Segger SystemView使用手册（译文）

> **version：** *v1.0　　「2022.7.20」　　格式有部分问题*
>
> **author：** *Y.Z.T.*
>
> **摘要：** *装载自CSDN，为方便查看而整理*
>
> > 原连接：Segger SystemView使用手册（译文）

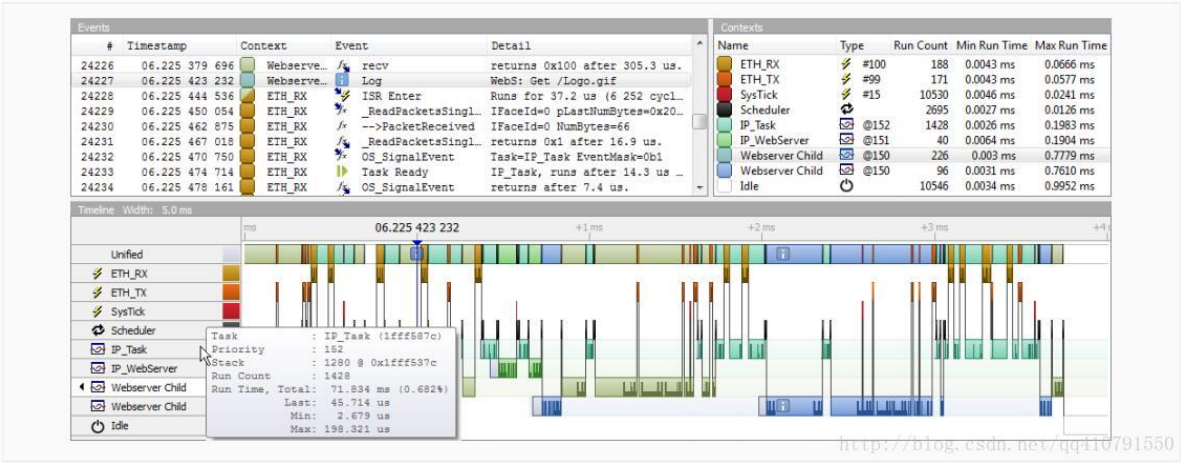
目录

章节1 概述 - Segger SystemView使用手册（译文）

1 概述

```
本节描述SEGGER SystemView的一般使用。

```



1.1 SEGGER SystemView 是什么?

SystemView 是一个用于虚拟分析嵌入式系统的工具包。SystemView 可以完整的深入观察一个应用程序的运行时行为，这远远超出一个调试器所能提供的。这在开发和处理具有多个线程和事件的复杂系统时尤其有效。

SystemView 由两个部分组成：

SystemView 的PC端程序，用于收集目标板上传的信息。

SystemView 嵌入式端程序可以分析嵌入式系统的行为。它记录嵌入式系统产生的监视数据，并在不同的窗口中显示这些信息。这些记录可以保存到文件中，用于以后的分析或者编写文档。

监视数据是通过调试接口来记录的，因此使用SystemView不需要额外增加硬件（甚至不需要增加额外的引脚）。它可以允许在任何一个包含调试接口的系统上。

使用一个SEGGER J-Link和实时传输技术（RTT），SystemView可以连续的记录数据，并实时的分析和展现这些数据。
SystemView可以分析中断、任务和软件定时器已经执行了多久，何时和它们使用了多长时间。它揭示了在某个命令中的发生了什么，哪个中断触发了任务切换，中断和任务调用了哪个底层RTOS的API函数。可以执行周期精确的分析，甚至可以对用户功能进行计时。
SystemView应该用来验证嵌入式系统预期的行为，可以发现问题与不足，比如不必要的、虚假的中断，和意料之外的任务变化。它可以被用于任何调用SystemView事件函数的实时或非实时的操作系统，但是也支持Non-Instrumented的RTOS或者不带RTOS的应用，它可以分析中断执行时间，以及像时间关键的子程序。

如何工作?

需要在目标板上调用一个小的软件模块，它包含了SYSTEMVIEW和RTT。SYSTEMVIEW模块用于收集和格式化监视数据，并将数据传送给RTT。RTT模块可以将数据保存在目标板的buffer中，使用J-Link可以实现连续的记录数据，用户可以选择覆盖式记录或者不覆盖记录方式（阻塞方式）。
目标板可以某些场合中调用SYSTEMVIEW函数来监视事件，例如中断开始和中断结束时。SystemView将这些事件以及一个可配置的高精度时间戳一起保存到RTT的目标缓冲区中。时间戳可以精确到1个CPU周期（在200MHz的CPU上是5ns）。

目标板资源需求

RTT和SYSTEMVIEW模块的ROM需求小于2 KB。在典型的系统中，大约600字节的RAM就足以实现使用J-Link进行连续记录。对于系统触发的记录，缓冲区大小取决于记录的时间以及事件的数量。不需要其他硬件。对于典型的事件记录，CPU只需要低于1us的时间来处理(基于200 MHz Cortex-M4 CPU)，也就是说，在系统每秒10000个事件的系统中，只需要不到1%的开销。由于调试接口(JTAG，SWD，FINE)用于传输数据，因此不需要额外的引脚。

哪些CPU可以使用SystemView?

SystemView可以用在任何CPU上。连续实时的记录可以在任何支持J-Link RTT技术的系统上进行。RTT需要在程序执行过程中通过调试接口读取内存，通常支持ARM Cortex-M0、M0+、M1、M3、M4处理器以及所有的Renesas RX设备。
对于那些不支持RTT技术的系统中，当系统halt时，可以手动读取缓冲区的内容，这允许填充缓冲区并后续分析之前保存的单段记录，以捕获最新的记录数据。当记录开始和停止时，系统可以触发单段记录或者后期记录。

添加到目标系统需要多少工作量?

不是很多。少量的文件需要加到makefile文件或者工程文件中。如果操作系统支持SystemView，那么只需要调用一个函数。在一个没有RTOS或者non-instrumented RTOS的系统中，需要将两行代码添加到应该监视的每个中断函数中。这些工作就是全部了，一共不超过几分钟就能完成。

1.2 SEGGER SystemView 程序包

下面这几节描述了如何安装SEGGER SystemView程序包及其内容。

1.2.1 下载及安装

SEGGER SystemView 程序包提供了Windows、OS X和Linux系统下安装程序以及可移植存档。

从 https://www.segger.com/systemview.html下载最新的程序包。

为了能够使用该程序包，需要安装配套的J-Link软件和文档包。可以在https://www.segger.com/jlinksoftware.html找到下载地址和说明。

1.2.1.1 Windows

安装包

从http://www.segger.com/systemview.html下载最新的安装包并执行。安装向导会指引完成安装。
安装完成之后，可以通过Windows开始菜单目录或者文件管理器访问程序包内容。

可移植压缩包
从http://www.segger.com/systemview.html下载最新的压缩包，解压到文件系统的任意目录。
这种方式不需要安装，解压后就可以直接使用程序包内容。

1.2.1.2 OS X

安装包

从http://www.segger.com/systemview.html下载最新的pkg包并执行。安装向导会指引完成安装。
安装完成之后，可以通过Launchpad访问程序包内容。

1.2.1.3 Linux

系统需求

要在Linux上运行SystemView，必须安装Qt V4.8库。

安装包

从http://www.segger.com/systemview.html下载最新的DEB包或者RPM包并执行。安装向导会指引完成安装。

可移植压缩包

从http://www.segger.com/systemview.html下载最新的存档，解压到系统的任意目录。
这种方式不需要安装，解压后就可以直接使用程序包内容。

1.2.2 程序包内容

SEGGER SystemView程序包 包含了用于应用程序跟踪的一切内容 — 主机可视化SystemView程序和用于快速容易使用的示例跟踪文件。
嵌入式应用程序中包含的目标源代码可以作为附加包单独下载。
包含有用于SEGGER软件（如embOS）下快速使用的额外代码。

下表列举了软件包内容。

SystemView 软件包

文件	描述
./SystemView.exe	SystemView分析和可视化软件
./Doc/UM08027_SystemView.pdf	本文档
./Description/SYSTEMVIEW_*.txt	SystemView API描述文件
./Sample/OS_IP_WebServer.SVDat	用于一个web服务器应用的SystemView示例跟踪文件
./Sample/OS_Start_LEDBlink.SVDat	一个简单embOS应用程序的SystemView示例跟踪文件
./Sample/uCOS_Start.SVDat	一个简单的uC/OS-III应用程序的SystemView示例跟踪文件

目标板源码包

文件	描述
./Src/Config/Global.h	用于SystemView的全局数据类型
./Src/Config/SEGGER_RTT_Conf.h	SEGGER RTT配置文件
./Src/Config/SEGGER_SYSVIEW_Conf.h	SEGGER SYSTEMVIEW配置文件
./Src/Sample/embOS	用于embOS的SystemView初始化和配置
./Src/Sample/FreeRTOSV8	用于FreeRTOS V8的SystemView初始化和配置
./Src/Sample/FreeRTOSV9	用于FreeRTOS V9的SystemView初始化和配置
./Src/Sample/MicriumOSKernel	用于Micrium OS Kernel的SystemView初始化和配置
./Src/Sample/NoOS	用于裸机系统的SystemView初始化和配置
./Src/Sample/uCOS-III	用于uC/OS-III的SystemView初始化和配置
./Src/SEGGER/SEGGER.h	全局类型和通用功能函数
./Src/SEGGER/SEGGER_RTT.c	SEGGER RTT模块源码
./Src/SEGGER/SEGGER_RTT.h	SEGGER RTT模块头文件
./Src/SEGGER/SEGGER_SYSVIEW.c	SEGGER SYSTEMVIEW模块源码
./Src/SEGGER/SEGGER_SYSVIEW.h	SEGGER SYSTEMVIEW模块头文件
./Src/SEGGER/SEGGER_SYSVIEW_ConfDefult.h	SEGGER SYSTEMVIEW配置回调
./Src/SEGGER/SEGGER_SYSVIEW_Int.h	SEGGER SYSTEMVIEW内部头文件

1.3 SystemView PRO

SystemView可以在任何商业或者非商业目标系统中免费使用。它包含了全面分析系统行为的所有功能。

SystemView PRO扩展了这些功能，以提供更好的系统分析方法。首先，它提高了100万条事件的限制，并允许无限制的记录。此外，它还附带了一些新特性，比如自定义过滤器，可以轻松地搜索列表中的事件。

1.3.1 许可

SystemView PRO许可是单用户许可方式。
许可可以储存在J-Link，之后这个J-Link扮演了USB Dongle的作用。这样，只需要简单的插入你的J-Link就可以在你的任何电脑上使用SystemView PRO，例如你的台式机、家用笔记本。

章节2 开始使用SystemView - Segger SystemView使用手册（译文）

2 开始使用SystemView

这一节描述如何开始使用SEGGER SystemView。解释了如何分析基于监视数据分析一个应用程序。

本章节参考的示例数据文件是 OS_IP_WebServer.SVDat。这个文件包含在SEGGER SystemView程序包中。

该示例数据文件展示了运行了embOS实时系统，embOS/IP TCP/IP协议栈和一个web服务器应用程序的目标系统的行为。

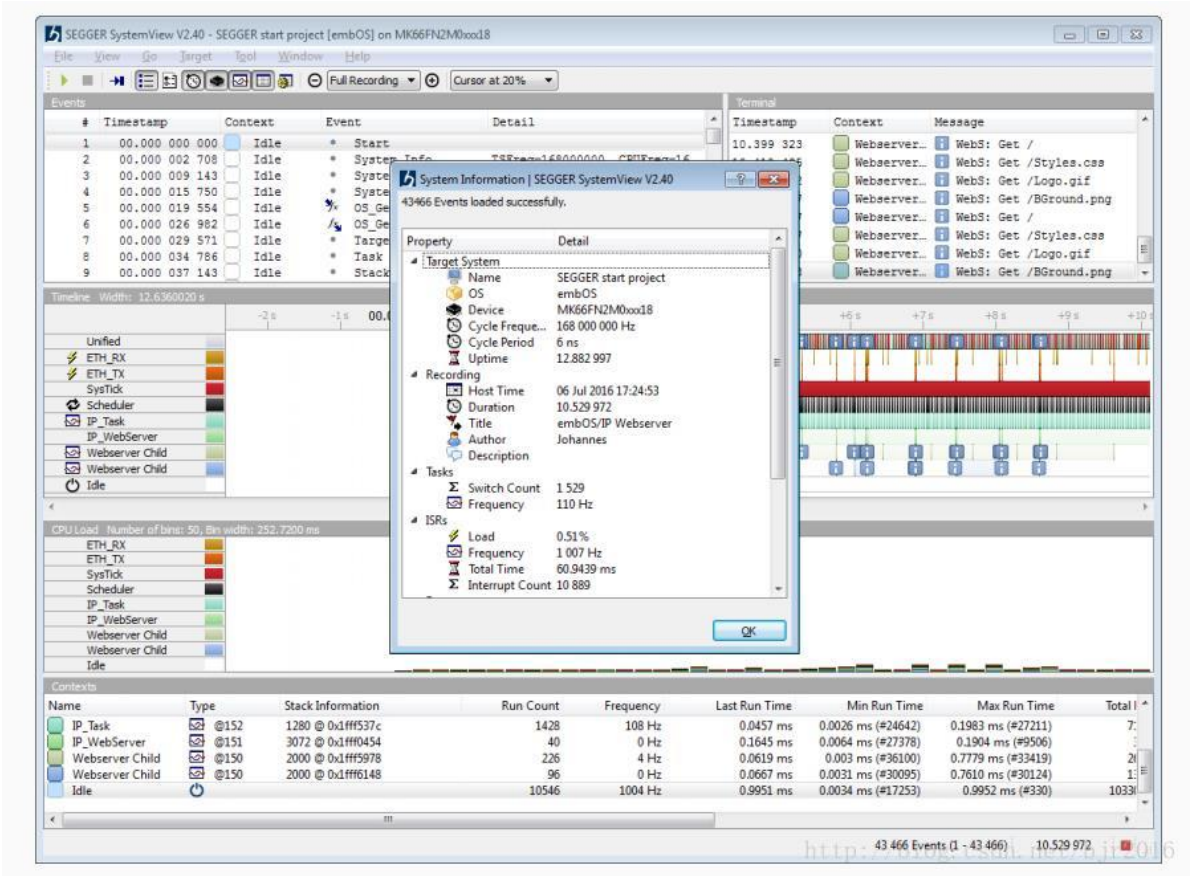
我们将使用SEGGER SystemView获取到的信息来分析应用程序正在做的事。

2.1 启动SystemView并加载数据

SystemView能够在线监控目标应用程序数据。这些监控数据能够保存成一个文件，用于后续工作。并且保存下来的数据能够在没有J-Link，甚至没有目标硬件或者目标应用程序的情况下进行分析。因此允许没有物理访问权的开发人员对系统进行分析。

- 从开始菜单或者安装目录启动SystemView程序（SystemView.exe）
- 在第一次启动SystemView时，它会提示打开示例记录。点击 Yes
- 在启动后，选择 File->Sample Recordings -> 程序包安装目录程序包安装目录/Sample/OS_IP_WebServer.SVdat。

