高云 GW5A FPGA 在线升级实验说明

基于网络 TCP 协议

武汉芯路恒科技有限公司
教学产品项目组 编写

修订说明：

V1.0.0	2025-09-10	第一版本发行

联系我们

厂家名称：武汉芯路恒科技有限公司
联系地址：武汉东湖新技术开发区大学园路长城园路 8 号光谷精工科技园 A 栋 301-5 室
联系电话： 027-59265620
技术网站：www.corecourse.cn
微信公众号（提供在线客服）：小梅哥电子
在线商城（淘宝）：https://xiaomeige.taobao.com
在线商城（天猫）：https://xiaomeigesm.tmall.com/

目录

1　基于 M1 软核的以太网在线升级固件实验 1
1.1　系统组成架构 1
1.1.1　核心组件定义 1
1.1.2　组件设计必要性 1
1.1.3　组件交互逻辑 2
1.2　实验核心原理 2
1.2.1　硬件关键特质 2
1.2.2　固件升级基本流程 3
1.3　设计支持 IAP 的 FPGA 系统 3
1.3.1　配置 M1 软核 4
1.3.2　PLL IP 配置 9
1.3.3　引脚分配 11
1.3.4　设置 Multi Boot 的地址 12
1.4　上下位机在 FPGA 固件更新中的交互机制与实现方案 12
1.4.1　上下位机交互流程 13
1.4.2　通信命令定义 14
1.4.3　开发工具与核心功能 15
1.5　编写 FPGA M1 软核数据传输与读写 C 程序 16
1.5.1　下位机核心代码模块说明 16
1.5.2　ITCM 初始化文件生成与配置 19
1.6　设计远程数据与指令传输测试上位机软件 22
1.6.1　上位机核心代码模块说明 22
1.7　实验操作流程 28
1.7.1　硬件连接 28
1.7.2　下载 IAP 工程 29
1.7.3　使用上位机软件更新 APP 工程 31
1.8　实验总结 36

1 基于 M1 软核的以太网在线升级固件实验

章节导读

本实验基于高云 ACG525 FPGA 开发板，以“FPGA+M1 软核”为核心硬件架构，通过以太网 TCP 协议实现固件的远程在线升级，无需依赖外部编程器即可完成从“IAP 固件部署→APP 固件传输→校验切换”的全流程。

传统 FPGA 固件升级需现场连接编程器、依赖本地操作，存在效率低、维护成本高、难以批量升级等痛点。本实验通过网络升级方案，可实现远程、批量、无缝更新，显著适配工业设备远程运维、分布式设备管理等场景需求。

实验围绕“IAP 固件开发→上下位机协同→硬件配置→实操验证”展开，核心利用 FPGA 重配置特性与 Flash 可编程能力，通过 uIP 协议栈实现网络通信，最终完成固件升级与功能验证。

1.1 系统组成架构

1.1.1 核心组件定义

实验系统由三大核心组件构成，各组件协同实现远程升级功能：

- IAP 固件：作为基础固件预先固化于 FPGA，由“FPGA 硬件逻辑 + M1 软核 C 程序” 组成 ——FPGA 逻辑负责硬件驱动与软核环境搭建，C 程序负责协议解析与 Flash 操作控制。
- 网络升级上位机程序：基于 Visual Studio 开发，通过 TCP 协议发送升级指令、传输固件数据，并完成校验结果反馈。
- 待升级 APP 固件：目标应用程序（如本实验的 “按键控制 LED” 程序），通过 IAP 固件写入 Flash 并触发加载。

1.1.2 组件设计必要性

1. IAP 固件的核心价值：作为 “上位机与 Flash 的桥梁”，解决了“FPGA 如何通过网络接收指令并操作存储” 的核心问题 ——FPGA 逻辑负责硬件驱动，M1 软核 C 程序负责协议解析，两者协同实现硬件控制与软件逻辑的解耦。
2. 上位机的设计意义：提供可视化、可操作的控制入口，避免直接操作

硬件的复杂性，同时通过校验机制保障升级可靠性。

3. APP 固件的角色定位：作为升级的目标对象，与 IAP 固件分离设计，可灵活替换不同应用功能，无需重新开发基础升级框架。

1.1.3 组件交互逻辑

各组件通过 “网络通信 + Flash 存储” 实现数据流转，交互关系如下：

1. 上位机通过以太网向 IAP 固件发送 TCP 指令（擦除、编程、校验）；
2. IAP 固件解析指令后操作外部 Flash，完成 APP 固件的写入与读取；
3. IAP 固件将操作结果（成功 / 失败）回传至上位机；
4. 触发 FPGA 重配置后，从 Flash 加载 APP 固件并运行。

系统的整体架构如下所示。

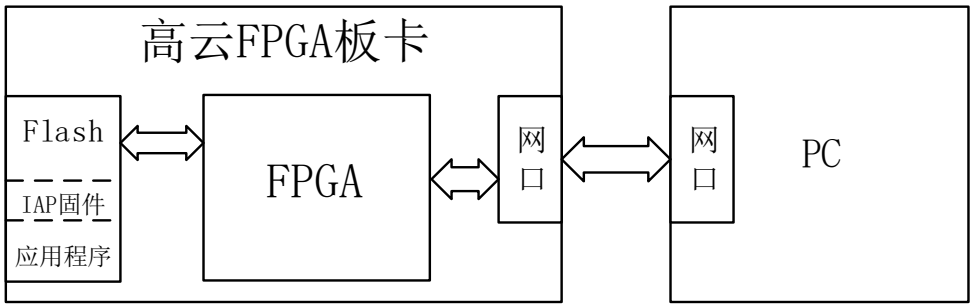

图 1-1 系统整体架构

1.2 实验核心原理

1.2.1 硬件关键特质

实验的实现依赖 FPGA 与外部 Flash 的两大核心特性，是远程升级的底层基础：

1. 特质一：FPGA 重配置的地址跳转特性

- 上电启动：FPGA 默认从外部 Flash 的 0x000000 地址读取并运行初始固件（IAP 程序）。
- 重配置触发：将 “Reconfig” 管脚拉低后拉高，FPGA 会根据当前运行固件内置的 “重配置地址”，从 Flash 对应地址加载新固件（APP 程序）。
- 交替跳转：若 IAP 与 APP 程序的重配置地址互指，可通过反复触发

Reconfig 实现两者交替加载。

该特性解决了 “如何在 FPGA 上切换不同固件” 的核心问题，无需断电重启即可完成程序切换，是实现 “在线升级” 的关键硬件支撑。

2. 特质二：外部 Flash 的可编程特质

FPGA 通过 SPI 接口可对外部 Flash 执行擦除、写入、读出三种操作，支持按“4K 块”擦除、按“256 字节页”编程。

Flash 作为非易失性存储介质，是固件的“载体”；其可编程能力使得 APP 固件可被远程写入更新，替代了传统 “拆板更换存储芯片” 的繁琐操作。

1.2.2 固件升级基本流程

基于上述特质，升级流程分为 “部署初始固件→传输校验目标固件→触发切换” 三步，具体如下：

1. 部署 IAP 固件（初始固件 A）

向外部 Flash 的 0 地址下载固件程序 A。程序 A 具备以下功能：接收以太网数据包，并将其解析为擦除、写入、读出三种操作指令及对应数据；执行指令完成 Flash 操作后，向 PC 端反馈操作结果。

2. 传输并校验 APP 固件（目标固件 B）

- 开发板上电后启动 IAP 程序，PC 通过网线与开发板建立 TCP 连接。
- 上位机发送 APP 固件数据，IAP 程序将其写入 Flash 的 “重配置地址”（如 0x2F0000）。
- IAP 程序从该地址读回数据并回传，上位机比对 “写入 - 读回” 数据，确认校验通过。

3. 触发固件切换

按下外部 Reconfig 按键（使管脚拉低），松开后管脚恢复拉高，此时 FPGA 将从程序 A 配置的重配置地址加载程序 B 并运行。若再次触发 Reconfig 按键，则从程序 B 配置的重配置地址重新加载程序 A。

1.3 设计支持 IAP 的 FPGA 系统

IAP（In-Application Programming，在应用编程）程序是 FPGA 在线升级系统的核心支撑组件。作为预先固化于 FPGA 的基础固件，它承担着连接远程控

制端（PC 上位机）与本地存储介质（外部 Flash）的桥梁作用，其架构与功能设计直接决定升级系统的稳定性与可靠性。

IAP 程序通常由两部分协同构成：

1. FPGA 逻辑部分：负责硬件资源的初始化与管理，包括以太网接口、外部 Flash 控制器等外设的驱动实现，同时构建并初始化 M1 软核处理器的运行环境，为上层软件提供硬件支撑。
2. 软核 C 程序部分：运行于 M1 软核之上，主要实现协议解析、指令处理与逻辑控制。通过以太网接收 PC 上位机发送的 TCP 数据包后，从中提取具体升级指令（如擦除外部 Flash 指定区域、写入新固件数据、校验存储内容完整性等），并将指令转化为对外部 Flash 的底层操作信号。

在实际固件升级流程中，IAP 程序的执行逻辑如下：首先对上位机的合法性及指令的完整性进行验证，防止非法操作或数据传输错误；验证通过后，依据指令完成外部 Flash 目标区域的擦除；随后分块接收目标程序数据并写入外部 Flash 的指定存储区域；最后通过 “读出 - 比对” 的校验机制，确认数据写入的准确性。

整个过程无需依赖外部编程器，真正实现了 FPGA 固件的远程、在线、无缝更新，显著提升了设备维护的灵活性与运维效率。

1.3.1 配置 M1 软核

M1 此处指 Cortex-M1 软核，它是 ARM 公司专门为 FPGA 开发场景推出的 32 位处理器内核，具备低功耗、高性价比的特性，可灵活集成到 FPGA 中实现嵌入式控制功能。

相较于其他支持 Cortex-M1 软核的 FPGA 平台，多数需要用户自行从 ARM 官网获取核授权文件、手动安装核驱动包，甚至要自行处理网表文件的导入与编译适配，整个过程操作繁琐且对用户技术门槛要求较高。而高云 FPGA 将 Cortex-M1 软核预先封装为可直接调用的标准化 IP 核，并集成在其官方开发软件中，用户无需额外进行授权申请、文件导入等复杂操作，可直接在开发环境中通过图形化界面调用该 IP，按需配置时钟频率、中断接口、存储容量等参数，极大简化了软核集成流程，提升了开发效率。

FPGA 程序的核心是调用 Cortex-M1 软核，M1 软核（Cortex-M1）是 FPGA 程序的 “大脑”，负责运行 C 程序解析指令，具体配置步骤如下：

