XHSC 小华半导体
XIAOHUA SEMICONDUCTOR

32 位微控制器

HC32F4 系列 Debug Monitor 原理与应用

应用笔记

Rev1.00　2026 年 04 月

适用对象

产品系列	产品型号
HC32F4 系列	ALL

声　明

★ 小华半导体有限公司（以下简称：“XHSC”）保留随时更改、更正、增强、修改小华半导体产品和/或本文档的权利，恕不另行通知。用户可在下单前获取最新相关信息。XHSC 产品依据购销基本合同中载明的销售条款和条件进行销售。

★ 客户应针对您的应用选择合适的 XHSC 产品，并设计、验证和测试您的应用，以确保您的应用满足相应标准以及任何安全、安保或其它要求。客户应对此独自承担全部责任。

★ XHSC 在此确认未以明示或暗示方式授予任何知识产权许可。

★ XHSC 产品的转售，若其条款与此处规定不同，XHSC 对此类产品的任何保修承诺无效。

★ 任何带有“®”或“™”标识的图形或字样是 XHSC 的商标。所有其他在 XHSC 产品上显示的产品或服务名称均为其各自所有者的财产。

★ 本通知中的信息取代并替换先前版本中的信息。

目　录

1 概述

常规调试 MCU 时，CPU 及大部分外设模块被迫停止运行。这种 Halting 调试被称作“侵入式调试”。侵入式调试虽然功能丰富，但其强制 CPU 和其他外设模块停止的做法可能会对系统正常工作造成不良影响。比如调试时，与外部进行通信的模块可能会因为 CPU 没有及时响应，导致通信失败，影响调试过程问题的定位。另外，有些产品未引出 SWD/JTAG 信号，仅留出 USART 等基本通信接口，则侵入式调试手段受限。

为了解决上述侵入式调试带来的问题，ARM Cortex-M4 内核中预留了一种非侵入式调试机制——Debug monitor。该方式的调试优先级可配置，在系统不停机的情况下进行调试，可以让一些实时性要求高的关键性任务持续运行。而且，这种调试方式通过 USART、SPI 等普通通信接口与调试者交互，可以低成本地部署在强实时、强连续性、远程/生产环境等硬件环境受限的场景。

2 原理

Debug Monitor 调试的流程：当触发调试事件时，CPU 和外设不会立即暂停，而是进入 DebugMon_Handler 异常服务函数。开发者可在该 Handler 中自定义调试动作，如查看数据、查看指令、输出信息等。这个调试过程相当于发生了一次中断，不影响其他高优先级中断的实时响应。

2.1 优先级

Debug Monitor 异常向量是一个编号为 12、优先级可配置的中断。Debug Monitor 中断与普通中断略有不同：其中断处理函数运行时，其他高优先级的中断会使 DebugMon_Handler 抢占 CPU 进入中断嵌套状态，直到此高优先级的中断处理函数执行完成之后再执行 DebugMon_Handler；但是，Debug Monitor 异常遇到正在执行的高优先级中断处理函数时不会 Pending，而是直接忽略本次调试事件。

2.2 调试事件

调试事件源有多种，如图 2-1 所示。包括 DWT（Data Watchpoint and Trace）跟踪数据产生的监测点调试事件（Watchpoint），FPB（Flash Patch and Breakpoint）产生的指令断点调试事件（Breakpoint），外部信号 EDBGREQ 产生的调试事件以及通过软件执行 BKPT 指令产生的调试事件。其中，EDBGREQ 产生调试事件依赖于外部信号，BKPT 软件方式产生调试事件多用于带 IDE 的侵入调试。