SystemView将加载并分析这些数据，展示加载的记录的系统信息，你会看到下图所示：



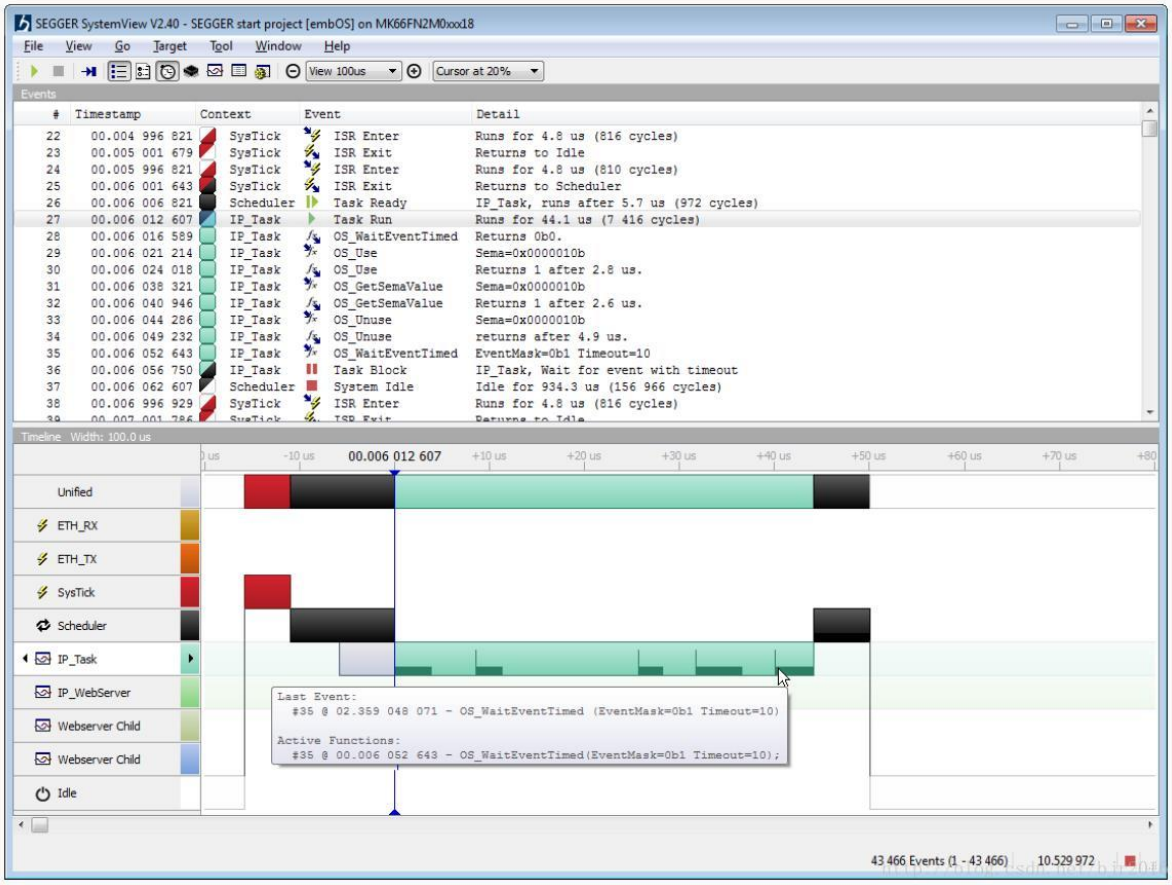
*加载了数据文件之后的SystemView**

2.2 首先看一下这个系统

我们将首先看一下这些数据，得到关于监控系统的一些信息。

系统信息

在加载数据后显示的系统信息对话框提供了对记录的第一个概述。它显示有关目标系统的信息、任务的记录和统计信息、中断和事件。系统信息由应用程序发送，因此SystemView不需要任何额外的配置来分析和显示系统行为。



SystemView时间线

时间轴（Timeline）

时间轴窗口显示完整的监控数据。在Events列表中，滚动到第一项开始。

时间轴窗口通过系统时间将系统活动可视化(任务、中断、调度器和空闲)。每行引用一个上下文项，我们可以看到在被监视的应用程序中使用的所有项

在开始时，我们可以看到有两个任务: IP_Task和IP_WebServer，用亮色的背景颜色标注。

放大的时间轴宽度到2.0ms，并双击'+1000us'下面的垂直线到中间并选择该项。(使用鼠标滚轮，工具栏项，[Ctrl]+[+]/[-]键,或View->Zoom In，View->Zoom Out来缩放)。

SysTick每毫秒中断都有一些系统活动。

将鼠标移动到背景名称上，以获得有关context类型的更多信息和运行时间信息。

单击IP_Task背景的右箭头按钮，跳转到下一个执行。放大或缩小显示活动的细节。

我们可以看到SysTick中断返回到OS调度器，这使得IP_Task已经就绪，由IP_Task的行中的灰色条表示，并让它运行。IP_Task从embOS API函数os_waiteventTimed中返回，返回值0，表示在当前的时间点没有事件发生。

IP_Task调用了另外三个具有快速返回和os_waiteventTimed的embOS API函数，它激活了调度器，使任务失效，并使系统处于空闲状态（idle）。当事件(EventMask = 1)发生时，或者10ms超时之后，IP_Task将再次激活。

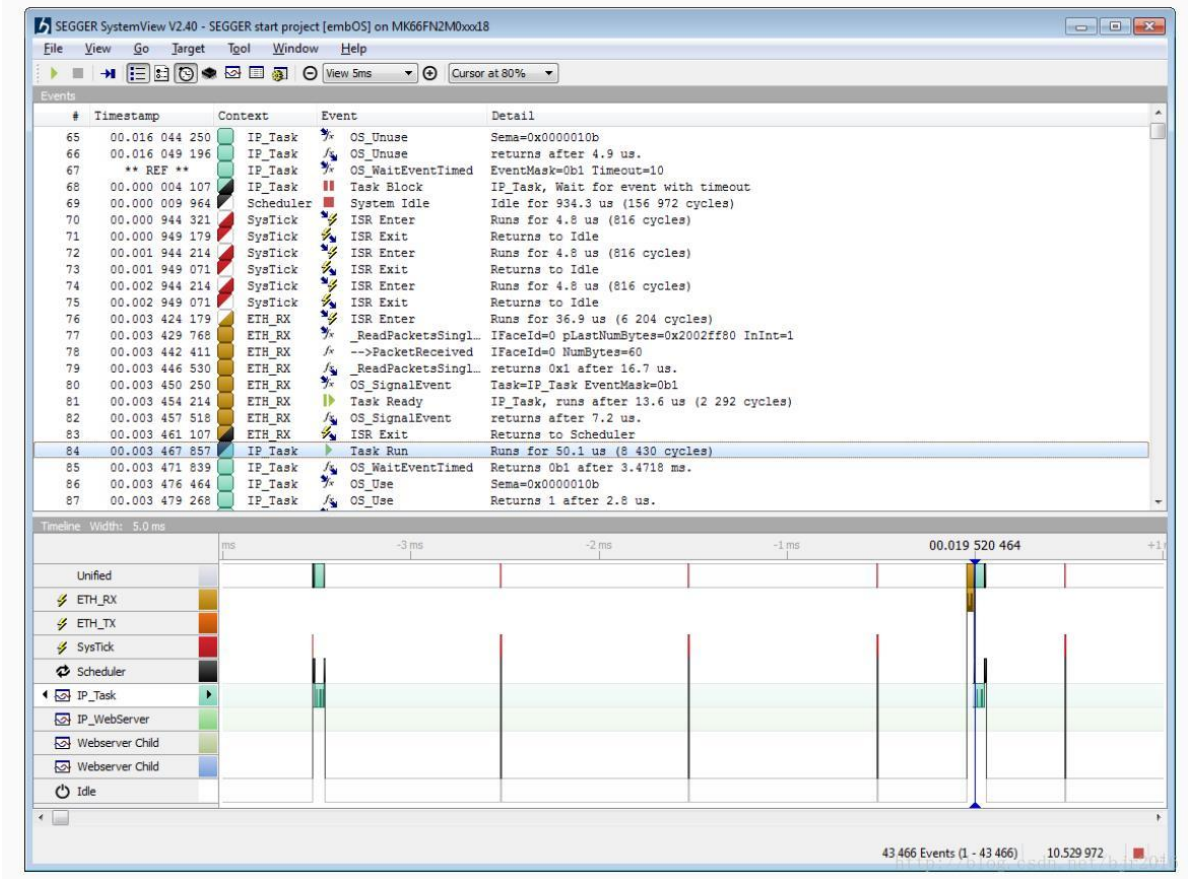
记录的函数调用在时间轴上是显示为背景栏中的小条。垂直峰值线表示函数的调用，这些小条背景显示调用的长度。堆叠栏图形展示了嵌套函数调用。

结论

我们已经得到一些关于监测系统的第一手资料。从时间轴我们知道应用程序使用了哪些任务和中断，它由1 kHz SysTick中断控制，同时IP_Task至少每10 ms激活一次。

2.3 分析系统活动

在得到一些系统信息之后，我们将分析系统是如何被激活的。



系统事件列表

事件列表

事件（Events）列表显示了从系统发送的所有事件，并显示它们的信息，包括事件的时间戳、活动上下文、事件类型和事件细节。它与时间轴同步。

我们已经知道Systick每毫秒进出一次中断，并且每10ms超时发生时激活 IP_Task任务。

通过 View->Go to Event... （快捷键为：Ctrl+G）我们转到67#事件，在00.016 052 607时间点，调用了OS_WaitEventTimed，超时参数设置为10ms，也就是在00.026 052 607时间点这个函数会超时退出。

可以通过View -> Events -> Toggle Reference 、 右键 ->Toggle Reference或者快捷键R 为事件设定一个参考时间点，之后的事件列表中的所有时间戳都将以最新的参考时间点作为0时间点。

现在看看IP_Task运行的是由于超时引起的还是由于等待的事件有效引起的，使用Go->Forward（快捷键F）我们切换到IP_Task的下一个活动。

此时的时间点事00.003 467 857，相对于参考时间点只过去了3ms，并未达到10ms的超时。所以任务是由于等待的事件变为有效而运行的。

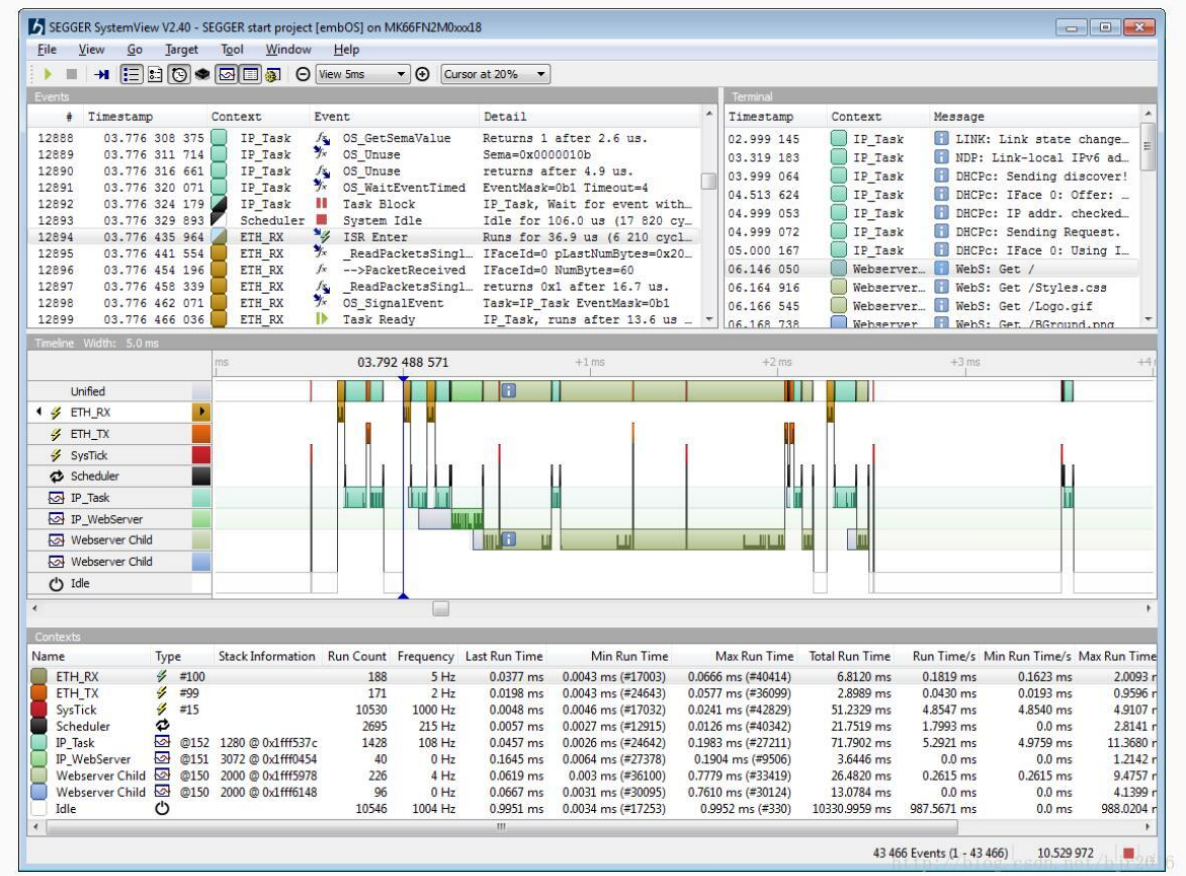
我们可以看到之前发生的ETH_Rx中断，我们通过以太网（在接口0上的60字节）。因此ETH_Rx中断标记了这个事件，正如时间轴上显示的那样。然后ETH_Rx中断返回到调度器（Scheduler）。IP_Task任务运行，并从OS_WaitEventTimed函数返回，返回值为0b1，表示事件发生而不是超时。

结论

通过深入跟踪事件，我们可以看到IP_Task有2种情况会被激活运行：一是10ms发生超时，二是我们收到数据包并且产生了ETH_Rx中断。

2.4 进一步分析应用程序核心

我们现在知道系统主要受ETH_Rx中断控制，下一步是去看看当它被激活时，系统还干了什么。



*SystemView 应用程序分析

时间轴（Timeline），事件列表（Events list），终端和上下文窗口

SystemView的各窗口是同步的，并且一起使用时，提供了最佳的分析策略。

应用程序创建了一个web服务器，可以通过浏览器访问embOS/IP示例web页面。在web服务器运行和浏览器多次加载web页面时已经收集了示例数据。

log输出、时间戳和活动上下文均通过SystemView发送，并显示在终端窗口中。

在终端窗口中选择一个消息，会同步在事件窗口和时间轴上同步选择。时间轴指示了所有的终端数据数据。

通过这些消息，我们可以查看以太网连接是什么时间建立的，并选择浏览器在加载根页面的时候发出的“WEBS: Get /”。

在#12894号事件前面右击，查看更详细的分析。

这里我们看到发生了ETH_Rx中断，其在中断调用了embOS/IP函数 ReadPacketsSingleIF，并接收数据包。在标记了收到正如之前所看到的那个embOS事件后，退出中断到调度器，然后激活IP_Task任务。

