F103
Core	Cortex-M3	Cortex-M3	Cortex-M3
Flash	96K~128K	256K	16K~1M
RAM	28K	36K	6K~96K
主频	72MHz	96MHz	72MHz
访问 Flash 等待周期	Cache,no wait cycle	Cache,no wait cycle	2 cycle
擦写次数	100k cycle	100k cycle	10k cycle
TIMER	4	4	4/5/8
U(S)ART	3	3	2/3/5
I2C	2	2	1/2
SPI	2	2	1/2/3
I2S	0~1	0~1	2
CAN	--	--	1
USB	Device	Device	Device
SDIO	--	--	1
ADC	1(10~16)	1(10~16)	2(10)/2(16)/3(21)
CMP	2	2	--
LED Driver	8 Segment	8 Segment	--
Active Power	100uA/MHz @3.3V	100uA/MHz @3.3V	292uA/MHz @3.3V
Sleep	5mA	5mA	5.5mA
Stop	30uA	30uA	24uA
Standby	4.5uA	4.5uA	2uA
Vbat	1.2uA	1.2uA	1.4uA

<注释>

访问 Flash 等待周期: MCU 运行时读取指令、变量等涉及 Flash 访问的操作都需要等待周期, 周期越长, 实际运行效率越低。

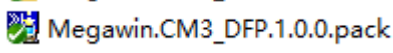
3. 开发环境搭建

3.1. 开发 MG32F10x 使用的 IDE

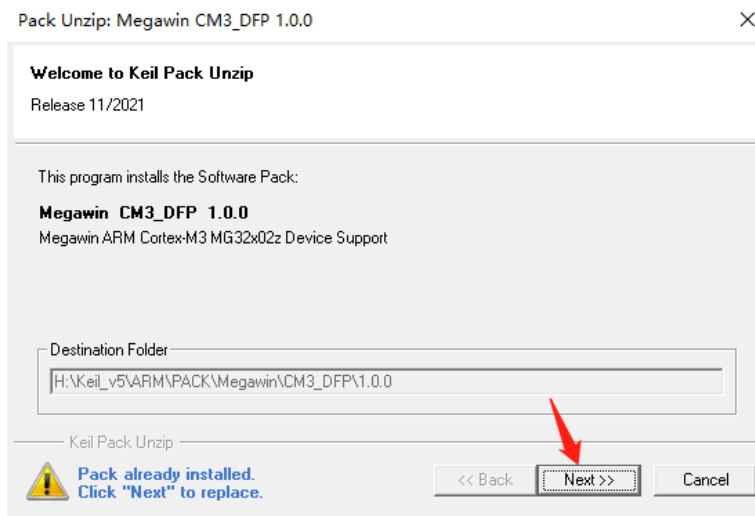
目前开发 MG32F10x 系列需要使用市面通用 Keil 5 的 MDK for ARM 版本，使用 **Keil 5 必须安装 5.26 及以上版本**，版本过低会导致无法识别开发包安装程序。

3.2. 开发包的安装

打开开发包中的 Megawin.CM3.DFP.1.0.0.pack。

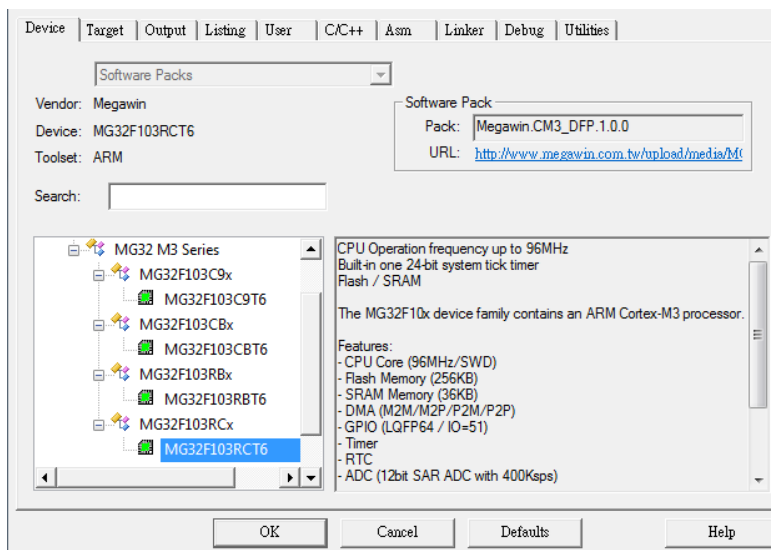


点击 Next，并等待 Finish 即可。



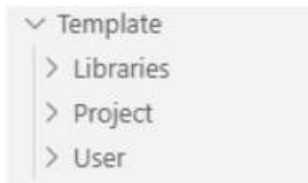
[注意]: 若无法打开上图中的安装包，就是 Keil 的版本过低，无法运行，需使用更新版本的 Keil。

安装成功后，即可选择到笙泉的 MG32F10x 系列 MCU 进行开发。

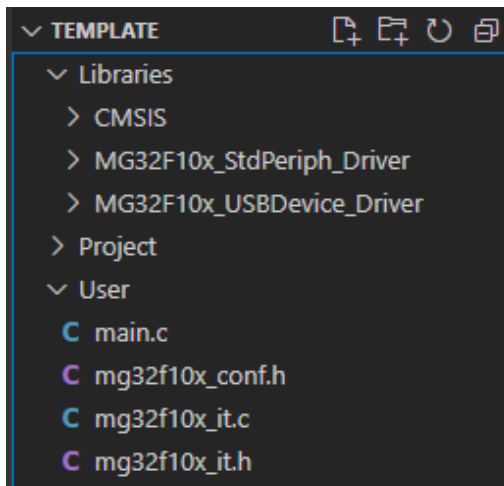


3.3. 建立一个工程

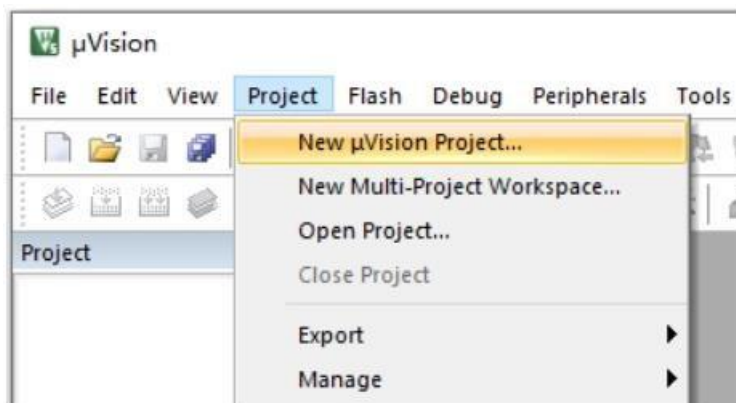
- 1) 新建一个文件夹命名为 **Template** 用以存放整个工程。
- 2) 在 **Template** 文件夹中新建 **Libraries**, **Project** 和 **User** 三个子文件夹（当然用户可定义自己工程目录结构）



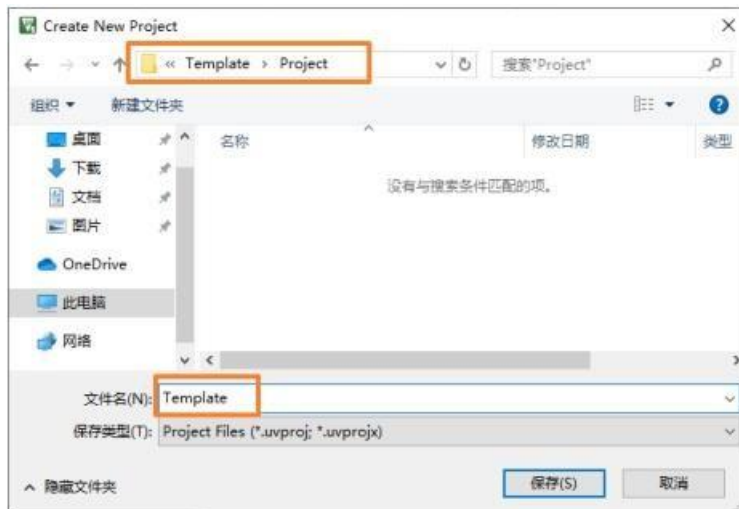
- 3) 将 **MG32F10x** 标准固件库中 **Libraries** 目录中的内容复制到 **Template\Libraries** 目录中。
将 **MG32F10x** 标准固件库中 **Project\MG32F10x_StdPeriph_Template** 目录中的内容复制到 **Template\User** 目录中。



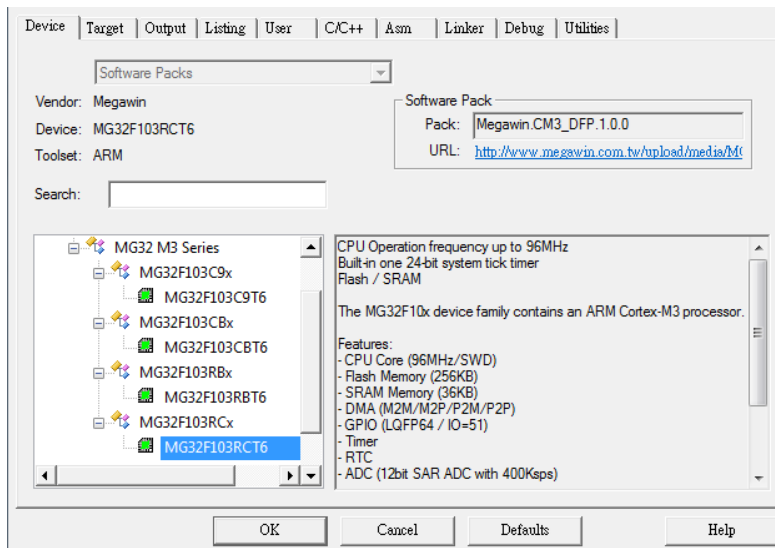
- 4) 打开 Keil MDK, 新建项目。



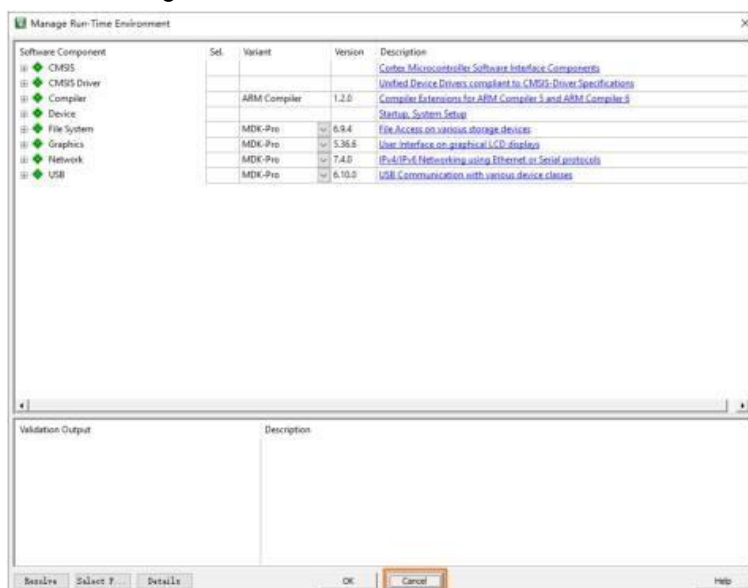
- 5) 在 **Template\Project** 目录中新建名为 **Template** 的工程。



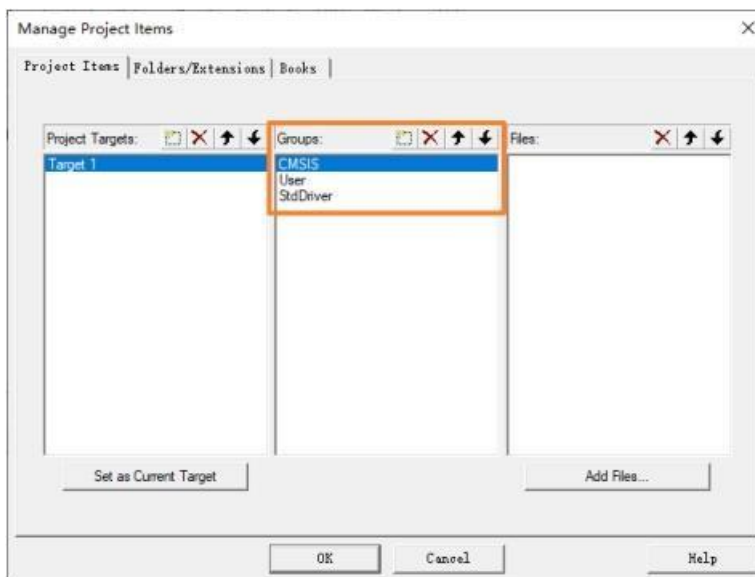
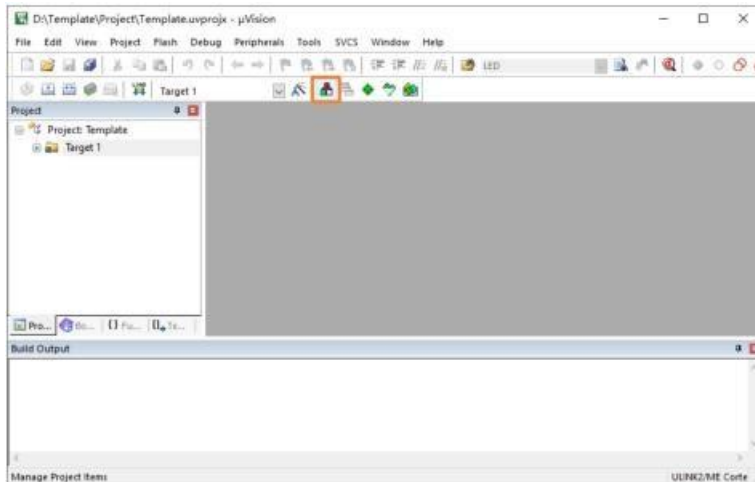
6) 选择项目使用的设备为需要的 MG32F10x 型号，并点击 OK。



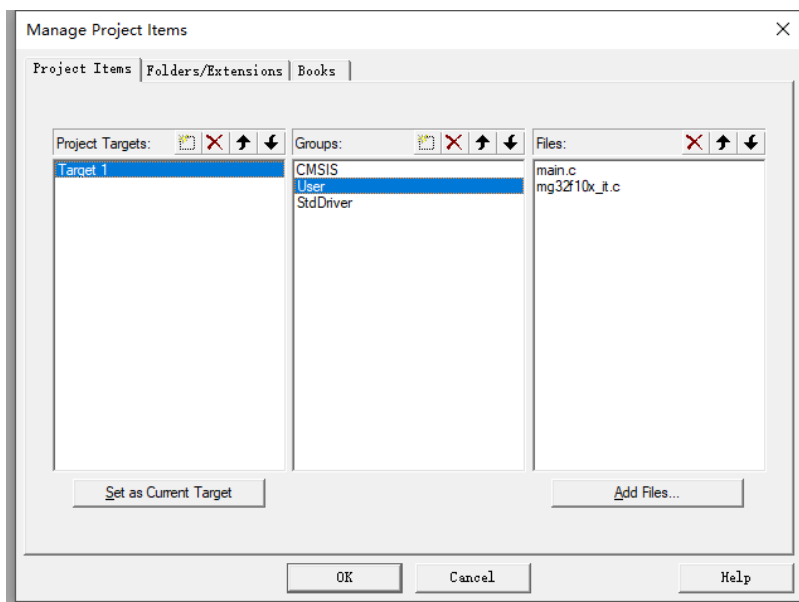
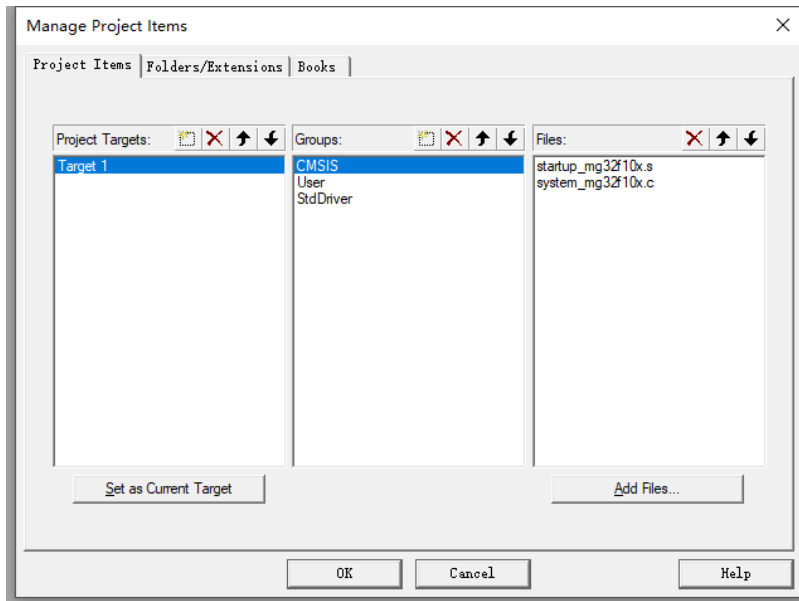
7) 此时弹出 Manage Run-Time Environment 对话框，在该对话框上点击 Cancel。

