SA5Z-50 嵌入式 ARM 参考例程用户指南

(UG05207, Version 3.1)

(20251204)

西安智多晶

西安智多晶微电子有限公司 Xi'an Intelligence Silicon Technology Co.,Ltd.
西安市高新区科技二路 72 号零壹广场裙楼 DEF101 室 邮编 710075

目录

1 前言 1
1.1 概述 1
1.2 相关资料 1
2 系统架构 2
2.1 概述 2
2.2 设计流程 2
2.2.1 硬件工程建立 2
2.2.2 软件工程建立 3
2.2.3 系统时钟设置 8
2.2.4 地址空间分配 8
2.2.5 程序下载步骤 9
2.2.6 程序在线调试 10
3 参考例程 13
3.1 GPIO 输出例程+SysTick 13
3.1.1 LED 电路 13
3.1.2 引脚分配 13
3.1.3 代码设计 14
3.1.4 下载验证 15
3.2 GPIO 输入例程 15
3.2.1 按键电路 15
3.2.2 引脚分配 15
3.2.3 代码设计 16
3.2.4 下载验证 17
3.3 外部中断例程 17
3.3.1 按键电路 17
3.3.2 引脚分配 17
3.3.3 代码设计 18
3.3.4 下载验证 19
3.4 串口例程 19
3.4.1 UART 简介 19
3.4.2 串口接口电路 19
3.4.3 引脚分配 20
3.4.4 代码设计 20
3.4.5 下载验证 22
3.5 I2C 例程 22

西安智多晶

3.5.1 I2C 简介 22
3.5.2 EEPROM 电路 23
3.5.3 引脚分配 23
3.5.4 代码设计 24
3.5.5 下载验证 27
3.6 ADC 例程 27
3.6.1 ADC 介绍 27
3.6.2 ADC 电路 28
3.6.3 引脚分配 28
3.6.4 代码设计 28
3.6.5 下载验证 31
3.7 定时器例程 31
3.7.1 定时器简介 31
3.7.2 引脚分配 31
3.7.3 代码设计 32
3.7.4 下载验证 33
3.8 DUAL TIMER 例程 33
3.8.1 DUAL Timer 简介 33
3.8.2 引脚分配 34
3.8.3 代码设计 34
3.8.4 下载验证 35
3.9 WATCHDOG 例程 36
3.9.1 WatchDog 介绍 36
3.9.2 引脚分配 36
3.9.3 代码设计 36
3.9.4 下载验证 37
3.10 SPI 例程 37
3.10.1 SPI 简介 38
3.10.2 FLASH 电路图 38
3.10.3 引脚分配 38
3.10.4 代码设计 39
3.10.5 下载验证 41
3.11 AHB MASTER 例程 42
3.11.1 AHB 简介 42
3.11.2 引脚分配 45
3.11.3 代码设计 46
3.11.4 下载验证 48

3.12 DMA 例程 48
3.12.1 DMA 简介 48
3.12.2 引脚分配 48
3.12.3 代码设计 49
3.12.4 下载验证 52
3.13 DMA-AHB 突发例程 52
3.13.1 DMA-AHB 简介 52
3.13.2 引脚分配 53
3.13.3 代码设计 53
3.13.4 下载验证 55
3.14 提高 STAR 系统时钟频率例程 56
3.14.1 串口接口电路 56
3.14.2 引脚分配 56
3.14.3 代码设计 57
3.14.4 下载验证 60
3.15 EXTINT 例程（来自 FPGA 侧的外部中断例程） 60
3.15.1 引脚分配 61
3.15.2 代码设计 61
3.15.3 下载验证 62
3.16 APB MASTER 例程 63
3.16.1 APB 简介 63
3.16.2 引脚分配 63
3.16.3 代码设计 64
3.16.4 下载验证 65
3.17 AXI-LITE MASTER 例程 65
3.17.1 AXI-lite 简介 65
3.17.2 引脚分配 66
3.17.3 代码设计 66
3.17.4 下载验证 67
3.18 AHB MASTER 接口扩展例程 67
3.18.1 AHB 扩展简介 67
3.18.2 引脚分配 67
3.18.3 代码设计 68
3.18.4 下载验证 69
3.19 APB MASTER 接口扩展例程 69
3.19.1 APB 扩展简介 69
3.19.2 引脚分配 70
3.19.3 代码设计 70

3.19.4 下载验证 71
3.20 DDR 作为扩展 RAM 例程 71
3.20.1 原理框图 71
3.20.2 文件链接到外部 RAM 72
3.20.3 下载验证 75
3.21 DDR 作为扩展 ROM 例程（IAP） 75
3.21.1 原理框图 76
3.21.2 bootloader 代码 76
3.21.3 App 代码 79
3.21.4 下载验证 80
3.22 CAN 例程 81
3.22.1 CAN 简介 81
3.22.2 系统框图 81
3.22.3 寄存器访问地址 81
3.22.4 API 函数 82
3.22.5 代码设计 83
3.22.6 下载验证 85
3.23 关断时钟降功耗例程 85
3.23.1 系统框图 86
3.23.2 代码设计 86
3.23.3 下载验证 87
3.23.4 功耗测试 87
3.24 FREERTOS 例程 87
3.24.1 FreeRTOS 简介 87
3.24.2 代码设计 88
3.24.3 下载验证 93

1 前言

1.1 概述

本手册针对的器件为 SA5Z-50，FPGA 内部集成了嵌入式 ARM STAR 硬核处理器，例程开发使用的板卡为智多晶 IST071_SA5Z_50_D0_7F484C_EVB_V3_1 开发板。文中首先介绍了 STAR 硬核开发的软硬件环境，之后结合开发板资源提供了各模块例程说明。本文档作为 STAR 参考例程的说明文档，旨在帮助用户快速熟悉并使用 ARM STAR 硬核处理器。

1.2 相关资料

- UG05205_SA5Z-50 ARM STAR User Guide
- UG05206_SA5Z-50 嵌入式 ARM 库函数用户指南
- UG00004_STAR IP 用户指南_V3.0

西安智多晶

2 系统架构

2.1 概述

SA5Z-50 FPGA 的 ARM STAR 核是通过 SOC 方式集成在芯片内部，STAR 核的 GPIO、UART、 Timer、I2C、ADC、SPI 等外设连接在内部 APB 总线，STAR 核与 FPGA 之间通过 AHB 总线进行通信。STAR 核内置 JTAG 及 SWD 调试电路，并且具有专用指令 SRAM 及数据 SRAM。STAR 内核系统架构如图 2-1 所示。

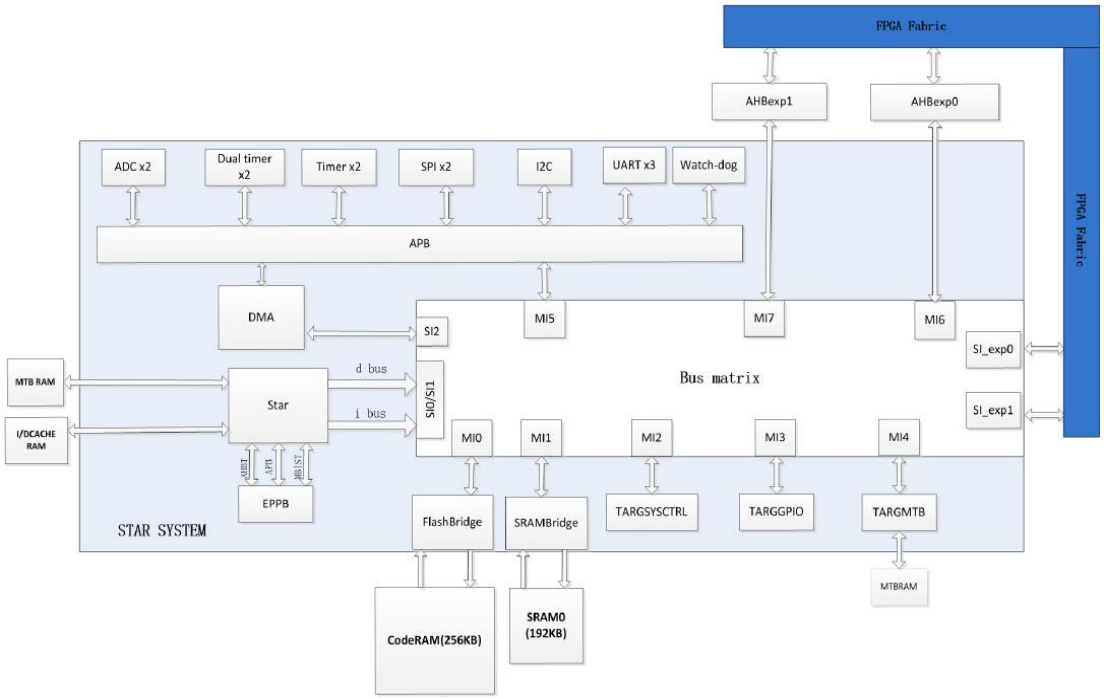


图 2-1 STAR 内核系统架构

2.2 设计流程

STAR 软件开发需要软硬件进行协同开发（软件指 STAR 侧、硬件指 FPGA 逻辑侧），硬件开发环境为智多晶 HQFPGA 软件，软件开发环境采用第三方软件，如 Keil 等支持 ARM 嵌入式软件开发的开发工具。

2.2.1 硬件工程建立

在 HQFPGA 编译环境下建立硬件工程，工程中通过 IP 生成 STAR 硬核。生成的 IP 接口数量随配置的 IP 功能而变化，比如未配置 GPIO 时，生成的 IP 便没有相应接口。

如图 2-2 为 IP 配置界面，界面左侧为功能框图，用户可通过点击功能框跳转到相应的配置页；界面右侧为配置页，顶部的配置页标签指示当前页面的内容，同时可通过鼠标点击标签跳转到对应配置页；配置页包含配置项和配置说明；统计报告页展示当前的配置详情表。

用户可通过以下按钮操作 IP 配置界面：点击“重置配置”可将所有配置项恢复至默认状态；点击“参考手册”可打开 IP 用户文档；勾选“输出范例”可在生成 IP 时同时输出

DEMO 工程；点击“确认”执行 IP 生成；点击“取消”则退出 IP 配置界面。

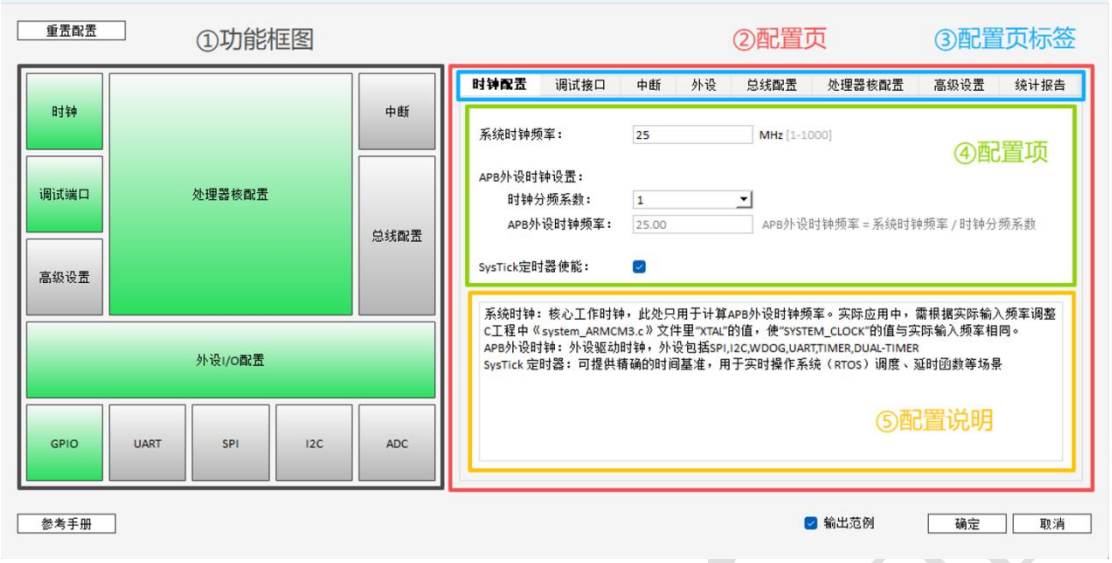


图 2-2 STAR IP 配置界面

2.2.2 软件工程建立

STAR 软件工程在第三方编译器 Keil uVision5 中建立及编译。Keil uVision5 版本推荐使用 V5.37.0.0。

1.打开 Keil 软件，点击 Project，选择下面的 New uVision Project，选择工程目录并确认好工程名称。器件类型按【ARM】->【ARM Cortex M33】->【ARMCM33_DSP_FP_TZ】进行选择，如下图所示。

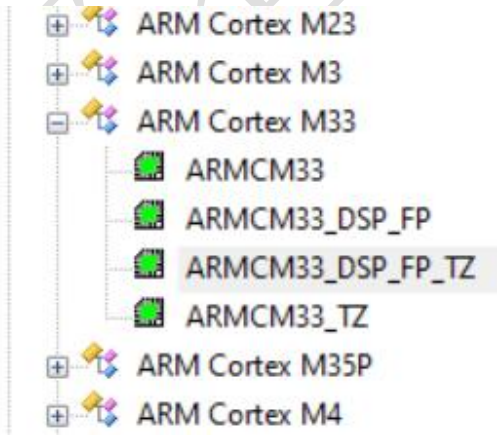


图 2-3 器件类型选择

2.点击 管理和添加工程的源文件：

a、在弹出的【Manage Project Items】窗口选择【Project Items】栏目，点击【Groups】的 图标，新建 3 个文件分组（startup，lib，user），分别用来存放启动文件、库函数文件和用户编写的文件，如下图所示；

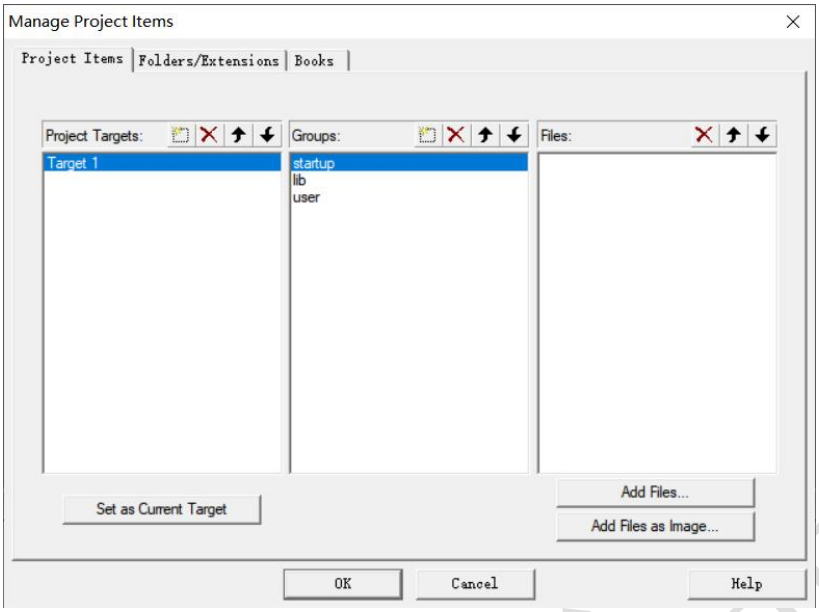


图 2-4 添加源文件

b、选择新建的分组，再点击【Add Files】选择要添加的。c 文件或。s 文件，在各分组的源文件添加完成后，点击【OK】完成工程源文件的添加。

3、点击设置工程属性：

a、【Target】选项设置，主要是设置 ROM 和 RAM 的起始地址和大小，如下图所示；

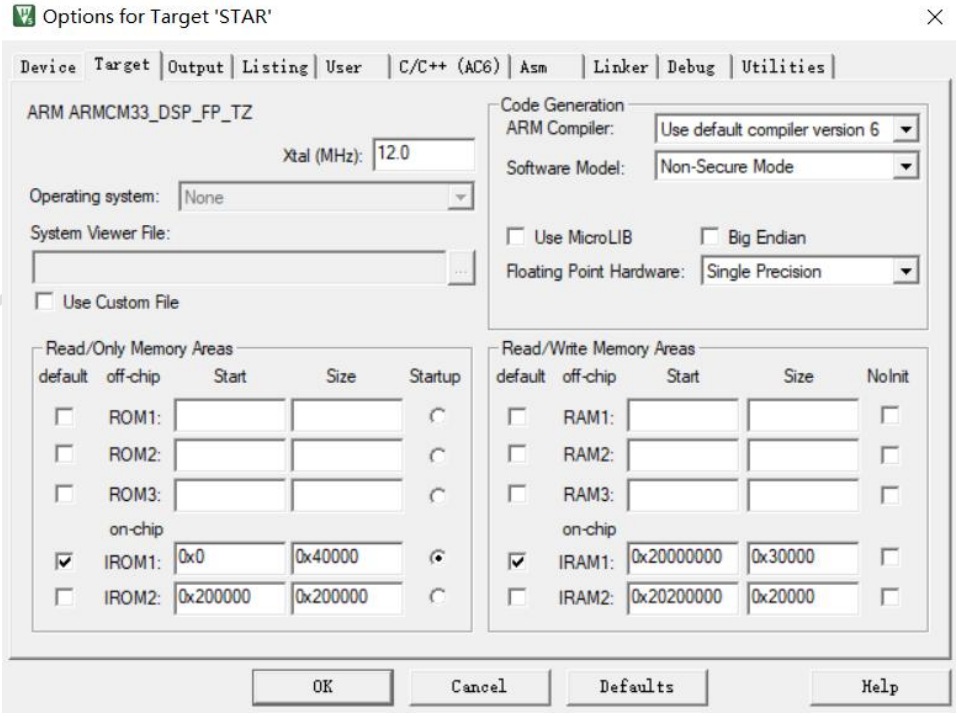


图 2-5 ROM 和 RAM 空间设置

b、【Output】和【Listing】选项使用默认配置；

c、设置【User】选项，“Run #1”主要是设置编译后自动将固件转为。bin 格式，便于后续与 FPGA 的 bin 文件合并为一个 bin 文件后下载到 Flash；“Run #2”是执行脚本文

件，将嵌入式输出 bin 文件与 FPGA 的 bin 文件合并，之后将合并的 bin 文件下载到 Flash，开发者平时可取消“Run #2”的勾选，只生成嵌入式的 bin 文件（mergeBinFileAndProgram.bat 脚本文件位于工程文件的 MDK 文件夹下，包含文件路径与工具路径，需根据实际路径配置才能正确执行脚本指令），如下图所示；

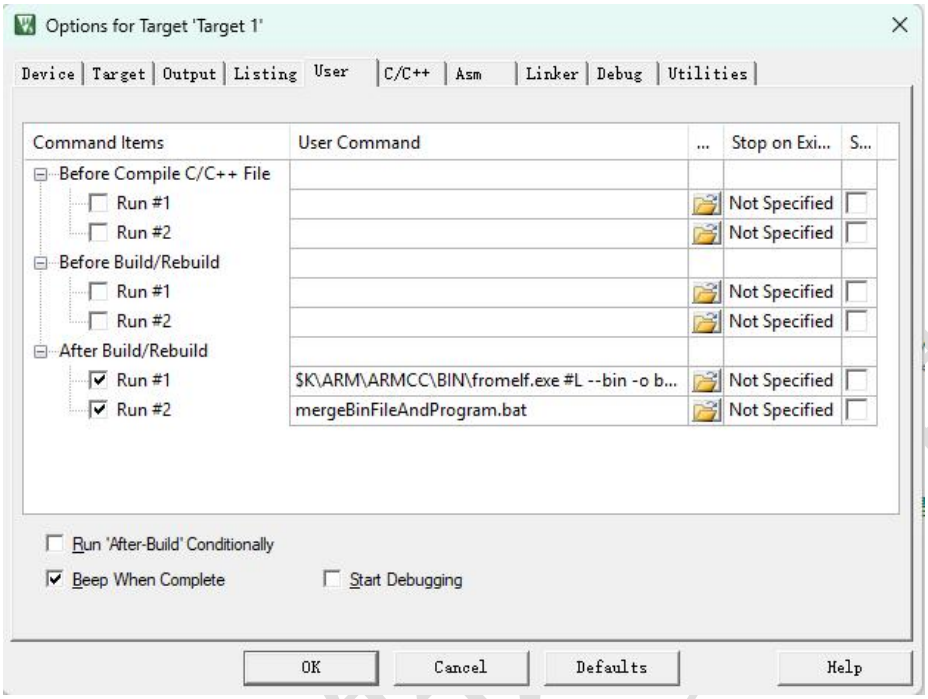


图 2-6 User 选项设置

d、【C/C++】选项设置，主要是添加预定义和工程源码中包含的。h 文件的路径；

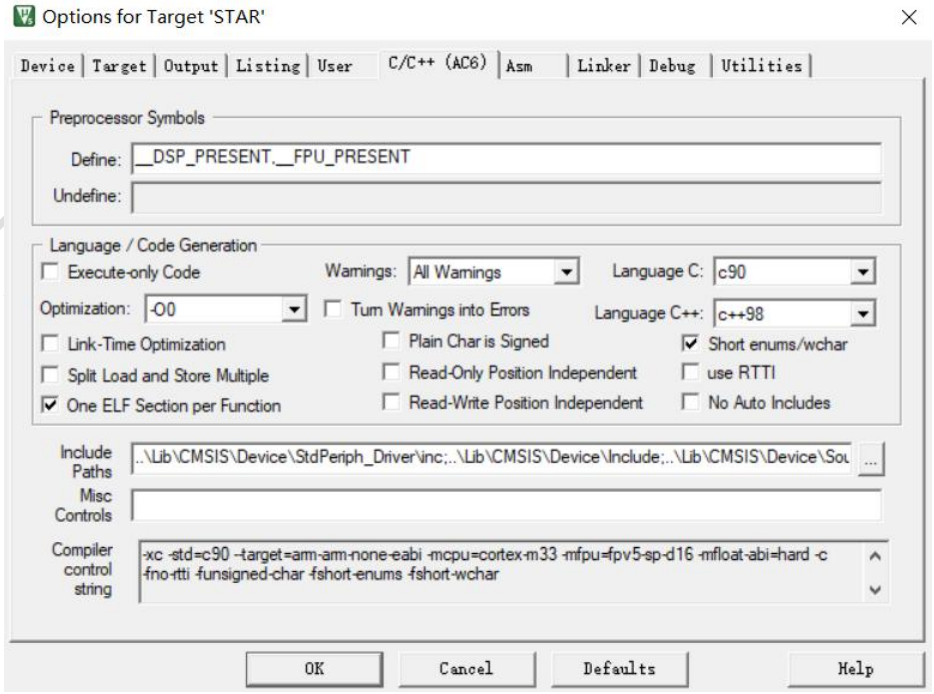


图 2-7 C/C++选项设置

e、【Asm】 和【Linker】 选项使用默认配置；

f、【Debug】 选项设置，需要选择 Debug 工具，我们这边使用的是 JLink，调试口是

SW 模式，dll 的版本需保证 6.86 及以上；

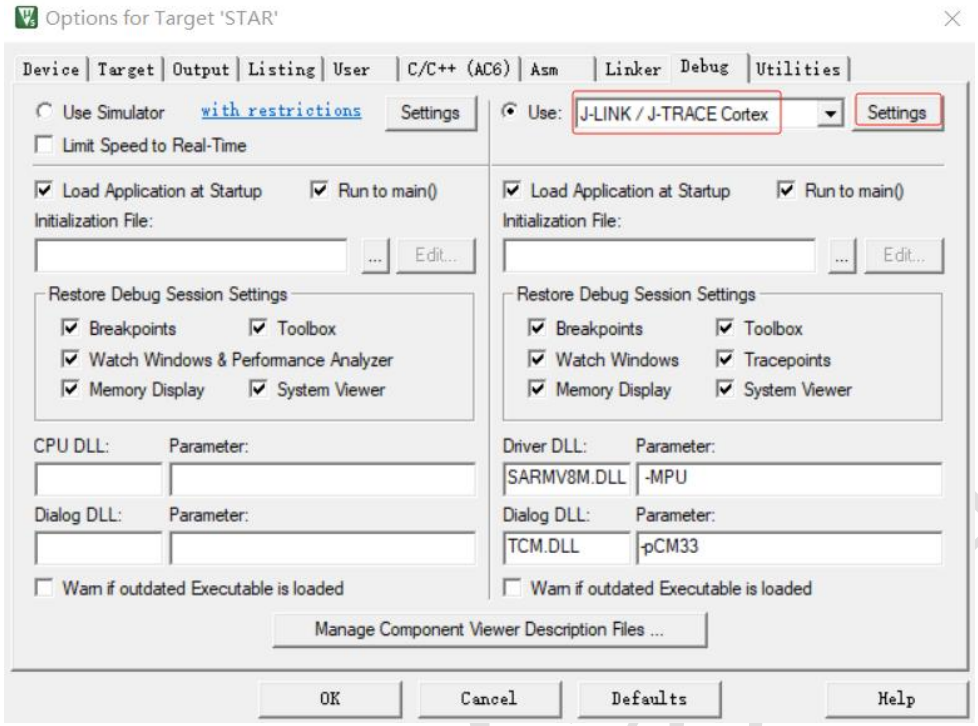


图 2-8 Debug 选项设置 1

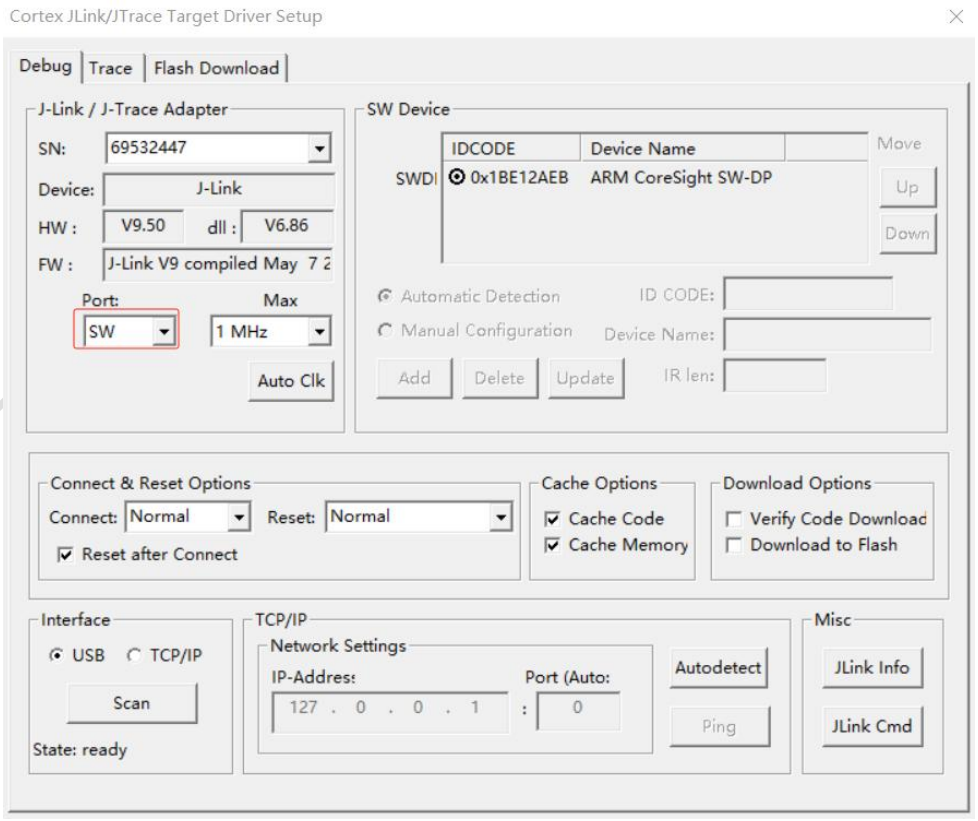


图 2-9 Debug 选项设置 2

g、【Utilities】选项设置，主要是 Flash 下载相关的设置，在进行该项设置前需将 SA50K_EFlash.FLM 文件添加到 Keil 安装路径下的 Flash 目录下，如

D:\Keil_v5\ARM\Flash 。

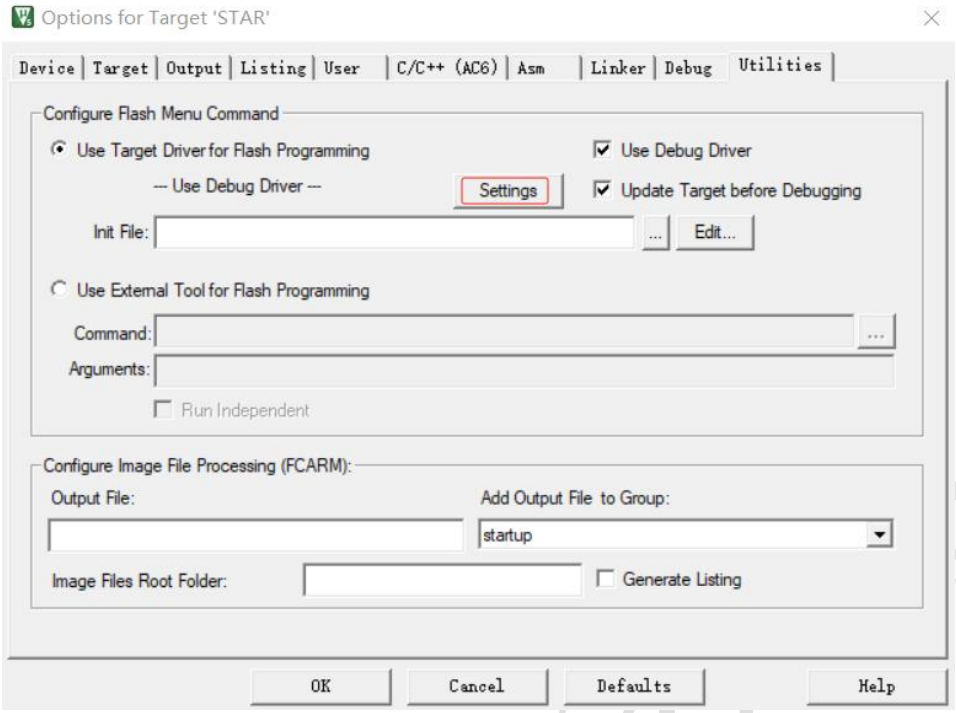


图 2- 10 Utilities 选项设置 1

点击【setting】按钮进入【Flash Download】设置界面，如下图所示。点击【add】按钮，选择"XiST 50k Flash V1.0"文件，具体设置如下图所示。

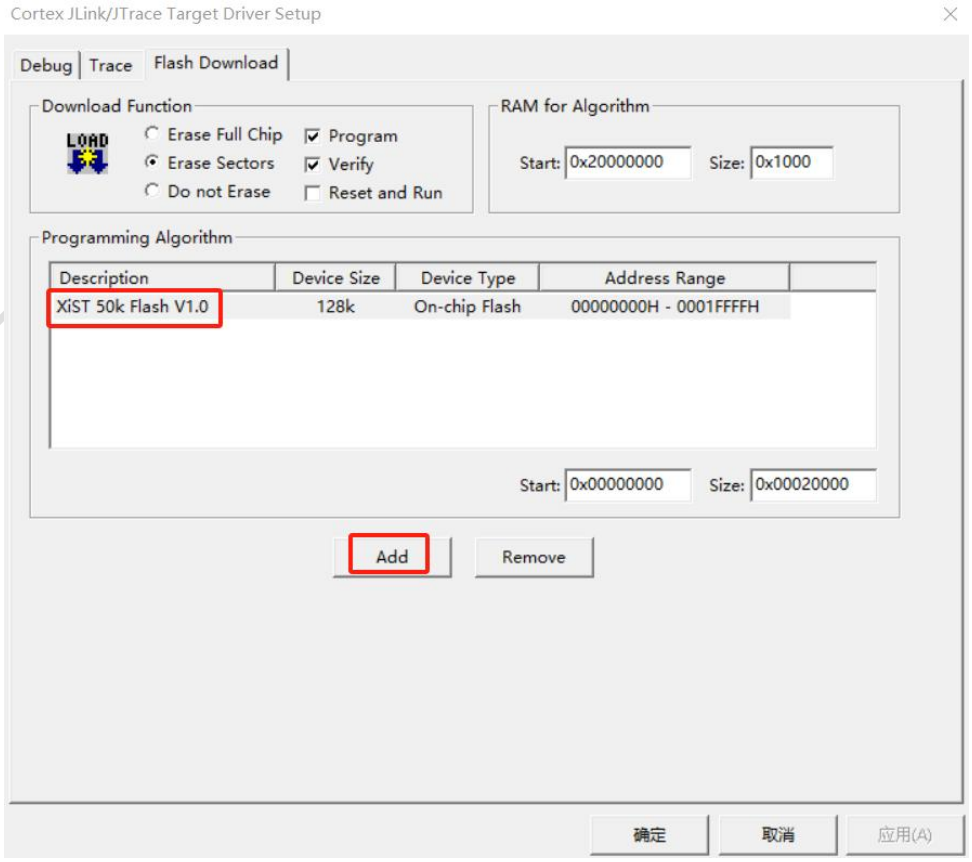


图 2- 11 Utilities 选项设置 2

按以上步骤完成工程属性设置和源文件添加完成后，就可以编译 STAR 的程序了。

2.2.3 系统时钟设置

建立硬件工程时在 FPGA 工程中为 STAR 设置了系统时钟源（只用于计算 APB 外设时钟频率），实际应用中，需在 STAR 软件工程中正确配置系统时钟频率参数，使之与 FPGA 侧提供的时钟频率一致。在《system_ARMCM3.c》文件中修改“__XTAL”的值即可，需要注意的是，STAR 实际工作的系统时钟频率为该值二分频后的结果。

```
/*----------------------------------------------------------------------------
  Define clocks
 *----------------------------------------------------------------------------*/
#define  XTAL            (50000000UL)      /* Oscillator frequency */

#define  SYSTEM_CLOCK    (XTAL / 2U)
```

