核交互能力；第三，探讨跨核调试的可行性，并在此基础上研究在跨核调试场景中，调试主机能否突破目标处理器的权限隔离机制，实现对高权限资源的越权访问。

为达成以上实验目的，设计如下实验流程。首先，STM32CubeProgrammer 配合 ST-Link 调试探针连接开发板，读取调试寄存器 DHCSR，验证 C_DEBUGEN 是否处于有效状态。其次，基于开发板支持的核间通信机制，搭建核间通信信道，实现跨核交互。随后，在核间通信信道基础上，将一核作为调试主机，向另一核发起调试操作。最后，若能实现跨核调试后，进一步尝试在高权限寄存器与内存资源访问层面进行验证，即分析调试主机能否突破常规访问限制，读写另一核中具有较高权限的资源。

通过以上步骤，综合评估 Cortex-M 多核处理器中调试访问路径是否存在滥用可能，验证基于调试机制发起的权限隔离突破是否能够在 Cortex-M 处理器上实现。

4.2 攻击实施流程

4.2.1 调试功能验证与程序烧录

本节在 Arch Linux 主机上验证目标开发板的 C_DEBUGEN 状态，并配置相应的开发环境。实验所用开发板为 STMicroelectronics NUCLEO-H755ZI-Q（STM32H7 Nucleo-144 系列），其集成 ST-LINK-V3E 调试器/编程器，无需外接 JTAG/SWD 探针。

首先，将开发板 CN1 接口通过 Micro-USB 线缆与 PC 相连。确保已加载 ST-LINK 驱动的情况下启动 STM32CubeProgrammer，在“Access Port”下拉菜单中选择对应的 AP 条目。不同 AP 对应不同资源，具体情况如下：

- AP0：通过 AHBLite 总线访问 Cortex-M7 核心的调试与追踪模块
- AP1：访问 D3 域的 AHB 总线矩阵，以获取 D3 域在 D1、D2 关闭时的存储与外设可视性
- AP2：访问系统 APB 调试总线，用于对处理器核外的所有组件进行调试与追踪
- AP3（仅双核设备）：通过内部 AHB 总线访问 Cortex-M4 核心的调试与追踪模块

根据本研究关注的调试目标，先后选择 AP0（Cortex-M7 核心）与 AP3（Cortex-M4 核心），随后点击 Connect 建立 DAP 会话。连接成功后，切换至“Memory & File editing”选项卡，在“Address”字段对应输入框输入 0xE000EDF0（DHCSR 寄存器物理地址），点击 Read 读取 DHCSR。

图4.1、图4.2分别展示了 M7 核与 M4 核的读取结果。C_DEBUGEN（DHCSR 最低

位）为 1 表示调试功能已开启，为 0 表示调试功能已禁用。由图可知，M7、M4 核调试使能均已开启。在确认 DHCSR 中 C_DEBUGEN 已在硬件层面启用后，下一步为程序烧录与运行验证（实验采用 STM32CubeIDE v1.17.0 作为 IDE）。