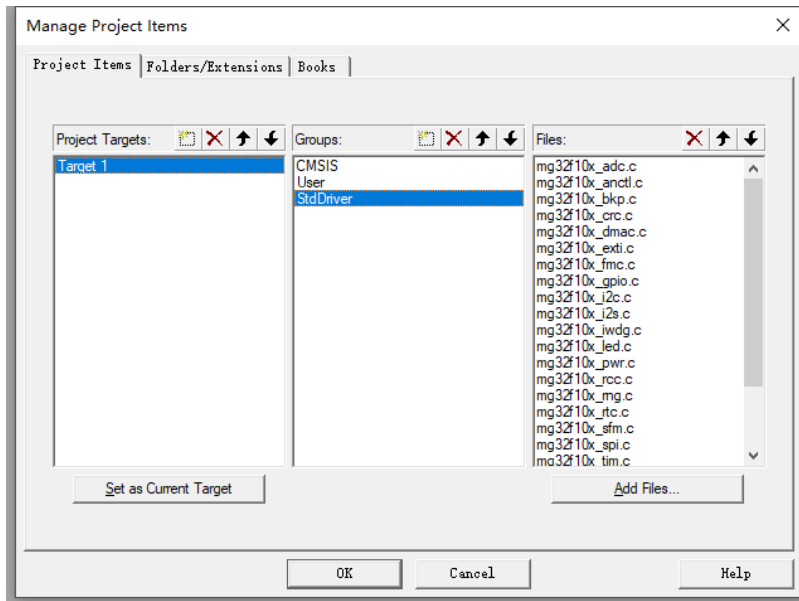


8) 在加入固件库文件之前，我们先建立三个 Groups: CMSIS, User, StdDriver。

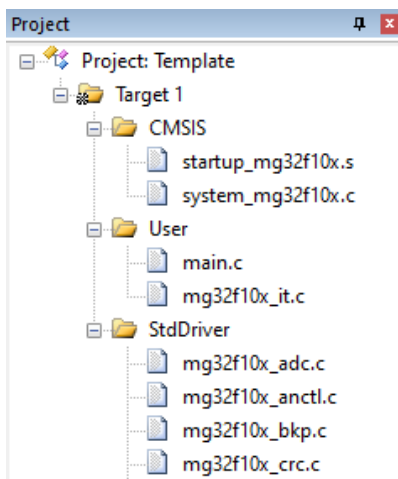


- 9) 向 Group 里面添加固件库文件。向 CMSIS Group 中添加：
[Template\Libraries\CMSIS\Device\MG\MG32F10x\startup\arm\startup_mg32f10x.s](#)
[Template\Libraries\CMSIS\Device\MG\MG32F10x\system_mg32f10x.c](#)
向 User Group 中添加：
[Template\User\main.c](#)
[Template\User\mg32f10x_it.c](#)
向 StdDriver Group 中添加 [Template\Libraries\MG32F10x_StdPeriph_Driver\src](#) 目录中所有的.c 文件。

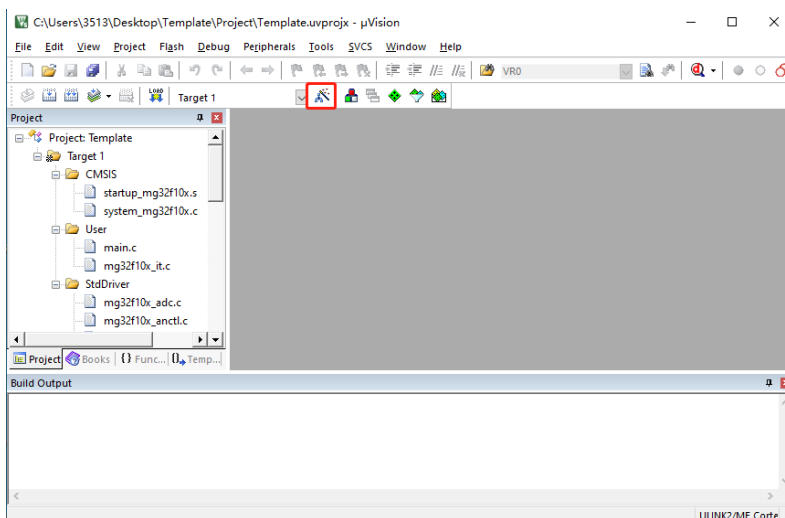




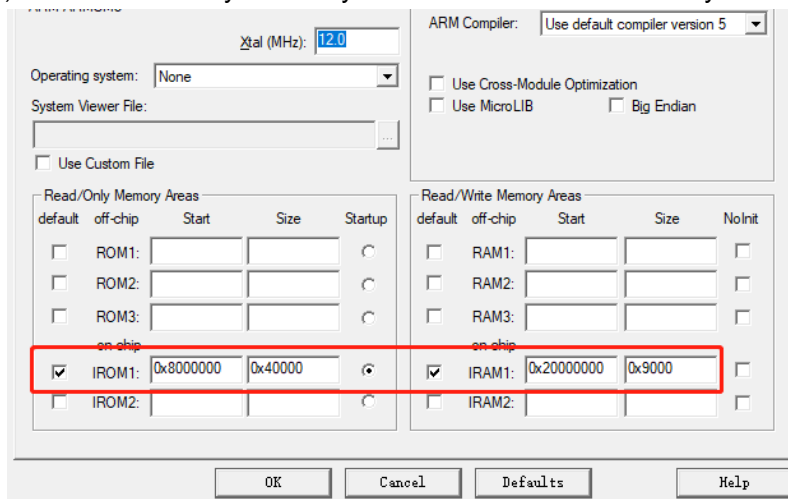
10) 最终项目的结构如下图:



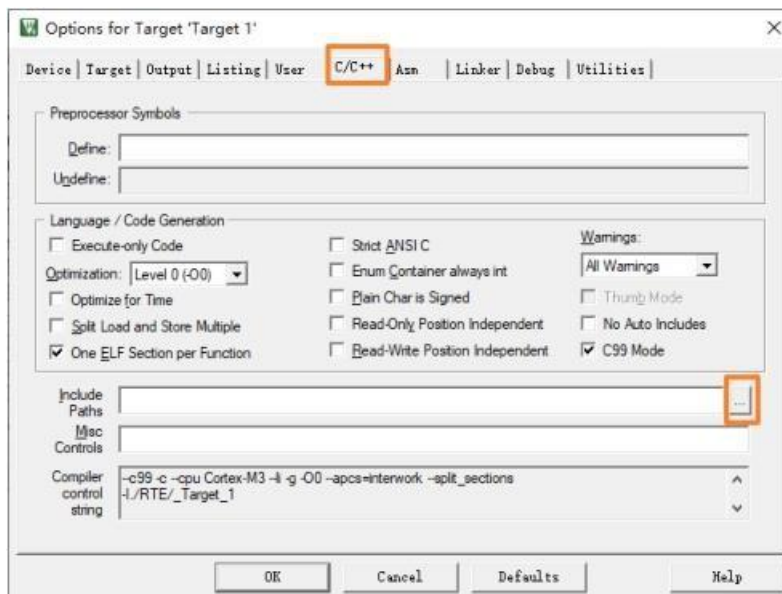
11) 打开 Options for Target 对话框。



- 12) 配置 Read/Only Memory Areas 和 Read/Write Memory Areas (配置 Flash 和 SRAM 的起始地址和大小)。

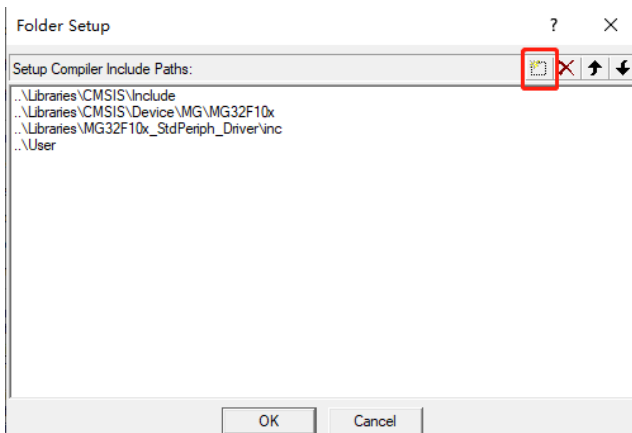


- 13) 在 C/C++选项卡中配置项目的头文件包含路径。

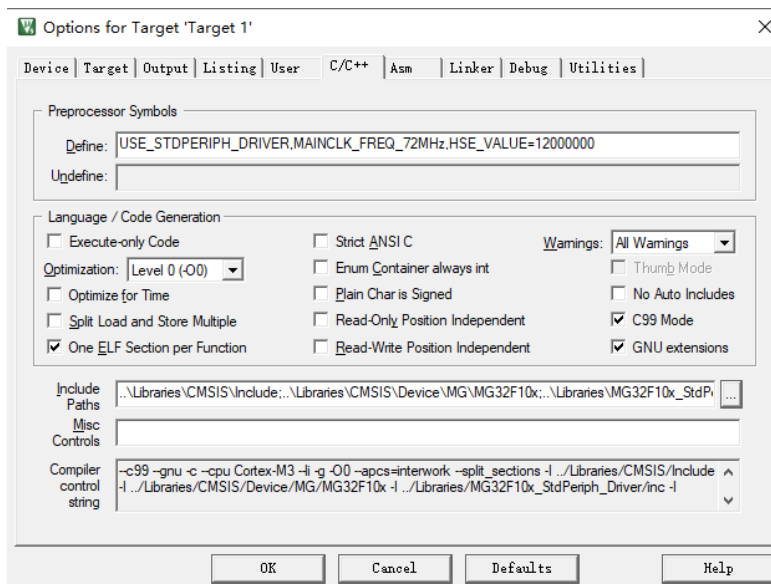


加入以下四个路径：

..\Libraries\CMSIS\Include
..\Libraries\CMSIS\Device\MG\MG32F10x
..\Libraries\MG32F10x_StdPeriph_Driver\inc
..\User



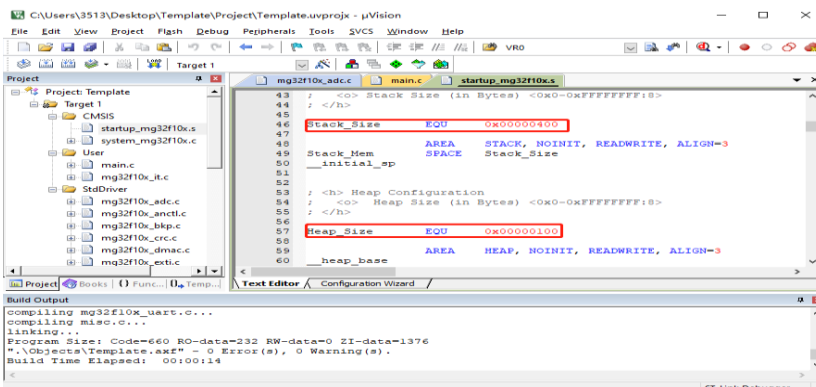
- 14) 在 Preprocessor Symbols 中加入 `USE_STDPERIPH_DRIVER,MAINCLK_FREQ_72MHz` 及 `HSE_VALUE=12000000` 定义。



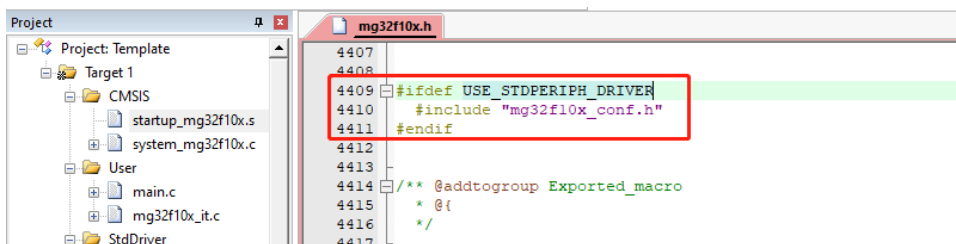
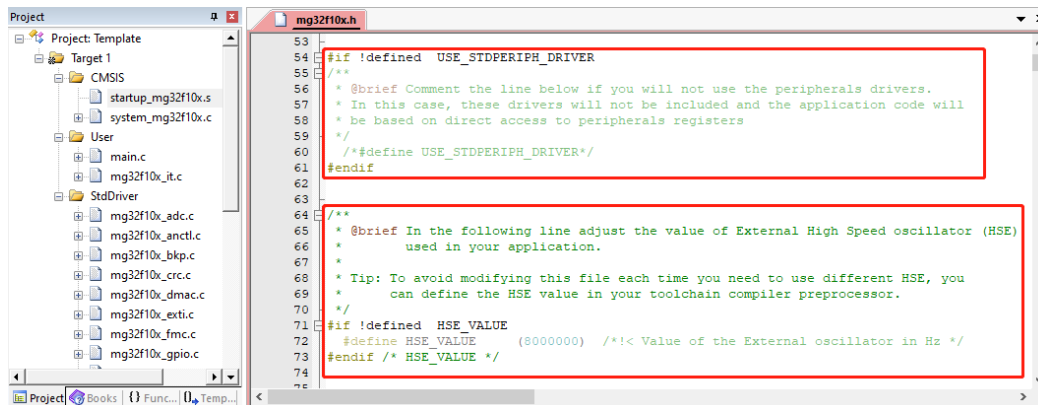
点击 OK。至此，项目建立配置完成。

3.4. 固件库配置

- 1) 在 `startup_mg32f10x.s` 可配置应用程序栈和堆的大小，如下图：



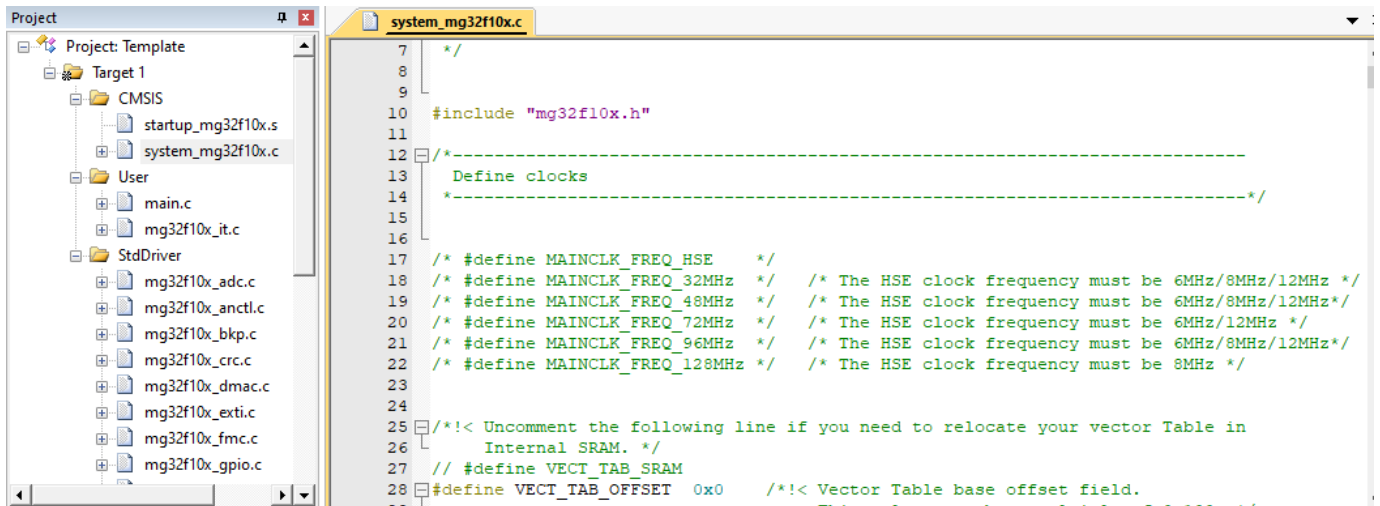
2) 在 mg32f10x.h 中存在两个宏定义，需要用户关注。



USE_STDPERIPH_DRIVER 定义这个宏表示应用程序需要使用固件库中的外设驱动，且会在项目中包含 mg32f10x_conf.h 头文件。

HSE_VALUE 该宏用于指定 MG32F10x芯片外接晶振的频率。默认情况下，固件库假定外部 HSE 晶振的频率是 8MHz。如果用户外接晶振不是**8MHz**，务必修改或在编译器全局预定义处覆盖该定义！！

3) 在 system_mg32f10x.c 中有几处定义需要用户关注。



MAINCLK_FREQ_* 这些宏定义用以配置程序在启动后芯片主时钟的频率。只能选择定义其中的一个（如果不定义任何一个，那么芯片主时钟是MHSI）。可以在编译器全局预定义处给出定义。这些宏定义对芯片外部晶振是有要求，比如要定义 **MAINCLK_FREQ_72MHz**，那么芯片外部晶振频率必须是 **12MHz**（切记：也要覆盖 **HSE_VALUE** 的定义）。