图 2- 12 系统时钟设置

2.2.4 地址空间分配

在编程之前首先要对 STAR 各外设的地址空间及寄存器进行分配，地址分配都统一写在 STAR.h 文件中，如下所示。

```
/* Peripheral and SRAM base address */
#define STAR_FLASH_BASE       (0x00000000UL) /*!< (FLASH     ) Base Address */
#define STAR_SRAM_BASE        (0x20000000UL) /*!< (SRAM      ) Base Address */
#define STAR_PERIPH_BASE      (0x40000000UL) /*!< (Peripheral) Base Address */
#define STAR_TARGEXP0_BASE    (0x60000000UL) /*!< (TARGEXP0  ) Base Address */
#define STAR_TARGEXP1_BASE    (0xA0000000UL) /*!< (TARGEXP1  ) Base Address */

#define STAR_RAM_BASE         (0x20000000UL)
#define STAR_APB_BASE         (0x40000000UL)
#define STAR_AHB_BASE         (0x40010000UL)
#define AHB_TARGEXP0_BASE           (0x60000000UL)

/* APB peripherals                                     */
#define STAR_TIMER0_BASE      (STAR_APB_BASE + 0x0000UL)
#define STAR_TIMER1_BASE      (STAR_APB_BASE + 0x1000UL)
//#define STAR_DUALTIMER_BASE   (STAR_APB_BASE + 0x2000UL)
#define STAR_DUALTIMER0_BASE  (STAR_APB_BASE + 0x2000UL)
#define STAR_DUALTIMER1_BASE  (STAR_APB_BASE + 0xD000UL)
#define STAR_UART0_BASE       (STAR_APB_BASE + 0x4000UL)
#define STAR_UART1_BASE       (STAR_APB_BASE + 0x5000UL)
#define STAR_UART2_BASE       (STAR_APB_BASE + 0xF000UL)
#define STAR_PERI_RSTCTL_BASE (STAR_APB_BASE + 0x3000UL)
#define STAR_I2C_BASE         (STAR_APB_BASE + 0x6000UL)
#define STAR_ADC0_BASE        (STAR_APB_BASE + 0x7000UL)
#define STAR_ADC1_BASE        (STAR_APB_BASE + 0xE000UL)
#define STAR_WATCHDOG_BASE    (STAR_APB_BASE + 0x8000UL)
#define STAR_SPI0_BASE        (STAR_APB_BASE + 0xB000UL)
#define STAR_SPI1_BASE        (STAR_APB_BASE + 0xC000UL)
/* Mixed Peripherals*/
#define STAR_DMA_BASE (0x40009000UL)
```

图 2- 13 地址空间分配

2.2.5 程序下载步骤

程序下载支持两种方式:一是将 FPGA 比特流文件与 STAR 软件工程编译生成的可执行文件分开下载；二是将两者合并为一个文件后下载。分开下载时，用户先使用 HqFpga 下载器将逻辑侧生成的比特流文件下载，然后通过 ARM 仿真器将 STAR 编译后输出的可执行文件下载，这种方式适合对 STR 进行在线调试时使用。合并下载时，通过 HQFPGA 的下载器界面将 FPGA bin 文件和 STAR bin 文件合并,然后通过下载器将该合并后的文件烧录至 Flash 中。这种方式适用于程序固化以及不需要在线调试的场景。bin 文件合并及下载界面如下图所示。



图 2-14 FPGA 与 STAR bin 文件合并界面

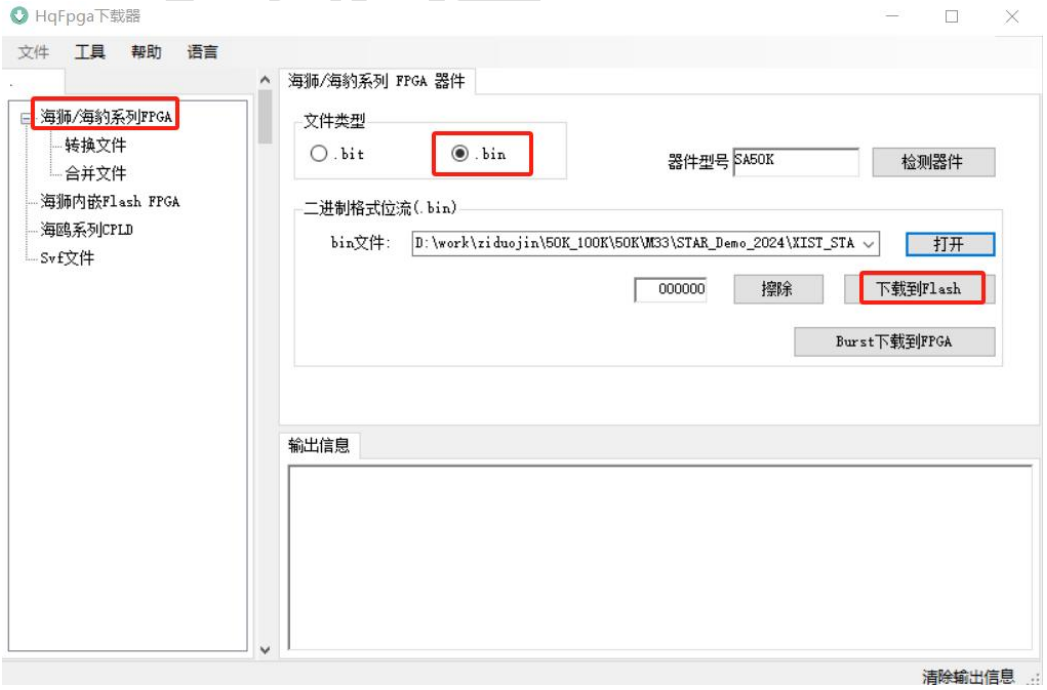


图 2-15 程序下载界面

2.2.6 程序在线调试

用户可在 Keil 开发环境中对 STAR 程序进行在线调试，程序调试需要软硬件环境支持，其中软件环境包括：

- J-Link 安装包：JLink_Windows_V880_x86_64.exe
- Flash 文件：SA50K_EFlash.FLM

硬件环境包括：

- FPGA 下载器：XIST USB Cable
- ARM 仿真器：J-Link 仿真器

调试步骤

可按照如下步骤建立 Debug 环境并进行程序在线调试。

1） 安装 J-Link 驱动程序并打开 J-Link Commander 工具，会出现如下图所示信息。

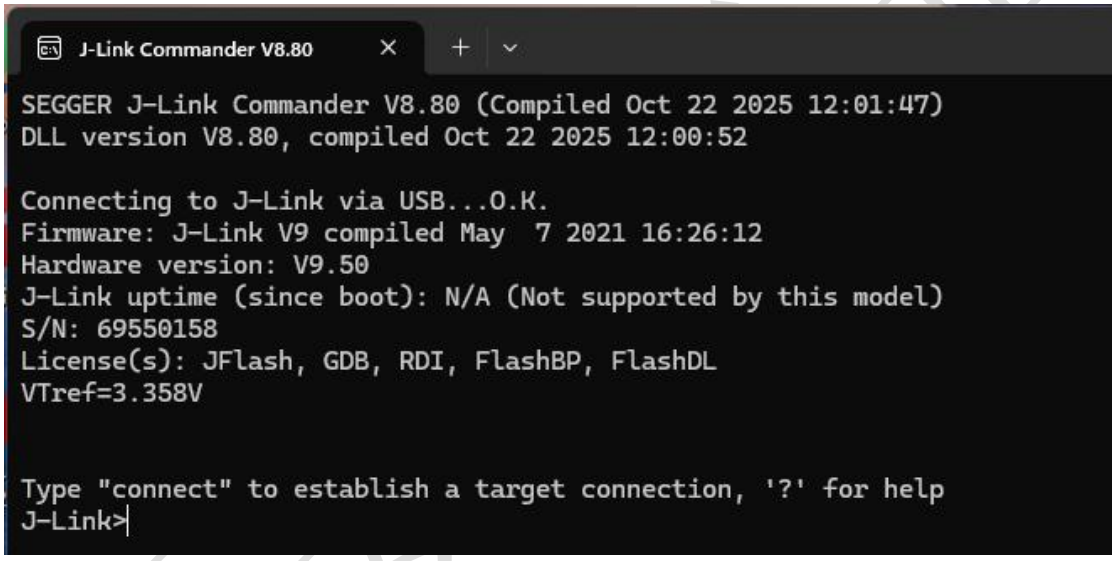


图 2-16 J-Link 命令行窗口

2） 使用 J-Link 进行 STAR 调试采用 SWD 调试模式。其硬件接线如下表所示(仅做参考，具体使用哪个 IO 可以在 FPGA 工程的物理约束文件中绑定）。

表 2-1 SWD 硬件引脚连接

ARM 仿真器	IST071-SA5Z-50-D0_7F484C_EVB_V3_1 开发板
DIO	B18(IO 接口座 J5 的 5 引脚)
CLK	B17(IO 接口座 J5 的 7 引脚)
GND	GND(IO 接口座 J5 的 2 引脚)

3） 使用 HQFPGA 软件下载器界面下载 FPGA 的 bin 文件和 STAR bin 文件合并后的 bin 文件。

4） 点击 Keil 软件中的图标，打开工程属性界面。

5） 选择【Debug】栏，选择 J-LINK/J-TRACE Cortex，然后点击【settings】按钮，如

下图所示。

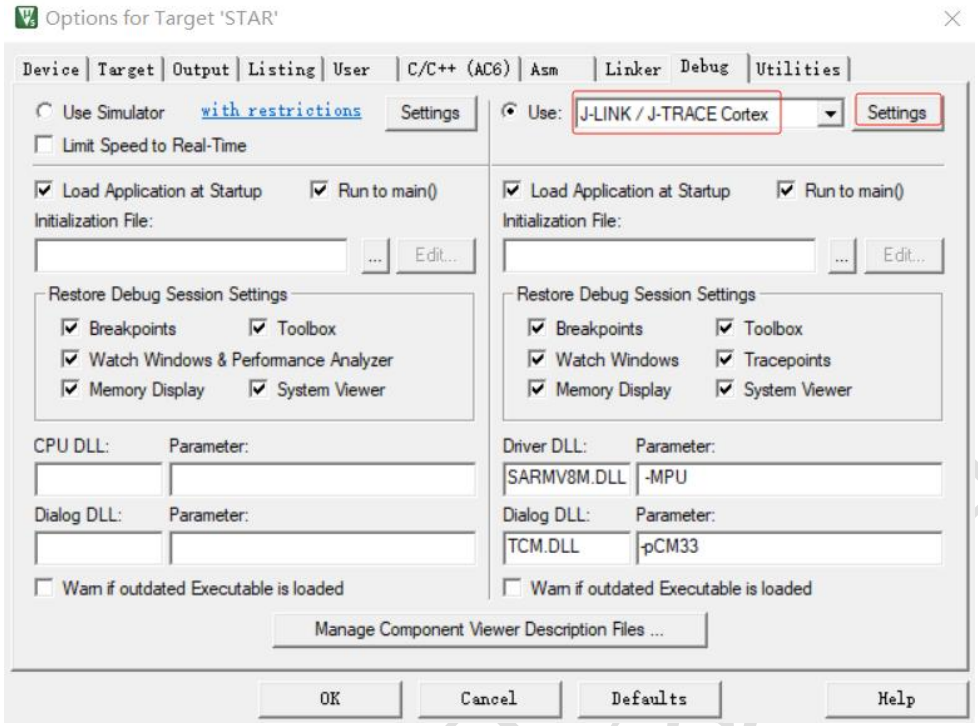


图 2-17 Debug 界面设置

6) 点击【setting】按钮后弹出对话框，设置如下图所示。注意：点击【setting】按钮时必须插上 ARM 仿真器，这样可以看是否检测到 STAR 硬核。

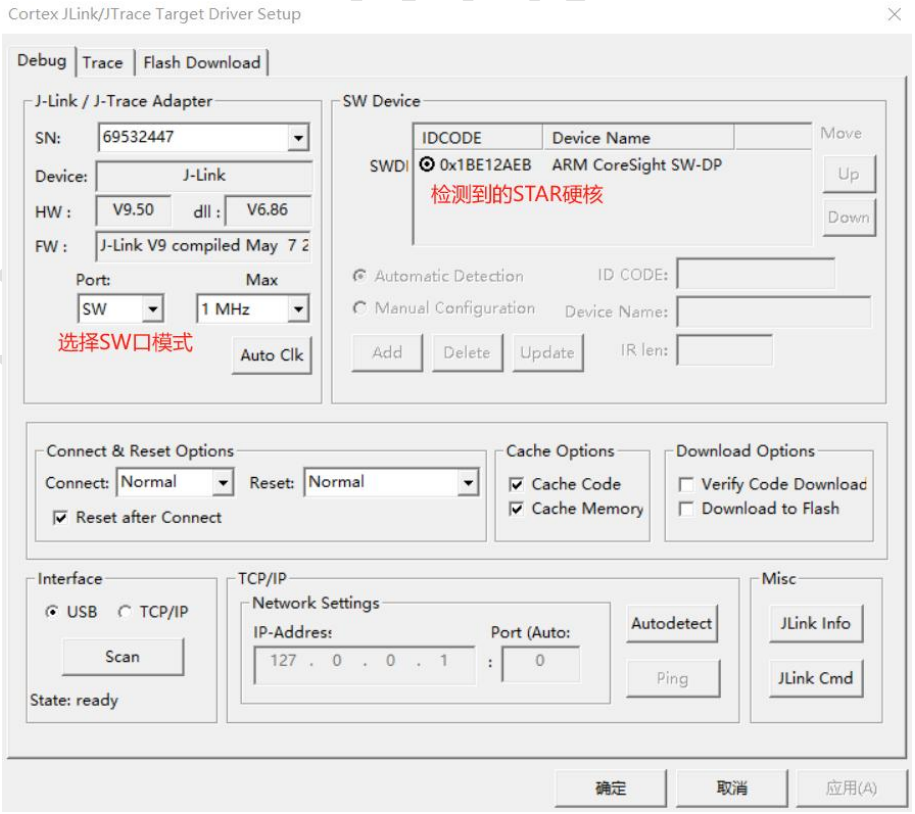


图 2-18 设置 Debug 调试模式

7） 点击 图标即可进入调试模式，如下图所示。

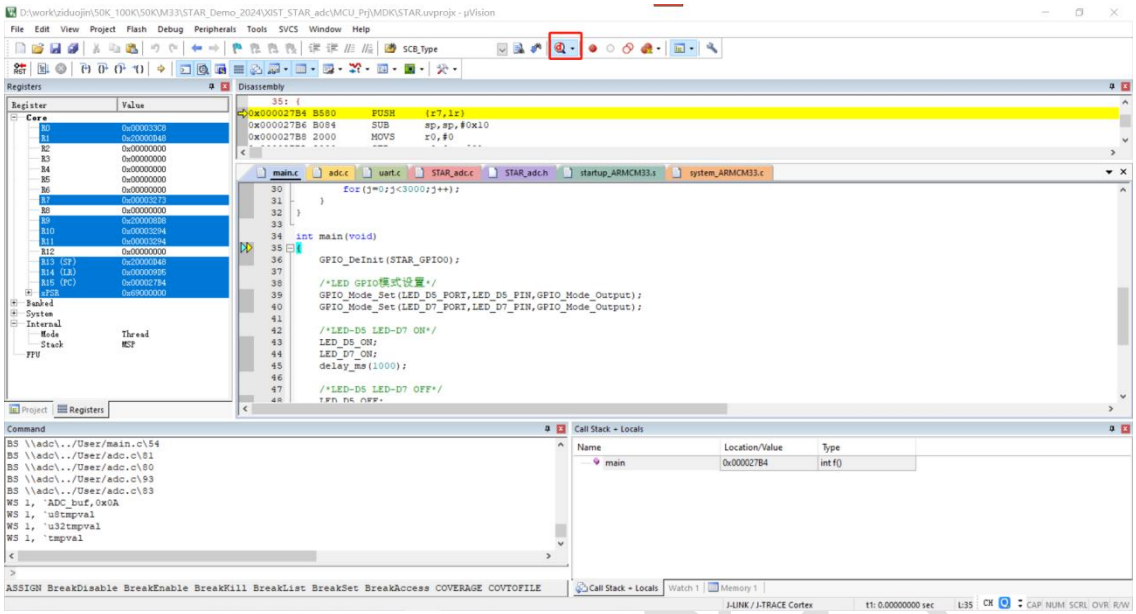


图 2-19 Debug 调试界面

3 参考例程

本章节描述的例程是基于智多晶 IST071-SA5Z-50-D0_7F484C_EVB_V3_1 开发板设计的 STAR 例程，基本涵盖了 STAR 硬核各模块的应用，可通过在 IP 配置界面勾选“输出范例”获取 DEMO 工程。

3.1 GPIO 输出例程+SysTick

本例程通过 STAR 的 GPIO 模块控制开发板上的 LED 点亮或熄灭，SysTick 作延时。通过该例程用户可了解 GPIO 模块的常用库函数以及 STAR GPIO 引脚的分配方式。

STAR 硬核的 GPIO 模块包含 32 个引脚，每个 GPIO 可设置为输入、输出或复用功能。GPIO 通过掩码访问，32 位 IO 划分为 byte0~byte3 四段。位掩码地址空间定义为四个数组，每个数组包括 256 个字。

3.1.1 LED 电路

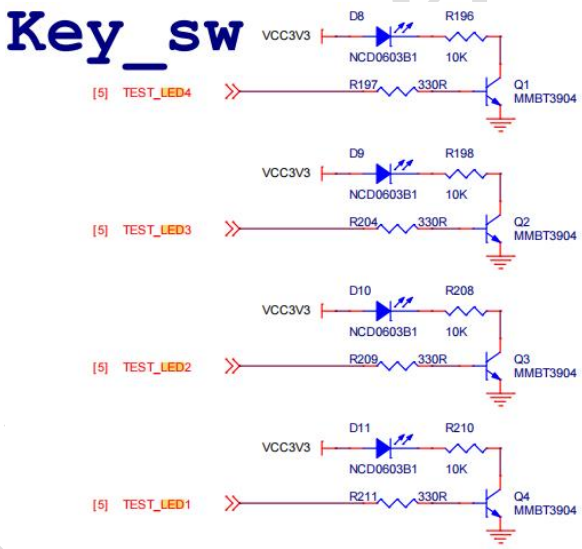


图 3-1 LED 电路

开发板的 LED 电路如上图所示。需要将 STAR 的 GPIO 分配到 LED2 和 LED3 上（例程中使用的 LED），控制对应的 GPIO 输出高低电平即可实现 LED 亮灭效果。

3.1.2 引脚分配

嵌入式 STAR 与独立的 ARM 芯片引脚分配的区别在于，SA5Z-50 器件 STAR 的 GPIO 是通过 FPGA 的引脚进行分配的，其引脚属性及上下拉状态也需要在 FPGA 的引脚分配界面中指定，本例程使用了 LED2 和 LED3，引脚约束如下图所示。

```
phycst.start
#clk
phycst.pin.set   {CLK_IN}      J15       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#SW4
phycst.pin.set   {RST_N}       P19       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#swd-port
phycst.pin.set   {SWD_IO}      B18       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.pin.set   {SWD_CLK}     B17       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
```

```
#LED-D1
phycst.pin.set   {FPGA_LED}        R19      -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D2
phycst.pin.set   {LED2}            T21      -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D3
phycst.pin.set   {LED3}            T20      -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.end
```

图 3-2 GPIO 物理约束

3.1.3 代码设计

在控制 LED 的 GPIO 输出高低电平之前，需使用图 3-3 所示的库函数将对应的 GPIO 配置为输出模式，该库函数有三个参数，第一个参数为 GPIO 的组别，固定设置为 STAR_GPIO0，第二个参数为具体的 GPIO，可选 GPIO_Pin_0~ GPIO_Pin_31，第三个参数为 GPIO 的模式，可设置为输出模式、输入模式和复用模式，在本例程中设置为输出模式。

```
void GPIO_Mode_Set(GPIO_TypeDef *GPIOx, uint16_t GPIO_Pin_x, GPIO_Mode_TypeDef GPIO_Mode)
```

图 3-3 GPIO 模式设置库函数

然后可采用图 3-4 所示库函数控制 LED 的亮灭，其中，GPIO_SetBit、GPIO_ResetBit 和 GPIO_TogglePin 这三个函数需要设置两个参数，第一个参数为 GPIO 的组别，固定设置为 STAR_GPIO0，第二个参数为具体的 GPIO，可选 GPIO_Pin_0~ GPIO_Pin_31。GPIO_WriteBit 函数需要设置三个参数，前两个参数与上同，第三个参数为输出高电平还是低电平。

```
1.  void GPIO_SetBit(GPIO_TypeDef *GPIOx, uint16_t GPIO_Pin_x);
2.  void GPIO_ResetBit( GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin_x);
3.  void GPIO_WriteBit(GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin_x, BitAction BitVal);
4.  void GPIO_TogglePin(GPIO_TypeDef *GPIOx, uint16_t GPIO_Pin_x);
```

图 3-4 GPIO 输出电平设置库函数

如下图所示两种 LED 控制方式，均可实现点亮 LED D2、LED D3，熄灭 LED D2、LED D3 的效果。

```
1.  GPIO_ResetBit( STAR_GPIO0, GPIO_Pin_0);
2.  GPIO_ResetBit( STAR_GPIO0, GPIO_Pin_1);
3.  GPIO_SetBit( STAR_GPIO0, GPIO_Pin_0);
4.  GPIO_SetBit( STAR_GPIO0, GPIO_Pin_1);
```

```
1.  GPIO_WriteBit( STAR_GPIO0, GPIO_Pin_0, Bit_RESET );
2.  GPIO_WriteBit( STAR_GPIO0, GPIO_Pin_1, Bit_RESET );
3.  GPIO_WriteBit( STAR_GPIO0, GPIO_Pin_0, Bit_SET );
4.  GPIO_WriteBit( STAR_GPIO0, GPIO_Pin_1, Bit_SET );
```

图 3-5 LED 控制方式

SysTick 的使用需要在 IP 配置时在时钟配置界面勾选“SysTick 定时器使能”，在使用前对其初始化配置，“systick.c”文件中提供了一个以 ms 为单位的延时函数，可以实现较精确延时，以下为主函数代码。

```
int main(void)
{
    GPIO_DeInit(STAR_GPIO0);
    SysTick_Init();

    /*LED GPIO 模式设置*/
    GPIO_Mode_Set(LED2_PORT,LED2_PIN,GPIO_Mode_Output);
```

```
    GPIO_Mode_Set(LED3_PORT,LED3_PIN,GPIO_Mode_Output);

    /*LED2 LED3 OFF*/
    LED2_OFF;
    LED3_OFF;
    Delay_ms(500);

    /*LED2 LED3 ON*/
    GPIO_WriteBit(LED2_PORT, LED2_PIN,Bit_SET);
    GPIO_WriteBit(LED3_PORT, LED3_PIN,Bit_SET);
    Delay_ms(2000);

    /*LED2 LED3 OFF*/
    GPIO_WriteBit(LED2_PORT, LED2_PIN,Bit_RESET);
    GPIO_WriteBit(LED3_PORT, LED3_PIN,Bit_RESET);
    Delay_ms(2000);
    while(1)
    {
        GPIO_TogglePin(LED2_PORT,LED2_PIN);
        Delay_ms(1000);
        GPIO_TogglePin(LED3_PORT,LED3_PIN);
        Delay_ms(1000);
    }
}
```

图 3-6 主函数代码

3.1.4 下载验证

程序下载之后，用户在开发板上可观测到 LED2 与 LED3 交替闪烁。

3.2 GPIO 输入例程

本例程实现按下按键 SW1，则点亮 LED2，松开 SW1 则熄灭 LED2。

3.2.1 按键电路

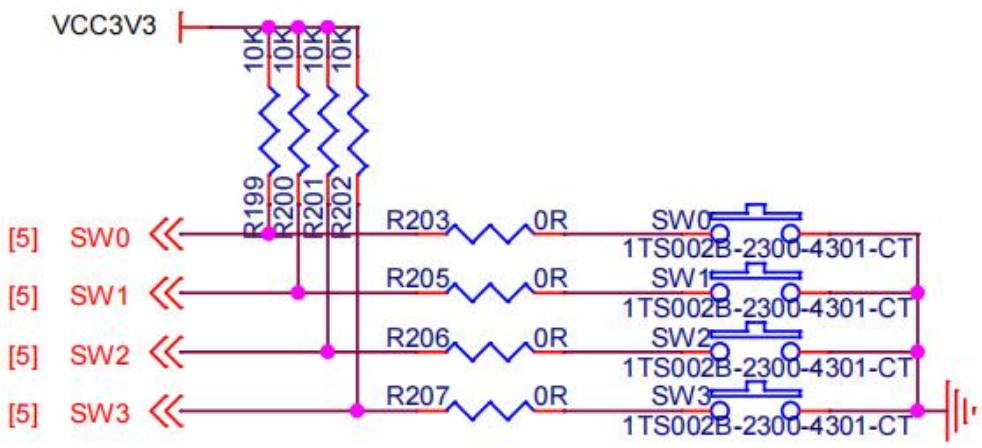


图 3-7 按键电路

3.2.2 引脚分配

本例程 LED2、LED3 使用 GPIO0、GPIO1，SW1 使用 GPIO2 ，约束结果如下图所示。

```
phycst.start
#clk
phycst.pin.set    {CLK_IN}      J15       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#SW0
```

```
phycst.pin.set   {RST_N}       P19       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#swd-port
phycst.pin.set   {SWD_IO}      B18       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.pin.set   {SWD_CLK}     B17       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D1
phycst.pin.set   {FPGA_LED}    R19       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D2
phycst.pin.set   {LED2}        T21       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D3
phycst.pin.set   {LED3}        T20       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#SW1
phycst.pin.set {SWITCH1}       P20       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.end
```

图 3-8 GPIO 物理约束

3.2.3 代码设计

读取按键状态需要使用到的库函数为：

```
1.    uint8_t GPIO_ReadInputBit(GPIO_TypeDef *GPIOx, uint16_t GPIO_Pin_x)
```

图 3-9 GPIO 读取输入引脚状态

此函数用来读取 GPIO 的输入电平，可以用于判断按键是否被按下。该函数需要设置两个参数，第一个参数为 STAR_GPIO0，第二个参数为按键对应的 GPIO。本实验将按键 SW1 绑定在 GPIO2 引脚上。

```
int main(void)
{
    uint8_t keyval;
    GPIO_DeInit(STAR_GPIO0);

    /*LED GPIO 模式设置*/
    GPIO_Mode_Set(LED2_PORT,LED2_PIN,GPIO_Mode_Output);
    GPIO_Mode_Set(LED3_PORT,LED3_PIN,GPIO_Mode_Output);

    /*LED2 LED3 OFF*/
    LED2_OFF;
    LED3_OFF;
    delay_ms(500);

    /*LED2 LED3 ON*/
    GPIO_WriteBit(LED2_PORT, LED2_PIN,Bit_SET);
    GPIO_WriteBit(LED3_PORT, LED3_PIN,Bit_SET);
    delay_ms(2000);

    /*LED2 LED3 OFF*/
    GPIO_WriteBit(LED2_PORT, LED2_PIN,Bit_RESET);
    GPIO_WriteBit(LED3_PORT, LED3_PIN,Bit_RESET);
    delay_ms(2000);

    while(1)
    {
        keyval = GPIO_ReadInputBit(STAR_GPIO0,GPIO_Pin_2); //读取按键对应 GPIO 的电平
        if(keyval==Bit_RESET)
        {
            LED2_ON;
        }
        else
        {
            LED2_OFF;
        }
    }
}
```

图 3-10 按键控制 LED

按键控制的代码如上图所示，通过读取按键 SW1 对应的 GPIO2 引脚的电平状态来控制 LED 亮灭，当为高电平时 LED 灭，为低电平时 LED 亮。

3.2.4 下载验证

程序下载之后，用户在开发板上按下按键 SW1 后 LED2 点亮，松开按键 SW1 后，LED2 熄灭。

3.3 外部中断例程

本例程修改自 3.2 章节按键输入例程，将按键状态的判断从之前的查询方式改为中断方式。在本例程中，中断触发类型设置为低电平触发，即当按键按下时进入中断服务程序，在中断服务函数中点亮 LED，然后清除中断，在中断未触发时 LED 为熄灭状态。这样可以观测到的现象为，按下按键 SW1 则 LED2 点亮，松开按键 SW1 则 LED2 熄灭 。

3.3.1 按键电路

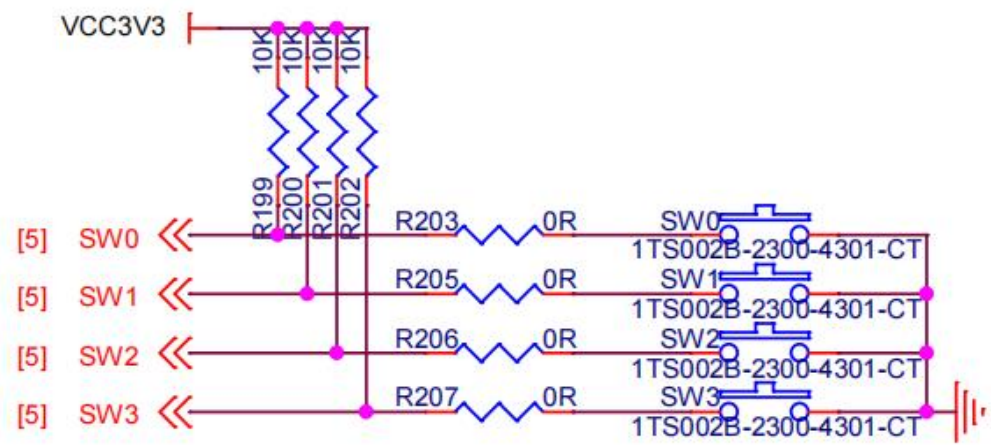


图 3-11 按键电路

3.3.2 引脚分配

本例程 LED2、LED3 使用 GPIO0、GPIO1，SW1 使用 GPIO2 ，约束结果如下图所示。

```
phycst.start
#clk
phycst.pin.set   {CLK_IN}      J15       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#SW0
phycst.pin.set   {RST_N}       P19       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#swd-port
phycst.pin.set   {SWD_IO}      B18       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.pin.set   {SWD_CLK}     B17       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D1
phycst.pin.set   {FPGA_LED}    R19       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D2
phycst.pin.set   {LED2}        T21       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D3
```

```
phycst.pin.set    {LED3}          T20        -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#SW1
phycst.pin.set {SWITCH1}          P20        -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.end
```

图 3-12 GPIO 物理约束

3.3.3 代码设计

将 GPIO2 引脚配置为低电平触发中断源并使能，如下图所示。

```
void gpio2_interrupt_init(void)
{
    /*GPIO2 设置为低电平触发*/
    GPIO_Interrupt_Config(STAR_GPIO0,GPIO_Pin_2,Bit_LOWLEVEL,Interrupt_ENABLE);
    NVIC_ClearPendingIRQ(PORT0_2_IRQn);
    NVIC_EnableIRQ(PORT0_2_IRQn);
}
```

图 3-13 按键中断初始化

GPIO_Interrupt_Config 函数为按键中断初始化的主要函数，本函数配置具体哪个 IO 引脚触发中断和中断的触发类型。NVIC_EnableIRQ 作用为使能中断控制器，使用外部中断时必须将相应 IO 使能。

```
void PORT0_2_Handler(void)
{
    /*LED2 ON*/
    GPIO_SetBit(STAR_GPIO0, GPIO_Pin_0);
    //清除中断
    GPIO_Interrupt_Clear(STAR_GPIO0,GPIO_Pin_2);
    NVIC_ClearPendingIRQ(PORT0_2_IRQn);
}
```

图 3-14 中断服务函数

使用对应的 I/O 引脚来声明中断服务函数，在函数内可编写中断触发后执行的程序。每一个 IO 对应的服务函数不同。GPIO_Interrupt_Clear 函数用来清除当前中断。

代码如下图所示，中断初始化完成后即可根据中断状态点亮 LED 灯。

```
int main(void)
{
    uint8_t keyval;
    GPIO_DeInit(STAR_GPIO0);

    /*LED GPIO 模式设置*/
    GPIO_Mode_Set(LED2_PORT,LED2_PIN,GPIO_Mode_Output);
    GPIO_Mode_Set(LED3_PORT,LED3_PIN,GPIO_Mode_Output);

    /*LED2 LED3 OFF*/
    LED2_OFF;
    LED3_OFF;
    delay_ms(500);

    /*LED2 LED3 ON*/
    GPIO_WriteBit(LED2_PORT, LED2_PIN,Bit_SET);
    GPIO_WriteBit(LED3_PORT, LED3_PIN,Bit_SET);
    delay_ms(2000);

    /*LED2 LED3 OFF*/
    GPIO_WriteBit(LED2_PORT, LED2_PIN,Bit_RESET);
    GPIO_WriteBit(LED3_PORT, LED3_PIN,Bit_RESET);
    delay_ms(2000);

    gpio2_interrupt_init();
    while(1)
    {
        LED2_OFF;
        LED3_OFF;
    }
```

```
}
```

图 3-15 用户代码设计

3.3.4 下载验证

程序下载之后，用户在开发板上按下按键 SW1 后 LED2 点亮，松开按键 SW1 后 LED2 熄灭。

3.4 串口例程

STAR 的 UART 模块支持无奇偶校验的 8 位通信，停止位固定为 1。

本例程实现 UART0 的收发回环测试，用以验证 STAR 的 UART 收发功能。

3.4.1 UART 简介

功能引脚

TXD：发送数据输出引脚。

RXD：接收数据输入引脚。

数据寄存器

UART 数据寄存器数据位宽为 8 位，可缓存发送及接收数据。

控制寄存器

CTRL	0x008	RW	7	0x00	[6] High-speed test mode for TX only. [5] RX overrun interrupt enable. [4] TX overrun interrupt enable. [3] RX interrupt enable. [2] TX interrupt enable. [1] RX enable. [0] TX enable.

