使用 S32DS+PE Micro Multilink 调试代码的一些小技巧

Alvin Liu

2022-08-19

这篇文章总结了 20 个使用 S32DS+PE Micro Multilink 调试代码的小技巧，希望可以帮助大家提高调试的效率，快速的定位问题。

注:

该文章里提到的 S32DS 版本为 V3.4，PE Micro Multilink 版本为 Rev C.主要讲解基于 S32K3XX MCU 的调试，但里面许多调试方法也使用 Power PC 和 S32K1XX 的调试。

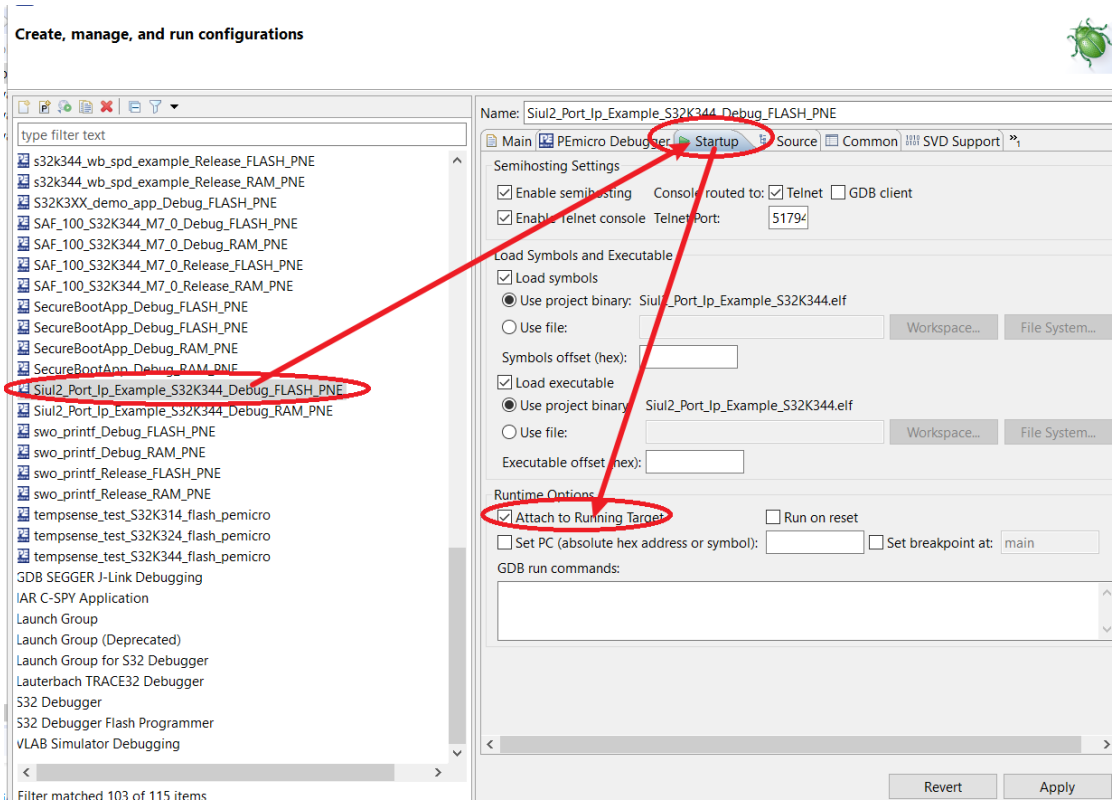
目录

1. 把仿真器 Attach 到运行的 MCU 上

Attach 功能主要是在不复位 MCU，不破坏 MCU 当前的运行状态下，建立调试器和调试目标之间的连接。当脱离仿真器运行的目标出现工作异常，可以通过 Attach 功能查看当前 MCU 的状态信息。

Debug Configurations->Startup->Runtime Options

使能 Attach to Running Target，去掉 Run on reset 和 Set breakpoint at

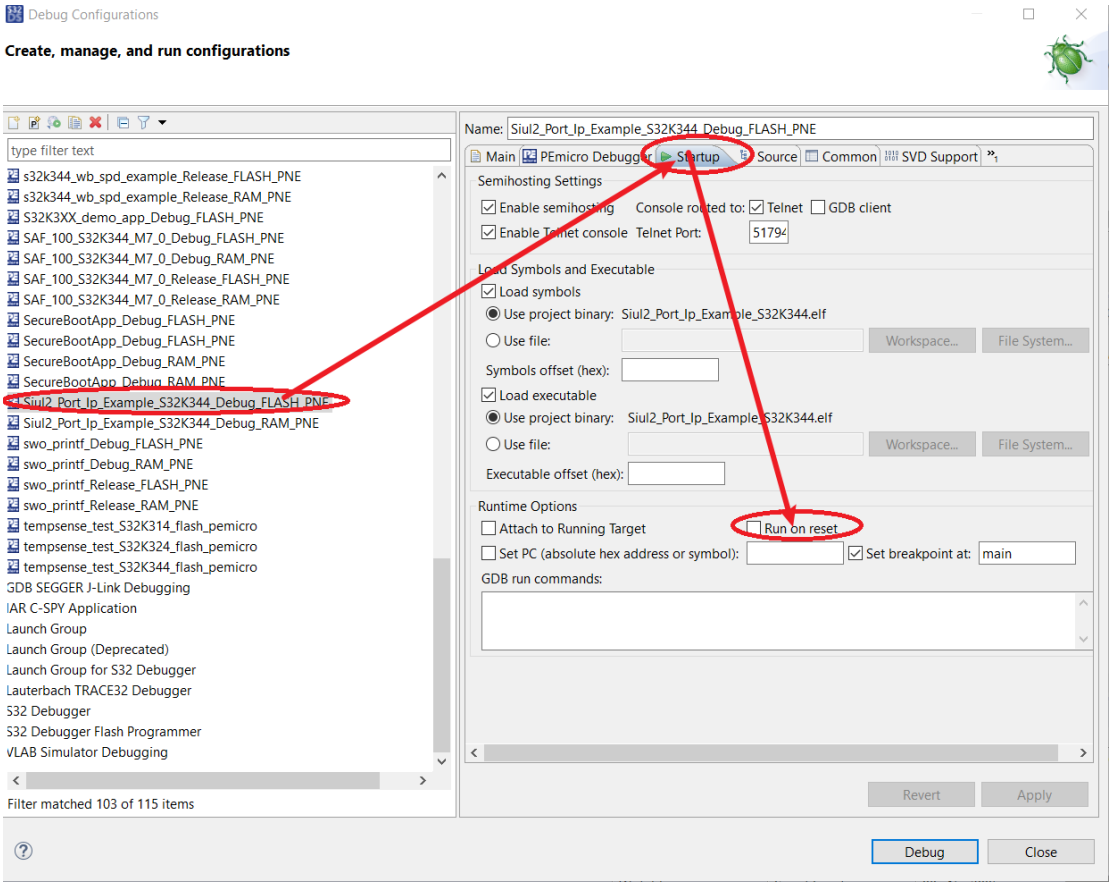


2. 调试 Startup Code

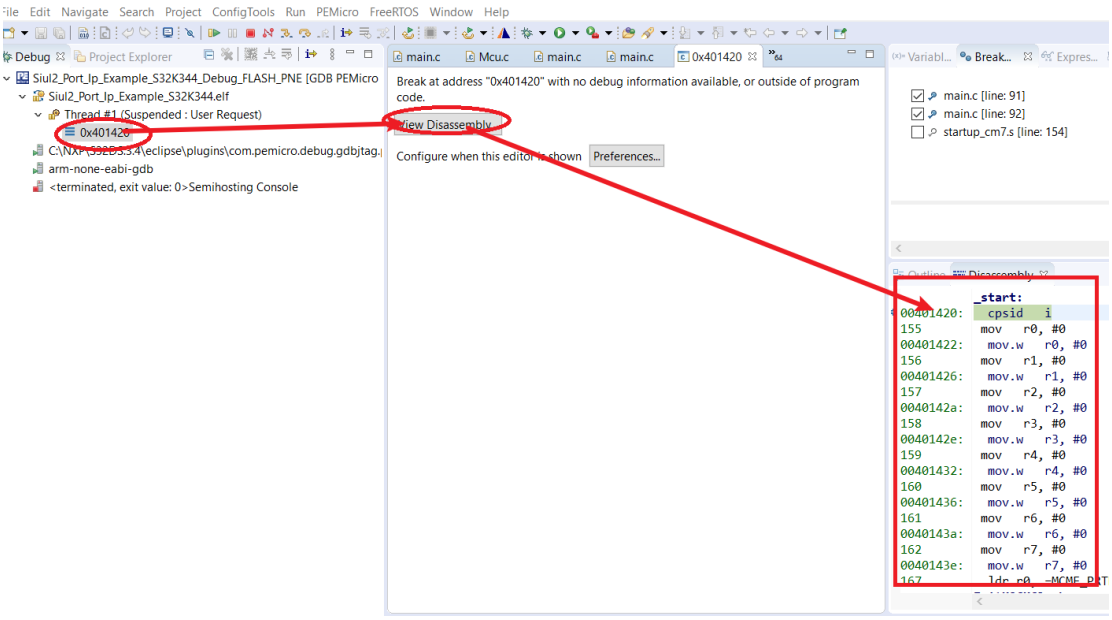
S32DS 调试工程配置的默认断点为 Main 函数，这样启动 debug 运行后，就会停在 Main 函数最开始的地方，如果要调试 Startup Code，希望断点停在 Startup Code 最开始的地方，及 Reset_Handler 处，可以这样配置

Debug Configurations->Startup->Runtime Options

去掉“Run on reset”



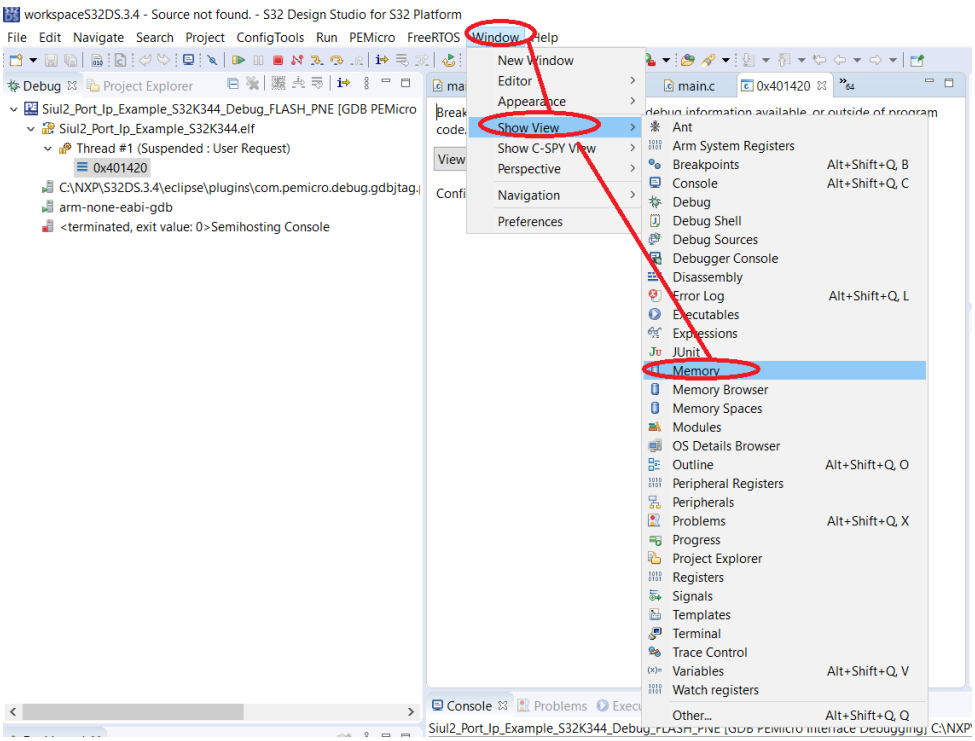
启动 debug 后，程序停在了 Reset_Handler 处。但是 Statup Code 的调试信息并没有加载进来(可能是 S32DS 的一个 Bug)，接下里可以通过汇编调试 Startup Code，或在调试器上执行下复位，MCU 复位再次起来后 Startup Code 的调试信息可以正常加载进来。