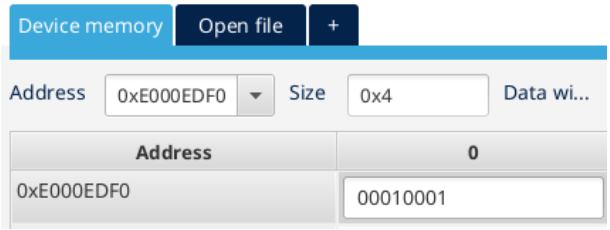

图 4.1 M7 核 DHCSR 读取结果

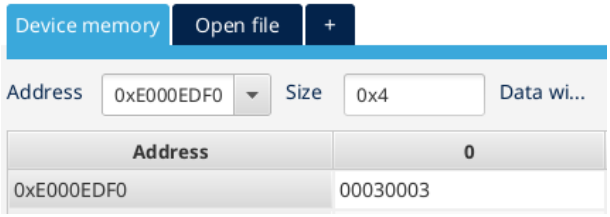

图 4.2 M4 核 DHCSR 读取结果

程序烧录的全流程：在 STM32CubeIDE 中启动烧录流程后，IDE 首先通过 USB 接口以 CMSIS-DAP 协议与板载 ST-LINK/V3E 调试器建立通信，随后将主机侧的 Flash 编程命令和固件二进制数据通过 USB 封装传输至 ST-LINK。ST-LINK/V3E 固件将接收到的 USB 包解析为 SWD-DP 命令，并通过 SWD-AP 向目标 MCU 发起对寄存器和内存的读写请求。调试器通过在 SWD-DP 的 SELECT 寄存器中选择 AHB-AP（APSEL=0），将编程事务映射至 Cortex-M 处理器的内部内存空间。主机依次写入 CSW、TAR 及 DRW 寄存器，携带待写入的数据及目标地址，触发 MCU 内置 Flash 控制器执行“解锁 → 编程 → 校验 → 锁定”程序序列；控制器状态寄存器在操作完成后报告编程结果，IDE 则通过读回已写入内容并进行校验（校验和或 CRC）以确认烧录成功。内部总线层面，SWD-AP 发起的事务通过片内 AHB-Lite 与 AXI 总线矩阵传递至 Flash 存储单元，实现对映射寄存器和数据区的可靠 MMIO 写操作。此外，ST-LINK 可模拟为 USB 大容量存储设备，用户仅需将固件文件复制至虚拟盘符，即可完成对 Flash 的自动写入与验证。该全流程涵盖 CMSIS-DAP、SWD、AHB-AP、AXI/AHB-Lite 等多个协议与硬件机制，确保对 STM32 系列器件进行高效且可验证的程序烧录。

在 STM32CubeIDE 中创建以 NUCLEO-H755ZI-Q 为目标的 STM32 项目后，IDE 会自动生成对应的.ioc 配置文件。打开该文件并切换至 Pinout & Configuration 视图后，首

先依据开发板电路图完成 GPIO 及其他外设引脚映射；接着进入 RCC 选项卡，将高速时钟源（HSE）设置为 8 MHz 外部晶振，并启用 32.768 kHz 的 LSE 作为独立的 RTC 时钟源。在 PLL Configuration 区域，将主 PLL（PLL1）输入分频 DIVM1 设为 5，以将 8 MHz 分频至 1.6 MHz；将 PLL 倍频因子 PLLN1 设为 500，以产生 800 MHz 的 VCO 输出；再将主输出分频 DIVP1 设为 2，以获得 400 MHz 的系统时钟（SYSCLK）。CPU1 直接取用 400 MHz 的 SYSCLK 作为核心时钟，CPU2 则经 D1CPRE 分频为 200 MHz。随后，将 AHB 总线预分频 HPRE 设为 2，使 HCLK 等于 200 MHz；在 AXI 外设总线配置中，AXBS1 时钟保留为 200 MHz，AXBS2 时钟通过 D2PPRE2 分频为 100 MHz。

各 APB 域的分频策略如下。APB3（D1 域）经 D1PPRE 得到 100 MHz；APB1（D2 低速域）经 D2PPRE1 得到 100 MHz，定时器时钟为 200 MHz；APB2（D2 高速域）经 D2PPRE2 得到 100 MHz，定时器时钟同为 200 MHz；APB4（D3 域）经 D3PPRE 得到 100 MHz。

完成上述配置后，STM32CubeIDE 的 Clock Configuration 视图将直观展示各分频器输出的频率，验证所有系统时钟、总线时钟及外设时钟均在器件规格范围内。图4.3直观展示了所有时钟信号的配置。