IP_Task任务置位了表明IP_WebServer任务准备好的事件。另外的数据包立即被收到，并由IP_Task任务处理。当IP_WebServer开始运行时，会在accept()函数中调用一些系统函数然后返回。然后它检查Webserver子任务是否存在，如果不存在则创建它。

在创建任务时，任务会被添加到上下文，并且当它没有被激活时会在时间轴上标记为一个亮色背景。

IP_WebServer 在accept()函数中等待另外一个连接到来，WebServer处理接收到的http请求，并为web页面服务。当WebServer被激活，可能被ETH_Rx中断打断，而这个中断可能会导致一个任务切换到更高优先级的IP_Task任务。

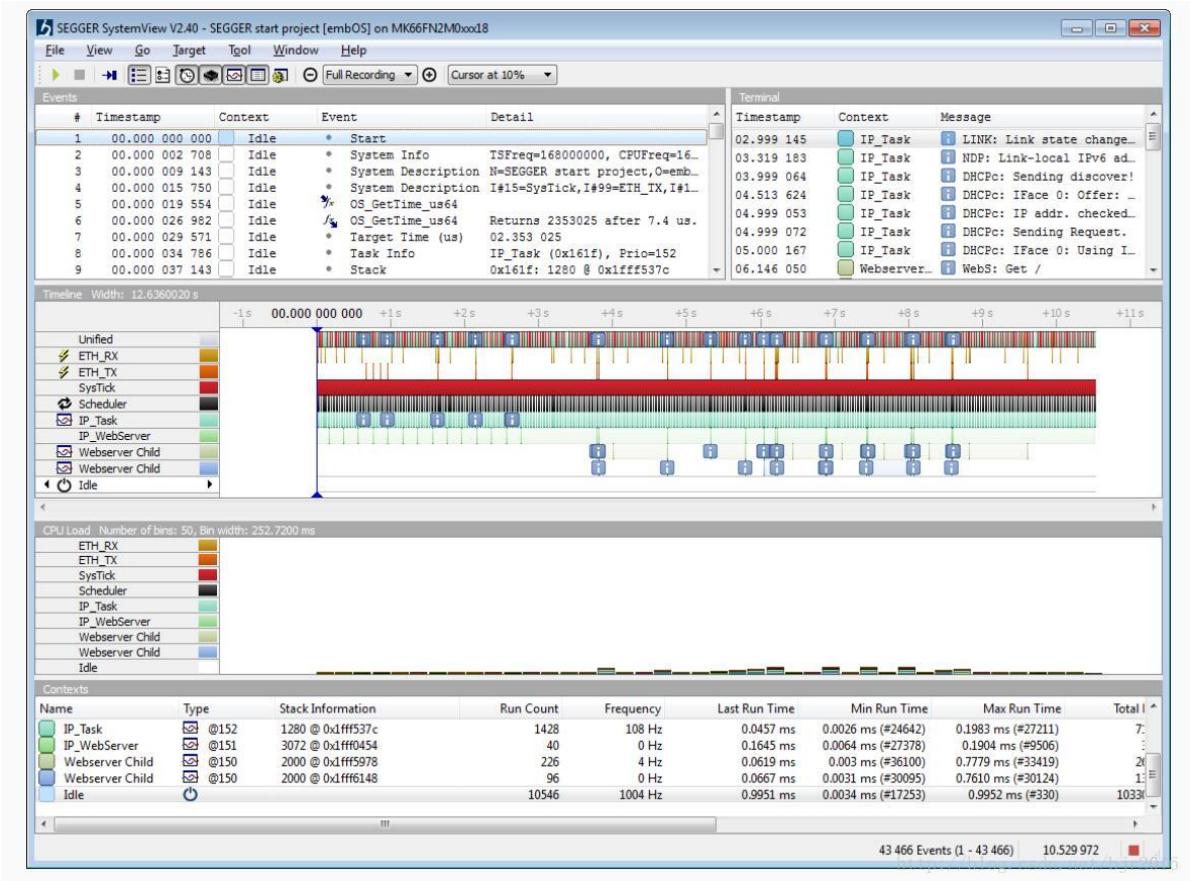
注意：任务在事件轴上按优先级排序，在上下文窗口中可以看到精确的任务优先级。

2.4.1 分析结论

我们在对应用程序代码不了解的情况下分析了系统具体做了哪些事情。通过应用程序源代码，我们可以使用SEGGER SystemView检查系统所做的事是否和期望的一致。

章节3 SystemView PC端应用程序 - Segger SystemView使用手册（译文）

3.1 介绍



SystemView 程序

这个SystemView程序是SEGGER SystemView主机端可视化工具软件。它和目标板通过J-Link连接，控制跟踪并读取应用程序的数据。实时分析监视数据并显示在SystemView不同的窗口中。再跟踪停止后，数据可以保存成一个文件以便于后续分析。

如何启动SystemView，请参考前面的章节。

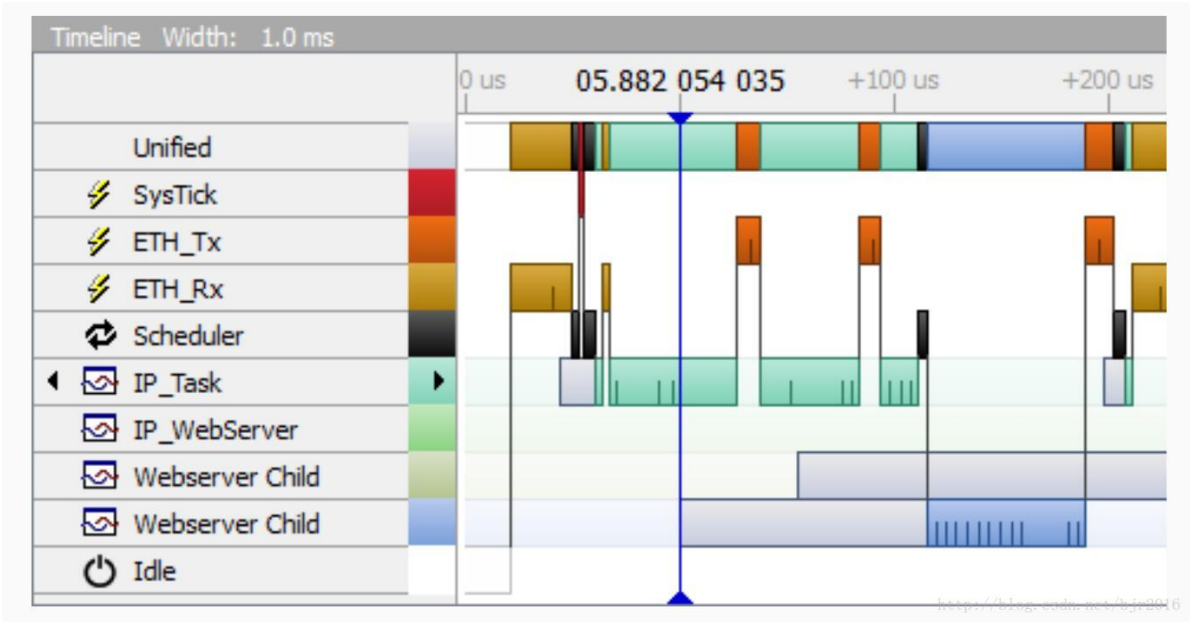
SystemView提供不同的窗口来展示系统的运行，测量时间并分析CPU负载。所有窗口都是同步的，以获得当前选定状态的所有信息。

有关应用程序窗口的描述，请参阅以下部分。

使用SystemView，可以通过监视的数据跟踪系统中发生的情况。

3.2 时间轴窗口（Timeline）

以下时间轴全部表示为Timeline



SystemView Timeline

Timeline窗口在一个界面中集中了所有系统信息。它在系统时间线上显示了系统活动(任务、中断、调度器、定时器和空闲)。它显示在被监视的应用程序中使用的所有上下文项，且每行显示一个上下文项。

在鼠标经过上下文项时，会有tip工具显示更多的细节和运行时信息。

在鼠标经过活动上下文时，tip工具提示显示当前事件的详细信息和当前活动的函数（如果可用的话）。

在上下文上面的那个尺子标记了当前活动的时间段范围。

存在的任务会被标记为浅色的背景，提供了从它第一次运行一直到它结束期间的每一次运行。

上下文切换会被现实为连接线，以便于地识别哪些事件导致上下文切换，并且何时发生。

处于ready状态的任务，在开始执行前会被显示在浅灰色的栏中。

上下文是按优先级排序的。第一行显示了一个统一上下文中的所有活动。列表的顶部是中断，用Id来排序的，接下来是scheduler调度器和软件定时器（如果在系统中使用这些的话）。在调度器(和定时器)下面，任务按优先级排序。当没有其他上下文处于活动状态时，底层上下文显示空闲时间。

Timeline是和事件列表Events list同步显示的。在光标（蓝色行）下面的事件是列表中被选中的事件。

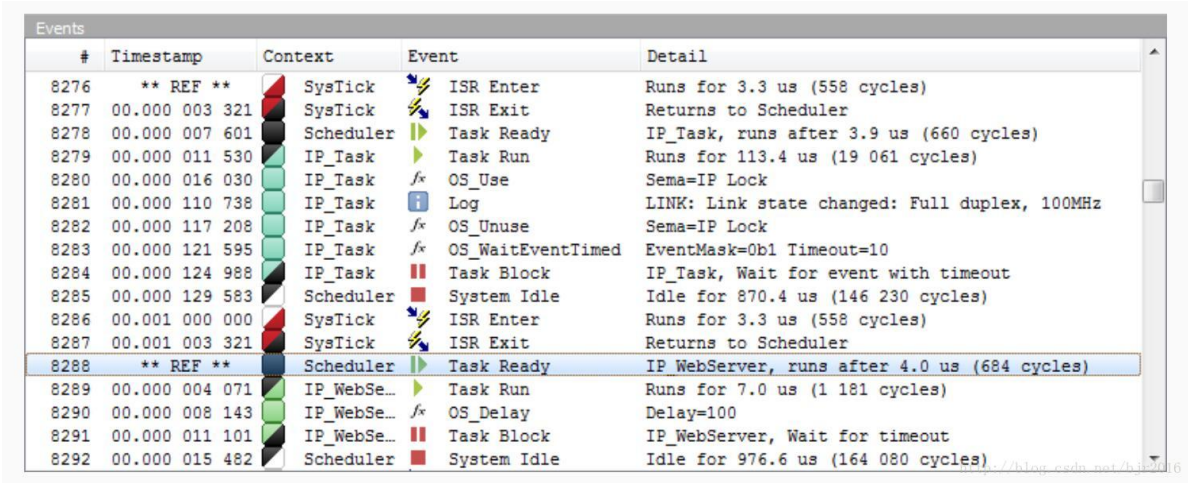
光标可以固定在10%到90%的窗口，并在滚动时间线时更新列表中的选择。

可以将事件拖放到事件列表中，来选择事件列表中的相应事件，反之亦然。

时间轴视图可以放大或缩小，来了解整个系统的概况，或者查看事件的确切时间。

要跳转到上下文的下一个或之前的活动，可以用鼠标单击左侧的箭头或者右侧的箭头。

3.3 Events 窗口



SystemView Events窗口

Events窗口显示系统发送的所有事件，并显示它们的信息。每个事件都有以下几项：

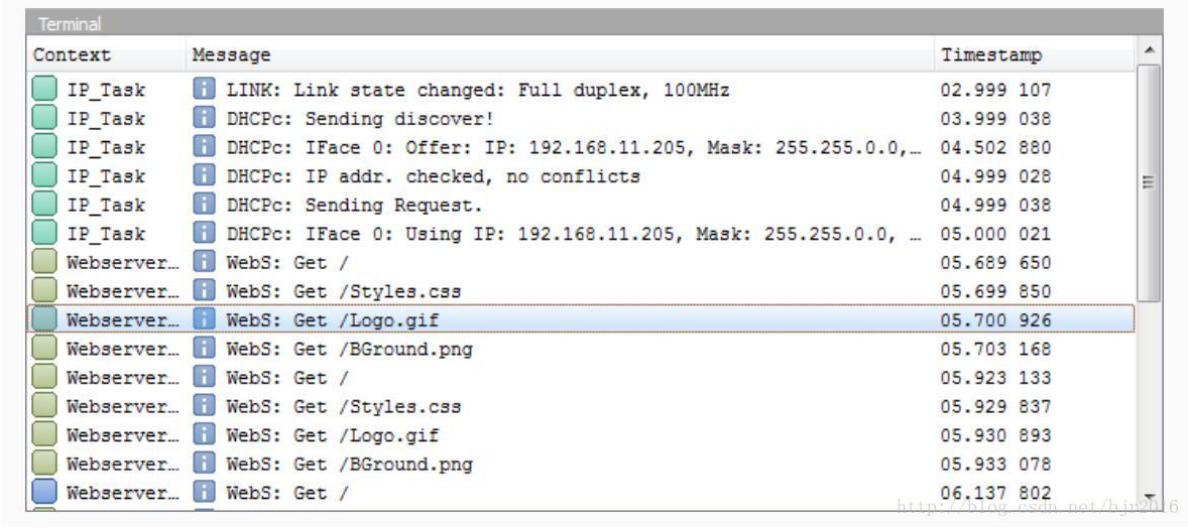
- 在目标时间或记录时间内的时间戳，可以用微秒或纳秒分辨率显示
- 创建Events的上下文，即运行的任务。
- Event描述，和事件类型一起显示，(IRS进入和退出，任务活动，API调用)。
- Event细节描述事件的参数，即API调用参数。
- 在列表中定位事件的ID。

Event窗口允许通过列表，跳转到下一个或之前的上下文，或跳转到下一个或之前的类似事件。Timeline和CPU负载窗口同步显示当前选中的Event。

Events列表中的时间戳可以相对于记录的开始或者系统发送的目标系统时间。Event可以设置为跟踪事件的时间引用，以便在Event发生后可以轻松地测量Event。

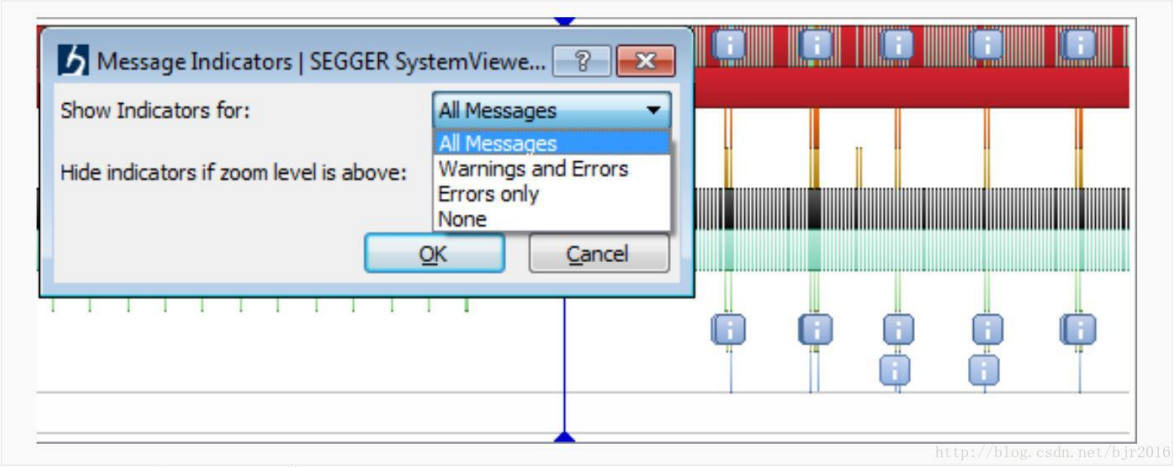
SystemView包括一个Events筛选器来选择show或hide api、ISRs、系统信息、消息、任务和用户事件。

