调试

通过探讨一些 C-SPY 调试器的特性，本章展示它们的性能以及如何使用它们:

- 准备调试
- 启动调试器
- 执行你的应用
- 查看变量
- 监视内存和寄存器
- 使用断点
- 查看终端 I/O
- 分析应用的运行时行为

注意,根据你安装的产品软件包，可能含也可能不含C-SPY 。

根据你的硬件,这里没探讨的附加特性或许会在你使用的 C-SPY 驱动里提供。 一般来说，它用来设置不同类型的观察点，附加的断点类型，各种触发系统，更复杂的追踪系统等等。

准备调试

1. 开始C-SPY之前，选择 **Project>Options**>**Debugger>Setup** 并选择满足你的调试系统的 C-SPY 驱动: 仿真或硬件调试系统。
2. 在 **Category** 列表中，选择合适的 C-SPY 驱动并审查你的设置。
3. 当你做好了 C-SPY 设置，单击 **OK**。
4. 选择 **Tools>Options>Debugger** 进行配置:
   - 调试器的行为
   - 调试器对堆栈使用的追踪。

在 C-SPY 开始之前设定硬件

你可以在 C-SPY 开始之前用 C-SPY 宏来初始化目标硬件。例如，如果你的硬件使用外部存储器,它在代码可以下载给它之前必须启用，C-SPY 需要宏，在你想调试的应用程序可以被下载之前,完成这一动作。像下面这样:

1 创建新文本文件并定义你的宏函数。比如，启用外部SDRAM的宏可能看起来像这样:

```
/* 你的宏函数。 */
enableExternalSDRAM()
{
  __message "Enabling external SDRAM\n";
  __writeMemory32( /* 在这里放你的代码。 */ );
  /* 如果有必要这里还可以放。 */
}

/* 设置确定执行时间的宏。 */
execUserPreload()
{
  enableExternalSDRAM();
}
```

因为使用了内置的 `execUserPreload` 设置宏函数，你的宏函数,在与目标系统的通信建立之后并在 C-SPY 下载你的应用程序之前,会被直接执行。

2 用文件扩展名mac保存文件。

3 启动 C-SPY 之前，选择 **Project>Options>Debugger** 并单击**Setup** 标签。 选择**Use Setup file** 选项并选择你刚创建的宏文件。 现在你的启动宏会在 C-SPY 启动序列执行过程中装载。

启动调试器

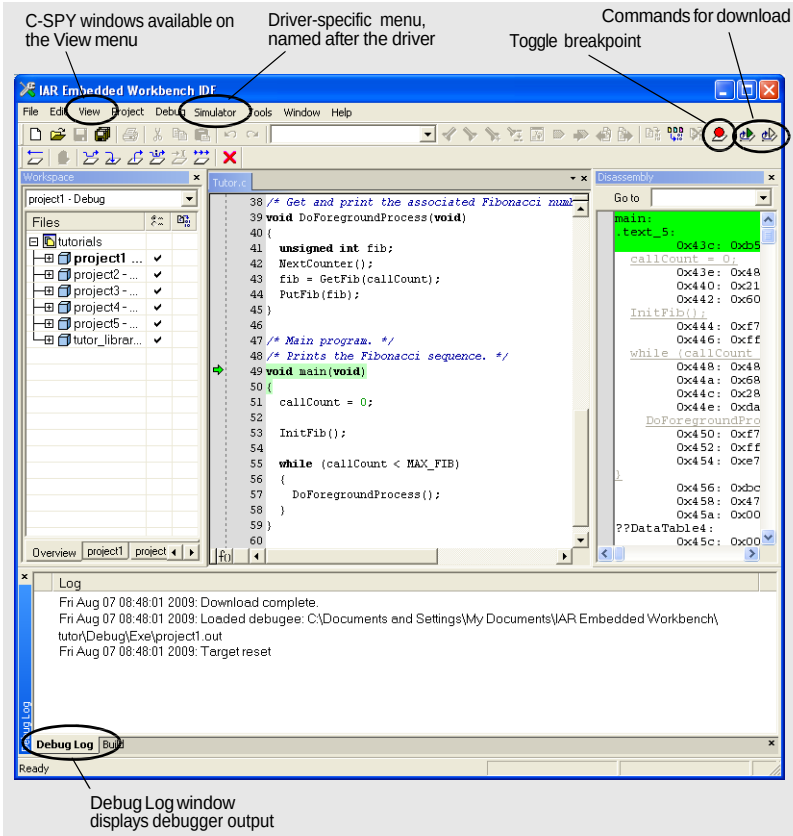
你可以有两种做法启动调试器:

Download and Debug 启动 C-SPY 并加载当前的项目到目标系统。

Debug without Downloading 启动 C-SPY 不用重新加载当前的项目到目标系统。假定代码映像已经在目标上,因此这条命令对仿真器不起作用。

你可以加载多个调试文件（映像）到目标系统。在 IDE 中加载附加的调试文件，选择 **Project>Options>Debugger>Images**。 这意味着整个程序是由几个映像组成的 。 例如，你的应用（一个映像）是由引导加载程序（另一个映像)启动的。应用程序映像和引导加载程序是由不同的项目构建并产生的不相关的输出文件。

C-SPY 伴随应用程序加载后启动。



C-SPY 必须读取目标系统来更新窗口的内容（ 需要更新的窗口，例如内存和追踪窗口)。 这在调试的时候会影响响应时间。 如果你有几个窗口同时打开，而响应时间太长（尤其是应用在硬件上执行)，就关闭一两个窗口来缩短响应时间。

退出C-SPY:

单击 **Debug** 工具条上的 **Stop Debugging** 按钮。

执行你的应用

你可以在 **Debug** 菜单上或 **Debug**工具条上找到执行命令，诸如:

Step Over 执行下一条语句，函数调用，或指令，不进入 C/C++ 函数或汇编语言子程序。

Step Into 执行下一条语句， 或指令，进入 C/C++ 函数或汇编语言子程序。

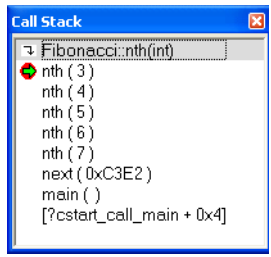
Next Statement 直接执行下一条C/C++ 语句,不用停在单个函调用上。

你还可以在菜单和工具条上找到命令，如 **Go**, **Break**, **Reset**, **Run to Cursor**, **Autostep**, 等。

C-SPY 允许与大部分其它调试器相比单步更精确，因为它不是面向行的而是面向语句的，是因为*单步断点*。能够单步进入单个函数调用，这个函数调用是较复杂的语句的一部分,在你使用包含许多嵌套函数调用的 C 源代码时特别有用。对于 C++ 也很有用，它往往会有很多隐性函数调用，例如构造函数，析构函数，赋值运算符和其他用户定义的运算符。

查看函数调用:

1 选择**View>Call Stack**打开调用堆栈窗口。显示当前函数在顶部的C/C++函数调用堆栈。双击任意函数, IDE中所有受影响的窗口的内容更新，显示该特定的调用帧的状态。

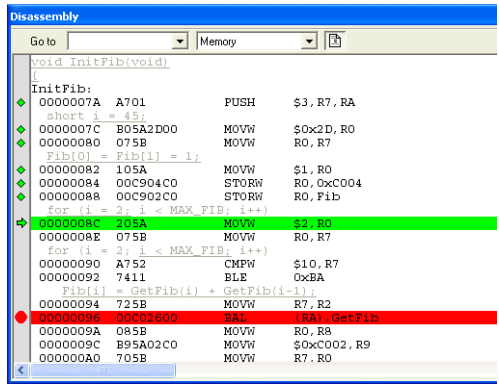


一般来说，这有助于两个目的:

- 确定哪段上下文中调用了当前的函数。
- 追踪变量或参数不正确的值的根源,如此在调用链中定位该函数问题出在哪里。

在反汇编模式中调试:

选择**View>Disassembly**打开反汇编窗口，如果还没打开的话。你将看到与当前C语句对应的汇编语言代码。



反汇编模式使你精确地一次一条汇编指令地执行应用程序。换句话说，C/C++ 模式一次一条语句或一个函数地执行你的应用程序。无论你用什么模式调试，你都可以显示并改变寄存器和内存的内容。

切换模式:

你想使用什么模式,就用鼠标指针激活你的编辑器或者反汇编窗口。

查看代码覆盖范围信息:

右键单击反汇编窗口并选择**Code Coverage>Enable，** 然后在上下文菜单选择**Code Coverage>Show**。绿色方块标注着已经执行过的代码。也可以查看 *代码覆盖* 第56页。