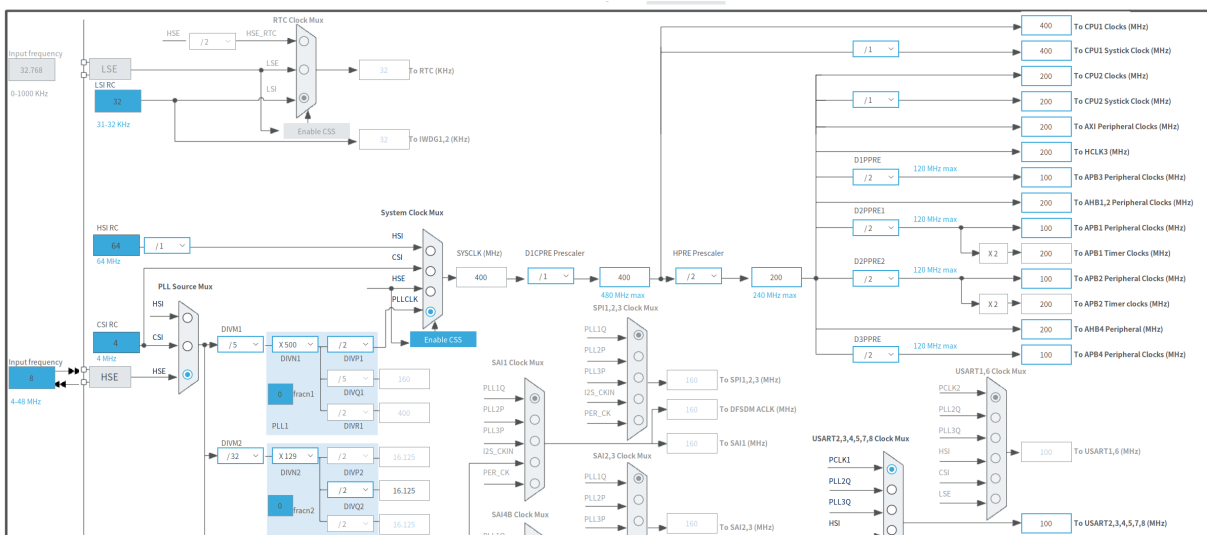

图 4.3 时钟源配置

配置完成后，通过点击“Generate Code”调用 STM32CubeMX 引擎，根据设定生成基于 HAL 库的初始化代码，并执行“Project → Build All”完成基础工程的首次编译，以确保项目能够无误生成。随后在对应处理器 Core/Src/main.c 文件中添加应用逻辑，同时根据功能需求引入必要的头文件与中间件库。

为了验证前述程序烧录流程的正确性，以点亮板载 LED 为实验案例，编写并运行一段 GPIO 输出控制代码。首先，根据开发板电路图定位目标 LED 的物理引脚，并在.ioc 文件 Pinout & Configuration 视图中将该引脚配置为 GPIO_Output，同时将该引脚的上下文分配给所需的核心。

在生成的 HAL 初始化代码中，添加以下 GPIO 初始化函数，用于打开 GPIOB 外设时钟并配置 PB0 为推挽输出、无上下拉、低速模式。

```
static void MX_GPIO_Init（void）
{
    GPIO_InitTypeDef GPIO_InitStruct = {0};

    /* GPIOB 外设时钟使能 */
    __HAL_RCC_GPIOB_CLK_ENABLE（）;

    /* 初始化输出电平为低 */
    HAL_GPIO_WritePin（GPIOB, GPIO_PIN_0, GPIO_PIN_RESET）;

    /* 配置 PB0 为推挽输出、无上下拉、低速 */
    GPIO_InitStruct.Pin   = GPIO_PIN_0;
    GPIO_InitStruct.Mode  = GPIO_MODE_OUTPUT_PP;
    GPIO_InitStruct.Pull  = GPIO_NOPULL;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
    HAL_GPIO_Init（GPIOB, &GPIO_InitStruct）;
}
```

在 main() 函数中，调用 MX_GPIO_Init() 完成端口配置后，需要切换 PB0 引脚的电平状态，其接口是 HAL_GPIO_TogglePin（GPIOB, GPIO_PIN_0）。配合 HAL_Delay() 函数，预计 LED 会以所设频率闪烁。

完成代码编写后，再次执行“Build”操作进行编译，并通过“Run As → STM32 C/C++ Application”启动会话。STM32CubeIDE 将调用板载 ST-Link 调试器，完成 Flash 擦写及程序烧录，部署程序至目标处理器。

在烧录过程中，首先出现 gdb-server 端口被占用、MI 命令执行失败以及 ST-Link CLI 无法打开等常见调试错误，而 STM32CubeProgrammer 在初始化板载 ST-LINK 调试器时连续返回“No device found on target”，导致开发板始终无法被识别，成为制约后续实验的主要障碍。