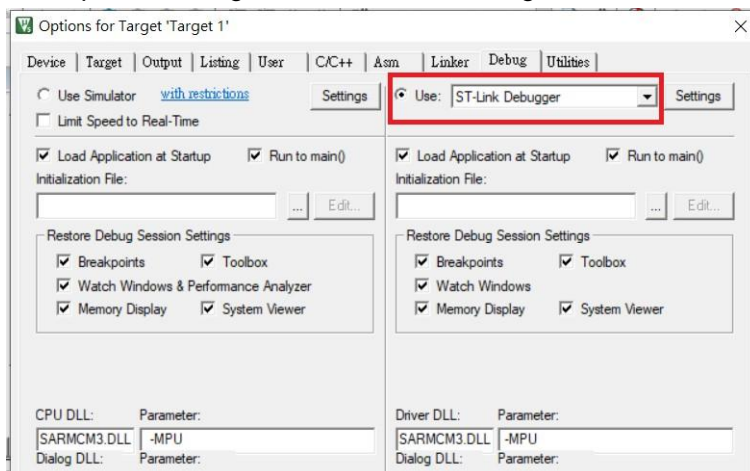
VECT_TAB_SRAM 定义这个宏表示将中断向量表映射到 **SRAM** 中（在 **SRAM** 中运行的工程才需要定义这个宏）。

VECT_TAB_OFFSET 该宏用以设置中断向量表起始地址偏移（相对于 **Flash** 或 **SRAM** 的起始地址）。

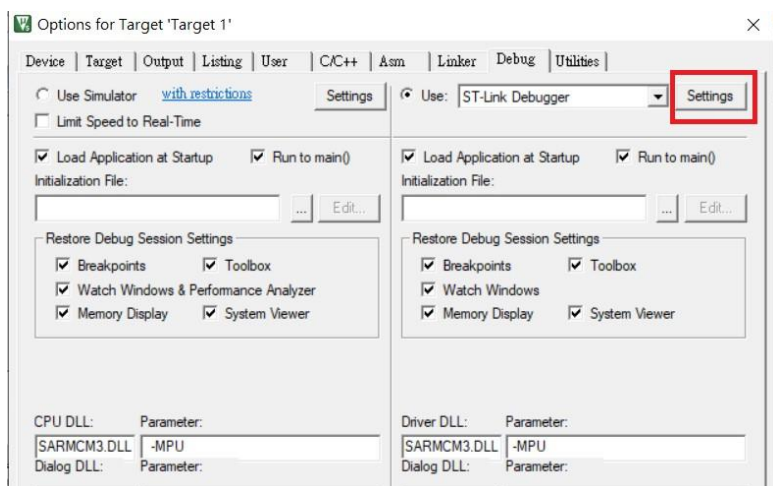
3.5. 仿真器配置

3.5.1. 使用 ST-Link 仿真

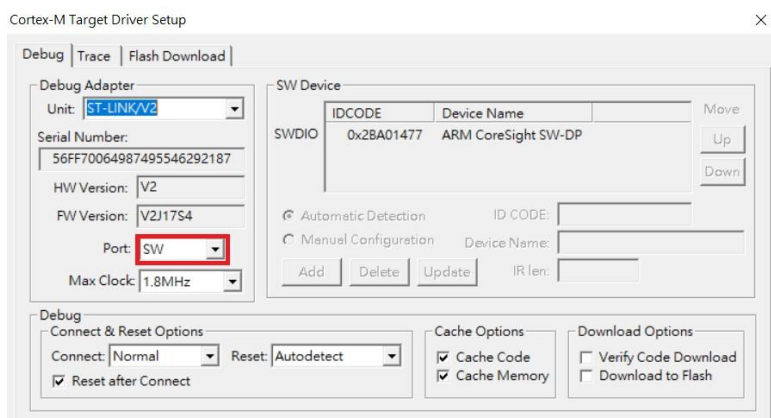
- 1) MG32F10x是ARM Cortex-M3的芯片，所以可以使用各种支持Cortex-M3的调试器调试程序（比如：JLink，ULink，STLink 和 CMSIS-DAP 等）。下面以 ST-link 为例演示 MG32F10x 的调试配置。
- 2) 将 ST-Link 连接到电脑，使用 ST-link 的 SWD 接口与 MG32F10x 芯片连接，并给芯片上电。
- 3) 打开 Options for Target 对话框，切换到 Debug 选项卡，选择使用 ST-Link 调试器。



- 4) 配置调试器选项。

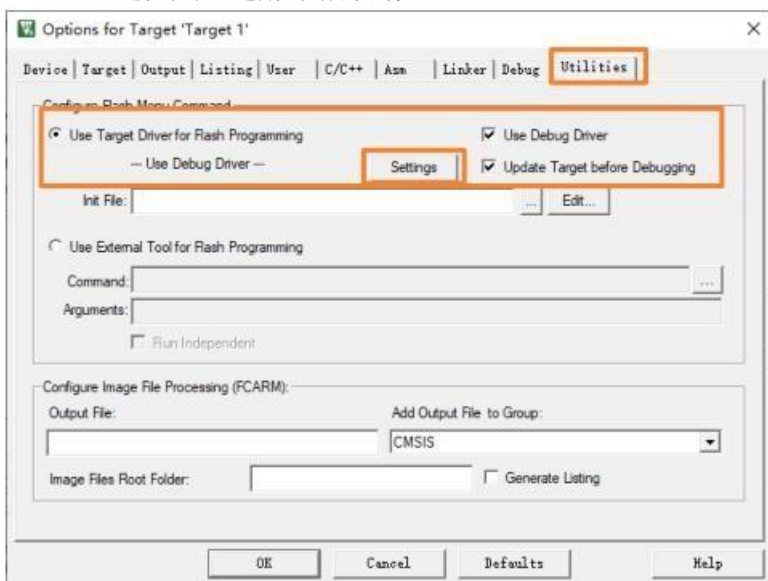


选择 SW 接口，可以在右侧看到 ST-Link 检测到了 MG32F10x 芯片。

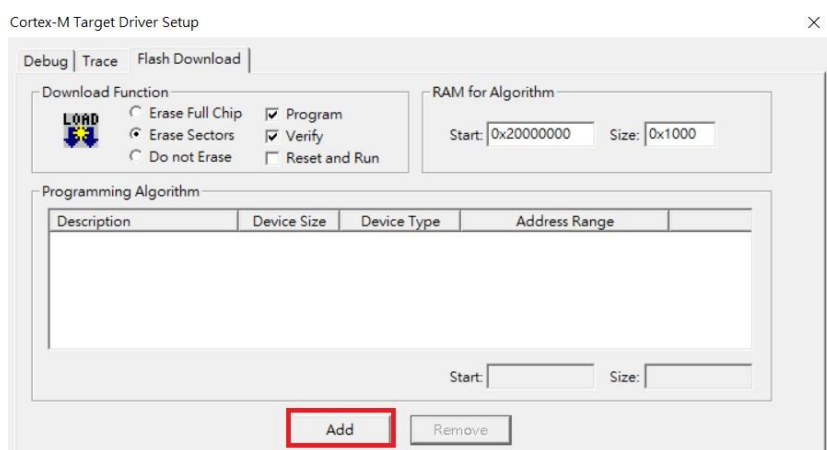
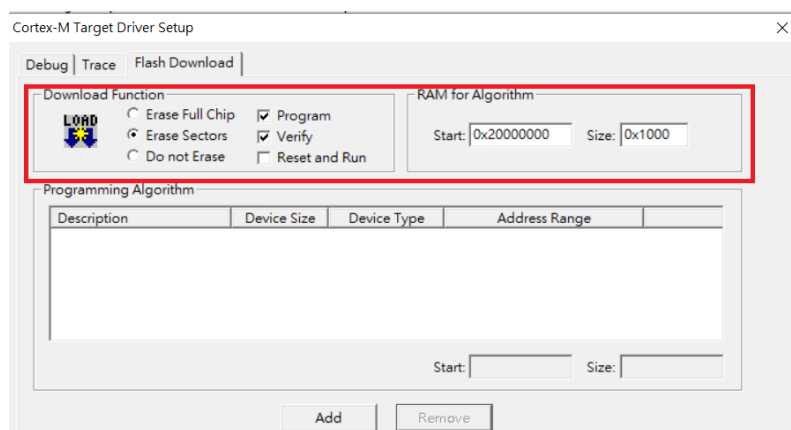


然后点击确定。

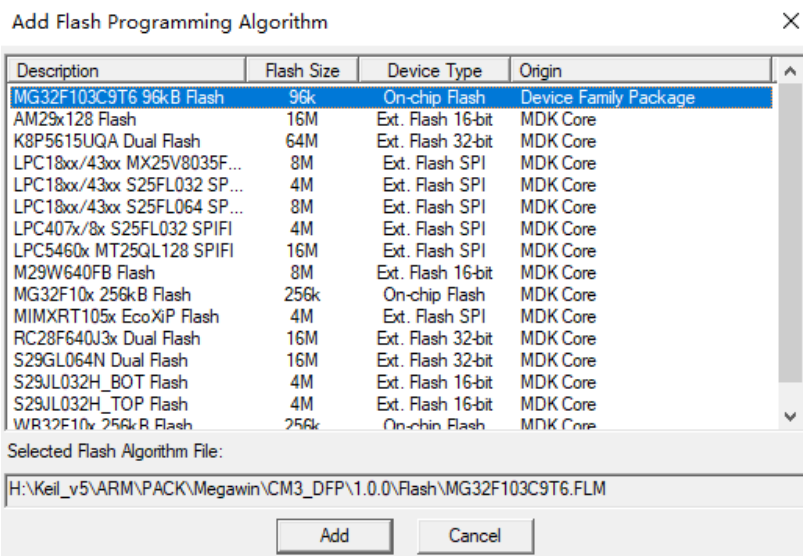
- 5) 在 Utilities 选项卡中，进行如图所示的设置。

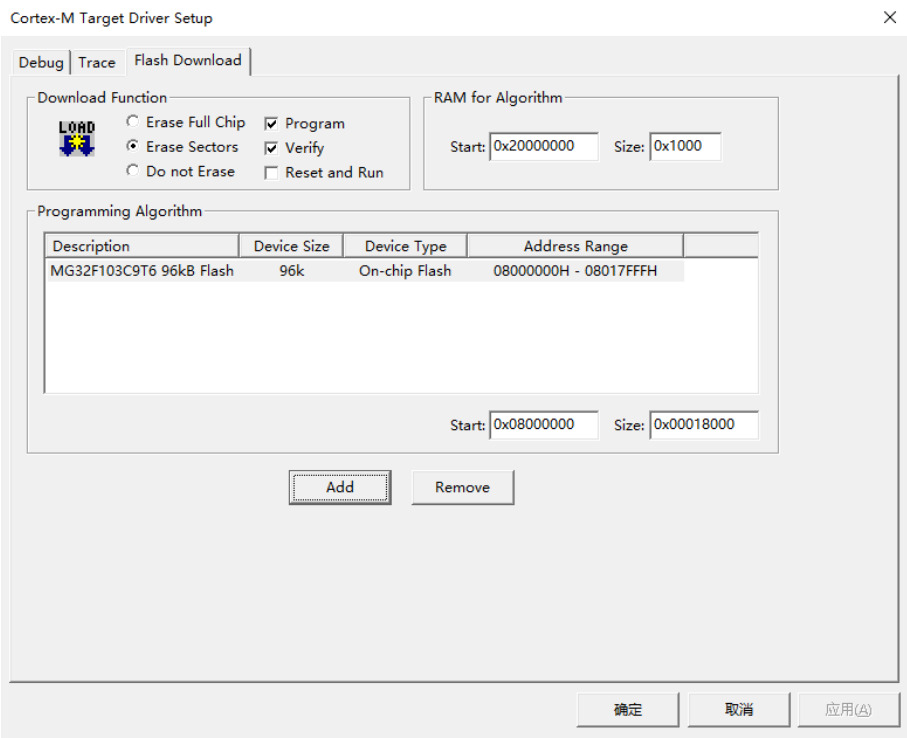


然后点击 Settings 按钮，打开烧录算法配置对话框，进行如图所示的配置。



找到对应型号和封装的烧录算法，并点击 Add。





最后点击确定，至此，用户可以编译，下载和调试该程序了。

3.5.2. 使用 J-Link 仿真

- 1) 在 J-Link 安装目录（安装目录自选，演示机器是安装在 H 盘）
H:\Program Files (x86)\SEGGER\JLink_V635g\Devices 中新建一个名为 Megawin 的文件夹；并在 Megawin 文件夹中新建一个名为 MG32F10x 的文件夹。
- 2) 将 MG32F10x 的烧录算法文件复制到 H:\Program Files (x86)\SEGGER\JLink_V635g\Devices\Megawin\MG32F10x 文件夹中，烧录算法文件在安装 Megawin.CM3_DFP.1.0.0.pack 后会位于安装路径
\Keil_v5\ARM\PACK\Megawin\CM3_DFP\1.0.0\Flash

此电脑 > 加速盘 (H:) > Keil_v5 > ARM > PACK > Megawin > CM3_DFP > 1.0.0 > Flash

名称	修改日期	类型	大小
MG32F103C9T6.FLM	2021/11/2 10:54	FLM 文件	15 KB
MG32F103CBT6.FLM	2021/11/2 10:54	FLM 文件	15 KB
MG32F103RBT6.FLM	2021/11/2 10:54	FLM 文件	15 KB
MG32F104RCT6.FLM	2021/11/2 10:54	FLM 文件	15 KB

Program Files (x86) > SEGGER > JLink_V635g > Devices > Megawin > MG32F10x

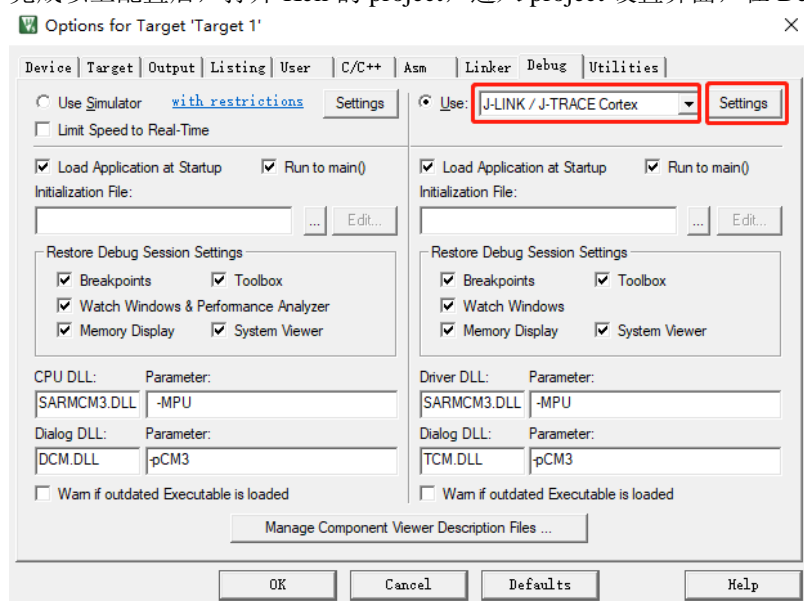
名称	修改日期	类型	大小
MG32F103C9T6.FLM	2021/11/2 10:54	FLM 文件	15 KB
MG32F103CBT6.FLM	2021/11/2 10:54	FLM 文件	15 KB
MG32F103RBT6.FLM	2021/11/2 10:54	FLM 文件	15 KB
MG32F104RCT6.FLM	2021/11/2 10:54	FLM 文件	15 KB

- 3) 在 H:\Program Files (x86)\SEGGER\JLink_V635g\JLinkDevices.xml 文件中添加 MG32F10x 的信息，并保存。