进入配置界面：点击软件界面的  图标，进入 IP Core Generator 界面搜索

“M1”即可找到目标 IP 核，如下所示：

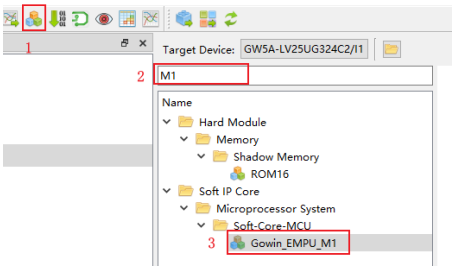

图 1-2 搜索找到 M1 软核

双击 IP 核进入 Cortex-M1 配置界面，如下所示：

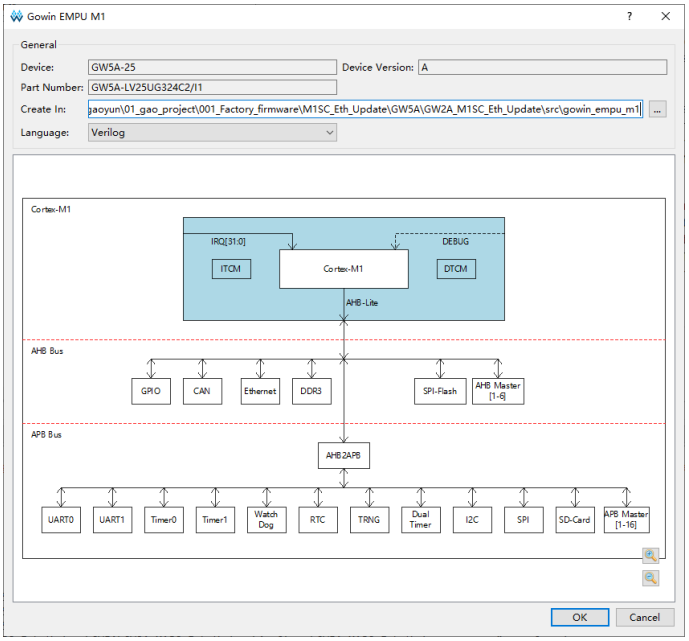

图 1-3 Cortex-M1 配置界面

进一步打开内核系统配置选项，包含通用配置、调试配置和存储配置三个部分，如下所示。

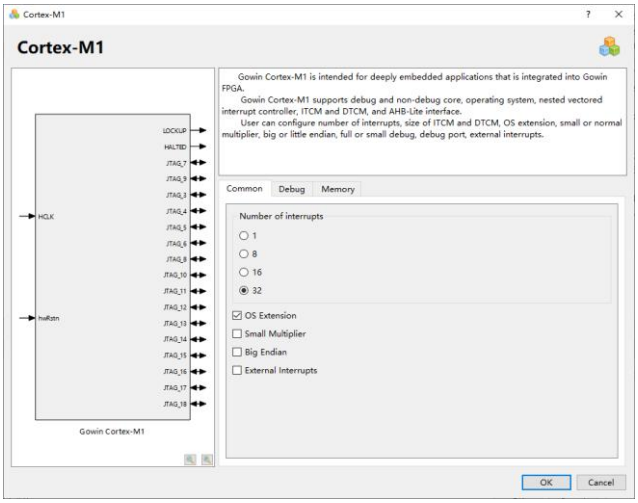

图 1-4 Cortex-M1 内核系统配置选项

（1）通用配置

通用配置选项可设置中断数量、操作系统扩展、乘法器模式、数据存储格式及外部中断。本实验采用默认配置，无需修改。

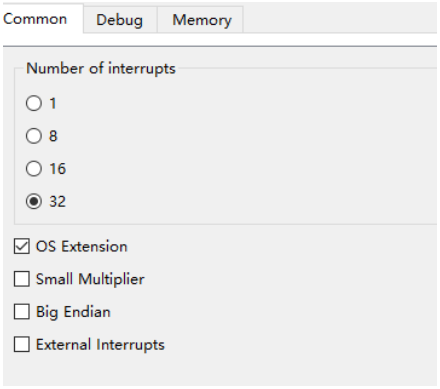

图 1-5 Cortex-M1 通用配置选项

（2）调试配置

调试配置需设置是否开启调试、调试接口及调试模式。此处将调试接口选择为“Serial Wire”。相比 JTAG 接口，引脚更少（仅 2 根），节省 FPGA 管脚资源，适配小型开发板设计。

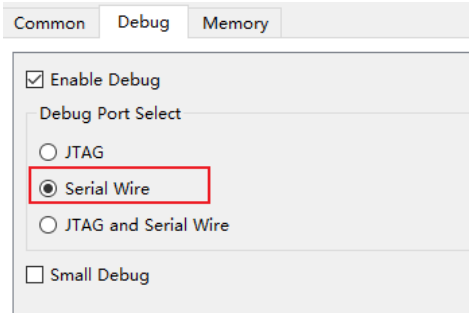

图 1-6 配置调试选项

（3）存储配置

存储配置用于设置 ITCM（指令传输总线）和 DTCM（数据传输总线）。

- ITCM：用于存放关键代码（如中断处理程序），支持 Internal Instruction Memory（片内 Block RAM，起始地址 0x00000000）或 External Instruction Memory（如 SPI-Flash，起始地址 0x00000000）。本实验选择 Internal Instruction Memory，默认容量 32KB；勾选“ITCM Initialization”，并在“ITCM Initialization Path”中预留 ITCM 初始值文件路径（后续通过 MDK 软件生成 C 代码对应的 ITCM 文件后导入）。

- DTCM：用于存放频繁访问的数据（如堆栈、全局变量），采用默认配置。

将代码载入 ITCM 的两种方法：

- 使用 gcc “属性标签”，为指定代码赋予 “ITCM” 属性；
- 将.c 源文件后缀改为.itcm.c，直接编译为 ITCM 运行的目标文件（本实验采用此方法，具体生成方式见后续 MDK 软件操作）。

DTCM 我们按照默认的配置就可以，配置界面如下所示。

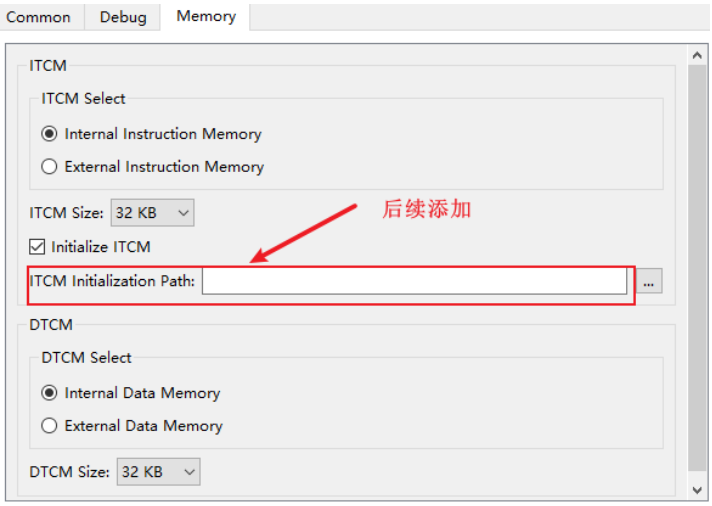

图 1-7 存储配置设置

（4）外设配置

需使能的外设包括 GPIO、UART0、Timer0、Ethernet 及 SPI-Flash（低版本软件可能无此选项，需更新），双击对应外设即可配置（使能）。其中 Ethernet 配置需重点注意：

- 勾选 “Enable Ethernet” 使能功能；
- “Interface” 根据硬件选择，ACG525 开发板需选择 “RGMII”；支持 1000Mbps，比 MII 接口（100Mbps）传输更快，缩短固件传输时间。
- “RGMII Input Delay”：ACG525 开发板默认设为 100，AC201-GW2A18 开发板设为 30 时网口更稳定；其它开发板可根据需求自行修改尝试
- “MIIM Clock Divider” 默认设为 20；
- 若 “Interface” 选择 “RGMII” 或 “GMII”，需确保端口 GTX_CLK 接入 125MHz 时钟。