为排查该故障，先后更换 Micro-USB 数据线并在主机与 USB 集线器间交替连接开发板，以排除连线问题；使用 STM32CubeProgrammer 对板载 ST-LINK 固件执行升级；查阅 NUCLEO-H755ZI-Q 用户手册与原理图，确认 BOOT0 / BOOT1 跳线及 5 V 供电指示灯（LD5）状态均正常；并在 Windows 平台通过 Keil MDK-Arm 重复烧录测试以排除软件兼容性问题。上述操作均未恢复设备识别后，依据 ST 官方针对 LDO / SMPS 配置不匹配导致死锁的恢复流程，在断电状态下利用杜邦线将 CN11 BT0 引脚短接至 VDD，以切换至系统存储区 Bootloader 模式。

上电后通过 STM32CubeProgrammer 建立连接并执行用户 Flash 擦除，再次断电、恢复 BOOT0 至低电平并重新上电后，系统自用户 Flash 正常启动，ST-LINK 调试器重新识别并能够稳定完成烧录。图4.4展示了程序烧录后开发板 LED 的状态。可以观察到，LED 按照预先设定的频率闪烁，该现象表明程序已烧录至开发板并开始执行，验证了烧录流程的正确性。

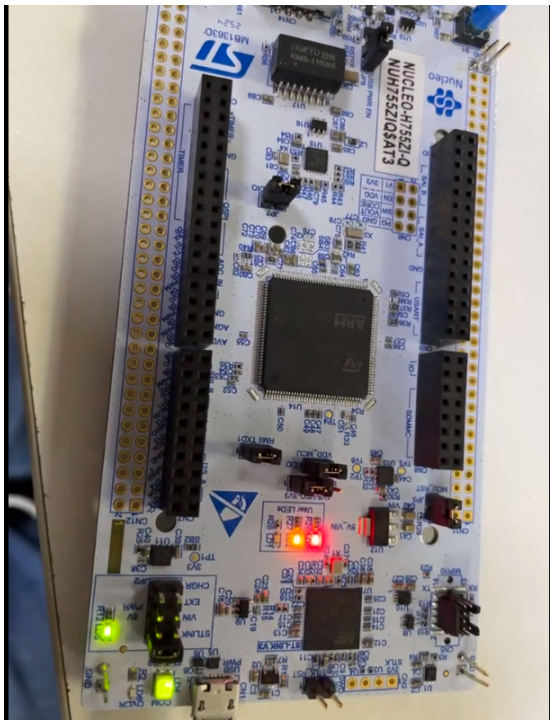

图 4.4 首次点亮 LED

本节在物理连接 ST-Link 调试探针与开发板的条件下，验证了调试功能可被开启，并摸索出程序烧录流程，为后续调试攻击实验的开展奠定了基础。

4.2.2 串口通信与核间通信构建

为了在不占用 SWD/JTAG 通道的前提下获取系统状态反馈，选用板载 USART3 作为独立串行通信接口。首先，在 STM32CubeIDE 生成的.ioc 配置文件中，将 PD8 与 PD9 分别映射为 USART3_TX 与 USART3_RX（参考开发板电路图），并启用对应的 GPIO 时钟与复用功能；随后，通过 HAL 库机制将标准输出重定向至串口传输接口，具体实现如下：

```
#define PUTCHAR_PROTOTYPE int __io_putchar（int ch）
PUTCHAR_PROTOTYPE {
    HAL_UART_Transmit（&huart3, （uint8_t *）&ch, 1, HAL_MAX_DELAY）;
    return ch;
}
```

在 PC 端使用 minicom 等终端工具监控时，基于 `HAL_UART_Transmit()` 或 `printf()` 的任何输出均未能显示。为排除外设映射冲突，先后将 USART3 切换成 USART1（对应引脚 PB6、PB7）和 USART2，但均未能接收到开发板传输回主机的数据。