图 3-16 串口控制寄存器

串口控制寄存器定义如上图所示，可对串口的发送、接收和中断的使能进行设置。

波特率设置

BAUDDIV 寄存器用于设置串口波特率。计算公式为：BAUDDIV = PCKClock / UART_BaudRate，其中，BAUDDIV 为串口波特率分频系数，PCKClock 为 APB 总线的时钟频率，UART_BaudRate 为串口波特率。

3.4.2 串口接口电路

开发板上有一个 USB 转串口接口，本例程使用该接口进行串口的收发测试。

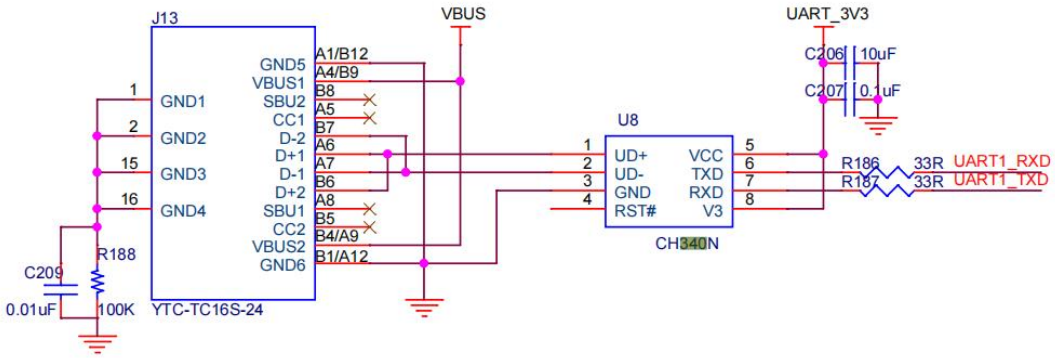


图 3-17 串口接口电路

3.4.3 引脚分配

STAR 的 UART0 收发引脚复用在 GPIO2 和 GPIO3 上，关于 GPIO 引脚复用的详细介绍请参考《UG05205_SA5Z-50 ARM STAR User Guide_V1.0》2.4.2 章节的内容。本例程物理约束如下图所示。

```
phycst.start
#clk
phycst.pin.set   {CLK_IN}      J15       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#SW0
phycst.pin.set   {RST_N}       P19       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#swd-port
phycst.pin.set   {SWD_IO}      B18       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.pin.set   {SWD_CLK}     B17       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D1
phycst.pin.set   {FPGA_LED}    R19       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D2
phycst.pin.set   {LED2}        T21       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D3
phycst.pin.set   {LED3}        T20       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
######UART0######
#uart0-rx
phycst.pin.set   {UART0_RX}    G15       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#uart0-tx
phycst.pin.set   {UART0_TX}    G16       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.end
```

图 3-18 UART 物理约束

3.4.4 代码设计

该例程首先对串口进行初始化，接着打印例程信息字符串，然后等待用户发送数据，在 UART0 接收中断中把数据回显给用户，实现 echo 的效果。

UART0 的初始化配置如下图所示，主要是对 UART0 的收发引脚所在的 GPIO 进行引脚复用，然后对串口进行初始化配置，设置好波特率并使能发送和接收。

```
static void uartx_init(UART_TypeDef *UARTx,uint32_t BaudRate)
{
    UART_InitTypeDef  UART_InitStructure;

    UART_DeInit(UARTx);

    /*将对应的 GPIO 引脚复用为对应 UART 的收发引脚*/
    if(UARTx ==STAR_UART0)
    {
        GPIO_AF_Config(STAR_GPIO0, GPIO_AF_USART0_RXD, ENABLE); //GPIO[2]
        GPIO_AF_Config(STAR_GPIO0, GPIO_AF_USART0_TXD, ENABLE); //GPIO[3]
    }
    else if(UARTx ==STAR_UART1)
    {
        GPIO_AF_Config(STAR_GPIO0, GPIO_AF_USART1_RXD, ENABLE); //GPIO[4]
        GPIO_AF_Config(STAR_GPIO0, GPIO_AF_USART1_TXD, ENABLE); //GPIO[5]
    }
    else
    {
        GPIO_AF_Config(STAR_GPIO0, GPIO_AF_USART2_RXD, ENABLE); //GPIO[22]
        GPIO_AF_Config(STAR_GPIO0, GPIO_AF_USART2_TXD, ENABLE); //GPIO[23]
    }

    UART_InitStructure.UART_BaudRate = BaudRate;
    /*收发使能*/
    UART_InitStructure.UART_CTRL =  UART_CTRL_TxEnable | UART_CTRL_RxEnable;
    UART_Init( UARTx, &UART_InitStructure);

    /*接收中断配置*/
    uart_interrupt_init(UARTx);
```

```
}
```

图 3-19 UART0 初始化

在初始化 UART0 时，还需开启 UART0 的接收中断，中断配置如下图所示。

```
static void uart_interrupt_init(UART_TypeDef *UARTx)
{
    /*使能接收中断*/
    UART_Interrupt_Config(UARTx, UART_RxInterrupt, ENABLE);

    if(UARTx ==STAR_UART0)
    {
        NVIC_ClearPendingIRQ(UART0_IRQn);
        NVIC_EnableIRQ(UART0_IRQn);
    }
    else if(UARTx ==STAR_UART1)
    {
        NVIC_ClearPendingIRQ(UART1_IRQn);
        NVIC_EnableIRQ(UART1_IRQn);
    }
    else
    {
        NVIC_ClearPendingIRQ(UART2_IRQn);
        NVIC_EnableIRQ(UART2_IRQn);
    }
}
```

图 3-20 UART0 中断配置

主函数中将 UART0 波特率设置为 115200，初始化完成后，等待用户发送数据；在 UART0 中断处理函数中将用户发送的数据发送给用户实现 echo 的效果。

```
int main(void)
{
    GPIO_DeInit(STAR_GPIO0);

    /*LED GPIO 模式设置*/
    GPIO_Mode_Set(LED2_PORT,LED2_PIN,GPIO_Mode_Output);
    GPIO_Mode_Set(LED3_PORT,LED3_PIN,GPIO_Mode_Output);

    /*LED2 LED3 ON*/
    LED2_ON;
    LED3_ON;
    delay_ms(1000);

    /*LED2 LED3 OFF*/
    LED2_OFF;
    LED3_OFF;

    uart_init(115200);
    printf("STAR uart0 echo test...\r\n");

    while(1)
    {
        ;主函数循环不做其他业务，等待用户通过 uart 发送数据，echo 在接收中断中实现
    }
}
```

图 3-21 主函数用户代码

```
void UART0_Handler(void)
{
    uint8_t ch;

    if(UART_Get_Interrupt_Status(STAR_UART0,UART_RxInterrupt)==SET) //接收中断
    {
        /*读取接收到的数据*/
        ch=UART_ReceiveData(STAR_UART0);
        /*将接收到的数据发送回去*/
        UART_SendData(STAR_UART0,ch);
        /*清除中断*/
        UART_Interrupt_Clear(STAR_UART0, UART_RxInterrupt);
    }
```

```
}
```

图 3-22 UART0 中断服务函数

3.4.5 下载验证

程序下载后，打开串口助手，设置好波特率后，在发送窗口发送任意字符，在接收窗口可收到刚发送的字符数据。



图 3-23 串口调试助手收发数据

3.5 I2C 例程

本例程使用 STAR 的硬件 I2C 向 EEPROM(24C256）写入数据，然后回读回来，以验证 STAR 的 I2C 通信功能。发送和回读的数据通过串口打印出来，这样可以直观地看到测试结果。

3.5.1 I2C 简介

STAR 的 I2C 模块符合标准 I2C 通信协议，目前只支持 I2C 主模式，主机和从机地址可设置为 7 位或 10 位地址。

I2C 时钟频率计算公式：I2C_ClkFreq=（PCLK/4）/(FREQ_DIV[15:3]+1）；其中 PCLK 为 APB 总线的时钟频率，FREQ_DIV[15:3]为频率分频系数。

3.5.2 EEPROM 电路

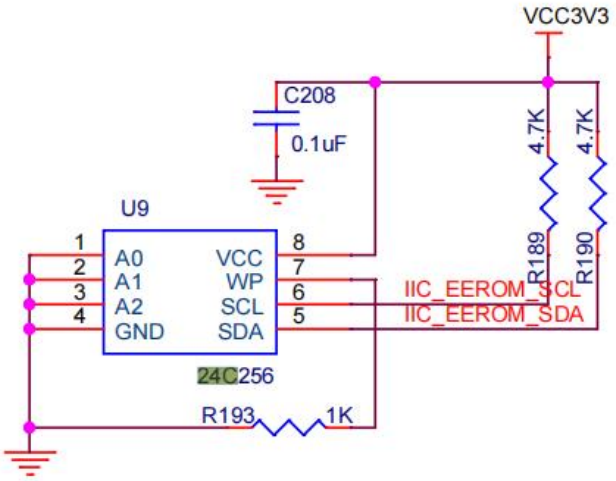


图 3-24 EEPROM 电路图

开发板上 EEPROM 的存储芯片为 24C256，64 Byte 存储容量，该芯片采用 I2C 接口。

3.5.3 引脚分配

STAR 的 I2C 引脚复用在 GPIO15 和 GPIO16 上，关于 GPIO 引脚复用的详细介绍请参考《UG05205_SA5Z-50 ARM STAR User Guide_V1.0》2.4.2 章节的内容，UART0 的引脚分配情况如串口例程一样，具体约束如下图所示。

```
phycst.start
#clk
phycst.pin.set   {CLK_IN}      J15       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#SW4
phycst.pin.set   {RST_N}       P19       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#swd-port
phycst.pin.set   {SWD_IO}      B18       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.pin.set   {SWD_CLK}     B17       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D1
phycst.pin.set   {FPGA_LED}    R19       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D2
phycst.pin.set   {LED2}        T21       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D3
phycst.pin.set   {LED3}        T20       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
######UART0######
#uart0-rx
phycst.pin.set   {UART0_RX}    G15       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#uart0-tx
phycst.pin.set   {UART0_TX}    G16       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
######I2C######
#uart0-rx
phycst.pin.set   {I2C_SCL}     H18       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#uart0-tx
phycst.pin.set   {I2C_SDA}     M21       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.end
```

图 3-25 物理约束

3.5.4 代码设计

该例程首先配置需要用到的外设资源，如 I2C、UART0 控制引脚，然后向 EEPROM 的 0 地址写入一串数据，然后从该地址回读数据，并将写入的数据和回读的数据通过串口打印，通过打印的信息可以查看写入的数据和回读的数据是否一致。

I2C 的初始化如下图所示，主要是对 I2C 引脚所在的 GPIO 进行引脚复用，然后对 I2C 进行初始化配置。

```
/*I2C 相关GPIO 引脚设置*/
void I2C_Pin_Init(void)
{
    /*将对应的GPIO 复用为I2C 的引脚*/
    GPIO_Mode_Set(STAR_GPIO0,GPIO_Pin_15,GPIO_Mode_AF);
    GPIO_AF_Config( STAR_GPIO0, GPIO_AF_I2C0SCL, ENABLE); //GPIO[15]

    GPIO_Mode_Set(STAR_GPIO0,GPIO_Pin_16,GPIO_Mode_AF);
    GPIO_AF_Config( STAR_GPIO0, GPIO_AF_I2C0SDA, ENABLE); //GPIO[16]
}

/*I2C 初始化设置*/
void EEPROM_I2C_Init(void)
{
    I2C_DeInit(STAR_I2C);
    /*I2C 配置:  I2C 主模式*/
    I2C_Init(STAR_I2C,I2C_CLK_FREQ,I2C_Ack_Enable, I2C_Mode_Master, I2C_OWN_ADDR);
}
```

图 3-26 I2C 初始化

STAR 往 EEPROM 读写数据时要按照 EEPROM 数据手册中的读写时序进行。写数据我们按 24C256 数据手册中的页写操作来完成，一次最多可以往 24C256 写入 64 字节的数据，页写的时序图如下所示。

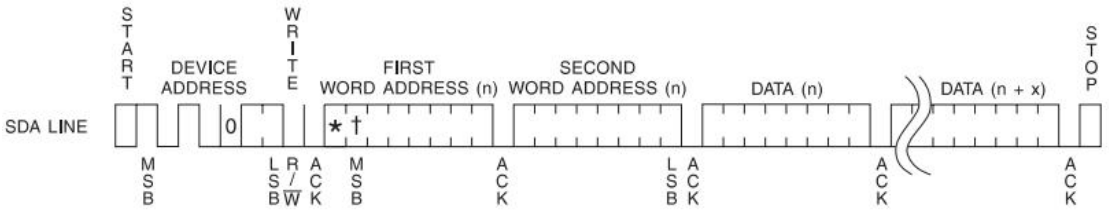


图 3-27 EEPROM 页写时序图

下图代码为使用STAR 的I2C 按照24C256 数据手册中提供的页写时序对EEPROM 器件进行数据写入操作。

```
void I2C_ByteWrite_nBytes (uint8_t slave_addr,uint8_t word_addr,uint8_t* Buffer,uint8_t Num)
{
    uint16_t addr_wtite = 0x0000;
    uint16_t num=0;

    EEPROM_I2C_Init();
    /*使能发送*/
    I2C_TxEnable(STAR_I2C,ENABLE);

    /*从设备地址*/
    addr_wtite = 0x0000 | slave_addr;
    /*设置I2C 从设备EEPROM 的地址*/
    I2C_SlaveAddr_Config(STAR_I2C,addr_wtite);

    I2C_GenerateSTART(STAR_I2C, ENABLE);
    /*设置收发数据的个数*/
    I2C_WriteByteNum_Config(STAR_I2C,2);
```

```
    /*发送要写入的存储位置的地址*/
    I2C_WriteByte(STAR_I2C,word_addr>8);   //24C64 wordAddr 是 16bit 分两次发送
    while(I2C_CheckACKIsFail(STAR_I2C));    //等待 ACK
    I2C_WriteByte(STAR_I2C, word_addr&0xff);
    while(I2C_CheckACKIsFail(STAR_I2C));    //等待 ACK
    //I2C_WaitWriteDataFinish(STAR_I2C);
    /*发送写入的数据*/
    while(Num)
    {
        I2C_WriteByteNum_Config(STAR_I2C,1);
        I2C_WaitTxFIFOIsNotFull(STAR_I2C);等待发送 FIFO 有空位
        I2C_WriteByte(STAR_I2C,*Buffer);
        while(I2C_CheckACKIsFail(STAR_I2C));等待 ACK
        Buffer ++;
        Num--;
    }
    I2C_WaitWriteDataFinish(STAR_I2C);
    I2C_GenerateSTOP(STAR_I2C,ENABLE );
}
```

图 3-28　I2C 页写

上图为 I2C 主机模式发送数据代码，最多可一次对 EEPROM 写入 64 个字节数据。使用此函数需要设置四个参数，第一个参数设置从机器件地址，第二个参数设置写数据的起始地址，第三个参数为写入的数据 Buff 地址，第四个数据为要写入的数据长度。I2C_TxEnable 函数设置发送功能，I2C_GenerateSTART 用来产生起始信号，表示本次数据传输开始。I2C_WriteByteNum_Config 函数设置发送的数据个数。I2C_WaitWriteDataFinish 函数表示等待数据发送结束，最后使用 I2C_GenerateSTOP 函数来产生 I2C 结束信号。

读操作我们按照 24C256 数据手册中的随机读操作来完成，随机读的时序如下图所示。

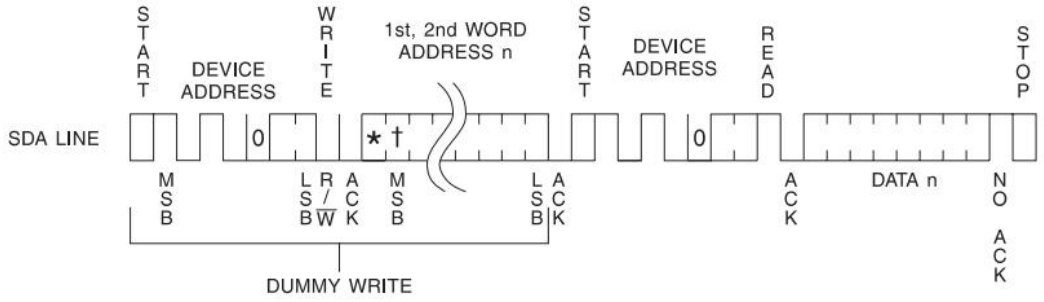


图 3-29 EEPROM 随机读时序图

```
void I2C_EEPROM_RandomRead(uint8_t slave_addr,uint8_t word_addr,uint8_t* Buffer,uint16_t
rx_data_num)
{
    uint16_t addr_read  = 0x8000;   //读写操作由 SLV_ADDR 的 bit[15]确定，该位为 0 表示写操作，为 1 表
示读操作
    uint16_t addr_wtite = 0x0000;

    EEPROM_I2C_Init();
    /*使能发送*/
    I2C_TxEnable(STAR_I2C,ENABLE); //TX ENABLE

    /*从设备地址*/
    addr_wtite |= slave_addr;
    /*设置 I2C 从设备 EEPROM 的地址*/
    I2C_SlaveAddr_Config(STAR_I2C,addr_wtite);


    I2C_GenerateSTART(STAR_I2C,ENABLE);
```

```
    /*设置收发数据的个数*/
    I2C_WriteByteNum_Config(STAR_I2C,2);

    /*发送读取存储位置的地址*/
    I2C_WriteByte(STAR_I2C,word_addr>8);   //24C64 wordAddr 是 16bit 分两次发送
    while(I2C_CheckACKIsFail(STAR_I2C));    //等待 ACK
    I2C_WriteByte(STAR_I2C,word_addr&0xff); //08  address
    while(I2C_CheckACKIsFail(STAR_I2C));    //等待 ACK
    I2C_WaitWriteDataFinish(STAR_I2C);

    I2C_GenerateSTOP(STAR_I2C,ENABLE );
    I2C_TxEnable(STAR_I2C,DISABLE);

    I2C_Init(STAR_I2C,I2C_CLK_FREQ,I2C_Ack_Enable, I2C_Mode_Master, I2C_OWN_ADDR);

    /*读从机的地址*/
    addr_read |= slave_addr;
    I2C_SlaveAddr_Config(STAR_I2C,addr_read);read
    /*使能接收*/
    I2C_RxEnable(STAR_I2C,ENABLE );

    /*开始接收读取到的数据*/
    I2C_GenerateSTART(STAR_I2C,ENABLE);
    while(rx_data_num)
    {
        I2C_WaitReceiveReady(STAR_I2C);
        *Buffer =  I2C_ReceiveByte(STAR_I2C);
        Buffer++;
        rx_data_num--;
    }
    I2C_GenerateSTOP(STAR_I2C,ENABLE);
}
```

图 3-30 I2C 主机从 EEPROM 读数据

上图按照 24C256 随机读时序，从 EEPROM 指定地址读取一段数据。本函数需要设置四个参数，第一个参数为从机器件地址，第二个参数为读数据的起始地址，第三个参数为读取数据的保存地址，第四个参数为读取数据的长度。

EEPROM 读写测试代码和主函数代码如下所示。

```
void EEPROM_ReadWrite_test(void)
{
    int i=0;
    uint16_t rw_num=64;
    uint16_t rw_addr=0;

    I2C_Pin_Init();

    /*向 EEPROM 的 rw_addr 地址开始写入 rw_num 字节数据*/
    printf("I2C write data to EEPROM...\r\n");
    memset(Tx_Buffer,0x0,sizeof(Tx_Buffer));
    for(i=0;i<rw_num;i++)
    {
        Tx_Buffer[i] =i;
    }
    I2C_ByteWrite_nBytes(EEPROM_DEV_ADDR,rw_addr,Tx_Buffer,rw_num);
    printf("write data:\r\n");
    for(i=0;i<rw_num;i++)
    {
        printf("0x%x ", Tx_Buffer[i]);
    }
    printf("\r\n");


    /*从 EEPROM 的 0 地址连续读取 rw_num 字节数据，验证之前写入的数据*/
    printf("I2C read data from EEPROM...\r\n");
    memset(Rx_Buffer,0x0,sizeof(Rx_Buffer));
    I2C_EEPROM_RandomRead(EEPROM_DEV_ADDR,rw_addr,Rx_Buffer,rw_num);

    printf("read data:\r\n");
```

```
    for(i=0;i<rw_num;i++)
    {
        printf("0x%x ", Rx_Buffer[i]);
    }
    printf("\r\n");
}

int main(void)
{
    GPIO_DeInit(STAR_GPIO0);

    /*LED GPIO 模式设置*/
    GPIO_Mode_Set(LED2_PORT,LED2_PIN,GPIO_Mode_Output);
    GPIO_Mode_Set(LED3_PORT,LED3_PIN,GPIO_Mode_Output);

    /*LED2 LED3 ON*/
    LED2_ON;
    LED3_ON;
    delay_ms(1000);

    /*LED2 LED3 OFF*/
    LED2_OFF;
    LED3_OFF;

    uart_init(115200);
    EEPROM_ReadWrite_test();
}
```

图 3-31 I2C 读写测试函数和主函数

3.5.5 下载验证

程序下载后，在串口助手界面能观察到 STAR 往 EEPROM 写入的数据和回读的数据，通过查看串口打印的信息可以直观地进行数据对比，以验证 I2C 读写操作是否正常。

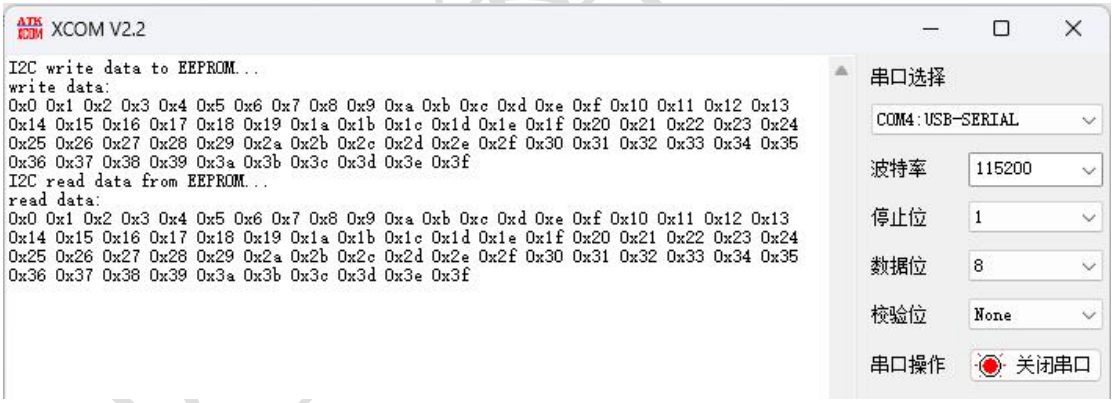


图 3-32 串口调试界面

3.6 ADC 例程

本例程使用 ADC 采样 ADC 通道 0 的电压值，并将采样的数据通过 UART0 打印出来。

3.6.1 ADC 介绍

STAR 具有 1 个精度为 12 位的 ADC，称为 SADC。SADC 可用于监测芯片内部温度，也可用在 FPGA 侧，但是 STAR 有使用 SADC 的最高优先级。SADC 有 12 个通道，其中通道 0 到通道 7 作为通用通道使用，通道 8 到通道 11 用于转换芯片信息。

ADC 是一种逐次逼近式的模数转换器，转换可在连续模式下进行，也可在单次模式下进行。模拟输入通道可以单独选择或设置为扫描模式。扫描模式时多个通道依次

转换，每次转换的通道数递增，然后将数字结果存储在寄存器中。ADC 转换结果为 12 位，ADC 数据寄存器为 16 位，数据的存储方式可以选择左对齐或右对齐。

ADC 模块由 APB 时钟驱动，ADC 模拟部分使用分频器产生 ADC_CLK，允许根据应用程序的要求选择 ADC 的时钟频率。

ADC 采样值存储位置有两个，一个是 ADC 数据缓存寄存器 ADC_DBRx ，另一个是 ADC 数据寄存器 ADC_DR。当 ADC 处于缓存连续或扫描模式时，采样的转换数据保存在数据缓存寄存器 ADC_DBRx 内;其他情形则保存在数据寄存器 ADC_DR 内。

3.6.2 ADC 电路

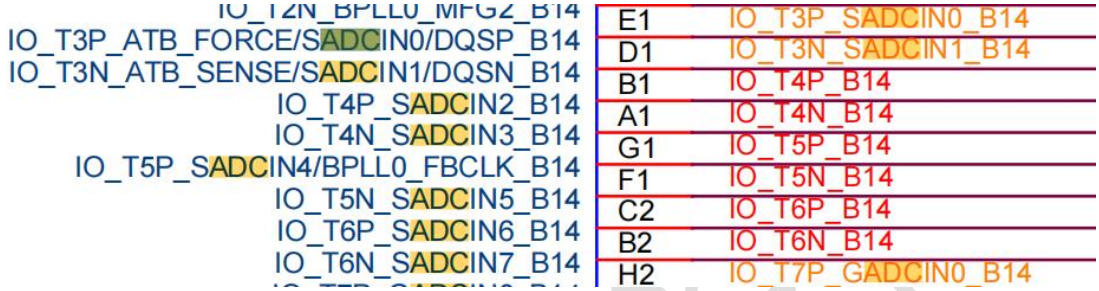


图 3-33 ADC 接口电路

3.6.3 引脚分配

本例程使用 UART0 打印采样数据，在串口例程基础上新增 ADC 通道约束，具体约束如下图所示。

```
phycst.start
#clk
phycst.pin.set   {CLK_IN}      J15       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#SW0
phycst.pin.set   {RST_N}       P19       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#swd-port
phycst.pin.set   {SWD_IO}      B18       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.pin.set   {SWD_CLK}     B17       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D1
phycst.pin.set   {FPGA_LED}    R19       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D2
phycst.pin.set   {LED2}        T21       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D3
phycst.pin.set   {LED3}        T20       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
######UART0######
#uart0-rx
phycst.pin.set   {UART0_RX}    G15       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#uart0-tx
phycst.pin.set   {UART0_TX}    G16       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#ADC(F484)
phycst.pin.set   {ADC_CH0}     E1        -attr "PULLMODE=NONE"
phycst.pin.set   {ADC_CH1}     B1        -attr "PULLMODE=NONE"
phycst.pin.set   {ADC_CH2}     D1        -attr "PULLMODE=NONE"
phycst.pin.set   {ADC_CH3}     A1        -attr "PULLMODE=NONE"
phycst.pin.set   {ADC_CH4}     G1        -attr "PULLMODE=NONE"
phycst.pin.set   {ADC_CH5}     C2        -attr "PULLMODE=NONE"
phycst.pin.set   {ADC_CH6}     F1        -attr "PULLMODE=NONE"
phycst.pin.set   {ADC_CH7}     B2        -attr "PULLMODE=NONE"
phycst.end
```

图 3-34 物理约束

3.6.4 代码设计

该例程设置 ADC 为连续转换、扫描模式、使能数据缓存的工作方式，并启用 ADC 中断，在 ADC 每次转换结束时，在中断响应程序中读取转换的值。

```
void ADC_adc_Init(ADC_TypeDef* ADCx)
{
    ADC_InitTypeDef  ADC_InitStruct;

    ADC_DeInit(ADCx);

    ADC_InitStruct.ADC_RefVol_Sel    = ADC_REF_VOL_INTERIOR; //参考电平选择
    ADC_InitStruct.ADC_PDn           = ADC_ACTIVE;           //ADC PowerDown Disable
    ADC_InitStruct.ADC_ClkPreDiv     = ADC_ClkPreDiv_4;      //时钟分频
    ADC_InitStruct.ADC_ConversionMode  = ADC_ConversionMode_Continuous;     //连续转换
    ADC_InitStruct.ADC_DataBufEnable   = ADC_DataBufEnable_Enable;          //数据缓存使能
    ADC_InitStruct.ADC_ExternalTrigConvEnable  = ADC_ExternalTrigConvEnable_Disable;外部触发关闭
    ADC_InitStruct.ADC_ExternalEventSel        = ADC_ExternalEventSel_Timer0;       //外部事件
    ADC_InitStruct.ADC_DataAlign        = ADC_DataAlign_Right;      //数据右对齐
    ADC_InitStruct.ADC_ScanModeEanbel   = ADC_ScanModeEanbel_Disable;启动扫描模式

    ADC_Init(ADCx,ADC_InitStruct);

    ADC_Interrupt_Init(ADCx);

    ADC_Channel_Select(ADCx,ADC_Channel_0); //选择ADC 通道

    ADC_Cmd(ADCx,ENABLE);   //使能ADC 并开始转换
}
```

图 3-35 ADC 初始化

上图为 ADC 初始化函数，结构体成员依次为参考电平选择、ADC 激活或掉电、时钟分频因子、转换模式（连续转换或单次转换）、数据缓存（数据缓存和连续转换同时启用，ADC 采样结果将存储在 ADC_DBRx 寄存器中）、外部触发使能、外部触发类型、数据对齐方式（左对齐，右对齐）、是否启用扫描模式。

```
void ADC_Interrupt_Init(ADC_TypeDef* ADCx)
{
    NVIC_ClearPendingIRQ(ADC_IRQn);
    NVIC_EnableIRQ(ADC_IRQn);
    ADC_EOCInterrupt_Config(ADCx, ENABLE);使能ADC 中断
}
```

图 3-36 ADC 中断初始化

```
void ADC_Handler(void)
{
    flag=1;
    /*ADC 转换值存储位置判断*/
    if(ADC_DataBufEnable_Get(ADCSel) )
    {
        ADC_GetValueFromDBR(ADCSel,ADC_buf); //从ADC_DBR 寄存器读取采样值
    }
    else
    {
        ADC_buf[0]=ADC_GetValueFromDR(ADCSel);
    }

    ADC_EOCFlag_Clear(ADCSel);

    if(ADC_DataBufOVRFlag_Get(ADCSel))
    {
        ADC_DataBufOVRFlag_Clear(ADCSel);
    }
}
```

图 3-37 ADC 中断服务函数

上图分别为 ADC 中断初始化函数和 ADC 的中断服务函数，在中断服务函数将 ADC 采集的值从 DBRx 寄存器中取出存放在 ADC_buf 中。

```
void ADC_printf(void)
{
```

```
    uint8_t i;
    while(flag==0);
    if(ADC_DataBufEnable_Get(ADCSel)==SET )
    {
        for(i=0;i<8;i++)
        {
            lower = ADC_buf[i];
            upper = ADC_buf[i]>16;
            printf("ADC_buf[%d]_upper=%x;ADC_buf[%d]_lower=%x.\r\n",i,upper,i,lower);
        }
        printf("\n");
    }
    else
    {
        printf("ADC_Value=%x.\r\n",ADC_buf[0]);
    }
    flag=0;
}
```

图 3-38 打印 ADC 转换的值

上图将 ADC_buf 中值打印出来。

```
void ADC_test(void)
{
    ADCSel = STAR_ADC0;
    ADC_adc_Init(ADCSel);

    while(1)
    {
        ADC_printf();
        Delay_ms(1000);
    }
}
```

图 3-39 ADC 测试函数

上图为 ADC 测试函数，主要通过调用相关函数初始化 ADC，然后在 ADC 转换完成后打印 ADC 转换数据。

下图为 main 函数的代码。

```
int main(void)
{
    GPIO_DeInit(STAR_GPIO0);
    SysTick_Init();

    /*LED GPIO 模式设置*/
    GPIO_Mode_Set(LED2_PORT,LED2_PIN,GPIO_Mode_Output);
    GPIO_Mode_Set(LED3_PORT,LED3_PIN,GPIO_Mode_Output);

    /*LED2 LED3 ON*/
    LED2_ON;
    LED3_ON;
    delay_ms(1000);

    /*LED2 LED3 OFF*/
    LED2_OFF;
    LED3_OFF;


    uart_init(115200);
    printf("STAR ADC Test...\r\n");

    ADC_test();

    while(1);
}
```

图 3-40 main 函数

3.6.5 下载验证

软件中设置采样通道 0 进行采样，连续转换模式，给开发板的 ADC 输入通道 0 探测点接入不同的电压，通过串口打印的 adc 采样值会有变化。



图 3-41 ADC 采样结果

3.7 定时器例程

本例程使用 STAR 的定时器 0 产生固定周期的定时器中断，在中断处理函数内控制 LED2 的亮灭，实现 LED2 固定周期的点亮和熄灭，以验证 STAR 的定时器功能。

3.7.1 定时器简介

STAR 具有 2 个定时器，均为 32 位向下计数器。当计数值为 0 时，可以产生一个中断请求信号。中断请求会被保持，直到写入 INTCLEAR 寄存器来清除。

CTRL	0x000	RW	4	0x0	[3] Timer interrupt enable. [2] Select external input as clock. [1] Select external input as enable. [0] Enable.