其相关配置如下所示。

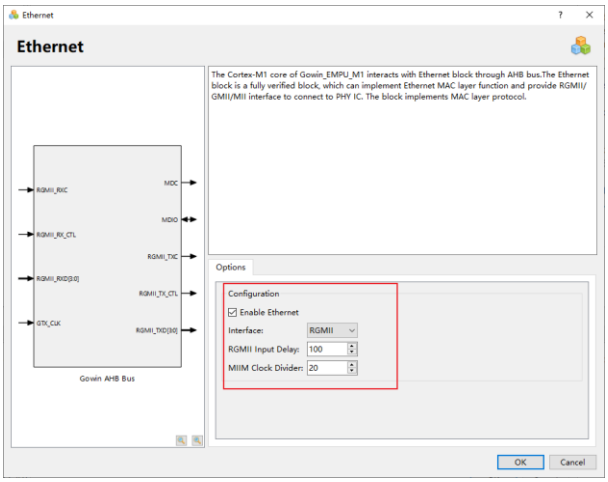

图 1-8Ethernet 配置

外设配置完成后，界面中已使能的模块会显示为绿色如下所示，便于确认状态。

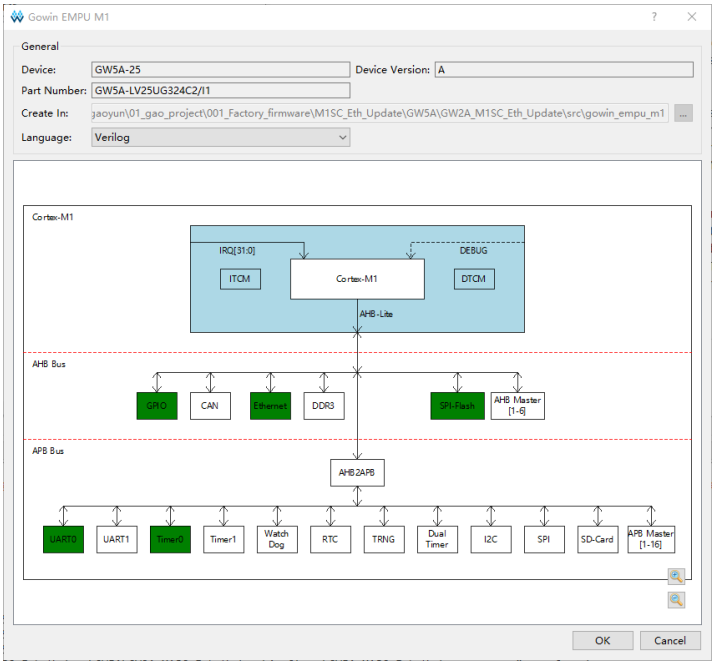

图 1-9 外设配置完成

配置完成后，在顶层模块中调用 M1 核的相关代码如下所示：

```
Gowin_EMPU_M1_Top u_Gowin_EMPU_M1_Top(
    .LOCKUP             (LOCKUP             ),//output LOCKUP
    .HALTED             (HALTED             ),//output HALTED
    .GPIO               (GPIO               ),//inout [15:0] GPIO
    .JTAG_7             (SWDIO              ),//inout JTAG_7
    .JTAG_9             (SWCLK              ),//inout JTAG_9
    .UART0RXD           (UART0RXD           ),//input UART0RXD
```

```
    .UART0TXD           (UART0TXD           ),//output UART0TXD
    .TIMER0EXTIN        (TIMER0EXTIN        ),//input TIMER0EXTIN
    .RGMII_TXC          (RGMII_TXC          ),//output RGMII_TXC
    .RGMII_TX_CTL       (RGMII_TX_CTL       ),//output RGMII_TX_CTL
    .RGMII_TXD          (RGMII_TXD          ),//output [3:0] RGMII_TXD
    .RGMII_RXC          (RGMII_RXC          ),//input RGMII_RXC
    .RGMII_RX_CTL       (RGMII_RX_CTL       ),//input RGMII_RX_CTL
    .RGMII_RXD          (RGMII_RXD          ),//input [3:0] RGMII_RXD
    .GTX_CLK            (Clk_125M            ),//input GTX_CLK
    .MDC                (MDC                ),//output MDC
    .MDIO               (MDIO               ),//inout MDIO
    .FLASH_SPI_HOLDN    (FLASH_SPI_HOLDN    ),//inout FLASH_SPI_HOLDN
    .FLASH_SPI_CSN      (FLASH_SPI_CSN      ),//inout FLASH_SPI_CSN
    .FLASH_SPI_MISO     (FLASH_SPI_MISO     ),//inout FLASH_SPI_MISO
    .FLASH_SPI_MOSI     (FLASH_SPI_MOSI     ),//inout FLASH_SPI_MOSI
    .FLASH_SPI_WPN      (FLASH_SPI_WPN      ),//inout FLASH_SPI_WPN
    .FLASH_SPI_CLK      (FLASH_SPI_CLK      ),//inout FLASH_SPI_CLK
    .HCLK               (Clk_50M               ),//input HCLK
    .hwRstn             (Rstn             ) //input hwRstn
);
```

1.3.2 PLL IP 配置

如前文所述，以太网接口若选择“RGMII”或“GMII”模式，其 GTX_CLK 端口必须接入 125MHz 时钟信号，若时钟频率错误，将导致网络通信中断或数据传输错误。因此，需在设计中添加 PLL（锁相环）模块，通过配置使其生成并输出 125MHz 时钟，PLL 的配置如下所示：

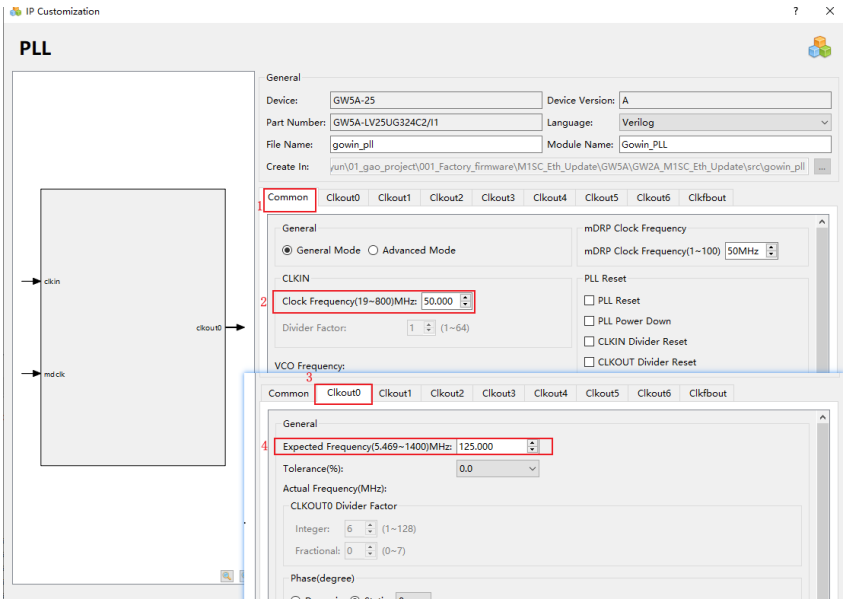

图 1-10 PLL IP 配置

完成 PLL 模块配置后，需在顶层模块中调用该 PLL IP 核；同时，需对以太网复位引脚进行处理，将其置为高电平状态。最终完善后的顶层模块代码如下所示：

```
module M1SC_Eth_Update_Top(
    output  LOCKUP,
    output  HALTED,
    inout   [15:0] GPIO,
    inout   SWDIO,
    inout   SWCLK,
    input   UART0RXD,
    output  UART0TXD,
    input   TIMER0EXTIN,
    output  RGMII_TXC,
    output  RGMII_TX_CTL,
    output  [3:0] RGMII_TXD,
    input   RGMII_RXC,
    input   RGMII_RX_CTL,
    input   [3:0] RGMII_RXD,
    output  ETH_RESENT,
    output  MDC,
    inout   MDIO,
    inout   FLASH_SPI_HOLDN,
    inout   FLASH_SPI_CSN,
    inout   FLASH_SPI_MISO,
    inout   FLASH_SPI_MOSI,
    inout   FLASH_SPI_WPN,
    inout   FLASH_SPI_CLK,
    input   Clk_50M,
    input   Rstn
);

wire Clk_125M;

Gowin_PLL Gowin_PLL(
    .clkin(Clk_50M), //input  clkin
    .clkout0(Clk_125M), //output  clkout0
    .mdclk(Clk_50M) //input  mdclk
);

assign ETH_RESENT = 1'd1;

Gowin_EMPU_M1_Top u_Gowin_EMPU_M1_Top(
    .LOCKUP             (LOCKUP             ),//output LOCKUP
    .HALTED             (HALTED             ),//output HALTED
    .GPIO               (GPIO               ),//inout [15:0] GPIO
    .JTAG_7             (SWDIO              ),//inout JTAG_7
```

```
    .JTAG_9             (SWCLK              ),//inout JTAG_9
    .UART0RXD           (UART0RXD           ),//input UART0RXD
    .UART0TXD           (UART0TXD           ),//output UART0TXD
    .TIMER0EXTIN        (TIMER0EXTIN        ),//input TIMER0EXTIN
    .RGMII_TXC          (RGMII_TXC          ),//output RGMII_TXC
    .RGMII_TX_CTL       (RGMII_TX_CTL       ),//output RGMII_TX_CTL
    .RGMII_TXD          (RGMII_TXD          ),//output [3:0] RGMII_TXD
    .RGMII_RXC          (RGMII_RXC          ),//input RGMII_RXC
    .RGMII_RX_CTL       (RGMII_RX_CTL       ),//input RGMII_RX_CTL
    .RGMII_RXD          (RGMII_RXD          ),//input [3:0] RGMII_RXD
    .GTX_CLK            (Clk_125M            ),//input GTX_CLK
    .MDC                (MDC                ),//output MDC
    .MDIO               (MDIO               ),//inout MDIO
    .FLASH_SPI_HOLDN    (FLASH_SPI_HOLDN    ),//inout FLASH_SPI_HOLDN
    .FLASH_SPI_CSN      (FLASH_SPI_CSN      ),//inout FLASH_SPI_CSN
    .FLASH_SPI_MISO     (FLASH_SPI_MISO     ),//inout FLASH_SPI_MISO
    .FLASH_SPI_MOSI     (FLASH_SPI_MOSI     ),//inout FLASH_SPI_MOSI
    .FLASH_SPI_WPN      (FLASH_SPI_WPN      ),//inout FLASH_SPI_WPN
    .FLASH_SPI_CLK      (FLASH_SPI_CLK      ),//inout FLASH_SPI_CLK
    .HCLK               (Clk_50M               ),//input HCLK
    .hwRstn             (Rstn             ) //input hwRstn
);

endmodule
```


1.3.3 引脚分配

根据开发板硬件接口，将 FPGA 信号与实际引脚对应，关键引脚分配如下所示：

表 1-1 引脚分配表

Signial Name	Pin NO	Signial Name	Pin NO
RGMII_TX_CTL	C2	Clk_50M	T9
RGMII_TXD[3]	F1	UART0TXD	V8
RGMII_TXD[2]	F2	UART0RXD	U8
RGMII_TXD[1]	D1	SWDIO	C17
RGMII_TXD[0]	D2	SWCLK	C18
RGMII_TXC	C1	Rstn	C15
RGMII_RX_CTL	E1	FLASH_SPI_MOSI	T13
RGMII_RXD[3]	F5	FLASH_SPI_MISO	R13
RGMII_RXD[2]	F6	FLASH_SPI_CSN	V3
RGMII_RXD[1]	G1	FLASH_SPI_CLK	R15
RGMII_RXD[0]	G3	ETH_RESENT	G6
RGMII_RXC	L5		
MDIO	E4		
MDC	D3		

1.3.4 设置 Multi Boot 的地址

为实现 IAP 到 APP 的跳转，需配置 “Multi Boot 地址”（即 APP 在 Flash 中的起始地址）。

1. 进入配置界面：在 Gowin 软件的 “IAP 工程配置” 中找到 “Multi Boot” 选项。
2. 参数设置：将地址设为 0x2F0000，与后续 APP 下载地址保持一致。IAP 固件大小约为 0x2F0000（2944KB）以内，0x2F0000 之后的空间足够存放 APP 固件（一般 APP 固件大小不超过 1MB）。Flash 擦除以 4K 为单位，0x2F0000 是 4K 的整数倍（0x2F0000÷4096=1888），避免擦除时影响其他数据。
3. 生效操作：设置完成后重新综合 FPGA 工程，确保配置生效。

配置界面如下所示。

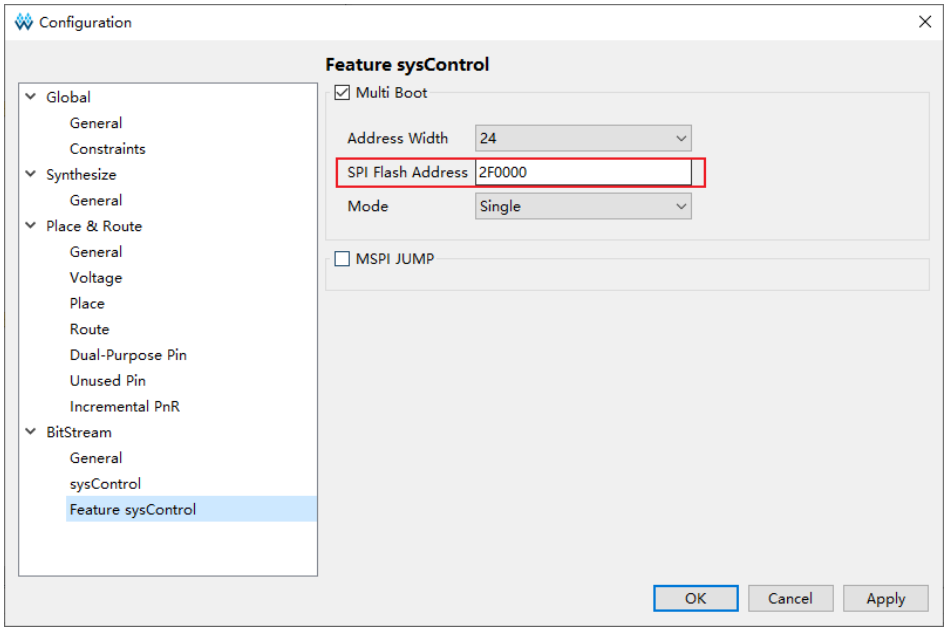

图 1-11 设置 Multi Boot 的地址

1.4 上下位机在 FPGA 固件更新中的交互机制与实现方案

在前文中我们已完成 IAP 程序中 FPGA 底层硬件部分的实现说明，而在推进 M1 软核与上位机协同开发前，需先构建二者交互的技术框架。以下内容将系统阐述 FPGA 固件更新场景下，上位机与下位机的全流程交互机制、标准化