3.4 Terminal窗口



SystemView Terminal窗口

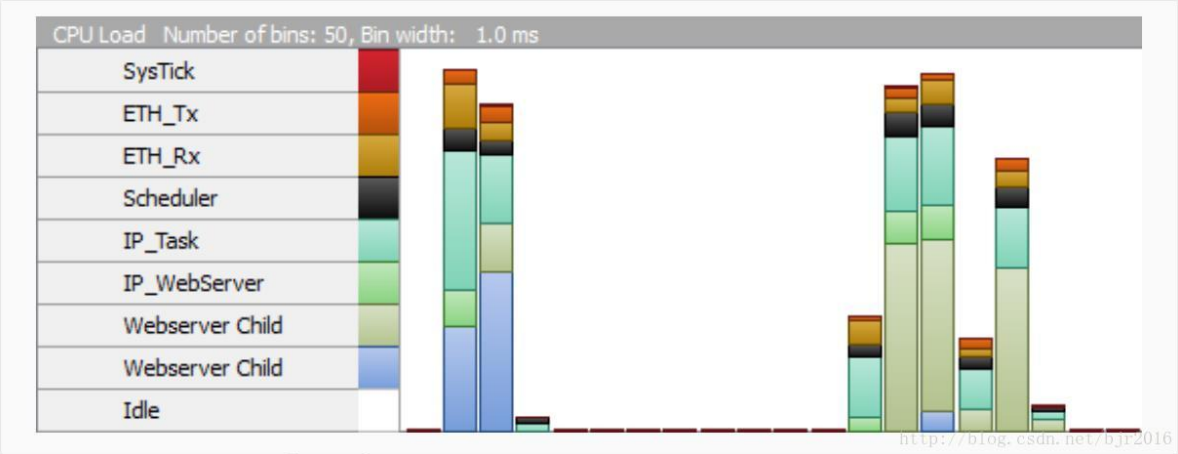
终端窗口显示来自目标应用程序的printf输出，以及发出log输出的任务上下文，以及发送消息时的时间戳。双击消息，可以显示出该消息在事件列表中的所有信息。
Timeline窗口显示的是输出的指示器，根据级别排序（错误总是排在顶部）。显示的日志级别可以通过View -> Message Indicators... 来配置。



Message Indicator 对话框

SystemView printf的输出是由通过用户应用程序格式化的，也可以所有参数先不格式化，而通过SystemView应用程序进行格式化。

3.5 CPU Load窗口

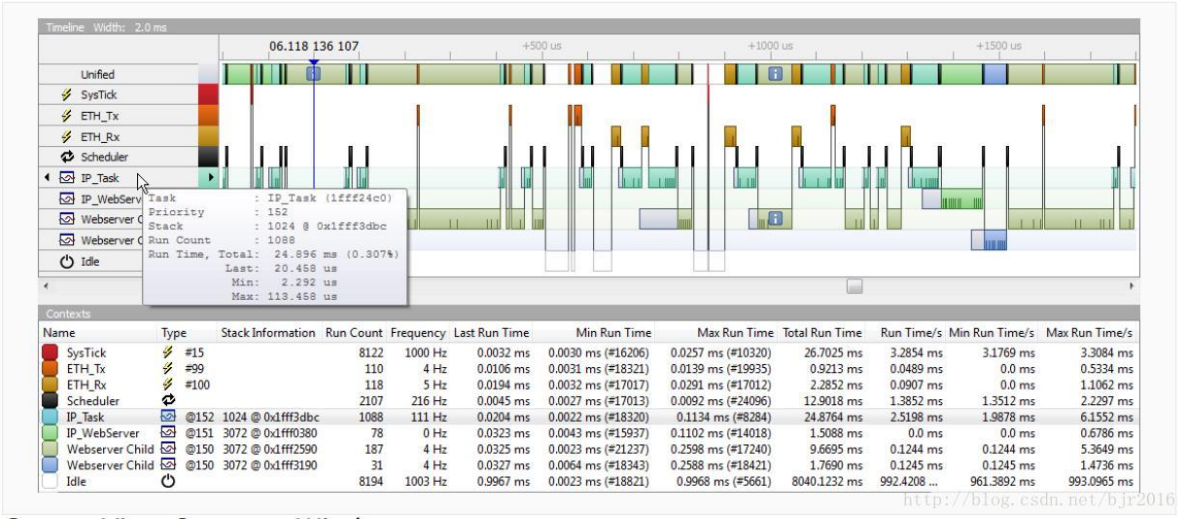


CPU Load窗口

CPU load窗口显示了一个周期内上下文占用的CPU时间。CPU负载是用一个使用Timeline当前分辨率的bin的宽度来测量的，因此与缩放级别是同步的。

可以通过选择bin的数量用来测量较短或较长时期的负载。使用一个bin，就可以在整个可见的时间轴上测量CPU负载。

3.6 Context窗口



Context窗口

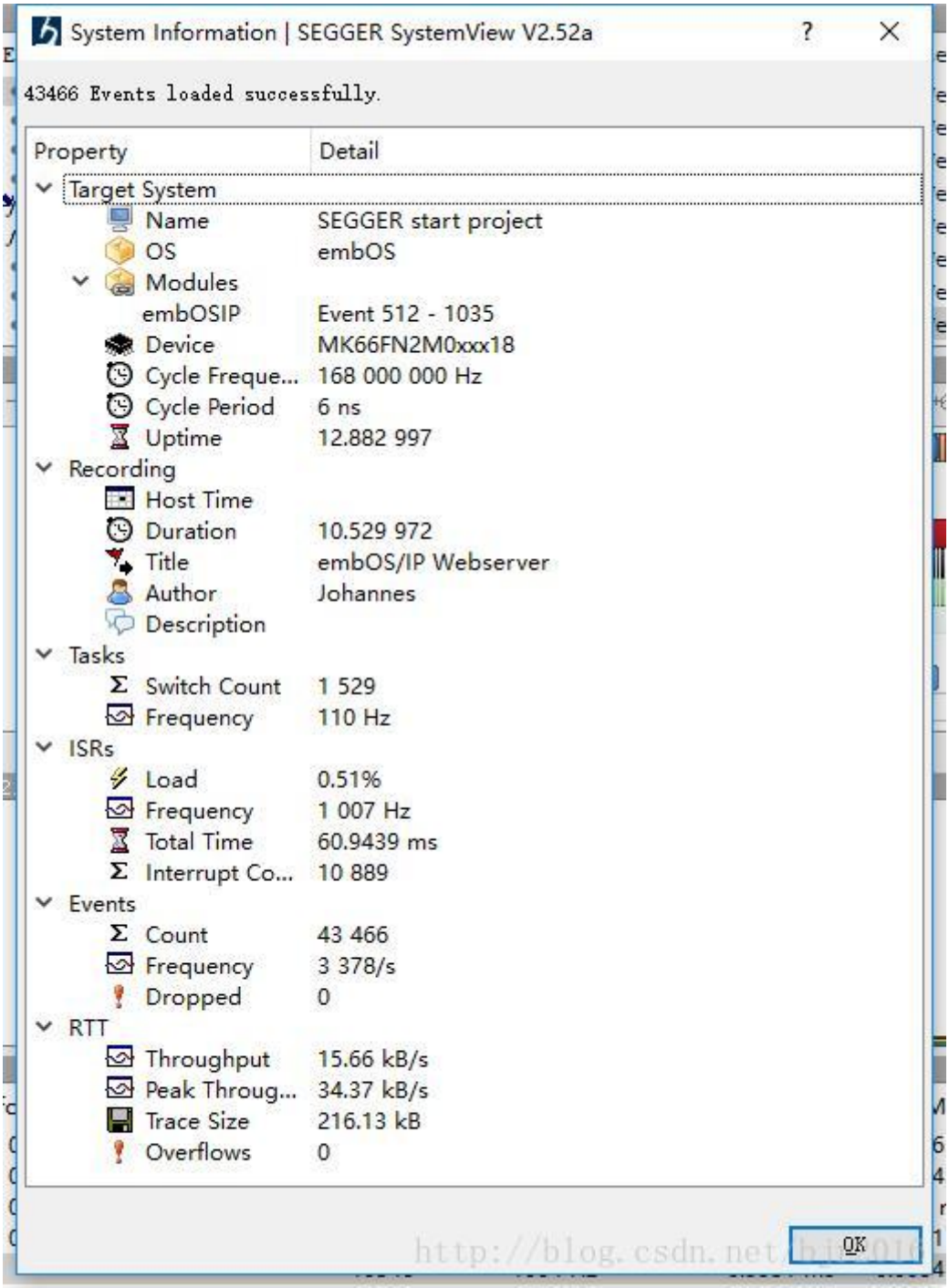
Context窗口显示上下文的统计信息(任务、中断、调度器和空闲)。每个上下文项可以通过它的名称和类型来标识。类型包括任务的优先级和中断的ID。(例如，cortex-m的SysTick是中断ID # 15)。

Context窗口信息包括以下内容：

- 上下文名称和类型。
- 任务堆栈信息。（如果有的话）
- 上下文的活动计数。
- 活动频率。
- 总的运行时间和最后运行时间。
- 每秒当前的、最小和最大运行时间，单位是ms和%。

在记录时，上下文窗口是实时更新的，而当前上下文可以通过选择行来指示。

3.7 系统信息（System information）窗口

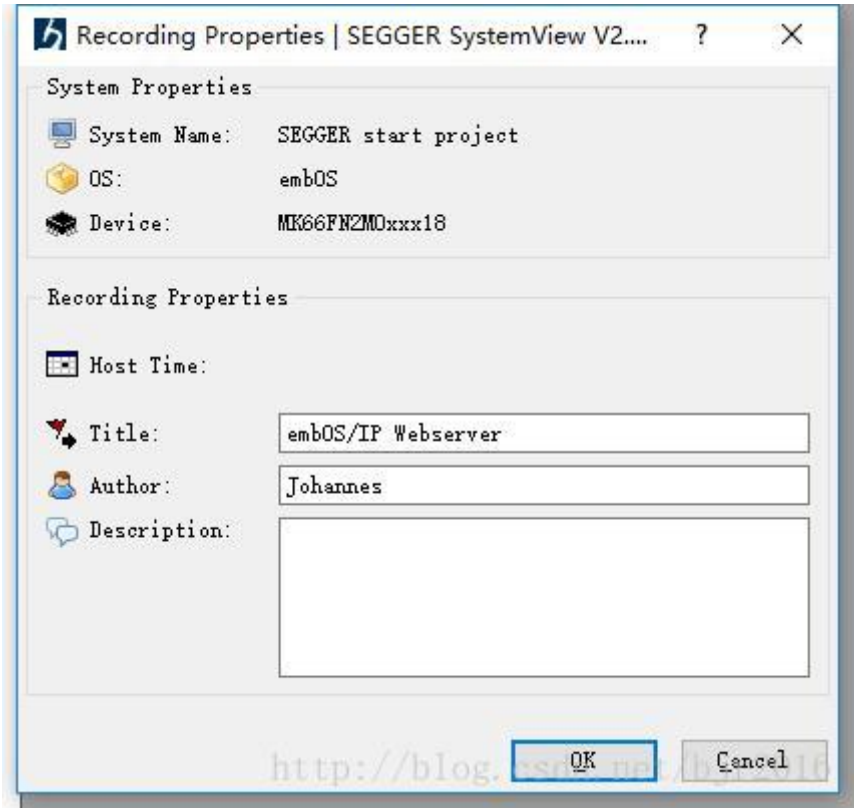


系统信息窗口

右上角的窗口显示：

- 系统的一些信息，是由应用程序发送用来识别该系统的。
- 记录属性，可由用户设置。
- 有关任务、中断、SystemView事件和记录吞吐量的统计信息。

系统信息包括应用程序名称、使用的OS、目标设备和时间信息。有关任务开关和中断频率的附加信息提供了系统的快速概览。



记录属性对话框

记录的属性可以由用户在保存时进行设定，并和记录一起存储，并允许在加载后识别记录，以便稍后进行分析。

3.8 事件过滤器 Event Filter

事件窗口对事件进行筛选。这可以用于隐藏中断事件或只显示任务执行。
可以设定不同的组过滤SystemView事件：

- api - OS或模块生成事件。
- ISRs - 进出中断。
- 消息 - 终端输出。
- 系统事件 - 系统和任务信息。
- 任务 - 任务执行。
- 用户事件 - 用户事件启动和停止。

SystemView PRO还提供自定义过滤器，它允许选择显示或隐藏的任何事件。使用curstom过滤器，可以单独选择所有系统事件和注册的操作系统和中间件事件。

3.9 触发模式（Trigger Modes）

当SystemView在连续模式下记录时，记录的事件会实时分析和显示。触发器模式允许在发生特定事件时自动选择事件。

触发器模式可以在工具栏中选择，也可以在任务的右键菜单中选择，并在Timeline窗口中中断。

在手动滚动模式下，选择不会自动更新。在记录时用户可以滚动事件，分析系统。

在自动滚动模式下，自动同步成100ms的时间步伐。会自动选择在最后一个100ms记录的事件。

在连续触发器模式下，用户可以配置上下文(任务或中断)，以及触发哪个事件。然后SystemView总是选择最后发生的符合配置条件的事件。

在单触发器模式下，SystemView只选择一次满足配置条件的事件，并自动切换回手动滚动模式。

3.10 界面操作

SystemView可以通过鼠标和键盘来控制，也可以通过菜单来控制。最重要的控件也可在工具栏中访问。

下表描述了SystemView的操作：

记录数据

含义	菜单	快捷键
目标板开始记录	Target->Start Recording	F5
停止记录	Target->Stop Recording	F9
从系统读取先前的或者单次数据	Target->Read Recorded Data	Ctrl+F5
保存记录的数据到一个文件中	File->Save Data	Ctrl+S，F2
加载一个记录文件	File->Load Data	Ctrl+O，F3
加载一个最近使用的文件	File->Recent Files	无
加载一个示例记录	File->Sample Recordings	无
查看/修改记录属性	File->Recording Properties	Ctrl+Shift+R

查看

含义	菜单	快捷键
设定/清除当前事件作为时间参考	View->Toggle Reference	R
移除所有时间参考	View->Clear References	Ctrl+Shift+R
显示目标的时间戳	View->Display Target Time	无
显示从记录起始所有时间戳	View->Display Recording Time	无
设定时间戳分辨率为1us	View->Resolution:1us	无
设定时间戳分辨率为100ns	View->Resolution:100ns	无
设定时间戳分辨率为10ns	View->Resolution:10ns	无
设定时间戳分辨率为1ns	View->Resolution:1ns	无
放大	View->Zoom->Zoom In	Ctrl++，滚轮向上
缩小	View->Zoom->Zoom Out	Ctrl–，滚轮向下
Timeline(时间轴）显示到XX us	View->Zoom->XX us Window	无
Timeline(时间轴）显示到XX ms	View->Zoom->XX ms Window	无
Timeline(时间轴）显示到XX s	View->Zoom->XX s Window	无
在Timeline显示所有记录	View->Zoom->Full Recording	无