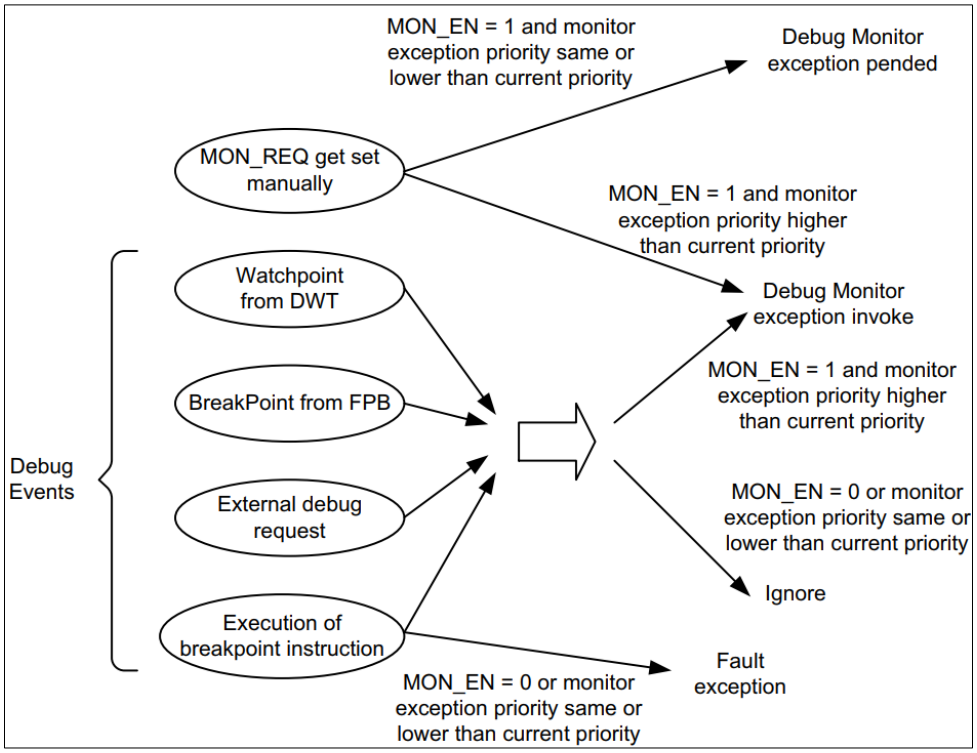


图 2-1 调试事件

本文实现的 Debug Monitor 调试功能基于 FPB 和 DWT 模块产生的调试事件。FPB 负责监测指令，当执行到所设置的指令断点时产生调试事件。DWT 负责数据跟踪，当读取或写入被监测数据时，产生调试事件。调试事件信号源可以通过寄存器 SCB->DFSR 的各个位判断，如图 2-2 所示。

Event cause	Exception support	DFSR bit	Notes
Internal halt request	Halt and DebugMonitor	HALTED	Step command, processor halt request, and similar
Breakpoint	Halt and DebugMonitor	BKPT	Breakpoint from BKPT instruction or match in FPB
Watchpoint	Halt and DebugMonitor	DWTTRAP	Watchpoint match in DWT, including PC match watchpoint
Vector catch	Halt only	VCATCH	One or more DEMCR.VC_* bits set to 1, and the processor took the corresponding exception
External	Halt and DebugMonitor	EXTERNAL	External Debug Request asserted

图 2-2 调试事件与其来源

Breakpoint 与 Watchpoint 的实现机制不同——Breakpoint 发生在指令执行前，而 Watchpoint 发生在指令执行后。意味着在 DebugMon_Handle 中对两者有不同的操作，下文说明。

2.3 FPB

FPB 核心由 8 个比较器组成。FP_COMP[0]~FP_COMP[5]进行指令比较，FP_COMP[6]~FP_COMP[7]进行文字池比较。设计为两个功能：提供硬件断点（Breakpoint）功能和代码修补（Flashpatch）功能。本文使用其硬件断点功能产生指令断点事件。即 FP_COMP 寄存器存储指令地址，当 CPU 取指的地址与 FPB 的 COMP 匹配时，FPB 向处理器内核生成断点调试事件，进而触发 Debug Monitor 中断。

断点调试事件发生在 CPU 取指前。如果未能在 Debug Monitor 中禁用或删除该断点地址，则 CPU 在执行完当前 Debug Monitor 中断处理函数之后，CPU 试图再次从同样的地址取指时，FPB 会再次产生断点调试事件，造成反复进入中断的情况。因此，需要在 DebugMon_Handle 或者其回调函数中删除所设置的断点，或者使所设置的断点失效。