通信规范及工程化实现方案：

1. 通过明确 "网络连接→Flash 擦除→固件编程→数据校验" 的闭环时序逻辑，细化各环节的指令交互规则与反馈机制；
2. 基于 "帧头 + 地址 + 数据" 的统一格式，定义擦除、编程、校验及状态反馈等核心命令的编码规范；
3. 结合工程需求，对比通信方式选型依据，明确 TCP 网口的技术优势。
4. 详细说明上位机（Visual Studio/MFC 开发）与下位机（MDK 开发）的工具链配置及核心功能模块划分。
5. 该方案不仅为后续 M1 软核与上位机的协同开发提供了清晰的技术路径，更从通信可靠性、操作规范性、开发高效性角度，为 FPGA 固件远程更新系统的稳定运行提供了全方位支撑。

1.4.1 上下位机交互流程

交互遵循 “连接→擦除→编程→校验” 的时序逻辑，具体步骤如下所示：

（1）下位机初始化：启动后完成以太网接口初始化，配置板卡 IP 地址等网络参数，进入等待连接状态。

（2）建立网络连接：上位机通过网线与开发板建立 TCP 连接，确认通信链路通畅。

（3）发送擦除指令：上位机发送帧头为 0xCC 的擦除指令，指定 Flash 待擦除区域的起始地址。

（4）执行擦除并反馈：下位机解析指令后执行 Flash 擦除操作，完成后发送帧头为 0xAA 的正确反馈；若指令错误，则发送帧头为 0x55 的错误反馈。

（5）发送编程数据：上位机收到正确反馈后，开始传输目标固件数据。以 4KB 为单位划分固件，每 4KB 拆分为 16 个 256 字节数据包，依次发送帧头为 0xBC 的编程指令（含地址与对应数据包）。若最后一个数据包不足 256 字节，不足部分填充 0xFF。

（6）执行编程并反馈：下位机接收数据包后写入指定 Flash 地址，每完成一次写入即反馈 0xAA，直至所有 4KB 数据传输完毕。

（7）发送校验指令：上位机发送帧头为 0xCD 的校验指令，指定需校验的 Flash 地址。

（8）校验并反馈结果：下位机从指定地址读出数据并回传至上位机，上位机将回传数据与本地源文件比对。若一致，则提示 “升级成功”；若不一致，则提示 “升级失败”。

具体的流程图如下所示。

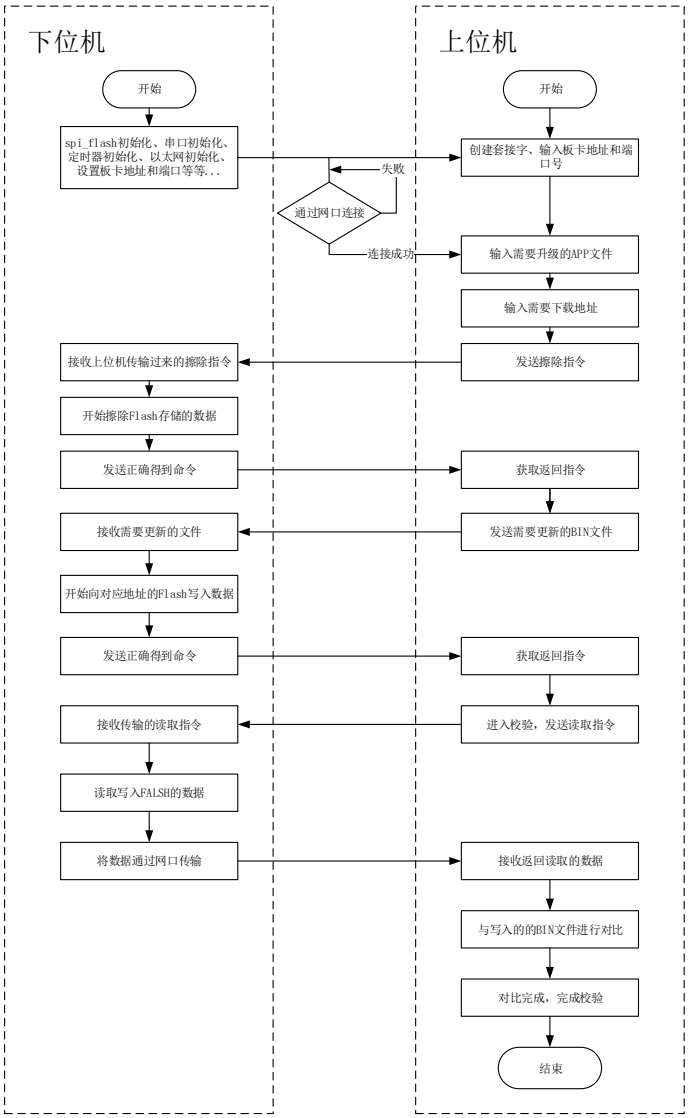

图 1-12 上位机和下位机交互传输流程图

1.4.2 通信命令定义

上下位机通过自定义 命令帧交互，帧格式由 “帧头＋地址＋数据” 组成，具体定义如下表所示：

表 1-2 命令说明表

功能	帧头	地址部分（32 位，分 4 字节）				数据部分		
擦除	0xCC	Addr[31:24]	Addr[23:16]	Addr[15:8]	Addr[15:8]	无		
编程	0xBC	Addr[31:24]	Addr[23:16]	Addr[15:8]	Addr[15:8]	data0	...	data255

校验	0xCD	Addr[31:24]	Addr[23:16]	Addr[15:8]	Addr[15:8]	data0	...	data255
下位机正确得到命令进行反馈	0xAA	Addr[31:24]	Addr[23:16]	Addr[15:8]	Addr[15:8]	无		
下位机得到错误命令进行反馈	0x55	0x55	无			无		

1.4.3 开发工具与核心功能

在 FPGA 固件更新及指令交互的整体流程中，上位机程序是实现 “用户操作 - 数据传输 - 下位机响应” 闭环的关键环节。其核心必要性在于：下位机（FPGA M1 软核）作为嵌入式设备，本身不具备直接接收用户输入、解析复杂固件文件或提供可视化操作界面的能力，因此需要通过上位机作为 “中间桥梁”，将用户需要更新的固件数据、控制指令等传递给下位机，并接收下位机返回的执行结果，实现双向交互。

实现上位机与下位机的通信方式有多种，包括 USB、串口、网口等。本方案选择网口（基于 TCP 协议）作为通信载体，主要因其具备显著优势：一是传输速率高，能满足固件文件等大容量数据的快速传输需求；二是支持远距离通信，可灵活适应不同的部署场景；三是兼容性强，易于与各类网络设备组网，便于后期系统扩展；四是 TCP 协议本身具备可靠传输机制，能有效保障数据传输的完整性和准确性，降低通信错误率。

基于上述需求，需要开发一款运行在 PC 端的上位机软件，其核心目标是通过网络与 FPGA 建立稳定连接，完成固件文件解析、指令编码生成、数据校验发送及结果反馈展示等功能。开发此类上位机的技术路径较为灵活，主流方案包括基于 MFC、QT、C# WinForm 等框架实现，核心需完成 TCP 套接字通信逻辑和文件读写操作。本方案中以 Visual Studio 结合 MFC 框架为例进行开发演示，主要考虑其在 Windows 平台下的成熟性和易用性，读者可根据实际需求，参考相同的功能逻辑，选择其他开发工具（如 QT）实现同等功能的上位机程序。

我们进行软件开发需要的开发工具和核心功能说明如下表所示

表 1-3 开发工具与核心功能说明表

程序类型	开发工具	运行载体	核心功能
上位机程序	Visual Studio	PC	TCP 套接字通信、固件文件解析、指令编码、数据校验、结果反馈
下位机程序	MDK	FPGA M1 软核	外设初始化、uIP 协议栈运行、指令解析、Flash 驱动、操作结果回传

1.5 编写 FPGA M1 软核数据传输与读写 C 程序

FPGA 中的 M1 软核作为嵌入式处理核心，其运行的 C 程序是实现下位机功能的 “灵魂”—— 它不仅是连接上位机指令与 FPGA 硬件操作的桥梁，更是实现数据可靠传输、Flash 精准读写等核心功能的直接载体。这些程序需要兼顾嵌入式系统的资源约束（如存储容量、运行效率）与功能完整性（如协议兼容性、操作安全性），通过模块化设计将复杂任务拆解为可独立维护的功能单元，最终实现与上位机的稳定通信及对 FPGA 外设的精准控制。

1.5.1 下位机核心代码模块说明

下位机程序基于 MDK 软件开发，主要实现外设初始化、uIP 协议栈运行、指令解析、Flash 驱动、操作结果回传等功能。以下对关键代码模块及核心流程进行说明（完整源码请参考对应工程文件）。

1. 系统初始化配置

首先完成外设及网络参数初始化，核心配置如下：

- 本机 IP 地址：192.168.31.78（需与上位机 IP 处于同一网段）；
- 网关地址：192.168.31.89；
- 通信端口号：1234（与上位机保持一致）。

初始化代码如下所示：

main.c | Eth_Update.c | tapdev.c | GOWIN_M1_ethernet.c | GOWIN_M1_ethernet.h

```
33  uint8_t  rx_data[1536];
34  uint32_t rx_data_length = 0;
35  uint32_t miim_rd_data = 0;
36
37  #define BUF ((struct uip_eth_hdr *)&uip_buf[0])
38
39  void Delay_ms(__IO uint32_t delay_ms);
40
41  int main()
42  {
43    SystemInit();       //Initializes system clock
44    UartInit();         //Initializes UART0
45    spi_flash_init();       //Initializes spi-flash
46    tapdev_init();        //Initializes Ethernet
47    NVIC_EnableIRQ(ENT_IRQn); //Enable ethernet interrupt
48
49    init_timer();       //Initializes TImer0
50    TIMER_EnableIRQ(TIMER0);  //Enable Timer0 interrupt
51    TIMER_StartTimer(TIMER0); //Start_RTC Timer0
52    NVIC_EnableIRQ(TIMER0_IRQn);
53
54    uip_init();         //Initilizes uip
55
56    uip_ipaddr_t ipaddr;
57
58    //Set IP
59    uip_ipaddr(ipaddr,192,168,31,78);
60    uip_sethostaddr(ipaddr);
61
62    //Set Rute
63    uip_ipaddr(ipaddr,192,168,31,89);
64    uip_setdraddr(ipaddr);
65    uip_ipaddr(ipaddr,255,255,255,0);
66    uip_setnetmask(ipaddr);
67
68    //Delay and wait stable
69    Delay_ms(10);
70
71    //Change spi-flash from download mode to memory mode
72    change_mode_spi_flash();
73
74    uip_listen(HTONS(1234));
75
```