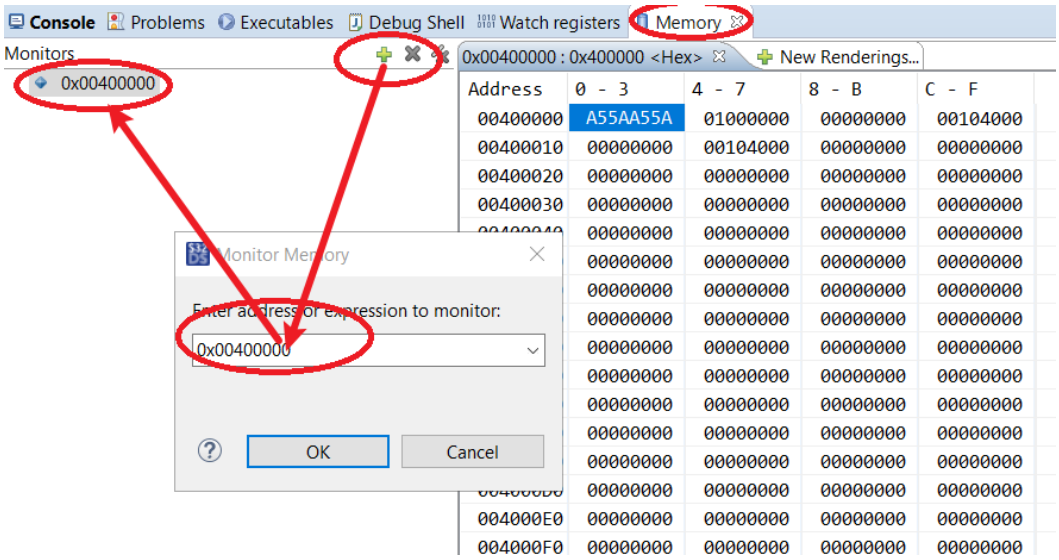
3. 浏览 Memory，导出 Memory 内容到文件

可以通过 Memory 调试窗口查看当前应用工程目标 MCU 的任意合法地址空间，如 Flash, SRAM, UTEST,外设寄存器上的值

通过 Window->Show View->Memory 打开 Memory Browse 窗口

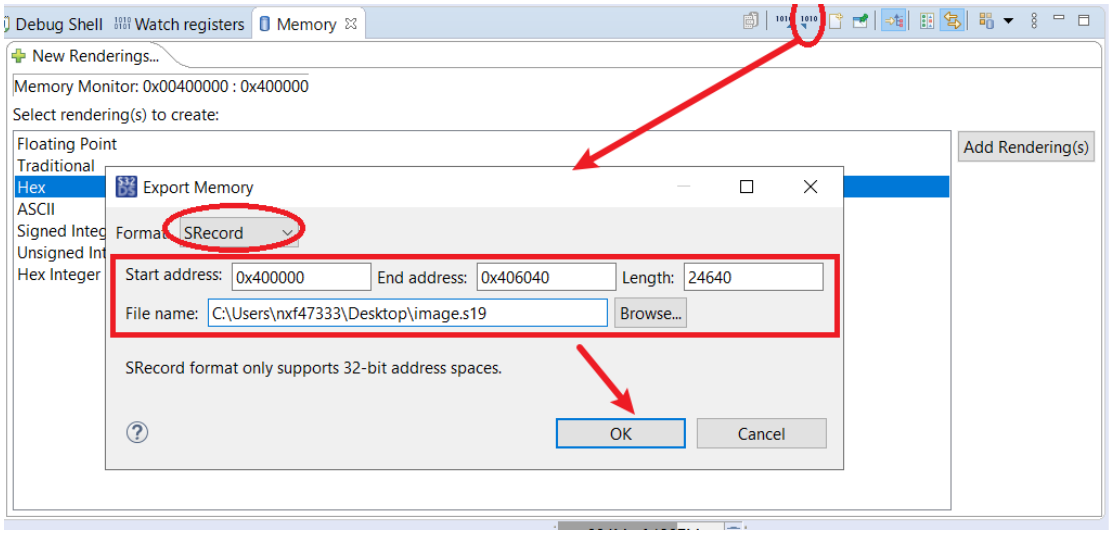


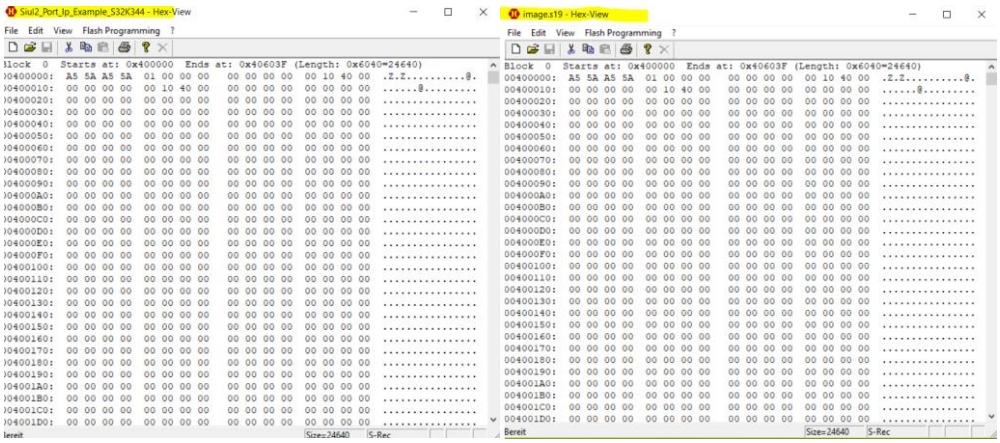
点击 Memory 窗口左侧的“+”，输入要查看的存储器地址，即可添加对该地址存储器中数据的查看



也可以将指定地址范围内的 Memory 数据导出到文件中。这个功能在故障件问题排查时有很大作用，对故障件我们要确定该 NG PART 的 Image 是否完整，我们可以将 Image 导出到文件

和正常件导出的做对比。Export Memory 支持 3 中导出格式，“Plain Test”，“Raw Binary”和“SRecord”

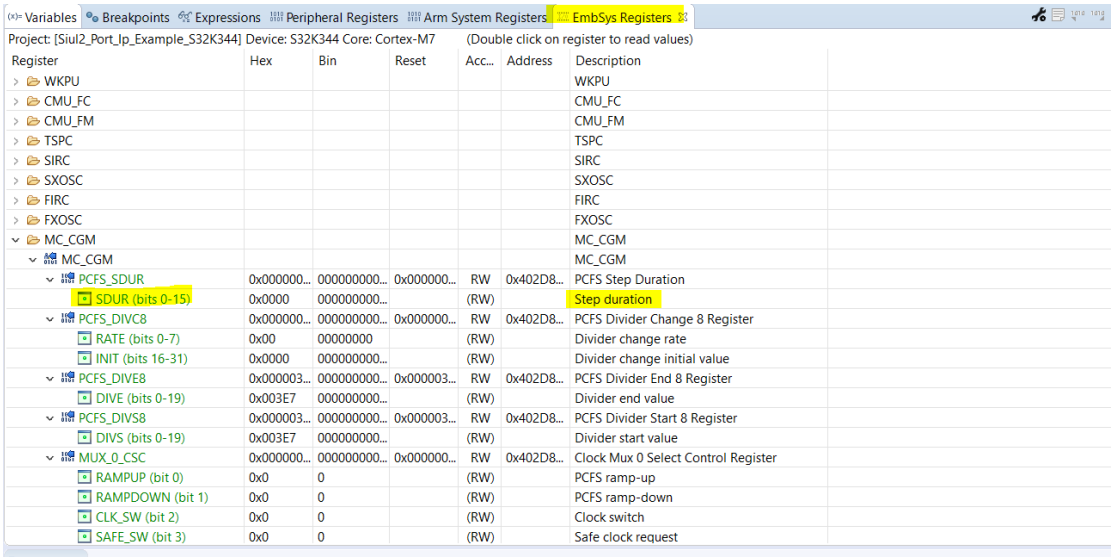




将 flash 上代码所占区间的数据导出到指定的 S19 文件，和编译生成的 S19 文件对比，以判断烧录的完整性。

4. 查看，修改外设寄存器，和导出寄存器信息到文件

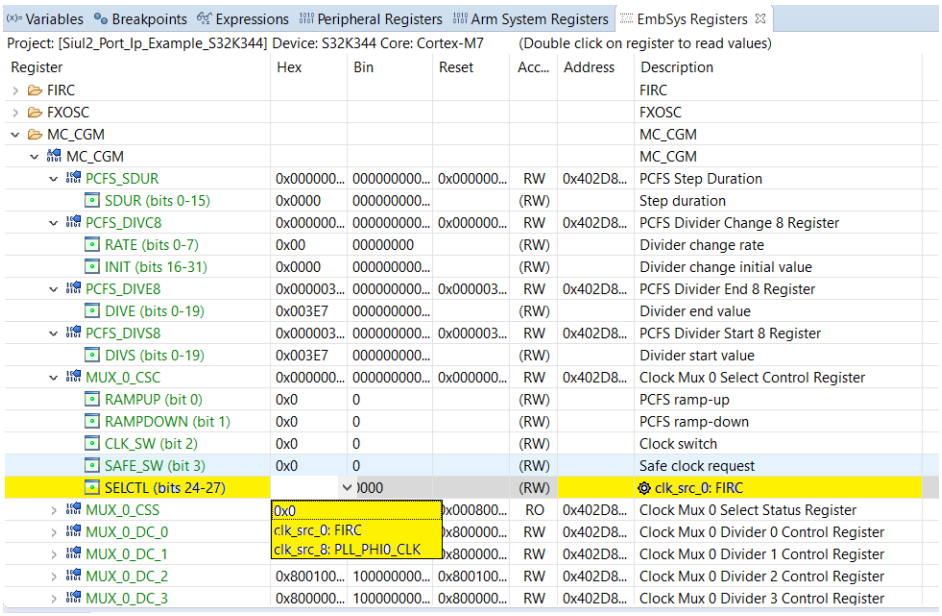
S32DS IDE 的调试界面提供了“EmbSys Registers”窗口用于查看和修改 MCU 外设寄存器，也可以将寄存器信息导出的文件。默认寄存器的值是不显示的，只有双击寄存器名如 MC_CGM 之后，调试器才会读取并显示寄存器的值，



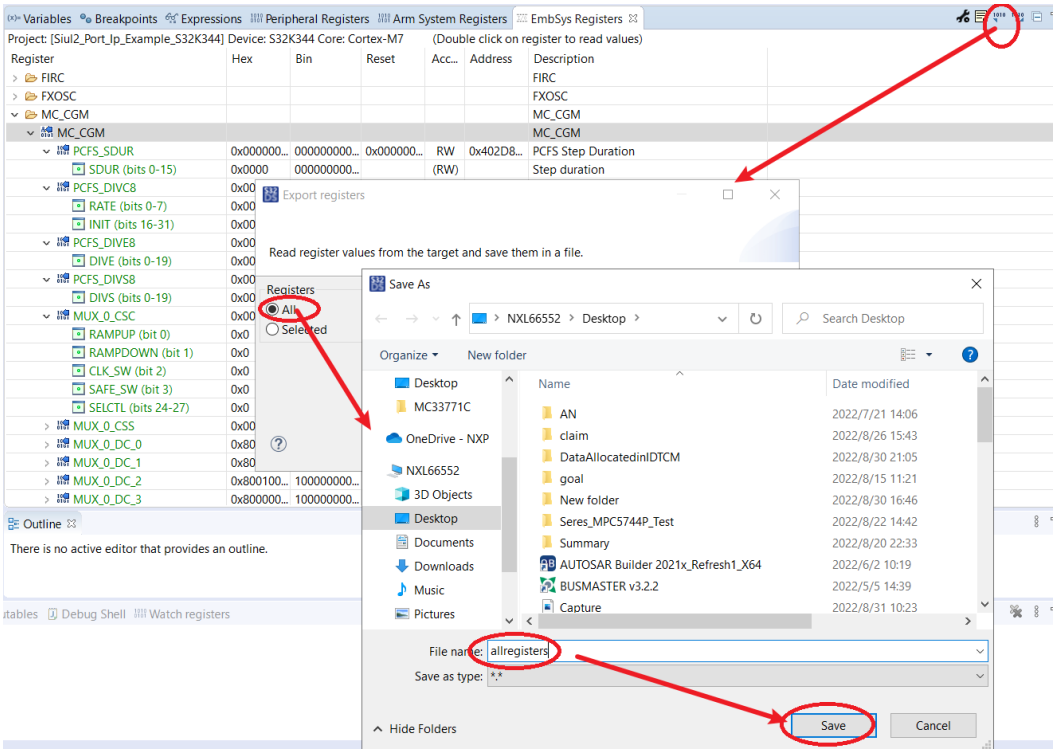
Register	Hex	Bin	Reset	Acc...	Address	Description
WKPU						WKPU
CMU_FC						CMU_FC
CMU_FM						CMU_FM
TSPC						TSPC
SIRC						SIRC
SXOSC						SXOSC
FIRC						FIRC
FXOSC						FXOSC
MC_CGM						MC_CGM
MC_CGM						MC_CGM
PCFS_SDUR	0x000000...	000000000...	0x000000...	RW	0x402D8...	PCFS Step Duration
SDUR (bits 0-15)	0x0000	000000000...		(RW)		Step duration
PCFS_DIVC8	0x000000...	000000000...	0x000000...	RW	0x402D8...	PCFS Divider Change 8 Register
RATE (bits 0-7)	0x00	00000000		(RW)		Divider change rate
INIT (bits 16-31)	0x0000	000000000...		(RW)		Divider change initial value
PCFS_DIVE8	0x000003...	000000000...	0x000003...	RW	0x402D8...	PCFS Divider End 8 Register
DIVE (bits 0-19)	0x003E7	000000000...		(RW)		Divider end value
PCFS_DIVS8	0x000003...	000000000...	0x000003...	RW	0x402D8...	PCFS Divider Start 8 Register
DIVS (bits 0-19)	0x003E7	000000000...		(RW)		Divider start value
MUX_0_CSC	0x000000...	000000000...	0x000000...	RW	0x402D8...	Clock Mux 0 Select Control Register
RAMPUP (bit 0)	0x0	0		(RW)		PCFS ramp-up
RAMPDOWN (bit 1)	0x0	0		(RW)		PCFS ramp-down
CLK_SW (bit 2)	0x0	0		(RW)		Clock switch
SAFE_SW (bit 3)	0x0	0		(RW)		Safe clock request