图 3-42 定时器 CTRL 寄存器

上图为定时器的控制寄存器，可以选择外部输入来使能或作为时钟，也可启动定时器工作和使能中断。

3.7.2 引脚分配

Timer 定时器外设没有额外需要连接的引脚，只需要约束时钟、复位和调试用的 LED 引脚即可，具体约束如下图所示。

```
phycst.start
#clk
phycst.pin.set   {CLK_IN}     J15      -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#SW4
phycst.pin.set   {RST_N}      P19      -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#swd-port
phycst.pin.set   {SWD_IO}     B18      -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
```

```
phycst.pin.set   {SWD_CLK}    B17       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D1
phycst.pin.set   {FPGA_LED}   R19       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D2
phycst.pin.set   {LED2}       T21       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D3
phycst.pin.set   {LED3}       T20       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.end
```

图 3-43 物理约束

3.7.3 代码设计

```
void timer_init(void)
{
    TIMER_TypeDef* timer;
    TIM_InitTypeDef TIM_InitStruct;

    timer = STAR_TIMER0;

    /*初始化设置*/
    TIM_DeInit(timer);
    TIM_InitStruct.TIM_InterruptEnable = TIM_Interrupt_Enable; //使能中断
    TIM_InitStruct.TIM_SelExtInputAsClk = TIM_Sel_PclkAsClk;
    TIM_InitStruct.TIM_SelExtInputAsEn = TIM_Sel_SoftSetAsEn;
    TIM_InitStruct.TIM_Value  = 25000000; //定时时间设为1秒 定时器时钟PCLK=25M，定时器装载值为
25000000
    TIM_InitStruct.TIM_Reload = 25000000;

    TIM_Init(timer, &TIM_InitStruct);

    /*中断设置*/
    if(timer ==STAR_TIMER0)
    {
        NVIC_ClearPendingIRQ(TIMER0_IRQn);
        NVIC_EnableIRQ(TIMER0_IRQn);
    }
    else
    {
        NVIC_ClearPendingIRQ(TIMER1_IRQn);
        NVIC_EnableIRQ(TIMER1_IRQn);
    }

    TIM_InterruptFlag_Clear(timer);

    TIM_Enable(timer,ENABLE);
}
```

图 3-44 定时器初始化

上图对定时器进行初始化设置，主要是设置定时器中断、时钟选择、使能选择、装载值和重装载值。本例程设置定时器中断周期为 1 秒，也就是每秒产生一次中断，由于定时器使用的时钟 PCLK 频率为 25MHz，因此定时器的装载值和重装载值都设置为 25000000。

```
void TIMER0_Handler(void)
{
    GPIO_TogglePin(LED2_PORT,LED2_PIN);
    /*清除中断*/
    TIM_InterruptFlag_Clear(STAR_TIMER0);
}
```

图 3-45 定时器中断服务函数

下图为定时器测试函数的代码。主要是调用相关函数完成定时器的初始化和 LED 引脚的初始化。

```
void Timer_Test(void)
{
```

```
    /*LED 指示灯引脚初始化*/
    GPIO_DeInit(STAR_GPIO0);
    GPIO_Mode_Set(LED2_PORT,LED2_PIN,GPIO_Mode_Output);
    GPIO_Mode_Set(LED3_PORT,LED3_PIN,GPIO_Mode_Output);

    LED2_OFF;
    LED3_OFF;

    /*定时器初始化*/
    timer_init();
}
```

图 3-46 定时器测试函数

下图为 main 函数代码。

```
int main(void)
{
    Timer_Test();
    while(1)
    {
        ;
    }
}
```

图 3-47 main 函数代码

3.7.4 下载验证

程序下载到板卡上，可以看到 LED2 会每隔 1 秒点亮/熄灭。

3.8 DUAL TIMER 例程

本例程使用 STAR 的 DUAL Timer 的定时器 1 和定时器 2 分别产生周期 1 秒和 2 秒的定时器中断，在中断处理函数内分别控制 LED D5 和 LED D7 的亮灭，以验证 STAR 的 DUAL Timer 定时器功能。

3.8.1 DUAL Timer 简介

STAR 有两个 DUAL Timer，该定时器由两个可编程的 32 位或 16 位向下计数器组成，当计数到 0 时可产生中断。

定时器有下列模式：

Free-running：计数器在到达 0 之后重新加载，并继续从最大值开始计数。

One-shot：计数器产生一次中断，当计数器到达 0 时会停止，直到被重新设置。

Periodic：计数器以固定的间隔生成一个中断，在超过零后重新加载原始值。

每一个定时器都有一组相同的寄存器，且操作相同。定时器通过写入 LOAD 寄存器来加载值，使能后一直计数到 0。当计数器正在运行的时候，写入 LOAD 寄存器会导致计数器在新加载值处立即重新启动，写入 BGLoad 加载值对当前计数没有影响，计数器会计数到 0，然后从新的加载值开始运行，前提是工作在周期模式。

当计数到 0 时，会产生一个中断，可以通过写 INTCLR 寄存器来清除中断，如果工作在 One-shot 模式，计数器在 0 时停止，直到被取消或重新启动。

当计数到 0 时，如果计数器在 Free-running 模式下运行，将继续从最大值开始计数。如果工作在 Periodic 模式，则计数器会从 LOAD 寄存器重新加载值并继续计数，在这种模式下，计数器会产生周期性的中断。

复位时，计数器被禁用，中断被清除，LOAD 寄存器的值被设 0，模式设为

Free-running，时钟分频和预分频设 1。

3.8.2 引脚分配

Dual Timer 的物理约束与 Timer 类似，都不需要约束额外的引脚即可使用，具体约束如下图所示。

```
phycst.start
#clk
phycst.pin.set   {CLK_IN}      J15       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#SW4
phycst.pin.set   {RST_N}       P19       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#swd-port
phycst.pin.set   {SWD_IO}      B18       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.pin.set   {SWD_CLK}     B17       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D1
phycst.pin.set   {FPGA_LED}    R19       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D2
phycst.pin.set   {LED2}        T21       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D3
phycst.pin.set   {LED3}        T20       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.end
```

图 3-48 物理约束

3.8.3 代码设计

该例程设置 DUAL Timer 工作在周期模式。

```
void DualTimer_Init(void)
{
    DTIM_InitTypeDef DTIM_InitStr;
    DUALTIMER_TypeDef* DUALTIMERx;

    DUALTIMERx = STAR_DUALTIMER0;

    DTIM_DeInit(DUALTIMERx);
    /*DualTimer Timer1 设置 */
    DTIM_InitStr.DTIM_Mode = DTIM_Mode_Periodic;                //周期模式
    DTIM_InitStr.DTIM_InterruptEnable = DTIM_Interrupt_Enable;  //中断使能
    DTIM_InitStr.DTIM_ClkDiv = DTIM_ClkDiv_1;                  //时钟 1 分频
    DTIM_InitStr.DTIM_CounterSize = DTIM_CounterSize_32bit;     //32 位计数器模式
    DTIM_InitStr.DTIM_LoadVal = 25000000;                       //Timer1 定时时间设置为 1 秒，PCLK
时钟为 25M，加载值设置为 25000000
    DTIM_Init(DUALTIMERx,DTIM_TIMER1,DTIM_InitStr);

    /*DualTimer Timer2 设置 */
    DTIM_InitStr.DTIM_LoadVal = 50000000;                       //Timer2 定时时间设置为 2 秒，PCLK
时钟为 25M，定时 2 秒 加载值设置为 50000000
    DTIM_Init(DUALTIMERx,DTIM_TIMER2,DTIM_InitStr);

    //DTIM_SetBGLoad(DUALTIMERx,DTIM_TIMER1,25000000);

    /*中断使能*/
    if(DUALTIMERx == STAR_DUALTIMER0)
    {
        NVIC_ClearPendingIRQ(DUALTIMER0_IRQn);
        NVIC_EnableIRQ(DUALTIMER0_IRQn);
    }
    else
    {
        NVIC_ClearPendingIRQ(DUALTIMER1_IRQn);
        NVIC_EnableIRQ(DUALTIMER1_IRQn);
    }

    DTIM_InterruptFlag_Clear(DUALTIMERx,DTIM_TIMER1);
    DTIM_InterruptFlag_Clear(DUALTIMERx,DTIM_TIMER2);

    /*使能 DualTimer，使其开始工作*/
    DTIM_Enable(DUALTIMERx,DTIM_TIMER1,ENABLE);
    DTIM_Enable(DUALTIMERx,DTIM_TIMER2,ENABLE);
```

```
}
```

图 3-49 DUAL Timer 配置

上图所示 DUAL Timer 的定时器 1 和定时器 2 都工作在周期工作模式，定时器 1 每 1 秒产生中断，定时器 2 每 2 秒产生中断。由于 APB 总线时钟频率为 25MHz，DUAL Timer 的分频系数设置为 1，因此定时器 1 和定时器 2 的加载值分别为 25000000 和 50000000。

DTIM_Init 函数用于完成 DUAL Timer 初始化，主要是设置工作模式、是否使能中断、时钟分频系数、计数器模式和加载值。DTIM_Enable 函数使能 DUAL Timer，使其开始工作。

下图所示为 DUAL Timer 的中断服务函数，在该函数内通过 DTIM_InterruptStatus_Get 函数来获取发生中断的定时器 1 和定时器 2。

```
void DUALTIMER0_Handler(void)
{
    if(DTIM_InterruptStatus_Get(STAR_DUALTIMER0,DTIM_TIMER1))
    {
        GPIO_TogglePin(LED2_PORT,LED2_PIN);
        DTIM_InterruptFlag_Clear(STAR_DUALTIMER0,DTIM_TIMER1);
    }

    if(DTIM_InterruptStatus_Get(STAR_DUALTIMER0,DTIM_TIMER2))
    {
        GPIO_TogglePin(LED3_PORT,LED3_PIN);
        DTIM_InterruptFlag_Clear(STAR_DUALTIMER0,DTIM_TIMER2);
    }
}
```

图 3-50DUAL Timer 中断服务函数

下图为 DUAL Timer 测试函数的代码。主要是调用相关函数完成 DUAL Timer 的初始化和 LED 引脚的初始化。

```
void DualTimer_Test(void)
{
    /*LED 指示灯引脚设置*/
    GPIO_DeInit(STAR_GPIO0);
    GPIO_Mode_Set(LED2_PORT,LED2_PIN,GPIO_Mode_Output);
    GPIO_Mode_Set(LED3_PORT,LED3_PIN,GPIO_Mode_Output);

    LED2_OFF;
    LED3_OFF;

    /*双定时器初始化*/
    DualTimer_Init();

    while(1);
}
```

图 3-51 DUAL Timer 测试函数

下图为 main 函数代码。

```
int main(void)
{
    DualTimer_Test();
    while(1)
    {
        ;
    }
}
```

图 3-52 main 函数

3.8.4 下载验证

下载程序后，可以看到 LED2 会每 1 秒点亮/熄灭，LED3 每 2 秒点亮/熄灭。

3.9 WatchDog 例程

本例程通过配置 STAR 的看门狗，在看门狗计数器递减至 0 之前进行'喂狗'（即重载计数器值）操作，以避免产生系统复位。如果将代码中的喂狗操作注释，则会产生系统复位。通过 LED 灯的亮灭情况可以
查看系统复位情况。

3.9.1 WatchDog 介绍

STAR 内部有一个独立看门狗，它的本质是一个递减计数器，若在计数器递减至 0 前重载计数值，则不会产生复位信号，这就是所谓的喂狗。

3.9.2 引脚分配

看门狗例程不需要约束额外的引脚，只需要对基本的时钟和复位信号和需要用到的 LED 的引脚进行约束即可，具体物理约束如下图所示。

```
phycst.start
#clk
phycst.pin.set   {CLK_IN}      J15        -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#SW4
phycst.pin.set   {RST_N}       P19        -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#swd-port
phycst.pin.set   {SWD_IO}      B18        -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.pin.set   {SWD_CLK}     B17        -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D1
phycst.pin.set   {FPGA_LED}        R19        -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D2
phycst.pin.set   {LED2}            T21        -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D3
phycst.pin.set   {LED3}            T20        -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.end
```

图 3-53 物理约束

3.9.3 代码设计

```
void watchdog_init(void)
{
    WatchDog_DeInit();

    WatchDog_Unlock();  //解锁看门狗
    WatchDog_Set_Counter(50000000); //设置看门狗加载值 设置为50000000(2s 内就要喂狗，PCLK=25M)
    WatchDog_Config(WDG_RST_Output_Enable,WDG_CNT_INT_Enable);  //使能开门狗复位输出和计数器中断
    WatchDog_Lock();    //锁定看门狗
}
```

图 3-54 Watchdog 初始化

上图为看门狗的初始化，先调用 WatchDog_Unlock 函数才能对其他所有的寄存器进行访问，WatchDog_Lock 函数则是关闭对其他寄存器的访问。看门狗初始化主要是将看门狗加载值设置为 50000000，也就是 2 秒内要进行喂狗操作；同时使能看门狗复位输出和智能看门狗。

```
void NMI_Handler(void)
{
    /*LED2 亮灭控制*/
    GPIO_TogglePin(STAR_GPIO0, GPIO_Pin_0);
    //WatchDog_Interrupt_Clear();   //清除中断---如果需要产生系统复位，则不要清除中断 在中断产生后会进行系统复位
}
```

图 3-55 Watchdog 中断服务函数

上图为看门狗中断的服务函数，在该函数内只翻转 LED D5 的状态，不能清除看门狗中断，否则无法系统复位。

下图为看门狗喂狗函数。

```
void watchdog_feed(uint32_t reload)
{
    WatchDog_Unlock();
    WatchDog_Set_Counter(reload); //重新设置 reload 值进行喂狗
    WatchDog_Lock();
}
```

图 3-56 Watchdog 喂狗函数

下图为看门狗测试函数，如果将喂狗操作注释，则在看门狗计数器值到 0 后产生系统复位。

```
void watchdog_test(void)
{
    uint32_t loadnum;
    watchdog_init();

    while(1)
    {
        loadnum=WatchDog_Get_CurrentCounter(); //获取看门狗当前计数值，如果计数值快要为 0 则重新喂狗，在看门狗计数值为 0 前要喂狗 否则会复位
        if(loadnum<500)
        {
            //watchdog_feed(50000000);  //喂狗－不喂狗会进行复位
        }
    }
}
```

图 3-57 Watchdog 测试函数

下图为 main 函数代码。

```
int main(void)
{
    /*LED 指示灯引脚设置*/
    GPIO_DeInit(STAR_GPIO0);
    GPIO_Mode_Set(LED2_PORT,LED2_PIN,GPIO_Mode_Output);
    GPIO_Mode_Set(LED3_PORT,LED3_PIN,GPIO_Mode_Output);

    /*Led 控制 便于观测复位情况*/
    LED2_ON;
    LED3_ON;
    delay_ms(1000);
    LED2_OFF;
    LED3_OFF;
    delay_ms(1000);

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

图 3-58 Watchdog main 函数

3.9.4 下载验证

程序下载后，可以看到 LED2 和 LED3 重复亮灭，也就是 STAR 一直在进行系统复位。

3.10 SPI 例程

本例程主要通过 STAR 的 SPI 读写开发板上的 Flash 芯片，以验证 STAR 的 SPI 通信功能，SPI 读写的数据通过串口打印，可以直观地看到测试情况。

3.10.1 SPI 简介

SPI 是一种高速的、全双工的、同步的通信总线。并且在芯片的引脚只占用四根线，节约了引脚资源。

STAR 有两个主模式的 SPI。SPI 通常有一个主设备和一个或多个从设备，需要至少四根线，支持 x1、x2 和 x4 模式。

CS：从设备使能信号，由主设备控制，CS 表示芯片是否被选中，只有芯片被选择才能进行操作，STAR 的片选无需额外控制，在 SPI 工作时自动拉低。

SCLK：时钟信号，主设备产生。

MISO：主机输入从机输出信号。

MOSI：主机输出从机输入信号。

SPI 有四种工作模式，主要由 CPOL 和 CPHA 共同决定，CPOL 表示 SPI 空闲时时钟的电平，CPHA 表示 SPI 工作时采样的时刻。

3.10.2 FLASH 电路图

下图为 FLASH 的电路图，采用的芯片为 GD25WQ16。

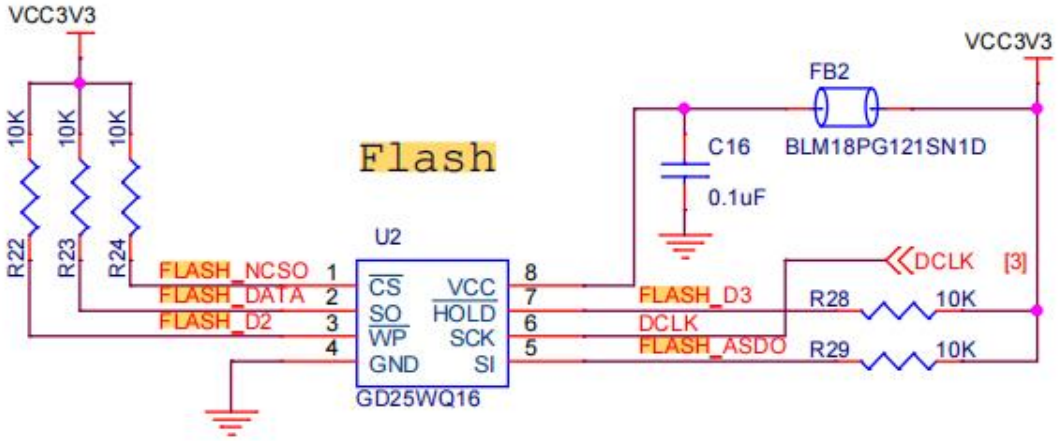


图 3-59 FLASH 电路

3.10.3 引脚分配

对于本例程的引脚分配，重点在于 SPI 的四根工作线，CLK、CS、MOSI、MISO，用于打印读写数据的 UART0 对应的引脚也需分配，具体物理约束如下图所示。

```
phycst.start
#clk
phycst.pin.set   {CLK_IN}      J15       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#SW4
phycst.pin.set   {RST_N}       P19       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#swd-port
phycst.pin.set   {SWD_IO}      B18       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.pin.set   {SWD_CLK}     B17       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D1
phycst.pin.set   {FPGA_LED}    R19       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D2
phycst.pin.set   {LED2}        T21       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D3
phycst.pin.set   {LED3}        T20       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
######UART0######
#uart0-rx
phycst.pin.set   {UART0_RX}    G15       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#uart0-tx
```

```
phycst.pin.set   {UART0_TX}   G16        -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
######SPI0,FLASH######
phycst.pin.set   {SPI0_SCK_OUT}   L12 -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.pin.set   {SPI0_SEL_OUT}   T19 -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.pin.set   {SPI0_DATA[0]}   P22 -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.pin.set   {SPI0_DATA[1]}   R22 -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.pin.set   {SPI0_DATA[2]}   P21 -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.pin.set   {SPI0_DATA[3]}   R21 -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.end
```

图 3-60 物理约束

3.10.4 代码设计

3.10.4.1 自环模式测试

自环测试：SPI 可设置工作在自环模式下，在该模式下，发送 fifo 的数据会直接传入接收 fifo，可测试 SPI 是否正常工作。

```
void SPI_LoopBack_Test(SPI_TypeDef* spix)
{
    SPI_InitTypeDef  SPI_InitStructure;
    uint8_t data1,data2,data3,data4;

    SPI_PinInit(spix);

    SPI_DeInit(spix);

    /*SPI 初始化配置*/
    SPI_InitStructure.SPI_ClkPreDiv = SPI_CLK_12Prescale;              //时钟预分频
    SPI_InitStructure.SPI_SCR = 0;                                     //时钟速率因子
    SPI_InitStructure.SPI_Mode = SPI_Mode_Master;                      //主从模式
    SPI_InitStructure.SPI_DataSize = SPI_Data_Size_8bit;               //数据位数
    SPI_InitStructure.SPI_LoopBackMode = SPI_LoopBackMode_Enable;      //自环模式
    SPI_InitStructure.SPI_ClkPolarityPhase = SPI_CLK_PL_Low_PH_1Edge;  //工作模式
    SPI_InitStructure.SPI_TxRxTransmitMode = SPI_TxRxTransmit_Simultaneous; //收发方式
    SPI_InitStructure.SPI_DataWidth = SPI_DataWidth_Standard;          //数据宽度
    SPI_Init(spix,SPI_InitStructure);

    /*通过 SPI 发送测试数据*/
    data1=0x89;
    data2=0x34;
    printf("SPI_LoopBack_Send:dat1=0x%x,dat2=0x%x\r\n",data1,data2);
    SPI_SendData(spix, data1);
    SPI_SendData(spix, data2);
    SPI_Cmd(spix,ENABLE);

    data3 = SPI_ReadByte(spix);
    data4 = SPI_ReadByte(spix);

    SPI_Cmd(spix,DISABLE);
    printf("SPI_LoopBack_Recv:dat1=0x%x,dat2=0x%x\r\n",data3,data4);
}
```

图 3-61 SPI 自环模式

上图是一个 SPI 自环模式的配置函数，在 SPI 的结构体变量中需要设置时钟预分频参数、数据位数、工作模式、是否自环、数据宽度、收发方式（分时收发或同时收发，在自环模式下采用全双工收发）。

3.10.4.2 读取 ID 序列

在使用读取 ID 指令读取 FlashID 前需要对 SPI 进行初始化，如下图所示。

```
void Flash_SPI_Init(void)
{
    SPI_InitTypeDef  SPI_InitStructure;

    SPI_PinInit(flashSPI);
```

```
    SPI_DeInit(flashSPI);

    /*SPI 初始化配置*/
    SPI_InitStructure.SPI_ClkPreDiv = SPI_CLK_12Prescale;              //时钟预分频
    SPI_InitStructure.SPI_SCR = 0;                                     //时钟速率因子
    SPI_InitStructure.SPI_Mode = SPI_Mode_Master;                      //主从模式
    SPI_InitStructure.SPI_DataSize = SPI_Data_Size_8bit;               //数据位数
    SPI_InitStructure.SPI_LoopBackMode = SPI_LoopBackMode_Disable;     //是否自环
    SPI_InitStructure.SPI_ClkPolarityPhase = SPI_CLK_PL_Low_PH_1Edge;      //工作模式
    SPI_InitStructure.SPI_TxRxTransmitMode = SPI_TxRxTransmit_TimeSharing;  //收发方式
    SPI_InitStructure.SPI_DataWidth = SPI_DataWidth_Standard;              //数据宽度
    SPI_Init(flashSPI,SPI_InitStructure);
}
```

图 3-62 SPI 初始化

SPI 对应的引脚是复用普通 GPIO 的，也需要对这些引脚进行初始化配置，如下图所示。

```
void SPI_PinInit(SPI_TypeDef* spix)
{
    if(spix ==STAR_SPI0)
    {
        GPIO_AF_Config( STAR_GPIO0, GPIO_AF_SPI0_CLK_OUT, ENABLE); //GPIO[7]  CLK_OUT
        GPIO_AF_Config( STAR_GPIO0, GPIO_AF_SPI0_SEL,     ENABLE); //GPIO[8]   CS_n
        GPIO_AF_Config( STAR_GPIO0, GPIO_AF_SPI0_DATA0,   ENABLE); //GPIO[9]   MOSI
        GPIO_AF_Config( STAR_GPIO0, GPIO_AF_SPI0_DATA1,   ENABLE); //GPIO[10]  MISO
        GPIO_AF_Config( STAR_GPIO0, GPIO_AF_SPI0_DATA2,   ENABLE); //GPIO[17]  WPn
        GPIO_AF_Config( STAR_GPIO0, GPIO_AF_SPI0_DATA3,   ENABLE); //GPIO[18]  HOLDn
    }
    else    //SPI1
    {
        GPIO_AF_Config( STAR_GPIO0, GPIO_AF_SPI1_CLK_OUT, ENABLE); //GPIO[11]  CLK_OUT
        GPIO_AF_Config( STAR_GPIO0, GPIO_AF_SPI1_SEL,     ENABLE); //GPIO[12]  CS_n
        GPIO_AF_Config( STAR_GPIO0, GPIO_AF_SPI1_DATA0,   ENABLE); //GPIO[13]  MOSI
        GPIO_AF_Config( STAR_GPIO0, GPIO_AF_SPI1_DATA1,   ENABLE); //GPIO[14]  MISO
        GPIO_AF_Config( STAR_GPIO0, GPIO_AF_SPI1_DATA2,   ENABLE); //GPIO[19]  WPn
        GPIO_AF_Config( STAR_GPIO0, GPIO_AF_SPI1_DATA3,   ENABLE); //GPIO[20]  HOLDn
    }
}
```

图 3-63 SPI 引脚初始化

读取开发板上 FLASH 芯片的 ID 函数如下图所示。

```
void SPI_Read_Flash_ID(void)
{
    uint8_t MID,ID1,ID2;

    //Flash_SPI_Init();

    SPI_ReceiveDataNum_Config(flashSPI, 0x0000003);
    SPI_WriteByte(flashSPI, WB_W25Q_READ_ID);
    SPI_Cmd(flashSPI,ENABLE);

    MID = SPI_ReadByte(flashSPI);
    ID1 = SPI_ReadByte(flashSPI);
    ID2 = SPI_ReadByte(flashSPI);
    SPI_Cmd(flashSPI,DISABLE);

    printf("FlashID:MID:0x%x ,ID:0x%x%x\r\n",MID,ID1,ID2);
}
```

图 3-64 读取 FLASH ID

3.10.4.3 读写 FLASH

FLASH 的数据读写：

在 flash_spi.c 文件中定义了和 FLASH 通信相关的函数，例如扇区擦除、整个芯片擦除、页写、页读等。

在读写 Flash 测试函数中，先对 FLASH 的一个扇区进行擦除。然后再调用页写函数写入一段数据。最后调用页读函数读取写入的数据，将写入和读出的数据通过串口

打印。

```
void STAR_FLASH_SectorErase(uint32_t startAddr)
{
    Flash_WriterEnable();
    Flash_SectorErase(startAddr);
    Flash_WaitBusyEnd();
}
```

图 3-65 扇区擦除函数

```
void STAR_FLASH_PageWrite(uint8_t* txBuf,uint32_t addr,uint16_t bufSize)
{
    Flash_WriterEnable();
    Flash_PageWrite(txBuf, addr, bufSize);
    Flash_WaitBusyEnd();
}
```

图 3-66 页写函数

```
void STAR_FLASH_PageRead(uint8_t* rxBuf,uint32_t addr,uint16_t num)
{
    int i=0;

    SPI_ReceiveDataNum_Config(flashSPI, num);
    SPI_SendData(flashSPI, WB_W25Q_READ);              //Read Data Order  0x03  x2:0x3b  x4:0x6b
    SPI_SendData(flashSPI, (addr & 0xFF0000)>16);  //Read Data Addr:high
    SPI_SendData(flashSPI, (addr & 0xFF00)>8);     //Read Data Addr:middle
    SPI_SendData(flashSPI, (addr & 0xFF));          //Read Data Addr:low

    SPI_Cmd(flashSPI, ENABLE);
    SPI_WaitSendFinish(flashSPI);

    for(i=0;i<num;i++)
    {
        *rxBuf = SPI_ReadByte(flashSPI);
        rxBuf ++;
    }
    SPI_Cmd(flashSPI, DISABLE);
}
```

图 3-67 页读函数

```
void SPI_Flash_Test(void)
{
    uint32_t testAddr =0x0;

    SPI_LoopBack_Test(STAR_SPI0);

    Flash_SPI_Init();
    /*读取 FlashID*/
    SPI_Read_Flash_ID();

    /*Flash 数据读写*/
    /*Sector 擦除*/
    STAR_FLASH_SectorErase(testAddr);
    printf ("STAR_FLASH_SectorErase,addr=0x%x\r\n",testAddr);
    /*按页写入数据*/
    STAR_FLASH_PageWrite(TxBuffer, testAddr,BufferSize);
    printf ("STAR_FLASH_PageWrite,addr=0x%x,write_data:%s\r\n",testAddr,TxBuffer);
    /*按页读取数据*/
    memset(RxBuffer,0x0,sizeof(RxBuffer));
    STAR_FLASH_PageRead(RxBuffer,testAddr,BufferSize);
    printf ("STAR_FLASH_PageRead,addr=0x%x,read_data:%s\r\n",testAddr, RxBuffer);
}
```

图 3-68 Flash 读写测试函数

3.10.5 下载验证

程序下载后，在串口调试助手可以看到通过 SPI 对 Flash 进行擦除、写入数据和读出数据的过程。通过对比写入的数据和读出的数据，可以看到写入的数据和读出的数据是一致的，说明 STAR 的 SPI 通信功能正常。



图 3-69 Flash 读写测试打印信息

3.11 AHB Master 例程

STAR IP 内部集成了两组 AHB 总线从端接口和两组扩展通道（可配置为 AHB、APB、AXI-full 和 AXI-lite 接口），通过这些接口可以使 FPGA 侧与 STAR 侧实现数据交互。在实际使用中，需要进行分布式开发，其一在 FPGA 中利用 Verilog 语言进行接口信号解析；其二使用 MDK5 工具对接口对应的寄存器地址进行读写操作。本例程 STAR 使用一组 AHB 总线主端接口与 FPGA 侧进行数据交互。

本例程通过 STAR 读写 FPGA 侧的 RAM 存储区，以实现 STAR 与 FPGA 的数据交互。本例程中 FPGA 侧的 RAM 作为从机挂接到 AHB 总线上，STAR 作为主机对该 RAM 空间进行读写操作，读写的数据通过 UART0 打印出来，可以直观地看到测试情况。

3.11.1 AHB 简介

AHB 总线的组成

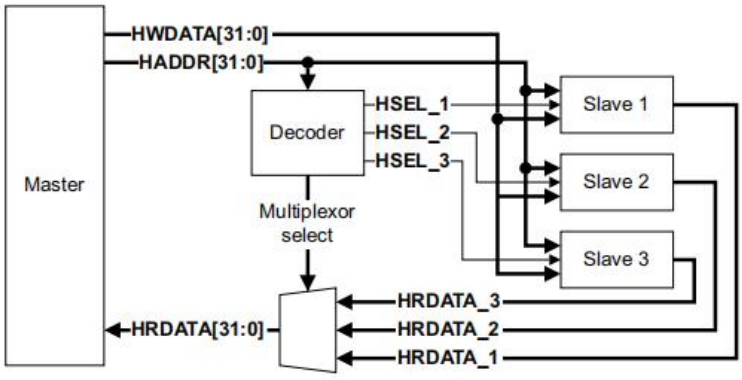


图 3-70 AHB 总线框图

AHB 信号描述

表 3—1 对 AHB 协议内部信号进行简单介绍。

表 3-1 AHB 协议内部信号

名称	来源	终点	描述

HCLK	时钟源		为所有总线传输提供基准频率。所有信号时序都和 HCLK 的上升沿相关。
HRESETn	复位控制器		总线复位信号，低电平有效，用于复位系统和总线。这是唯一低电平有效的信号。
HADDR[31:0]	主机	主机和译码器	32 位地址总线。
HBURST[2:0]	主机	从机	表示传输是否组成了突发的一部分。 000-SINGLE-单一传输 001-INCR-未指定长度的增量突发 010-WARP4-4 拍回环突发 011-INCR4-4 拍增量突发 100-WRAP8-8 拍回环突发 101-INCR8-8 拍增量突发 110-WRAP16-16 拍回环突发 111-INCR16-16 拍增量突发
HMASTLOCK	主机	从机	表示当前主机正在执行一个锁定顺序的传输。该信号和 HMaster 有相同的时序。
HPROT[3:0]	主机	从机	提供总线访问的附加信息，主要是给那些希望执行某种保护级别的模块使用的。这个信号指示当前传输是否为预取指令或者数据传输，同时也表示传输是保护模式访问还是用户模式访问。对带存储器管理单元的总线主机而言这些信号也用来指示当前传输是高速缓存的（cache）还是缓冲的（buffer）。 ---0：预取值 ---1：数据访问 --0-：用户模式访问 --1-：特权模式访问 -0--：无缓冲 -1--：带缓冲 0---：无高速缓冲 1---：带高速缓冲
HSIZE[2:0]	主机	从机	表示传输数据的比特位大小。 000:8bit 001:16bit 010:32bit 011:64bit 100:128bit 101:256bit 110:512bit 111:1024bit
HTRANS[1:0]	主机	从机	表示当前传输类型。

			00: IDLE 01: BUSY 10.NONSEQ 11: SEQ
HWDATA[31:0]	主机	从机	写数据总线用来在写操作期间将数据从主机传输到从机。建议最小的数据总线宽度为 32 位。
HWRITE	主机	从机	表示传输方向，高电平表示写传输，低电平表示读传输
HRDATA[31:0]	从机	多路选择器	在读操作期间
HREADYOUT	从机	多路选择器	高电平表示传输已完成，低电平表示可进行下一次传输
HRESP	从机	多路选择器	传输响应给传输状态提供了附加信息。提供四种不同的响应： OKEY、 ERROR、 RETRY 和 SPLIT。
HSELx	译码器	从机	每个从机都有自己独立的从机选择信号并且该信号可以表示当前传输的是否被选中的从机。
HRDATA[31:0]	多路选择器	主机	读数据总线，通过译码器选择通道。
HREADY	多路选择器	主机和从机	该信号为高电平时，表示主机和所有的从机传输已完成。
HRESP	多路选择器	主机	转换响应，通过译码器选择通道。

AHB 基本传输时序分析

下两图为没有等待信号的 single transfer。

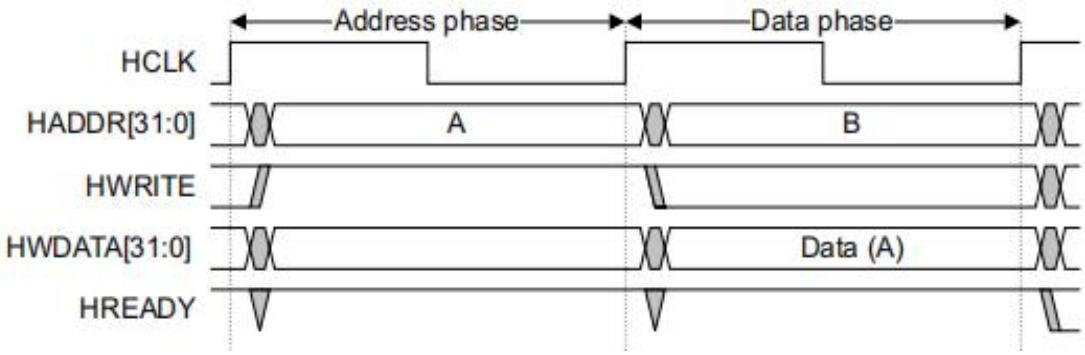


图 3-71 AHB 写时序

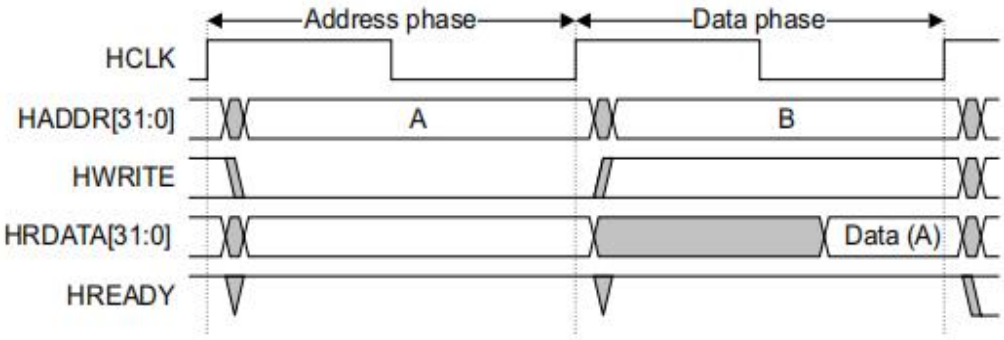


图 3-72 AHB 读时序

系统框图

如下图所示，FPGA 逻辑侧 AHB Slave 模块包含两个主要模块 AHB 从机接口模块和 RAM 模块，AHB 从机模块包含两个子模块从机接口模块和从机寄存器模块，分别实现 AHB 传输时序描述和寄存器的读写，RAM 模块主要实现数据存储。

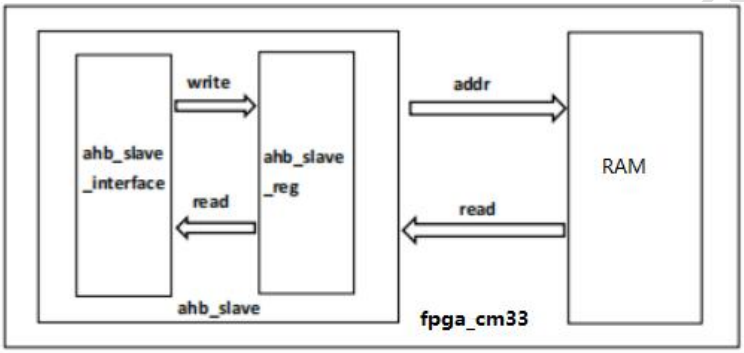


图 3-73 AHB 例程框架图

3.11.2 引脚分配

AHB 属于 STAR 内部总线，因此无需对其进行引脚约束，该例程只需要对复位、时钟等基本信号和用于打印读写数据的 UART0 的引脚进行约束即可，具体物理约束如下图所示。