图 1-13 进行初始化及 IP 地址等相关设置

2. uIP 协议栈核心处理

本次实验我们使用的是 uIP 协议栈，uIP 是轻量级 TCP/IP 协议栈，代码量仅几千行，RAM 占用小于 10KB，适合 FPGA M1 软核等资源受限场景（传统 TCP/IP 协议栈如 LWIP 占用资源更高）、实现网络通信，核心依赖 uip_polling() 函数处理网络事件，其主要职责包括：

- 从网络硬件读取并解析 IP/ARP 数据包；
- 维护 TCP 连接（如超时重传、窗口更新）；
- 更新 ARP 缓存表（实现 IP 与 MAC 地址映射）

先初始化两个定时器：

- periodic_timer：每 0.5 秒触发一次，用于 TCP 连接的周期性维护；
- arp_timer：每 10 秒触发一次，用于 ARP 缓存表更新。

随后通过循环读取网络数据包，依次处理 IP、ARP 协议及超时等事件，具

体代码如下所示：

```
//uip事件处理函数
//必须将该函数插入用户主循环,循环调用.
void uip_polling(void)
{
  uint8_t i;
  static struct timer periodic_timer, arp_timer;
  static uint8_t timer_ok=0;

  if(timer_ok==0)                                 //仅初始化一次
  {
    timer_ok = 1;
    timer_set(&periodic_timer,CLOCK_SECOND/2);    //创建1个0.5秒的定时器
    timer_set(&arp_timer,CLOCK_SECOND*10);        //创建1个10秒的定时器
  }

  uip_len=tapdev_read();                          //从网络设备读取一个IP包,得到数据长度.uip_len在uip.c中定义

  if(uip_len>0)                                   //有数据
  {
    //处理IP数据包(只有校验通过的IP包才会被接收)
    if(BUF->type == htons(UIP_ETHTYPE_IP))        //是否是IP包?
    {
      uip_arp_ipin();                             //去除以太网头结构，更新ARP表
      uip_input();                                //IP包处理

      //当上面的函数执行后，如果需要发送数据，则全局变量 uip_len > 0
      //需要发送的数据在uip_buf, 长度是uip_len  (这是2个全局变量)
      if(uip_len>0)                               //需要回应数据
      {
        uip_arp_out();                            //加以太网头结构，在主动连接时可能要构造ARP请求
        tapdev_send();                            //发送数据到以太网
      }
    }
    else if (BUF->type==htons(UIP_ETHTYPE_ARP))//处理arp报文,是否是ARP请求包?
    {
      uip_arp_arpin();

      //当上面的函数执行后，如果需要发送数据，则全局变量uip_len>0
      //需要发送的数据在uip_buf, 长度是uip_len(这是2个全局变量)
      if(uip_len>0)
      {
        tapdev_send();                          //需要发送数据,则通过tapdev_send发送
      }
    }
  }
  else if(timer_expired(&periodic_timer))     //0.5秒定时器超时
  {
    timer_reset(&periodic_timer);             //复位0.5秒定时器

    //轮流处理每个TCP连接, UIP_CONNS缺省是40个
    for(i=0;i<UIP_CONNS;i++)
    {
      uip_periodic(i);                        //处理TCP通信事件

      //当上面的函数执行后，如果需要发送数据，则全局变量uip_len>0
      //需要发送的数据在uip_buf, 长度是uip_len (这是2个全局变量)
      if(uip_len>0)
      {
        uip_arp_out();                        //加以太网头结构，在主动连接时可能要构造ARP请求
        tapdev_send();                        //发送数据到以太网
      }
    }
    //每隔10秒调用1次ARP定时器函数 用于定期ARP处理,ARP表10秒更新一次，旧的条目会被抛弃
    if(timer_expired(&arp_timer))
    {
      timer_reset(&arp_timer);
      uip_arp_timer();
    }
  }
```

图 1-14 uip_polling()函数实现

3. 上位机指令解析与执行

指令解析与 Flash 操作逻辑在 Eth_Update.c 文件中实现，核心是根据表 1-2 的命令格式匹配帧头，执行对应操作并反馈结果：

（1）接收上位机发送的 TCP 数据包；

（2）提取帧头（0xCC/0xBC/0xCD），判断指令类型；

（3）执行对应操作：帧头 0xCC 触发 Flash 擦除、0xBC 触发数据写入、0xCD 触发数据读取；

（4）操作完成后，向上位机返回反馈帧（0xAA 表示成功，0x55 表示失败）。

部分代码如下所示：

```
//    memset(Pack_Buf,0x00,1029);
    memcpy(Pack_Buf,uip_appdata,uip_len);
//    Pack_Len += uip_len;

    ECHO("Receive data (%d Bytes) is %s\r\n", uip_len, (char *)Pack_Buf);

//    do{
      //解析指令并处理
      switch(Pack_Buf[0]) {

      /* 擦除4K区域 0xCC Address_Byte0 Address_Byte1 Address_Byte2 Address_Byte3 */
      case 0xCC: //擦除
        //先判断完整性
          Addr = (Pack_Buf[4] << 24) | (Pack_Buf[3] << 16) | (Pack_Buf[2] << 8) | (Pack_Buf[1]);
          spi_flash_4ksector_erase(Addr);
          Pack_Buf[0] = 0xAA;
          uip_send(Pack_Buf,5); //uip_send(Return_Buf,1029);
        break;

      /* 编程Length字节  0xBC Address_Byte0 Address_Byte1 Address_Byte2 Address_Byte3 Data0 Data1 ... */
      case 0xBC:  //编程
          Addr = (Pack_Buf[4] << 24) | (Pack_Buf[3] << 16) | (Pack_Buf[2] << 8) | (Pack_Buf[1]);
          spi_flash_page_program(256, Addr, pData);
          Pack_Buf[0] = 0xAA;
          uip_send(Pack_Buf,5);
        break;

      /* 读取Length字节  0xCD Address_Byte0 Address_Byte1 Address_Byte2 Address_Byte3 */
      case 0xCD:  //读取1K数据
        //先判断完整性
          Addr = (Pack_Buf[4] << 24) | (Pack_Buf[3] << 16) | (Pack_Buf[2] << 8) | (Pack_Buf[1]);
          Return_Buf[0] = 0xAA;
          memcpy(Return_Buf + 1,&Addr,4);
          while(spi_flash_read(256, READ_CMD, Addr, Return_Buf + 5) == -2);// + 256*i);
          //返回 0xAA Address_8Bits_High Address_8Bits_Low Data0 Data1 ...
          uip_send(Return_Buf,261); //uip_send(Return_Buf,1029);
        break;

      //无效指令 返回0x55 + 无效指令
      default:
        Return_Buf[0] = 0x55;
        Return_Buf[1] = Pack_Buf[0];
        uip_send(Return_Buf,2);
        printf("Invalid instruction: 0x%02X\n",Pack_Buf[0]);
        break;
      }

    }
}
```

图 1-15 指令解析并处理

1.5.2 ITCM 初始化文件生成与配置

如前文所述，M1 软核配置需导入 ITCM 初始化文件，具体生成与配置流程如下：

（1）工具准备

使用 make_hex 工具（存放在 ITCM 文件夹中）将.bin 可执行文件转换为 4 个十六进制初始化文件（itcm0、itcm1、itcm2、itcm3）。将该工具添加至 MDK 工程，如下所示。

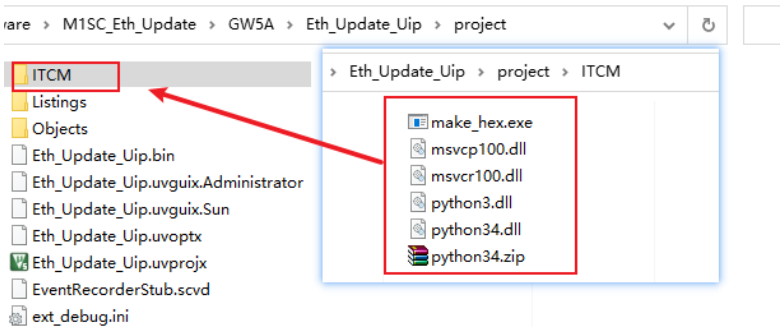

图 1-16 将 make_hex 工具放入 MDK 工程中

（2）配置外部工具

在 MDK 中配置 make_hex.exe 为外部工具，需依次执行两条命令：

- H:\keil\ARM\ARMCC\bin\fromelf.exe --bin -o Eth_Update_Uip.bin .\Objects\Eth_Update_Uip.axf（前面的 H:\keil\ARM\ARMCC\bin\fromelf.exe 需要根据自己软件安装的位置进行修改，将.axf 转换为.bin）
- ITCM/make_hex.exe Eth_Update_Uip.bin（将.bin 转换为 itcm0~itcm3）