EmbSys Registers 窗口，不仅显示了寄存器的值，也将寄存器每个位域的值，及其代表的含义也呈现了出来。

EmbSys Registers 窗口还可以对寄存器的值做修改



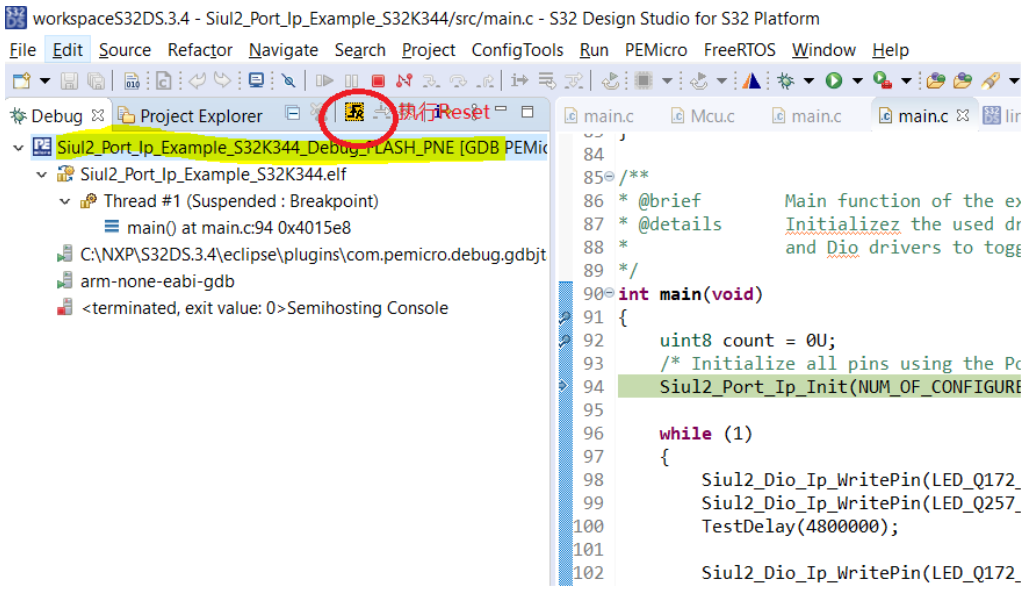
将寄存器信息导出到文件，可以导出所选的寄存器或将全部寄存器信息导出来。



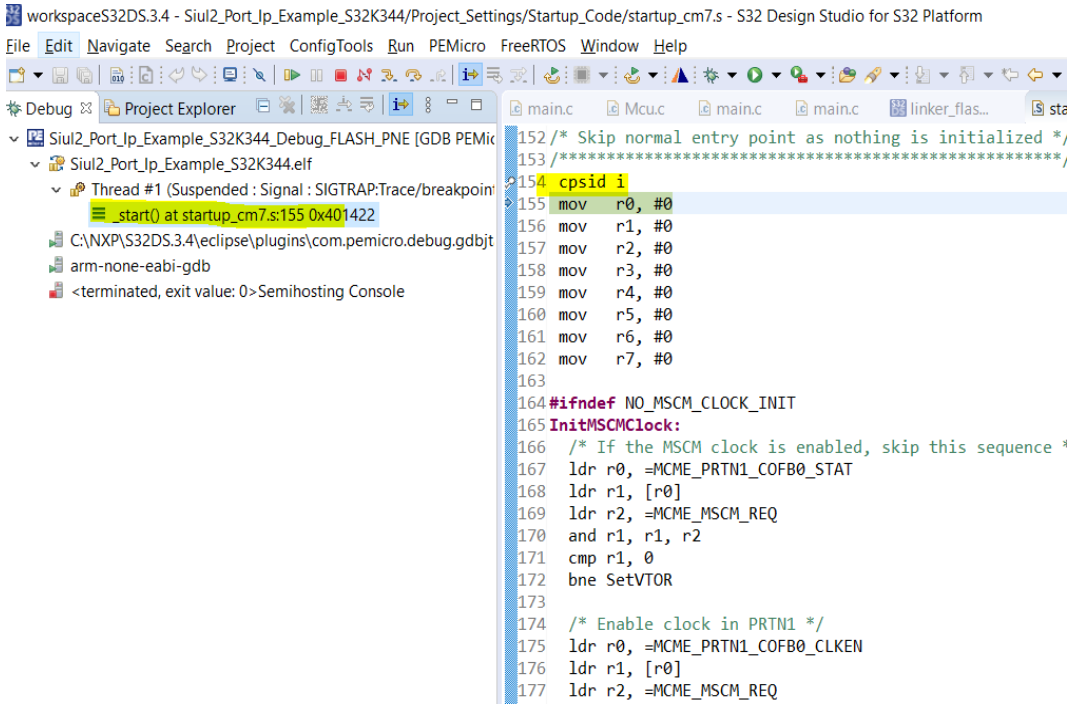
在问题排查时，将正常工作下的寄存器信息和异常下的寄存器信息对比有助于我们快速定位问题。

5. 在调试器上执行 Reset 操作

S32DS IDE 支持在调试器上执行复位操作，实际上是触发了 JTAG Reset，而 JTAG Reset 信号线连接到了 MCU 的外部复位脚，所以调试器上的 Reset 操作对应 MCU 完成了一次外部复位。

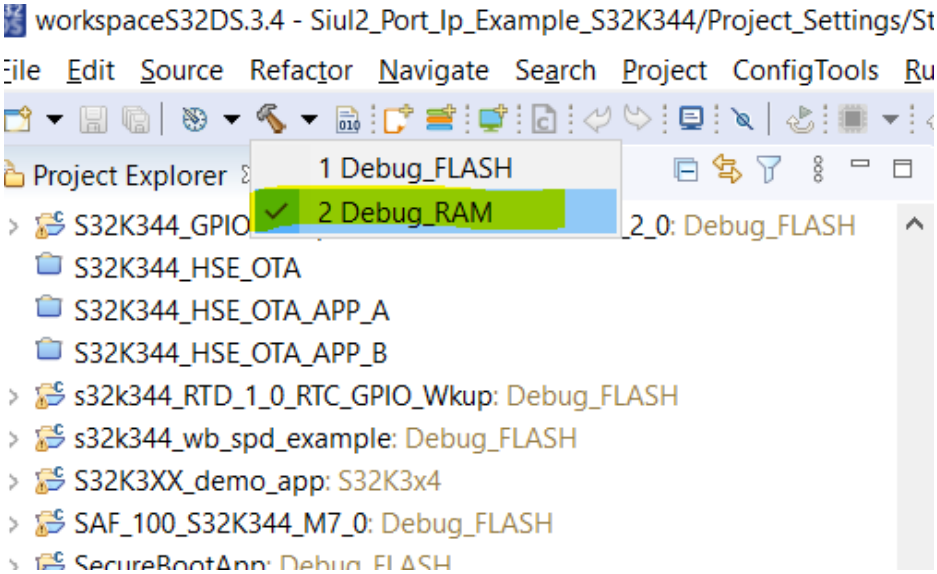


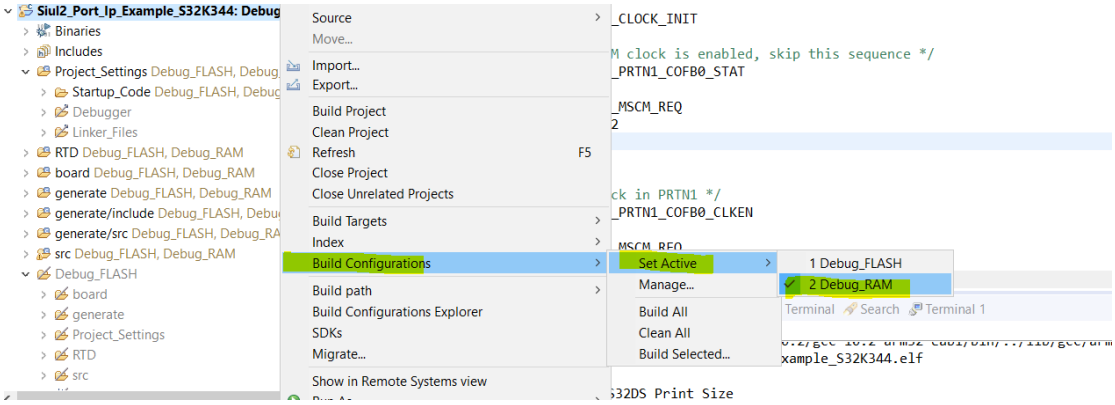
选中调试工程，点击复位按钮，MCU 会复位，复位后 MCU 停在 Startup 的第一条语句



6. 把代码下载到 RAM 中运行

S32DS IDE 构建的应用工程有两种运行模式，Debug_FLASH 和 Debug_RAM 前者生成的固件是要烧写到 MCU 的 Flash 上，从 Flash 上运行，而后者生成的固件则可以通过调试器直接加载到 MCU 的 RAM 中运行。Dedug_RAM 在故障件问题排查时有大用场，为了隔离软，硬件问题，需要运行测试代码，但又不能檫除 MCU 原有代码，檫除原有代码可能导致故障不复现，这时可以运行 Debug_RAM 代码。

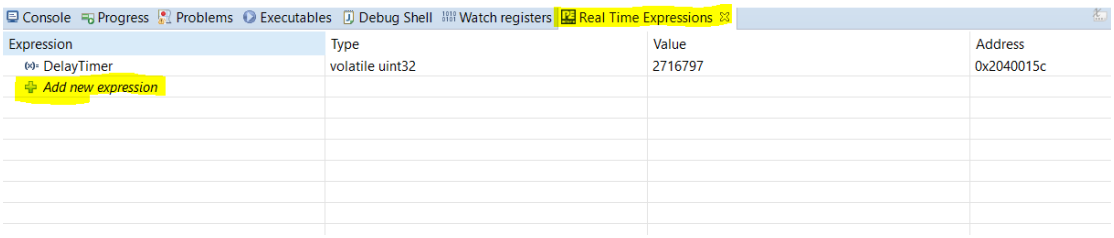




7. 实时显示全局变量

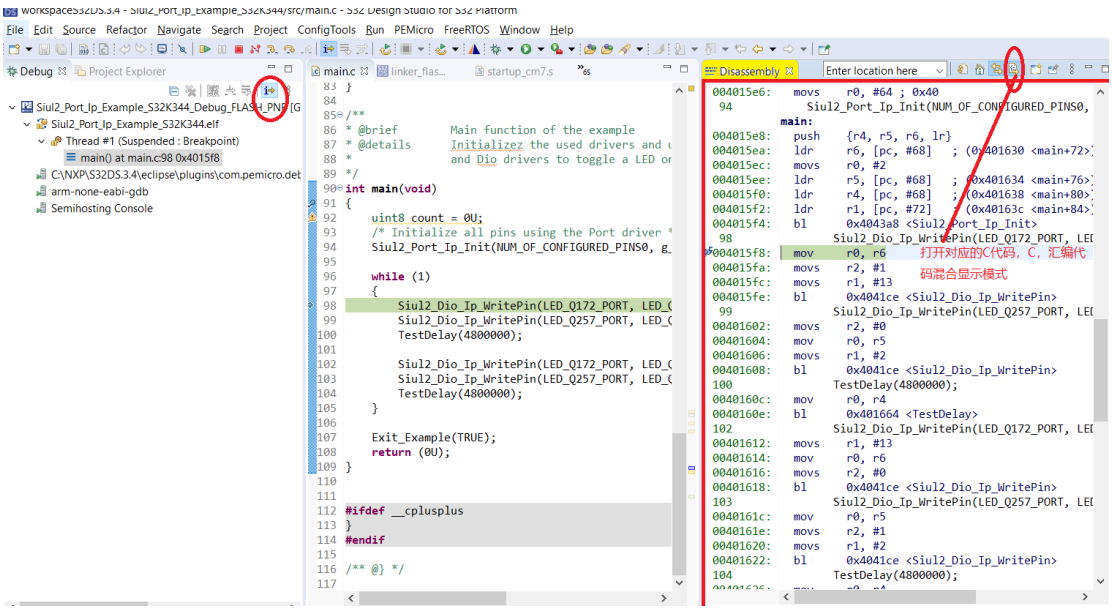
S32DS V3.4 支持在不停止 MCU 运行下，实时观察全局变量的值。

通过 Window->show view->other->PEmicro->Real Time Expressions 打开 Real Time Expressions 调试窗口，添加要观察的全局变量。当程序全速运行是，可以看到要观察的全局变量会周期行的刷新。



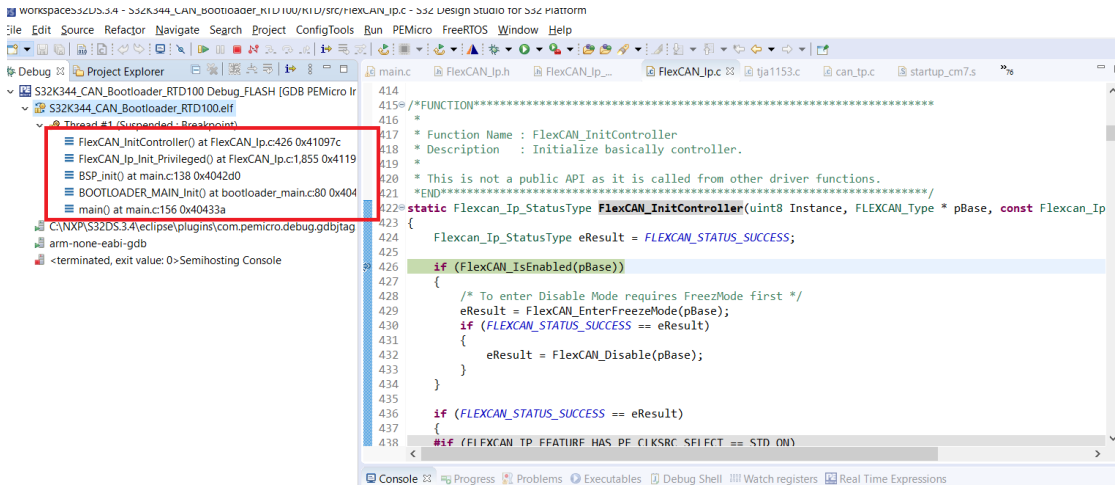
8. 汇编指令调试

有些情况下，需要深入到汇编指令级调试，可通过如下方式打开汇编调试窗口



9. 查看函数调用栈，分析函数调用过程

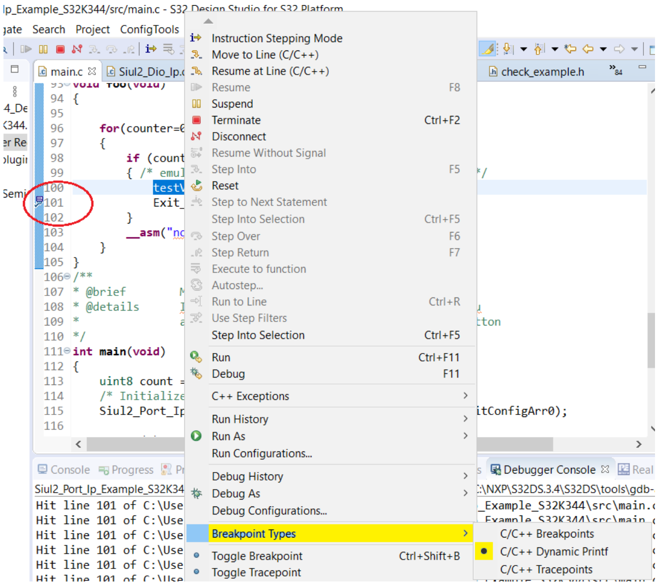
S32DS IDE 支持查看函数调用栈，可以分析函数调用过程，当程序遇到断点停下，或程序遇到某种异常跑飞进入 MCU 异常中断，或是手动暂停程序的运行。S32DS IDE 都会将 MCU 停下时刻，函数的调用栈呈现出来。