```
phycst.start
#clk
phycst.pin.set   {CLK_IN}      J15      -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#SW0
phycst.pin.set   {RST_N}       P19      -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#swd-port
phycst.pin.set   {SWD_IO}      B18      -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.pin.set   {SWD_CLK}     B17      -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D1
phycst.pin.set   {FPGA_LED}    R19      -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D2
phycst.pin.set   {LED2}        T21      -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D3
phycst.pin.set   {LED3}        T20      -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
######UART0######
#uart0-rx
phycst.pin.set   {UART0_RX}    G15      -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#uart0-tx
phycst.pin.set   {UART0_TX}    G16      -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.end
```

图 3-74 物理约束

3.11.3 代码设计

3.11.3.1 FPGA 侧代码设计

下图中的 fpga_cm33.v 是 FPGA 的顶层文件，cmsdk_ahb_eg_slave.v 包含 cmsdk_ahb_eg_slave_interface.v 和 cmsdk_ahb_eg_slave_reg.v 两个模块，它们的功能分别是实现 AHB 传输时序描述和寄存器的数据读写，xsIP_EBR_PDP.v 是 RAM IP，用于存储 STAR 发送过来的数据。

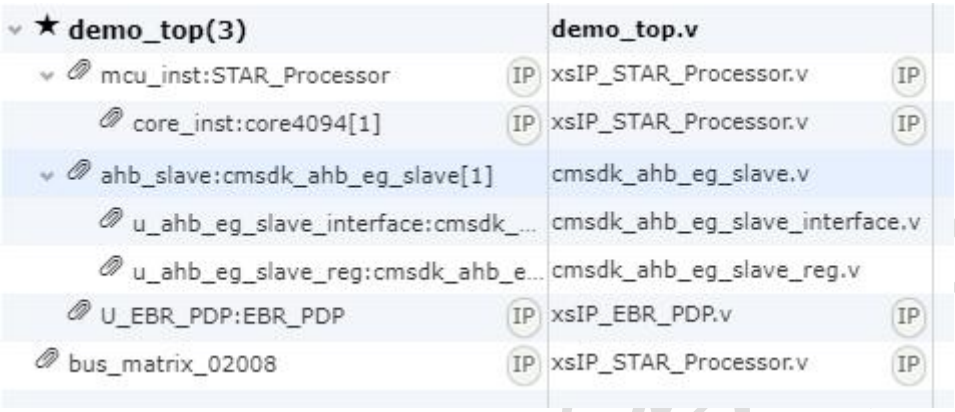


图 3-75 FPGA 侧源码文件

3.11.3.2 STAR 侧代码设计

STAR 侧有两个 AHB 主端拓展口，其基地址如下：

0x6000_0000	0x9FFF_FFFF	AHB Expansion0	1GB	TARGEXP0	connected to FPGA fabric	-
0xA000_0000	0xDFFF_FFFF	AHB Expansion1	1GB	TARGEXP1	connected to FPGA fabric	-

图 3-76 AHB 主端接口基地址

本例程中使用 AHB 拓展口 0，其基地址为 0x60000000。

读写操作的地址寄存器和数据寄存器相对基地址的偏移如下图所示；偏移量由 FPGA 侧的寄存器定义决定。

```
#define WR_ADDR_OFFSET  0x04   //写操作地址寄存器偏移
#define WR_DATA_OFFSET  0x08   //写操作数据寄存器偏移
#define RD_ADDR_OFFSET 0x0C   //读操作地址寄存器偏移
#define RD_DATA_OFFSET 0x10   //读操作数据寄存器偏移
#define FPGA_AHB_SLAVE_ID_OFFSET 0xFFC //FPGA AHB SLAVE ID 偏移
```

图 3-77 AHB 读写寄存器操作地址偏移

以上地址偏移与 FPGA 侧对寄存器偏移地址的相关解析是对应的，以下是逻辑侧的部分代码，具体参考逻辑侧 cmsdk_ahb_eg_slave_reg.v 的描述。

```
// Address decoding for write operations
assign wr_sel[0] = ((addr[11:2]==10'b0000000000)&(write_en)) ? 1'b1: 1'b0;0xA0000000
assign wr_sel[1] = ((addr[11:2]==10'b0000000001)&(write_en)) ? 1'b1: 1'b0;0xA0000004
assign wr_sel[2] = ((addr[11:2]==10'b0000000010)&(write_en)) ? 1'b1: 1'b0;0xA0000008
assign wr_sel[3] = ((addr[11:2]==10'b0000000011)&(write_en)) ? 1'b1: 1'b0;0xA000000c
assign wr_sel[4] = ((addr[11:2]==10'b0000000100)&(write_en)) ? 1'b1: 1'b0;0xA0000010
...
begin
    if (addr[11:5] == 8'h00) begin
      case(addr[4:2])
        3'b000: rdata =  regControl;   //0xA0000000
        3'b001: rdata =  regWAddress;  //0xA0000004
        3'b010: rdata =  regWData;     //0xA0000008
```

```
        3'b011: rdata =  regRAddress;   //0xA000000c
           3'b100: rdata =  regRData;     //0xA0000010
        default: rdata = {32{1'bx}};
      endcase
end
```

图 3-78 FGPA 侧寄存器读写解析

写操作流程

写操作，以往 FPGA RAM 的 0x10 地址写入数据 0xAE 为例：

写操作地址寄存器写入需要写的 RAM 地址，即 0x60000004 写入 0x10；

然后写操作数据寄存器写入需要写入的数据，即 0x60000008 写入 0xAE;

完成上述操作即实现数据写入。

读操作流程

读操作，以读 FPGA RAM 的 0x10 地址的数据为例：

读操作地址寄存器写入需要读取的 RAM 地址，即 0x6000000C 写入 0x10；

然后读取读操作数据寄存器的数据；即读取 0x60000010 的数据，读回的数据即为 RAM 内地址为 0x10 的数据；

AHB 总线读写接口函数如下图所示。

```
uint32_t ahbslave_read(uint32_t address)
{
    uint32_t *data;
    data = (unsigned int *)address;
    return *data;
}
```

图 3-79 AHB 总线读操作函数

```
void ahbslave_write(uint32_t *address, uint32_t data)
{
    *address = data;
}
```

图 3-80 AHB 总线写操作函数

本例程实验流程为，通过 AHB 往 FPGA RAM 内写入一段数据，然后回读回来，并把写入的地址、数据和回读的地址数据通过串口打印出来，用以确认回读的数据是否是写入的数据。具体实现代码如下图所示。

认回读的数据是否是写入的数据。具体实现代码如下图所示。

```
void AHB_readWrite_test(void)
{
    int id_ok = 0;
    int i = 0;
    uint32_t rData = 0;
    uint32_t rwNum= 512;

    //读取AHB Slave ID
    printf("read ahb slave0...\r\n");
    id_ok = ahbslave_id_check();
    if (id_ok == 0)
    {
        printf("ahb slave ID ok.\r\n");
    }
    else
    {
        printf("ahb slave ID error.\r\n");
    }

    //往FPGA RAM 写入数据
    for (i = 0; i < rwNum; i++)
    {
        ahbslave_write((uint32_t *)(CH0_M00_BASE_ADDR + WR_ADDR_OFFSET), i);
```

```
        ahbslave_write((uint32_t *)(CH0_M00_BASE_ADDR + WR_DATA_OFFSET), i);
        printf("Write:wAddress:%d, wData: %d\r\n", i, i);
    }
    delay_ms_soft(10);
    //将写入的数据从 FPGA RAM 回读
    for (i = 0; i < rwNum; i++)
    {
        ahbslave_write((uint32_t *)(CH0_M00_BASE_ADDR + RD_ADDR_OFFSET), i);
        rData = ahbslave_read(CH0_M00_BASE_ADDR + RD_DATA_OFFSET);
        printf("Read:rAddress:%d, rData: %d\r\n", i, rData);
    }
}
```

图 3-81 AHB 总线读写测试代码

3.11.4 下载验证

下载程序后在串口调试助手可以看到通过 AHB 总线读写的数据信息，如下图所示。

```
STAR AHB test...
read ahb slave0...
data: 0xb1;
ahb slave ID ok.
Write:wAddress:0, wData: 0
Write:wAddress:1, wData: 1
Write:wAddress:2, wData: 2
Write:wAddress:3, wData: 3
Write:wAddress:4, wData: 4
Write:wAddress:5, wData: 5
Write:wAddress:6, wData: 6
Write:wAddress:7, wData: 7
Write:wAddress:8, wData: 8
Write:wAddress:9, wData: 9
Write:wAddress:10, wData: 10
Write:wAddress:11, wData: 11
Write:wAddress:12, wData: 12
Write:wAddress:13, wData: 13
Write:wAddress:14, wData: 14
Write:wAddress:15, wData: 15
Write:wAddress:16, wData: 16
Write:wAddress:17, wData: 17
Write:wAddress:18, wData: 18
Write:wAddress:19, wData: 19
Write:wAddress:20, wData: 20
Write:wAddress:21, wData: 21
```

图 3-82 AHB 总线读写数据信息

3.12 DMA 例程

本例程分别验证 DMA 的内存到内存、外设到内存的数据传输功能。

3.12.1 DMA 简介

DMA（直接存储器存取）是一个独立的外设控制器。主要功能是用于数据搬运，但不占用 CPU。

STAR 内部有一组 DMA 可供使用。

3.12.2 引脚分配

本例程中使用 SPI 作为测试使用的外设，使用 UART0 来打印测试信息，需对这两个外设的引脚进行约束，如下图所示。

```
phycst.start
#clk
phycst.pin.set    {CLK_IN}      J15       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#SW4
phycst.pin.set    {RST_N}       P19       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#swd-port
```

```
phycst.pin.set   {SWD_IO}      B18       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.pin.set   {SWD_CLK}     B17       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D1
phycst.pin.set   {FPGA_LED}    R19       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D2
phycst.pin.set   {LED2}        T21       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D3
phycst.pin.set   {LED3}        T20       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
######UART0######
#uart0-rx
phycst.pin.set   {UART0_RX}    G15       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#uart0-tx
phycst.pin.set   {UART0_TX}    G16       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
######SPI0,FLASH######
phycst.pin.set   {SPI0_SCK_OUT}   L12 -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.pin.set   {SPI0_SEL_OUT}   T19 -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.pin.set   {SPI0_DATA[0]}   P22 -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.pin.set   {SPI0_DATA[1]}   R22 -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.pin.set   {SPI0_DATA[2]}   P21 -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.pin.set   {SPI0_DATA[3]}   R21 -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.end
```

图 3-83 物理约束

3.12.3 代码设计

3.12.3.1 内存到内存

DMA 内存到内存传输的测试代码如下图所示，通过 DMA 将 testDataBuf 的数据搬运到 testDataBuf2。testDataBuf 和 testDataBuf2 是代码中定义的大小为 32 字节的数组。配置 DMA 时将传输类型设置为 MtoM_DMA，同时开启 DMA 传输完成中断。

```
void DMA_Test_MtoM(void)
{
    int i;
    uint8_t w_data=0x22;
    uint8_t sendNum =10;
    DMA_InitTypedef DMA_InitStr;

    memset((char*)&testDataBuf,0x0,sizeof(testDataBuf));        //Source Buffer
    memset((char*)&testDataBuf2,0x0,sizeof(testDataBuf2));      //Destination  Buffer

    printf("---------------DMA Memory-To-Memory Test---------------\r\n");
    printf("Source_Data:");
    for(i=0;i<sendNum;i++)
    {
        testDataBuf[i]= w_data+i;
        printf("0x%x ",testDataBuf[i]);
    }
    printf("\r\n");
    printf("Start DMA transfer...\r\n");

    DMA_InitStr.DMA_Source_Addr = (uint32_t)&testDataBuf;
    DMA_InitStr.DMA_Destination_Addr = (uint32_t)&testDataBuf2;
    DMA_InitStr.DMA_SourceAddr_Inc = DMA_Addr_Inc_Enable;
    DMA_InitStr.DMA_DestinationAddr_Inc = DMA_Addr_Inc_Enable;
    DMA_InitStr.DMA_SourceDataWidth = DMA_DataWidth_Byte;
    DMA_InitStr.DMA_DestinationDataWidth = DMA_DataWidth_Byte;
    DMA_InitStr.DMA_SourceBurstSize = DMA_BurstSize_1;
    DMA_InitStr.DMA_DestinationBurstSize = DMA_BurstSize_1;
    DMA_InitStr.DMA_Transfertype = MtoM_DMA;                        //DMA 传输类型 内存到内存
    DMA_InitStr.DMA_DestinationPeripheral = DMA_Peripheral_Unused;
    DMA_InitStr.DMA_SourcePeripheral = DMA_Peripheral_Unused;
    DMA_InitStr.DMA_TCInterruptEnable = DMA_TC_INT_Enable;          //使能 DMA 传输完成中断


    DMA_Init(STAR_DMA,DMA_Channel_0,DMA_InitStr);

    DMA_TransferSize_Config(STAR_DMA,DMA_Channel_0,sendNum);

    DMA_Interrupt_Config(STAR_DMA,DMA_Channel_0,DMA_INT_TC,ENABLE);
```

```
    DMA_Channel_Enable(STAR_DMA,DMA_Channel_0,ENABLE);

    printf("Destination_Data: ");

    /*读取收到的数据*/
    for(i=0;i<sendNum;i++)
    {
        printf ("0x%x ", testDataBuf2[i]);

    }
    printf("\r\n");

    DMA_Channel_Enable(STAR_DMA,DMA_Channel_0,DISABLE);
}
```

图 3-84 DMA 内存到内存测试函数

下图所示为 DMA 传输完成中断初始化函数和对应的中断服务函数。

```
void DMA_interrupt_init(void)
{
    /*使能DMA 终端计数中断 用于DMA 的发送完成中断*/
    NVIC_ClearPendingIRQ(DMACINTTC_IRQn);
     NVIC_EnableIRQ(DMACINTTC_IRQn);
}
```

图 3-85 DMA 传输完成中断初始化

```
void DMACINTTC_Handler(void)
{
    ITStatus state;

    state=DMA_Get_TCInterruptStatus(STAR_DMA,DMA_Channel_0);
    if(state==SET) //channel0
    {
        DMA_Interrupt_Clear(STAR_DMA,DMA_Channel_0,DMA_INT_TC);
    }

    state=DMA_Get_TCInterruptStatus(STAR_DMA,DMA_Channel_1);
    if(state==SET)  //channel1
    {
        DMA_Interrupt_Clear(STAR_DMA,DMA_Channel_1,DMA_INT_TC);
    }

    printf("DMA transfer complete!\r\n");
}
```

图 3-86 DMA 传输完成中断服务函数

3.12.3.2 外设到内存

将 SPI0 的接收端数据搬运到数组 testDataBuf 内，testDataBuf 是代码中定义的大小为 32 字节的数组。

下图为 DMA 外设到内存数据传输的测试代码。

```
void DMA_SPI_Test_PtoM(void)
{
    int i;
    uint8_t sendNum =23;
    DMA_InitTypedef DMA_InitStr;

    memset((char*)&testDataBuf,0x0,sizeof(testDataBuf));

    printf("---------------DMA Peripheral-To-Memory Test---------------\r\n");

    DMA_InitStr.DMA_Source_Addr = 0x4000B008;   //SPI0 Data Reg Addr
    DMA_InitStr.DMA_Destination_Addr = (uint32_t)&testDataBuf;
    DMA_InitStr.DMA_SourceAddr_Inc = DMA_Addr_Inc_Disable;
    DMA_InitStr.DMA_DestinationAddr_Inc = DMA_Addr_Inc_Enable;
    DMA_InitStr.DMA_SourceDataWidth = DMA_DataWidth_Byte;
    DMA_InitStr.DMA_DestinationDataWidth = DMA_DataWidth_Byte;
    DMA_InitStr.DMA_SourceBurstSize = DMA_BurstSize_1;
    DMA_InitStr.DMA_DestinationBurstSize = DMA_BurstSize_1;
    DMA_InitStr.DMA_Transfertype = PtoM_DMA;                    //DMA 传输类型 外设到内存
```

```
    DMA_InitStr.DMA_DestinationPeripheral = DMA_Peripheral_Unused;
    DMA_InitStr.DMA_SourcePeripheral = DMA_Peripheral_SPI0RX;
    DMA_InitStr.DMA_TCInterruptEnable = DMA_TC_INT_Enable;      //使能DMA 传输完成中断

    DMA_Init(STAR_DMA,DMA_Channel_0,DMA_InitStr);
    DMA_TransferSize_Config(STAR_DMA,DMA_Channel_0,sendNum);
    DMA_Interrupt_Config(STAR_DMA,DMA_Channel_0,DMA_INT_TC,ENABLE);
    DMA_Channel_Enable(STAR_DMA,DMA_Channel_0,ENABLE);

    printf("Start DMA transfer...\r\n");

    /*SPI 初始化并发送数据*/
    SPI_Init_And_Send();

    printf("Destination_Data: ");
    /*读取SPI 发送的数据*/
    for(i=0;i<sendNum;i++)
    {
        printf ("0x%x ", testDataBuf[i]);
    }
    printf("\r\n");

    DMA_Channel_Enable(STAR_DMA,DMA_Channel_0,DISABLE);
}
```

图 3-87 DMA 外设到内存测试函数

SPI0 的初始化和数据发送代码如下图所示，为了方便测试将 SPI0 设置为自环模式。

```
static void SPI_Init_And_Send(void)
{
    uint8_t i;
    uint8_t dat=0xa1;
    uint8_t sendNum = 23;

    SPI_InitTypeDef  SPI_InitStructure;
    uint8_t data1,data2,data3,data4;

    SPI_PinInit(SPIx);

    SPI_DeInit(SPIx);

    /*SPI 初始化配置*/
    SPI_InitStructure.SPI_ClkPreDiv = SPI_CLK_12Prescale;              //时钟预分频
    SPI_InitStructure.SPI_SCR = 0;                                     //时钟速率因子
    SPI_InitStructure.SPI_Mode = SPI_Mode_Master;                      //主从模式
    SPI_InitStructure.SPI_DataSize = SPI_Data_Size_8bit;               //数据位数
    SPI_InitStructure.SPI_LoopBackMode = SPI_LoopBackMode_Enable;      //自环模式
    SPI_InitStructure.SPI_ClkPolarityPhase = SPI_CLK_PL_Low_PH_1Edge;  //工作模式
    SPI_InitStructure.SPI_TxRxTransmitMode = SPI_TxRxTransmit_Simultaneous; //收发方式
    SPI_InitStructure.SPI_DataWidth = SPI_DataWidth_Standard;          //数据宽度
    SPI_Init(SPIx,SPI_InitStructure);

    /*使能SPI 接收DMA*/
    SPI_DMA_Cmd(SPIx,SPI_DMA_RX_ENABLE,ENABLE);

    /*SPI 发送数据*/
    printf("SPI send data:");

    SPI_SendData(SPIx, dat);    //先往SPI 发送FIFO 写第一个数据 然后再开启SPI 进行发送
    SPI_Cmd(SPIx, ENABLE);
    printf("0x%x ",dat);

    for(i=1;i<sendNum;i++)
    {
        SPI_SendData(SPIx, dat+i);
        printf("0x%x ",dat+i);
    }
    printf("\r\n");
    SPI_WaitSendFinish(SPIx);
}
```

图 3-88 SPI 初始化和数据发送函数

下图为 DMA 内存到内存和外设到内存的测试代码。

```
void DMA_test(void)
{
    DMA_interrupt_init(); //DMA 中断配置
    DMA_Test_MtoM();
    DMA_SPI_Test_PtoM();
    DMA_SPI_Test_MtoP();
}
```

图 3-89 DMA 测试函数

3.12.4 下载验证

下载程序后在串口调试助手可以看到 DMA 内存到内存和外设到内存的测试信息，如下图所示。

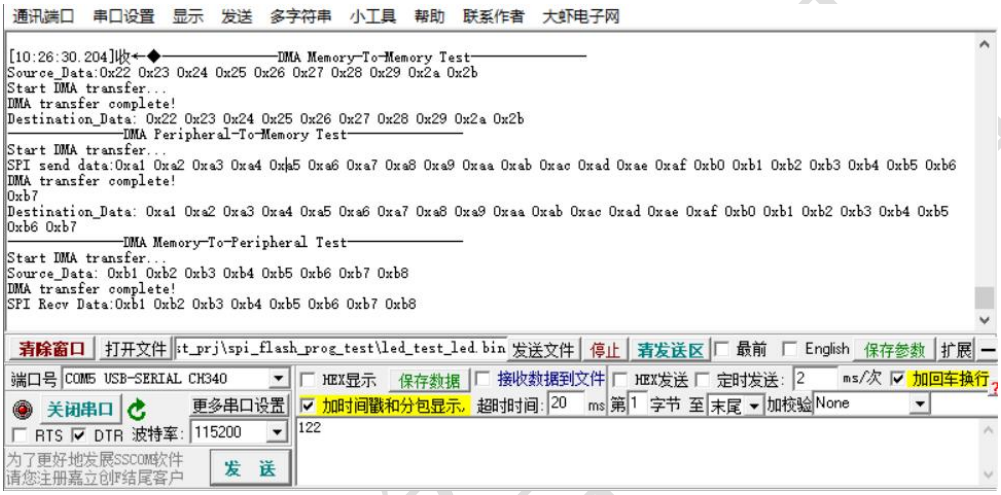


图 3-90 DMA 测试串口打印信息

3.13 DMA-AHB 突发例程

3.13.1 DMA-AHB 简介

STAR 内部的 DMA 和 AHB 功能在上文均已介绍，并进行了相关实验。本次实验将使用 AHB 总线的 burst 模式进行数据的读写，通过 STAR 侧的 DMA 来触发 AHB 总线在 burst 模式下写数据，并将数据写入 FPGA 侧的 FIFO 然后再通过 burst 模式读出，数据最终由串口进行打印，具体系统框图如下图所示。

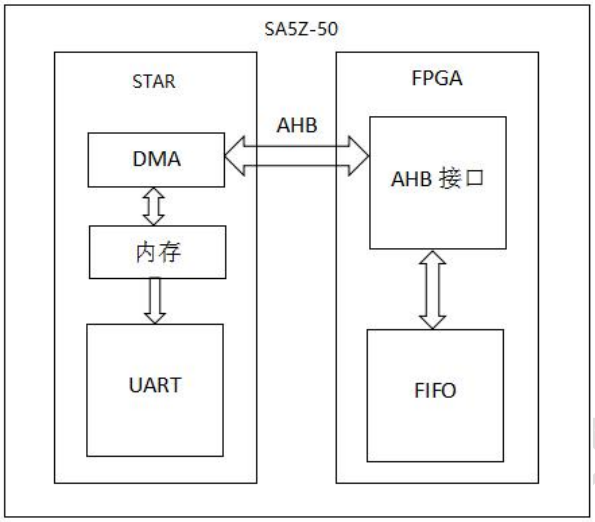


图 3-91 DMA-AHB 例程框图

3.13.2 引脚分配

本例程不需要约束额外的引脚，只需要对基本的时钟、复位信号和打印测试信息用到的 UART0 的引脚进行约束即可，具体物理约束如下图所示。

```
phycst.start
#clk
phycst.pin.set   {CLK_IN}      J15       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#SW0
phycst.pin.set   {RST_N}       P19       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#swd-port
phycst.pin.set   {SWD_IO}      B18       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.pin.set   {SWD_CLK}     B17       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D1
phycst.pin.set   {FPGA_LED}    R19       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D2
phycst.pin.set   {LED2}        T21       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D3
phycst.pin.set   {LED3}        T20       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
######UART0######
#uart0-rx
phycst.pin.set   {UART0_RX}    G15       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#uart0-tx
phycst.pin.set   {UART0_TX}    G16       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.end
```

图 3-92 物理约束

3.13.3 代码设计

通过 AHB 突发往 FIFO 写数据代码如下图所示。

```
void DMA_BURST_WRITE_AHB(void)
{
    int i;
    uint32_t  w_data=0xA0B0C0D0;
    DMA_InitTypedef DMA_InitStr;

    memset((char*)&wrBuffArr,0x0,sizeof(wrBuffArr));
    TrFinishFlag =0;

    DMA_interrupt_init();

    /*初始化需要写的数据*/
    for(i=0;i<wr_data_num;i++)
    {
        wrBuffArr[i] = w_data+i;
        printf("write data_%d to AHB by DMA is: 0x%x\r\n",i,wrBuffArr[i]);
    }
```

```
    /*DMA 初始化配置*/
    DMA_InitStr.DMA_Source_Addr = (uint32_t)&wrBuffArr;
    DMA_InitStr.DMA_Destination_Addr = 0x60000000;          //AHB 总线基地址
    DMA_InitStr.DMA_SourceAddr_Inc = DMA_Addr_Inc_Enable;
    DMA_InitStr.DMA_DestinationAddr_Inc = DMA_Addr_Inc_Enable;
    DMA_InitStr.DMA_SourceDataWidth = DMA_DataWidth_Word;
    DMA_InitStr.DMA_DestinationDataWidth = DMA_DataWidth_Word;
    DMA_InitStr.DMA_SourceBurstSize = DMA_BurstSize_4;
    DMA_InitStr.DMA_DestinationBurstSize = DMA_BurstSize_4;
    DMA_InitStr.DMA_Transfertype = MtoM_DMA;                        //DMA 传输类型 内存到内存
    DMA_InitStr.DMA_DestinationPeripheral = DMA_Peripheral_Unused;
    DMA_InitStr.DMA_SourcePeripheral = DMA_Peripheral_Unused;
    DMA_InitStr.DMA_TCInterruptEnable = DMA_TC_INT_Enable;          //DMA 传输完成中断

    DMA_Init(STAR_DMA,DMA_Channel_0,DMA_InitStr);

    DMA_TransferSize_Config(STAR_DMA,DMA_Channel_0,wr_data_num);
    DMA_Interrupt_Config(STAR_DMA,DMA_Channel_0,DMA_INT_TC,ENABLE);
    DMA_Channel_Enable(STAR_DMA,DMA_Channel_0,ENABLE);      //使能对应通道的DMA 开始传输

    /*等待DMA 传输完成*/
    while(TrFinishFlag==0);

    DMA_Channel_Enable(STAR_DMA,DMA_Channel_0,DISABLE);
}
```

图 3-93 DMA-AHB 数据写入

上图是通过对 DMA 的配置来触发 AHB 的 burst 模式，首先是源地址和起始地址的设定，设置传输数据以字为单位，突发大小设置为 4，待传输数据量 wr_data_num 设置为 32。此设置会使得 AHB 总线进行 8 次 4 拍增量突发，待传输数据量最好设置为突发大小的整数倍。此处只对与 AHB 总线传输相关的函数进行了说明，其余函数介绍见 DMA 实验。

FPGA 侧主要对 AHB 的数据进行解析，在检测到数据来到之后拉高 FIFO 的写使能将数据写入，具体代码见 cmsdk_ahb_eg_slave_interface 模块。

STAR 控制 DMA 触发 AHB 进行突发读操作代码如下图所示，和突发写操作设置基本类似，只是源地址和目的地址进行了调换。

```
void DMA_BURST_READ_AHB(void)
{
    int i;
    DMA_InitTypedef DMA_InitStr;

    TrFinishFlag=0;
    memset((char*)&rdBuffArr,0x0,sizeof(rdBuffArr));

    DMA_interrupt_init();

    /*DMA 初始化配置*/
    DMA_InitStr.DMA_Source_Addr = 0x60000000;                   //AHB 总线基地址
    DMA_InitStr.DMA_Destination_Addr = (uint32_t)&rdBuffArr;
    DMA_InitStr.DMA_SourceAddr_Inc = DMA_Addr_Inc_Enable;
    DMA_InitStr.DMA_DestinationAddr_Inc = DMA_Addr_Inc_Enable;
    DMA_InitStr.DMA_SourceDataWidth = DMA_DataWidth_Word;
    DMA_InitStr.DMA_DestinationDataWidth = DMA_DataWidth_Word;
    DMA_InitStr.DMA_SourceBurstSize = DMA_BurstSize_4;
    DMA_InitStr.DMA_DestinationBurstSize = DMA_BurstSize_4;
    DMA_InitStr.DMA_Transfertype = MtoM_DMA;                        //DMA 传输类型 内存到内存
    DMA_InitStr.DMA_DestinationPeripheral = DMA_Peripheral_Unused;
    DMA_InitStr.DMA_SourcePeripheral = DMA_Peripheral_Unused;
    DMA_InitStr.DMA_TCInterruptEnable = DMA_TC_INT_Enable;          //DMA 传输完成中断

    DMA_Init(STAR_DMA,DMA_Channel_0,DMA_InitStr);

    DMA_TransferSize_Config(STAR_DMA,DMA_Channel_0,wr_data_num);
    DMA_Interrupt_Config(STAR_DMA,DMA_Channel_0,DMA_INT_TC,ENABLE);
    DMA_Channel_Enable(STAR_DMA,DMA_Channel_0,ENABLE);          //使能对应通道的DMA 开始传输

    /*等待DMA 传输完成*/
```

```
    while(TrFinishFlag==0);

    /*打印读取到的数据*/
    for(i=0;i<wr_data_num;i++)
    {
        printf("read data_%d from AHB by DMA is :0x%x\r\n",i,rdBuffArr[i]);

    }
}
```

图 3-94 DMA-AHB 数据读出

main 函数代码如下图所示，主要是完成 UART0 初始化和调用 DMA-AHB 数据写入和数据读出函数来测试 DMA-AHB 突发读写功能。

```
int main(void)
{
    GPIO_DeInit(STAR_GPIO0);

    /*LED GPIO 模式设置*/
    GPIO_Mode_Set(LED2_PORT,LED2_PIN,GPIO_Mode_Output);
    GPIO_Mode_Set(LED3_PORT,LED3_PIN,GPIO_Mode_Output);

    /*LED2 LED3 ON*/
    LED2_ON;
    LED3_ON;
    delay_ms(1000);

    /*LED2 LED3 OFF*/
    LED2_OFF;
    LED3_OFF;



    uart_init(115200);
    DMA_BURST_WRITE_AHB();
    delay_ms(200);
    DMA_BURST_READ_AHB();
    while(1)
    {
        delay_ms(100);
    }
}
```

图 3-95 main 函数

3.13.4 下载验证

程序下载后，在串口调试助手可以看到通过 DMA-AHB 写入和读出的数据信息，如下图所示。

```
write data_0 to AHB by DMA is: 0xa0b0c0d0
write data_1 to AHB by DMA is: 0xa0b0c0d1
write data_2 to AHB by DMA is: 0xa0b0c0d2
write data_3 to AHB by DMA is: 0xa0b0c0d3
write data_4 to AHB by DMA is: 0xa0b0c0d4
write data_5 to AHB by DMA is: 0xa0b0c0d5
write data_6 to AHB by DMA is: 0xa0b0c0d6
write data_7 to AHB by DMA is: 0xa0b0c0d7
write data_8 to AHB by DMA is: 0xa0b0c0d8
write data_9 to AHB by DMA is: 0xa0b0c0d9
write data_10 to AHB by DMA is: 0xa0b0c0da
write data_11 to AHB by DMA is: 0xa0b0c0db
write data_12 to AHB by DMA is: 0xa0b0c0dc
write data_13 to AHB by DMA is: 0xa0b0c0dd
write data_14 to AHB by DMA is: 0xa0b0c0de
write data_15 to AHB by DMA is: 0xa0b0c0df
write data_16 to AHB by DMA is: 0xa0b0c0e0
write data_17 to AHB by DMA is: 0xa0b0c0e1
write data_18 to AHB by DMA is: 0xa0b0c0e2
write data_19 to AHB by DMA is: 0xa0b0c0e3
write data_20 to AHB by DMA is: 0xa0b0c0e4
write data_21 to AHB by DMA is: 0xa0b0c0e5
write data_22 to AHB by DMA is: 0xa0b0c0e6
write data_23 to AHB by DMA is: 0xa0b0c0e7
write data_24 to AHB by DMA is: 0xa0b0c0e8
write data_25 to AHB by DMA is: 0xa0b0c0e9
write data_26 to AHB by DMA is: 0xa0b0c0ea
write data_27 to AHB by DMA is: 0xa0b0c0eb
write data_28 to AHB by DMA is: 0xa0b0c0ec
write data_29 to AHB by DMA is: 0xa0b0c0ed
write data_30 to AHB by DMA is: 0xa0b0c0ee
write data_31 to AHB by DMA is: 0xa0b0c0ef
---------- DMA Transfer Complete----------
```