2.4 DWT

Cortex-M4 的 DWT 是一个具有 COMP0~COMP3 共 4 个比较器的数据跟踪单元。每个比较器都可以被设置为监控数据地址（即变量地址）。配置后，当 CPU 访问的地址与比较器中预设的值匹配时，就会触发一个监测点调试事件，进而触发 Debug Monitor 中断。

无论数据断点的观察方式设置为读取、写入或者读写，监测点调试事件总是在数据访问完成之后产生。因此，该调试事件不会像 FPB 的断点调试事件那样，使 CPU 陷入 DebugMon_Handler 中断函数的死循环中。

3 实现

本文实现了一个未开启 FPU 的简单 Debug Monitor 调试样例。支持一个数据监测点和一个指令断点。用户调试指令的下发和 MCU 硬件信息回显通过串口实现。

3.1 Debug Monitor 配置

包括两个步骤：开启 Debug Monitor，以捕获调试事件；实现 DebugMon_Handler 中断处理函数，以保存调试事件发生时的上下文，以及输出这些上下文信息。

3.1.1 开启非侵入式调试

配置步骤包括 Debug Monitor 功能配置和其中断优先级配置。

- Debug Monitor 功能配置

主要关注两个寄存器：CoreDebug->DHCSR 和 CoreDebug->DEMCR。

CoreDebug->DHCSR 寄存器 C_DEBUGEN 位负责使能侵入式调试，在上电复位后为 0，在侵入式调试中由调试器设置为 1。Cortex-M4 的侵入式调试功能和非侵入式调试功能互斥，当该位置 1 后，非侵入调试 Debug Monitor 便不能实现。因此，在 DEMCR 配置前，需要先保证 CoreDebug -> DHCSR.C_DEBUGEN 为 0。

CoreDebug->DEMCR 的 TRCENA 位控制追踪模块 DWT、ETM、ITM 和 TPIU 的开启，要使用 DWT 必须将该位置 1。

CoreDebug->DEMCR 的 MON_EN 用于开启 debug monitor 中断，置 1 后当发生调试事件时，CPU 进入 DebugMon_Handle 执行用户定义的调试逻辑。

- Debug Monitor 优先级配置

使用 SCB->SHP[8]合理配置优先级，可以控制调试时 CPU 进入 DebugMon_Handler 时机，以便紧急程度高的任务运行不受调试影响。本文将 Debug Monitor 异常的优先级设置为高于串口中断处理函数、低于其他中断。

3.1.2 DebugMon_Handler

中断处理函数（ISR）一般流程：进入 ISR 时，硬件会自动保存 xPSR、PC、LR、R12、R3-R0 到堆栈。而 R4-R11、EXC_RETURN 等寄存器需要手动保存。为了提供上下文基本信息，这些寄存器的值都需要被保存。

DebugMon_Handler 在启动文件中采用了弱定义的默认处理函数，因此该中断处理函数需要自定义以替换启动文件中的原本弱函数。关于中断处理函数，汇编指令和 C 代码都可以实现。但若使用 C 代码，函数在跳转时可能会破坏 R4-R11、EXC_RETURN 的值。因此，DebugMon_Handler 应采用汇编代码实现上下文保存，之后进入用户自定义的回调函数进行调试操作，最后恢复现场退出中断处理函数。

表 3-2 为本文在 debug_monitor.S 中实现的中断处理函数 DebugMon_Handler。在程序的开头保存了 R4-R11、EXC_RETURN，加上硬件保存的栈帧，共 12 个寄存器。这些寄存器值在内存中从小到大的顺序为：R4-R11、EXC_RETURN、R0-R3、R12、LR、PC、xPSR。上述寄存器按照结构进行组织，如表 3-1 所示。