含义	菜单	快捷键
将光标设定到Timeline(时间轴)的10%	View->Cursor->Cursor at 10%	1
将光标设定到Timeline(时间轴)的20%	View->Cursor->Cursor at 20%	2
将光标设定到Timeline(时间轴)的30%	View->Cursor->Cursor at 30%	3
将光标设定到Timeline(时间轴)的40%	View->Cursor->Cursor at 40%	4
将光标设定到Timeline(时间轴)的50%	View->Cursor->Cursor at 50%	5
将光标设定到Timeline(时间轴)的60%	View->Cursor->Cursor at 60%	6
将光标设定到Timeline(时间轴)的70%	View->Cursor->Cursor at 70%	7
将光标设定到Timeline(时间轴)的80%	View->Cursor->Cursor at 80%	8
将光标设定到Timeline(时间轴)的90%	View->Cursor->Cursor at 90%	9
在1个bin中测量CPU负载	View->CPU Load->Single Bin	无
在10个bin中测量CPU负载	View->CPU Load->10 Bins	无
在50个bin中测量CPU负载	View->CPU Load->50 Bins	无
在100个bin中测量CPU负载	View->CPU Load->100 Bins	无
在200个bin中测量CPU负载	View->CPU Load->200 Bins	无
显示/隐藏事件列表中的API调用事件	View->Event Filter->Show APIs	Shift+A
显示/隐藏事件列表中的中断进出事件	View->Event Filter->Show ISRs	Shift+I
显示/隐藏事件列表中的消息（Message）	View->Event Filter->Show Messages	Shift+M
显示/隐藏事件列表中的系统事件	View->Event Filter->Show System Events	Shift+S
显示/隐藏事件列表中的任务活动事件	View->Event Filter->Show Tasks	Shift+T
显示/隐藏事件列表中的用户事件	View->Event Filter->Show User Events	Shift+U

含义	菜单	快捷键
在事件列表中只显示API调用事件	View->Event Filter->Show APIs only	Ctrl+Shift+S
在事件列表中只显示中断进出事件	View->Event Filter->Show ISRs only	Ctrl+Shift+I
在事件列表中只显示消息Message	View->Event Filter->Show Messages only	Ctrl+Shift+M
在事件列表中只显示任务活动事件	View->Event Filter->Show Tasks only	Ctrl+Shift+T
在事件列表中只显示用户事件	View->Event Filter->Show User Events only	Ctrl+Shift+U
复位所有事件过滤器	View->Event Filter->Reset all Filters	Ctrl+Shift+Space
自动滚动到新事件上的最后一项	View->Auto Scroll	无
选择要在时间轴上显示的消息指示器	View->Message Indicators	Ctrl+M
在记录时，手动滚动事件	工具条	Ctrl+1
在记录时，自动滚动事件	工具条	Ctrl+2
在记录时，不停地触发和滚动一个条件	工具条	Ctrl+3
在记录时，单次触发和滚动一个条件	工具条	Ctrl+4
配置触发条件	工具条	Ctrl+T

运行

含义	菜单	快捷键
跳到下一个上下文切换	Go->Forward	F
跳到前一个上下文切换	Go->Back	B
跳到下一个相似事件	Go->Next [Event]	N
跳到前一个相似事件	Go->Previous [Event]	P
跳到同一个上下文中的下一个相似事件	Go->Next [Event] in [Context]	Shift+N
跳到同一个上下文中的前一个相似事件	Go->Previous [Event] in [Context]	Shift+P
打开根据Id跳转到事件的对话框	Go->Go to Event...	Ctrl+G

含义	菜单	快捷键
打开根据时间戳跳转到事件的对话框	Go->Go to Timestamp...	Ctrl+Shift+G
向前方滚动（未来）	Go->Scroll Forward	左键，Ctrl+滚轮向上滚，单击并拖动
向后方滚动（以前）	Go->Scroll Back	右键，Ctrl+滚轮向下滚，单击并拖动

窗口

含义	菜单	快捷键
显示/隐藏 事件窗口	Window->Events View	E
显示/隐藏 Timeline（时间轴）窗口	Window->Time View	T
显示/隐藏 CPU Load（CPU负载）窗口	Window->CPU Load View	L
显示/隐藏 Contexts（上下文）窗口	Window->Context View	C
显示/隐藏 Terminal（终端）窗口	Window->Terminal View	M
显示/隐藏 System information（系统信息）窗口	Window->Sytem View	S
显示/隐藏 Log（日志）窗口	Window->Log View	O
显示/隐藏 状态栏	Window->Show/Hide Status Bar	无
显示/隐藏 工具栏	Window->Show/Hide Tool Bar	无
打开程序首选项对话框	Tool->Preferences	Alt+,

帮助

含义	菜单	快捷键
打开SystemView手册（即本手册）	Help->SystemView Manual	F11
显示SystemView版本信息	Help->About SystemView	F12

3.11 命令行选项

SystemView可以通过命令行选项进行控制和配置。在启动记录时要跳过配置对话框，可以通过命令行选项给出目标配置。

```
C:> SystemView.exe [-usb [<SN>]]|[-ip <Host>] [-device <Device>] [-if SWD|
JTAG|FINE] [-speed <Speed>] [-rttcbaddr <Addr>]|[-rttcbrange auto|<Range>]
[-
single [-port <Port>] [-wait]] [-start|-stop|-read|-quit|-save
[<Filename>]|-
```

```
load[<Filename>]]
Command Line Options:
-usb    Connect to J-Link via USB. Parameter: S/N of JLink. (Optional)
（要连接的J-Link S/N号，参数为J-Link的S/N号，可选）

-ip     Connect to J-Link via IP. Parameter: IP or S/N of J-Link.
（通过IP地址连接J-Link,参数为IP地址或者J-Link的S/N号）

-device     Set the target device. Parameter: Device name as supported by J-Link.
（目标设备名，参数为J-Link支持的设备名，一般为芯片型号）

-if         Set the target interface. Parameter: SWD, JTAG,or FINE.
（目标板连接接口。参数为：SWD,JTAG或者FINE）

-speed      Set the target interface speed. Parameter: Speed in kHz.
（目标板接口连接速率。参数为：速率，单位是kHz）

-jtagconf   Set the JTAG scan chain configuration. Parameter: IRPre and DRPRe of the target device.
（设定JTAG扫描链配置。参数为：目标设备的IRPre和DRPRe）

-rttcbaddr      Set the RTT Control Block address. Parameter: Address in hexadecimal.
（RTT控制块地址。参数为：十六进制地址）

-rttcbrange     Set the search range for RTT Control Block. Parameter: auto or ranges as "<Address> <Size>".
（RTT控制块寻址范围。参数为：auto或者表示成"<Address><Size>".）

-single     Start SystemView in single instance mode.
（单一实例模式下的单启动系统视图。）

-port       Set local port for single instance mode. Parameter: Port.
（设定单实例模式下的本地端口。参数为：端口号）

-wait       Wait for first instance to return result.
（等待第一个实例返回结果）

-start      Start recording.
（启动记录）
-stop       Stop recording.
（停止记录）

-read       Read recorded data from target.
（从目标板读取记录）

-quit       Quit SystemView.
（退出SystemView）

-save       Save current recording. Parameter: File to save to. (Optional)
（保存当前记录到文件，参数为：要保存到的文件名。（可选））

-load       Load a recording from file. Parameter: File to
（从文件中加载一个记录，参数为：要加载的文件名）
```

```
55
56 load.       (Optional)
```

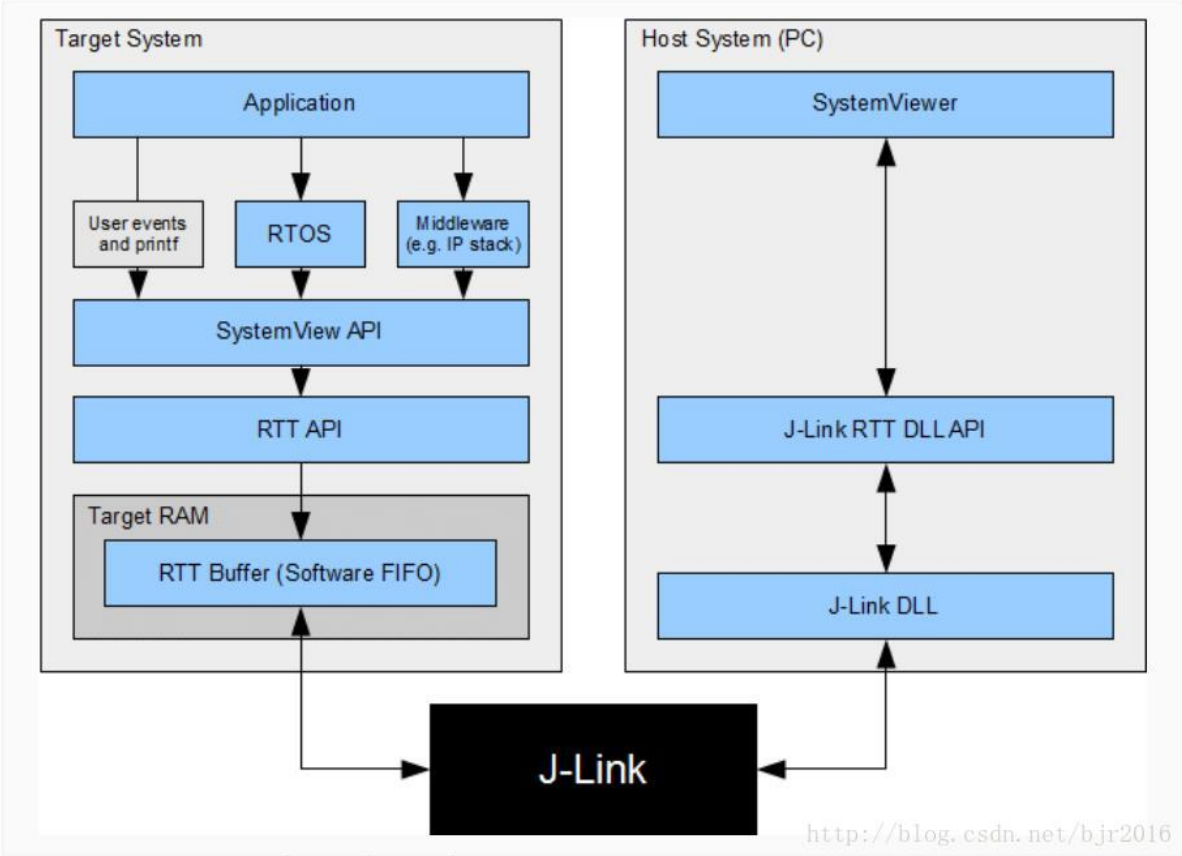
章节4 使用SystemView进行记录 - Segger SystemView使用手册（译文）

章节4 使用SystemView进行记录

本节描述了如何使用SystemView程序进行连续的记录，和如何使用调试器手动进行单次记录。

4.1 连续录制

基于J-Link调试探针技术和SEGGER实时传输技术（RTT），SystemView可以在目标程序运行时实时地记录目标执行情况。RTT需要在程序执行过程中通过调试接口读取内存的能力。这特指包括ARM Cortex-M0、M0 +、M1、M3、M4和M7处理器以及所有的Renesas RX设备。



SystemView使用J-Link是如何工作的

开始录制

要使用SystemView开始记录，首先需要连接J-Link和目标板，并单击Target->Start Recording。

输入或者选择设备名。下拉列表会列举最近使用的设备。如果当前设备不是列表的一部分，可以手动输入。可以在 https://www.segger.com/jlink_supported_devices.html 查看支持的设备名。

> **注意**

> 对于RTT控制块自动检测，以及对于某些设备来说，准确的设备必须是已知的。建议不要仅仅选择一个核心。 *(brj2016注：比如只选Cortex-m0，而不选择具体的芯片名）。*

如果连接到一个特定的J-Link或者一个通过TCP / IP连接的J-Link时，需要配置J-Link连接。

选择连接目标设备的接口和接口速度。

配置RTT控制块检测。在大多数情况下可以使用自动检测。如果RTT控制块不能被检测到，则选择一个RTT控制块的搜索范围，或者直接输入准确的地址。

单击 OK 来连接到目标设备并开始录制。

> **注意**
>
> SystemView可以与调试器同时使用。在这种情况下，正在进行调试时可以同时进行记录。确保在调试器中完成了所有必需的配置。当调试器停止时，SystemView记录也会停止。

SystemView会连续地从目标板读取数据，并在运行时更新窗口。

停止录制

要停止录制，选择 Target->Stop Recording。

免费版本的SystemView只能录制1 000 000个事件，之后会自动停止。

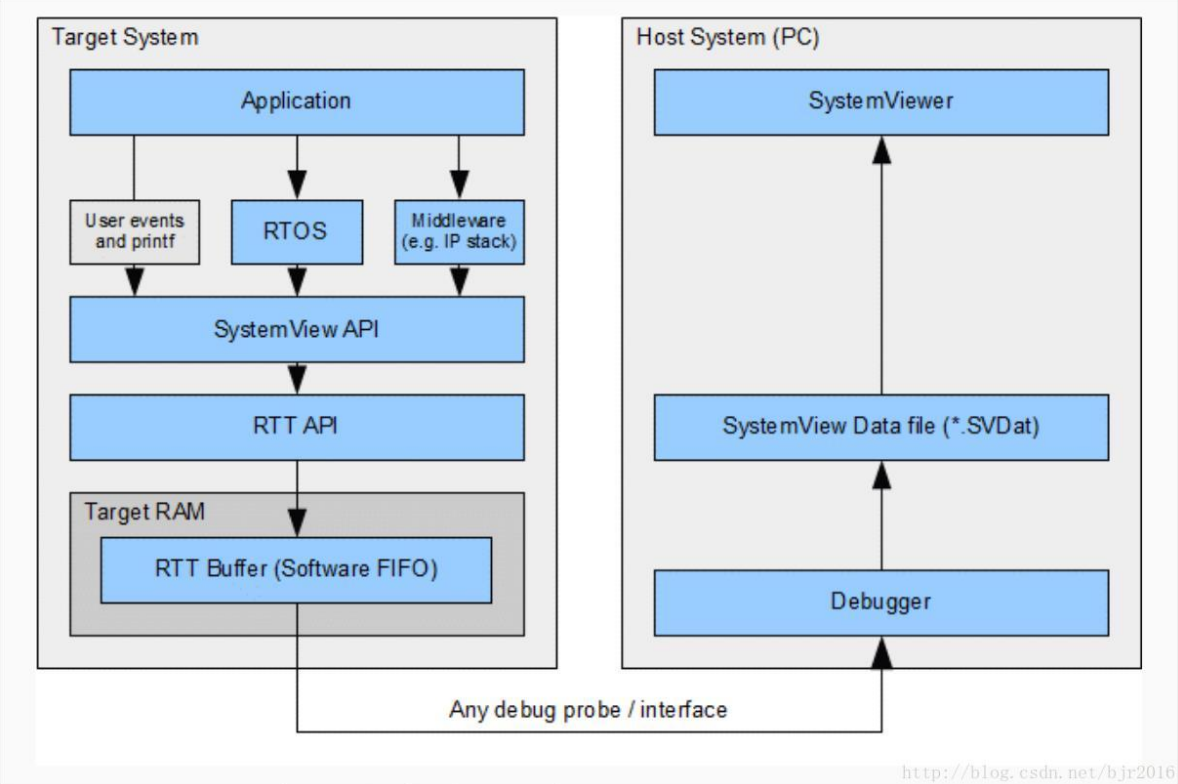
4.2 Single-shot录制

这里的Single-shot是指 只录制一个事件的意思，而不是单次录制直到停止录制

当目标设备不支持RTT或没有使用J-Link时，SEGGER SystemView可以用于记录数据，直到其目标缓冲区被填充满。

在Single-shot模式下，在应用程序中手动启动录制，只允许录制特定的感兴趣的部分。

由于通常的应用程序每秒可以生成大约5到15个kByte的记录数据，而在100%负载时能达到更高的速率，甚至内部RAM中的一个小缓冲区也可以用于记录关键部分的分析数据。当使用外部RAM SystemView时，即使是在Single-shot模式下也可以记录很长时间。