查看变量

C-SPY 允许你在源代码中监视变量或表达式，以便你执行应用程序时持续追踪它们的值。你可以有几种方法查看一个变量:

提示条监视 提供最简单的方式在编辑窗口中查看变量的值或者较复杂的表达式。只要用鼠标指到该变量。值就显示在变量的旁边。

局部窗口，在 **View** 菜单提供，自动显示局部变量，也就是，当前在用的函数的自动变量和函数参数。

监视窗口，在 **View** 菜单提供，允许你监视 C-SPY 表达式和你选择的变量的值。

实时监视窗口，在 **View** 菜单提供，执行应用时重复采样并显示表达式的值。表达式里的变量必须是静态分配的，就像全局变量。 请注意这个窗口要求目标系统支持程序执行期间读取内存。

静态窗口，在 **View** 菜单提供，自动显示变量静态存储期间的值。 另外，你可以自己选择这样的变量来显示。

自动窗口，在 **View** 菜单提供，自动选择显示当前语句或附近的变量和表达式。

快速查看窗口，精准控制何时评估或监视一个变量或表达式值的快速方法。

收集追踪数据，由驱动专用的菜单提供，可以收集一系列目标系统中的事件，通常是已执行的机器指令。根据你的目标系统，可以收集其他类型的追踪数据。例如,对内存的读写访问，以及C-SPY表达式的值。另见 *追踪* 第 59 页。

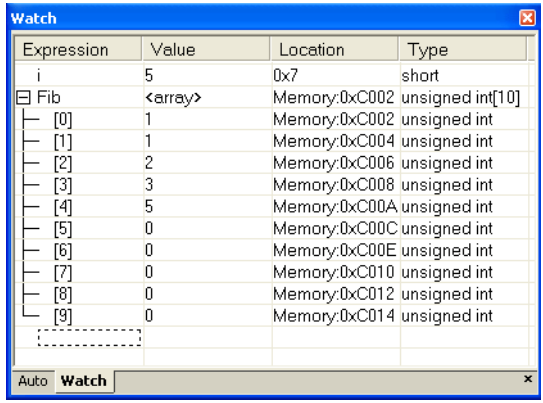
注意: 当优化级别使用了None时，所有非静态变量在其整个作用域一直存活,正因为如此,一直可以调试到该变量。当使用了较高等级的优化时，调试变量可能就不会完全做到。

你可以增加，修改,删除表达式,以及改变显示格式。所有窗口中的操作命令都提供有上下文菜单。支持窗口之间合适的拖拽。

查看变量的值:

1. 例如，选择 **View>Watch** 打开监视窗口。
2. 遵循下面的步骤选择变量:
   - 单击监视窗口里的虚线框。
   - 在出现的输入字段中，输入变量名并按下回车键。
   - 还可以从编辑窗口拖拽一个变量到监视窗口。

此例中，监视窗口显示了当前变量的值i和 Fib数组。 你可以展开 Fib 数组查看它更多细节。



3 要从监视窗口删除一个变量，选它并按下删除键。

监视内存和寄存器

C-SPY 提供了许多窗口来监视内存和寄存器,它们都在**View**菜单中提供:

内存窗口 展示了内存特定区域最新的内容—在C-SPY中称之为*内存区*—而且允许你对它进行编辑。 色彩用于指示数据范围 (根据你的产品软件包) 及你的应用怎样执行。你可以填充特定的具有特殊值的区域并可以直接在内存某个位置或范围设置断点。你可以打开多个这种窗口实例，来监视不同存储区域。