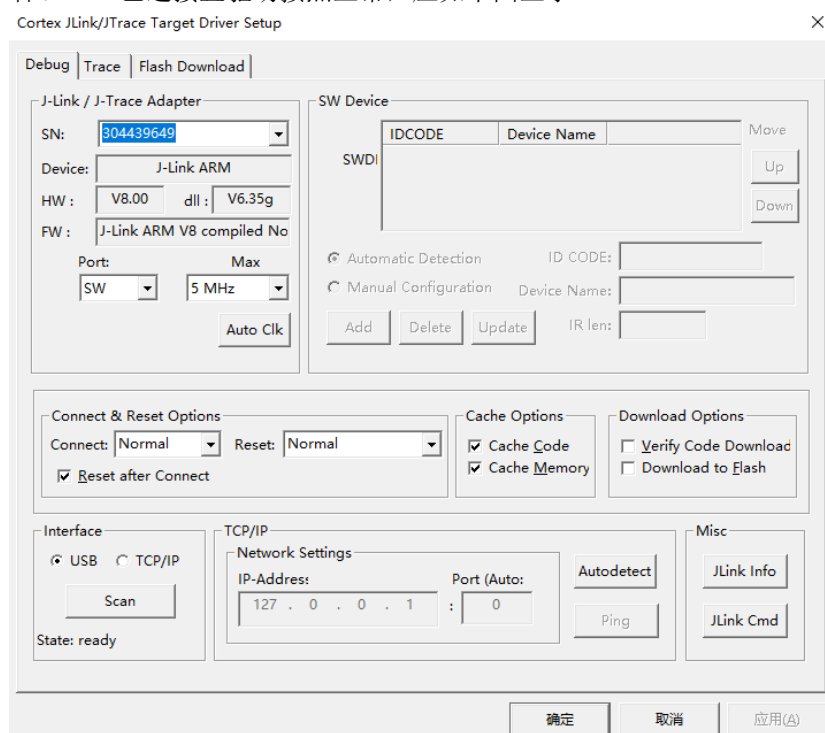
```
<Device>
  <ChipInfo Vendor="Megawin" Name="MG32F103C9T6" Core="JLINK_CORE_CORTEX_M3" WorkRAMAddr="0x20000000" WorkRAMSize="0x1000" />
  <FlashBankInfo Name="Internal Flash" BaseAddr="0x08000000" MaxSize="0x18000" Loader="Devices\Megawin\MG32F10x\MG32F103C9T6.FLM"
  LoaderType="FLASH_ALGO_TYPE_CMSIS" AlwaysPresent="1" />
</Device>
<Device>
  <ChipInfo Vendor="Megawin" Name="MG32F103CBT6" Core="JLINK_CORE_CORTEX_M3" WorkRAMAddr="0x20000000" WorkRAMSize="0x1000" />
  <FlashBankInfo Name="Internal Flash" BaseAddr="0x08000000" MaxSize="0x20000" Loader="Devices\Megawin\MG32F10x\MG32F103CBT6.FLM"
  LoaderType="FLASH_ALGO_TYPE_CMSIS" AlwaysPresent="1" />
</Device>
```

```
<Device>
  <ChipInfo Vendor="Megawin" Name="MG32F103RBT6" Core="JLINK_CORE_CORTEX_M3" WorkRAMAddr="0x20000000" WorkRAMSize="0x1000" />
  <FlashBankInfo Name="Internal Flash" BaseAddr="0x08000000" MaxSize="0x20000" Loader="Devices\Megawin\MG32F10x\MG32F103RBT6.FLM"
LoaderType="FLASH_ALGO_TYPE_CMSIS" AlwaysPresent="1" />
</Device>
<Device>
  <ChipInfo Vendor="Megawin" Name="MG32F104RCT6" Core="JLINK_CORE_CORTEX_M3" WorkRAMAddr="0x20000000" WorkRAMSize="0x1000" />
  <FlashBankInfo Name="Internal Flash" BaseAddr="0x08000000" MaxSize="0x40000" Loader="Devices\Megawin\MG32F10x\MG32F104RCT6.FLM"
LoaderType="FLASH_ALGO_TYPE_CMSIS" AlwaysPresent="1" />
</Device>
<Device>
  <ChipInfo Vendor="Megawin" Name="MG32F103C9T6" Core="JLINK_CORE_CORTEX_M3" WorkRAMAddr="0x20000000" WorkRAMSize="0x1000" />
  <FlashBankInfo Name="Internal Flash" BaseAddr="0x08000000" MaxSize="0x18000" Loader="Devices\Megawin\MG32F10x\MG32F103C9T6.FLM" LoaderType="FLASH_ALGO_TYPE_CMSIS" AlwaysPresent="1" />
</Device>
<Device>
  <ChipInfo Vendor="Megawin" Name="MG32F103CBT6" Core="JLINK_CORE_CORTEX_M3" WorkRAMAddr="0x20000000" WorkRAMSize="0x1000" />
  <FlashBankInfo Name="Internal Flash" BaseAddr="0x08000000" MaxSize="0x20000" Loader="Devices\Megawin\MG32F10x\MG32F103CBT6.FLM" LoaderType="FLASH_ALGO_TYPE_CMSIS" AlwaysPresent="1" />
</Device>
<Device>
  <ChipInfo Vendor="Megawin" Name="MG32F103RBT6" Core="JLINK_CORE_CORTEX_M3" WorkRAMAddr="0x20000000" WorkRAMSize="0x1000" />
  <FlashBankInfo Name="Internal Flash" BaseAddr="0x08000000" MaxSize="0x20000" Loader="Devices\Megawin\MG32F10x\MG32F103RBT6.FLM" LoaderType="FLASH_ALGO_TYPE_CMSIS" AlwaysPresent="1" />
</Device>
<Device>
  <ChipInfo Vendor="Megawin" Name="MG32F104RCT6" Core="JLINK_CORE_CORTEX_M3" WorkRAMAddr="0x20000000" WorkRAMSize="0x1000" />
  <FlashBankInfo Name="Internal Flash" BaseAddr="0x08000000" MaxSize="0x40000" Loader="Devices\Megawin\MG32F10x\MG32F104RCT6.FLM" LoaderType="FLASH_ALGO_TYPE_CMSIS" AlwaysPresent="1" />
</Device>
```

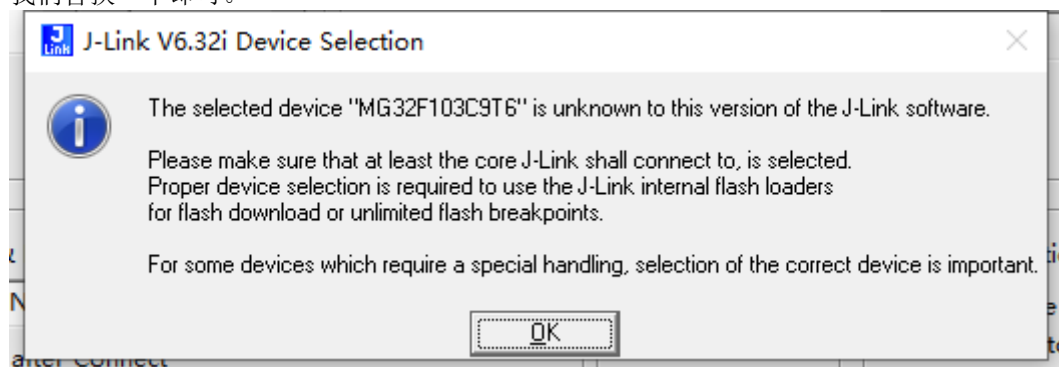
完成以上配置后，打开 Keil 的 project，进入 project 设置界面，在 Debug 选项卡中选择 J-Link，并点击 Settings



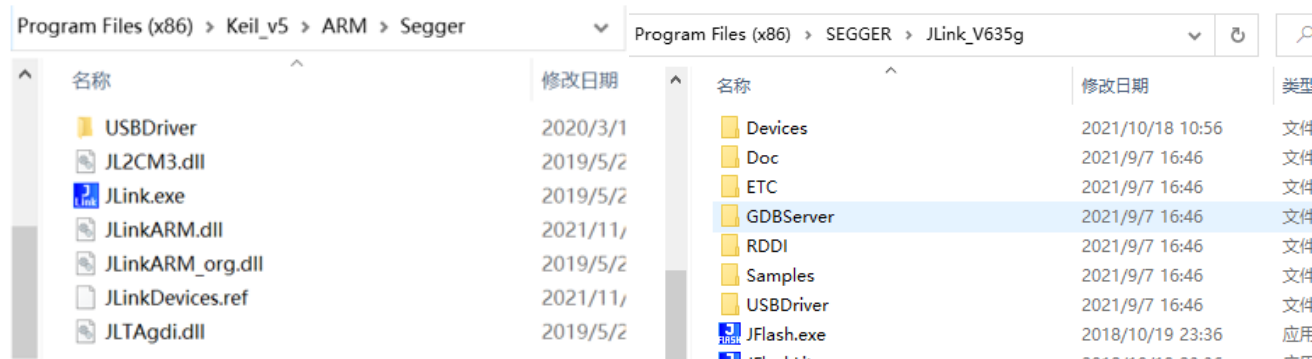
若 J-Link 已连接且驱动按照正常，应如下图所示



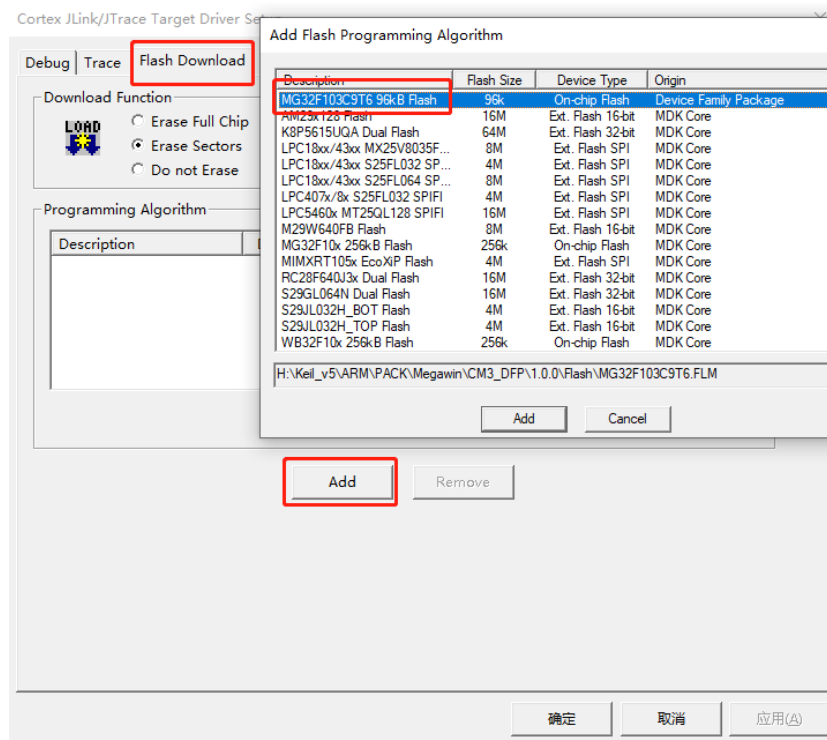
若点击 Settings 时提示类似下图 The selected device xxx is unknown, 这应该是 Keil 版本自带的 J-link 驱动不匹配造成。我们替换一下即可。



下左图是 Keil 自带的 J-Link 驱动目录，右图则是第一章我们提到的自己安装的 J-Link 目录，我们把自己安装的 J-link 目录下的所有东西全部替换到左图的位置下，覆盖旧有的数据即可。覆盖完成，重新打开 Debug 的 Settings，此时就应该能正常显示了。



点击 Flash Download，选择实际使用的芯片即可，最后点击确定即可使用 J-Link 进行仿真。



4. 硬件布线建议

4.1. 印制电路板

出于技术的考虑，最好使用有专门独立的接地层(VSS)和专门独立的供电层(VDD)的多层印制电路板，这样能提供好的耦合性能和屏蔽效果。很多应用中，受经济条件限制不能使用这样的印制电路板，那么就需要保证一个好的接地和供电的结构。

4.2. 器件位置

为了减少 PCB 上的交叉耦合，设计版图时就需要根据各自对 EMI 影响的不同，而把不同的电路分开。比如，大电流电路、低电压电路以及数字器件等。

4.3. 接地和供电 (VSS/VDD)

每个模块(噪声电路、敏感度低的电路、数字电路)都应该单独接地，所有的地最终都应应在一个点上连到一起。尽量避免或者减小回路的区域。为了减少供电回路的区域，电源应该尽量靠近地线，这是因为，供电回路就像个天线，成为 EMI 的发射器和接收器。PCB 上没有器件的区域，需要填充为地，以提供好的屏蔽效果(特别是对单层 PCB，尤其如此)。

4.4. 去耦合

所有的引脚都需要适当地连接到电源。这些连接，包括焊盘、连线 and 过孔应该具备尽量小的阻抗。通常采用增加连线宽度的办法，包括在多层 PCB 中使用单独的供电层。同时，MG32F10x 上每个电源引脚应该并联去耦合的滤波陶瓷电容 C(100nF)和化学电容 C(10μF)。这些电容应该尽可能的靠近电源/地引脚；或者在 PCB 另一层，处于电源/地引脚之下。典型值一般从 10nF 到 100nF，具体的容值取决于实际应用的需要。图 1 显示了这样的电源/地引脚的典型布局。

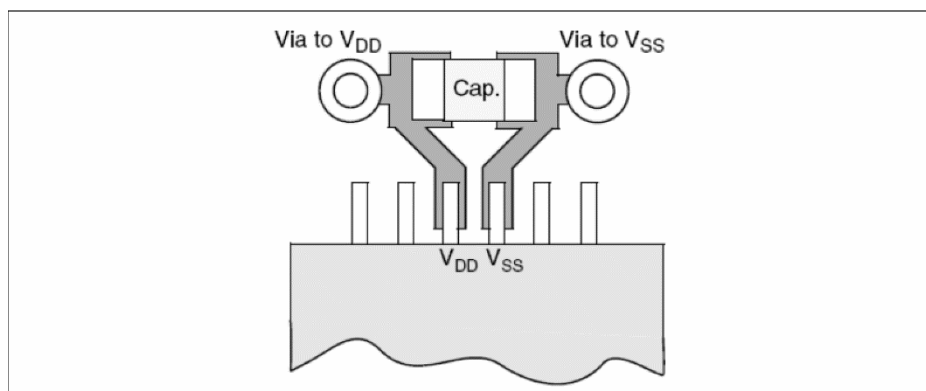


图 1 VDD /VSS 引脚的典型布局

4.5. 供电方案

电路由稳定的电源 VDD 供电。

- 注意：

- 如果使用 ADC，VDD 的范围必须在 2.4V 到 3.6V 之间
- 如果没有使用 ADC，VDD 的范围为 2V 到 3.6V

- VDD 引脚必须连接到带外部稳定电容(11 个 100nF 的陶瓷电容和一个钽电容(最小值 4.7μF，典型值 10μF)) 的 VDD 电源。

- VBAT 引脚必须被连接到外部电池($1.8V < V_{BAT} < 3.6V$)。如果没有外部电池，这个引脚必须和 100nF 的陶瓷电容一起连接到 VDD 电源上

- VDDA 引脚必须连接到两个外部稳定电容(10nF 陶瓷电容+1μF 钽电容)。

- VREF+ 引脚可以连接到 VDDA 外部电源。如果在 VREF+ 上使用单独的外部参考电压，必须在引脚上连接一个 10nF 和一个 1μF 的电容。在所有情况下，VREF+ 必须在 2.4V 和 VDDA 之间。

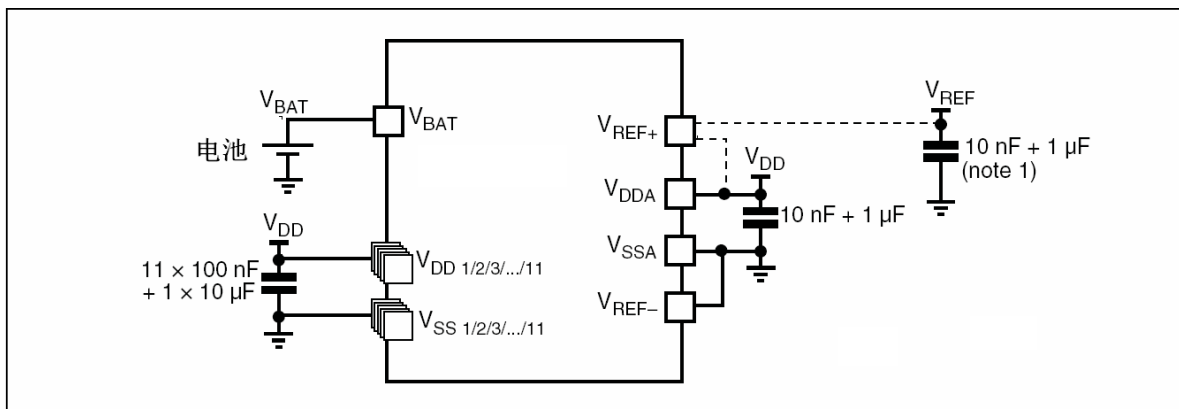


图 2 供电方案

1. 可选。如果在 VREF+ 上使用单独的外部参考电压，必须连接两个电容(10nF 和 1μF)。
2. VREF+ 连接到 VDDA 或 VREF+ 。

4.6. 其他信号

实际应用中，关注以下几点可以提高 EMC 性能：

- 那些受暂时的干扰会影响运行结果的信号(比如中断或者握手抖动信号，而不是 LED 命令之类的信号)。

对于这些信号，信号线周围铺地，缩短走线距离，消除邻近的噪声和敏感的连线都可以提高 EMC 性能。

对于数字信号，为有效地区别 2 种逻辑状态，必须能够达到最佳可能的信号特性余量(译注：

尽可能抬高逻辑‘1’的高电平，拉低逻辑‘0’的低电平)。推荐使用慢速施密特触发器来消除寄生状态。

- 布线时尽可能满足 3W 原则，尽可能远离相邻走线减小耦合减少干扰。如果 ADC，CMP 对精度要求高，一定要做包地处理。

- 噪声信号(时钟等)。
- 敏感信号(高阻等)。

4.7. 未用到的 IO 及其性

所有微控制器都为各种应用而设计，而通常的应用都不会用到所有的微控制器资源。

为了提高 EMC 性能，不用的时钟、计数器或者 I/O 管脚，需要做相应处理，比如，I/O 端口应该被设置为‘0’或‘1’(对不用到的 I/O 引脚上拉或者下拉)；没有用到的模块应该禁止或者“冻结”。