在Single-shot模式下SystemView如何工作

从系统中获取single-shot数据

要获取录制在single-shot模式下的数据，必须通过SystemView程序或外部调试器读取SystemView缓冲区。

- 连接调试器，并加载目标板应用程序
- 使用函数 SEGGER_SYSVIEW_Conf() 或者 SEGGER_SYSVIEW_Init() 配置并初始化SystemView。
- 在应用程序中需要被分析的地方使用函数 SEGGER_SYSVIEW_Start() 启动录制。

SystemView可以使用J-Link从目标板自动读取single-shot数据。

- 启动SystemView程序，并选择 Target->Read Recorded Data。

在没有J-Link或者没有SystemView的情况下，可以通过下列步骤读取数据：

- 在缓冲区满了或者录制完成后，在调试器中暂停应用程序。
- 得到SystemView RTT缓冲区地址和已使用字节数。（正常是 *SEGGER_RTT.aUp[1].pBuffer和* SEGGER_RTT.aUp[1].WrOff 两个参数）
- 从缓冲区取出已使用字节数个数的字节，保存成后缀名为 .SVDat的文件。
- 使用SystemView程序打开这个文件。

为了能够录制不止一次，缓冲写入偏移量(_SEGGER_RTT.aUp[1]. WrOff)可以在读取数据时设置为0。为了防止发生SystemView溢出事件，应该在缓冲区填满且不能保存下一个SystemView事件时停止应用程序。

4.3 事后分析 Post-mortem

Post-mortem分析和single-shot很像，但有一个区别：SystemView事件会被连续地记录下来，而SystemView缓冲区会在缓冲区填满时覆盖旧的事件。在读取缓冲区时，可以读取到最新的事件。

当系统运行很长时间并突然崩溃时，Post-mortem分析会很有用。在这种情况下，可以从目标板读取读取SystemView缓冲区，SystemView程序可以显示系统在崩溃前发生的情况。

> **注意**
>
> 使用Post-mortem分析，调试器必须连接在目标设备上。（不需要复位设备或者修改RAM）。

为了得到尽可能多的有用数据，建议给SystemView配置一个大的缓冲区（大于等于8k字节）。外部RAM可以用作SystemView缓冲区。
为了配置目标系统为Post-mortem模式，请参阅第59页的章节目标配置中的 SEGGER_SYSVIEW_POST_MORTEM_MODE 和 SEGGER_SYSVIEW_SYNC_PERIOD_SHIFT 。

从系统中获得Post-mortem数据

要获取在Post-mortem模式中记录的数据，必须通过SystemView应用程序或外部调试器读取SystemView缓冲区。

- 使用函数 SEGGER_SYSVIEW_Conf() 或者 SEGGER_SYSVIEW_Init() 配置并初始化SystemView。
- 在应用程序中应该被分析的地方启动录制。
- 连接调试器，加载目标应用程序，并让系统跑起来。

使用J-Link，SystemView程序可以自动从目标板读取post-mortem数据。

- 启动SystemView，并选择Target->Read Recorded Data。

在没有J-Link或者没有SystemView的情况下，可以通过下列步骤读取数据：

- 在缓冲区满了或者录制完成后，在调试器中暂停应用程序。

- 得到SystemView RTT缓冲区地址和已使用字节数。（正常是 *SEGGER_RTT.aUp[1].pBuffer和* SEGGER_RTT.aUp[1].WrOff 两个参数）
- 从缓冲区取出已使用字节数个数的字节，保存成后缀名为 .SVDat的文件。
- 使用SystemView程序打开这个文件。

SystemView可以使用J-Link从目标板自动读取single-shot数据。

- 启动SystemView程序，并选择 Target->Read Recorded Data。

在没有J-Link或者没有SystemView的情况下，可以通过下列步骤读取数据：

由于SystemView缓冲区是一个环形缓冲区，因此数据可能需要在两个块中读取，以便在开始时开始读取数据，并尽可能多地保存数据

- 在应用程序中配置和初始化SystemView，使用函数SEGGER_SYSVIEW_Conf() 和 SEGGER_SYSVIEW_Init。
- 在应用程序应该被分析的地方启动录制。
- 连接目标板和调试器，下载目标板应用程序，并启动。
- 当系统崩溃或所有测试完成时，使用调试器停止系统。
- 得到SystemView RTT缓冲区（通常是_SEGGER_RTT.aUp[1].pBuffer）。
- 将pBuffer + WrOff 直到缓冲区末尾的数据保存到一个文件。
- 将从pBuffer到pBuffer + RdOff - 1的数据追加到文件中。
- 将这个文件保存成 *.SVdat 或者* .bin。
- 使用SystemView程序打开这个文件。

4.4 保存和加载记录

当停止记录时，可以将记录的数据保存到文件中，以便稍后进行分析和文档化。选择 File->Save Data。记录属性对话框弹出，允许保存标题、作者和数据文件的描述。单击OK。选择保存目录并单击Save。可以通过File->Load Data打开已保存的数据。也可以通过File->Recent菜单看到最近使用的数据文件。SystemView程序可以打开*.bin和*。SVDat 文件。

章节5 目标板程序SystemView模块的实现 - Segger SystemView使用手册（译文）

章节5 目标板程序SystemView模块的实现

本节描述了目标应用程序如何添加SEGGER SystemView模块。

5.1 先决条件

想要使用SEGGER SystemView，首先要把SystemView源码文件加到目标应用程序工程中。SEGGER SystemView源码包可以从https://www.segger.com/systemview.html下载。

要实现连续实时跟踪，还有以下几项要求：

- 一个ARM Cortex-M 或者 Renesas RX 核的目标设备
- 一个J-Link调试器

~~要想快速把SystemView用起来，推荐使用embOS V4.12或者更新版本（已经集成在SystemView中子）。~~

5.1 SEGGER SystemView目标实现模块

以下文件是SEGGER SystemView目标实现的一部分。我们建议将所有文件复制到应用程序工程中，并维持目录结构。

文件	描述
/Config/Global.h	SEGGER代码的全局类型定义
/Config/SEGGER_RTT_Conf.h	SEGGER RTT配置文件
/Config/SEGGER_SYSVIEW_Conf.h	SEGGER SYSTEMVIEW配置文件
/Config/SEGGER_SYSVIEW_Config_[SYSTEM].c	[SYSTEM]的SystemView初始化
/OS/SEGGER_SYSVIEW_[OS].c	SYSTEMVIEW和[OS]之间的接口
/OS/SEGGER_SYSVIEW_[OS].h	接口头文件
/SEGGER/SEGGER.h	定义了SEGGER全局类型的全局头文件和通用函数
/SEGGER/SEGGER_RTT.c	SEGGER RTT模块源码
/SEGGER/SEGGER_RTT.h	SEGGER RTT模块头文件
/SEGGER/SEGGER_SYSVIEW.c	SEGGER SYSTEMVIEW模块源码
/SEGGER/SEGGER_SYSVIEW.h	SEGGER SYSTEMVIEW模块头文件
/SEGGER/SEGGER_SYSVIEW_ConfDefault.h	SEGGER SYSTEMVIEW模块默认配置头文件
/SEGGER/SEGGER_SYSVIEW_Int.h	用于SEGGER SYSTEMVIEW模块内部调用的头文件

5.2 在应用程序中引用SEGGER_SystemView

要使用SEGGER SystemView，目标实现模块必须添加到目标程序中。从文件夹/Config/和/SEGGER/中拷贝源码文件，以及复制匹配你程序所使用的系统和OS的SEGGER_SYSVIEW_Config*[SYSTEM].c和SEGGER_SYSVIEW*[OS].c及其头文件，并在工程中引用它们。

示例

例如一个在Cortex-M3芯片上使用了embOS的系统，那么应该引用SEGGER_SYSVIEW_Config_embOS.c，SEGGER_SYSVIEW_embOS.c和SEGGER_SYSVIEW_embOS.h。

而对于一个在Cortex-M3芯片上的没有操作系统或者no instrumented OS的系统，那么只需要引用SEGGER_SYSVIEW_NoOS.c。

5.3 初始化SystemView

系统信息是由应用程序发送的。这个信息可以通过SEGGER_SYSVIEW_Config_[SYSTEM]. c中的定义来配置。我们可以在main函数中添加一个调用SEGGER_SYSVIEW_Conf()来初始化SystemView。

```
#include "SEGGER_SYSVIEW.h"
/*********************************************************************
*
* main()
*
* Function description
* Application entry point
*/
    int main(void) {
    OS_IncDI(); /* Initially disable interrupts */
    OS_InitKern(); /* Initialize OS */
    OS_InitHW(); /* Initialize Hardware for OS */
    BSP_Init(); /* Initialize BSP module */


    /* You need to create at least one task before calling OS_Start() */
    OS_CREATETASK(&TCB0, "MainTask", MainTask, 100, Stack0);
    OS_Start(); /* Start multitasking */
    return 0;
}
```

SEGGER SystemView的通用部分现在已经准备好监视应用程序了。
当使用启用了分析功能的embOS V4.12或更高版本时，将生成中断、任务和API调用这四种SystemView事件。当不使用embOS时，应用程序必须产生适当的事件。

将应用程序下载到目标板并运行。只要SystemView程序没有连接，并且不调用SEGGER_SYSVIEW_Start()，应用程序就不会生成SystemView事件。当连接了SystemView程序或者调用了SEGGER_SYSVIEW_Start时，它将激活记录SystemView事件。

5.4 启动和停止录制

当使用SystemView连续读取数据时，SystemView自动开始和停止记录。当SystemView没有录制时，目标系统不会生成SystemView事件。
对于single-shot录制，在应用程序中必须调用SEGGER_SYSVIEW_Start，用来激活录制SystemView事件。事件会一直被记录，直到SystemView缓冲区被填满或调用了SEGGER_SYSVIEW_Stop。
对于事后分析，在应用程序中必须调用SEGGER_SYSVIEW_Start，用来激活录制SystemView事件。事件会一直被记录，直到调用了SEGGER_SYSVIEW_Stop。当SystemView缓冲区被填满时，老的事件将被覆盖。

5.5 SystemView系统信息配置**

SEGGER_SYSVIEW_Config_[SYSTEM].c文件提供了SystemView的配置。大多数情况下，不需要修改就可以使用。

```
/*********************************************************************
*                SEGGER Microcontroller GmbH & Co. KG                *
*                        The Embedded Experts                        *
**********************************************************************
*                                                                    *
*       (c) 2015 - 2017  SEGGER Microcontroller GmbH & Co. KG        *
```

```
*                                                                    *
*       www.segger.com     Support: support@segger.com               *
*                                                                    *
**********************************************************************
*                                                                    *
*       SEGGER SystemView * Real-time application analysis           *
*                                                                    *
**********************************************************************
*                                                                    *
*       SystemView version: V2.52a                                    *
*                                                                    *
**********************************************************************
---------------------------- END-OF-HEADER -----------------------------

File    : SEGGER_SYSVIEW_Config_embOS.c
Purpose : Sample setup configuration of SystemView with embOS.
Revision: $Rev: 7750 $
*/
#include "RTOS.h"
#include "SEGGER_SYSVIEW.h"
#include "SEGGER_SYSVIEW_Conf.h"
#include "SEGGER_SYSVIEW_embOS.h"

//
// SystemcoreClock can be used in most CMSIS compatible projects.
// In non-CMSIS projects define SYSVIEW_CPU_FREQ.
//
extern unsigned int SystemCoreClock;

/*********************************************************************
*
*       Defines, configurable
*
**********************************************************************
*/
// The application name to be displayed in SystemViewer
#ifndef   SYSVIEW_APP_NAME
  #define SYSVIEW_APP_NAME        "Demo Application"
#endif

// The target device name
#ifndef   SYSVIEW_DEVICE_NAME
  #define SYSVIEW_DEVICE_NAME     "Cortex-M4"
#endif

// Frequency of the timestamp. Must match SEGGER_SYSVIEW_Conf.h
#ifndef   SYSVIEW_TIMESTAMP_FREQ
  #define SYSVIEW_TIMESTAMP_FREQ  (SystemCoreClock)
#endif

// System Frequency. SystemcoreClock is used in most CMSIS compatible
projects.
#ifndef   SYSVIEW_CPU_FREQ
  #define SYSVIEW_CPU_FREQ        (SystemCoreClock)
#endif
```

```
61
62  // The lowest RAM address used for IDs (pointers)
63  #ifndef   SYSVIEW_RAM_BASE
64    #define SYSVIEW_RAM_BASE        (0x20000000)
65  #endif
66
67  #ifndef   SYSVIEW_SYSDESC0
68    #define SYSVIEW_SYSDESC0        "I#15=SysTick"
69  #endif
70
71  // Define as 1 if the Cortex-M cycle counter is used as SystemView
    timestamp. Must match SEGGER_SYSVIEW_Conf.h
72  #ifndef   USE_CYCCNT_TIMESTAMP
73    #define USE_CYCCNT_TIMESTAMP    1
74  #endif
75
76  // Define as 1 if the Cortex-M cycle counter is used and there might be no
    debugger attached while recording.
77  #ifndef   ENABLE_DWT_CYCCNT
78    #define ENABLE_DWT_CYCCNT       (USE_CYCCNT_TIMESTAMP &
    SEGGER_SYSVIEW_POST_MORTEM_MODE)
79  #endif
80
81  //#ifndef   SYSVIEW_SYSDESC1
82  //  #define SYSVIEW_SYSDESC1      ""
83  //#endif
84
85  //#ifndef   SYSVIEW_SYSDESC2
86  //  #define SYSVIEW_SYSDESC2      ""
87  //#endif
88
89  /*********************************************************************
90  *
91  *       Defines, fixed
92  *
93  **********************************************************************
94  */
95  #define DEMCR                     (*(volatile OS_U32*) (0xE000EDFCuL))  //
    Debug Exception and Monitor Control Register
96  #define TRACEENA_BIT              (1uL << 24)                           //
    Trace enable bit
97  #define DWT_CTRL                  (*(volatile OS_U32*) (0xE0001000uL))  //
    DWT Control Register
98  #define NOCYCCNT_BIT              (1uL << 25)                           //
    Cycle counter support bit
99  #define CYCCNTENA_BIT             (1uL << 0)                            //
    Cycle counter enable bit
100
101 /*********************************************************************
102 *
103 *       _cbSendSystemDesc()
104 *
105 *  Function description
106 *    Sends SystemView description strings.
107 */
```

```
static void _cbSendSystemDesc(void) {
  SEGGER_SYSVIEW_SendSysDesc("N=" SYSVIEW_APP_NAME ",O=embOS,D="
SYSVIEW_DEVICE_NAME);
#ifdef SYSVIEW_SYSDESC0
  SEGGER_SYSVIEW_SendSysDesc(SYSVIEW_SYSDESC0);
#endif
#ifdef SYSVIEW_SYSDESC1
  SEGGER_SYSVIEW_SendSysDesc(SYSVIEW_SYSDESC1);
#endif
#ifdef SYSVIEW_SYSDESC2
  SEGGER_SYSVIEW_SendSysDesc(SYSVIEW_SYSDESC2);
#endif
}

/*********************************************************************
*
*       Global functions
*
**********************************************************************
*/
void SEGGER_SYSVIEW_Conf(void) {
#if USE_CYCCNT_TIMESTAMP
#if ENABLE_DWT_CYCCNT
  //
  // If no debugger is connected, the DWT must be enabled by the
application
  //
  if ((DEMCR & TRACEENA_BIT) == 0) {
    DEMCR |= TRACEENA_BIT;
  }
#endif
  //
  //  The cycle counter must be activated in order
  //  to use time related functions.
  //
  if ((DWT_CTRL & NOCYCCNT_BIT) == 0) {       // Cycle counter supported?
    if ((DWT_CTRL & CYCCNTENA_BIT) == 0) {    // Cycle counter not enabled?
      DWT_CTRL |= CYCCNTENA_BIT;              // Enable Cycle counter
    }
  }
#endif
  SEGGER_SYSVIEW_Init(SYSVIEW_TIMESTAMP_FREQ, SYSVIEW_CPU_FREQ,
                      &SYSVIEW_X_OS_TraceAPI, _cbSendSystemDesc);
  SEGGER_SYSVIEW_SetRAMBase(SYSVIEW_RAM_BASE);
  OS_SetTraceAPI(&embOS_TraceAPI_SYSVIEW);    // Configure embOS to use
SYSVIEW.
}

/************************** End of file ****************************/
```