在进一步排查过程中，发现 STM32CubeMX 在配置 USART3 复用时提示“IP under HMI BSP driver control”错误，因此重建项目工程并在人机交互界面（Human-Machine Interaction, HMI）中取消 BSP 驱动对该外设的控制后重新启用 USART3。该做法消除了错误提示，在主机对应端口接受到开发板返回数据，但数据以乱码形式呈现。为解决乱码问题，验证了引脚复用关系（包括以太网控制器与 USB_OTG_FS 的冲突）以及通信参数（波特率、校验方式、停止位等），均符合设计规范。

结合 STM32H7 系列参考手册 UM2408 第 6.9 节“OSC Clock”中关于外设时钟源的说明，串口外设时钟必须固定在 8 MHz。为满足此要求，本研究首先升级了 ST-LINK 固件以启用 MCO 输出功能，并将 MCO 的时钟源配置为 HSI/2，其频率约为 8 MHz。随后，在 STM32CubeIDE 的.ioc 时钟树配置界面中，将外部高速时钟（HSE）的输入值以及工程文件 stm32h7xx_hal_conf.h 中的宏定义常量 HSE_VALUE 统一设置为 8 MHz。

经过上述时钟源校正，通过在主机终端执行 `ls /dev/ttyACM*` 指令，可以观察到串口设备已被系统正确识别，确认了底层驱动的正常工作。为建立可靠的调试通信链路，在 STM32CubeIDE 的控制台配置中新增了串口通信端口，并设定通信参数为：波特率 115200 bit/s，数据位 8 bit，校验位无，停止位 1 bit。

图4.5展示了 USART3 串口通信输出的实验结果。实验观察到，USART3 能够稳定地输出调试信息，并且这些信息能够被终端工具正确接收和显示。可靠的串口通信链路的建立，是进行后续固件调试、信息监控以及实验结果输出的关键前提，为研究过程中与目标设备的交互提供了必要的通信通道。