4.8. 时钟

尽可能减少 LSE 和 HSE 之间的平行走线。下图 3 中 LSE 和 HSE 从焊盘引出时，走线直接分开。

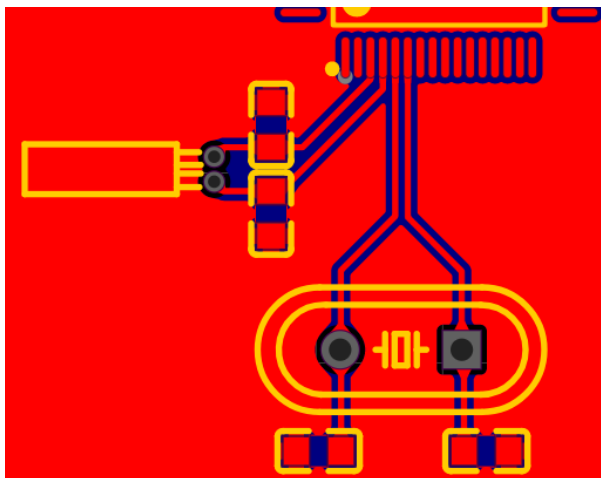


图 3 LSE 和 HSE 布线

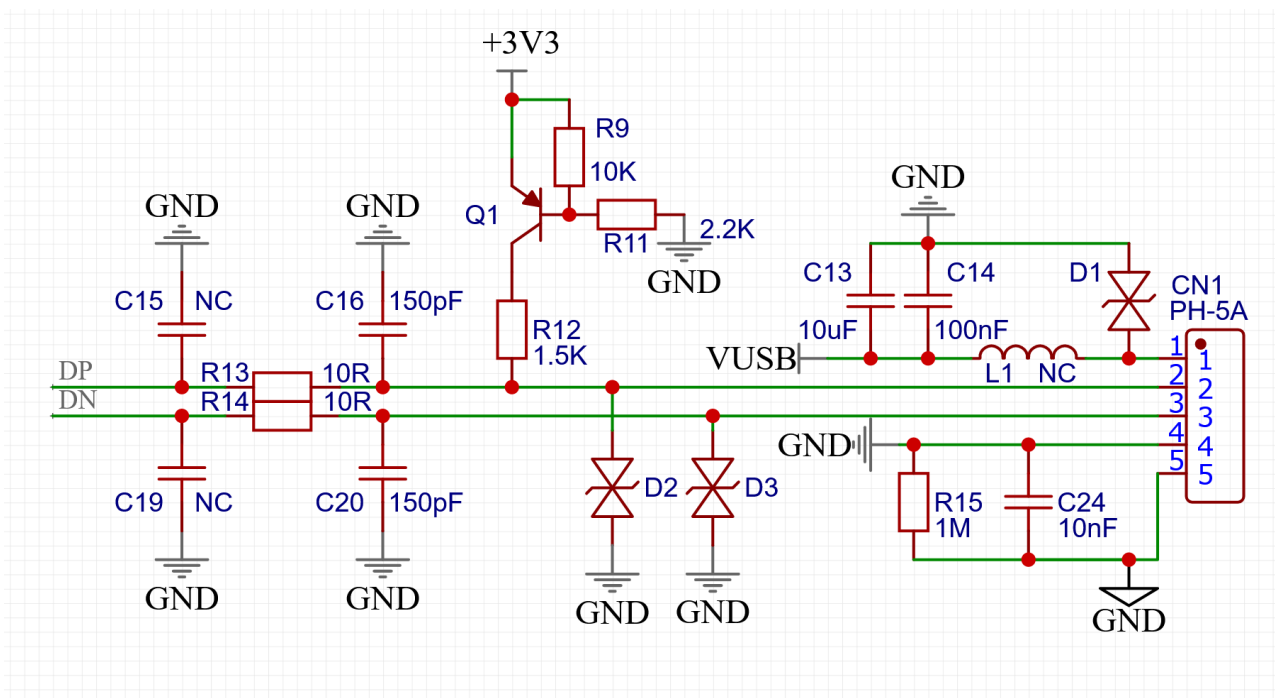
4.9. 模拟信号

模拟信号跟数字信号走线分开，并且模拟信号需要用地线屏蔽，这样可以尽可能保证采样的精度。

4.10. EMI

1. 确保电源额定值适用于应用，并使用去耦电容器进行优化。
2. 在电源上提供足够的滤波电容器。大容量/旁路和去耦电容器应具有低等效串联电感（ESL）。
3. 如果布线层上有可用空间，则创建地平面。将这些接地区域利用过孔连接到地平面。
4. 电流回路尽可能小。添加尽可能多的去耦电容器。
5. 差分线对要保持线长匹配，否则会导致时序偏移、降低信号质量以及增加 EMI。
6. 差分走线要求在同一板层上，因为不同层之间的阻抗、过孔等差别会降低差模传输的效果而引入共模噪声。
7. 高速信号线不要有过孔，确保背面地平面完整，同时缩短走线距离，远离相邻走线。如果 usb 接口芯片需串联端电阻或者 D 线接上拉电阻时，务必将这些电阻尽可能的靠近芯片放置。
8. MCU 每一个 VDD 的电源引脚尽可能留 2 个电容的位置 1uF 0.1uF
9. SPI 或 IIC 等通讯线上，每个信号线上串一颗 10R 左右的电阻，预留一个 120pf 的电容。信号线尽量够短。
10. 晶振走线要足够短，晶振背面不要走信号线，确保晶振地平面完整。布局允许的情况下，再晶振周围多打地过孔。

原理图设计参考:



1. DP 线上的上拉需要用三极管来控制。确保上路电路到 DP 线足够短。
2. D1 D2 D3 TVS 管需要靠接口放置。
3. PIN1 是电源，在靠近 PIN1 处防止一颗 10uF 和 10nF 用作滤波。
4. DP DN 到 USB 接口的距离尽可能短。

5. 外设移植

5.1. 移植前的准备

用户可以使用 MG32F10x 替代 STM32F10x 同类型的芯片，进行学习或者产品程序移植，大幅度的降低用户使用成本。

另一方面，MG32F10x 系列是独立设计的芯片，底层设计与库函数封装势必与 STM32F10x 有很多不同之处，在编写程序时不能简单粗暴的将在 STM32F10x 上实现的程序直接移植到 MG32F10x 上，要根据 MG 驱动库进行简单的修改，需要用户付出较低的学习和时间成本。但是不必担心，需要修改的部分仅涉及到与 MCU 寄存器访问相关的部分代码，而软件算法、上层架构、变量等主要代码内容是不需要改动的，也就是说，需要修改的，仅仅包含使用到的外设的初始化、中断函数结构替换以及代码中涉及到的外设数据读写的驱动函数而已，修改量并不会很大。

已经熟悉 STM32F10x 开发的朋友可以根据 MG32F10x 参考手册、固件库指南以及固件库给的例程快速上手。

移植软件时，请注意按照开发环境搭建章的[建立一个工程](#)节，做好头文件的关联。

[注意]: 开发包提供的范例程序几乎都默认使用外振作为时钟源，因此若用户需要用内振进行开发，请参照软件移植章的 [RCC](#) 节进行修改。

5.2. ADC

关于 ADC 的初始化部分,可参照 ADC 的相关范例程序,整体替换掉 STM32F10x 的相关 ADC 初始化和 ADC 值获取的底层驱动程序,使用到的驱动位于 mg32f10x_pwr.c、mg32f10x_adc.c、mg32f10x_rcc.c、mg32f10x_anctl.c、mg32f10x_gpio.c, 请注意添加驱动文件。

```

ADC_AnalogWatchdog
ADC_ChipTemperature
ADC_DMA
ADC_DMA_Injected
ADC_ExtLinesTrigger
ADC_GetVDD
ADC_Interrupt
ADC_TIMTrigger_AutoInjection

/* ADC configuration -----*/
PWR_UnlockANA();
ANCTL_SARADCCmd(ENABLE);
PWR_LockANA();
ADC_InitStructure.ADC_ScanConvMode = DISABLE;
ADC_InitStructure.ADC_ContinuousConvMode = ENABLE;
ADC_InitStructure.ADC_ExternalTrigConv = ADC_ExternalTrigConv_None;
ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;
ADC_InitStructure.ADC_NbrOfChannel = 1;
ADC_Init(&ADC_InitStructure);
/* ADC regular channel3 configuration */
ADC-RegularChannelConfig(ADC_Channel_3, 1, ADC_SampleTime_7Cycles5);
/* Enable EOC interrupt */
ADC_ITConfig(ADC_IT_EOC, ENABLE);
/* Enable ADC external trigger conversion */
ADC_ExternalTrigConvCmd(ENABLE);
/* Enable ADC */
ADC_Cmd(ENABLE);
/* Start ADC calibration */
ADC_StartCalibration();
/* Check the end of ADC calibration */
while(ADC_GetCalibrationStatus());
/* Enable ADC reset calibration register */
ADC_ResetCalibration();
/* Check the end of ADC reset calibration register */
while(ADC_GetResetCalibrationStatus());

/* Start ADC Software Conversion */
ADC_SoftwareStartConvCmd(ENABLE);

/* Waiting for EOC */
while(ADC_GetFlagStatus(ADC_FLAG_EOC) != SET){}
X= ADC_GetADValue(ADC->DR); //读取 ADC 值

```

[注意]: 请不要用直接读 ADC 的数据寄存器的方式来读取 ADC 值, 这样读出来的值是错误的, 请使用库函数 ADC_GetADValue(uint16 t data);来读取。

中断函数结构模板:

```
void ADC_IRQHandler(void)
{
    if(ADC_GetITStatus(ADC_IT_EOC) != RESET)
    {
        ADC_ClearITPendingBit(ADC_IT_EOC);
        printf("\rADC Channel 3: %-5d", ADC_GetADValue(ADC->DR));
    }
}
```

此外，对于模拟信号的布线，建议如前面硬件布线建议章节中所说，模拟信号跟数字信号走线分开，并且模拟信号需要用地线屏蔽，这样可以尽可能保证采样的精度。

5.3. ANCTL（模拟控制器）

5.3.1. CMP

模拟比较器是 STM32F10x 系列里没有，但是也非常实用的外设，其范例程序在 ANCTL\CMP。需要的用户可在里面调用，使用到的驱动位于 mg32f10x_pwr.c、mg32f10x_rcc.c、mg32f10x_anctl.c、mg32f10x_gpio.c，请注意添加驱动文件。

```
PWR_UnlockANA();
ANCTL_CMPAConfig(CMPA_PSEL_PB4, CMPA_NSEL_PB6);
ANCTL_CMPACmd(ENABLE);
PWR_LockANA();
```

```
CMP_Result = ANCTL_CMPAGetOutputLevel();//读取比较值
```

通过以上代码即可设置比较器的正极和负极，并完成初始化 CMP。

[注意]: CMP 没有中断功能。

5.3.2. DCSS

DCSS 是时钟安全系统，在 HSE 稳定后开始工作，在 HSE 停止时停止工作，可以用作外振异常的检测器，因此用户若有外振异常检测功能的需要，可以用以下代码整体替换掉 STM32F10x 的相关初始化代码，使用到的驱动位于 mg32f10x_pwr.c、mg32f10x_rcc.c、mg32f10x_anctl.c、mg32f10x_gpio.c，请注意添加驱动文件。

```
RCC_DCSSCLKCmd(ENABLE);
ANCTL_ClockSecuritySystemCmd(ENABLE);
```

中断函数结构模板：

```
void NMI_Handler(void)
{
    if (ANCTL_GetITStatus(ANCTL_IT_DCSS) != RESET)
    {
        ANCTL_ClearITPendingBit(ANCTL_IT_DCSS);

        while (1)
        {
            /* LED2 blink */
            GPIO_ToggleBits(GPIOB, GPIO_Pin_13);
            Delay(100000);
        }
    }
}
```

通过以上代码即可配置完成 DCSS 功能。

5.4. BKP

BKP 是数据备份寄存器，可以存放一些备份数据，由于在关闭 VDD 时，备份域（BKP）仍由 VBAT 供电，因此如果电池连接到 VBAT 引脚，则其内容不会丢失。使用非常简单，使能后直接存值即可，使用到的驱动位于 mg32f10x_pwr.c、mg32f10x_rcc.c、mg32f10x_bkp.c，请注意添加驱动文件。

```
PWR_BackupAccessCmd(ENABLE);
BKP->DR1 = 0x55AA;    //将 0x55AA 存入 BKP 备份域中
```

5.5. CRC

CRC 的初始化非常简单，初始化完成后直接调用库即可进行 CRC 校验。用户也可以用下方 Deinit() 整体替换掉 STM32F10x 的相关初始化代码，需要计算的地方，改用相应的 CRC 校验库函数即可，使用到的驱动位于 mg32f10x_crc.c、mg32f10x_rcc.c、mg32f10x_sfm.c、computeBytes.c、computeHalfWords.c、computeWords.c，请注意添加驱动文件。

```
CRC_SFM_DeInit();
result1 = CRC8_ComputeBytes(bytes, _countof(bytes));
```

5.6. DMAC

直接存储器存取(DMA)用来提供在外设和外设之间、外设和存储器之间或者存储器和存储器之间的高速数据传输。无须 CPU 干预，数据可以通过 DMA 快速地移动，这就节省了 CPU 的资源来做其他操作。用户可根据实际使用的组合，整体替换掉 STM32F10x 的相关 DMA 初始化功能代码，使用到的驱动位于 mg32f10x_rcc.c、mg32f10x_dmac.c，请注意添加驱动文件。

实际的移植代码可以参考 DMAC 相关范例程序。

- DMAC_MemoryToMemory
- DMAC_MemoryToUart
- DMAC_MemoryToUart_MultiBlock
- DMAC_UartToMemory
- DMAC_UartToMemory_MultiBlock
- DMAC_UartToUart_MultiBlock

```
DMAC_ChannelCmd(DMAC1, DMAC_Channel_0, ENABLE);
```

初始化完成后，调用上方的库函数，修改对应的通道和 DMA 模块即可启动 DMA 传输。

中断函数结构模板：

```
void DMAC1_IRQHandler(void)
{
    if(DMAC_GetITStatus(DMAC1, DMAC_Channel_0, DMAC_IT_BLOCK) != RESET)
    {
        DMAC_ClearITPendingBit(DMAC1, DMAC_Channel_0, DMAC_IT_BLOCK);
        printf("DMA block transfer complete.\r\n");
    }

    if(DMAC_GetITStatus(DMAC1, DMAC_Channel_0, DMAC_IT_TFR) != RESET)
    {
        DMAC_ClearITPendingBit(DMAC1, DMAC_Channel_0, DMAC_IT_TFR);
        printf("DMA transfer complete.\r\n");
        if(memcmp(memDst, memSrc, sizeof(memSrc)) == 0) {
```

```
        printf("DMA transfer success!!!\r\n");
    }
    else {
        printf("DMA transfer failed!!!\r\n");
    }
}

if(DMAC_GetITStatus(DMAC1, DMAC_Channel_0, DMAC_IT_ERR) != RESET)
{
    DMAC_ClearITPendingBit(DMAC1, DMAC_Channel_0, DMAC_IT_ERR);
    printf("DMA transfer error!!!\r\n");
}
}
```

[注意]: 建议不要使用 DMAC 的方式进行 ADC 数据采集且 ADC 仅支持 single DMA 传输。

5.7. EXTI

外部中断 EXTI 支持不同中断或事件，也支持上下沿触发或者双沿触发。配置方法如下，可参考 EXTI_Config 范例。用户可根据实际使用的组合，整体替换掉 STM32F10x 的相关 EXTI 初始化功能代码，使用到的驱动位于 mg32f10x_rcc.c、mg32f10x_exti.c、mg32f10x_gpio.c，请注意添加驱动文件。

```
/* Connect EXTI0 Line to PA0 pin */
GPIO_EXTILineConfig(GPIO_PortSourceGPIOA, GPIO_PinSource0);

/* Configure EXTI0 line */
EXTI_InitStructure.EXTI_Line = EXTI_Line0;
EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising;
EXTI_InitStructure.EXTI_LineCmd = ENABLE;
EXTI_Init(&EXTI_InitStructure);

/* Configure and enable EXTI0 interrupt */
NVIC_InitStructure.NVIC_IRQChannel = EXTI0_IRQn;
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
NVIC_Init(&NVIC_InitStructure);
```