章节6 目标板配置 - Segger SystemView使用手册（译文）

章节6 目标板配置

为了匹配目标设备和应用程序，可以对SEGGER SystemView进行配置。默认的编译配置标志都已经预先配置了一个有效值，用来匹配大多数系统的需求。通常不需要修改。
可以通过更改编译Flags来修改SystemView的默认配置，这些Flags可以添加在SEGGER_SYSVIEW_Conf.h文件中。

6.1 系统特定配置

为了匹配目标系统，需要进行以下编译配置。SEGGER_SYSVIEW_Conf.h中的示例配置定义了配置能够匹配大多数系统(例如，带有Embedded Studio、GCC、IAR或Keil ARM的Cortex-M芯片设备)。如果使用的系统不在示例配置中，则必须进行相应地调整。

有关系统特定配置的详细描述，请参阅第69页的支持的cpu。

6.1.1 SEGGER_SYSVIEW_GET_TIMESTAMP()

获取用于SystemView事件的系统时间戳的函数宏定义。
在Cortex-M3/M4芯片中，可以将Cortex-M核的循环计数器用于系统时间戳。
在Cortex-M3/M4中：地址为(∗U32∗)(0xE0001004))(∗U32∗)(0xE0001004))

在其他大多数芯片中，系统时间戳一般用定时器产生。在默认配置下，系统时间戳可以使用用户提供的函数SEGGER_SYSVIEW_X_GetTimestamp()来获取。
其他核心中：地址为SEGGER_SYSVIEW_X_GetTimestamp()

参考SEGGER_SYSVIEW_Config_embOS_CM0.c 或者 SEGGER_SYSVIEW_Config_NoOS_RX.c 中的例子。

> **注意**
>
> 在SEGGER_SYSVIEW_Init的参数中，必须提供系统的时间戳频率。

6.1.2 SEGGER_SYSVIEW_TIMESTAMP_BITS

作为系统时间戳的时钟源的低位有效bit位数。
如果使用未修改的时钟源作为系统时间戳，有效位的数量是时钟源的位宽(即32或16位)。
默认值:32(使用32位时钟源)

节省带宽的例子
由于SystemView包使用了可变长度的编码，所以较少时间戳位数可以节省缓冲空间和带宽。

就像在Cortex-M3/4芯片上的Cortex-M的循环计数器，一个32位时钟源可以移动4位，使得只有28位有效的时间戳位，而在SEGGER_SYSVIEW_Init使用的时间戳频率，是核心时钟频率除以16。

#define SEGGER_SYSVIEW_GET_TIMESTAMP() ((*(U32 *)(0xE0001004)) >> 4)
#define SEGGER_SYSVIEW_TIMESTAMP_BITS 28.

6.1.3 SEGGER_SYSVIEW_GET_INTERRUPT_ID

用于定义获取当前活动中断ID的宏定义函数。
在Cortex-M芯片中，可以从ICSR中读到激活的向量表。
在Cortex-M3/4芯片上的默认值： ((*(U32 *)(0xE000ED04)) & 0x1FF)
在Cortex-M0/1芯片上的默认值： (((U32 *)(0xE000ED04)) & 0x3F)

在其他设备上，活动中断可以直接从中断控制器中获取，可以保存在通用中断处理函数中的变量中，或者必须在每个中断函数中手动指定。
默认情况下，可以使用用户提供的函数SEGGER_SYSVIEW_X_GetInterruptId()来指定，或者直接替换宏定义。

6.1.4 SEGGER_SYSVIEW_LOCK()

用于锁定SystemView，防止SystemView传输被打断。比如防止中断。
SEGGER_SYSVIEW_LOCK()必须保存前一个锁状态，并在SEGGER_SYSVIEW_UNLOCK()中恢复锁状态。

记录一个SystemView事件不能被记录另一个事件打断。因此，所有被SystemView记录的中断中(调用SEGGER_SYSVIEW_RecordEnterISR / SEGGER_SYSVIEW_RecordExitISR)，调用一个instrumented的函数(比如一个OS API函数)，会立即引起一个上下文切换，或者可能创建其他的SystemView事件，因此这种操作是必须禁止的。
SEGGER_SYSVIEW_LOCK()可以使用与SEGGER_RTT_LOCK()相同的锁定机制。
默认值：SEGGER_RTT_LOCK()
SEGGER_RTT_LOCK()在SEGGER_RTT_Conf.h中为大多数系统(也就是带有嵌入式Studio、GCC、IAR或Keil ARM和RX设备的cortexm设备)定义。如果宏没有定义或空，则必须提供它以匹配目标系统。

6.1.5 SEGGER_SYSVIEW_UNLOCK()

用于解锁被中断的SystemView传输，会恢复前一个中断状态。

SEGGER_SYSVIEW_UNLOCK()函数可以和SEGGER_RTT_UNLOCK一样的用法。
默认值：SEGGER_RTT_UNLOCK()

SEGGER_RTT_UNLOCK() 在SEGGER_RTT_Conf.h中已经为适配大多数系统作了定义(带有嵌入式Studio、GCC、IAR或Keil ARM和使用IAR的RX芯片的Cortex-M核芯片)。如果没有定义这个宏或宏定义为空，则必须提供适合目标系统的定义。

6.2 通用配置

下面的编译标志主要用于优化或者更改SystemView事件录制的方式。
默认的编译配置标志已经预先配置了匹配大多数系统的需求的有效值，通常不需要修改。

6.2.1 SEGGER_SYSVIEW_RTT_BUFFER_SIZE

SystemView用于记录缓冲区的字节数。
在大多数情况下，对于连续记录，1024字节的缓冲区就足够了。根据目标接口速度、目标速度和系统负载，缓冲区大小可能要增加到4096字节。
对于单次记录，缓冲区大小决定了可以记录的事件数。根据负载的不同，系统可能会产生10到50个kByte / s。推荐一个至少8 kByte的缓冲区，到整个空闲的RAM空间。缓冲区也可以在外部RAM中。
对于死后分析，缓冲区大小决定了可供分析的事件的最大数目。推荐的缓冲区大小和单次记录一样。
默认值:1024字节

SEGGER_SYSVIEW_RTT_CHANNEL

RTT通道是用于SystemView事件记录和通信。0表示自动选择

> 注意：
>
> 在SEGGER_RTT_Conf.h中定义的SEGGER_RTT_MAX_NUM_UP_BUFFERS的值不能大于SEGGER_SYSVIEW_RTT_CHANNEL。
>
> 默认值：0

6.2.3 SEGGER_SYSVIEW_USE_STATIC_BUFFER

如果这个值设为1，SystemView将使用静态缓冲区创建SystemView事件。这样通常能够节省空间，因为只有一个缓冲区是必需的，而任务栈可以尽可能小。当使用静态缓冲区时，SystemView锁调用之间执行的关键代码需要稍微长一点时间。
如果设为0，SystemView事件是创建在堆栈上 。确保所有的任务栈，以及用于中断的C堆栈都足够大，可以容纳最大的SystemView事件(~ 228字节)。只有在将堆栈缓冲区传输到RTT缓冲区时，才可以使用SystemView锁。

默认值：1

6.2.4 SEGGER_SYSVIEW_POST_MORTEM_MODE

设为1，可以使能死后分析模式。
在事后分析模式中，系统使用一个循环缓冲区，这样可以让所有事件保持最新的记录，而不是缓冲区满时就删除事件。（*译者注：存在疑问，与之前的意思表达冲突*）

> 注意
>
> 当使用J-Link调试器主动读取RTT数据时，不要使用事后分析模式。
> 默认值：0

6.2.5 SEGGER_SYSVIEW_SYNC_PERIOD_SHIFT

配置在死后模式中发送同步和系统信息事件的频率。确保在SystemView缓冲区中至少有一个同步。
推荐的同步频率： Buffer Size / 16
默认值：8 = 每256个数据包同步一次

6.2.6 SEGGER_SYSVIEW_ID_BASE

从SystemView包中记录的ID中减去的值。
ID是任务ID，定时器ID或者资源ID，它们通常指向RAM中的结构。在操作系统和中间件API事件中发送的参数也可以通过instrumentation来编码为id。

> 注意
>
> 如果操作系统没有指向任务ID，定时器ID或者资源ID的话，SEGGER_SYSVIEW_ID_BASE必须设为0
> 当SystemView包使用一个可变长度的指针编码时，正确地重新建立地址可以节省缓冲空间和带宽。
> 定义系统中使用的最低内存地址。
> 应用程序通过SEGGER_SYSVIEW_SetRAMBase可以覆盖这个值。
> 在不确定的情况下，将SEGGER_SYSVIEW_ID_BASE设为0
> 默认值：0x1000 0000

6.2.7 SEGGER_SYSVIEW_ID_SHIFT

在SystemView包中记录的ID偏移的位数。

> 注意
>
> 如果操作系统没有指向任务ID，定时器ID或者资源ID的话，SEGGER_SYSVIEW_ID_SHIFT必须设为0
> 当SystemView包使用一个可变长度的指针编码时，正确地偏移地址可以节省缓冲空间和带宽。
> 对于32位处理器上的大多数应用程序，SystemView事件中记录的所有id都是真正的指针，并且是4的倍数，因此最低的2位可以被安全地忽略。
> 在不确定的情况下，把SEGGER_SYSVIEW_ID_SHIFT设为0
> 默认值：2

SEGGER_SYSVIEW_MAX_STRING_LEN

SystemView事件记录的最大字符串长度。
字符串被用于SystemView printf样式的用户函数，以及SEGGER_SYSVIEW_SendSysDesc和SEGGER_SYSVIEW_RecordModuleDescription。确保SEGGER_SYSVIEW_MAX_STRING_LEN匹配这些函数中使用的字符串长度。
最大支持的字符串长度为255字节
默认值：128

6.2.9 SEGGER_SYSVIEW_MAX_ARGUMENTS

使用SEGGER_SYSVIEW_PrintfHost、SEGGER_SYSVIEW_PrintfHostEx、SEGGER_SYSVIEW_WarnfHost SEGGER_SYSVIEW_ErrorfHost时发送参数的最大数量。
如果在应用程序中没有使用这些函数，则SEGGER_SYSVIEW_MAX_ARGUMENTS可以设置为0，使静态缓冲区最小。
默认值:16

6.2.10 SEGGER_SYSVIEW_BUFFER_SECTION

可以将SystemView RTT缓冲区放入专用的数据段，而不是默认的数据段。这允许将缓冲区放入外部内存或在指定的地址。
当定义SEGGER_SYSVIEW_BUFFER_SECTION时，该部分必须在链接器脚本中定义。
默认:SEGGER_RTT_SECTION或未定义
在Embedded Studio中的例子

```
//
// SEGGER_SYSVIEW_Conf.h
//
#define SEGGER_SYSVIEW_BUFFER_SECTION "SYSTEMVIEW_RAM"
//
// flash_placement.xml
//
<MemorySegment name="ExtRAM">
<ProgramSection load="No" name="SYSTEMVIEW_RAM" start="0x40000000" />
</MemorySegment>12345678910
```

6.2.11 RTT配置

下面的编译标志是用于优化或者修改RTT模块的。
大多数情况下不需要修改这些标志。

6.2.11.1 BUFFER_SIZE_UP

用于RTT终端输出通道的字节数
RTT可用于printf终端输出，无需修改。BUFFER_SIZE_UP定义了可以缓冲多少字节。
如果不使用RTT终端输出，则将BUFFER_SIZE_UP定义为最小值4。
默认值:1024字节

6.2.11.2 BUFFER_SIZE_DOWN

用于RTT终端输入通道的字节数。
RTT可以在终端输入通道接收来自主机的输入。BUFFER_SIZE_DOWN定义了从主机发送一次可以缓冲多少字节。
如果不使用RTT终端输入，则将BUFFER_SIZE_DOWN定义为它的最小值为4。
默认值:16字节

6.2.11.3 SEGGER_RTT_MAX_NUM_UP_BUFFERS

最大RTT输出缓冲区的数量。缓冲区0总是用于RTT终端输出，因此要使用SystemView，SEGGER_RTT_MAX_NUM_UP_BUFFERS必须至少有2个。
默认值:2

6.2.11.4 SEGGER_RTT_MAX_NUM_DOWN_BUFFERS

最大RTT输入缓冲区的数量。缓冲区0总是用于RTT终端输入，因此要使用SystemView，SEGGER_RTT_MAX_NUM_UP_BUFFERS必须至少有2个。(*译者注：此处是否应该是DOWN_BUFFERS至少有2个*)
默认值:2