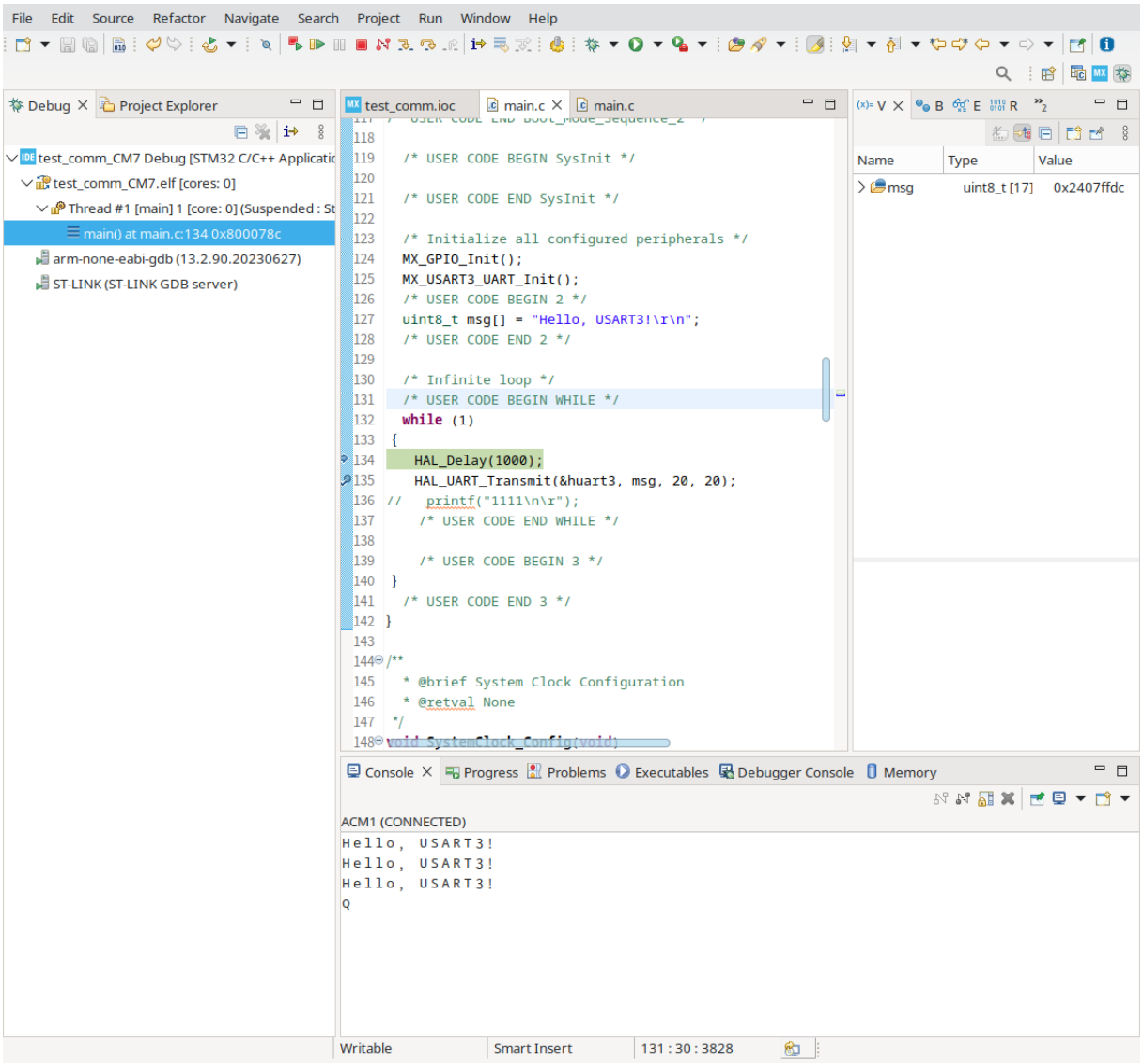

图 4.5 建立 USART3 串口通信

在实现串口通信的基础上，进一步探索多核系统间的通信机制，旨在构建核间通信信道。为实现此目标，需明确指定开发板的主从处理器。本实验选用 NUCLEO-H755ZI-Q 开发板，该开发板集成了一个 Cortex-M7 核心和一个 Cortex-M4 核心。根据实验设计，Cortex-M7 核心被配置为主处理器，负责执行主要的系统控制和任务调度；Cortex-M4 核心则被设定为从处理器，用于执行特定的辅助功能。

为实现 Cortex-M7 与 Cortex-M4 核心间通信，本实验采用了开放非对称多处理（Open Asymmetric Multi-Processing, OpenAMP）协议栈，并利用 STM32H7 微控制器内置的硬件信号量（Hardware Semaphore, HSEM）模块。OpenAMP 协议栈为异构多核系统提供了通信框架，简化了不同核心间的数据交换和同步；HSEM 模块则作为硬件同步机制，协调 Cortex-M7 和 Cortex-M4 对共享资源的访问，避免潜在的资源冲突。

OpenAMP 协议栈和 HSEM 模块的协同运作依赖于底层的共享内存机制。图4.6 描述了 STM32H755 系统架构，该架构划分为 D1、D2 和 D3 三个主要的电源和时钟域。Cortex-M7 位于 D1 域，Cortex-M4 位于 D2 域。D1 和 D2 域通过总线互联实现通信和资源共享。D3 域包含低功耗外设和特定的存储资源，并能与 D1 和 D2 域直接通信，实现跨域资源访问。特别地，D3 域下的 SRAM4（物理地址范围 0x38000000-0x3800FFFF）被指定为双核共享内存区域，适用于存储跨核通信所需的消息缓冲区、控制结构等数据。两个核心均具备对 SRAM4 的访问权限，且 SRAM4 在 D1 或 D2 域进入低功耗模式时仍保持可用，适用于 OpenAMP 等异步通信协议。

为将 SRAM4 配置为共享内存区域，在 STM32CubeIDE 的.ioc 文件中，针对对应处理器核心启用了 MPU Region，并将该 MPU Region 的基地址设置为 0x38000000。OpenAMP 协议栈在共享内存的基础上实现了消息传递和远程过程调用等高级通信抽象。HSEM 模块则用于确保在多核并发访问共享内存时的数据一致性和完整性。