符号存储窗口 显示静态存储的变量在内存中布置存放的情况。这有助于更好地理解内存的用法或者探究变量被覆盖导致的问题，例如缓冲区超出了限制。

堆栈窗口 显示堆栈的内容，包括堆栈变量在内存中如何布置。更多详细资料，查看 *堆栈的用法*，第 58 页。

寄存器窗口 显示处理器的寄存器及 SFRs 的最新内容，并允许你编辑它们。

要查看特定变量在内存中的内容，简单地将该变量拖拽到内存窗口或符号存储窗口，放置变量的内存区域就会显示出来。

使用断点

基于你正在使用的 C-SPY 驱动，可以设置各种断点：

代码断点 用于代码定位以研判你的程序逻辑是否正确或打印输出追踪过程。

日志断点 提供一种增加打印输出踪迹的方便的方法，不必给应用程序的源代码添加任何代码。

追踪的起停断点 收集开始和停止追踪之间的数据—便于分析两个执行点之间的指令。 另见 *追踪*，第 59 页。

数据断点 由内存读写访问触发。一般来说，数据断点用来研判数据何时和怎样发生的变化。

除了这些断点外，C-SPY 驱动还会支持更复杂或其它断点或不同种类的触发器，取决于你正在使用的调试系统。

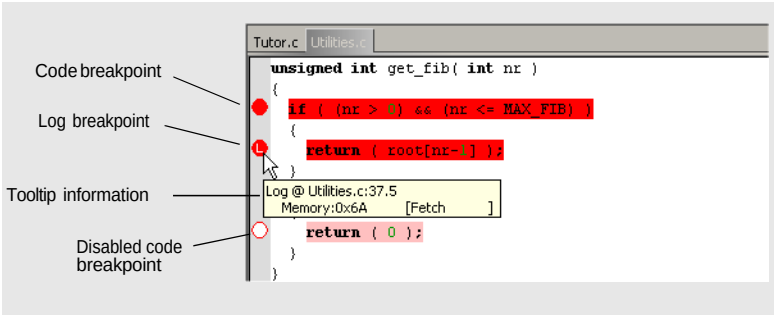
设置断点：

在左边页边空白处定位插入点。在一条语句上或附近双击，切换代码断点的开启和关闭。

或者，使用编辑窗口，断点窗口和反汇编窗口的上下文菜单提供的 **断点** 对话框。 对话框给你更加详细的方法来设置不同类型的断点并编辑它们。

注意： 大多数硬件调试系统只能在应用没有执行时设置断点。

断点在编辑窗口的左边空白处用一个图标做标记:

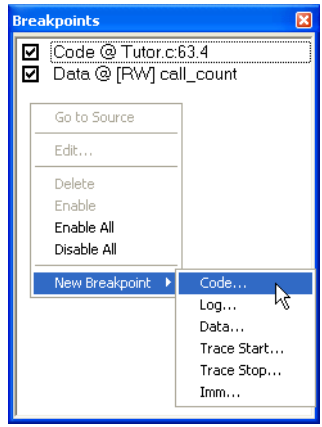


如果没有出现断点图标,那么就要确保，在 **IDE Options>Editor** 对话框中，已经选择了 **Show bookmarks** 选项 。

用鼠指到断点图标处，以获取详细的在同一个位置设置的所有断点的文字提示信息。第一行是断点信息，接下来的行描述用来实现用户断点的物理断点。后面的信息还可以在**断点用法**对话框中看到。

查看所有已定义的断点:

选择 **View>Breakpoints** 打开断点窗口，列出所有的断点。这里你可以方便地监视,开启和关闭断点；你还可以定义新断点，修改和删除现有的断点。



研究断点消费者:

打开断点使用窗口—在C-SPY驱动专用菜单上—可以查看到底层所有断点，你定义的和 C-SPY 内部使用的都有。

通常，目标硬件的 (C-SPY 用来设置断点的)硬件断点数量有限，有时只有一两个。超过可用硬件断点数量会迫使调试器单步执行程序。这会严重降低执行速度。

在硬件断点数量有限的硬件调试系统中，使用断点使用窗口来实现:

- 识别所有断点的消费者
- 检查目标系统支持的活动断点数量
- 如果可能的话,配置调试器以更好的方式使用可用的断点。

执行到达断点:

1 单击工具条上的**Go**按钮。应用程序会执行到设置的下一个断点。 调试日志窗口会包含断点触发的信息。

2 选择断点，右键单击并从上下文菜单选择 **Toggle Breakpoint (*xxx*)** 来删除一个断点。

查看终端 I/O

有时你不得不调试使用`stdin`和`stdout`的应用构造过程，没有硬件可以支持。C-SPY允许你通过使用终端I/O窗口来模拟`stdin`和`stdout`。

使用终端 I/O 窗口:

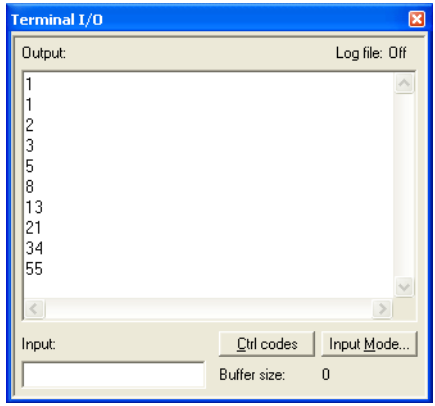
1 使用以下这些选项来构建你的应用:

类别	设置
Linker > Config (供XLINK使用)	有 I/O 仿真模块
General Options>Library Configuration (供ILINK使用)	库底层接口实现

这意味着一些底层程序的链接使得stdin和stdout直接指向终端I/O窗口。

2 构建你的应用并启动C-SPY。

3 选择 **View>Terminal I/O** 打开终端窗口，根据 I/O 操作显示输出结果。



分析应用的运行时行为

C-SPY提供多种功能用来分析你的应用的运行时行为，定位任何瓶颈并验证你的用来测试的应用的所有部件:

- 梗概
- 代码覆盖
- 堆栈用法
- 追踪。

梗概

你可以在两个梗概变体之间选择:

函数梗概 有助于找到函数执行最花费时间的地方。这些函数是你优化代码时应该集中注意的部分。最优化一个函数的简单的方法是使用速度最优化来编译它。或者,将函数转移到最高效寻找模式的内存中。

指令梗概 在反汇编窗口显示—每条指令已执行次数—的信息,可以在非常细的水平上帮助你调优代码，尤其是汇编源代码。

使用梗概:

1 使用以下这些选项来构建你的应用:

类别	设置
C/C++ 编译器	Output>Generate debug information
链接器(供XLINK 使用)	Format>Debug information for C-SPY
链接器(供ILINK使用)	Output>Include debug information in output

2 构建你的应用并启动C-SPY。

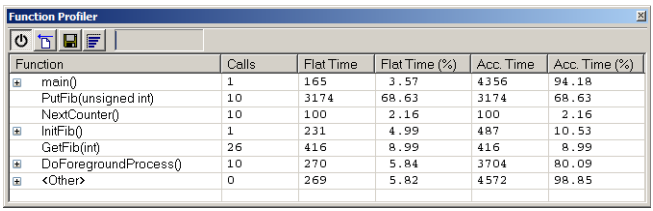
3 在使用梗概之前，必须先设置好。设置会随着C-SPY驱动和目标系统变化。

4 打开函数的梗概窗口，在驱动专用的菜单上选择**Profiling**。

5 单击 **Enable** 按钮开启梗概。

6 开始执行你的应用并收集梗概信息。

7 梗概信息显示在函数梗概窗口。



整理信息，单击相关的栏目标题。

8 在你开始新的采样之前，单击 **Clear** 按钮。

9 单击 **Graph** 按钮进行切换，将百分比栏目要么按数值要么者按直方图显示。

代码覆盖

代码覆盖有助于你的测试过程确保代码所有部分都得到执行。还有助于你识别那块代码没有执行到。

注意: 当你在硬件上调试时，代码覆盖可能会有限制；特别是，循环计数器的统计可能不能用。

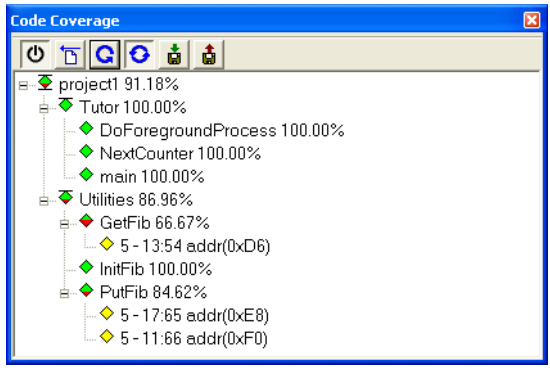
使用代码覆盖:

1 使用以下这些选项来构建你的应用:

类别	设置
C/C++ 编译器	Output>Generate debug information
链接器 (供 XLINK 使用)	Format>Debug information for C-SPY
链接器 (供 ILINK 使用)	Output>Include debug information in output
Debugger	Plugins>Code Coverage

2 构建你的应用并启动C-SPY。

3 选择 **View>Code Coverage** 打开代码覆盖窗口。



4 单击 **Activate**

按钮开启代码覆盖分析。

5 开始执行。当执行停止的时候，例如因为程序到达退出点或触发了断点，单击 **Refresh** 按钮来查看代码覆盖信息。

代码覆盖窗口现在报告当前代码覆盖分析的状态，也就是，代码的哪部分从开始分析起执行了至少一次。编译器以在每条语句和每个函数调用上的*节点*的形式生成详细的步骤信息。报告包括所有模块和函数的有关信息，直到应用停止执行时的全部节点的数量，按百分比，列出执行的以及所有没有执行的节点。