```
---------- DMA Transfer Complete----------
read data_0 from AHB by DMA is :0xa0b0c0d0
read data_1 from AHB by DMA is :0xa0b0c0d1
read data_2 from AHB by DMA is :0xa0b0c0d2
read data_3 from AHB by DMA is :0xa0b0c0d3
read data_4 from AHB by DMA is :0xa0b0c0d4
read data_5 from AHB by DMA is :0xa0b0c0d5
read data_6 from AHB by DMA is :0xa0b0c0d6
read data_7 from AHB by DMA is :0xa0b0c0d7
read data_8 from AHB by DMA is :0xa0b0c0d8
read data_9 from AHB by DMA is :0xa0b0c0d9
read data_10 from AHB by DMA is :0xa0b0c0da
read data_11 from AHB by DMA is :0xa0b0c0db
read data_12 from AHB by DMA is :0xa0b0c0dc
read data_13 from AHB by DMA is :0xa0b0c0dd
read data_14 from AHB by DMA is :0xa0b0c0de
read data_15 from AHB by DMA is :0xa0b0c0df
read data_16 from AHB by DMA is :0xa0b0c0e0
read data_17 from AHB by DMA is :0xa0b0c0e1
read data_18 from AHB by DMA is :0xa0b0c0e2
read data_19 from AHB by DMA is :0xa0b0c0e3
read data_20 from AHB by DMA is :0xa0b0c0e4
read data_21 from AHB by DMA is :0xa0b0c0e5
read data_22 from AHB by DMA is :0xa0b0c0e6
read data_23 from AHB by DMA is :0xa0b0c0e7
read data_24 from AHB by DMA is :0xa0b0c0e8
read data_25 from AHB by DMA is :0xa0b0c0e9
read data_26 from AHB by DMA is :0xa0b0c0ea
read data_27 from AHB by DMA is :0xa0b0c0eb
read data_28 from AHB by DMA is :0xa0b0c0ec
read data_29 from AHB by DMA is :0xa0b0c0ed
read data_30 from AHB by DMA is :0xa0b0c0ee
read data_31 from AHB by DMA is :0xa0b0c0ef
run...
run...
```

图 3-96 DMA-AHB 串口打印信息

3.14 提高 STAR 系统时钟频率例程

STAR 的系统时钟可以在 FPGA 侧进行修改，之前的例程 STAR 的时钟直接来自外部 25MHz 晶振。为了提高 STAR 的系统时钟，可以在 FPGA 侧使用 PLL 将外部输入时钟进行倍频，将倍频后的时钟作为 STAR 的系统时钟。由于 STAR 的 UART 可设置的最大波特率与系统时钟有关，本例程将 UART0 波特率设为开发板硬件可支持的最大值，并在此速率下进行串口收发测试。

本例程将 STAR 的系统时钟设置为 200MHz，UART0 的波特率设置为 2000000bps（开发板 usb 转串口芯片能支持的最大波特率为 2Mbps）进行验证测试。

3.14.1 串口接口电路

开发板上有一个 USB 转串口接口，本例程使用该接口进行串口的收发测试。

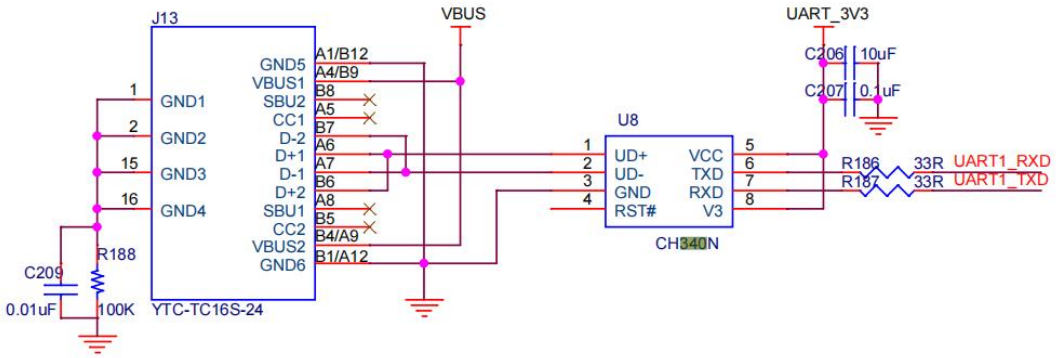


图 3-97 串口接口电路

3.14.2 引脚分配

STAR 的 UART0 收发引脚复用在 GPIO2 和 GPIO3 上，关于 GPIO 引脚复用的详细介绍

请参考《UG05205_SA5Z-50 ARM STAR User Guide_V1.0》2.4.2 章节的内容。本例程物理约束如下图所示。

```
phycst.start
#clk
phycst.pin.set   {CLK_IN}      J15       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#SW0
phycst.pin.set   {RST_N}       P19       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#swd-port
phycst.pin.set   {SWD_IO}      B18       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.pin.set   {SWD_CLK}     B17       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D1
phycst.pin.set   {FPGA_LED}    R19       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D2
phycst.pin.set   {LED2}        T21       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D3
phycst.pin.set   {LED3}        T20       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
######UART0######
#uart0-rx
phycst.pin.set   {UART0_RX}    G15       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#uart0-tx
phycst.pin.set   {UART0_TX}    G16       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.end
```

图 3-98 UART 物理约束

3.14.3 代码设计

3.14.3.1 FPGA 侧代码设计

在 FPGA 工程中，使用 PLL 将 25MHz 的晶振输入倍频到 200MHz 后作为 STAR 的系统时钟。



图 3-99 PLL 设置

STAR 的系统时钟源还需要改为 PLL 的输出，具体如下图所示。

```
43  wire clk_200;
44  PLL_FREQ PLL_FREQ_inst(
45      .CLKI(CLK_IN),
46      .CLKOP(clk_200)
47  );
48
49  //core instance
50  STAR_Processor mcu_inst(
51      .CLK_IN             (clk_200    ),
52      .SWD_CLK            (SWD_CLK    ),
53      .SWD_IO             (SWD_IO     ),
54      .GPIO0              (LED2       ),
55      .GPIO1              (LED3       ),
56      .UART0_RX           (UART0_RX   ),
57      .UART0_TX           (UART0_TX   ),
58      .RST_N              (RST_N      )
59  );
```

图 3-100 STAR 系统时钟源修改

3.14.3.2 STAR 侧代码设计

系统时钟变量 SystemCoreClock 在 system_ARMCM33.c 内定义，需要做对应修改，如下图所示，将 XTAL 由原来的 50000000 改为 400000000。

```
/*----------------------------------------------------------------------------
  Define clocks
 *----------------------------------------------------------------------------*/
#define  XTAL            (400000000UL)     /* Oscillator frequency */

#define  SYSTEM_CLOCK    (XTAL / 2U)
/*----------------------------------------------------------------------------
  System Core Clock Variable
 *----------------------------------------------------------------------------*/
uint32_t SystemCoreClock = SYSTEM_CLOCK;  /* System Core Clock Frequency */
```

图 3-101 STAR 系统时钟频率

该例程首先对串口进行初始化，然后打印例程信息字符串，然后等待用户发送数据，在 uart 接收中断中将用户发送的数据发送给用户，实现 echo 的效果。

UART0 的初始化配置如下图所示，主要是对 UART0 的收发引脚所在的 GPIO 进行引脚复用，然后对串口进行初始化配置，设置好波特率并使能发送和接收。

```
static void uartx_init(UART_TypeDef *UARTx,uint32_t BaudRate)
{
    UART_InitTypeDef  UART_InitStructure;

    UART_DeInit(UARTx);

    /*将对应的GPIO 引脚复用为对应UART 的收发引脚*/
    if(UARTx ==STAR_UART0)
    {
        GPIO_AF_Config(STAR_GPIO0, GPIO_AF_USART0_RXD, ENABLE); //GPIO[2]
        GPIO_AF_Config(STAR_GPIO0, GPIO_AF_USART0_TXD, ENABLE); //GPIO[3]
    }
    else if(UARTx ==STAR_UART1)
    {
        GPIO_AF_Config(STAR_GPIO0, GPIO_AF_USART1_RXD, ENABLE); //GPIO[4]
        GPIO_AF_Config(STAR_GPIO0, GPIO_AF_USART1_TXD, ENABLE); //GPIO[5]
    }
    else
```

```
    {
        GPIO_AF_Config(STAR_GPIO0, GPIO_AF_USART2_RXD, ENABLE); //GPIO[22]
        GPIO_AF_Config(STAR_GPIO0, GPIO_AF_USART2_TXD, ENABLE); //GPIO[23]
    }

    UART_InitStructure.UART_BaudRate = BaudRate;
    /*收发使能*/
    UART_InitStructure.UART_CTRL =  UART_CTRL_TxEnable | UART_CTRL_RxEnable;
    UART_Init( UARTx, &UART_InitStructure);

    /*接收中断配置*/
    uart_interrupt_init(UARTx);
}

void uart_init(uint32_t BaudRate)
{
    UART_TypeDef *UARTx;

    UARTx = STAR_UART0;
    uartx_init(UARTx,BaudRate);
}
```

图 3-102 UART0 初始化

在初始化 UART0 时，还需开启 UART0 的接收中断，中断配置如下图所示。

```
static void uart_interrupt_init(UART_TypeDef *UARTx)
{
    /*使能接收中断*/
    UART_Interrupt_Config(UARTx, UART_RxInterrupt, ENABLE);

    if(UARTx ==STAR_UART0)
    {
        NVIC_ClearPendingIRQ(UART0_IRQn);
        NVIC_EnableIRQ(UART0_IRQn);
    }
    else if(UARTx ==STAR_UART1)
    {
        NVIC_ClearPendingIRQ(UART1_IRQn);
        NVIC_EnableIRQ(UART1_IRQn);
    }
    else
    {
        NVIC_ClearPendingIRQ(UART2_IRQn);
        NVIC_EnableIRQ(UART2_IRQn);
    }
}
```

图 3-103 UART0 中断配置

主函数中将 UART0 波特率设置为 2000000，初始化完成后，等待用户发送数据；在 UART0 中断处理函数中将用户发送的数据发送给用户实现 echo 的效果。

```
int main(void)
{
    GPIO_DeInit(STAR_GPIO0);

    /*LED GPIO 模式设置*/
    GPIO_Mode_Set(LED2_PORT,LED2_PIN,GPIO_Mode_Output);
    GPIO_Mode_Set(LED3_PORT,LED3_PIN,GPIO_Mode_Output);

    /*LED2 LED3 ON*/
    LED2_ON;
    LED3_ON;
    delay_ms(1000);

    /*LED2 LED3 OFF*/
    LED2_OFF;
    LED3_OFF;


    uart_init(2000000);
    printf("STAR uart0 echo test...\r\n");
```

```
    while(1)
    {
        ;主函数循环不做其他业务，等待用户通过 uart 发送数据，echo 在接收中断中实现
    }
}
```

图 3-104 主函数用户代码

```
void UART0_Handler(void)
{
    uint8_t ch;

    if(UART_Get_Interrupt_Status(STAR_UART0,UART_RxInterrupt)==SET) //接收中断
    {
        /*读取接收到的数据*/
        ch=UART_ReceiveData(STAR_UART0);
        /*将接收到的数据发送回去*/
        UART_SendData(STAR_UART0,ch);
        /*清除中断*/
        UART_Interrupt_Clear(STAR_UART0, UART_RxInterrupt);
    }
}
```

图 3-105 UART0 中断服务函数

3.14.4 下载验证

程序下载后，打开串口助手，波特率设置为 2000000，在发送窗口发送任意字符，在接收窗口都可收到刚发送的字符数据。



图 3-106 串口调试助手收发数据

3.15 EXTINT 例程（来自 FPGA 侧的外部中断例程）

STAR 除了通过 GPIO 实现外部中断，还可以在 FPGA 侧来触发 EXTINT 外部中断。EXTINT 中断源有 16 个，分别是 EXTINT-0 到 EXTINT-15。EXTINT 中断只支持高电平触发。

External Interrupt Ports			
EXTINT	input	16	Interrupts from external (FPGA fabic)

图 3-107 EXTINT 接口说明

本例程在 FPGA 侧使用开发板上的按键 SW8 来触发 STAR 的 EXTINT0 中断。按键 SW1

经去抖后，每释放一次便触发一次中断，STAR 收到 EXTINT0 中断后，会翻转 LED2 的亮灭状态并通过串口打印中断信息。

3.15.1 引脚分配

本例程需要分配 LED 引脚和 UART0 引脚及系统运行需要的时钟、复位引脚。具体物理约束如下图所示。

```
phycst.start
#clk
phycst.pin.set   {CLK_IN}      J15       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#SW0
phycst.pin.set   {RST_N}       P19       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#swd-port
phycst.pin.set   {SWD_IO}      B18       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.pin.set   {SWD_CLK}     B17       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D1
phycst.pin.set   {FPGA_LED}    R19       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D2
phycst.pin.set   {LED2}        T21       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D3
phycst.pin.set   {LED3}        T20       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#SW1
phycst.pin.set {SWITCH1}       P20       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
######UART0######
#uart0-rx
phycst.pin.set   {UART0_RX}    G15       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#uart0-tx
phycst.pin.set   {UART0_TX}    G16       -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.end
```

图 3-108 物理约束

3.15.2 代码设计

3.15.2.1 FPGA 侧代码设计

FPGA 侧主要是在按键 SW1 按下并去抖动后给 STAR 的 EXTINT0 中断输入源高电平，进而触发 STAR 的 EXTINT0 中断。

```
key_debounce debounce_inst(
        .clk(CLK_IN),
        .rst_n(RST_N),
        .key_in(SWITCH1),
        .key_pulse(EXTINT0)  // 按键按下是触发中断
    );
```

图 3-109 FPGA 侧按键处理代码

3.15.2.2 STAR 侧代码设计

STAR 侧主要是 EXTINT0 的初始化和中断服务程序的处理。

EXTINT0 的初始化如下图所示，先用 NVIC_ClearPendingIRQ 清除 EXT0_IRQn 之前可能挂起的中断，再设置 EXT0_IRQn 的中断优先级，最后用 NVIC_EnableIRQ 使能 EXT0_IRQn 中断。

```
void extint0_init(void)
{
    NVIC_InitTypeDef nvic_init_t;

    NVIC_ClearPendingIRQ(EXT0_IRQn);    //clear NVIC pending interrupts

    /*中断优先级设置*/
    nvic_init_t.NVIC_IRQChannel = EXT0_IRQn;
```

```
    nvic_init_t.NVIC_IRQChannelPreemptionPriority = 0;  //抢占优先级
    nvic_init_t.NVIC_IRQChannelSubPriority = 0; //响应优先级
    nvic_init_t.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(nvic_init_t);

    NVIC_EnableIRQ(EXT0_IRQn);
}
```

图 3-110 EXTINT0 初始化

EXTINT0 的中断服务程序如下图所示；进入中断程序后，先翻转 LED D5 的状态，然后打印中断信息，再清除中断。

```
void EXT0_Handler(void)
{
    /*点亮/熄灭 LED2*/
    GPIO_TogglePin(LED2_PORT, LED2_PIN);
    printf("EXT0_Handler\r\n");
    /*清除中断标志*/
    NVIC_ClearPendingIRQ(EXT0_IRQn);
}
```

图 3-111 EXTINT0 中断服务函数

外设相关的初始化代码如下图所示。

```
void mcu_bsp(void)
{
    GPIO_DeInit(STAR_GPIO0);

    /*LED GPIO 模式设置*/
    GPIO_Mode_Set(LED2_PORT,LED2_PIN,GPIO_Mode_Output);
    GPIO_Mode_Set(LED3_PORT,LED3_PIN,GPIO_Mode_Output);

    /*LED2 LED3 OFF*/
    LED2_OFF;
    LED3_OFF;


    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);
    uart0_init(115200);
    extint0_init();
}
```

图 3-112 外设相关初始化函数

主函数代码如下图所示。

```
int main(void)
{
    mcu_bsp();
    printf("STAR EXTINT test...\r\n");
    while(1)
    {
        ;
    }
}
```

图 3-113 主函数代码

3.15.3 下载验证

程序下载后，按下按键 SW1 可以看到 LED2 点亮和熄灭状态翻转，在串口调试助手可以看到打印的中断信息。



图 3-114 打印的中断信息

3.16 APB Master 例程

本例程配置 MCU 侧为 APB Master 接口，接 Slave 模块，MCU 侧实现 APB 总线读写。

3.16.1 APB 简介

APB 主要用在低速且低功率的外围，可针对外围设备做功率消耗及复杂接口的最佳化，在 IP 配置中可通过总线配置界面，勾选扩展通道，配置 APB Master 总线接口。本例中 FPGA 侧逻辑作为 APB Slave 模块，支持 4 个 32bit 寄存器的读写操作。系统框图如下图所示。

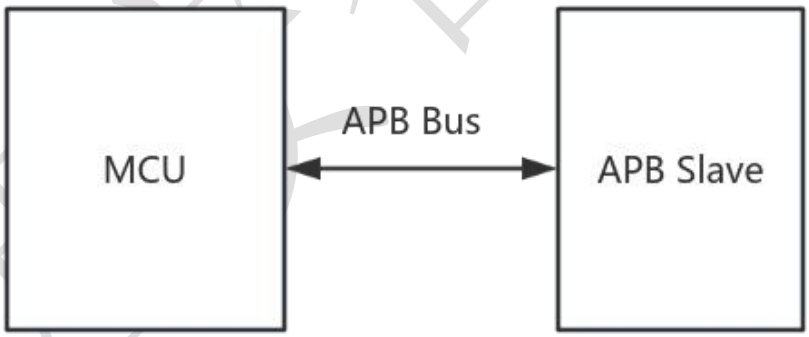


图 3-115 APB 例程框图

3.16.2 引脚分配

本例程不需要约束额外的引脚，只需要对基本的时钟、复位信号和打印测试信息用到的 UART0 的引脚进行约束即可，具体物理约束如下图所示。

```
phycst.start
#clk
phycst.pin.set  {CLK_IN}    J15     -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#SW0
phycst.pin.set  {RST_N}     P19     -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#swd-port
phycst.pin.set  {SWD_IO}    B18     -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.pin.set  {SWD_CLK}   B17     -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
```

```
#LED-D1
phycst.pin.set  {FPGA_LED}      R19     -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D2
phycst.pin.set  {LED2}          T21     -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D3
phycst.pin.set  {LED3}          T20     -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
######UART0######
#uart0-rx
phycst.pin.set  {UART0_RX}  G15 -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#uart0-tx
phycst.pin.set  {UART0_TX}  G16 -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.end
```

图 3-116 物理约束

3.16.3 代码设计

通过 APB 进行读操作代码如下：

```
uint32_t read_apb_reg(uint32_t addr)
{
    unsigned int *data;
    data = (unsigned int *)addr;
    return *data;
}
```

图 3-117 APB 读操作

其中 `BASE_ADDR` 为总线基地址，由当前 APB 总线使用的扩展通道基地址决定，若 APB 接口当前为扩展模块，此基地址还受当前 APB 接口序号影响，可参考 `BASE_ADDR` 的定义。

通过 APB 进行总线写操作的代码如下：

```
void write_apb_reg(uint32_t addr, uint32_t data)
{
    *(uint32_t *)addr = data;
}
```

图 3-118 APB 写操作

APB 读写测试函数如下，先分别向地址 0x0,0x4,0x8 和 0xC 写入数据 1~4，在 FPGA 侧的 APB Slave 模块中，寄存器地址 0x0 的数据受地址 0x4 的数据最低位影响，当最低位为 0 时，数据保持，当最低位为 1 时，数据以伪随机数更新。故我们可读写地址 0x4,0x8 和 0xC 的数据，同时可修改地址 0x4 数据的最低位，读取地址 0x0 的随机数据，实现 APB 总线读写测试，测试函数如下。

```
void APB_readWrite_test()
{
    uint32_t data;
    //write
    printf("APB write:\r\n");
    for(uint32_t i = 0;i <4;i++)
    {
        write_apb_reg(BASE_ADDR + i*4,i+1);
        printf("reg_addr %x: data = %x\r\n",i*4,i+1);
    }
    //read
    printf("APB read:\r\n");
    for(uint32_t i = 0;i <4;i++)
    {
        data = read_apb_reg(BASE_ADDR + i*4);
        printf("reg_addr %x: data = %x\r\n",i*4,data);
    }
    //read reg0
    for(uint32_t i = 0;i <2;i++)
    {
        data = read_apb_reg(BASE_ADDR + 0);
        printf("reg_addr0: data = %x\r\n",data);
    }
```

```
    write_apb_reg(BASE_ADDR + 4,1);
    printf("Write reg_addr %x: data = %x\r\n",4,1);
    //read reg0
    for(uint32_t i = 0;i <2;i++)
    {
        data = read_apb_reg(BASE_ADDR + 0);
        printf("reg_addr0: data = %x\r\n",data);
    }
}
```

图 3-119 APB 读写测试函数

3.16.4 下载验证

程序下载后，在串口调试助手可以看到通过 APB 写入和读出的数据信息，如下图所示。



图 3-120 串口打印信息

3.17 AXI-lite Master 例程

本例程配置 MCU 侧为 AXI-lite Master 接口，接 Slave 模块，MCU 侧实现 AXI-lite 总线读写。

3.17.1 AXI-lite 简介

AXI_Lite 是 AXI 协议的简化版，主要用于配置寄存器，具有单次读写、资源消耗少的特点。它包括分离的读写通道，每个通道有特定的地址和数据传输。握手协议通过 DATA、READY、VALID 信号确保数据安全传输，防止亚稳态问题。在 IP 配置中可通过总线配置界面，勾选扩展通道，配置 AXI-lite Master 总线接口。本例中 FPGA 侧逻辑作为 AXI-lite Slave 模块，支持 32 个 32bit 寄存器的读写操作。系统框图如下图所示。

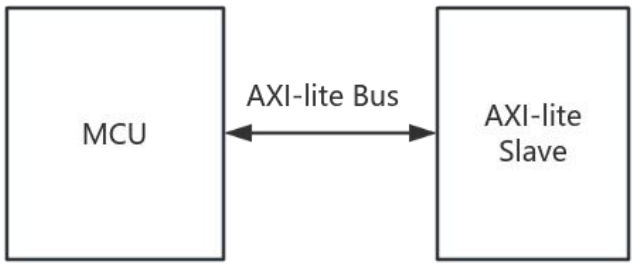


图 3-121 APB 例程框图

3.17.2 引脚分配

本例程不需要约束额外的引脚，只需要对基本的时钟、复位信号和打印测试信息用到的 UART0 的引脚进行约束即可，具体物理约束如下图所示。

```
phycst.start
#clk
phycst.pin.set  {CLK_IN}    J15     -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#SW0
phycst.pin.set  {RST_N}     P19     -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#swd-port
phycst.pin.set  {SWD_IO}    B18     -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.pin.set  {SWD_CLK}   B17     -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D1
phycst.pin.set  {FPGA_LED}  R19     -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D2
phycst.pin.set  {LED2}      T21     -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D3
phycst.pin.set  {LED3}      T20     -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
######UART0######
#uart0-rx
phycst.pin.set  {UART0_RX}  G15     -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#uart0-tx
phycst.pin.set  {UART0_TX}  G16     -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.end
```

图 3-122 物理约束

3.17.3 代码设计

通过 APB 进行读操作代码如下：

```
uint32_t read_axil_reg(uint32_t addr)
{
    unsigned int *data;
    data = (unsigned int *)(BASE_ADDR + addr);
    return *data;
}
```

图 3-123 AXI-lite 读操作

其中 BASE_ADDR 为总线基地址，由当前 AXI-lite 总线使用的扩展通道基地址决定。

通过 APB 进行总线写操作的代码如下：

```
void write_axil_reg(uint32_t addr, uint32_t data)
{
    *(uint32_t *)(BASE_ADDR + addr) = data;
}
```

图 3-124 AXI-lite 写操作

AXI-lite 读写测试函数如下，先分别向地址 0x0,0x4,0x8 和 0xC 写入数据，随后再读出 4 个地址的数据，实现 AXI-lite 总线读写测试，测试函数如下。

```
void AXIL_readWrite_test()
{
    uint32_t data;
    uint32_t i;
    //write
    printf("AXIL write:\r\n");
    for(i = 0;i <4;i++)
    {
        write_axil_reg(i*4,0xa0b0c0d0+i+1);
        printf("reg_addr %x: data = %x\r\n",i*4,0xa0b0c0d0+i+1);
    }
    //read
    printf("AXIL read:\r\n");
    for(i = 0;i <4;i++)
    {
        data = read_axil_reg(i*4);
        printf("reg_addr %x: data = %x\r\n",i*4,data);
```

```
    }
}
```

图 3-125 AXI-lite 读写测试函数

3.17.4 下载验证

程序下载后，在串口调试助手可以看到 AXI-lite 总线写入和读出的数据信息，如下图所示。

```
STAR AXI-lite test...
AXIL write:
reg_addr 0: data = a0b0c0d1
reg_addr 4: data = a0b0c0d2
reg_addr 8: data = a0b0c0d3
reg_addr c: data = a0b0c0d4
AXIL read:
reg_addr 0: data = a0b0c0d1
reg_addr 4: data = a0b0c0d2
reg_addr 8: data = a0b0c0d3
reg_addr c: data = a0b0c0d4
```

图 3-126 串口打印信息

3.18 AHB Master 接口扩展例程

本例程配置 MCU 侧为 2 个 AHB Master 接口，各挂接 Slave 模块，MCU 侧实现对 AHB Slave 模块的读写操作。

3.18.1 AHB 扩展简介

在 IP 配置中可通过总线配置界面，勾选扩展通道，配置接口类型为 AHB，便可选择 Master 接口数量，范围 1~16。本例中接口数量选择为 2，分别连接 Slave 模块，Slave 实现 AHB 接口到 FPGA RAM 接口的转换，实现 RAM 读写操作，与 AHB Master 例程一致。系统框图如下图所示。

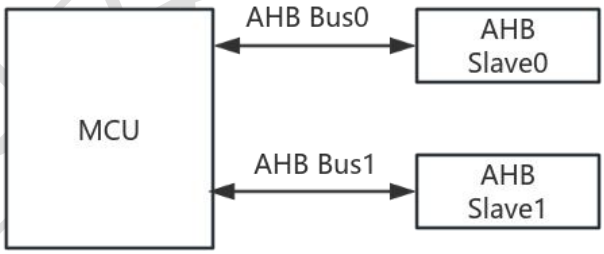


图 3-127 AHB 扩展例程框图

3.18.2 引脚分配

本例程不需要约束额外的引脚，只需要对基本的时钟、复位信号和打印测试信息用到的 UART0 的引脚进行约束即可，具体物理约束如下图所示。

```
phycst.start
#clk
phycst.pin.set  {CLK_IN}    J15     -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#SW0
phycst.pin.set  {RST_N}     P19     -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#swd-port
```

```
phycst.pin.set  {SWD_IO}    B18     -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.pin.set  {SWD_CLK}   B17     -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D1
phycst.pin.set  {FPGA_LED}      R19     -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D2
phycst.pin.set  {LED2}          T21     -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D3
phycst.pin.set  {LED3}          T20     -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
######UART0######
#uart0-rx
phycst.pin.set  {UART0_RX}  G15 -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#uart0-tx
phycst.pin.set  {UART0_TX}  G16 -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.end
```

图 3-128 物理约束

3.18.3 代码设计

本例程在 AHB Master 例程基础上修改而来，只是由一组 AHB 接口变更为 2 组，同时 Slave 模块也由 1 个变更为 2 个，总线读写函数与之前一致。

每组 AHB 接口有不同的基地址，本例程中，使用扩展通道 0 扩展为 2 组 AHB 通道，扩展通道 0 的基地址为 0x60000000，在 C 函数中为 **STAR_TARGEXP0_BASE**，接口地址增量为 64MB，所以有如下基地址定义：

```
#define CH0_M00_BASE_ADDR   STAR_TARGEXP0_BASE + 0*0x04000000
#define CH0_M01_BASE_ADDR   STAR_TARGEXP0_BASE + 1*0x04000000
```

AHB 读写测试函数如下，在 AHB Master 例程的基础上，增加了对 M01 接口的读写操作，可通过读写数据验证扩展成功。

```
void AHB_readWrite_test(void)
{
    int id_ok = 0;
    int i = 0;
    uint32_t rData = 0;
    uint32_t rwNum= 512;

    //读取AHB Slave ID
    printf("read ahb slave0...\r\n");
    id_ok = ahbslave_id_check();
    if (id_ok == 0)
    {
        printf("ahb slave ID ok.\r\n");
    }
    else
    {
        printf("ahb slave ID error.\r\n");
    }
    delay_ms_soft(1000);
    //往FPGA RAM写入数据
    for (i = 0; i < rwNum; i++)
    {
        ahbslave_write((uint32_t *)(CH0_M00_BASE_ADDR + WR_ADDR_OFFSET), i);
        ahbslave_write((uint32_t *)(CH0_M00_BASE_ADDR + WR_DATA_OFFSET), i);
        printf("Write:wAddress:%d, wData: %x\r\n", i, i);
    }

    //ch0_m01
    //往FPGA RAM写入数据
    for (i = 0; i < rwNum; i++)
    {
        ahbslave_write((uint32_t *)(CH0_M01_BASE_ADDR + WR_ADDR_OFFSET), 0xa000+i);
        ahbslave_write((uint32_t *)(CH0_M01_BASE_ADDR + WR_DATA_OFFSET), 0xa000+i);
        printf("M01_Write:wAddress:%d, wData: %x\r\n", i, 0xa000+i);
    }
    delay_ms_soft(10);
```

```
    delay_ms_soft(10);


    //将写入的数据从 FPGA RAM 回读
    for (i = 0; i < rwNum; i++)
    {
        ahbslave_write((uint32_t *)(CH0_M00_BASE_ADDR + RD_ADDR_OFFSET), i);
        rData = ahbslave_read(CH0_M00_BASE_ADDR + RD_DATA_OFFSET);
        printf("Read:rAddress:%d, rData: %x\r\n", i, rData);
    }

    //ch0_m01
    //将写入的数据从 FPGA RAM 回读
    for (i = 0; i < rwNum; i++)
    {
        ahbslave_write((uint32_t *)(CH0_M01_BASE_ADDR + RD_ADDR_OFFSET), i);
        rData = ahbslave_read(CH0_M01_BASE_ADDR + RD_DATA_OFFSET);
        printf("M01_Read:rAddress:%d, rData: %x\r\n", i, rData);
    }
}
```

图 3-129 APB 读写测试函数

3.18.4 下载验证

程序下载后，在串口调试助手可以看到通过 APB 写入和读出的数据信息，如下图所示。

```
STAR AHB test...
read ahb slave0...
data: 0xb1;
ahb slave ID ok.
Write:wAddress:0, wData: 0
Write:wAddress:1, wData: 1
Write:wAddress:2, wData: 2
Write:wAddress:3, wData: 3
Write:wAddress:4, wData: 4
Write:wAddress:5, wData: 5
Write:wAddress:6, wData: 6
Write:wAddress:7, wData: 7
Write:wAddress:8, wData: 8
Write:wAddress:9, wData: 9
Write:wAddress:10, wData: a
Write:wAddress:11, wData: b
Write:wAddress:12, wData: c
Write:wAddress:13, wData: d
Write:wAddress:14, wData: e
Write:wAddress:15, wData: f
Write:wAddress:16, wData: 10
Write:wAddress:17, wData: 11
Write:wAddress:18, wData: 12
Write:wAddress:19, wData: 13
Write:wAddress:20, wData: 14
Write:wAddress:21, wData: 15
```

图 3-130 串口打印信息

3.19 APB Master 接口扩展例程

本例程配置 MCU 侧为 2 个 APB Master 接口，各挂接 Slave 模块，MCU 侧实现对 APB Slave 模块的读写操作。

3.19.1 APB 扩展简介

在 IP 配置中可通过总线配置界面，勾选扩展通道，配置接口类型为 APB，便可选择 Master 接口数量，范围 1~16。本例中接口数量选择为 2，分别连接 Slave 模块，Slave 模块支持 4 个 32bit 寄存器的读写操作。系统框图如下图所示。

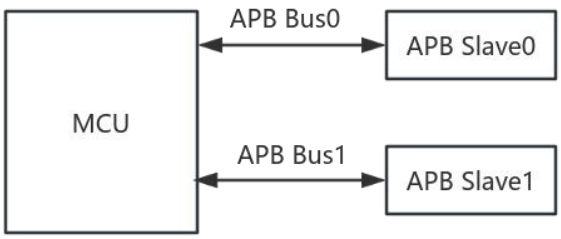


图 3-131 AHB 扩展例程框图

3.19.2 引脚分配

本例程不需要约束额外的引脚，只需要对基本的时钟、复位信号和打印测试信息用到的 UART0 的引脚进行约束即可，具体物理约束如下图所示。

```
phycst.start
#clk
phycst.pin.set  {CLK_IN}    J15     -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#SW0
phycst.pin.set  {RST_N}     P19     -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#swd-port
phycst.pin.set  {SWD_IO}    B18     -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.pin.set  {SWD_CLK}   B17     -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D1
phycst.pin.set  {FPGA_LED}  R19     -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D2
phycst.pin.set  {LED2}      T21     -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#LED-D3
phycst.pin.set  {LED3}      T20     -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
######UART0######
#uart0-rx
phycst.pin.set  {UART0_RX}  G15     -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
#uart0-tx
phycst.pin.set  {UART0_TX}  G16     -attr "IO_TYPE=LVCMOS33 PULLMODE=NONE"
phycst.end
```

图 3-132 物理约束

3.19.3 代码设计

本例程在 APB Master 例程基础上修改而来，只是由一组 APB 接口变更为 2 组，同时 Slave 模块也由 1 个变更为 2 个，总线读写函数与之前一致。

每组 APB 接口有不同的基地址，本例程中，使用扩展通道 0 扩展为 2 组 APB 通道，扩展通道 0 的基地址为 0x60000000，在 C 函数中为 **STAR_TARGEXP0_BASE**，接口地址增量为 64MB，所以有如下基地址定义：

```
#define CH0_M00_BASE_ADDR   STAR_TARGEXP0_BASE + 0*0x04000000
#define CH0_M01_BASE_ADDR   STAR_TARGEXP0_BASE + 1*0x04000000
```

APB 读写测试函数如下，在 AHB Master 例程的基础上，增加了对 M01 接口的读写操作，可通过读写数据验证扩展成功。

```
void APB_readWrite_test()
{
    uint32_t data;
    //M00 write
    printf("APB0 write:\r\n");
    for(uint32_t i = 0;i <4;i++)
```

```
    {
        write_apb_reg(CH0_M00_BASE_ADDR+i*4,0xaabbcc00+i+1);
        printf("reg_addr %x: data = %x\r\n",i*4,0xaabbcc00+i+1);
    }
    //M01 write
    printf("APB1 write:\r\n");
    for(uint32_t i = 0;i <4;i++)
    {
        write_apb_reg(CH0_M01_BASE_ADDR+i*4,0x11bbcc00+i+1);
        printf("reg_addr %x: data = %x\r\n",i*4,0x11bbcc00+i+1);
    }