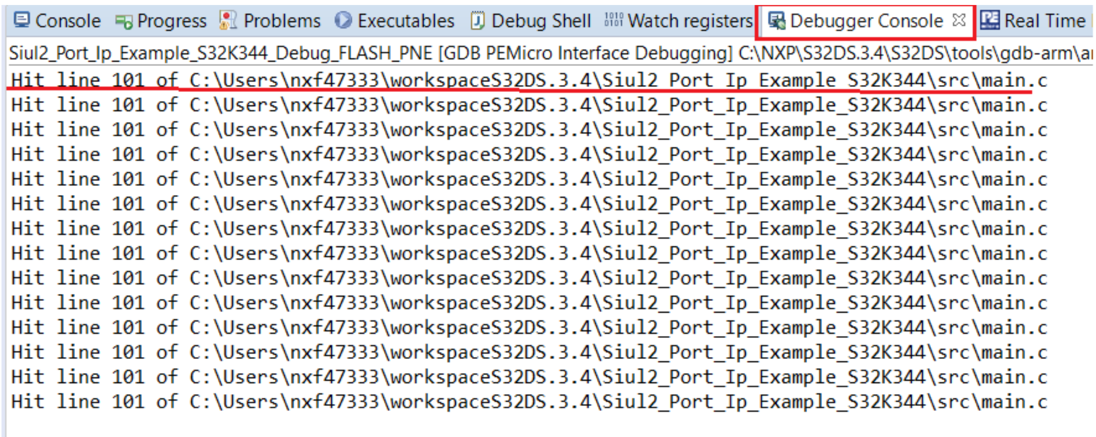
可以看到 main 调用了 BSP_init, BSP_init 里调用了 FlexCAN_Ip_Init_Privileged.在函数上双击可以跳到该函数被调用的代码段。

10. 巧用断点的类型

默认设置的断点，当代码运行到断点处 MCU 就会被挂起，从而停在断点处。但在某些情况下，如调试 FCCU 时，如果设置断点，MCU 停下后将会导致 FCCU 配置 WDG 超时。或在调试类似电机控制类的应用时，不适当的断点会导致三相波某相高边和底边同时打开导致短路。这类情况我们可以改变断点的类型，当代码运行到断点处，MCU 不会 Halt，可以通过 Debugger Console 查看代码是否有运行到断点。



通过 Run->Breakpoint Types，选择断点类型为 Dynamic_Printf。在 line 101 设置断点。全速运行，可以看到 MCU 不会 STOP，通过 Debugger Console 可以看到代码有运行到 Line 101



11. 条件断点

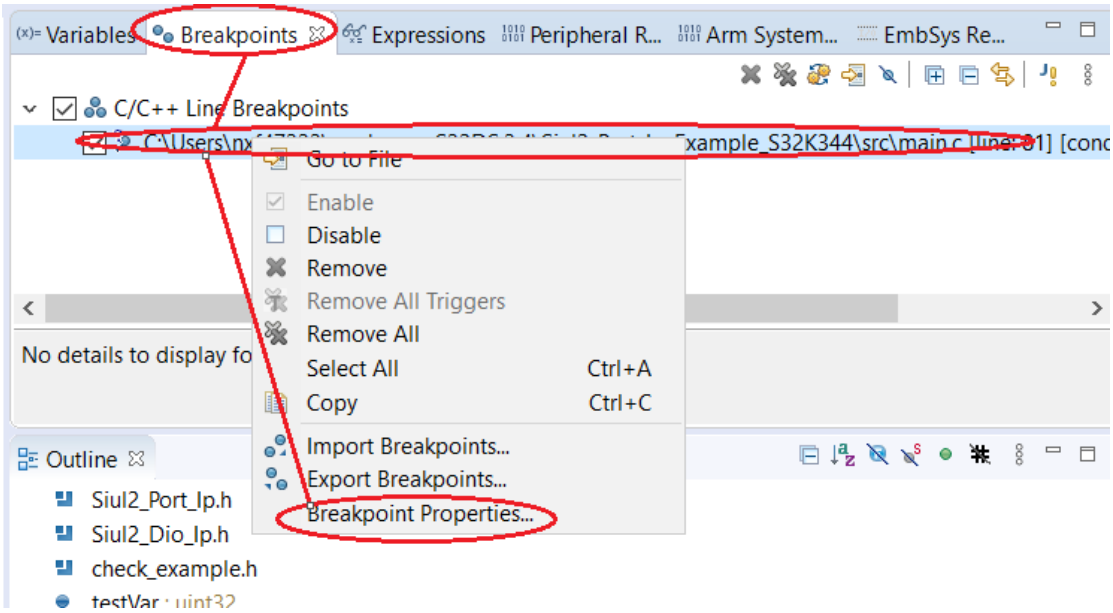
条件断点是一种可以设置条件属性，满足一定条件才会触发的断点。条件可以是某个变量被设置为某个值，也可以是外设某个状态位置位，也可以是它们的形成的逻辑组合

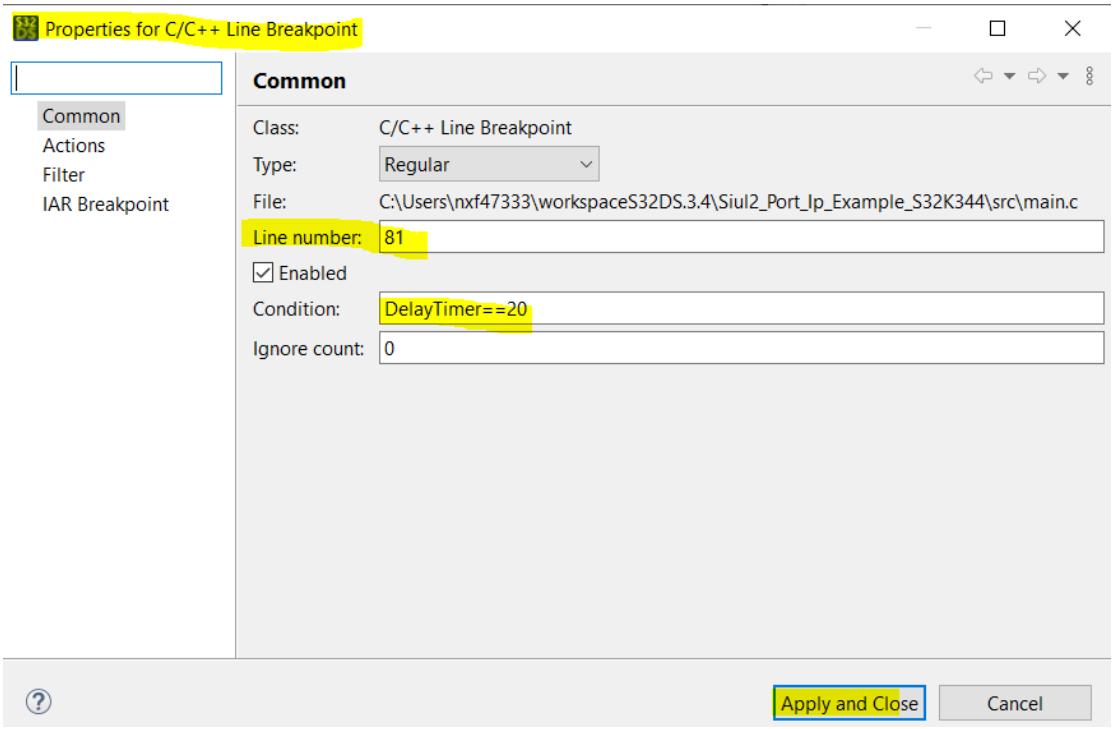
```
76 void TestDelay(uint32 delay)
77 {
78     static volatile uint32 DelayTimer = 0;
79     while(DelayTimer<delay)
80     {
81         DelayTimer++;
82         if (DelayTimer == 1000000)
83         {
84             testVar = 1234; /* init */
85         }
86
87     }
88     DelayTimer=0;
89
90 }
```

如图中代码片段，在 81 行设置断点，默认每循环一次，MCU 会在断点处停一次。如果我们想让 DelayTimer 加到 20 再触发断点。可以这样做。

在 Line 81 放置断点。

打开 Breakpoint properties，配置对应断点的 Condition





全速运行，可以看到当 DelayTimer 的值为 20 的时候，MCU 停在了断点处，再次全速运行，MCU 将不会停下来。因为条件不再满足

12. 用 Watchpoint 捕捉异常的 Memory 访问

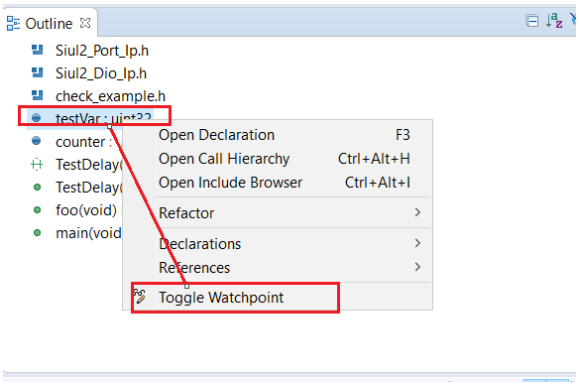
调试 MCU 软件时，经常会遇到这类问题，某个全局变量被意外的修改了，由于软件庞大，调用关系复杂，变量多处访问，在调试这种问题是，通常做法是不断的放断点，查看变量的值。这种调试方法不是特别的高效。我们可以通过另外的方式快速定位变量异常赋值的地方。这种方式就是用 Watchpoint。Watchpoint 也被叫做数据断点。可以通过 Watchpoint 捕获变量被读，或被写的位置。还可以更进一步通过设置条件，在变量被改为某个具体值的时候暂停代码运行。

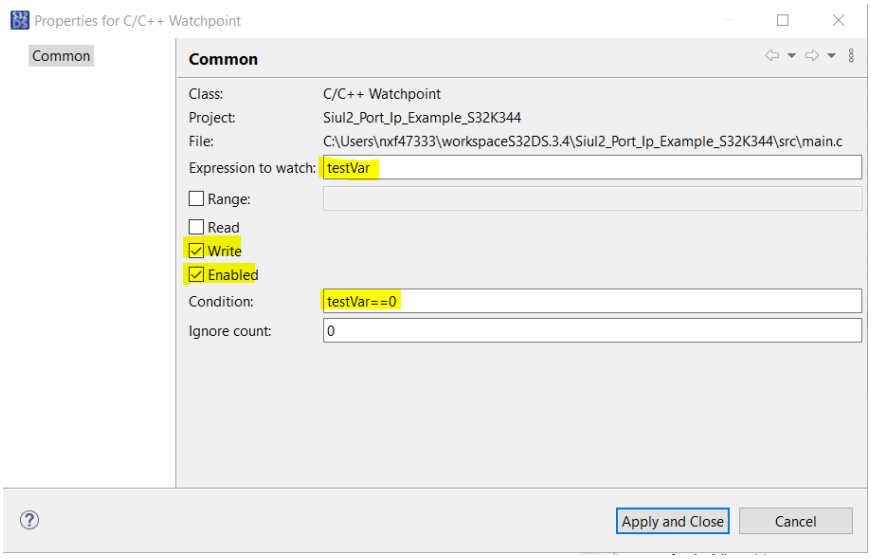
```
93 void foo(void)
94 {
95
96     for(counter=0; counter<5000000; counter++)
97     {
98         if (counter==77832)
99         { /* emulating something else corrupting my testVar */
100            testVar = 0; /* outsch! */
101            Exit_Example(TRUE);
102        }
103        __asm("nop"); /* just doing something */
104    }
105 }
106 /**
```

如上代码片段，捕获变量 testVar 被赋值为 0 的地方

选中变量 testVar，Run->Toggle Watchpoint，打开 Watchpoint 属性配置窗口

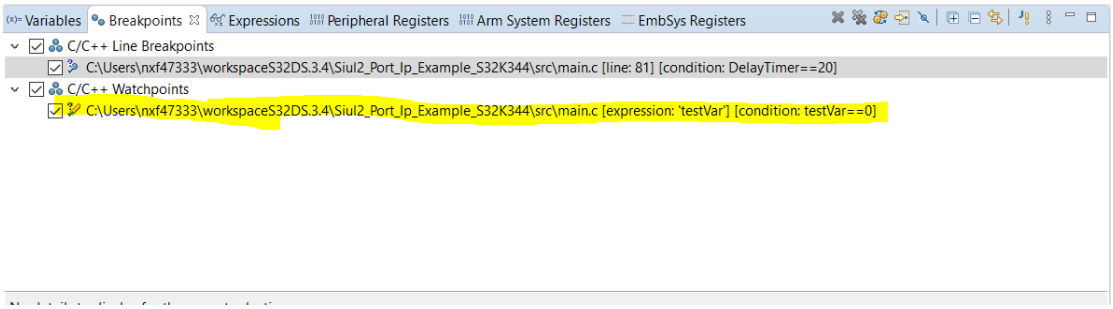
也可以在 Outline，选中变量，然后鼠标右键，在弹出的菜单里选择 Toggle Watchpoint.