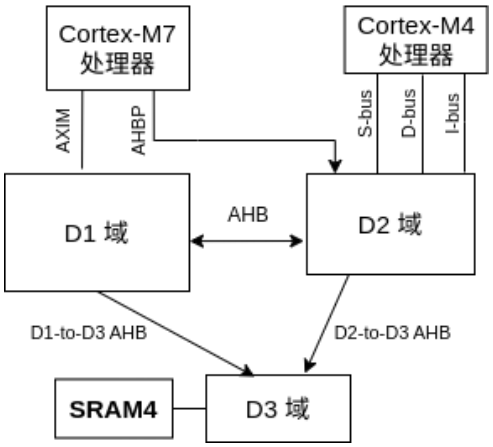

图 4.6 STM32H755 系统架构

主从处理器首先完成 HSEM 初始化，并通过调用 `MAILBOX_Init()` 函数配置通道，使得在任一核端写入共享寄存器时可触发对方中断，从而实现异步通知。在共享内存区 SRAM4 中，主核 M7 通过 `MX_OPENAMP_Init()` 初始化 OpenAMP 环境，并以服务名

openamp_test 注册 RPMsg 通信端点。

```
/* Master （M7） */
#define RPMSG_CHAN_NAME  "openamp_test"
static struct rpmsg_endpoint rp_endpoint;
static volatile unsigned int received_data, message_received, service_created;

static int rpmsg_recv_callback（struct rpmsg_endpoint *ept, void *data,
                                size_t len, uint32_t src, void *priv） {
    received_data = *（unsigned int*）data;
    message_received = 1;
    return 0;
}
void new_service_cb（struct rpmsg_device *rdev, const char *name, uint32_t dest
    ） {
    OPENAMP_create_endpoint（&rp_endpoint, RPMSG_CHAN_NAME, dest,
                            rpmsg_recv_callback, NULL）;
    service_created = 1;
}

/* Slave （M4） */
#define RPMSG_SERVICE_NAME  "openamp_test"
static struct rpmsg_endpoint rp_endpoint;
static volatile int message_received;
static volatile uint32_t *shared_DHCSR;

static int rpmsg_recv_callback（struct rpmsg_endpoint *ept, void *data,
                                size_t len, uint32_t src, void *priv） {
    shared_DHCSR = （uint32_t*）data;
    message_received = 1;
    return 0;
}
uint32_t receive_message（void） {
    while （!message_received） {
        OPENAMP_check_for_message（）;
    }
    message_received = 0;
    return *shared_DHCSR;
}
```

端点的接收回调函数 rpmsg_recv_callback() 用于接收从核返回的响应数据，并将其保存在全局变量中，同时设置标志位表示消息接收完成。与此同时，从核 M4 同样调用 MX_OPENAMP_Init() 初始化 RPMsg 子系统，并在检测到服务 openamp_test 创建后调用 OPENAMP_create_endpoint() 注册自身端点，其回调函数同样用于接收来自主核的命令或数据，并更新状态变量。

在 Master 端，使用 OPENAMP_send() 下发送命令或数据；在 Slave 端，通过循环调用 OPENAMP_check_for_message() 等待并获取消息，回传确认或响应。该架构在不依赖 SWD/JTAG 通道的情况下，实现了 Master 对 Slave 的数据传送，为后续跨核调试攻击流程提供了通信渠道。

为实现上述基于 OpenAMP 的跨核通信机制，需要在 STM32H755ZITX_FLASH.ld 链接脚本配置 OpenAMP 所需的专用内存区域。在 MEMORY 段中，定义了两个与 OpenAMP 协议栈相关的内存块。

```
OPENAMP_RSC_TAB (xrw) : ORIGIN = 0x38000000, LENGTH = 1K}
// 该区域用于存放 OpenAMP 的资源表，起始地址为 0x38000000，长度为 1 KB。

OPEN_AMP_SHMEM (xrw) : ORIGIN = 0x38000400, LENGTH = 63K}
// 该区域被指定为 OpenAMP 的共享内存，起始地址为 0x38000400，长度为 63 KB。该地
    址范围位于 SRAM4 区域内，并与 .ioc 文件中配置的 MPU Region 相对应。
```

此外，还定义了两个全局符号分别用于标记共享内存区域的起始地址与结束地址。

```
__OPENAMP_region_start__ = ORIGIN(OPEN_AMP_SHMEM);
__OPENAMP_region_end__ = ORIGIN(OPEN_AMP_SHMEM) + LENGTH(OPEN_AMP_SHMEM);
```