将外部中断初始化部分代码替换为上面代码，并根据实际情况，选择触发事件沿和引脚即可。

中断函数结构模板：

```
void EXTI0_IRQHandler(void)
{
    if(EXTI_GetITStatus(EXTI_Line0) != RESET)
    {
        /* Toggle LED1 */
        GPIO_ToggleBits(GPIOB, GPIO_Pin_14);

        /* Clear the EXTI line 0 pending bit */
        EXTI_ClearITPendingBit(EXTI_Line0);
    }
}
```


不同的外部中断函数结构模板：


```
void EXTI9_5_IRQHandler(void)
{
    if(EXTI_GetITStatus(EXTI_Line6) != RESET)
    {
        /* Toggle LED2 */
        GPIO_ToggleBits(GPIOB, GPIO_Pin_13);

        /* Clear the EXTI line 6 pending bit */
        EXTI_ClearITPendingBit(EXTI_Line6);
    }
}
```


5.8. FMC（闪存器控制模块）

FMC 是闪存控制器，修改 Flash 内容就是靠这个外设了。关于 FMC 的范例程序有以下两个，第一个是计算 Flash 中的 CRC，有需要的话可以使用。第二个就是闪存的编程了。用户可根据情况在操作 Flash 的程序部分，替换掉 STM32F10x 的相关 Flash 读写功能代码，使用到的驱动位于 mg32f10x_pwr.c、mg32f10x_rcc.c、mg32f10x_anctl.c、mg32f10x_fmc.c，请注意添加驱动文件。

 FMC_CalculateCRC

 FMC_Program

```
/* Before flash operation, FHSI must be enabled */
PWR_UnlockANA();
ANCTL_FHSICmd(ENABLE);
PWR_LockANA();


/* Erase the specified FLASH page */
FMC_ErasePage(TEST_PAGE_ADDR);
/* Clear page latch */
FMC_ClearPageLatch();
/* Write data to page latch */
for(iter = 0; iter < 64; iter++) {
    FMC->BUF[iter] = 0x12345678 + iter;
}
/* Program data in page latch to the specified FLASH page */
FMC_ProgramPage(TEST_PAGE_ADDR);
```

[注意]: 进行 Flash 的操作前，必须使能 FHSI 时钟，使能一次后不要禁用即可。


如上代码所示，擦除对应的 Flash 空间，然后清除页锁存，即可向 Flash 中写数据了。数据写完后要执行 FMC_ProgramPage(TEST_PAGE_ADDR);进行应用。


5.9. GPIO

关于 GPIO 的范例程序有以下 4 个，可在范例程序 GPIO 路径下找到，使用到的驱动位于 mg32f10x_rcc.c、mg32f10x_gpio.c，请注意添加驱动文件。

 GPIO_BitBand

 GPIO_I2C_Master

 GPIO_InputOutput

 GPIO_IOToggle

```
RCC_APB1PeriphClockCmd(RCC_APB1Periph_BMX1 |RCC_APB1Periph_GPIOA
|RCC_APB1Periph_QSPI, ENABLE);
GPIO_Init(GPIOA, GPIO_Pin_4 |GPIO_Pin_5 |GPIO_Pin_6 |GPIO_Pin_7, GPIO_MODE_AF
|GPIO_OTYPE_PP |GPIO_PUPD_NOPULL |GPIO_SPEED_HIGH |GPIO_AF5);
```

GPIO 的初始化非常简单，使能端口时钟后，仅需要调用 GPIO_Init()即可，第一个参数选择端口，第二个参数可以将要设置的引脚、功能、模式全部填进去，GPIO 就会自动配置完成，非常简便，不需要类似 STM32 的 HAL 库使用结构体做大量的代码。

5.10. I2C

MG32F10x 内有两个 I2C，可以控制所有 I2C 总线特定的序列、协议、仲裁和时序，可以支持标准模式、快速模式和高速模式。I2C1 提供多主模式功能，而且支持 SMBUS(系统管理总线) 协议。

范例程序中，I2C 路径下的范例程序有如下类型，使用到的驱动位于 mg32f10x_rcc.c、mg32f10x_i2c.c、mg32f10x_gpio.c，请注意添加驱动文件，若是使用到了 24C02，还需要添加 drv_eeprom_24c02.c。

- I2C_24C02
- I2C_24C02_Interrupt
- I2C_Master_HighSpeed
- I2C_MasterDMARx_SlaveDMATx
- I2C_MasterDMATx_SlaveDMARx
- I2C_Simulate_24C02
- I2C_SMBus_Master

可以看到范例程序类型比较多，也包含了主流 I2C 存储器 24C02 的范例，方便用户移植使用。用户可根据实际使用频率进行配置，再替换掉 STM32F10x 的相关 I2C 初始化功能代码。

```
eeprom_24c02_init();
```

```
result = eeprom_24c02_random_read(0x06, &rdata1);
printf("Read from [0x06] is 0x%02X\r\n", rdata1);
printf("eeprom_24c02_random_read() - TX_ABORT_SOURCE = %08X\r\n", result); // I2C 读取数据函数
if(result != 0) {
    errCode = 0x11;
    goto finish;
}
```

```
result = eeprom_24c02_byte_write(0x06, rdata1 + 1); //I2C 写数据函数
```

初始化完成后，即可使用库函数来读写 24C02 的数据。

中断函数结构模板：

```
// I2C2 Interrupt Routine
void I2C2_IRQHandler(void)
{
    uint32_t cmd;
    uint32_t tx_limit, rx_limit;

    if(I2C_GetITStatus(I2C2, I2C_IT_TX_ABORT) != RESET)
    {
        g_i2c_xfer_info.tx_abrt_source = I2C_GetTxAbortSource(I2C2);
        I2C2->INTR_MASK = I2C_INTR_STOP_DET; // Disable all interrupt except STOP_DET interrupt
        goto tx_aborted;
    }

    if(I2C_GetITStatus(I2C2, I2C_IT_RX_FULL) != RESET)
    {
        while((I2C_GetFlagStatus(I2C2, I2C_FLAG_RFNE) != RESET) && (g_i2c_xfer_info.rx_len))
        {
            *g_i2c_xfer_info.rx_buf = I2C_ReadData(I2C2);
            g_i2c_xfer_info.rx_buf++;
            g_i2c_xfer_info.rx_len--;
        }
    }
}
```

```
if(I2C_GetITStatus(I2C2, I2C_IT_TX_EMPTY) != RESET)
{
    tx_limit = 8 - I2C_GetTxFIFOLevel(I2C2);
    rx_limit = 8 - I2C_GetRxFIFOLevel(I2C2);
    while((tx_limit > 0) && (rx_limit > 0))
    {
        if((g_i2c_xfer_info.tx_len + g_i2c_xfer_info.rx_cmd_len) == 0) {
            I2C_ITConfig(I2C2, I2C_IT_TX_EMPTY, DISABLE);    // Disable TX Empty Interrupt
            break;
        }

        cmd = 0;
        if((g_i2c_xfer_info.tx_len + g_i2c_xfer_info.rx_cmd_len) == 1) {
            cmd |= I2C_DATA_CMD_STOP;
        }

        if(g_i2c_xfer_info.tx_len != 0)
        {
            I2C_WriteDataCmd(I2C2, cmd | *g_i2c_xfer_info.tx_buf);
            g_i2c_xfer_info.tx_buf++;
            g_i2c_xfer_info.tx_len--;
        }
        else if(g_i2c_xfer_info.rx_cmd_len != 0)
        {
            I2C_WriteDataCmd(I2C2, cmd | I2C_DATA_CMD_READ);
            g_i2c_xfer_info.rx_cmd_len--;
            rx_limit--;
        }
        tx_limit--;
    }
}


tx_aborted:
if(I2C_GetITStatus(I2C2, I2C_IT_STOP_DET) != RESET) {
    I2C_ClearITPendingBit(I2C2, 0xFFFF);    // Clear all interrupt flag
    I2C_ITConfig(I2C2, 0xFFFF, DISABLE);    // Disable all interrupt
    g_i2c_xfer_info.flag_complete = 1;
}
}
```


若是需要使用 DMA 也没问题，范例提供了使用 DMA 收发数据的范例程序，移植时将 DMA 和 I2C 相关调用复制到需要移植的项目即可。

5.11. I2S

MG32F10x 内置 1 个 I2S 总线接口，支持多种音频传输协议，工作于主控模式，支持双通道输入和输出。提供主时钟（MCLK）和串行时钟（SCLK）以及帧时钟（WS）和串行数据（SD0/SD1）。

I2S 一般用于播放音频，因此也提供了录播音频的范例程序，参考范例程序 I2S 路径下的范例即可。用户可整体替换掉 STM32F10x 的相关 I2S 初始化功能代码，使用到的驱动位于 mg32f10x_gpio.c、mg32f10x_rcc.c、mg32f10x_i2c.c、mg32f10x_i2s.c，请注意添加驱动文件，若是运用到 es8316，还可以添加 drv_es8316.c、wav_data.c。

 I2S_PlayAudio

 I2S_RecordPlayAudio

```
BSP_I2S_Init();  
StartPlay();
```

根据以上两个函数，调整需要的功能和封包大小，就可以与 I2S 设备通信和播放音频了。

中断函数结构模板：

```
void I2S_IRQHandler(void)  
{  
    if (I2S_Channel_GetITStatus(1, I2S_IT_TXFE) != RESET)  
    {  
        I2S_Channel_WriteLeftData(1, (audio_data[audio_index + 1] << 8) | audio_data[audio_index]);  
        I2S_Channel_WriteRightData(1, (audio_data[audio_index + 3] << 8) | audio_data[audio_index + 2]);  
  
        audio_index += 4;  
        if(audio_index >= audio_data_length)  
        {  
            audio_index = 0;  
        }  
    }  
}
```

[注意]: 不建议用户使用 MG32F10x 做 USB audio 功能，因为 USB buffer 为 128 字节，不足以满足 audio 的需求。

5.12. IWDG（独立看门狗）

参考 IWDG 的范例程序，使能看门狗就可以了，看门狗的配置较为简单。用户可根据实际溢出频率进行配置，再替换掉 STM32F10x 的相关 I2C 初始化功能代码，使用到的驱动位于 mg32f10x_pwr.c、mg32f10x_anctl.c、mg32f10x_rcc.c、mg32f10x_iwdg.c、mg32f10x_gpio.c，请注意添加驱动文件。

```
/* Enable IWDG clock */  
RCC_AHBPeriphClockCmd(RCC_AHBPeriph_IWDG, ENABLE);  
  
/* Enable write access to IWDG_PR and IWDG_RLR registers */  
IWDG_WriteAccessCmd(IWDG_WriteAccess_Enable);  
  
/* IWDG counter clock: LSI/32 */  
IWDG_SetPrescaler(IWDG_Prescaler_32);  
while(IWDG_GetFlagStatus(IWDG_FLAG_PVU) != RESET);  
  
/* IWDG timeout is about 250ms */
```

```
IWDG_SetReload(LSI_FREQ / 32 * 0.250);
while(IWDG_GetFlagStatus(IWDG_FLAG_RVU) != RESET);

/* ATTENTION: It is best to reload IWDG counter when the RVU bit is 0. */
while(IWDG_GetFlagStatus(IWDG_FLAG_RVU) != RESET);
IWDG_ReloadCounter();

/* Enable IWDG */
IWDG_Enable();

/* ATTENTION: It is best to reload IWDG counter when the RVU bit is 0. */
if (IWDG_GetFlagStatus(IWDG_FLAG_RVU) == RESET) {
    IWDG_ReloadCounter();
}
```

需要注意目前 IWDG 存在的一些限制：

1. 一旦启用了 IWDG，即使发生复位也无法禁用它。

解决方法：在程序运行之前将 IWDG 超时设置配置为最大，并且之后要不断重载 IWDG 计数器。

2. 如果 IWDG 启动之后发生复位，那么 IWDG 域需要 3 个 LSI 时钟周期才能就绪。

解决方法：在 LSI 就绪后，配置 IWDG 之前，等待大概 1 毫秒。

3. 不断执行 IWDG reload counter 操作有几率无法 reload IWDG counter.

解决方法：在执行 reload IWDG counter 操作之前确保 RVU 位是 0。（RVU 位指示 reload counter 操作是否完成）

4. 使能看门狗后，Debug 功能会失控，若是仍需要进行调试，建议在 DBGMCU_CR 寄存器里使能 DBG_IWDG_STOP 以停止看门狗在调试中的动作。

5. 独立看门狗不支持产生中断，如需要产生中断，请使用窗口看门狗

目前提供的范例程序已按照以上标准来做，用户移植时请注意复制。另外 LSI 时钟有一定偏差，不应做精准计时，需要多预留一些时间。

5.13. LED

LED 驱动控制器内含硬件 8 段 LED 驱动串行输出电路，模块时钟默认为内部系统时钟。LED 驱动外设也有相应的范例程序，模仿范例程序下 LED 路径的范例即可，使用到的驱动位于 mg32f10x_rcc.c、mg32f10x_led.c、mg32f10x_gpio.c，请注意添加驱动文件。

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_BMX2 | RCC_APB2Periph_LED, ENABLE);

/* Reset LED module */
LED_DeInit();




/* LED configuration */
LED->CYC = 200;
LED->ECO = 180;
LED->CON = (0x00 << 4);
LED->CON |= 0x01;

/* Infinite loop */
while (1)
{
    for(iter = 0; iter < 16; iter++)
    {
        LED_SetSegmentCode(0, table[iter]);
    }
}
```

初始化完成外设后，调用库函数 LED_SetSegmentCode() 就可以输出了，实际的显示数组需要根据实际数码管进行调整。

5.14. NVIC





NVIC 主要涉及中断优先级的控制和屏蔽，一般移植时候不会用到，但是如果需要的话，可以参考范例程序下的 NVIC 路径的以下程序。

-  NVIC_DMA_WFIMode
-  NVIC_IRQ_Mask
-  NVIC_IRQ_Priority

相关范例程序实现的功能可参考各范例下的 readme.txt。里面有详细的解释。

5.15. PWR（电源模式控制）

设置电源模式是很常见的设置，MG32F10x 系列提供 SLEEP、STANDBY 和 STOP 三种低功耗模式。都有相应的范例程序提供，使用到的驱动位于 mg32f10x_pwr.c、mg32f10x_anctl.c、mg32f10x_rcc.c，请注意添加驱动文件。

-  PWR_PVD
-  PWR_SLEEP
-  PWR_STANDBY
-  PWR_STOP

```
/* Enter SLEEP Mode */
PWR_EnterSLEEPMode(PWR_FCLK_Div2, PWR_EntryMode_WFI);
/* Enter STANDBY Mode */
PWR_EnterSTANDBYMode();
/* Enter STOP Mode */
PWR_EnterSTOPMode(PWR_STOPMode_LP4_S32KOFF, PWR_EntryMode_WFI);
```

根据实际需求，选择相应的模式即可。用户可在需要切换模式的程序位置替换掉 STM32F10x 的相关电源模式切换功能代码。

PVD 是电压监测器，可以监测 VDD 电压，用户可根据需要，设置监测电压。复制 PVD 范例程序中的 void PVD_Config(void) 内容，并根据实际情况，配置需要的电压，就可以实现电压监控，使用到的驱动同样位于 mg32f10x_pwr.c、mg32f10x_anctl.c、mg32f10x_rcc.c，请注意添加驱动文件。

```
/* Configure the PVD Level to 5 (refer to the electrical characteristics of
you device datasheet for more details) */
ANCTL_PVDLevelConfig(ANCTL_PVDLevel_5);
```

表 12.3: PVD 检测电压配置表

PLS	电压下降检测点	电压上升检测点
3'b000	2.14	2.25
3'b001	2.24	2.35
3'b010	2.34	2.45
3'b011	2.44	2.55
3'b100	2.54	2.65
3'b101	2.64	2.75
3'b110	2.74	2.85
3'b111	2.84	2.95

参考数据手册，选择电压监测档位。

中断函数结构模板：



```
void PVD_IRQHandler(void)
{
    if(EXTI_GetITStatus(EXTI_Line16) != RESET)
    {
        /* Clear the EXTI line 16 pending bit */
        EXTI_ClearITPendingBit(EXTI_Line16);