表 3-1 栈帧及上下文结构

```
struct stack_frame
{
    uint32_t r0;
    uint32_t r1;
    uint32_t r2;
    uint32_t r3;
    uint32_t r12;
    uint32_t lr;
    uint32_t pc;
    uint32_t xpsr;
};

struct exception_info
{
    /* r4 - r11 register */
    uint32_t r4;
    uint32_t r5;
    uint32_t r6;
    uint32_t r7;
    uint32_t r8;
    uint32_t r9;
    uint32_t r10;
    uint32_t r11;
    uint32_t exc_return;
    struct stack_frame sf;
};
```

上述表格中，struct stack_frame 表示硬件自动处理的栈帧所含寄存器，struct exception_info 表示硬件上下文所含寄存器。

表 3-2 中断处理函数 DebugMon_Handler

```
R_CFSR    EQU    0xE000ED28
R_HFSR    EQU    0xE000ED2C
R_MMFAR   EQU    0xE000ED34
R_BFAR    EQU    0xE000ED38
```

```
    PRESERVE8
    THUMB
    AREA    |.text|, CODE, READONLY
DebugMon_Handler    PROC
    EXPORT  DebugMon_Handler
    IMPORT  DebugMonitorCallBack
    TST     lr, #0x04                   ; if(!EXC_RETURN[2])
    ITE     EQ
    MRSEQ   r0, msp                     ; [2]=0 ==> Z=1, get fault context from handler.
    MRSNE   r0, psp                     ; [2]=1 ==> Z=0, get fault context from thread.
    STMFD   r0!, {r4 - r11, lr}     ; push r4 - r11 register, lr is EXC_RETURN
    TST     lr, #0x04                   ; if(!EXC_RETURN[2])
    ITE     EQ
    MSREQ   msp, r0                     ; [2]=0 ==> Z=1, update stack pointer to MSP.
    MSRNE   psp, r0                     ; [2]=1 ==> Z=0, update stack pointer to PSP.
    BL      DebugMonitorCallBack
    POP     {r4 - r11, lr}
    BX      lr
    ENDP
```

表 3-3 为 Debug Monitor 的回调函数，在其中主要执行两个事务：FPB 硬件断点调试事件功能清除和调试信息打印。

表 3-3　回调函数 DebugMonitorCallBack

```
void DebugMonitorCallBack(struct exception_info *ecpt_info)
{
    NVIC_ClearPendingIRQ(DebugMonitor_IRQn); /* Clear pending */
    if (SCB->DFSR & SCB_DFSR_BKPT_Msk) {
        if (FPB->CTRL & 0x01UL) {
            /* Disable FPB breakpoint */
            FPB->CTRL = (FPB->CTRL | 0x02UL) & (~0x01UL);
        }
    }
    __DSB();
    __ISB();
    DDL_Printf("\nDebug Monitor: Actived\n");
    DDL_Printf("Info Addr %p: \n", ecpt_info);
    InfoPrint(ecpt_info);
}
```

在回调处理逻辑中，先判断调试事件是否由 FPB 引起。若是，则清除 FPB 的比较使能位。当 FPB 的比较使能位清除后，便不会产生新的断点调试事件。

在调试信息打印逻辑程序中，完成了上下文寄存器的输出。这些寄存器反映了 CPU 进入 DebugMon_Handler 前的硬件环境。

3.2 FPB 和 DWT 配置

当程序运行到设定的某条指令或访问某个数据时，便会触发调试事件，使 CPU 进入 DebugMon_Handler 执行。这里的调试事件由 FPB 或者 DWT 触发。

3.2.1 FPB 配置

配置 FPB 是一个动态过程——每次配置对应一个指令断点调试事件。本文配置 FPB 主要操作 FPB->CTRL 和 FPB->COMP[0]两个寄存器。

表 3-4 FPB->CTRL

Bits	Name	Type	Reset Value
31:15	-	-	-
14:12	NUM_CODE2	RO	0b000
11:8	NUM_LIT	RO	0/2
7:4	NUM_CODE1	RO	0/2/6
3:2	-	-	-
1	KEY	W	-
0	ENABLE	R/W	0

FPB->CTRL.ENABLE 复位后为 0，FPB 未使能，设置为 1 使能 FPB 模块。该位的写入操作需要 KEY 位配合，即 ENABLE 位置 1 时写入 FPB->CTRL|=0x3UL，ENABLE 位清零时写入 FPB->CTRL = (FPB->CTRL | 0x02UL) & (~0x01UL)。