6 覆盖会持续不断,直到关闭时才停止。

注意: 代码覆盖还可以在反汇编窗口显示。执行的代码用绿色方块标识。

堆栈使用

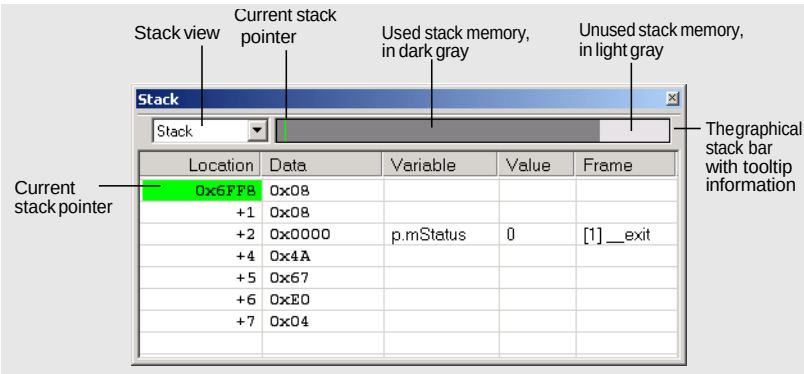
堆栈窗口显示堆栈的内容，包括堆栈变量在内存中如何布置。另外，一些堆栈的完整性检查可以用来发现堆栈溢出问题并报警。

堆栈窗口展示了堆栈的内容。 有助于以下时刻:

- 探究 C 模块调用汇编模块或反之时堆栈的使用。
- 探究要素是否正确地放置在堆栈中。
- 探究堆栈恢复是否正确。

追踪堆栈使用:

1. 选择 **Project>Options>Debugger>Plugins** 并在插件列表中选择 **Stack**。
2. 选择 **Tools>Options>Stack** 配置堆栈追踪。 请特别注意,你可能要指明堆栈指针什么时候是有效的。
3. 构建你的应用并启动C-SPY。
4. 选择 **View>Stack**。



你可以打开多个堆栈窗口的实例，每个都显示不同的堆栈—如果几个堆栈都可用—或者不同显示设置的同一个堆栈。

将鼠标指针放在堆栈条的上方就可以获取堆栈用法的工具提示信息。

侦测堆栈溢出:

选择 **Tools>Options>Stack** 并选择选项 **Enable stack checks**。

这意味着当应用停止执行的时候C-SPY可以发出堆栈溢出警告。要么堆栈使用超出你指定的阈值，要么堆栈指针超出堆栈存储区之外时都会发出警告。

追踪

通过收集追踪数据，你可以分析程序流到的特定状态（例如一个应用崩溃），并使用该追踪数据定位问题的源头。追踪数据有助于定位程序症状不规则并偶尔发生的错误。

追踪 是执行机器指令的一个序列集合。可用的追踪数据很大程度上取决于你使用的 C-SPY 驱动:

- C-SPY 仿真器收集你在追踪表达式窗口选择的C-SPY表达式的值。函数追踪窗口只显示相应的调用函数和函数返回的追踪数据，然而追踪窗口显示全部指令。
- 如果你使用的硬件支持的话,C-SPY硬件调试系统的驱动可以收集追踪数据。例如如果有专用的通信通道或者专用的追踪缓冲区供追踪收集。在这种情况下，追踪窗口会反映收集到的数据。

收集追踪数据:

1. 在仿真器中收集追踪数据不需要专门的构建设置。如果你正在使用硬件调试系统，则必须提前配置其追踪数据的产生。相关信息参考驱动的文档。
2. 构建你的应用并启动C-SPY。
3. 
在驱动专用菜单选择 **Trace** 打开追踪窗口，然后单击 **Activate** 按钮开启追踪数据收集。
4. 开始执行。当执行停止的时候，例如因为触发了断点，追踪数据显示在追踪窗口中。

利用断点开始追踪数据收集:

收集两个执行点之间的追踪数据的一种便利的做法是使用专用的断点起停数据采集。在编辑器或反汇编窗口中的上下文菜单上，右键单击来切换 **Trace Start** 或 **Trace Stop** 断点。 在 C-SPY 仿真器中，也可以使用 C-SPY 系统宏 `__setTraceStartBreak` 和 `__setTraceStopBreak`。