在 SECTIONS 段中，添加了名为.openamp_section 的专用段，用于链接 OpenAMP 生成的资源表。该段被标记为 NOLOAD，表示不会被加载至 RAM，而是被直接定位在指定物理地址。

该段将.resource_table 输入节链接至 OPENAMP_RSC_TAB 指定的地址区域，其运行地址与加载地址一致，均为 0x38000000。上述内存划分确保资源表与共享内存在主从核间可见并物理隔离，为 OpenAMP 的跨核消息机制提供了必要的内存支持。

```
.openamp_section （NOLOAD） : {
    . = ABSOLUTE（0x38000000）;
    *（.resource_table）
} >OPENAMP_RSC_TAB AT > FLASH
```

为保证双核系统的正确加载与启动，在完成主从核通信代码开发与内存区域配置后，需要进行刷写参数的设置。图 4.7 介绍了 M7 核的调试配置。配置是在 STM32CubeIDE 的“Application Configurations”对话框中，于应用配置的 Startup 选项卡内的“Load Image and Symbols”区域添加 M7 核的 ELF 镜像（例如，Debug/test_IPCC_CM7.elf）。针对该 ELF 文件，需要勾选 Download 选项以确保代码被烧录至 Flash，并勾选 Load symbols 选项以便进行符号调试。此外，若 CM7 代码尚未自动构建，可在“Main”选项卡中启用 Build (before launch) 选项，以在启动调试前自动构建 CM7 核的固件。

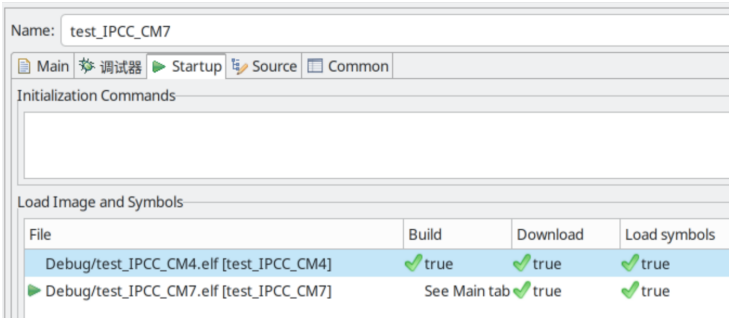

图 4.7 M7 应用配置

图 4.8 介绍了 M4 核的调试配置。同样在 STM32CubeIDE 的“Application Configurations”对话框中，于应用配置的 Startup 选项卡内的“Load Image and Symbols”区域添加 CM4 核的 ELF 镜像（例如，Debug/test_IPCC_CM4.elf）。关键在于，针对 CM4 核的 ELF 文件，应仅勾选 Load symbols 选项，并将 Download 选项取消勾选。此配置旨在避免在调试启动时重复下载 CM4 核的符号文件，因为通常 CM4 核的固件会与 CM7 核的固件一同被刷写。此外，为了确保芯片复位后能够正确加载并启动两个核心的固件，需要在 Common 选项卡中启用“Reset and halt”选项。此项配置使得 STM32CubeIDE 在调试启动时，能够按照预设顺序准确地擦写和下载 CM4 与 CM7 核的 Flash，并按序执行代码。

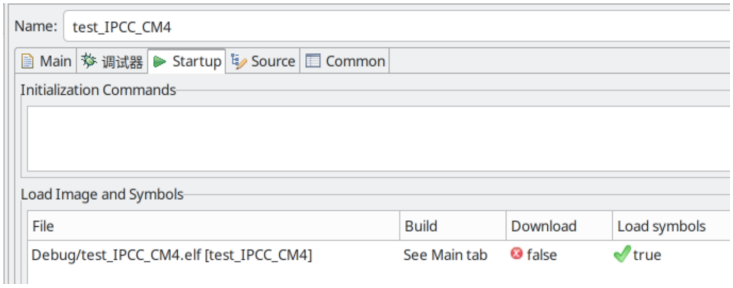

图 4.8 M4 应用配置