表 3-5 FPB->COMP0

Bits	Name	Type	Reset Value
31:30	REPLACE	R/W	0b00
29	-	-	-
28:2	COMP	R/W	-
1	-	-	-
0	ENABLE	R/W	0

FPB->COMP[0].COMP 设置需要产生指令断点调试事件的地址，该字段含[28:2]共 27 位。理论上，一条 Cortex-M4 的指令地址范围在 0x0000 0000~0x1FFF FFFF 之间，这个范围可用 29 位二进制数表示。另外，CPU 取指操作默认 4 字节对齐，即低 2 位为 0。所以，27 位的 COMP 完全能够表达指令地址。

FPB->COMP[0].REPLACE 是对 FPB->COMP[0].COMP 的补充，后者包含 4 字节对齐的指令地址。CPU 获取 32 位的指令后，还需要通过 FPB->COMP[0].REPLACE 值确定实际的断点地址位于高半字还是低

半字。FPB->COMP[0].REPLACE=0x01，表示 CPU 取到的一个字中（32 位指令），低半字是目标指令；FPB->COMP[0].REPLACE=0x02，表示 CPU 取到的一个字中（32 位指令），高半字是目标指令；FPB->COMP[0].REPLACE=0x03，表示 CPU 取到的一个字中（32 位指令），高半字和低半字均是目标指令，即这个字的内容是一条完整的 32 位指令。注意，对于一个存储在非 4 字节对齐地址的指令，设置其断点需要 FPB 两个 FPB->COMPx 配合，本文不做详细分析。

FPB->COMP[0].ENABLE 复位后为 0，表示寄存器 FPB->COMP[0]不参与比较。设置指令断点时需要将该位置 1。

3.2.2 DWT 配置

DWT 产生的监测点调试异常发生在数据被访问之后，无需动态去配置一个同样的监测点。本文配置 DWT 主要关注两个寄存器：DWT->COMP0 和 DWT->FUNCTION0。

表 3-6 DWT->COMP0

Bits	Name	Type	Reset Value
31:0	COMP	R/W	-

DWT->COMP0 只有 COMP 一个段，可设置数据的 32 位地址作为监测点。

表 3-7 DWT->FUNCTION0

Bits	Name	Type	Reset Value
31:25	-	-	0b00
24	MATCHED	RO	-
23:20	-	-	-
19:16	DATAVADDR1	R/W	-
15:12	DATAVADDR0	R/W	0
11:10	DATAVSIZE	R/W	-
9	LNK1ENA	RO	-
8	DATAVMATCH	R/W	-
7	CYCMATCH	R/W	-
6	-	-	-
5	EMITRANGE	-	-
4	-	-	-
3:0	FUNCTION	R/W	0

DWT->FUNCTION0 寄存器的功能较多，本文仅阐述设置一个数据（变量）监测需要用到的位段。通过这些位段的配置，DWT 可以产生数据监测点调试事件。

DWT->FUNCTION0.MATCHED，读清零，表示自上次读取之后又有地址匹配的情况发生。

DWT->FUNCTION0.DATAVSIZE 指示 DWT->COMP0 寄存器所设置的地址对应的数据宽度。0x00U，表示数据大小为 1 字节；0x01U，表示数据大小为 2 字节；0x02U，表示数据大小为 4 字节。

DWT->FUNCTION0.FUNCTION，指示比较器 COMP0 的具体功能。与数据监测点调试事件相关的几个配置含义如下：0x05U（0b0101），当匹配的地址内数据完成读操作时产生 Watchpoint 调试异常；0x06U（0b0110），当匹配的地址内数据完成写操作时产生 Watchpoint 调试异常；0x07U（0b0111），当匹配的地址内数据完成读或写操作时产生 Watchpoint 调试异常。