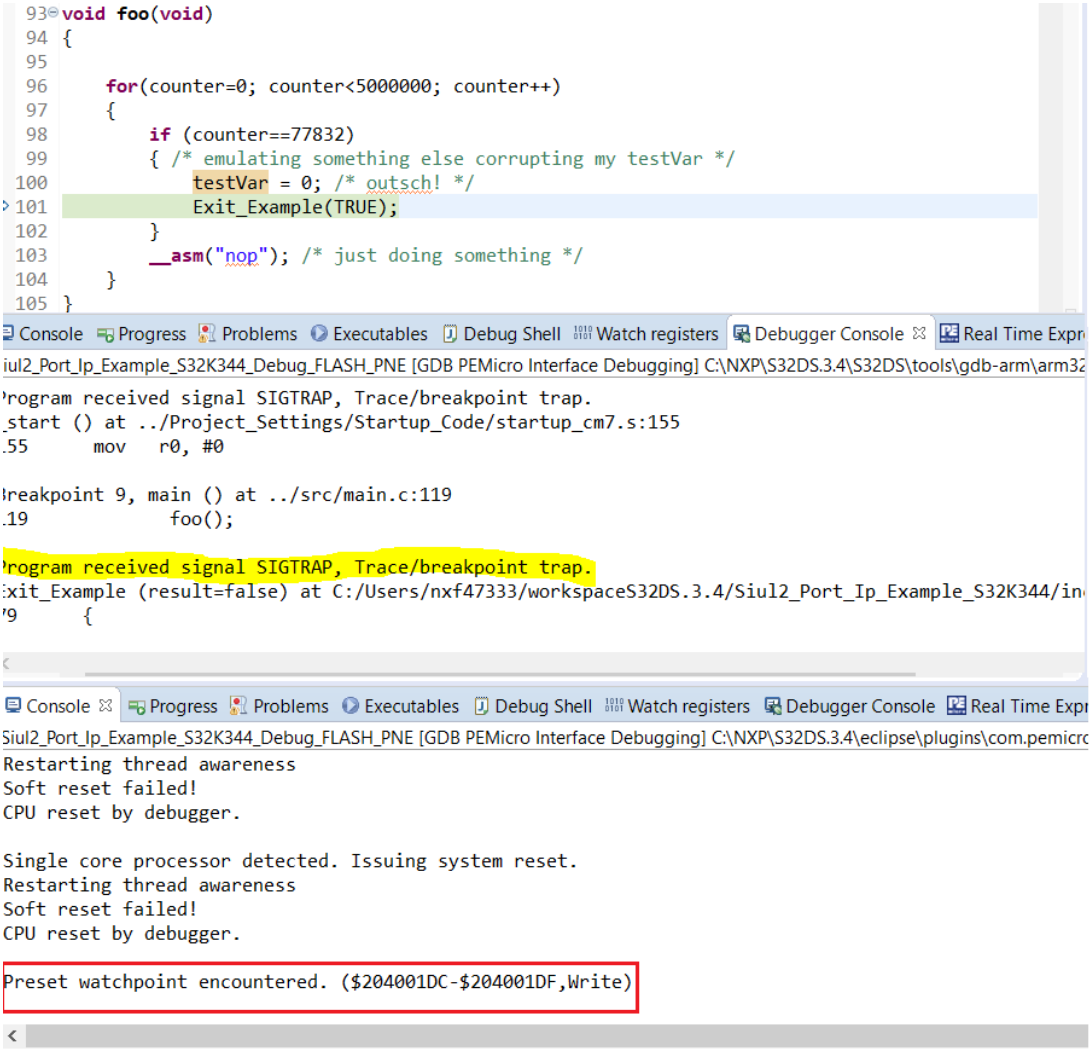


这里我们配置捕获变量 testVar 的写访问，且捕捉 testVar 被置为 0 的时候。

配置完后，可以在 Breakpoints 看到该 Watchpoint



全速运行，会看到代码停在了 testVar 的下一条语句。同时在 Console 可以看到 Watchpoint 被捕获到。Debugger Console 可以看到调试器抛了一个 SIGTRAP 信号，停止了代码的运行。

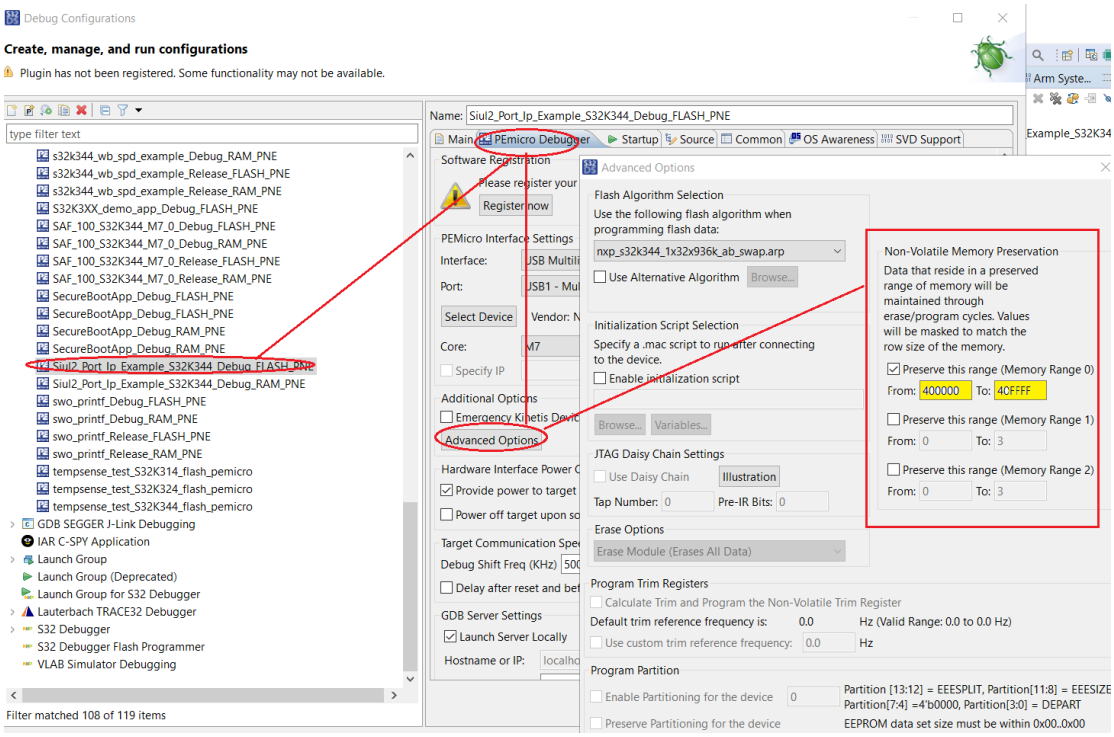


13. 调试 APP 时，保留 Boot 不被擦除

在汽车电子 ECU 中，Bootloader 和 App 会同时驻留在 MCU Flash 上，Bootloader 的存在使得应用的更新变得更加便利。通常 Boot 的开发会早于 App，这样在调试以经烧录了 Boot 的应用时，需要确保在烧录 App 时 Boot 不能被擦除

通过此种方法最多可以保护 3 段 NVM 的地址空间。为了保证擦除功能正常工作，设置的起始地址必须与 Flash 的擦除的最小单元地址对齐，如 S32K3 的擦除 Sector Size 为 8KB，则地址需要和 8KB 对齐。

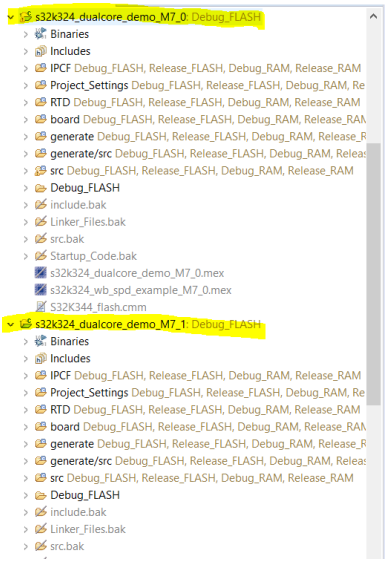
如下图，保留 S32K3 P-Flash 前 64K 不被擦除。

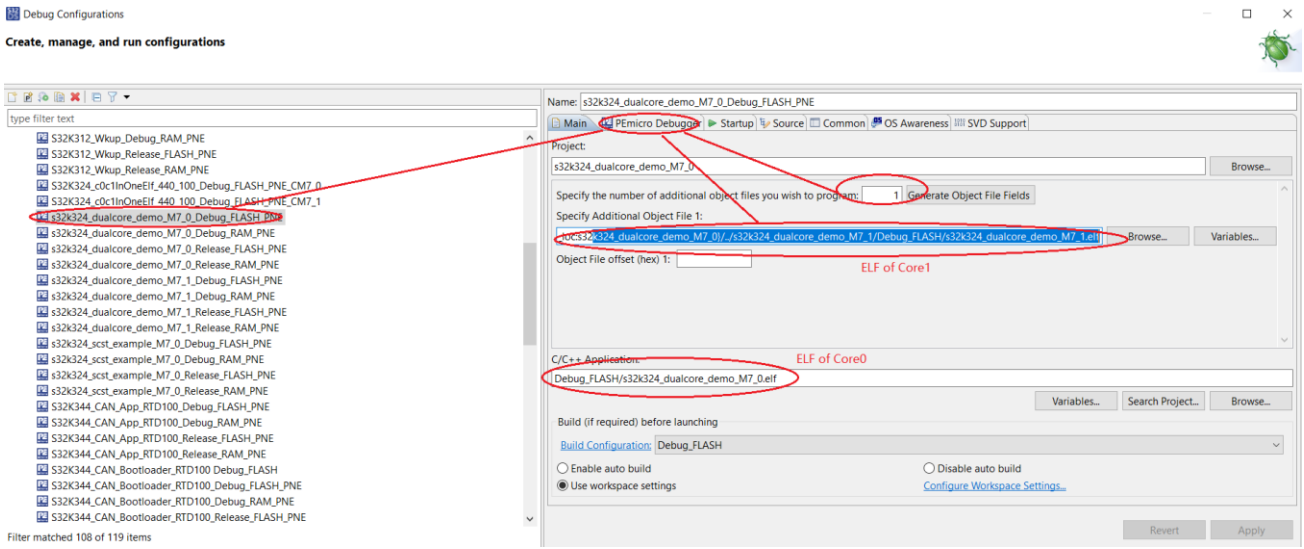


14. 同时烧录多个 ELF,S19,HEX 文件到 MCU Flash

在使用多核 MCU，如 S32K324，S32DS 会为每个核单独创建一个工程，编译生成独立的 ELF, S19 或 HEX 文件。在烧录多个如 ELF 文件，可以手动将多个文件

合并成一个再烧录，或通过一个 S32DS 工程同时烧录多个 ELF 文件。





15. 用一个 Debug 会话，调试 Boot 和 APP

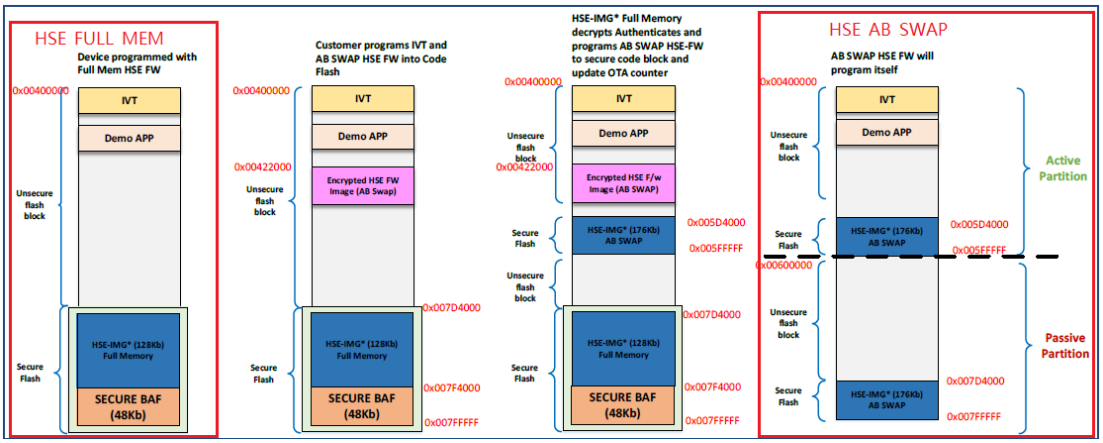
在 Boot 和 App 联调时，我们可以用 Boot 工程将 Boot 和 App 的 ELF 文件同时烧录到 MCU Flash，(操作方式参考 14. 同时烧录多个 ELF,S19,HEX 文件到 MCU Flash)。烧录完成，调试器默认停在 Boot 的 Main 入口处，这时我们可以先调试 Boot，Boot 调试过后，在 Boot 跳转到 App 后，可以将 App 的工程 Attach 到 MCU 上（操作方式参考 1. 把仿真器 Attach 到运行的 MCU 上）然后继续调

试 App.