    //M00 read
    printf("APB0 read:\r\n");
    for(uint32_t i = 0;i <4;i++)
    {
        data = read_apb_reg(CH0_M00_BASE_ADDR+i*4);
        printf("reg_addr %x: data = %x\r\n",i*4,data);
    }
    //M01 read
    printf("APB1 read:\r\n");
    for(uint32_t i = 0;i <4;i++)
    {
        data = read_apb_reg(CH0_M01_BASE_ADDR+i*4);
        printf("reg_addr %x: data = %x\r\n",i*4,data);
    }
}
```

图 3-133 APB 读写测试函数

3.19.4 下载验证

程序下载后，在串口调试助手可以看到通过 APB 写入和读出的数据信息，如下图所示。



图 3-134 串口打印信息

3.20 DDR 作为扩展 RAM 例程

本例程使用 FPGA 侧 DDR 作为 MCU 的外部 RAM 使用，以实现 MCU 程序运行空间的扩展。

3.20.1 原理框图

如图，FPGA 配置文件（。bin）和 MCU 程序（。bin）存放在外部 FLASH 中，FPGA 上电后从 FLASH 中加载配置信息，同时将 MCU 程序写入 MCU 的 ROM，地址 0x00000000；MCU 在启动后，将初始化 RAM，将被指定的内容加载到 RAM 中运行，地址为 0x20000000。

我们启用 MCU 总线扩展通道 0，接口类型为 AHB，在 FPGA 侧通过 ahb2ddrc 模块，实现 MCU 通过 AHB 总线（地址 0x60000000）到 DDR 的访问，即可将 DDR 作为 MCU 的扩展 RAM 使用。

使用 IDE 工具 Keil 的配置功能，将在片外 RAM 运行的源码文件链接至外部 RAM 区，实现 MCU 在启动后，将链接到外部 RAM 的内容加载到外部 RAM 中运行。

DDR 作为 MCU AHB 总线上，基地址为 0x60000000 的外挂设备，由于 AHB 总线一个地址对应 1byte 数据，而 DDR 地址与数据的对应关系由实际的板上数据位宽决定，地址对应关系通过 ahb2ddrc 模块实现，对 MCU 侧而言，就是通用 AHB 外设。

由于 DDR 作为外部 RAM，实际访问效率远低于内部 RAM，所以在实际使用中，建议可以将初始化或效率要求不高的代码放到外部 RAM 中运行。

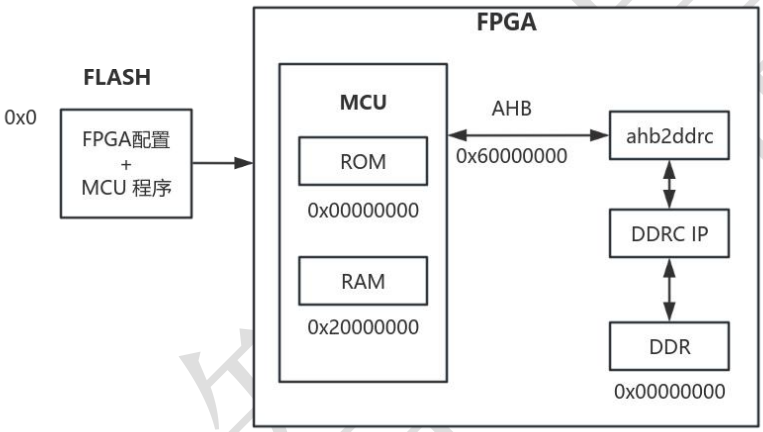


图 3-135 RAM 扩展原理框图

3.20.2 文件链接到外部 RAM

本例程功能为实现 LED 闪烁，在 led.c 文件包含 2 个函数，delay_ms( ) 和 led_test( )，本例程将 led.c 文件的代码段链接到外部 RAM，便可让文件中的 2 个函数在外部 RAM 中运行。

首先，我们需要在 Keil 中启用并设置外部 RAM 空间，如下图。由于我们将 DDR 作为外部 RAM，DDR 是 MCU 通过总线扩展通道 0 的 AHB 接口，再由 FPGA 侧逻辑模块 ahb2ddrc 连接到 DDRC 控制器，实现 AHB 接口对 DDR 的读写操作，所以外部 RAM 的地址即为总线扩展通道 0 的基地址，大小我们设置为 1 MB。

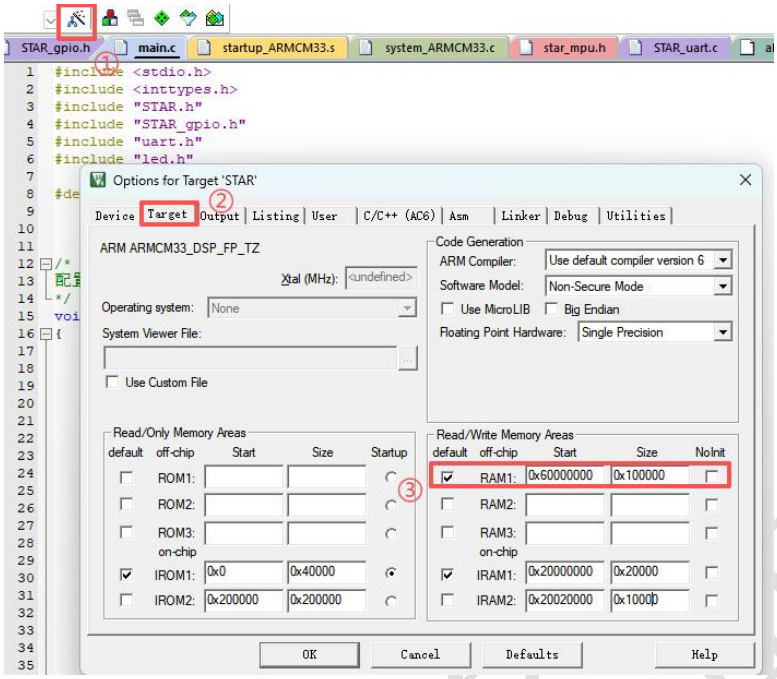


图 3-136 Keil 启用外部 RAM

同时，我们需要在“Linker”项中确保勾选了“Use Memory Layout from Target Dialog”，以实现代码编译时根据【Target】选项中设定的 RAM 和 ROM 地址自动加载生成一个加载文件。最后链接器会根据此文件中的信息对目标文件进行链接，生成 axf 镜像文件。

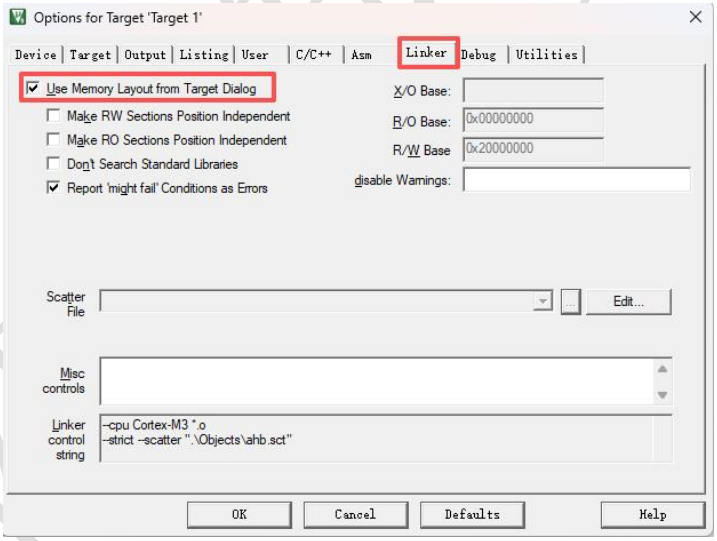


图 3-137 Keil Linker 界面

其次，我们需要将在外部 RAM 中运行的 Code 链接到外部 RAM 的地址。在 Project 界面，鼠标右键单击 led.c 文件，选择“Options for...”。

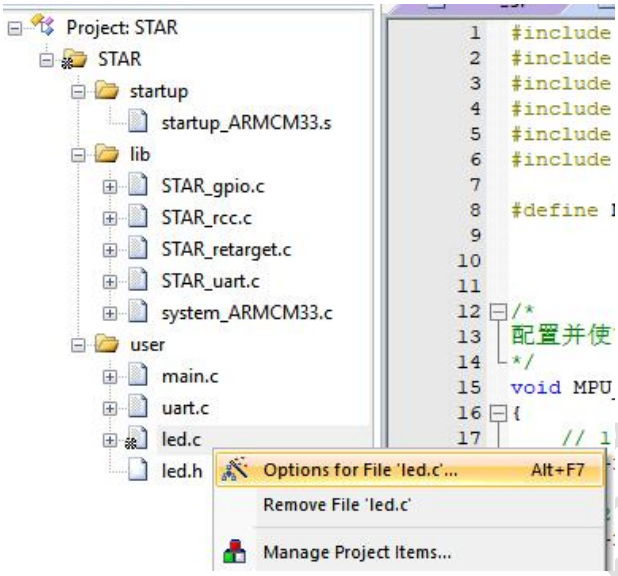


图 3-138 Keil 工程界面

如下图，我们进入到 led.c 文件配置页，在“Memory Assignment”项里，选择“Code/Const”的下拉框，选择设置好的外部 RAM 地址，再点击“OK”。在 Project 界面 led.c 文件的图标左下角出现一个“x”，表示 led.c 文件的 Code 已修改内存分配地址。

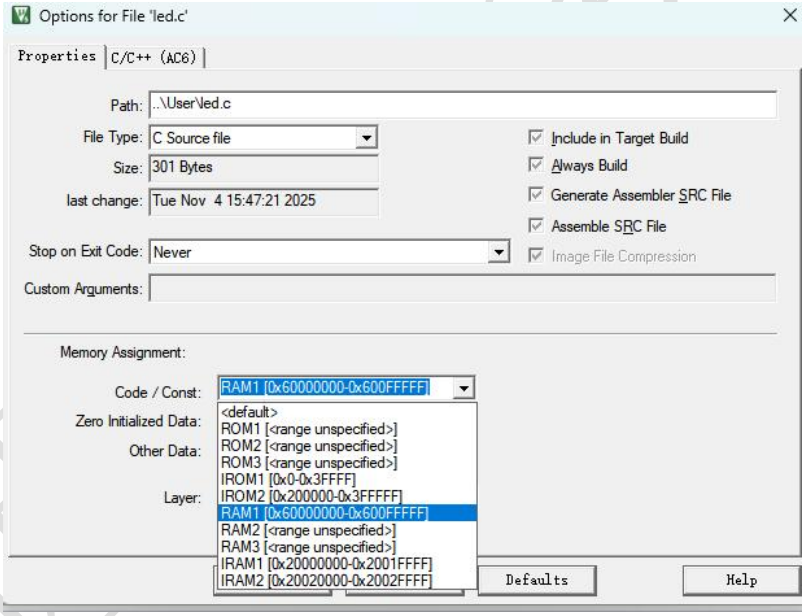


图 3-139 Keil C 文件内存分配界面

第三，为了让 MCU 可以访问外部 RAM，我们需要设置 MPU，见如下代码。

```
void MPU_set(void)
{
    // 1. 选择区域0
    MPU->RNR = 0;

    // 2. 设置区域0的基址寄存器
    MPU->RBAR = (0x60000000 & ~0x1F)    // 保证基地址对齐
              | (0 << 3)                // SH（0，不共享）
              | (1 << 1)                // AP（01，任何权限可读写）
              | 0;                      // XN=0（允许执行）

    MPU->MAIR0=0x00;                    //Device-nGnRnE

    // 3. 设置区域0的限制地址和属性寄存器
    MPU->RLAR = ((0x60000000 + 0x00100000 - 1) & ~0x1F)    // 限制地址对齐
```

```
                | (0 << 1)      //AttrIndx
                | 1;          //enable

    // 4. 使能 MPU 和特权默认访问
    MPU->CTRL = 7;

    // 5. 屏障指令
    __DSB();
    __ISB();
}
```

图 3-140 MPU 设置函数

至此，MCU 工程的设置已完成。编译工程，在。map 文件中可以看到 led.c 中的两个函数已链接到 0x60000000 的外部 RAM 空间：

```
__I$use$fp                                0x00002bca   Number         0  usenofp.o(x$fpl$usenofp)
Region$$Table$$Base                       0x00002cc8   Number         0  anon$$obj.o(Region$$Table)
Region$$Table$$Limit                      0x00002cf8   Number         0  anon$$obj.o(Region$$Table)
__ctype                                   0x00002d21   Data           0  lc_ctype_c.o(locale$$data)
delay_ms                                  0x60000001   Thumb Code    64  led.o(.text.delay_ms)
led_test                                  0x60000041   Thumb Code    76  led.o(.text.led_test)
Long Thumb to Thumb Veneer to GPIO_ResetBit 0x6000008d   Thumb Code    10  anon$$obj.o(Veneer$$Code)
Long Thumb to Thumb Veneer to GPIO_SetBit 0x60000097   Thumb Code    10  anon$$obj.o(Veneer$$Code)
Long Thumb to Thumb Veneer to printf      0x600000a1   Thumb Code    10  anon$$obj.o(Veneer$$Code)
SystemCoreClock                           0x600000cc   Data           4  system_armcm33.o(.data.SystemCoreC
__libspace_start                          0x600000d0   Data          96  libspace.o(.bss)
__stdout                                  0x60000130   Data          84  star_retarget.o(.bss.__stdout)
```

图 3-141 工程。map 文件

通过 Jlink 连接开发板进行在线调试时，在函数 led_test( ) 内部打断点，也可以看到代码在外部 RAM 运行。

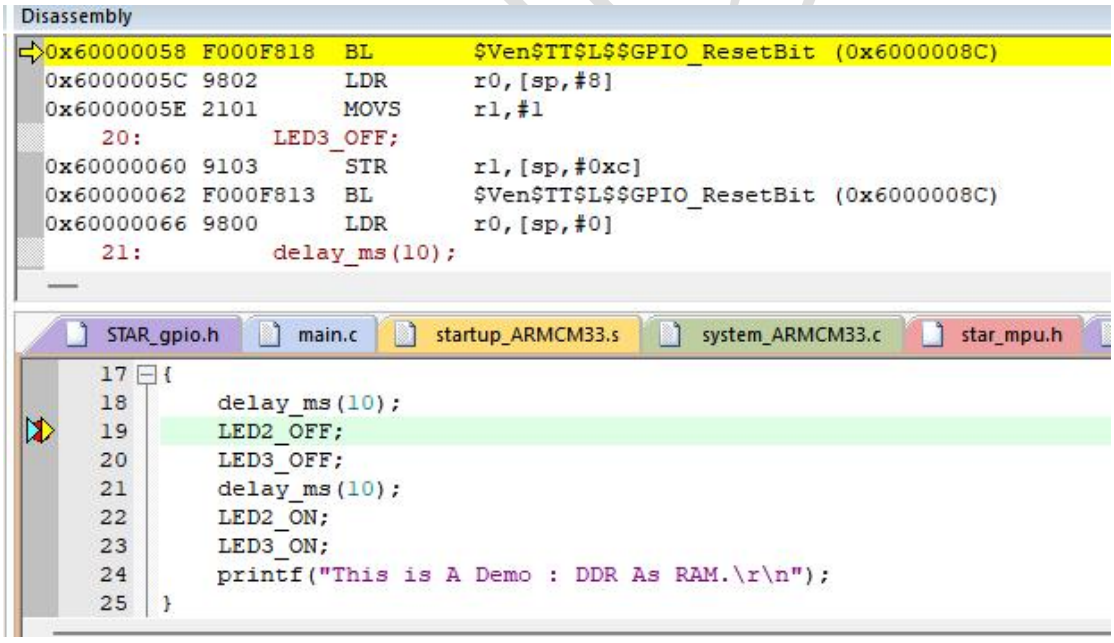


图 3-142 Keil 调试界面

3.20.3 下载验证

程序下载后，我们可以看到 LED2 和 LED3 闪烁，DDR 作为 MCU 扩展 RAM 成功运行。

3.21 DDR 作为扩展 ROM 例程（IAP）

本例程使用 FPGA 侧 DDR 作为 MCU 的外部 ROM 使用，以实现 MCU 程序存储空间的扩展，同时也能实现 MCU 程序的在线升级。

3.21.1 原理框图

如图，FPGA 配置文件（。bin）和 MCU bootloader 文件（。bin）存放在外部 FLASH 中，FPGA 上电后从 FLASH 中加载配置信息，同时将 MCU bootloader 写入 MCU 的 ROM，地址 0x00000000；MCU 在启动后，将初始化 RAM，将被指定的内容加载到 RAM 中运行，地址为 0x20000000。

bootloader 代码可实现通过串口接收 app 代码，并将 app 代码写入外部 FLASH 的 0x1F0000 地址段(需随实际 FPGA 工程 bin 文件+MCU bootloader 工程 bin 文件合并后的 bin 文件大小调整此地址，此地址需大于 bin 文件大小）。再将 FLASH 中的 App 代码搬移至 DDR，随后跳转程序运行地址到 app 代码起始地址，即可实现 app 代码的运行。

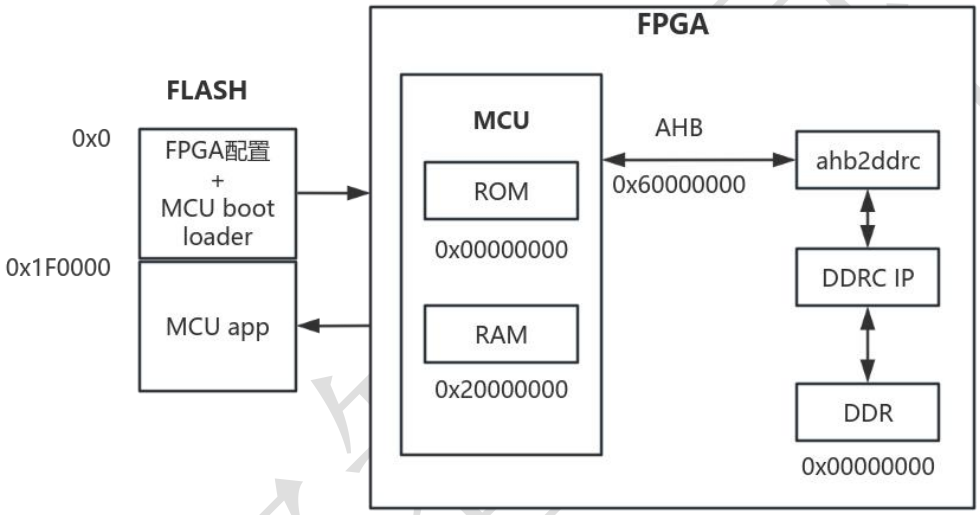


图 3-143 ROM 扩展原理框图

3.21.2 bootloader 代码

bootloader 主要实现以下四个功能：

1，实现 MCU app 代码通过串口接收，写进外部 FLASH；

2，从外部 FLASH 搬运 app 代码到外部 ROM；

3，设置 MPU 允许外部 ROM 空间是可访问的；

4，跳转到外部 ROM 运行；

在本例程中，功能 2 和 4 是通过按键实现的，实际项目中可使用其他交互方式实现。

bootloader 代码的 ROM 和 RAM 配置使用默认配置，如下图：

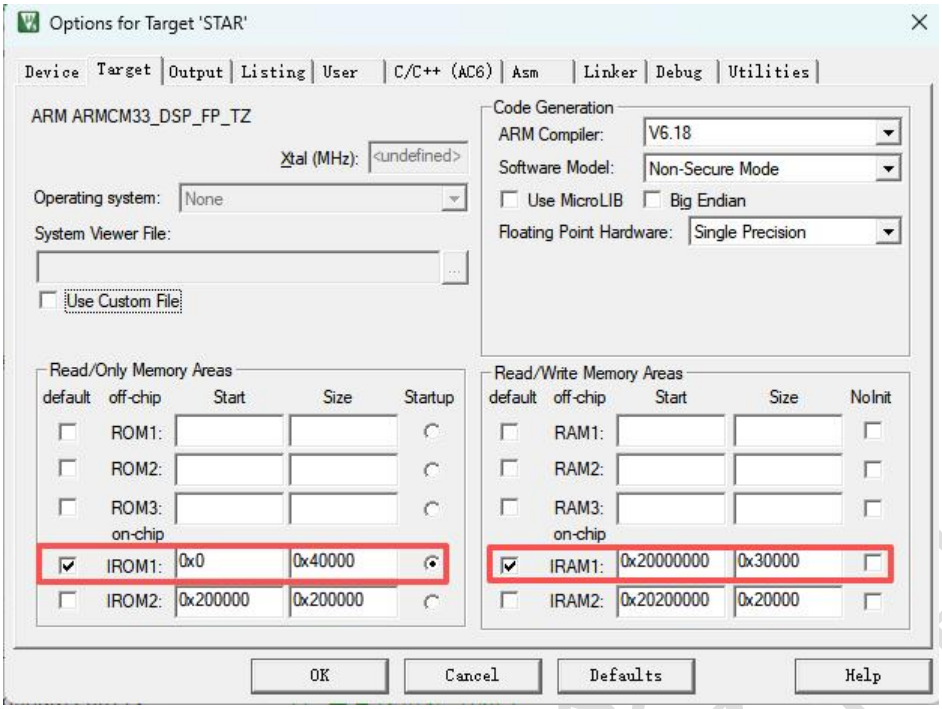


图 3-144 bootloader ROM 和 RAM 配置

bootloader 代码的工作流程图如下：

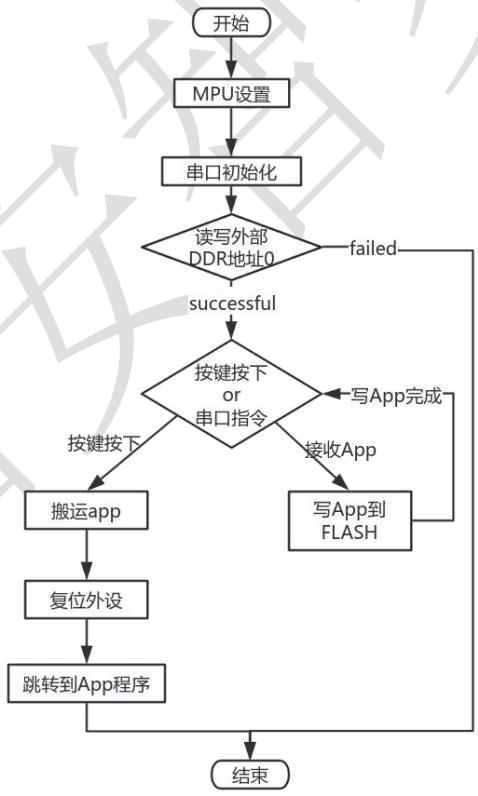


图 3-145 bootloader 代码工作流程

bootloader 代码运行后，首先进行 MPU 设置，如下代码，允许 MCU 访问 0x60000000 到 0x60100000 的空间，若需要调整空间大小，可对 `MPU->RBAR` 和 `MPU->RLAR` 的值进行修改。

```
void MPU_set(void)
```

```
{
    // 1. 选择区域0
    MPU->RNR = 0;

    // 2. 设置区域0的基址寄存器
    MPU->RBAR = (0x60000000 & ~0x1F)    // 保证基地址对齐
              | (0 << 3)               // SH（0，不共享）
              | (1 << 1)               // AP（01，任何权限可读写）
              | 0;                     // XN=0（允许执行）

    MPU->MAIR0=0x00;                    //Device-nGnRnE
```

```
    // 3. 设置区域0的限制地址和属性寄存器
    MPU->RLAR = ((0x60000000 + 0x00100000 - 1) & ~0x1F)    // 限制地址对齐
              | (0 << 1)      //AttrIndx
              | 1;            //enable

    // 4. 使能MPU和特权默认访问
    MPU->CTRL = 7;
```

```
    // 5. 屏障指令
    __DSB();
    __ISB();
}
```

图 3-146 MPU 设置

通过对地址 0x6000000 的读写，判断外挂的存储器是否可正确访问，如下代码：

```
write_ahb_reg(ApplicationAddress,0x12345678);      //向外部DDR写数据
if  (0x12345678 == read_ahb_reg(ApplicationAddress))    //从外部DDR读数据，判断DDR能成功访问
```

图 3-147 DDR 访问测试

在能正确访问 DDR 后进入 while 循环，在循环中**等待按键按下**或者**等待串口指令。**

```
        while(1)
        {
            keyval = GPIO_ReadInputData(CM3DS_MPS2_GPIO0,GPIO_Pin_4); //读取按键对应GPIO的电平

            if(keyval==Bit_RESET)
            {
                printf("Start Application Code Copy.\r\n");
                flash2ext_mem(FLASH_PROG_ADDR_START,ApplicationAddress,80); //搬运通过串口写入
flash的app代码到外部存储器空间，1次写256byte,4次1kB，此demo搬运20 KB
                                                    //请根据实际需求设置
                printf("Jump To Application Code.\r\n");
                //在跳转到外部SDRAM执行对应的app程序前需要将bootloader中使用到的外设复位，不然会影
响app程序中对这些外设的使用。
                uart0_Deinit();
                //跳转到App程序执行
                jump_to_ext_sdram_app();

                return 0;
            }
            cmd_process(); //串口升级指令处理
        }
```

图 3-148 等待按键按下或者等待串口指令

先介绍串口指令，支持 FLASH 擦除指令、App 更新指令和 FLASH 读指令。可通过串口升级工具 SerialUpgradeTool 实现以上指令的下发。

如下图为串口升级工具，首先需要在板卡串口与 PC 机连接后，正确配置串口并打开串口，接着选择我们用于升级的 App 代码（。bin）；接着点击“擦除 Flash”，等待返回“Flash 擦除成功!”(擦除地址与空间大小在 bootloader 代码 cmd.c 中通过修改函数 flash_sector_erase 的参数）；再点击下载，可看到数据传输的进度条，直到返回“升级文件发送完成！”，表示 app 代码已写入 Flash。可通过设置“起始地址”和“读取长度”，执行“回读升级文件”，比较回到文件与下载的。bin 文件数据内容对比，判断 Flash 写入是否正确。



图 3-149 串口升级工具界面

完成了 app 代码的更新，我们便可以通过按下按键，搬运 Flash 中的 app 代码到外部 DDR 中，通过函数

```
flash2ext_mem(FLASH_PROG_ADDR_START,ApplicationAddress,80); //搬运通过串口写入 flash 的 app 代码到外部存储器空间，1 页 256byte,4 页 1kB，此 demo 搬运 20 KB
```

实现，这里 `FLASH_PROG_ADDR_START` 为 app 在 Flash 中的起始地址，`ApplicationAddress` 为外部扩展 RAM 的基地址，`80` 为搬运的次数。需根据实际项目修改相应参数值。

随后对在 bootloader 代码中已使用的外设进行复位操作，此 demo 中只使用了串口，故只完成串口的复位。

最后跳转程序执行地址到 App 程序在 ROM 中的地址，即可跳转到 App 程序运行。

```
void jump_to_ext_sdram_app(void)
{
    uint32_t destAdr;
    pFun pMainFun;
    uint32_t dat;

    destAdr = ApplicationAddress;
    dat = *((uint32_t*)destAdr);                                    // 读取向量表第一个字（初始栈指针）
    if(0x20000000ul == (dat&0x2fff0000ul))
    {
        destAdr = *((uint32_t*)(ApplicationAddress+4));             // 读取向量表第二项（复位函数地址）
        //pMainFun = (pFun)destAdr;                                 // 转换为函数指针
        __set_MSP(*((uint32_t*)ApplicationAddress));                // 重置栈指针（MSP）
        __DSB();                                                    // 确保内存操作完成
        __ISB();                                                    // 清空处理器流水线
        pMainFun = (pFun)(*((uint32_t*)(ApplicationAddress+4)));    // 转换为函数指针
        pMainFun();                                                 // 跳转到应用程序
    }
}
```

图 3-150 跳转到 app 入口函数

3.21.3 App 代码

App 代码需要存储在外部 ROM 中，故我们需要配置外部 ROM 相关信息，起始地址为外部 ROM 的挂载地址，空间大小此处设置为 1 MB（可根据实际需求进行调整。请注意，若修改此大小，需同步修改 bootloader 代码中的 MPU 设置以及 FPGA 代码中 ahb2ddrc 模块的 AHB 地址位宽）。如下图：

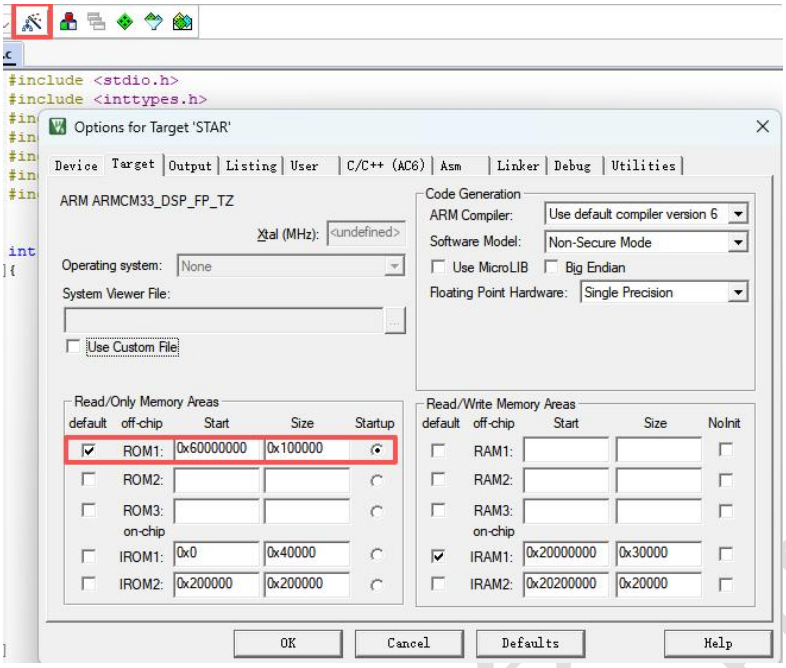


图 3-151 配置并启用外部 ROM

同时，我们需要在“Linker”项中确保勾选了“Use Memory Layout from Target Dialog”，以实现代码编译时根据【Target】选项中设定的 RAM 和 ROM 地址自动加载生成一个加载文件。最后链接器会根据此文件中的信息对目标文件进行链接，生成 axf 镜像文件。

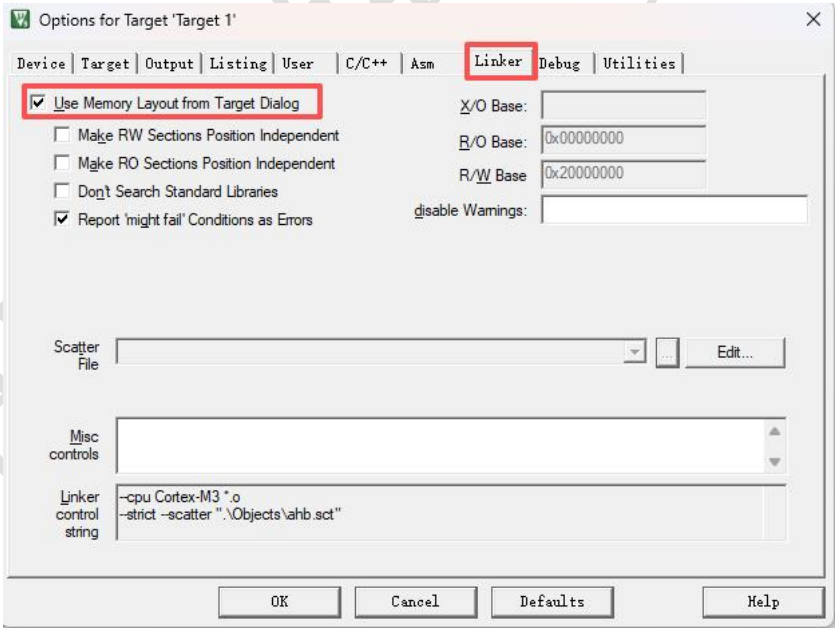


图 3-152 Keil Linker 界面

App 代码功能为实现 LED 的闪烁功能，此处不作介绍。

3.21.4 下载验证

下载 bootloader 代码并运行，通过按键可跳转到 App 代码运行（通过串口升级工具已完成 App 代码更新），能看到 LED2 和 LED3 闪烁，同时串口将打印相关信息。



图 3-153 下载运行串口打印信息

3.22 CAN 例程

本例程使用 FPGA 侧的 CAN_Controller IP，搭建了 CAN 总线的通信例程，可以实现帧发送（直接驱动 CAN 总线电机）和帧接收（串口打印接收到的帧信息）。

3.22.1 CAN 简介

CAN 总线（Controller Area Network）是一种广泛应用于工业控制、汽车电子等领域的通信协议。CAN 总线以其高可靠性、实时性和抗干扰能力，广泛应用于需要高效通信和复杂控制的领域。

3.22.2 系统框图

通过板载 CAN 总线收发器将 IP 的收发逻辑信号转换为差分 CAN 总线；右侧电机为 CAN 总线接口电机，可与开发板间实现帧收发，能直接驱动电机并获取电机状态。

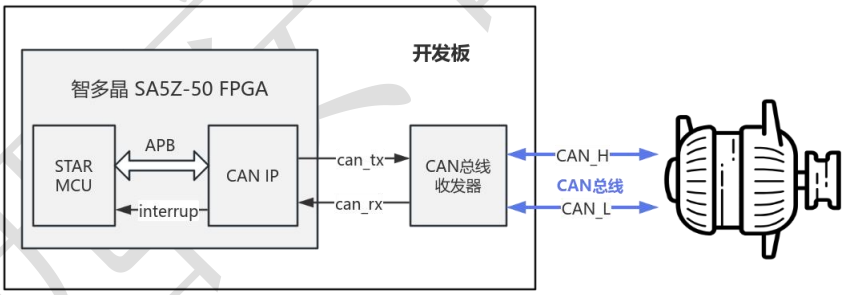


图 3-154 CAN demo 系统框图

3.22.3 寄存器访问地址

CAN IP 通过 APB 总线与 STAR 硬核连接，STAR 通过对 CAN 的寄存器操作实现对 CAN 总线的收发控制和状态获取。寄存器访问为 **32bit 总线地址=设备挂载基地址+寄存器偏移地址**，寄存器偏移地址可配置为 10~16bit（在 IP 生成界面配置）。

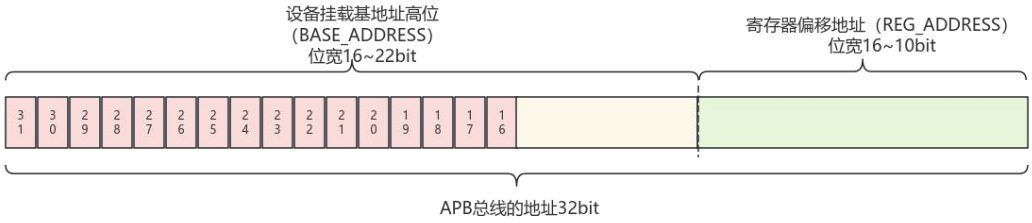


图 3-155 CAN 寄存器访问地址

3.22.4 API 函数