        /* Change LED2 status */
        GPIO_ToggleBits(GPIOB, GPIO_Pin_13);
    }
}
```

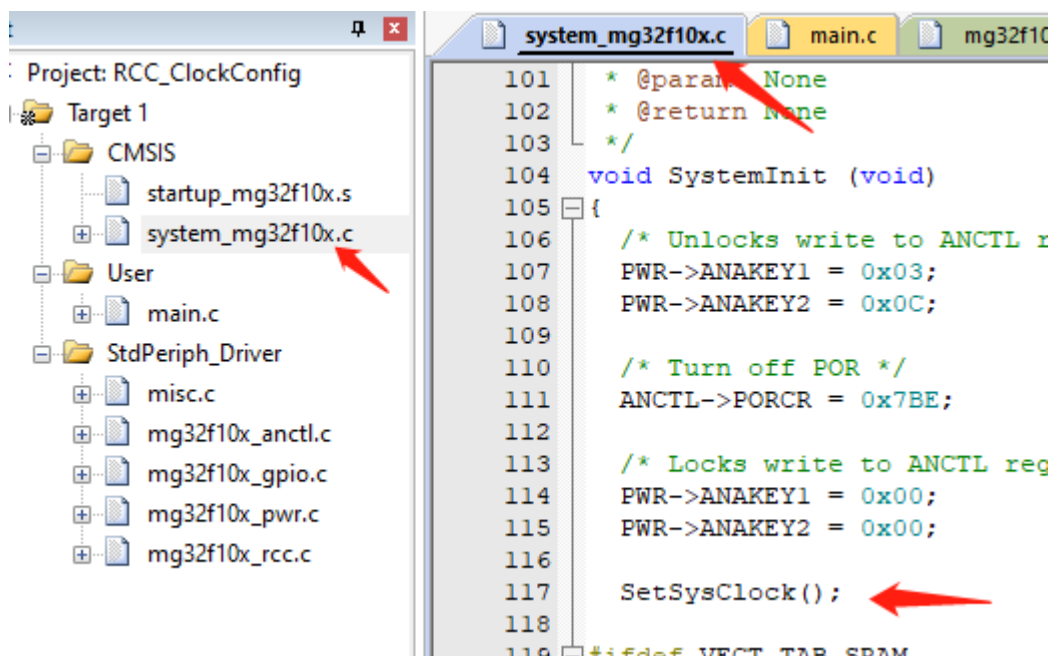
5.16. RCC

MG32F10x 系列时钟树提供了 4 种系统时钟源：MHSI(8MHz)内部振荡器时钟、FHSI(48MHz)内部振荡器时钟、PLL 时钟和 HSE 外部振荡器时钟。除此之外，还提供了次级时钟源：LSI(32KHz)内部低速振荡器时钟，用于驱动独立看门狗(IWDG)和 LSE(32.768KHz)外部低速振荡器时钟，用于驱动 RTC 时钟。

范例程序中提供了以下两种程序，使用到的驱动位于 mg32f10x_pwr.c、mg32f10x_anctl.c、mg32f10x_rcc.c、mg32f10x_gpio.c，请注意添加驱动文件。

-  RCC_ClockConfig
-  RCC_ClockConfig2

RCC_ClockConfig 路径下的范例程序是使用外振进行初始化的，而 RCC_ClockConfig2 则是使用内振。一开始，用户可能找不到时钟初始化在哪里，其实时钟初始化的函数并未放在 main.c 中，而是在如下图的位置，当然，用户可以根据习惯，调整时钟初始化的位置，比如放回 main.c 中。




```
static void SetSysClockTo72(void)
{
    __IO uint32_t StartUpCounter = 0, HSEStatus = 0;

    /* Unlocks write to ANCTL registers */
    PWR->ANAKEY1 = 0x03;
    PWR->ANAKEY2 = 0x0C;

    /* APB1CLK = MAINCLK */
    RCC->APB1PRE = RCC_APB1PRE_SRCEN;
    RCC->APB1PRE |= 0x00;

    /* Configure PD0 and PD1 to analog mode */
    RCC->APB1ENR = RCC_APB1ENR_BMX1EN | RCC_APB1ENR_GPIODEN;
    GPIOD->CFGMSK = 0xFFFFC;
    GPIOD->MODER = 0x0F;

    /* Enable HSE */
    ANCTL->HSECR1 = ANCTL_HSECR1_PADOEN;
    ANCTL->HSECR0 = ANCTL_HSECR0_HSEON;

    /* Wait till HSE is ready and if Time out is reached exit */
    do
    {
        HSEStatus = ANCTL->HSESR & ANCTL_HSESR_HSERDY;
        StartUpCounter++;
    } while((HSEStatus == 0) && (StartUpCounter != HSE_STARTUP_TIMEOUT));

    if (HSEStatus != 0)
    {
        /* Configure Flash prefetch, Cache and wait state */
        CACHE->CR = CACHE_CR_CHEEN | CACHE_CR_PREFEN_ON | CACHE_CR_LATENCY_2WS;

        /* AHBCLK = MAINCLK */
        RCC->AHBPRE = 0x00;

        /* APB2CLK = MAINCLK */
        RCC->APB2PRE = RCC_APB2PRE_SRCEN;
        RCC->APB2PRE |= 0x00;

#ifdef HSE_VALUE == 6000000
        /* PLL configuration: PLLCLK = 6MHz * 12 = 72 MHz */
        RCC->PLLSRC = RCC_PLLSRC_HSE;
        RCC->PLLPRE = RCC_PLLPRE_SRCEN;
        RCC->PLLPRE |= 0x00;
        ANCTL->PLLCR = ANCTL_PLLCR_PLLMUL_12;
#elif (HSE_VALUE == 12000000)
        /* PLL configuration: PLLCLK = 12MHz / 2 * 12 = 72 MHz */
        RCC->PLLSRC = RCC_PLLSRC_HSE;
        RCC->PLLPRE = RCC_PLLPRE_SRCEN;
        RCC->PLLPRE |= RCC_PLLPRE_RATIO_2;
        RCC->PLLPRE |= RCC_PLLPRE_DIVEN;
        ANCTL->PLLCR = ANCTL_PLLCR_PLLMUL_12;
```

```
#endif
```

```
    /* Enable PLL */
    ANCTL->PLENR = ANCTL_PLENR_PLLON;

    /* Wait till PLL is ready */
    while(ANCTL->PLLSR != 0x03)
    {
    }

    /* Select PLL as system clock source */
    RCC->MAINCLKSRC = RCC_MAINCLKSRC_PLLCLK;
    RCC->MAINCLKUEN = RCC_MAINCLKUEN_ENA;
}
else
{ /* If HSE fails to start-up, the application will have wrong clock
   configuration. User can add here some code to deal with this error */
    while (1);
}

/* Locks write to ANCTL registers */
PWR->ANAKEY1 = 0x00;
PWR->ANAKEY2 = 0x00;
}
```

以上代码，就是时钟初始化的流程，用户可以复制上方代码，整体替换掉 STM32F10x 的相关时钟初始化功能代码，并根据实际需要的频率，调整分频值，只要记得#include "mg32f10x.h"。

其中，最关键的部分是倍频时，选择 PLL 源的时候。

```
RCC->PLLSRC = RCC_PLLSRC_HSE;
RCC->PLLPRE = RCC_PLLPRE_SRCEN;
RCC->PLLPRE |= RCC_PLLPRE_RATIO_2;
RCC->PLLPRE |= RCC_PLLPRE_DIVEN;
ANCTL->PLLCR = ANCTL_PLLCR_PLLMUL_12;
```

PLLSRC 就是选择 PLL 时钟源，如果您不使用外振，那 SRC 就不使用 HSE 了，可以改成 RCC_PLLSRC_MHSI，这个是内振 8MHz。然后删去上方包含 HSE 内容的代码，再做好对应的倍频倍数和除频倍数，即可输出正确的 PLL 时钟。

```
/* Select PLL as system clock source */
RCC->MAINCLKSRC = RCC_MAINCLKSRC_PLLCLK;
RCC->MAINCLKUEN = RCC_MAINCLKUEN_ENA;
```

最后，在选择系统主时的时候，选择 PLLCLK，就可以将调制好的 PLL 时钟作为时钟主时了，这样，需要内振或者外振，都可以自由做配置，非常简单。

5.17. RNG（随机数发生器）

随机数发生器(RNG)使用 24 位 LFSR 产生 8 位随机数给其他模块使用，同时也可以从 APB2 总线读取这个随机数。有需要的用户可随时使用这个外设，使用到的驱动位于 mg32f10x_rng.c、mg32f10x_rcc.c，请注意添加驱动文件。

RNG 的使用方式非常简单，使能模块后即可直接读取随机数。

```
/* Reset RNG module */
RNG_DeInit();

/* Enable RNG generation */
RNG_Cmd(ENABLE);
printf("Generated random number is %d\r\n", RNG_RandByte());
```

如上代码所示，使用 RNG_RandByte()即可进行读取。

5.18. RTC

RTC 移植起来比较简单，参照范例程序的 RTC 路径下的范例即可，使用到的驱动位于 mg32f10x_pwr.c、mg32f10x_rcc.c、mg32f10x_rtc.c、mg32f10x_bkp.c，请注意添加驱动文件。

void RTC_Configuration(void)就是 RTC 配置函数，可以根据实际移植情况，修改时间参数。

```
/* Reset Backup Domain */
BKP_DeInit();

/* Enable LSE */
BKP_LSEConfig(BKP_LSE_ON);
/* Wait till LSE is ready */
while (BKP_GetLSEReadyFlagStatus() == RESET)
{}

/* Select LSE as RTC Clock Source */
BKP_RTCCLKConfig(BKP_RTCCLKSource_LSE);

/* Enable RTC Clock */
BKP_RTCCLKCmd(ENABLE);

/* Wait for RTC registers synchronization */
RTC_WaitForSynchro();

/* Wait until last write operation on RTC registers has finished */
RTC_WaitForLastTask();

/* Enable the RTC Second */
RTC_ITConfig(RTC_IT_SEC, ENABLE);

/* Wait until last write operation on RTC registers has finished */
RTC_WaitForLastTask();

/* Set RTC prescaler: set RTC period to 1sec */
RTC_SetPrescaler(32767); /* RTC period = RTCCLK/RTC_PR = (32.768 KHz)/(32767+1) */
```

```
/* Wait until last write operation on RTC registers has finished */
RTC_WaitForLastTask();

/* Sets the RTC counter */
RTC_SetCounter(40271);
```

以上，根据相应时钟设置好分频值和定时器值就可以正常工作了。用户可以复制上方代码，整体替换掉 STM32F10x 的相关 RTC 初始化功能代码，并根据实际需要的频率，调整分频值和计数器值。

```
void Time_Display(uint32_t TimeVar)
{
    uint32_t THH = 0, TMM = 0, TSS = 0;

    /* Reset RTC Counter when Time is 23:59:59 */
    if (RTC_GetCounter() >= 0x0001517F)
    {
        RTC_SetCounter(0x0);
        /* Wait until last write operation on RTC registers has finished */
        RTC_WaitForLastTask();
    }

    /* Compute  hours */
    THH = TimeVar / 3600;
    /* Compute minutes */
    TMM = (TimeVar % 3600) / 60;
    /* Compute seconds */
    TSS = (TimeVar % 3600) % 60;

    printf("Time: %0.2d:%0.2d:%0.2d\r", THH, TMM, TSS);
}
```

范例程序还提供了转换为时分秒单位的时间，方便用户进行移植。


中断函数结构模板：


```
void RTC_IRQHandler(void)
{
    if (RTC_GetITStatus(RTC_IT_SEC) != RESET)
    {
        /* Clear the RTC Second interrupt */
        RTC_ClearITPendingBit(RTC_IT_SEC);

        /* Enable time update */
        TimeDisplay = 1;
    }
}
```

5.19. SFM（特殊功能宏）

SFM 这个外设非常实用，可以用硬件计算的方式计算 32 位二进制数中 1 的个数，不需要用户用软件来一个个算，节约时间和代码空间，另外，根据需要，也可以对 32 位数进行倍宽操作。参考范例程序下 SFM 路径下的两个范例程序，分别是计算 1 的个数和数据倍宽，使用到的驱动位于 mg32f10x_sfm.c、mg32f10x_rcc.c，请注意添加驱动文件。

 SFM_ComputeBit1

 SFM_ExpandBits


```
CRC_SFM_DeInit();
```


```
printf("The number of bit 1 in 0xAAAAAAAA is %d\r\n", SFM_ComputeBit1Number(0xAAAAAAAA));
printf("The number of bit 1 in 0x55555555 is %d\r\n", SFM_ComputeBit1Number(0x55555555));
printf("The number of bit 1 in 0xFFFFFFFF is %d\r\n", SFM_ComputeBit1Number(0xFFFFFFFF));
printf("The number of bit 1 in 0x7FFFFFFF is %d\r\n", SFM_ComputeBit1Number(0x7FFFFFFF));
printf("The number of bit 1 in 0x00000000 is %d\r\n", SFM_ComputeBit1Number(0x00000000));
printf("The number of bit 1 in 0x1BC4D029 is %d\r\n", SFM_ComputeBit1Number(0x1BC4D029));
printf("The number of bit 1 in 0xFFFF0000 is %d\r\n", SFM_ComputeBit1Number(0xFFFF0000));
printf("The number of bit 1 in 0x0000F0FF is %d\r\n", SFM_ComputeBit1Number(0x0000F0FF));
printf("The number of bit 1 in 0x5503AAFF is %d\r\n", SFM_ComputeBit1Number(0x5503AAFF));
```


使用起来也很简单，初始化 SFM 模块后，就直接使用库函数进行计算即可。


5.20. SPI


SPI 外设提供的范例程序非常多，用户可根据需要自行进行选择移植。就在范例程序 SPI 路径下，使用到的驱动位于 mg32f10x_gpio.c、mg32f10x_rcc.c、mg32f10x_spi.c，请注意添加驱动文件。


 QSPI_Master_DMA


 QSPI_Master_Interrupt


 QSPI_QuadSPI_FLASH


 QSPI_SPI_FLASH


 SPIM2_Master_DMA


 SPIM2_Master_Interrupt

 SPIM2_SPI_FLASH

 SPIS1_Slave_DMA

 SPIS1_Slave_Interrupt

 SPIS2_Slave_DMA

 SPIS2_Slave_Interrupt

```
/* SPI configuration */
SPI_DeInit(SPI2);
SPI_InitStructure.SPI_TransferMode = SPI_TransferMode_TxAndRx;
SPI_InitStructure.SPI_DataSize = SPI_DataSize_8b;
SPI_InitStructure.SPI_CPOL = SPI_CPOL_Low;
SPI_InitStructure.SPI_CPHA = SPI_CPHA_1Edge;
SPI_InitStructure.SPI_BaudRatePrescaler = 8;
SPI_InitStructure.SPI_FrameFormat = SPI_FrameFormat_SPI;
SPI_Init(SPI2, &SPI_InitStructure);
SPI_ITConfig(SPI2, 0xFF, DISABLE);
```

```
SPI_NSSConfig(SPI2, SPI_NSS_0, ENABLE);
```

SPI 的初始化如上代码，用户可以根据预分频值调整 SPI 速率，再调整 SPI 数据格式就可以完成初始化，并替换掉 STM32F10x 的相关 SPI 初始化功能代码。

```
while (SPI_GetFlagStatus(QSPI, SPI_FLAG_TFE) == RESET);
SPI_WriteData(QSPI, x); //通过 SPI 写 x 变量
```

```
while (SPI_GetFlagStatus(QSPI, SPI_FLAG_RFNE) == RESET);
x = SPI_ReadData(QSPI); //读取 SPI 数据并赋值给 x
```

中断函数结构模板：

```
void QSPI_IRQHandler(void)
{
    if(SPI_GetITStatus(QSPI, SPI_IT_RXF) != RESET)
    {
        while(SPI_GetFlagStatus(QSPI, SPI_FLAG_RFNE) != RESET)
        {
            master_rx_buf[rx_index] = SPI_ReadData(QSPI);
            rx_index++;
            if(rx_index >= 20) {
                SPI_ITConfig(QSPI, SPI_IT_RXF, DISABLE);
                break;
            }
        }
    }

    if(SPI_GetITStatus(QSPI, SPI_IT_TXE) != RESET)
    {
        while(SPI_GetFlagStatus(QSPI, SPI_FLAG_TFNF) != RESET)
        {
            SPI_WriteData(QSPI, master_tx_data[tx_index]);
            tx_index++;
            if(tx_index >= 20) {
                SPI_ITConfig(QSPI, SPI_IT_TXE, DISABLE);
                break;
            }
        }
    }
}
```

5.21. SYSTICK

Systick 一般用于毫秒级延时，配置也很容易。参考范例程序 SysTick 路径下，使用到的驱动位于 mg32f10x_rcc.c，请注意添加驱动文件。

```
SystemCoreClockUpdate();
if (SysTick_Config(SystemCoreClock / 1000))
{
    /* Capture error */
    while (1);
}
```

以上代码即可完成 Systick 初始化，systemCoreClock 要根据实际频率修改值，用户可整体替换掉 STM32F10x

的相关 SysTick 初始化功能代码。

```
void Delay(__IO uint32_t nTime)
{
    TimingDelay = nTime;