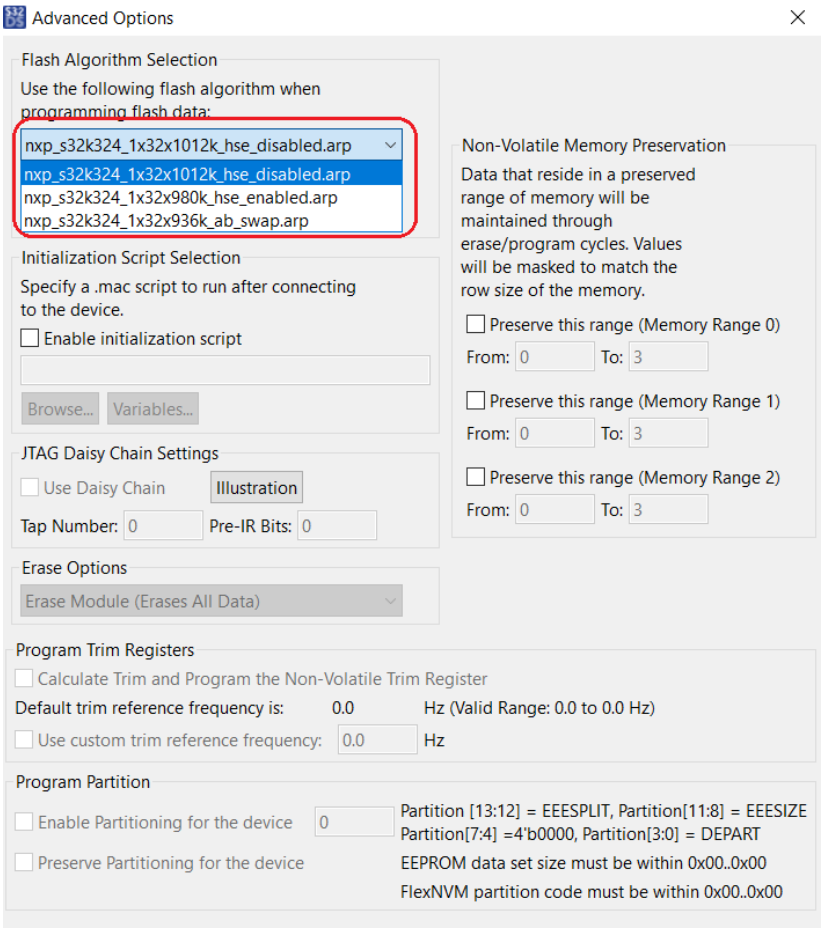
16. 调试已安装 HSE_B FW 的代码

S32K3 内嵌 HSE_B 硬件安全加密模块，安全加密等级支持到 ISO21434。提供给用户的加密服务需要配合运行在 HSE_B 上的 HSE FW 实现。芯片出厂默认是没有装 HSE FW 的，FW 的安装需要客户自己完成。NXP 提供两种类型的 HSE_FW，分别是 HSE_FULL_MEM 和 HSE_AB_SWAP。

前者是不支持 OTA 的，而后者是支持硬件 OTA 的



安装了 HSE_FW 后，Secure Flash 对应的区域会被 Lock 住，编程器是不能对这些区域进行檫除，编程操作的。S32DS PE 提供了三种编程算法，分别针对没有安装 HSE_FW，安装了 HSE_FULL_MEM FW,及装了 HSE_AB_SWP FW 的情况。



在调试软件时，要跟具 FW 的安装情况选择对应的.arp.没有安装 HSE FW 要选择 hse_disabled.arp，安装了 full_mem 的固件要选择 hse_enabled.arp，安装了 ab_swap 的固件要选择 ab_swap.arp.编程算法选择不正确，会报 flash 檫除失败错误。

17. 使用 DWT 测量代码执行时间

通常想要评估某个函数或代码段的执行时间，都是通过 IO Toggle 来做的，这种做法首先需要在要测量的函数或代码段前后加入 IO 操作，代码修改繁琐，同时还需要示波器，逻辑分析仪等工具抓取 IO Toggle 的时间来确定代码的时间开销。对于没有空闲 IO 或没有测试设备在手的话，测试将无法进行。这里给大家介绍下如何在调试模式下使用 DWT 模块评估代码的运行时间开销。

DWT(Data Watchpoint And Trace Unit)，是 ARM Cortex M 系列内核(ARM Cortex-M V7(K1 M4F, K3, M7))提供的数据监测点和跟踪单元，以支持数据断点功能。关于 DWT，这里不做更多的介绍，主要说下如何使用 DWT 的周期计数器 CYCCNT 来测量执行某个任务所花的周期数。CYCCNT 是一个 32BIT 的 UP 计数器，记录内核时钟的运行个数，内核时钟跳动一次，改计数器就加 1，对 K3X4 内核时钟最高 160M，每个时钟节拍周期是 6.25ns，最长能记录 26.84S。要实现时间测量功能，总共涉及到三个寄存器，DEMCR, DWT_CTRL, DWT_CYCCNT，分别用于开启 DWT 功能，开启 CYCCNT 及获得系统时钟计数值。关于这三个寄存器更详细的说明，请参考**"Armv7-M Architecture Reference Manual"**

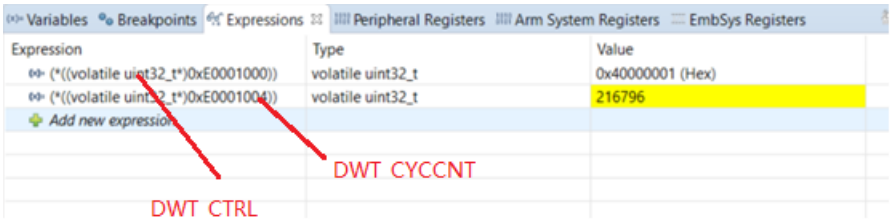
如下是如何在调试模式下编程这三个寄存器使能 CYCCNT 及读取 CYNCNT 操作的。

(1).在 S32K344.mac 文件(该文件是 PE Micro 调试工具下的一个脚本文件，路径:

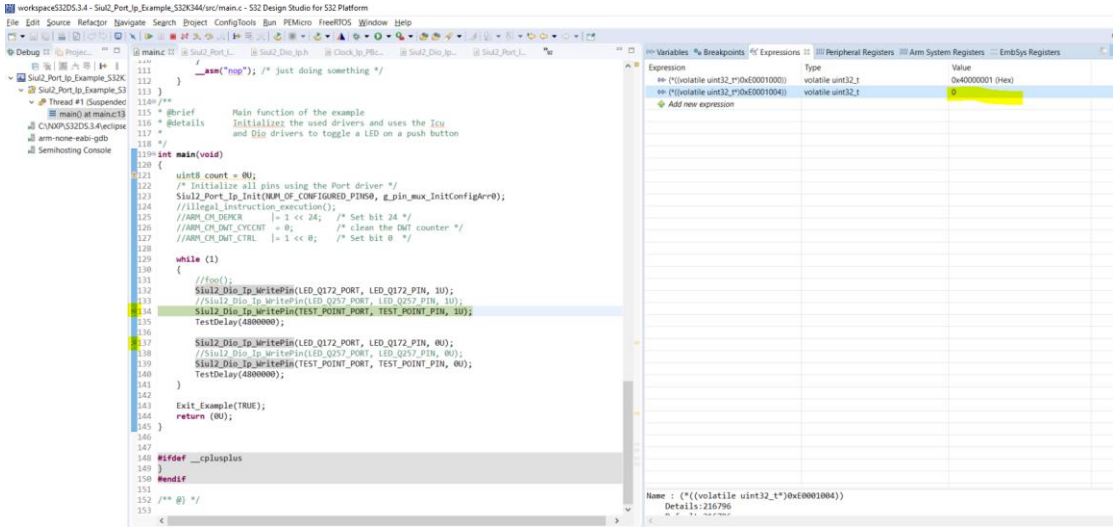
C:\NXP\S32DS.3.4\eclipse\plugins\com.pemicro.debug.gdbjtag.pne_5.1.7.202112141853\win32\gdi\P&E\supportFiles_ARM\NXP\S32K3xx\S32K344.mac，在该脚本里可以执行一些寄存器的初始化，该脚本在 CPU Core 运行起来之前由调试器加载，解释，并执行)开启 DWT,和 CYCCNT 并清空 CYCNT。

```
52  REM Enable DWT CYCCNT:
53  mm.l $E000EDFC $010007F0
54  mm.l $E0001000 $40000001
55  mm.l $E0001004 $00000000
56  delay 20
```

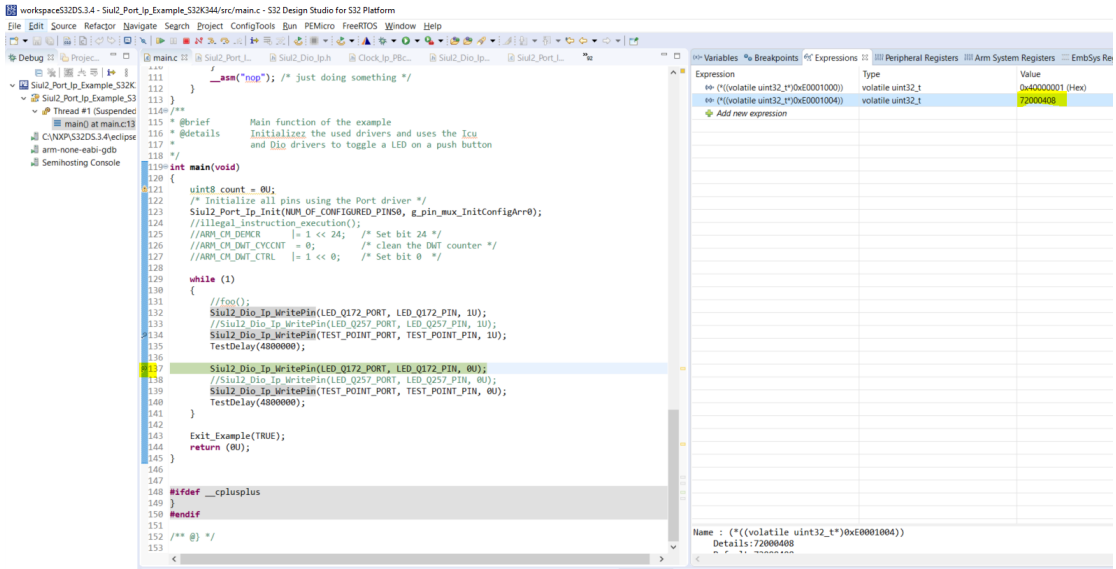
(2)在 Expressions 窗口添加 DWT_CTRL 和 DWT_CYCCNT 寄存器，



(3)在要测试的函数或代码段前,和后设置断点，并清零 CYCCNT



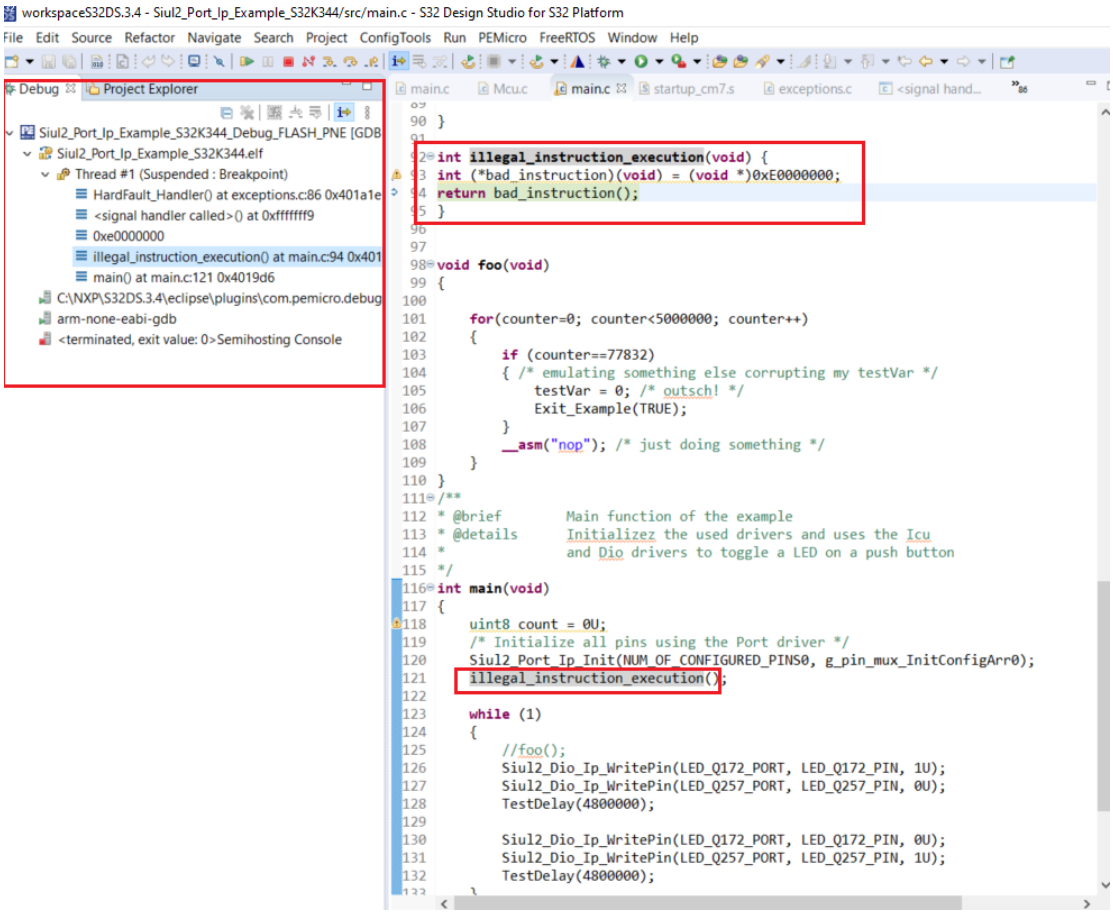
(4)RUN，代码停在要测试的代码段后，此时可以看到 CYCCNT 更新为 72000408。此时的 CYCCNT 就是函数“TestDelay(4800000)”所消耗的 CPU 系统时钟个数。，该测试代码没有配置系统时钟，默认系统时钟为 FIRC 48M，换算成对应的时间为 144 ms