配置界面如下所示：

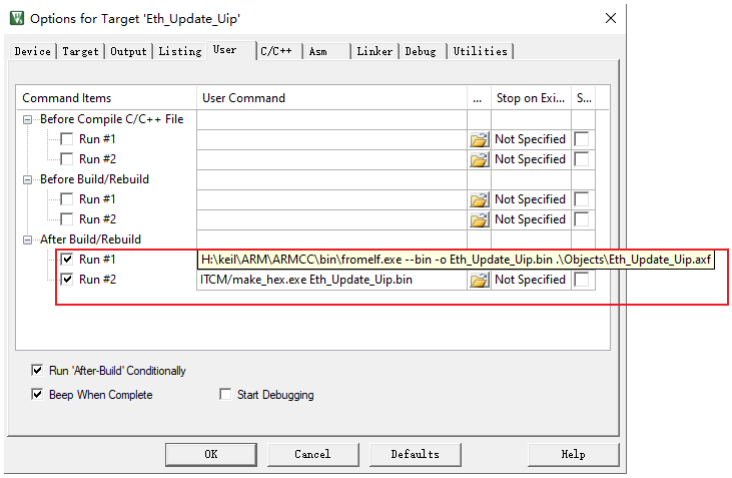

图 1-17 配置 make_hex.exe 作为外部工具

（3）生成 ITCM 文件

编译 MDK 工程，自动生成 itcm0~itcm3 四个初始化文件，如下图 所示。

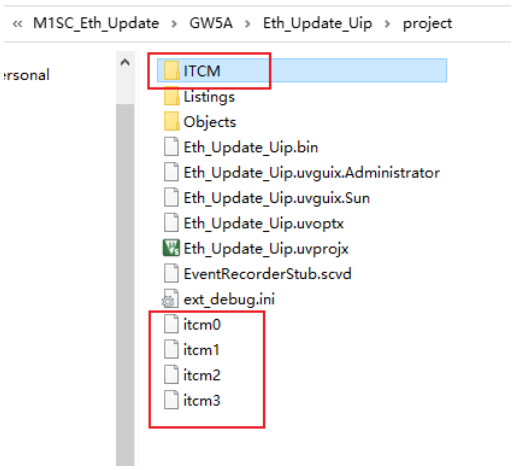

图 1-18 得到 itcm 文件

（4）导入 FPGA 工程

将生成的 4 个 ITCM 文件复制到 FPGA 工程的 BootLoader 文件夹中，如下图所示。

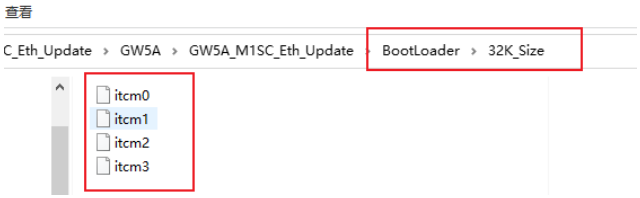

图 1-19 复制文件到 FPGA 工程中

（5）配置 M1 软核

重新打开 M1 软核配置界面，在 “ITCM Initialization Path” 中导入上述 4 个文件，完成后重新编译 FPGA 工程，如下图 所示。

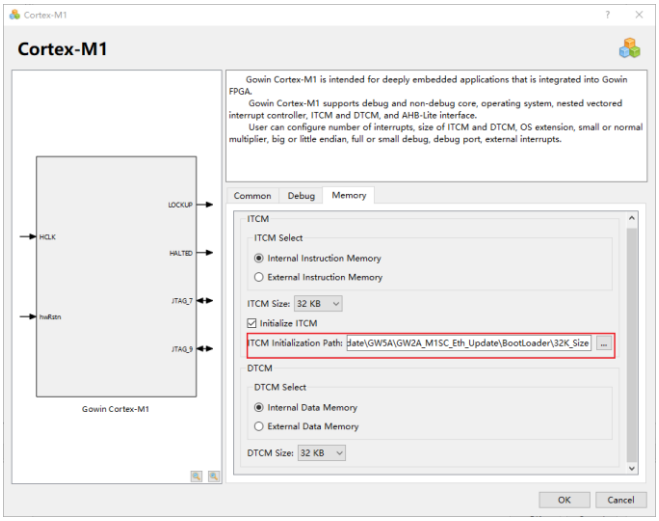

图 1-20 将 ITCM 文件写入 M1 软核中

1.6 设计远程数据与指令传输测试上位机软件

本节聚焦于远程数据与指令传输测试上位机软件的设计实现，作为连接用户操作与下位机（FPGA M1 软核）的核心交互载体，该上位机承担着固件更新全流程的控制与协调功能。

1.6.1 上位机核心代码模块说明

上位机程序基于 Visual Studio 开发，以下按功能流程对关键代码模块进行说明（完整源码请参考配套文件）。

1. 网络连接建立

上位机首先完成 TCP 套接字初始化，随后读取用户输入的板卡 IP 地址及端口号，发起网络连接。需重点注意：板卡 IP 地址与上位机（PC）IP 地址必须处于同一网段，否则连接失败。

本实验示例中：PC IP 地址为 192.168.31.79，板卡 IP 地址为 192.168.31.78，通信端口号统一为 1234。

建立连接的核心代码如下所示。

```
43    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
44        printf("创建Socket失败! \n");
45        exit(1);
46    }
47
48    //发送客户端连接请求
49    printf("# 请输入ip地址: \n");//提示输入ip地址
50    printf("-> ");
51    scanf("%s", ip_addr);//获取ip地址
52    printf("\n");   //换行
53
54    printf("# 请输入端口号: \n");//提示输入端口号
55    printf("-> ");
56    scanf("%s", port_num);//获取端口号
57    printf("\n");   //换行
58
59    struct sockaddr_in serveraddr;
60    socklen_t addrlen = sizeof(serveraddr);
61    serveraddr.sin_family = AF_INET;
62    serveraddr.sin_addr.s_addr = inet_addr(ip_addr);
63    serveraddr.sin_port = htons(atoi(port_num));
64
65    //超时设置
66    int nNetTimeout = 1000;//1秒
67    //发送时限
68    setsockopt(sockfd, SOL_SOCKET, SO_SNDTIMEO, (char *)&nNetTimeout, sizeof(int));
69    //接收时限
70    setsockopt(sockfd, SOL_SOCKET, SO_RCVTIMEO, (char *)&nNetTimeout, sizeof(int));
71
72    while(connect(sockfd, (struct sockaddr *)&serveraddr, addrlen) == -1)
73    {
74        printf("连接失败! \n");
75        Sleep(2000);
76    }
77
78    printf("连接成功! \n");
```

图 1-21 创建连接部分代码

2. 升级文件与参数准备

连接成功后，需完成两项关键准备工作：

- 加载升级文件：获取待升级 APP 固件的 bin 格式文件，作为数据传输的源文件，代码如下所示；

```
83          printf("\n开始升级！！！\n");
84          printf("# 请输入BIN文件名：\n");//提示输入文件名
85          printf("-> ");
86          scanf("%s", &File_Name);//获取名称
87          printf("\n");   //换行
88
89          pFile = fopen(File_Name, "rb");
90          //进行判定
91          if (pFile == NULL)
92          {
93              printf("打开文件失败！！！");
94              Sleep(3000);
95              exit;
96          }
97
98          //获取文件长度
99          fseek(pFile, 0, SEEK_END);
100         File_Size = ftell(pFile);
101         rewind(pFile);
```

图 1-22 获取需要更新的文件

- 配置写入地址：设定该 bin 文件在外部 Flash 中的写入起始地址，代码如下所示。

```
120
121         printf("# 请输入下载地址\n");//提示输入下载地址
122         printf("-> ");
123         scanf("%x", &Base_Address);//获取下载地址
124         printf("\n");   //换行
125         //Base_Address = 0x2F0000;
126
127         printf("开始编程...\n");
128         Current_Address = Base_Address;
```

图 1-23 获取下载地址

3. 4K 块级 Flash 操作（擦除与编程）

本实验以 4KB 为基本操作单元（匹配 Flash 的块擦除特性），依次执行擦除与编程动作：

（1）4K 块擦除

上位机发送帧头为 0xCC 的擦除指令，指定待擦除的 4K 块地址；等待下位机返回反馈指令（0xAA 表示擦除成功，0x55 表示失败）。相关代码如下图所示。

```
127         printf("开始编程...\n");
128         Current_Address = Base_Address;
129         //每次处理4K数据
130         for (j = 0; j < (File_Size / 4096); j++)
131         {
132             //先擦除4K数据
133             Send_Buff[0] = 0xCC;
134             memcpy(Send_Buff + 1, &Current_Address, 4);
135
136             do {
137                 if (send(sockfd, Send_Buff, 5, 0) == -1) {
138                     printf("%dth erase packet sending failed!\n", j);
139                     Fail_Flag = 1;
140                     continue;
141                 } else {
142                     Fail_Flag = 0;
143                 }
144
145                 DELAY;
146                 //再获取返回指令
147                 if (recv(sockfd, Recv_Buff, 5, 0) == -1) {
148                     printf("%dth erase packet reception failed!\n", j);
149                     Fail_Flag = 1;
150                 } else {
151                     Fail_Flag = 0;
152                 }
153             } while (Fail_Flag);
```

图 1-24 发送擦除指令

（2）4K 数据分包发送

将当前 4K 块数据拆分为 16 个 256 字节的数据包，依次发送帧头为 0xBC 的编程指令（含目标地址与数据包）；每发送一包后，接收下位机的写入确认反馈，确保数据可靠传输。相关代码如下所示。

```
155             //再将4K数据分成16个256字节的包依次发送
156             for (i = 0; i < 16; i++)
157             {
158                 Send_Buff[0] = 0xBC;
159                 memcpy(Send_Buff + 1, &Current_Address, 4);
160                 memcpy(Send_Buff + 5, pBuffer, 256);
161                 pBuffer += 256;
162                 Current_Address += 256;
163
164
165                 do {
166                     if (send(sockfd, Send_Buff, 261, 0) == -1) {
167                         printf("%dth data packet sending failed!\n",j*16 + i);
168                         Fail_Flag = 1;
169                         continue;
170                     } else {
171                         Fail_Flag = 0;
172                     }
173                     DELAY;
174                     //再获取返回指令
175                     if (recv(sockfd, Recv_Buff, 5, 0) == -1) {
176                         printf("%dth date packet reception failed!\n", j);
177                         Fail_Flag = 1;
178                     } else {
179                         Fail_Flag = 0;
180                     }
181                 } while (Fail_Flag);
182
183             }
184         }
```

图 1-25 发送数据

（3）最后一块不足 4K 的处理

若固件文件的最后一块数据不足 4K，仍需先按 4K 块地址发送擦除指令；编程时按实际数据长度处理，拆分数据包时若最后一包不足 256 字节，不足部分填充 0xFF 后发送。代码如下所示：

```
        Remain_Size = File_Size % 4096;
        //最后一次数据不足4K也不为0，先擦除4K再按照实际长度处理
        if (Remain_Size)
        {
            //先擦除4K数据
            Send_Buff[0] = 0xCC;
            memcpy(Send_Buff + 1, &Current_Address, 4);
            do {
                if (send(sockfd, Send_Buff, 5, 0) == -1) {
                    printf("%dth erase packet sending failed!\n", j);
                    Fail_Flag = 1;
                    continue;
                } else {
                    Fail_Flag = 0;
                }
                DELAY;
                //再获取返回指令
                if (recv(sockfd, Recv_Buff, 5, 0) == -1) {
                    printf("%dth erase packet reception failed!\n", j);
                    Fail_Flag = 1;
                } else {
                    Fail_Flag = 0;
                }
            } while (Fail_Flag);

            //再将4K数据分成16个256字节的包依次发送，若不满256则不足部分补0xFF
            for (i = 0; i < 16; i++)
            {
                if (Remain_Size > 256)
                {
                    Send_Buff[0] = 0xBC;
                    memcpy(Send_Buff + 1, &Current_Address, 4);
                    memcpy(Send_Buff + 5, pBuffer, 256);
                    pBuffer += 256;
                    Current_Address += 256;
                    Remain_Size -= 256;

                    do {
                        if (send(sockfd, Send_Buff, 261, 0) == -1) {
                            printf("%dth data packet sending failed!\n", j * 16 + i);
                            Fail_Flag = 1;
                            continue;
                        } else {
                            Fail_Flag = 0;
                        }
                        DELAY;
                        //再获取返回指令
                        if (recv(sockfd, Recv_Buff, 5, 0) == -1) {
                            printf("%dth date packet reception failed!\n", j);
                            Fail_Flag = 1;
                        } else {
                            Fail_Flag = 0;
                        }
                    } while (Fail_Flag);

                }
                else if (Remain_Size > 0)
                {
                    Send_Buff[0] = 0xBC;
                    memcpy(Send_Buff + 1, &Current_Address, 4);
                    memcpy(Send_Buff + 5, pBuffer, Remain_Size);
                    memset(Send_Buff + 5 + Remain_Size, 0xFF, 256 - Remain_Size);

                    do {
                        if (send(sockfd, Send_Buff, 261, 0) == -1) {
                            printf("%dth data packet sending failed!\n", j * 16 + i);
                            Fail_Flag = 1;
                            continue;
                        } else {
                            Fail_Flag = 0;
                        }
                        DELAY;
                        //再获取返回指令
                        if (recv(sockfd, Recv_Buff, 5, 0) == -1) {
                            printf("%dth date packet reception failed!\n", j);
                            Fail_Flag = 1;
                        } else {
                            Fail_Flag = 0;
                        }

                    } while (Fail_Flag);