3.3 串口配置

串口在本文的作用是与用户交互，接收来自用户的断点信息，发送调试过程中上下文等信息。

3.3.1 硬件配置

启用串口发送、接收、接收中断功能，在接收中断函数中处理调试协议。

3.3.2 协议开发

本文定义了一个简单的协议，如表 3-8 所示。注意，协议中字符全部为小写。

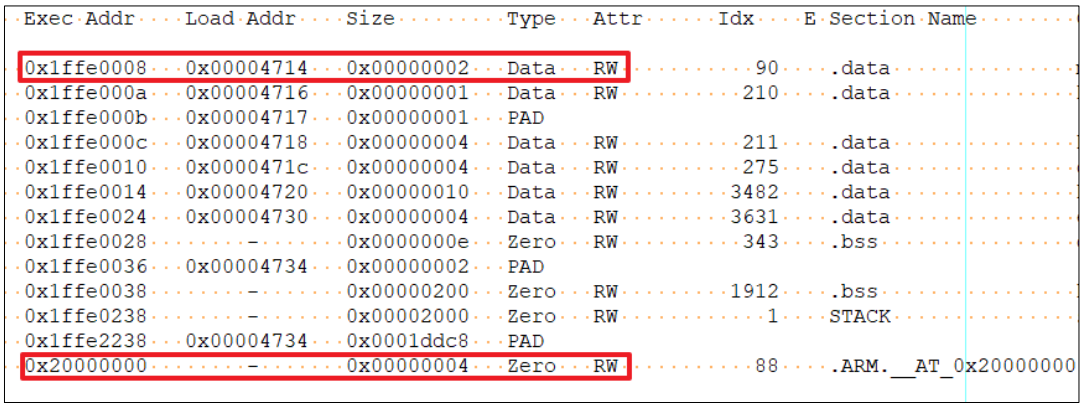
表 3-8 本文定义的简单调试协议

顺序	内容	大小	说明
0	数据类型	1byte	‘i’，表示指令断点 ‘d’，表示数据断点
1	操作类型	1byte	‘s’，设置断点 ‘c’，取消断点
2	指令或数据宽度	1byte	‘b’，1byte，仅数据监测点 ‘s’，2byte ‘w’，4byte
3	断点地址	4byte	带前缀‘0x’的 8 位 16 进制的地址
4	数据结束标志	1byte	‘#’表示数据结束

例如，通过串口窗口发送“iss0x00000e98#”，表示在 0x0000 0e98 设置指令断点，指令长度为 2byte。发送“dss0x1ffe0008#”，表示在 0x0000 0e98 设置数据监测点，数据宽度为 2byte。发送“dcs0x1ffe0008#”，表示取消 0x1ffe 0008 设置的数据监测点。

4 结果示例

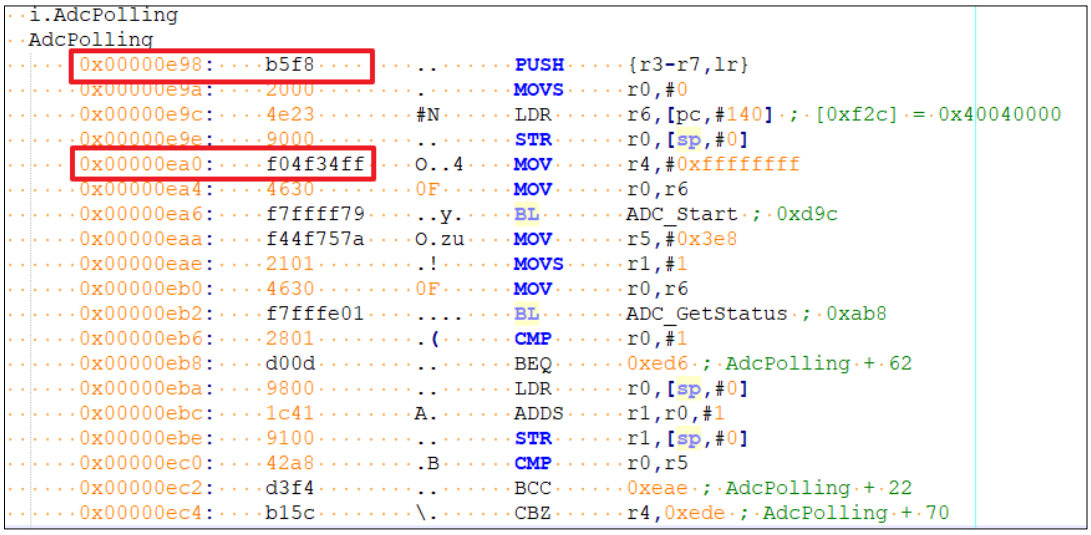
Debug Monitor 需结合 map 文件查询要设置的监测点/断点的数据或者指令地址。如图 4-1 所示，通过查询 map 文件选取两个数据监测点：位于 SRAMH 的 0x1FFE 0008 和位于 SRAM1 的 0x2000 0000。