18. 调试 Cortex-M MCU HardFault

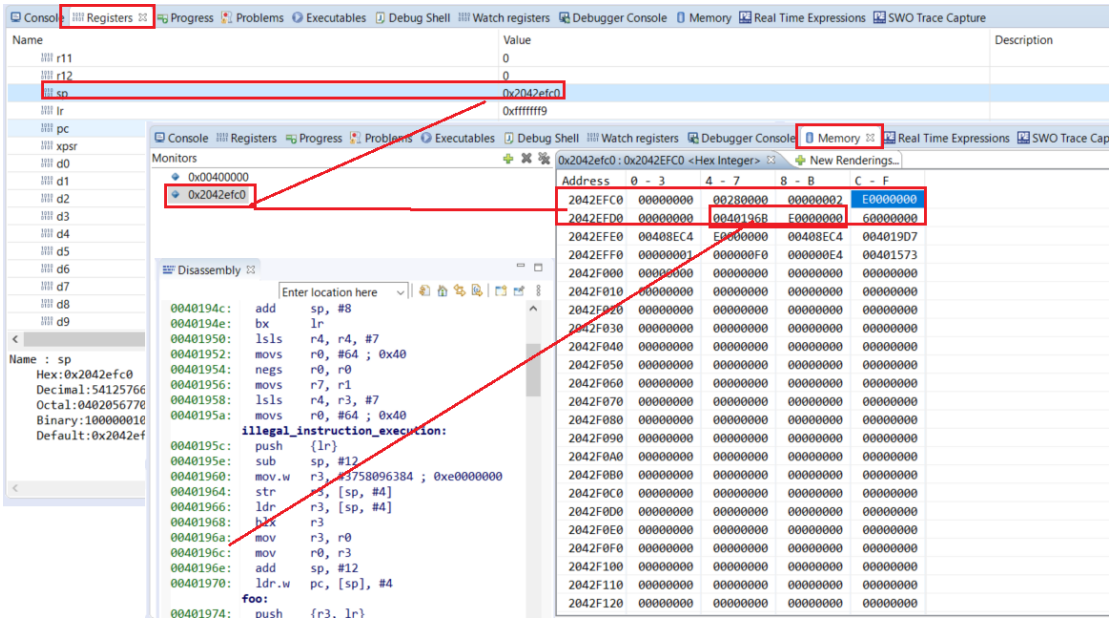
HardFault 的调试是 MCU 软件调试中最让工程师头疼的，究其原因是故障点很难定位。那如何快速，高效的定位故障，及故障所在的代码行，这里给大家介绍一套组合拳。

(1).通过函数调用栈，定位发生 HardFault 的函数。函数调用栈的查看见**(9. 查看函数调用栈，分析函数调用过程)**



如上图，代码模拟了非法地址访问导致的 HardFault。我们可以通过函数调用栈看到代码运行到函数“illegal_instruction_execution()”的 94 行，产生了 HardFault。由此我们可以大致定位故障点在这个地方。该方法对于代码逻辑比较简单的情况比较奏效，但对于代码和逻辑操作比较复杂的情况这种方法很难定位到具体的故障点。

(2).通过异常时的通用寄存器值准确定位故障点。Cortex-M MCU 在接收到一个异常后，芯片硬件会自动将 8 个通用寄存器(依次为 R0~R3, R12, LR, PC, xPSR)压入到当前栈空间。Cortex-M MCU 有两个栈 MSP 主栈和 PSP 线程栈。异常发生后 LR 会被更新为异常返回时需要用到的特殊值 EXC_RETURN，其中 BIT2 指示了当前用到的栈是 MSP 还是 PSP。了解到了异常时 MCU 寄存器保存情况，我们就很容易找到发生异常时的指令地址即 PC。具体操作见：



(注：S32DS 的 Registers 窗口只能看到 MSP，看不到 PSP)，如图根据 SP: 0x2042efc0,可以看到异常产生后保存在 MSP 栈上的 8 个通用寄存器的值，PC：0xe000000，异常时的 PC，该地址是一个非法的地址。LR: 0x0040196b，异常返回地址，根据这些信息可以看到，代码执行到 0x0040196b 后执行了非法地址 0xe000000 访问导致了 HardFault.

如上一系列的数据回溯还是挺繁琐的。如下介绍一种自动化工具.gdbinit .gdbinit 是一个脚本，gdb 启动时会从指定路径搜索该文件，如果找到文件，则执行其中的命令。也可以在该脚本里自定义命令，自定义的命

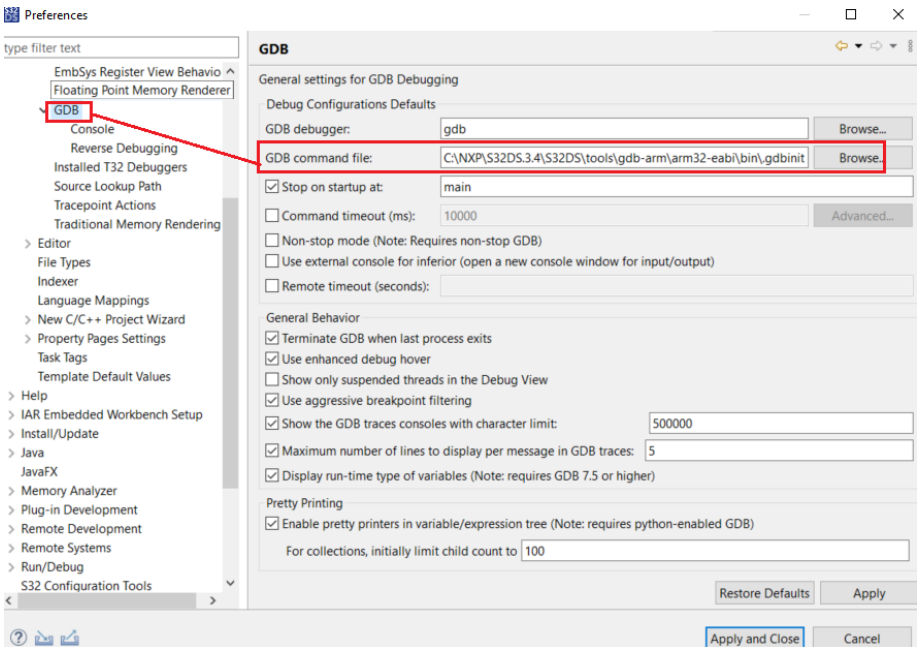
令会被 gdb 所识别，之后可以在 debugger console 中执行自定义的命令。

如下.gdbinit，定义 armex 命令用于如上栈数据的回溯。

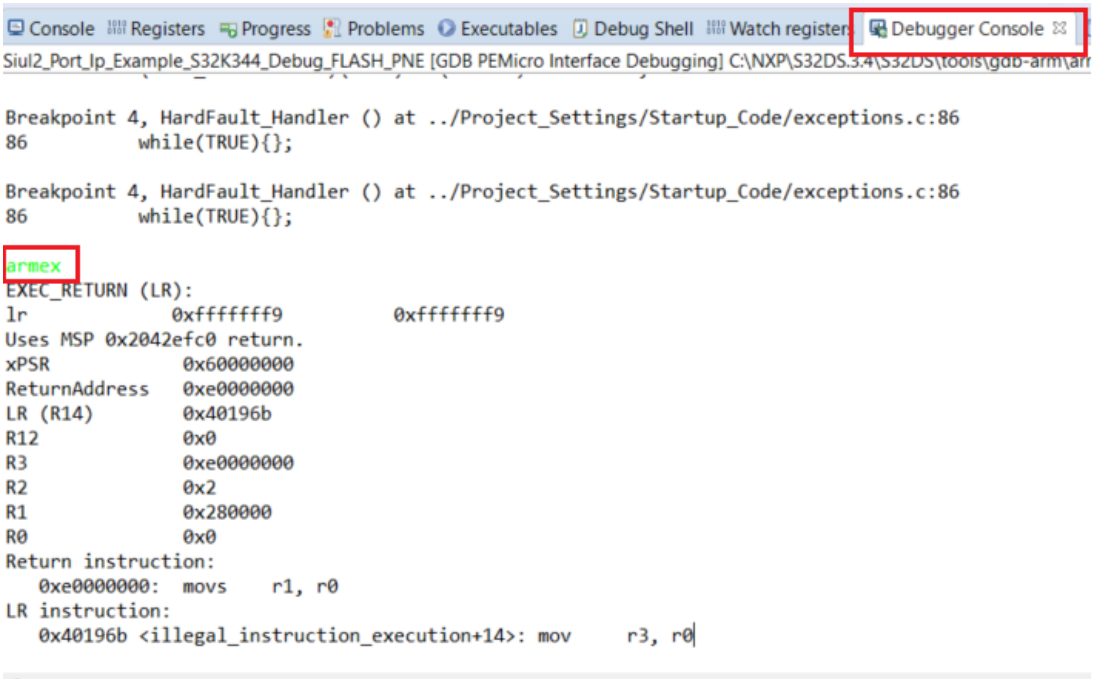
```
set language c
define armex
  printf "EXEC_RETURN (LR):\n",
  info registers $lr
    if (((unsigned int)$lr & 0x04) == 0x4)
      printf "Uses PSP 0x%x return.\n", $psp
      set $armex_base = (unsigned int)$psp
    else
      printf "Uses MSP 0x%x return.\n", $msp
      set $armex_base = (unsigned int)$msp
    end

    printf "xPSR            0x%x\n", *($armex_base+28)
    printf "ReturnAddress   0x%x\n", *($armex_base+24)
    printf "LR (R14)        0x%x\n", *($armex_base+20)
    printf "R12             0x%x\n", *($armex_base+16)
    printf "R3              0x%x\n", *($armex_base+12)
    printf "R2              0x%x\n", *($armex_base+8)
    printf "R1              0x%x\n", *($armex_base+4)
    printf "R0              0x%x\n", *($armex_base)
    printf "Return instruction:\n"
    x/i *($armex_base+24)
    printf "LR instruction:\n"
    x/i *($armex_base+20)
end

document armex
ARMv7 Exception entry behavior.
xPSR, ReturnAddress, LR (R14), R12, R3, R2, R1, and R0
end
```

关于.gdbinit 及其编程操作这里不做过多介绍。

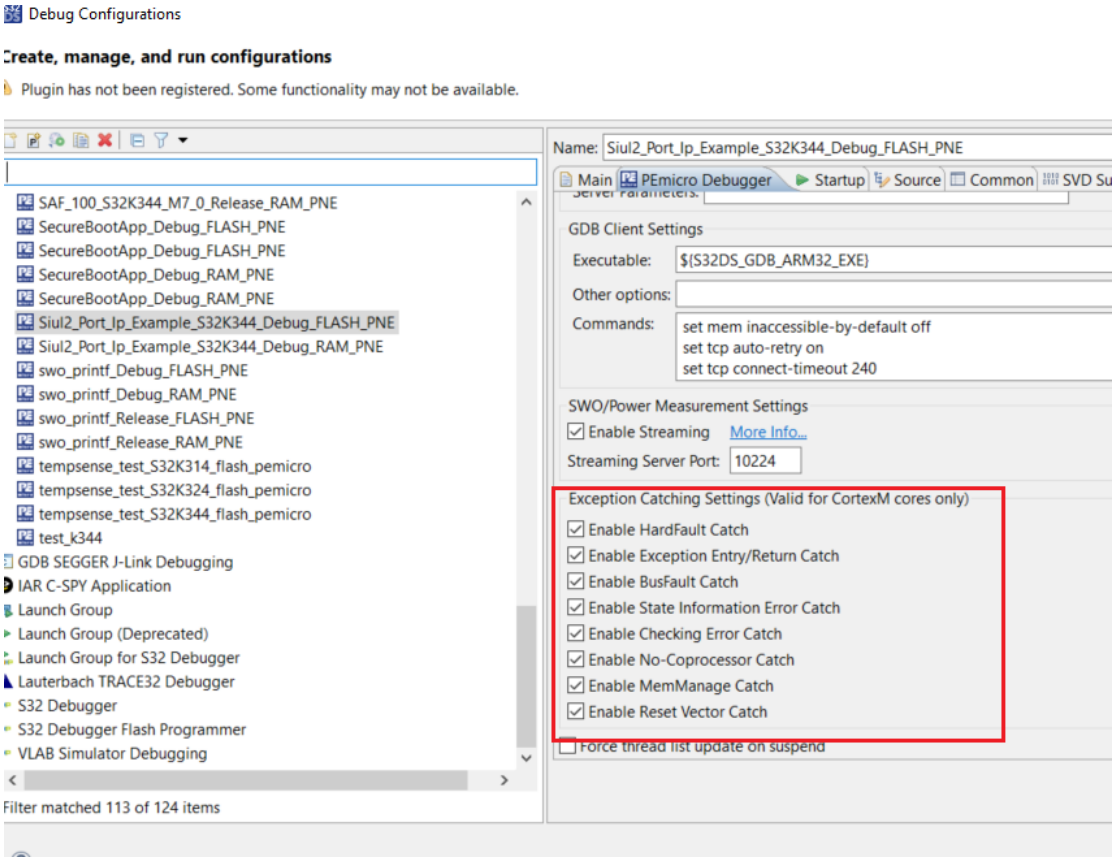
在 Window->Preferences->GDB 配置.gdbinit 脚本路径



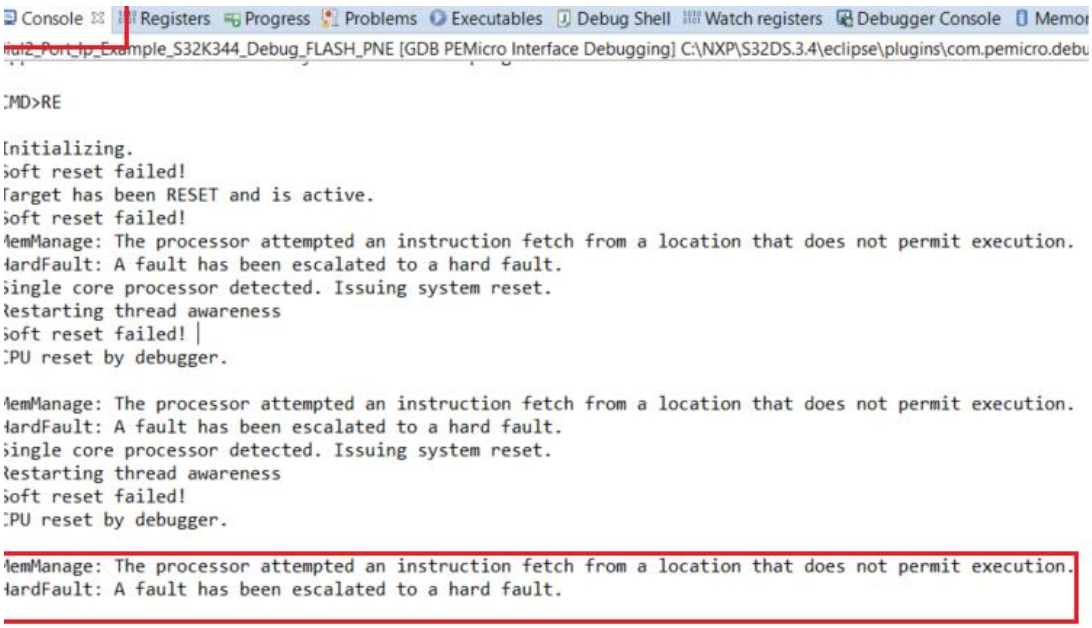
HardFault 产生后，在 Debugger Console 窗口输入自定义的 armex 命令，可以看到之前手工分析的栈数据回溯被自动的解析出来。根据这些信息可以看到，代码执行到 0x0040196b 后执行了非法地址 0xe000000 访问导致了 HardFault.