```

图 1-26 最后一次传输不足 4K 数据的处理方式

4. 数据校验实现

编程完成后，上位机通过读取 Flash 数据与源文件比对，验证写入准确性，同样以 256 字节为校验单位：

（1）常规 256 字节校验

发送帧头为 0xCD 的读取指令，指定校验地址；接收下位机回传的 256 字节 Flash 数据后，与源文件中对应位置的 256 字节数据逐一比对。相关代码如下所示。

```
274         printf("编程成功，开始校验...\n");
275         //进入校验，每次读取256字节进行校验
276         pBuffer = File_Buffer;
277         Current_Address = Base_Address;
278         for (i = 0; i < (File_Size / 256); i++)
279         {
280             //先发送读取指令
281             Send_Buff[0] = 0xCD;
282             memcpy(Send_Buff + 1, &Current_Address, 4);
283             Current_Address += 256;
284 
285             do {
286                 if (send(sockfd, Send_Buff, 5, 0) == -1) {
287                     printf("%dth read packet sending failed!\n", i);
288                     Fail_Flag = 1;
289                     continue;
290                 } else {
291                     Fail_Flag = 0;
292                 }
293                 DELAY;
294 
295                 //再获取数据包
296                 if (recv(sockfd, Recv_Buff, 261, 0) == -1) {
297                     printf("%dth date packet reception failed!\n", i);
298                     Fail_Flag = 1;
299                 } else {
300                     Fail_Flag = 0;
301                 }
302                 DELAY;
303             } while (Fail_Flag);
304 
305             //比较返回的数据与文件中的数据是否一致
306             if (memcmp(pBuffer, Recv_Buff + 5, 256) != 0)
307             {
308                 printf("Packet verification failed for address 0x%08X\n", Current_Address);
309             }
310             pBuffer += 256;
311         }
```

图 1-27 读取返回的数据并进行对比

（2）最后一包不足 256 字节的处理

若最后一块数据的校验包不足 256 字节，仅按实际数据长度进行比对，忽略填充的 0xFF 部分，避免误判。代码如下所示：

```
312         Remain_Size = File_Size % 256;
313         //最后一次数据不足256字节也不为0，按照实际长度处理
314         if (Remain_Size)
315         {
316             //先发送读取指令
317             Send_Buff[0] = 0xCD;
318             memcpy(Send_Buff + 1, &Current_Address, 4);
319
320
321             do {
322                 if (send(sockfd, Send_Buff, 5, 0) == -1) {
323                     printf("%dth read packet sending failed!\n", i);
324                     Fail_Flag = 1;
325                     continue;
326                 } else {
327                     Fail_Flag = 0;
328                 }
329                 DELAY;
330
331                 //再获取数据包
332                 if (recv(sockfd, Recv_Buff, 261, 0) == -1) {
333                     printf("%dth date packet reception failed!\n", i);
334                     Fail_Flag = 1;
335                 } else {
336                     Fail_Flag = 0;
337                 }
338                 DELAY;
339             } while (Fail_Flag);
340
341
342             //比较返回的数据与文件中的数据是否一致
343             if (memcmp(pBuffer, Recv_Buff + 5, Remain_Size) != 0)
344             {
345                 printf("Packet verification failed for address 0x%08X\n", Current_Address);
346             }
347         }
348         printf("检验完成，编程结束！\n");
349     }
```


图 1-28 最后一次不足 256 字节的数据

1.7 实验操作流程

1.7.1 硬件连接

按以下步骤完成开发板硬件连接：

1. 将下载器、电源适配器、网线分别按对应接口连接至开发板；
2. 确认所有连接无误后，为开发板上电。

连接示意图如下所示：

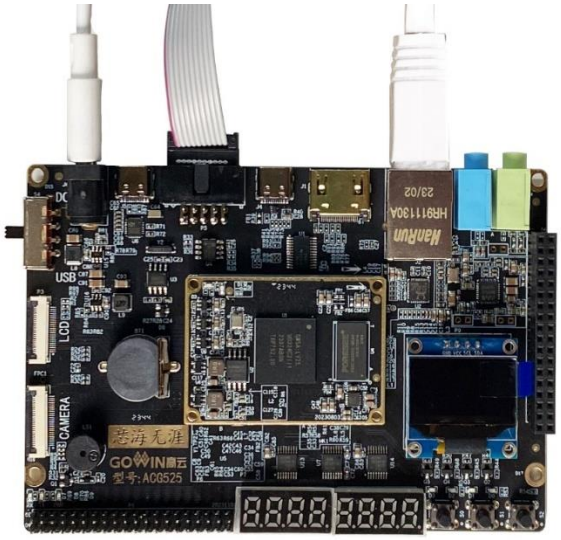

图 1-29 硬件连接

1.7.2 下载 IAP 工程

通过 Gowin 软件将 IAP 工程烧写至开发板外部 Flash，具体步骤如下：

1. 打开工程并进入下载界面：启动 Gowin 软件，打开 IAP 工程文件 GW5A_M1SC_Eth_Update；点击软件界面上方的 “下载” 图标，进入下载配置界面如下所示：

图 1-30 进入下载界面

2. 保存下载器配置：弹出 “Cable Setting” 窗口后，点击 “Save” 保存当前配置，如下所示。

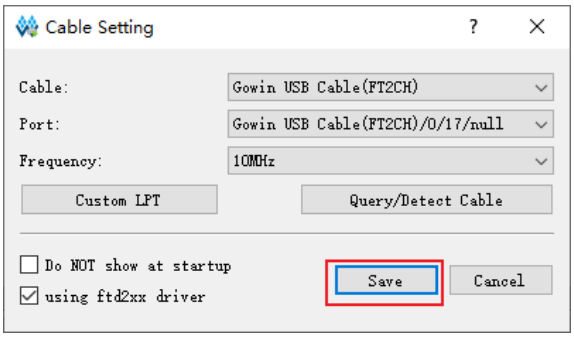

图 1-31 弹出 Cable Setting 窗口

3. 进入设备配置：右键点击下载界面列表第一行，选择 “Config Device”，如下图所示。

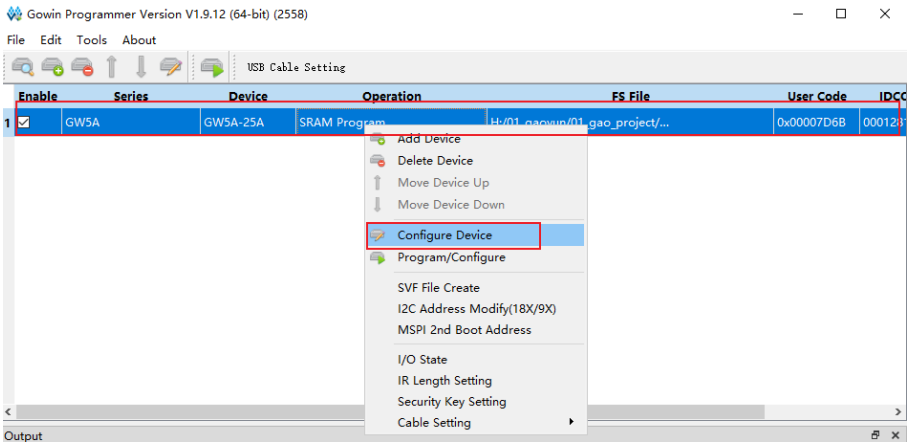

图 1-32 选择“Config Device”

4. 配置 Flash 操作参数：在弹出的 “Device configuration” 界面中，依次选择：

- 模式：External Flash Mode Arora V
- 操作类型：exFlash Erase,Program,Verify Arora V，先擦除旧数据，再写入新数据，最后校验，确保烧写正确。如下图所示。