Exec Addr	Load Addr	Size	Type	Attr	Idx	E	Section Name
0x1ffe0008	0x00004714	0x00000002	Data	RW	90		.data
0x1ffe000a	0x00004716	0x00000001	Data	RW	210		.data
0x1ffe000b	0x00004717	0x00000001	PAD				
0x1ffe000c	0x00004718	0x00000004	Data	RW	211		.data
0x1ffe0010	0x0000471c	0x00000004	Data	RW	275		.data
0x1ffe0014	0x00004720	0x00000010	Data	RW	3482		.data
0x1ffe0024	0x00004730	0x00000004	Data	RW	3631		.data
0x1ffe0028	-	0x0000000e	Zero	RW	343		.bss
0x1ffe0036	0x00004734	0x00000002	PAD				
0x1ffe0038	-	0x00000200	Zero	RW	1912		.bss
0x1ffe0238	-	0x00002000	Zero	RW	1		STACK
0x1ffe2238	0x00004734	0x0001ddc8	PAD				
0x20000000	-	0x00000004	Zero	RW	88		.ARM.__AT_0x20000000

图 4-1 本文选取的两个数据监测点

如图 4-2 所示，通过查询汇编文件（也可直接查询 map 文件），选取 0x0000 0E98 处的 16 位指令和 0x0000 0EA0 处的 32 位指令作为断点地址。



```
i.AdcPolling
AdcPolling
    0x00000e98:    b5f8        ..      PUSH    {r3-r7,lr}
    0x00000e9a:    2000        .       MOVS    r0,#0
    0x00000e9c:    4e23        #N      LDR     r6,[pc,#140] ; [0xf2c] = 0x40040000
    0x00000e9e:    9000        ..      STR     r0,[sp,#0]
    0x00000ea0:    f04f34ff    O..4    MOV     r4,#0xffffffff
    0x00000ea4:    4630        0F      MOV     r0,r6
    0x00000ea6:    f7ffff79    ..y.    BL      ADC_Start ; 0xd9c
    0x00000eaa:    f44f757a    O.zu    MOV     r5,#0x3e8
    0x00000eae:    2101        .!      MOVS    r1,#1
    0x00000eb0:    4630        0F      MOV     r0,r6
    0x00000eb2:    f7fffe01    ....    BL      ADC_GetStatus ; 0xab8
    0x00000eb6:    2801        .(      CMP     r0,#1
    0x00000eb8:    d00d        ..      BEQ     0xed6 ; AdcPolling + 62
    0x00000eba:    9800        ..      LDR     r0,[sp,#0]
    0x00000ebc:    1c41        A.      ADDS    r1,r0,#1
    0x00000ebe:    9100        ..      STR     r1,[sp,#0]
    0x00000ec0:    42a8        .B      CMP     r0,r5
    0x00000ec2:    d3f4        ..      BCC     0xeae ; AdcPolling + 22
    0x00000ec4:    b15c        \.      CBZ     r4,0xede ; AdcPolling + 70
```