(3)使能异常捕获功能，捕获 HardFault.

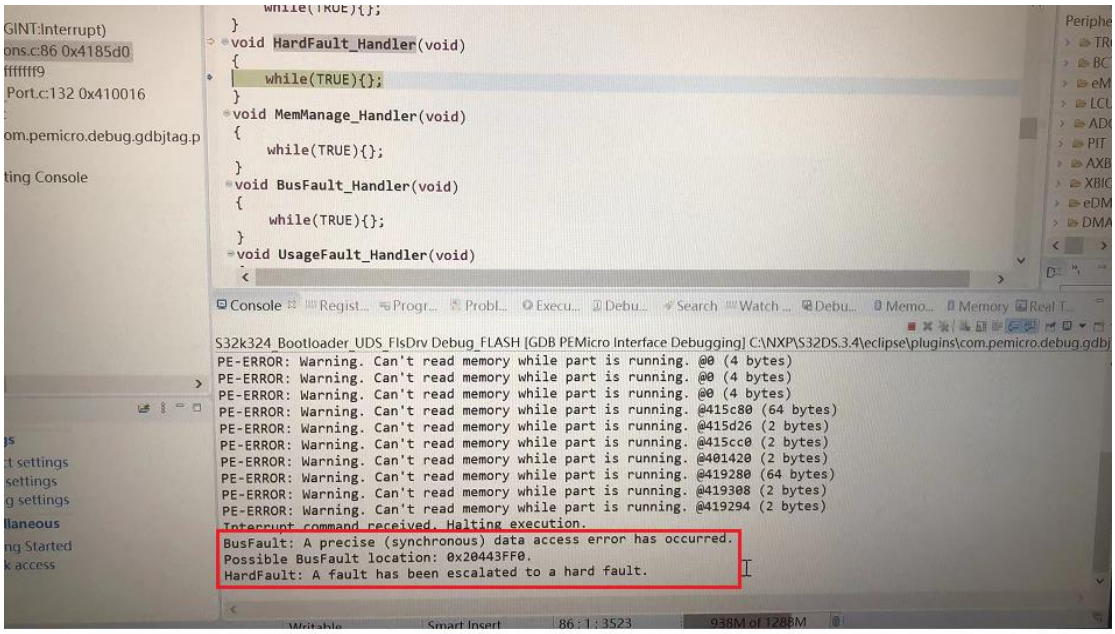


如上图，使能异常捕获，当 CPU 进入异常后，调试器会捕捉到异常进入，进而挂起 CPU，同时 Console 会报告异常产生



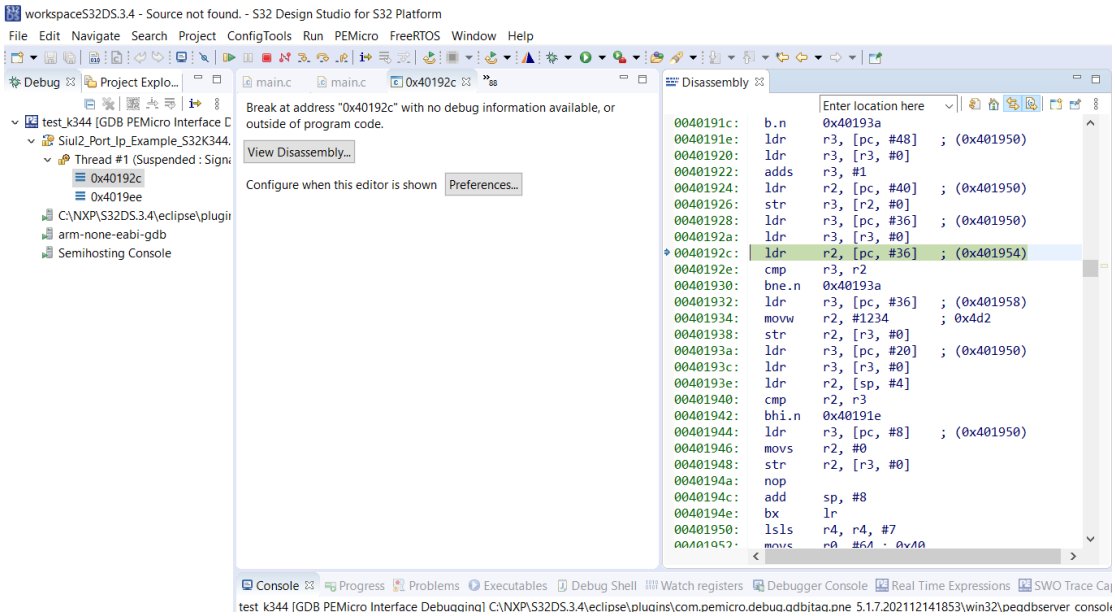
如图可以看到，指令非法地址访问触发了 MemManage 异常，但由于并没有使能 MemManage 异常及定义其异常处理，该异常升级到了 HardFault.

如果是 BusFault 异常，且异常 precise 访问，Console 窗口还会给出导致 BusFault 的数据地址。



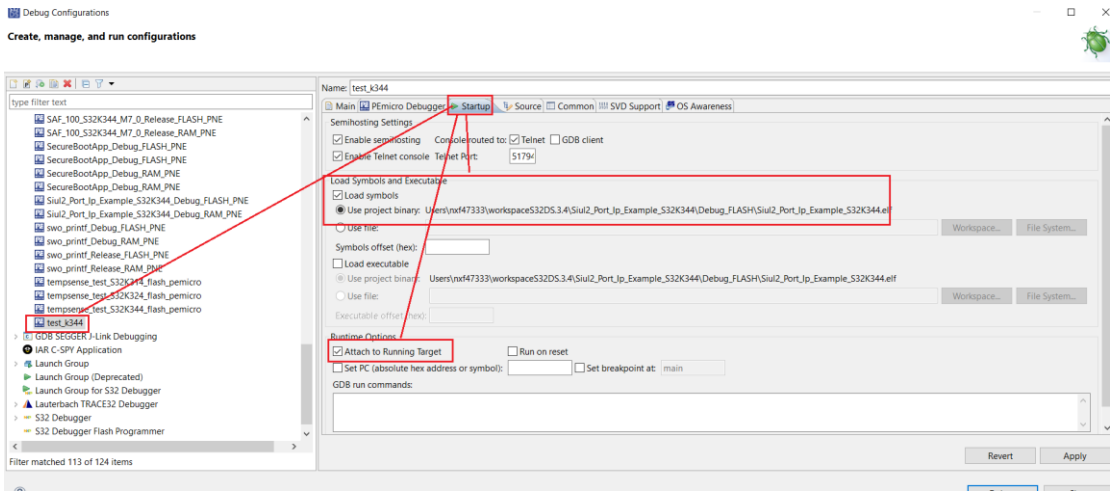
19. C 源码关联

通常启动调试器，调试器会将可执行文件烧录到 MCU 的 Flash,并将调试信息加载到调试器，这样我们就可以直接以 C 源代码的方式进行调试。但有时烧录到 MCU 的 ELF 文件和生成该 ELF 的该工程不对应，或是生成 ELF 的工程源文件结构发生了很大变化，导致调试信息没法关联到 C 源文件，这样的话，需要我们手动进行源码关联

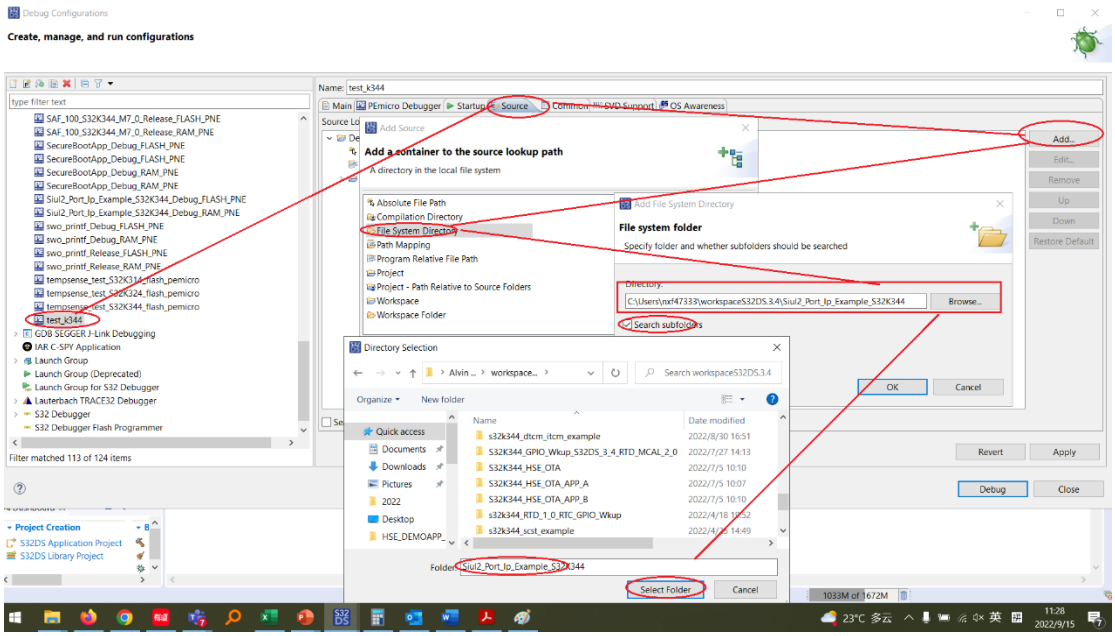


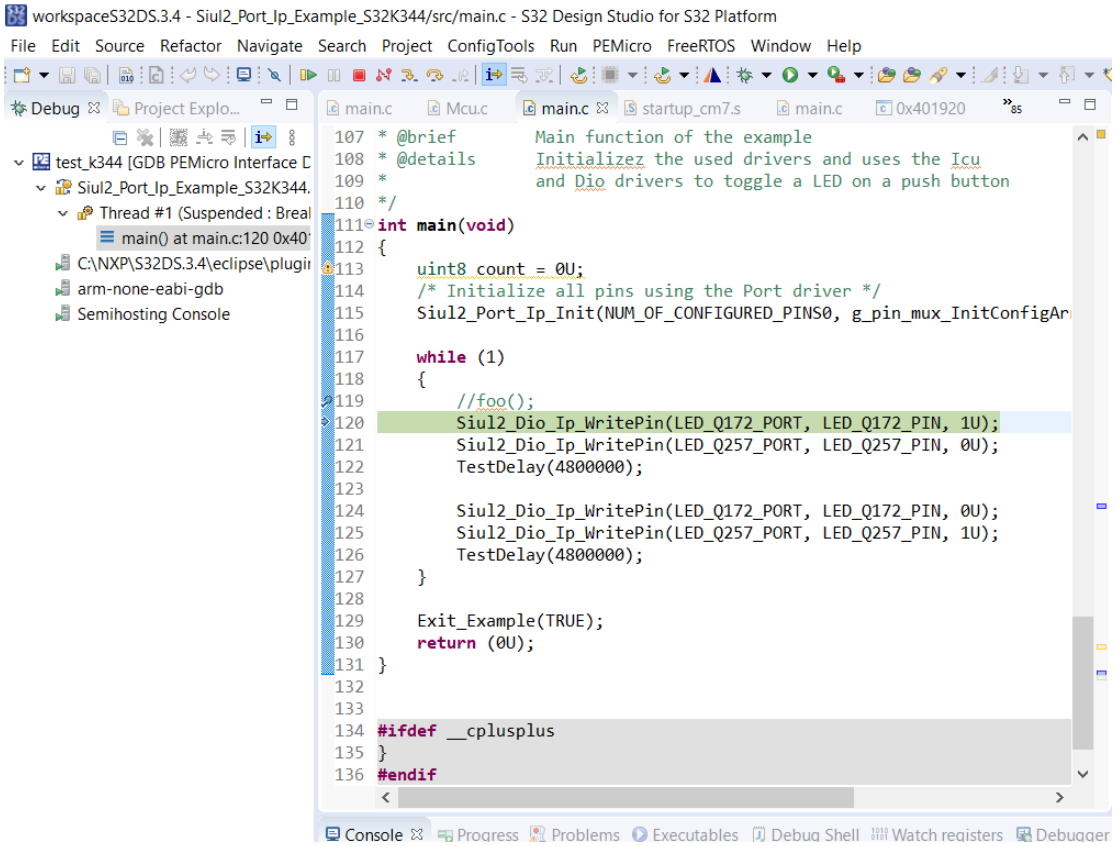
如上图调试，只能看到汇编代码，无法看到 C 代码。可根据如下操作将 C 代码和调试信息关联起来.

关闭当前的调试会话，重新启动一个新的调试会话，以 Attach 的方式将仿真器挂到 MCU 上(具体操作见 1. 把仿真器 Attach 到运行的 MCU 上)，将调试信息下载到调试器



关联 C 代码，注意：在浏览 C 代码路径时，一定要选择工程的根目录，并勾选 Search subfolders，这样才能保证源代码和调试信息一一对应





重新启动调试器，就可以看到 C 代码了。

20. Solving “Launching: Configuring GDB Aborting Configuring GDB”

当调试器报出“Launching: Configuring GDB Aborting Configuring GDB”信息，调试器一直卡在启动状态时。检查如下配置