```
/* 功能：can 位时序值赋值
 * 参数：bit_timing 设备位时序参数结构体的地址指针
 */
void can_bit_timing_set(CAN_BIT_TIMING *bit_timing)

/* 功能：CAN 速率配置，位时序配置，相当于 can 的初始化
 * 参数：dev_addr 设备基地址（16bit，其中对应寄存器偏移地址的低位 bit 全部为 0）
 * 参数：bit_timing 设备位时序参数结构体指针
 */
u8 can_rate_set(u32 dev_addr,CAN_BIT_TIMING *bit_timing)

/* 功能：中断使能配置
 * 参数：dev_addr 设备基地址
 * 参数：int_en 中断使能值，参考寄存器表：0x01 为 crc 错误中断使能，0x02 为 bit 错误中断使能，0x04 为 ack
错误中断使能，
 *      0x08 为 form 错误中断使能，0x10 为 stuff 错误中断使能，0x40 为 passive_err 标志使能，0x80 为 bus_off
标志使能，
 *      0x40000000 为接收完成中断使能，0x80000000 为发送完成中断使能
 *     配置时若需要同时使能时，将对应使能值相加，对应位为 0 时，表示失能
 */
u8 set_int_en(u32 dev_addr,u32 int_en)

/* 功能：错误清除，错误信号清零
 * 参数：dev_addr 设备基地址
 */
u8 clear_err(u32 dev_addr)

/* 功能：设备复位
 * 参数：dev_addr 设备基地址
 */
u8 dev_rst(u32 dev_addr)

/* 功能：can 发送 1 帧数据
 * 参数：dev_addr 设备基地址
 * 参数：txMsg 帧内容信息
 */
u8 can_send_msg(u32 dev_addr,CAN_MSG *txMsg)

/* 功能：接收 1 帧数据
 * 参数：dev_addr 设备基地址
 * 参数：rxMsg 帧内容信息
 */
u8 can_rec_msg(u32 dev_addr,CAN_MSG *rxMsg)

/* 功能：读取 can 中断源寄存器
 * 参数：dev_addr 设备基地址
 * 参数：int_src 设备中断源字段的结构体指针
 */
u32 read_int_src(u32 dev_addr,CAN_INT_SRC *int_src)
```

3.22.5 代码设计

FPGA 代码主要由 CAN IP 和 STAR IP 组成，按键消抖使用了“key_filter”模块，APB 总线时钟频率为 50MHz，CAN IP 工作时钟 100MHz，STAR 核心时钟 200MHz。

MCU 代码基于 CAN API 函数，在 CAN 设备初始化完成后发送指令，实现电机控制，具体指令信息可通过串口打印输出或者接入 CAN 监控设备查看，如下为 demo 流程图。

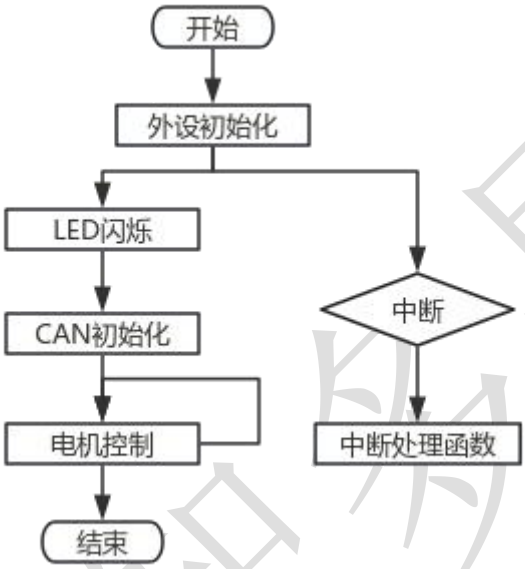


图 3-156 demo 流程图

如下为主函数，电机控制代码为**电机转动→电机停止→电机反向转动→电机停止**的循环。

```
int main(void)
{
    GPIO_DeInit(STAR_GPIO0);
    gpio_init();

    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);
    uart_init(115200);
    extint0_init();     //can interrrput
    switch_init();      //switch interrrput
    timer_init();
    printf("\r\nThis Is A CAN Demo!\r\n");
    led_test();         //LED 闪烁，mcu 运行

    //CAN 初始化
    //1mb/s
    can_bit_timing_set(can_bit_timing);            //位时序参数赋值
    can_rate_set(DEV_BASEADDR,can_bit_timing);     //位时序参数设置
    set_int_en(DEV_BASEADDR,0x400000df);           //启动错误中断和接收中断

    //电机控制
    while(1)
    {
        if(stop_flag != 1)
        {
            //1
            motor_run();
            delay_ms(15000);
            if(stop_flag == 1)
                return 0;
            //2
            motor_stop();
            delay_ms(10000);
            if(stop_flag == 1)
```

```
                return 0;
            //3
            motor_run2();
            delay_ms(15000);
            if(stop_flag == 1)
                return 0;
            //4
            motor_stop();
            delay_ms(10000);
            if(stop_flag == 1)
                return 0;
        }
    }

    while(1)
    {
        delay_ms(100);
    };
}
```

图 3-157 main 函数

demo 使用了 3 个按键，SW1 实现电机退出循环控制并停止，SW2 实现电机转动控制，SW3 实现电机停止，通过消抖后的按键脉冲信号连接到外部中断 1~3，按键功能通过外中断处理函数实现。

demo 同时支持帧消息接收。CAN 模块的中断信号连接到 STAR 的外部中断 0，使能 CAN 接收中断后，接收到帧消息将触发中断，进入中断处理函数，在中断处理函数中实现了帧消息获取并通过串口打印。以下为 CAN 中断处理函数和按键中断处理函数。

```
void EXT0_Handler(void)
{
    uint32_t value =0;
    read_int_src(DEV_BASEADDR,can_int_src0);

    //接收到帧
    if(can_int_src0.rx_fnsh == 1)
    {
        can_rec_and_print(DEV_BASEADDR,can_int_src0,rx_msg);
        can_int_src0.rx_fnsh = 0;
    }
    else
    {
        clear_err(DEV_BASEADDR);
        err_flag = 1;
        //printf("Int 0!\r\n");
    }
    NVIC_ClearPendingIRQ(EXT0_IRQn);
}

void EXT1_Handler(void)
{
    printf("Exit AutoDemo\r\n");
    sw_clr();
    stop_flag = 1;
    NVIC_ClearPendingIRQ(EXT1_IRQn);
}
```

```
void EXT2_Handler(void)
{
    if(stop_flag == 1)
    {
        printf("Running\r\n");
        motor_run();
    }
    sw_clr();
    NVIC_ClearPendingIRQ(EXT2_IRQn);
}
```

```
void EXT3_Handler(void)
{
    if(stop_flag == 1)
```

```
    {
        printf("Stop\r\n");
        motor_stop();
    }
    sw_clr();
    NVIC_ClearPendingIRQ(EXT3_IRQn);
}
```

图 3-158 中断处理函数

3.22.6 下载验证

1）完成 demo 的硬件连接后给开发板上电，如果程序运行正常，板载 LED1 将持续闪烁，标志 FPGA 逻辑正常运行，板载 LED2 快速闪烁 10 次后熄灭，标志 MCU 启动运行。

2）随后电机开始转动，10s 后电机减速直至停止，很快电机将重新开始反向转动，10s 后电机减速直至停止，随后重复上述启停过程。

3）板载按键 SW0 为复位按键，SW1 为退出电机循环状态（处于循环状态下，将持续发送转动和停止指令），SW2 为发送电机转动指令，SW3 为发送电机停止指令。

4）若接入 CAN 发帧模块，发送帧消息，串口可打印输出接收到的帧消息。

如图为 demo 自动运行情况下串口的打印信息。



图 3-159 串口打印输出

3.23 关断时钟降功耗例程

本例程使用 FPGA 侧的 xsDCC 原语，实现了 STAR 工作时钟的通断控制，使 STAR 可暂停运行，实现降低功耗。此做法可用于 MCU 不需要一直保持运行状态并对降低功耗有需求的应用场景，用户可根据应用需求自行控制 STAR 工作时钟的关断与开启，本例程通过按键 SW1 控制工作时钟的通断，并在串口连续打印累加数（通过累加数在时钟恢复后的连续输出，说明在时钟关断时，MCU 只是处于暂停运行状态，在时钟恢复后，将从停止点继续运行。注意：由于与外设的通信可能需要多个时钟周期才能完成，比如串口通信传输一个字节需要传输多 bit，并且每 bit 的传输需要多个时钟周期，如果在传输 4bit 数据时就关断时钟，那么在外设接口上一定会出现通信错误，此类错误不可避免，因此，时钟的关断时机需要用户自行避开 STAR 与外设的通信时段）。

3.23.1 系统框图

通过按键 SW1 实现 STAR 工作时钟的通断控制，串口打印连续累加数，以此展示通过关断 STAR 的工作时钟实现降低功耗的可行性。

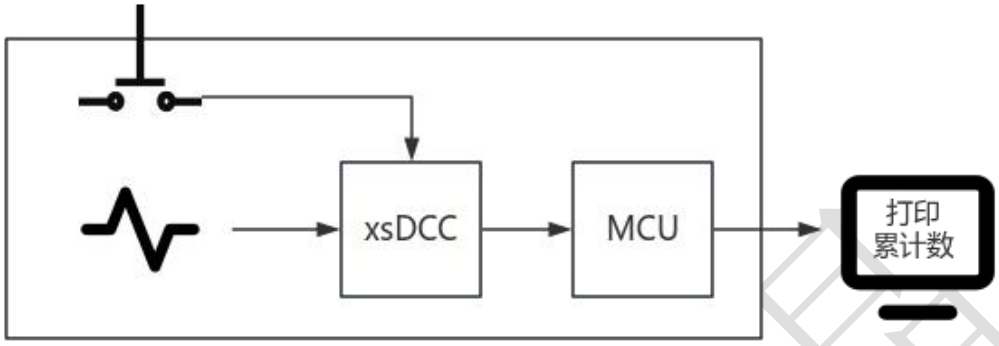


图 3-160 降功耗 demo 系统框图

3.23.2 代码设计

本例程的代码主要在 FPGA 侧实现 STAR 工作时钟的通道控制。如下代码就是按键 SW1 通过滤波后，控制 xsDCC 原语实现 STAR 工作时钟的通断控制。

```
key_debounce debounce_inst(
    .clk        (CLK_IN     ),
    .rst_n      (RST_N      ),
    .key_in     (SW1        ),
    .key_pulse  (key_release)  // 按键释放时输出脉冲
    );

always @(posedge CLK_IN)
begin
    if  (RST_N | !mcu_run)
        clk_en <= 1'b1;
    else if (key_release)
        clk_en <= ~clk_en;
    else
        clk_en <= clk_en;
end

xsDCC xsDCC_inst(
    .CLKI   (clk_200    ),
    .CE     (clk_en     ),
    .CLKO   (clk_mcu    )
);

//core instance
STAR_Processor mcu_inst(
    .CLK_IN             (clk_mcu     ),
    .SWD_CLK            (SWD_CLK     ),
    .SWD_IO             (SWD_IO      ),
    .GPIO0              (LED2        ),
    .GPIO1              (LED3        ),
    .UART0_RX           (UART0_RX    ),
    .UART0_TX           (UART0_TX    ),
    .RST_N              (RST_N       )
);
```

图 3-161 FPGA 侧逻辑

MCU 侧实现串口打印累加数，如下。

```
int main(void)
{
    uint32_t i = 0;
```

```
    GPIO_DeInit(STAR_GPIO0);

    /*LED GPIO 模式设置*/
    GPIO_Mode_Set(LED2_PORT,LED2_PIN,GPIO_Mode_Output);
    GPIO_Mode_Set(LED3_PORT,LED3_PIN,GPIO_Mode_Output);

    /*LED2 LED3 ON*/
    LED2_ON;
    LED3_ON;
    delay_ms(1000);

    /*LED2 LED3 OFF*/
    LED2_OFF;
    LED3_OFF;

    uart_init(115200);
    printf("STAR uart0 echo test...\r\n");

    while(1)
    {
        delay_ms(10);
        printf("counter = %d\r\n",i++);
    }
}
```

图 3-162 MCU 主函数

3.23.3 下载验证

程序下载后，我们可以看到窗口开始打印累加数，按下按键 SW1 后，打印停止，再次按下按键，打印从停止点继续开始打印。

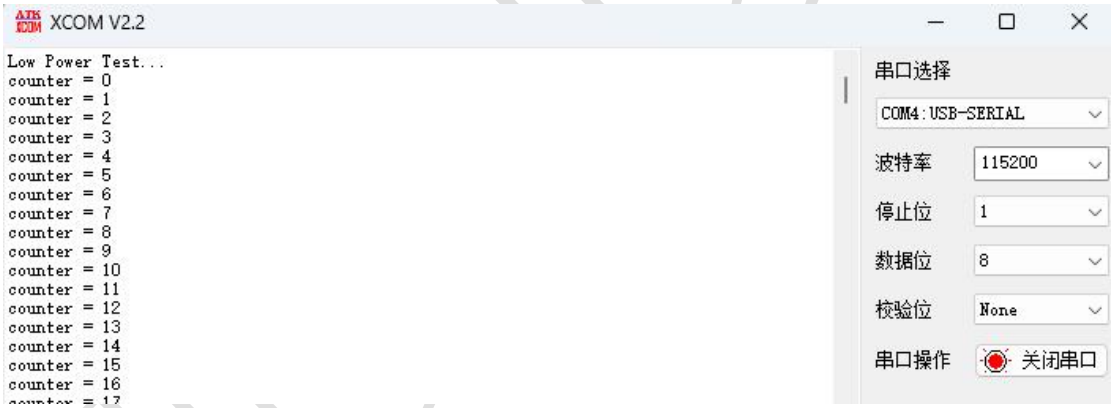


图 3-163 串口打印输出

3.23.4 功耗测试

在常温下，FPGA 核独立 1 V 供电时测试，关断 STAR 工作时钟后，FPGA 核电流降低约 30 mA，相当于功耗下降约 30 mW。

3.24 FreeRTOS 例程

本例程包含 1 个 FPGA 工程和 3 个 MCU 工程，MCU 工程分别实现①多任务，②通过队列实现任务间通信，③通过二值信号量实现中断处理，参考这 3 个工程，用户可快速开始 FreeRTOS 的入门级开发。更多教程可参考官方发布的《掌握 FreeRTOS 实时内核》一书。

3.24.1 FreeRTOS 简介

FreeRTOS 是一个轻量级的实时操作系统（RTOS）内核，适用于资源受限的嵌入式系统。

其功能包括：任务管理、时间管理、信号量、消息队列、内存管理、记录功能、软件定时器、协程等，可满足资源受限的小型系统的需要。

3.24.2 代码设计

本例程中 FPGA 工程为例程 4 的串口工程，MCU 工程有 3 个基础工程，以下依次介绍。

3.24.2.1 多任务（MCU_Prj）

在 FreeRTOS 的应用中，最多的应用需求就是实现多任务，使用 FreeRTOS 实现多任务相较裸机开发，可以实现任务与任务之间的独立性，并非像裸机开发中将不同的任务（功能函数）放入一个 while 循环，依次循环执行（存在阻塞），而是每个任务有自己的栈，任务间数据隔离，通过任务调度实现任务轮流执行，其模块化非常利于应用开发与扩展。

下面通过 main 函数介绍多任务的实现。

```
int main(void)
{
    GPIO_DeInit(STAR_GPIO0);

    /*LED GPIO 模式设置*/
    GPIO_Mode_Set(LED2_PORT,LED2_PIN,GPIO_Mode_Output);
    GPIO_Mode_Set(LED3_PORT,LED3_PIN,GPIO_Mode_Output);


    uart_init(115200);
    printf("FreeRTOS Multitask Demo.\r\n");


    BaseType_t xReturn =pdPASS; //创建任务返回值

    xReturn=xTaskCreate((TaskFunction_t)Task1,          //任务函数
                        (const char *)"Task1Create",    //任务名
                        (uint16_t)128,(void *)NULL,     //任务栈大小
                        (UBaseType_t)1,                 //任务优先级
                        (TaskHandle_t *)&Task1_Handle); //任务控制块指针

    xReturn=xTaskCreate((TaskFunction_t)Task2,          //任务函数
                        (const char *)"Task2Create",    //任务名
                        (uint16_t)128,(void *)NULL,     //任务栈大小
                        (UBaseType_t)1,                 //任务优先级
                        (TaskHandle_t *)&Task2_Handle); //任务控制块指针

     if(xReturn==pdPASS) //如果创建成功
     {
        vTaskStartScheduler(); //启动任务，开启调度
     }
     else
         return -1;

     while(1);
}
```

图 3-164 主函数代码

创建任务

代码中通过 `xTaskCreate()` 创建了 2 个任务。

```
BaseType_t xTaskCreate( TaskFunction_t pxTaskCode,
                        const char * const pcName,
                        const configSTACK_DEPTH_TYPE uxStackDepth,
                        void * const pvParameters,
                        UBaseType_t uxPriority,
                        TaskHandle_t * const pxCreatedTask )
```

图 3-165 xTaskCreate() API 函数原型

`pxTaskCode` 为具体的任务函数名（指向任务的实现函数的指针）；

pcName 为描述性的任务名，方便辅助调试；

uxStackDepth 用于告诉内核为任务分配多大的栈空间；

pvParameters 的值是传递到任务中的一个指向 void 的指针，可向任务传递参数；

uxPriority 指定任务执行的优先级。优先级的取值范围可以从最低优先级 **0** 到最高优先级（configMAX_PRIORITIES – 1）；

pxCreatedTask 用于传出任务的句柄。这个句柄将在 API 调用中对该创建出来的任务进行引用，比如改变任务优先级，或者删除任务。

有两个可能的返回值：

pdPASS：表明任务创建成功。

pdFAIL：表明内存堆空间不足，FreeRTOS 无法分配足够的空间来保存任务结构数据和任务栈，因此无法创建任务。

启动调度器

代码中通过 vTaskStartScheduler() 实现启动调度器，开始执行任务。至此创建的任务便根据优先级在系统的调度下实现轮流执行。若需要扩充功能，通过增加任务即可便捷实现。

任务函数

本例程中，“Task1”控制 LED2 闪烁，“Task2”控制串口打印和 LED3 闪烁。

```
static void Task1(void * parameter)
{
    while(1)
    {
        LED2_ON;
        vTaskDelay(500/portTICK_RATE_MS);
        LED2_OFF;
        vTaskDelay(500/portTICK_RATE_MS);
    }
}

static void Task2(void * parameter)
{
    while(1)
    {
        LED3_OFF;
        printf("You Can See The LED Flashing And The Serial Port Printing.\r\n");
        vTaskDelay(1000/portTICK_RATE_MS);
        LED3_ON;
        printf("You Can See The LED Flashing And The Serial Port Printing.\r\n");
        vTaskDelay(1000/portTICK_RATE_MS);
    }
}
```

图 3-166 任务函数

vTaskDelay() 函数实现任务阻塞，阻塞时间以心跳周期为单位，常量 portTICK_RATE_MS 可以用来在毫秒和心跳周期之间转换。处于阻塞状态的任务将不会被调度执行，相当于延时函数。

3.24.2.2 任务间通信（MCU_Prj2）

基于 FreeRTOS 的应用程序由一组独立的任务构成——每个任务都是具有独立权限的小程序。这些独立的任务之间很可能会通过相互通信实现有用的系统功能。FreeRTOS 中所有的通信与同步机制都是基于队列实现的。

下面通过 main 函数介绍任务间通信的实现。

```
int main( void )
{
    uart_init(115200);
```

```
    /* 创建的队列用于保存最多5 个值，每个数据单元都有足够的空间来存储一个long 型变量 */
    xQueue = xQueueCreate( 5, sizeof( long ) );
    if( xQueue != NULL )
    {
        /* 创建两个写队列任务实例，任务入口参数用于传递发送到队列的值。所以一个实例不停地往队列发送
        100，而另一个任务实例不停地往队列发送200。两个任务的优先级都设为1。 */
        xTaskCreate( vSenderTask, "Sender1", 1000, ( void * ) 100, 1, NULL );
        xTaskCreate( vSenderTask, "Sender2", 1000, ( void * ) 200, 1, NULL );
        /* 创建一个读队列任务实例。其优先级设为2，高于写任务优先级 */
        xTaskCreate( vReceiverTask, "Receiver", 1000, NULL, 2, NULL );
        /* 启动调度器，任务开始执行 */
        vTaskStartScheduler();
    }
    else
    {
        /* 队列创建失败*/
    }
    /* 如果一切正常，main()函数不应该会执行到这里。但如果执行到这里，很可能是内存堆空间不足导致空闲
    任务无法创建。第五章讲述更多关于内存管理方面的信息 */
    for( ; );
}
```

图 3-167 主函数代码

本例中通过 xQueueCreate()API 函数创建深度为 5，数据长度为 long 的队列。然后创建了 2 个构建数值到队列的“Sender”任务和 1 个从队列获取数值的“Receiver”任务，最后启动调度器。

任务“Sender1”和“Sender2”使用同一个任务函数，传输不同的 pvParameters 实现不同的队列写入值。函数 xQueueSendToBack() 用于将数据发送到队列尾，函数 xQueueReceive() 用于从队列中接收（读取）数据单元。接收到的单元同时会从队列中删除。相关的函数参数在代码注释中有介绍，更多的队列相关函数可通过官方文档学习。

```
static void vSenderTask(void * pvParameters)
{
    long lValueToSend;
    portBASE_TYPE xStatus;

    /* 该任务会被创建两个实例，所以写入队列的值通过任务入口参数传递 – 这种方式使得每个实例使用不同的
    值。队列创建时指定其数据单元为long 型，所以把入口参数强制转换为数据单元要求的类型 */
    lValueToSend = ( long ) pvParameters;
    /* 和大多数任务一样，本任务也处于一个死循环中 */
    for( ; )
    {
        /* 往队列发送数据
        第一个参数是要写入的队列。队列在调度器启动之前就被创建了，所以先于此任务执行。
        第二个参数是被发送数据的地址，本例中即变量lValueToSend 的地址。
        第三个参数是阻塞超时时间 – 当队列满时，任务转入阻塞状态以等待队列空间有效。本例中没有设定超
        时时间，因为此队列绝不会保持有超过一个数据单元的机会，所以也绝不会满。
        */

        xStatus = xQueueSendToBack( xQueue, &lValueToSend, 0 );
        if( xStatus != pdPASS )
        {
            /* 发送操作由于队列满而无法完成 – 这必然存在错误，因为本例中的队列不可能满。 */
            printf( "Could not send to the queue.\r\n" );
        }

        /* 允许其他发送任务执行。 taskYIELD()通知调度器现在就切换到其他任务，而不必等到本任务的时
        间片耗尽 */
        taskYIELD();
    }
}

static void vReceiverTask( void *pvParameters )
{
    /* 声明变量，用于保存从队列中接收到的数据。 */
    long lReceivedValue;
    portBASE_TYPE xStatus;
```

```
    const portTickType xTicksToWait = 100 / portTICK_RATE_MS;
    /* 本任务依然处于死循环中。 */
    for( ; )
    {
        /* 此调用会发现队列一直为空，因为本任务将立即删除刚写入队列的数据单元。 */
        if( uxQueueMessagesWaiting( xQueue ) != 0 )
        {
            printf( "Queue should have been empty!\r\n" );
        }
        /* 从队列中接收数据
        第一个参数是被读取的队列。队列在调度器启动之前就被创建了，所以先于此任务执行。
        第二个参数是保存接收到的数据的缓冲区地址，本例中即变量 lReceivedValue 的地址。此变量类型与
        队列数据单元类型相同，所以有足够的大小来存储接收到的数据。
        第三个参数是阻塞超时时间 – 当队列空时，任务转入阻塞状态以等待队列数据有效。本例中常量
        portTICK_RATE_MS 用来将 100 毫秒绝对时间转换为以系统心跳为单位的时间值。
        */
        xStatus = xQueueReceive( xQueue, &lReceivedValue, xTicksToWait );
        if( xStatus == pdPASS )
        {
            /* 成功读出数据，打印出来。 */
            printf( "Received = %d\r\n", lReceivedValue );
        }
        else
        {
            /* 等待 100ms 也没有收到任何数据。
            必然存在错误，因为发送任务在不停地往队列中写入数据 */
            printf( "Could not receive from the queue.\r\n" );
        }
    }
}
```

图 3-168 任务函数

3.24.2.3 中断处理（MCU_Prj3）

FreeRTOS 需要对整个系统环境产生的事件做出反应。举个例子，以太网外围部件收到了一个数据包（事件），需要送到 TCP/IP 协议栈进行处理（反应）。更复杂的系统需要处理来自各种源头产生的事件，这些事件对处理时间和响应时间都有不同的要求。通常情况下，中断服务（ISR） 应当越短越好，在某个特殊的中断发生时，让中断事件处理量大的工作（中断处理）在跳出 ISR 后立即执行，中断处理可以说是被”推迟（deferred）”到一个”处理
(handler)”任务，结合任务概念，便可实现 FreeRTOS 中的中断处理，本例程采样二值信号量实现延迟中断处理。

下面通过 main 函数介绍延迟中断处理的实现。

```
int main(void)
{
    // 串口初始化
    uart_init(115200);

    // 创建消息队列（128 字节大小）
    xUartRxQueue = xQueueCreate(128, sizeof(uint8_t));

    // 创建二值信号量
    xBinarySemaphore = xSemaphoreCreateBinary();

    if (xUartRxQueue != NULL && xBinarySemaphore != NULL)
    {
        // 创建数据处理任务
        xTaskCreate(vDataProcessTask, "DATA_PROC",
                    configMINIMAL_STACK_SIZE * 2, NULL, 1, NULL);

        printf("FreeRTOS 串口接收中断演示程序启动\r\n");

        // 启动调度器
```

```
        vTaskStartScheduler();
    }

    while (1) {
        // 程序不应执行到这里
    }
}
```

图 3-169 主函数代码

本例中通过 xQueueCreate()API 函数创建深度为 128，数据长度为 uint8_t 的队列，用于任务间通信。通过 xSemaphoreCreateBinary()API 函数创建了二值信号量 xBinarySemaphore，用于退出中断，跳转至数据处理任务。然后创建了数据处理任务“DATA_PROC”，最后启动调度器。

在数据处理函数中等待 xSemaphoreTake() 返回值为 pdTRUE，表示接收完整一帧数据结束（以“n”结尾），接着通过 xQueueReceive() 取出写入队列的数据，直到“r”为止，最后串口打印输出接收数据和数据长度。

```
void vDataProcessTask(void *pvParameters)
{
    uint8_t rx_index =0;
    while (1)
    {
        // 等待二值信号量（表示数据接收完成）
        if (xSemaphoreTake(xBinarySemaphore, portMAX_DELAY) == pdTRUE)
        {
            //获取消息队列中的数据到Buffer 中直至获取到\n
            do{
                if(xQueueReceive(xUartRxQueue, &uart_rx_buffer[rx_index], portMAX_DELAY) ==
pdTRUE) {
                // 存储接收到的字节
                    rx_index++;
                }
                else
                    vTaskDelay(10);
            }while(uart_rx_buffer[rx_index - 1] != '\r');
            uart_rx_buffer[rx_index - 1] = '\0';
            // 处理数据
            printf("接收到完整数据: %s\r\n", uart_rx_buffer);
            printf("数据长度: %d\r\n", rx_index-1);

            // 清空缓冲区
            rx_index = 0;
            memset(uart_rx_buffer, 0, RX_BUFFER_SIZE);
        }
    }
}
```

图 3-170 任务函数

在串口中断服务函数（ISR）中，当判断触发了接收中断事件，就将接收数据通过 xQueueSendFromISR() 函数写入队列（在中断服务函数中必须使用带 ISR 的函数），如果结束的数据为“n”，则表示完整的一帧数据接收完成，通过 xSemaphoreGiveFromISR() 函数通知数据处理任务。portYIELD_FROM_ISR() 函数实现任务切换。

```
void UART0_Handler(void)
{
    uint8_t ch;
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    // 处理接收中断
    if(UART_Get_Interrupt_Status(STAR_UART0,UART_RxInterrupt)==SET) //接收中断
    {
        /*读取接收到的数据*/
        ch = UART_ReceiveData(STAR_UART0);

        if(ch == '\n')
```

```
        {
            // 发送二值信号量，通知数据处理任务
            xSemaphoreGiveFromISR(xBinarySemaphore, &xHigherPriorityTaskWoken);
        }
        else
        {
            // 将数据发送到队列
            xQueueSendFromISR(xUartRxQueue, &ch, &xHigherPriorityTaskWoken);
        }

        /*清除中断*/
        UART_Interrupt_Clear(STAR_UART0, UART_RxInterrupt);
    }

    // 如果有更高优先级任务就绪，进行任务切换
    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```

图 3-171 串口中断服务函数

3.24.3 下载验证

多任务工程验证

程序下载后，我们可以看到 LED2 1s 点亮 1 次，LED3 2s 点亮一次，同时串口打印任务 2 中的打印信息。

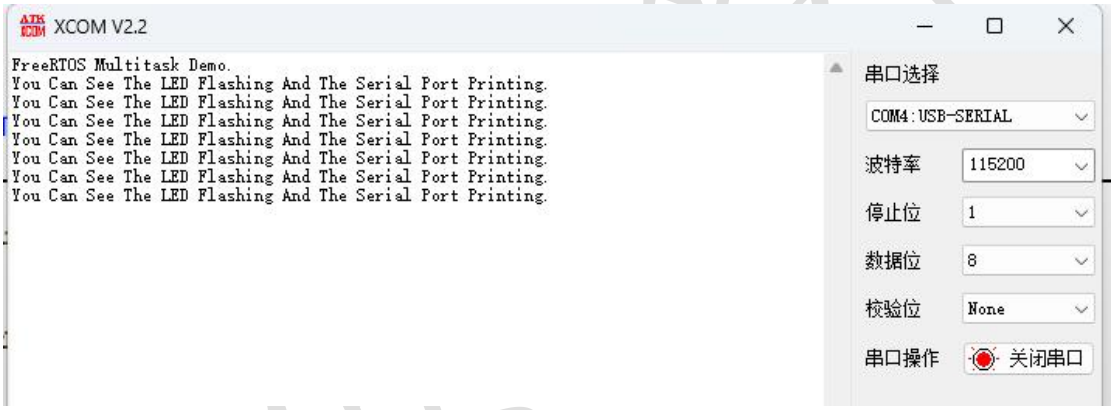


图 3-172 多任务工程串口打印输出

任务间通信工程验证

程序下载后，我们可以看到串口连续打印任务“Receiver”的打印信息，即写入队列中的数据，表明从任务“Sender1”和“Sender2”写入队列中的数据成功传输到了任务“Receiver”中，实现了任务间的通信。

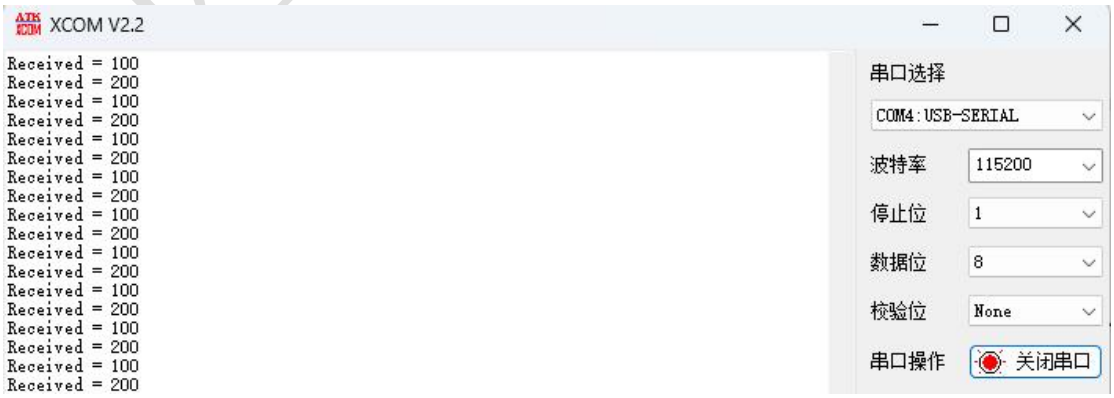


图 3-173 串口打印输出

中断处理工程验证

程序下载后，我们可以看到串口打印起始信息，当我们发送数据后（发送的数据此处勾选了“发送新行”，即数据以“r\n”结尾，在接收处理中，使用了“n”用于判断接收完成，“r”用于判断队列数据读出完成），串口将打印我们发送的数据和数据长度。



图 3-174 串口打印输出

文档修订历史

日期	版本	修订内容
2024.07	1.0	1.首次发布。
2025.11	3.0	1. 例程所用 IP 版本已更新，相应 FPGA 工程亦同步更新； 2. 新增例程章节 3.16 至 3.21。
2025.12	3.1	1. 新增例程章节 3.22 至 3.24。 2. 部分语句做了表述优化，提升了可读性。

版权所有©2025西安智多晶微电子有限公司
未经本公司书面许可，任何单位和个人都不得擅自摘抄、复制、翻译本文档内容的部分或全部，并不得以任何形式传播。
免责声明
本文档并未授予任何知识产权的许可，并未以明示或暗示，或以禁止发言或其它方式授予任何知识产权许可。除西安智多晶在其产品的销售条款和条件中声明的责任之外，西安智多晶概不承担任何法律或非法律责任。西安智多晶对西安智多晶产品的销售和/或使用不作任何明示或暗示的担保，包括对产品的特定用途适用性、适销性或对任何专利权、版权或其它知识产权的侵权责任等，均不作担保。西安智多晶对文档中包含的文字、图片及其它内容的准确性和完整性不承担任何法律或非法律责任，西安智多晶保留修改文档中任何内容的权利，恕不另行通知。西安智多晶不承诺对这些文档进行适时的更新。