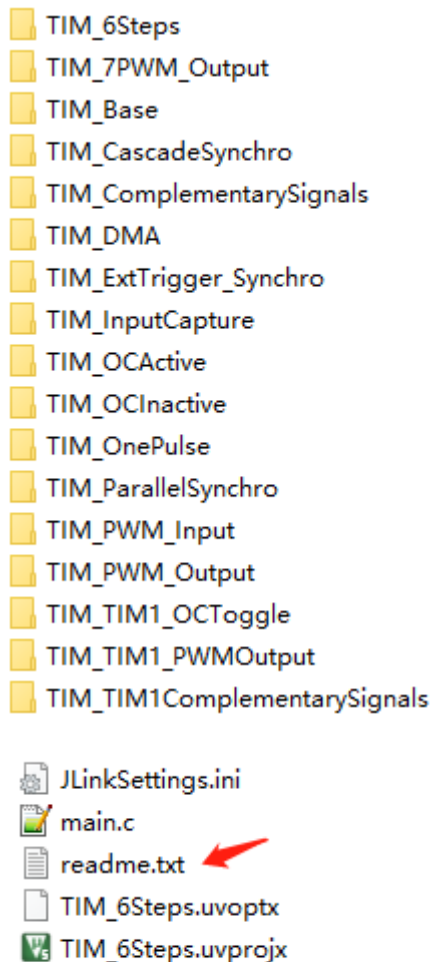
    while(TimingDelay != 0);
}

/**
 * @brief This function handles SysTick Handler.
 * @param None
 * @return None
 */
void SysTick_Handler(void)
{
    if (TimingDelay != 0x00)
    {
        TimingDelay--;
    }
}
```

最后加上上方的中断函数就可以调用上面的 Delay 函数做延时，这个延时函数也需要替换掉 STM32F10x 的 Delay 函数。

5.22. TIM

定时器是非常常用的外设，功能也非常多，为了方便用户移植，我们也提供了大量关于定时器的范例程序，用户可根据范例程序 TIM 路径下的各个不同程序的 `readme.txt` 中，了解各个程序的实现功能，并根据实际使用频率和模式进行配置，再替换掉 STM32F10x 的相关 TIM 初始化功能代码，使用到的驱动位于 `mg32f10x_tim.c`、`mg32f10x_rcc.c`，请注意添加驱动文件。



[注意]: 若用户需要使用带死区控制的 PWM，请使用 TIM1；若不需要使用，TIM2、TIM3、TIM4 都可以输出 PWM。

TIM1ComplementarySignals 范例即 PWM 互补输出，用户可以参考该范例用于类似 BLDC 等应用。

```
/* Time Base configuration */
TIM_TimeBaseStructure.TIM_Prescaler = 0;
TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
TIM_TimeBaseStructure.TIM_Period = TimerPeriod;
TIM_TimeBaseStructure.TIM_ClockDivision = 0;
TIM_TimeBaseStructure.TIM_RepetitionCounter = 0;

TIM_TimeBaseInit(TIM1, &TIM_TimeBaseStructure);

/* Channel 1, 2 and 3 Configuration in PWM mode */
TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM2;
TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
TIM_OCInitStructure.TIM_OutputNState = TIM_OutputNState_Enable;
TIM_OCInitStructure.TIM_Pulse = Channel1Pulse;
TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_Low;
```



```
TIM_OCInitStructure.TIM_OCNPolarity = TIM_OCNPolarity_Low;
TIM_OCInitStructure.TIM_OCIdleState = TIM_OCIdleState_Set;
TIM_OCInitStructure.TIM_OCNIdleState = TIM_OCIdleState_Reset;

TIM_OC1Init(TIM1, &TIM_OCInitStructure);

TIM_OCInitStructure.TIM_Pulse = Channel2Pulse;
TIM_OC2Init(TIM1, &TIM_OCInitStructure);

TIM_OCInitStructure.TIM_Pulse = Channel3Pulse;
TIM_OC3Init(TIM1, &TIM_OCInitStructure);

/* Automatic Output enable, Break, dead time and lock configuration*/
TIM_BDTRInitStructure.TIM_OSSRState = TIM_OSSRState_Enable;
TIM_BDTRInitStructure.TIM_OSSIState = TIM_OSSIState_Enable;
TIM_BDTRInitStructure.TIM_LOCKLevel = TIM_LOCKLevel_1;
TIM_BDTRInitStructure.TIM_DeadTime = 11;
TIM_BDTRInitStructure.TIM_Break = TIM_Break_Enable;
TIM_BDTRInitStructure.TIM_BreakPolarity = TIM_BreakPolarity_High;
TIM_BDTRInitStructure.TIM_AutomaticOutput = TIM_AutomaticOutput_Enable;

TIM_BDTRConfig(TIM1, &TIM_BDTRInitStructure);

/* TIM1 counter enable */
TIM_Cmd(TIM1, ENABLE);

/* Main Output Enable */
TIM_CtrlPWMOutputs(TIM1, ENABLE);
```

若是要简单的定时功能，则可以参考 TIM_Base 范例，使用到的驱动位于 mg32f10x_tim.c、mg32f10x_rcc.c，请注意添加驱动文件。主要区别就是 TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_Timing; PWM 是 PWMMode。

```
/* -----
TIM2 Configuration: Output Compare Timing Mode:
TIM2 counter clock at 12 MHz
CC1 update rate = TIM2 counter clock / CCR1_Val = 286.72 Hz
CC2 update rate = TIM2 counter clock / CCR2_Val = 523.97 Hz
CC3 update rate = TIM2 counter clock / CCR3_Val = 811.08 Hz
CC4 update rate = TIM2 counter clock / CCR4_Val = 1389.85 Hz
----- */

/* Compute the prescaler value */
PrescalerValue = (uint16_t) (SystemCoreClock / 12000000) - 1;

/* Time base configuration */
TIM_TimeBaseStructure.TIM_Period = 0xFFFFF;
TIM_TimeBaseStructure.TIM_Prescaler = 0;
TIM_TimeBaseStructure.TIM_ClockDivision = 0;
TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;

TIM_TimeBaseInit(TIM2, &TIM_TimeBaseStructure);
```

```
/* Prescaler configuration */
TIM_PrescalerConfig(TIM2, PrescalerValue, TIM_PSCReloadMode_Immediate);

/* Output Compare Timing Mode configuration: Channel1 */
TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_Timing;
TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
TIM_OCInitStructure.TIM_Pulse = CCR1_Val;
TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High;

TIM_OC1Init(TIM2, &TIM_OCInitStructure);

TIM_OC1PreloadConfig(TIM2, TIM_OCPreload_Disable);

/* Output Compare Timing Mode configuration: Channel2 */
TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
TIM_OCInitStructure.TIM_Pulse = CCR2_Val;

TIM_OC2Init(TIM2, &TIM_OCInitStructure);

TIM_OC2PreloadConfig(TIM2, TIM_OCPreload_Disable);

/* Output Compare Timing Mode configuration: Channel3 */
TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
TIM_OCInitStructure.TIM_Pulse = CCR3_Val;

TIM_OC3Init(TIM2, &TIM_OCInitStructure);

TIM_OC3PreloadConfig(TIM2, TIM_OCPreload_Disable);

/* Output Compare Timing Mode configuration: Channel4 */
TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
TIM_OCInitStructure.TIM_Pulse = CCR4_Val;

TIM_OC4Init(TIM2, &TIM_OCInitStructure);

TIM_OC4PreloadConfig(TIM2, TIM_OCPreload_Disable);

/* TIM IT enable */
TIM_ITConfig(TIM2, TIM_IT_CC1 | TIM_IT_CC2 | TIM_IT_CC3 | TIM_IT_CC4, ENABLE);

/* TIM2 enable counter */
TIM_Cmd(TIM2, ENABLE);
```

5.23. UART

串口也是很常见的功能，因此我们也有提供很多范例程序，用户可根据范例程序 UART 路径下的各个不同程序的 readme.txt 中，了解各个程序的实现功能。一般情况参考 UART_Printf 这个范例就够了，使用到的驱动位于 mg32f10x_uart.c、mg32f10x_rcc.c、mg32f10x_gpio.c，请注意添加驱动文件。

```
/* UART1 configuration */
UART_DeInit(UART1);
UART_InitStructure.UART_BaudRate = 115200;
UART_InitStructure.UART_WordLength = UART_WordLength_8b;
UART_InitStructure.UART_StopBits = UART_StopBits_One;
UART_InitStructure.UART_Parity = UART_Parity_None;
UART_InitStructure.UART_AutoFlowControl = UART_AutoFlowControl_None;
UART_Init(UART1, &UART_InitStructure);
UART_FIFOCmd(UART1, ENABLE);
```

根据以上代码，调整需要的波特率、数据格式即可完成初始化，实际使用的引脚可以根据 GPIO 的初始化做调整。

初始化完成后，就可以使用 printf 来输出串口数据了，但是注意，使用 printf 的前提是先 #include <stdio.h>。

```
scanf("%d", &value); //读取 uart 数据
printf("You enter is %d\r\n", value); //写 uart 数据
```

或者

```
while(!(UART_GetLineStatus(UART1) & UART_LINE_STATUS_DR));
value = UART_ReadData(UART1); //读取 uart 数据
UART_WriteData(UART1, value); //写 uart 数据
while(!(UART_GetLineStatus(UART1) & UART_LINE_STATUS_THRE));
```

中断函数结构模板：

```
void UART1_IRQHandler(void)
{
    uint8_t rbyte;
    uint8_t int_id;
    int_id = UART_GetIntID(UART1);
    if(int_id == UART_INTID_RDA)
    {
        rbyte = UART_ReadData(UART1);

        rxBuffer[rxIndex++] = rbyte;
        if (rxIndex >= 100) {
            flag = 1;
            rxIndex = 0;
        }
    }
    else if (int_id == UART_INTID_THRE)
    {
        if (txIndex < sizeof(txBuffer)) {
            UART_WriteData(UART1, txBuffer[txIndex]);
            txIndex++;
        }
        else {
            UART_ITConfig(UART1, UART_IT_THRE, DISABLE);
        }
    }
}
```

5.24. USB

USB 外设我们提供了 4 种不同应用的范例程序，用户可根据不同应用做移植。具体的实现，也可以根据范例程序 USB 路径下的各个不同程序的 readme.txt 中了解，使用到的驱动位于 mg32f10x_anctl.c、mg32f10x_rcc.c、mg32f10x_pwr.c、mg32f10x_gpio.c、usbd_user.c，请注意添加驱动文件。

- USB_CDC_Echo
- USB_HID_Mouse
- USB_Mass_Storage_SPI_FLASH
- USB_Mass_Storage_SRAM

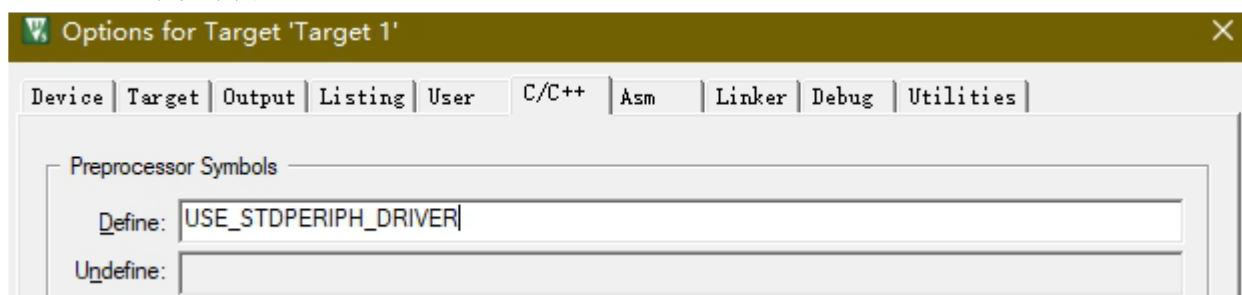
[注意]: 若用户使用 **MHSI 内振** 给 USB 做时钟源，需要另外添加代码，具体代码已存放在开发包使用文档路径下的 USB 免外振 Code 中。

以下是 USB 免外振的配置流程，需要占用 SysTick 或者 TIM4，按需选择一个外设的文件进行以下配置即可。

一、使用环境

[注意]: 此配置仅在使用 USB 外设时才有效。

配置 KEIL 工程如下图。



二、占用资源

滴答定时器(SysTick)或者通用定时器(TIM4)

三、使用教程

1. 添加 MHSI Trim 程序。



jmntTrim.c



jmntTrim.h

在工程中添加 jmntTrim.c 和 jmntTrim.h 文件

2. 配置 USB

a. 使能 USB 的 SOF 中断

```

/**
 * @brief Connects the device to the USB host.
 * @return None
 */
void USBD_User_Connect(void)
{
    /* Enable BMX1, GPIOA clock */
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_BMX1 | RCC_APB1Periph_GPIOA, ENABLE);
    /* Configure the drive current of PA11 and PA12 */
    GPIO_DriveCurrentConfig(GPIOA, GPIO_Pin_11 | GPIO_Pin_12, 0x03);
    /* Configure PA11 and PA12 as Alternate function mode */
    GPIO_Init(GPIOA, GPIO_Pin_11 | GPIO_Pin_12, GPIO_MODE_AF | GPIO_AF3);

    USB->INTRUSBE = USB_INTRUSBE_RSTIE | USB_INTRUSBE_RSUIE | USB_INTRUSBE_SOFIE;
}

```

b. 在 usbd_user.c 里包含 jmntTrim.h 文件

```
#include "jmntTrim.h"
```

c. 在 USBD_User_SOF 函数里调用 CheckTune 函数

```

void USBD_User_SOF(void)
{
    CheckTune();
}

```

d. USB 中断抢占和子优先级都必须配置为最高。

e. 其余配置根据用户使用场景自行配置。

3. 在 main 函数里调用 jmntTrimInit 函数

```

int main(void)
{
    MAINCLKConfig_MHSI_48MHz();
    jmntTrimInit();
}

```

注意需要包含 jmntTrim.h 文件

4. 根据不同系统主频配置 Trim 参数

默认主频为 48Mhz，用户主频为 48Mhz 时参数不用修改。

若主频不是 48Mhz，用户可根据当前值线性修改。

```

/* The range of sof frame interval TIMER counts. */
#define INR_HEAD 44500
#define INR_TAIL 51500

```

5.25. WWDG（窗口看门狗）

窗口看门狗通常被用来监测，由外部干扰或不可预见的逻辑条件造成的应用程序背离正常的运行序列而产生的软件故障。除非递减计数器的值在 T6 位变成 0 前被刷新，看门狗电路在达到预置的时间周期时，会产生一个 MCU 复位。在递减计数器达到窗口寄存器数值之前，如果 7 位的递减计数器数值(在控制寄存器中) 被刷新，那么也将产生一个 MCU 复位。这表明递减计数器需要在一个有限的时间窗口中被刷新。

我们也提供了 WWDG 的范例程序供用户进行移植,用户可根据实际使用频率进行配置,再替换掉 STM32F10x 的相关 WWDG 初始化功能代码,使用到的驱动位于 mg32f10x_wwdg.c、mg32f10x_rcc.c, 请注意添加驱动文件。

```
/* WWDG clock counter = (PCLK(96MHz)/4096)/8 = 2929.6875 Hz (~0.341 ms) */
WWDG_SetPrescaler(WWDG_Prescaler_8);

/*
Enable WWDG and set counter value to 127, WWDG timeout = ~0.341 ms * 64 = 21.8 ms
In this case the refresh window is: ~0.341 ms * (127-80) = 16.027 ms < refresh window < ~0.341 ms *
64 = 21.8ms
*/
WWDG_SetWindowValue(80);
WWDG_Enable(127);

WWDG_SetCounter(127);
```

窗口看门狗被设定为在窗口值~计数值为 64 之间可进行喂狗，其余时间喂狗或者计数值小于 64，就会发生复位。

喂狗很简单，就是在正确的时间范围内运行 WWDG_SetCounter(x);即可。

中断函数结构模板：

```
void WWDG_IRQHandler(void)
{
    WWDG_ClearFlag();
}
```

6. 版本历史

版本 1.01 (2022_0303)		章节
1	将硬件资源差异表中的“LCD Driver”修正为“LED Driver”	2.2
2	ADC 软件移植增加 ADC 读取函数的使用和中断函数模板	5.2
3	CMP 软件移植增加 CMP 比较值读取函数的使用，并新增注意事项	5.3.1
4	DCSS 软件移植增加 DCSS 中断函数模板	5.3.2
5	DMAC 软件移植增加 DMAC 中断函数模板	5.6
6	EXTI 软件移植增加 EXTI 中断函数模板	5.7
7	I2C 软件移植增加 I2C 中断函数模板	5.10
8	I2S 软件移植增加 I2S 中断函数模板	5.11
9	IWDG 软件移植增加注意事项	5.12
10	PVD 软件移植增加 PVD 中断函数模板	5.15
11	RTC 软件移植增加 RTC 中断函数模板	5.18
12	SPI 软件移植增加读写数据函数的使用和中断函数模板	5.20
13	UART 软件移植增加读写数据函数的使用和中断函数模板	5.23
14	WWDG 软件移植增加看门狗中断函数模板	5.25
版本 1.0 (2022_0214)		章节
1	初始版本	