图 4-2 本文选取的两个指令断点

4.1 设置指令断点

设置两个长度不同的指令进行观察。

4.1.1 16 位指令

将数据类型‘i’、操作类型‘s’、数据宽度‘s’、数据地址 ‘0x0000 0e98’、结束标志‘#’按协议规范组合，通过串口发送至 MCU，结果如图 4-3 所示。

```
iss0x00000e98#FPB[0] Breakpoint at 0xE98 Set

Debug Monitor: Actived
Info Addr 1ffe21dc:
 xPSR: 0x61000000
  R00: 0xFFFFFFFF
  R01: 0x00000000
  R02: 0x00000536
  R03: 0xFFFFFFFF
  R04: 0x00001388
  R05: 0x000047F0
  R06: 0x00000000
  R07: 0x00000000
  R08: 0x77FCFDFE
  R09: 0x1FFE06B8
  R10: 0x15DD6EAD
  R11: 0xF4D9D0E7
  R12: 0x00000014
   LR: 0x0000452F
   PC: 0x00000E98
```

图 4-3　设置 16 指令断点

4.1.2　32 位指令

将数据类型‘i’、操作类型‘s’、数据宽度‘w’、数据地址 ‘0x0000 0ea0’、结束标志‘#’按协议规范组合，通过串口发送至 MCU，结果如图 4-4 所示。

```
isw0x00000ea0#FPB[0] Breakpoint at 0xEA0 Set

Debug Monitor: Actived
Info Addr 1ffe21c4:
 xPSR: 0x61000000
  R00: 0x00000000
  R01: 0x00000000
  R02: 0x00000536
  R03: 0xFFFFFFFF
  R04: 0x00001388
  R05: 0x000047F0
  R06: 0x40040000
  R07: 0x00000000
  R08: 0x77FCFDFE
  R09: 0x1FFE06B8
  R10: 0x15DD6EAD
  R11: 0xF4D9D0E7
  R12: 0x00000014
   LR: 0x0000452F
   PC: 0x00000EA0
```

图 4-4　设置 32 指令断点

4.2　设置数据断点

设置两个数据类型不同、地址区域不同的变量进行观察。

4.2.1　SRAMH

将数据类型‘d’、操作类型‘s’、数据宽度‘s’、数据地址 ‘0x1ffe 0008’、结束标志‘#’按协议规范组合，通过串口发送至 MCU，结果如图 4-5 所示。

```
dss0x1ffe0008#DWT Breakpoint at 0x1FFE0008 Set

Debug Monitor: Actived
Info Addr 1ffe21c4:
 xPSR: 0x81000000
  R00: 0x00001234
  R01: 0x0000000D
  R02: 0x00000688
  R03: 0x00000000
  R04: 0x0000081C
  R05: 0x000003E8
  R06: 0x40040000
  R07: 0x00000000
  R08: 0x77FCFDFE
  R09: 0x1FFE06B8
  R10: 0x15DD6EAD
  R11: 0xF4D9D0E7
  R12: 0x00000014
   LR: 0x00000F09
   PC: 0x00000F14
```

图 4-5 在 SRAMH 设置数据断点

4.2.2 SRAM1

将数据类型‘d’、操作类型‘s’、数据宽度‘w’、数据地址 ‘0x2000 0000’、结束标志‘#’按协议规范组合，通过串口发送至 MCU，结果如图 4-6 所示。

```
dsw0x20000000#DWT Breakpoint at 0x20000000 Set

Debug Monitor: Actived
Info Addr 1ffe21c4:
 xPSR: 0x81000000
  R00: 0x0000081C
  R01: 0x20000000
  R02: 0x00000536
  R03: 0xFFFFFFFF
  R04: 0x0000081C
  R05: 0x000003E8
  R06: 0x40040000
  R07: 0x00000000
  R08: 0x77FCFDFE
  R09: 0x1FFE06B8
  R10: 0x15DD6EAD
  R11: 0xF4D9D0E7
  R12: 0x00000014
   LR: 0x00000EF1
   PC: 0x000003E0
```

图 4-6 在 SRAM1 设置数据断点

5 总结

本文介绍了 Cortex-M4 的 Debug Monitor 调试功能，并使用 HC32F4A0 进行简单演示。用户可参考本文，按照实际情况实现所需的非侵入式调试。

版本修订记录

版本号	修订日期	修订内容
Rev1.00	2026/04/16	初版发布。