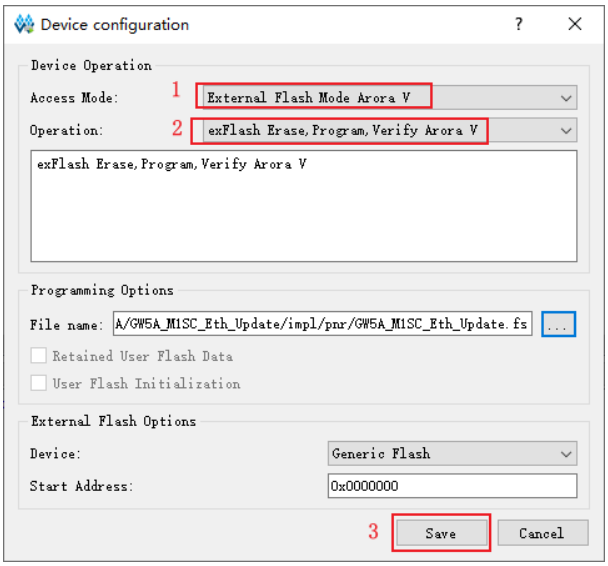

图 1-33 Device configuration 配置界面

5. 执行烧写：点击 “Program” 按钮，开始将 IAP 程序烧写至外部 Flash 的 0x000000 地址，如下所示。

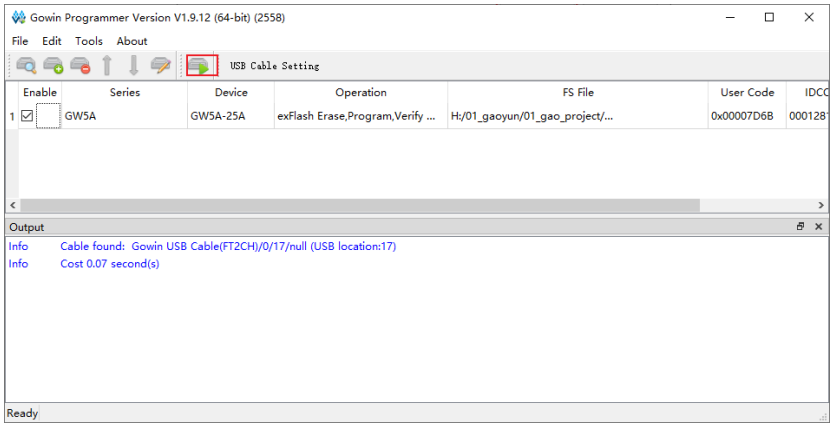

图 1-34 下载到 Flash

6. 确认烧写结果：当软件 “Output” 输出栏显示 “烧写和校验 Flash 成功” 时，表明 IAP 工程烧写完成，如下所示。

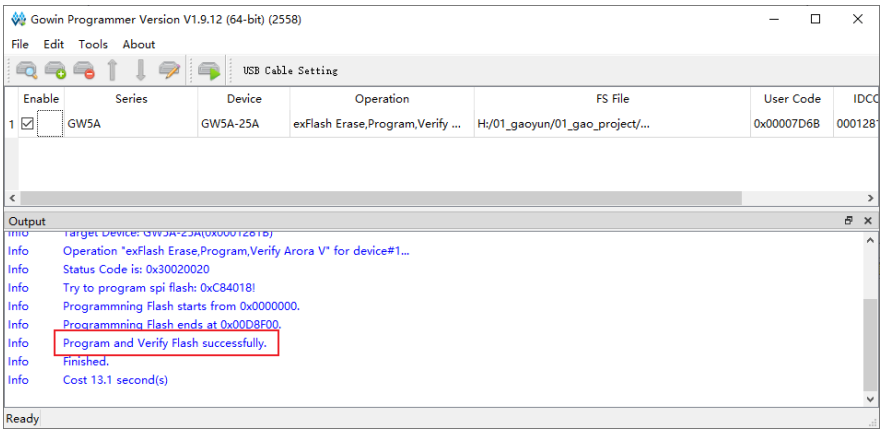

图 1-35 烧写 Flash 成功

1.7.3 使用上位机软件更新 APP 工程

1.7.3.1 APP 工程编写

本次实验的 APP 工程非常简单，就是通过 S0 控制 LED0 的亮灭，代码如下所示：

```
3  module Led_Key(
4      input Key,
5      output Led
6  );
7
8  assign Led = ~Key;
9
10
11 endmodule
```

图 1-36APP 代码如下所示

然后分配引脚，编译工程，得到 APP 工程的 bin 文件，如下所示：

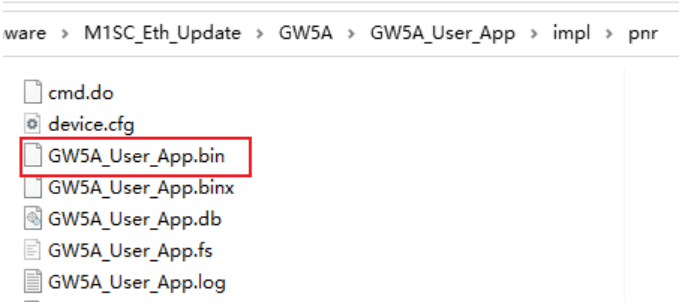

图 1-37 得到 APP 的 bin 文件

1.7.3.2 修改电脑网口的 IP 地址

修改网口的 IP 地址为 192.168.31.79。

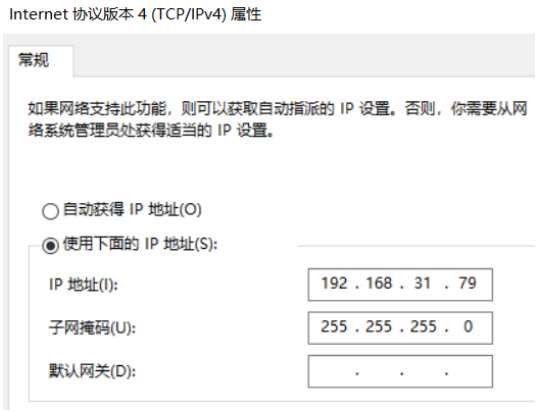

图 1-38 修改电脑 IP 地址

1.7.3.3 上位机操作

通过 “网络升级工具.exe” 实现 APP 固件的远程烧写，具体操作步骤如下：

1. 启动工具并建立网络连接

（1）运行 “网络升级工具.exe”。

（2）输入开发板 IP 地址 192.168.31.78，按下回车键；

（3）输入通信端口号 1234，按下回车键。

若界面显示 “连接成功”，则表明 PC 与开发板已建立通信；若连接失败，需先关闭电脑防火墙再重试，连接界面如图 1-39 所示。

图 1-39 网络连接

2. 加载 APP 固件文件

将待升级的 APP 固件文件 GW5A_User_App.bin 拖入工具对话框，工具会自动识别并填入该文件的完整路径如下图所示。

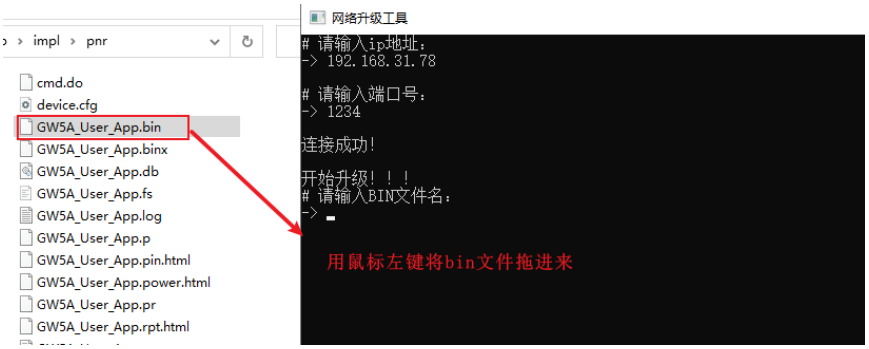

图 1-40 输入需要更新的 bin 文件

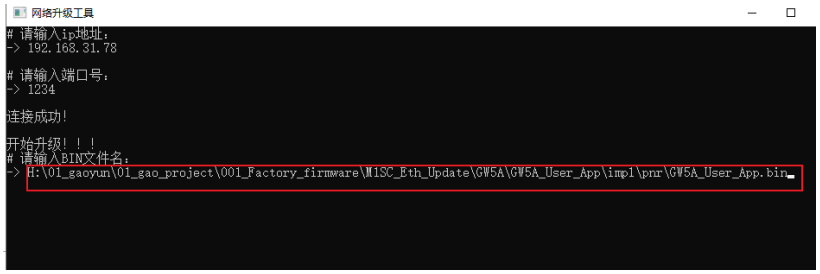

图 1-41 上位机识别 APP 文件路径

3. 配置下载地址并启动更新

（1）按下回车键确认已填入的固件文件路径；

（2）输入 APP 固件在 Flash 中的下载地址 0x2F0000（需与 1.3.3 节设置的 Multi Boot 地址一致），按下回车键，如下图所示。

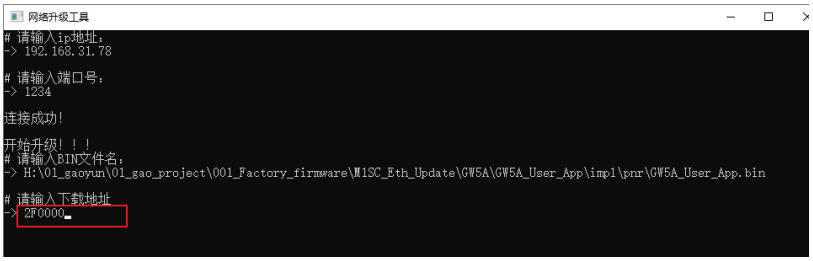

图 1-42 输入下载地址

工具将自动启动编程流程，如下所示。

图 1-43 开始编程

4. 等待编程与校验完成

（1）编程过程约需数十秒，完成后工具将自动进入校验阶段，如下所示；

图 1-44 进入校验

（2）校验完毕后，若界面无任何报错信息，则说明 APP 固件更新成功，如下所示。

图 1-45 校验完成

1.7.3.4 APP 程序切换操作

由于 ACG525 开发板未将 “Reconfig” 管脚直接引出，需通过硬件操作触发程序切换，具体步骤如下：

1. 触发 Reconfig 信号：短接核心板上电阻 R4 的右端（对应 Reconfig 管脚），短接动作会触发 FPGA 重新加载固件；
2. 确认切换触发：观察开发板的 Done 指示灯 —— 短接后指示灯会短暂熄灭，随后重新亮起，表明 FPGA 已开始执行固件跳转，如下图所示。

图 1-46 短接 Reconfig 脚

APP 程序功能验证：程序切换后，通过以下操作验证 APP 是否正常加载：按下开发板上的 S0 按键，LED0 指示灯点亮；松开 S0 按键，LED0 指示灯熄灭。若符合上述现象，说明开发板已成功加载新的 APP 程序，如下图所示。

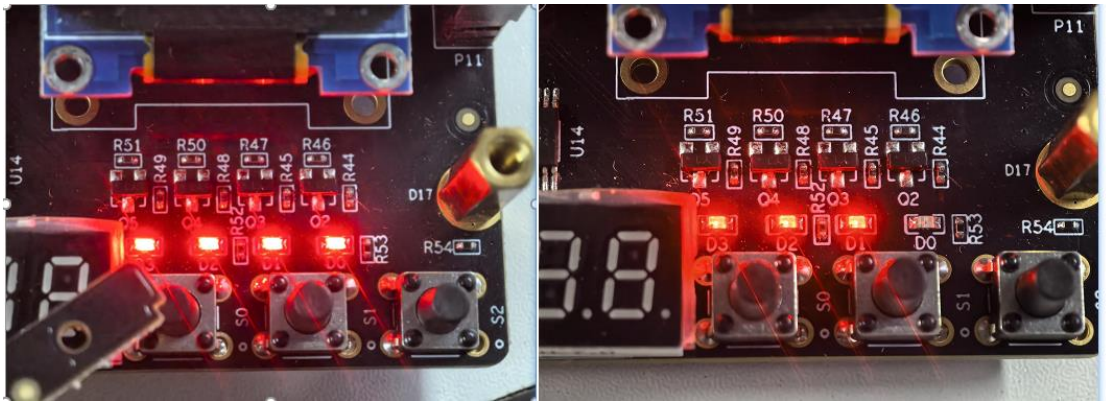

图 1-47APP 程序验证（左边，按下灯亮，右边，松开灯灭）

1.8 实验总结

本次实验成功实现了基于以太网的 FPGA 开发板固件远程在线升级，是一个融合硬件配置与软件开发的综合性案例。实验核心涉及以下关键技术与知识的应用：

1. FPGA 硬件开发：高云 FPGA 的 M1 软核配置、ITCM 初始化文件生成与导入、Multi Boot 地址设置、PLL 时钟模块配置；
2. 嵌入式软件开发：基于 MDK 的 M1 软核 C 编程、uIP 轻量级 TCP/IP 协议栈应用、Flash 擦除 / 写入 / 读出底层控制；
3. 上位机开发：基于 Visual Studio 的 TCP 通信程序编写、固件数据分包传输与校验逻辑实现；
4. 系统交互设计：上下位机自定义通信协议（命令帧格式）、硬件触发（Reconfig）与软件控制协同。

通过本实验的实践，可进一步拓展相关应用场景，例如基于以太网实现文件传输、远程设备状态监控、多设备批量固件升级等功能。