6.2.11.5 SEGGER_RTT_MODE_DEFAULT

预初始化RTT终端通道的模式(缓冲区0)
默认值：SEGGER_RTT_MODE_NO_BLOCK_SKIP

6.2.11.6 SEGGER_RTT_PRINTF_BUFFER_SIZE

通过RTT printf发送字符块时的缓冲区大小。如果不用SEGGER_RTT_Printf时，可以定义为0。
默认值：64

6.2.11.7 SEGGER_RTT_SECTION

RTT控制块可以被放置到专门的部分，而不是默认的数据部分。这允许将其放置在已知地址，以便使用J-Link能够自动检测或容易指定搜索范围。
如果定义了SEGGER_RTT_SECTION，应用程序必须确保该部分是有效的，要么是在启动代码中初始化为0，要么在应用程序开始时显式地调用SEGGER_RTT_Init()。SEGGER_RTT_Init()是由SEGGER_SYSVIEW_Init()隐式调用的。
默认值:未定义

6.2.11.8 SEGGER_RTT_BUFFER_SECTION

RTT终端缓冲区放置的位置，最好放置在专门的部分，而不是放到默认数据段。允许放到外部RAM或者一个给定的地址。
默认值：SEGGER_RTT_SECTION 或者不定义

6.3 优化SystemView

为了从目标系统获得最精确的运行时信息，记录工具代码需要快速、最少侵入性、小且高效。
SystemView代码被编写为高效且不具有干扰性。SystemView的速度和大小是目标和编译器配置的问题。下面的部分描述如何优化SystemView。

6.3.1 编译器优化

SystemView目标系统模块的编译器优化应该始终打开（即使是在调试版本中），用于生成快速的记录路径，从而减少开销，并减少对应用程序的干扰。
是否支持速度或大小优化的配置依赖于编译器的。在某些情况下，平衡的配置可能比只优化速度的配置更快。

6.3.2 录制优化

SystemView使用一个可变长度的编码来存储和传输事件，这使得在调试接口上节省了缓冲空间和带宽。一些事件参数的大小可以通过编译时配置进行优化。
缩小id位数
ID是指向RAM中的符号的指针，例如任务ID是指向任务控制块的指针。为了尽量减少记录的id长度，它们可以缩小。
可以从指针中减去SEGGER_SYSVIEW_ID_BASE从而得到ID。它可以从指针中减去基RAM地址，而指针仍然是唯一的，但是变成了更小的ID。例如，如果RAM范围是0x20000000到0x20001000，那么建议将SEGGER_SYSVIEW_ID_BASE定义为0x20000000，它将使得指针0x20000100变为ID 0x100，这样只需要两个而不是四个位来存储它。
SEGGER_SYSVIEW_ID_SHIFT是一个指针通过右移多少位数得到ID。如果所有的记录指针都是4字节对齐，那么SEGGER_SYSVIEW_ID_SHIFT可以被定义为2。然后，一个指针0x20000100右移2位将会有ID 0x8000040，或者与之前的SEGGER_SYSVIEW_ID_BASE的减法一样，这个ID为0x20000000,ID为0x40，只需要记录一个字节。

时间戳的来源

SystemView中的事件时间戳被记录为时间戳与前一个事件的差值。这节省了缓冲空间。
当建议使用时间戳源使用最高时间分辨率的CPU时钟频率时，较低的时间戳频率可能会节省额外的缓冲空间，因为时间戳增量较低（*时间间隔会比较低，导致需要保存的时间位数变短*）。
使用CPU时钟频率为160 MHz时，时间戳会右移到4位，导致一个时间戳10 MHz的频率(100 ns决议),少和4位编码。
当时间戳大小不再是32 -比特时，例如它在0xFFFFFFFF之前结束，SEGGER_SYSVIEW_TIMESTAMP_BITS必须被定义为时间戳大小，例如为28时将一个32位的时间戳右移4位。

6.3.3 缓冲区配置

SystemView和RTT的记录和通信缓冲区大小可以设置在目标配置中。
在大多数情况下,连续记录1-4k字节用于连续记录是足够的，另外甚至允许SystemView使用一个小的内部RAM。
对于单次和死后模式，建议采用更大的缓冲区。在这种情况下，SEGGER_SYSVIEW_RTT_BUFFER_SIZE可以设置为更大的值。为了将SystemView记录缓冲区放入外部RAM中，可以定义SEGGER_SYSVIEW_BUFFER_SECTION，并相应地修改链接器脚本。

如果只使用SystemView，且不使用RTT的终端输出，则在SEGGER_RTT_Conf.h中使用的BUFFER_SIZE_UP。可以设置为较小的值以节省内存。

章节7 支持的CPU - Segger SystemView使用手册（译文）

章节7 支持的CPU

本节描述如果设置和配置不同CPU的SystemView模块。
SEGGER SystemView几乎支持任何目标CPU，然而，只有CPU支持后台内存访问（ARM cortexm和Renesas RX）时，才支持连续录制。
在其他CPU中，SystemView可以用单次或死后分析模式。请参阅第46页的单次录制。
为了使SystemView正常运行，需要完成一些特定于目标的配置。
下面的一些cpu描述了这个配置。

7.1 Cortex-M3/ Cortex-M4

录制模式	是否支持
连续模式	支持
单次模式	支持
死后模式	支持

7.1.1 事件时间戳

在Cortex-M3/Cortex-M4上，时间戳源可以是一个循环计数器，可以保证循环准确的事件记录。
在记录事件时，为了节省带宽，循环计数器可以选择右移，即4位，以使核心速度的时间戳频率除以16。

配置

```
//
// Use full cycle counter for higher precision
//
#define SEGGER_SYSVIEW_GET_TIMESTAMP() (*(U32 *)(0xE0001004))
#define SEGGER_SYSVIEW_TIMESTAMP_BITS (32)
//
// Use cycle counter divided by 16 for smaller size / bandwidth
//
#define SEGGER_SYSVIEW_GET_TIMESTAMP() ((*(U32 *)(0xE0001004)) >> 4)
#define SEGGER_SYSVIEW_TIMESTAMP_BITS (28)
```

7.1.2 中断ID

当前活动的中断可以直接通过读取Cortex0-M ICSR[8:0]来识别，这是中断控制器状态寄存器(ICSR)中的向量表区域。

配置

```
//
// Get the interrupt Id by reading the Cortex-M ICSR[8:0]
//
#define SEGGER_SYSVIEW_GET_INTERRUPT_ID() ((*(U32 *)(0xE000ED04)) & 0x1FF)
```

7.1.3 SystemView锁定和解锁

锁定和解锁可以通过禁止中断来防止传输中的记录被打断。在Cortex-M3/Cortex-M4中，不是所有的中断都需要被禁止，只有那些本身可能产生SystemView事件或者在OS中导致任务切换的中断。
默认情况下，优先级掩码设置为32，已使用一个优先级为32或更低(更高的数值)来禁用所有中断。
确保屏蔽所有可以发送RTT数据的中断，例如生成SystemView事件，或导致任务切换。当在发送RTT数据时，当高优先级中断不能被掩盖时，SEGGER_RTT_MAX_INTERRUPT_PRIORITY需要相应地调整。(高优先级=低优先级)栓塞的默认值:128u
FreeRTOS的缺省配置:configMAX_SYSCALL_INTERRUPT_PRIORITY:
(configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY < <(8 - configPRIO_BITS))
如有疑问，请禁用所有中断

配置

```
//
// RTT locking for GCC toolchains in SEGGER_RTT_Conf.h
//
#define SEGGER_RTT_LOCK() {                             \
                          unsigned int LockState; \
                          __asm volatile ("mrs %0, basepri \n\t" \
                          "mov r1, $32 \n\t" \
                          "msr basepri, r1 \n\t" \
                          : "=r" (LockState) \
                          : \
                          : "r1" \
                          );
#define SEGGER_RTT_UNLOCK() __asm volatile ("msr basepri, %0 \n\t" \
                          : \
                          : "r" (LockState) \
                          : \
                          ); \
                          }
//
// Define SystemView locking in SEGGER_SYSVIEW_Conf.h
//
#define SEGGER_SYSVIEW_LOCK() SEGGER_RTT_LOCK()
#define SEGGER_SYSVIEW_UNLOCK() SEGGER_RTT_UNLOCK()
```

7.1.4 示例配置

SEGGER_SYSVIEW_Conf.h

```
/*********************************************************************
* (c) SEGGER Microcontroller GmbH & Co. KG *
**********************************************************************
------------------------- END-OF-HEADER -----------------------------
File : SEGGER_SYSVIEW_Conf.h
Purpose : SEGGER SysView configuration for Cortex-M3 / Cortex-M4.
*/
#ifndef SEGGER_SYSVIEW_CONF_H
#define SEGGER_SYSVIEW_CONF_H
/*********************************************************************
*
* SysView timestamp configuration
*/
// Cortex-M cycle counter.
#define SEGGER_SYSVIEW_GET_TIMESTAMP() ((*(U32 *)(0xE0001004)))
// Number of valid bits low-order delivered as timestamp.
#define SEGGER_SYSVIEW_TIMESTAMP_BITS 32
/*********************************************************************
*
* SysView Id configuration
*/
// Default value for the lowest Id reported by the application.
// Can be overridden by the application via SEGGER_SYSVIEW_SetRAMBase().
#define SEGGER_SYSVIEW_ID_BASE 0x20000000
// Number of bits to shift the Id to save bandwidth.
// (i.e. 2 when all reported Ids (pointers) are 4 byte aligned)
#define SEGGER_SYSVIEW_ID_SHIFT 0
/*********************************************************************
*
* SysView interrupt configuration
*/
// Get the currently active interrupt Id. (read Cortex-M ICSR[8:0]
= active vector)
#define SEGGER_SYSVIEW_GET_INTERRUPT_ID() ((*(U32 *)(0xE000ED04)) & 0x1FF)
/*********************************************************************
*
* SysView locking
*/
// Lock SysView (nestable)
#define SEGGER_SYSVIEW_LOCK() SEGGER_RTT_LOCK()
// Unlock SysView (nestable)
#define SEGGER_SYSVIEW_UNLOCK() SEGGER_RTT_UNLOCK()
#endif
/*************************** End of file ****************************/
```

SEGGER_SYSVIEW_Config_NoOS_CM3.c

```
/*********************************************************************
* (c) SEGGER Microcontroller GmbH & Co. KG *
* The Embedded Experts *
```

```
* www.segger.com *
**********************************************************************
---------------------- END-OF-HEADER -----------------------------
File : SEGGER_SYSVIEW_Config_NoOS.c
Purpose : Sample setup configuration of SystemView without an OS.
Revision: $Rev: 7745 $
*/
#include "SEGGER_SYSVIEW.h"
#include "SEGGER_SYSVIEW_Conf.h"
// SystemcoreClock can be used in most CMSIS compatible projects.
// In non-CMSIS projects define SYSVIEW_CPU_FREQ.
extern unsigned int SystemCoreClock;
/*********************************************************************
*
* Defines, configurable
*
**********************************************************************
*/
// The application name to be displayed in SystemViewer
#define SYSVIEW_APP_NAME "Demo Application"
// The target device name
#define SYSVIEW_DEVICE_NAME "Cortex-M4"
// Frequency of the timestamp. Must match SEGGER_SYSVIEW_Conf.h
#define SYSVIEW_TIMESTAMP_FREQ (SystemCoreClock)
// System Frequency. SystemcoreClock is used in most CMSIS compatible
projects.
#define SYSVIEW_CPU_FREQ (SystemCoreClock)
// The lowest RAM address used for IDs (pointers)
#define SYSVIEW_RAM_BASE (0x10000000)
// Define as
1 if the Cortex-M cycle counter is used as SystemView timestamp. Must match
SEGGER_SYSVIEW_Conf.h
#ifndef USE_CYCCNT_TIMESTAMP
    #define USE_CYCCNT_TIMESTAMP 1
#endif
// Define as
1 if the Cortex-M cycle counter is used and there might be no debugger
attached while re
#ifndef ENABLE_DWT_CYCCNT
    #define ENABLE_DWT_CYCCNT    (USE_CYCCNT_TIMESTAMP &
SEGGER_SYSVIEW_POST_MORTEM_MODE)
#endif
/*********************************************************************
*
* Defines, fixed
*
**********************************************************************
*/
#define DEMCR (*(volatile unsigned long*) (0xE000EDFCuL))
// Debug Exception and Monitor Control Register
#define TRACEENA_BIT (1uL << 24)
// Trace enable bit
#define DWT_CTRL (*(volatile unsigned long*) (0xE0001000uL))
// DWT Control Register
#define NOCYCCNT_BIT (1uL << 25)
```

```
// Cycle counter support bit
#define CYCCNTENA_BIT (1uL << 0)
// Cycle counter enable bit
/*********************************************************************
*
* _cbSendSystemDesc()
*
* Function description
* Sends SystemView description strings.
*/
static void _cbSendSystemDesc(void) {

 SEGGER_SYSVIEW_SendSysDesc("N="SYSVIEW_APP_NAME",D="SYSVIEW_DEVICE_NAME);
    SEGGER_SYSVIEW_SendSysDesc("I#15=SysTick");
}
/*********************************************************************
*
* Global functions
*
**********************************************************************
*/
void SEGGER_SYSVIEW_Conf(void) {
#if USE_CYCCNT_TIMESTAMP
#if ENABLE_DWT_CYCCNT
    //
    // If no debugger is connected, the DWT must be enabled by the
application
    //
    if ((DEMCR & TRACEENA_BIT) == 0) {
        DEMCR |= TRACEENA_BIT;
    }
#endif
    //
    // The cycle counter must be activated in order
    // to use time related functions.
    //
    if ((DWT_CTRL & NOCYCCNT_BIT) == 0) { // Cycle counter supported?
        if ((DWT_CTRL & CYCCNTENA_BIT) == 0) { // Cycle counter not enabled?
            DWT_CTRL |= CYCCNTENA_BIT; // Enable Cycle counter
        }
    }
#endif
    SEGGER_SYSVIEW_Init(SYSVIEW_TIMESTAMP_FREQ,SYSVIEW_CPU_FREQ,0,
_cbSendSystemDesc);
    SEGGER_SYSVIEW_SetRAMBase(SYSVIEW_RAM_BASE);
}
/*************************** End of file ****************************/
```

7.2 Cortex-M7

与Cortex-M4相同的特性/设置等。要了解更多信息，请参阅第70页的Cortex-M3 / Cortex-M4。

缓存Cache

当将RTT缓冲区设置为可缓存（cacheable）的SystemView时，通过J-Link和RTT进行连续记录模式的性能略低(性能下降了小于1%)。这是因为J-Link需要在访问RTT缓冲区时执行缓存维护操作。

7.3 Cortex-M0 / Cortex-M0+ /Cortex-M1

录制模式	是否支持
连续模式	支持
单次模式	支持
死后分析	支持

7.3.1 Cortex-M0事件时间戳

Cortex-M0、Cortex-M0 +和Cortex-M1没有循环计数寄存器。事件时间戳必须由应用程序时钟源(例如系统定时器，SysTick)提供。可以使用SEGGER_SYSVIEW_X_GetTimestamp()来实现该功能。
当应用程序使用SysTick中断时，如RTOS，SysTick处理程序应该增加SEGGER_SYSVIEW_TickCnt，否则必须将SysTick处理程序添加到应用程序并作相应地配置。

配置

```
//
// SEGGER_SYSVIEW_TickCnt has to be defined in the module which
// handles the SysTick and must be incremented in the SysTick
// handler before any SYSVIEW event is generated.
//
// Example in embOS RTOSInit.c:
//
// unsigned int SEGGER_SYSVIEW_TickCnt; // <<-- Define SEGGER_SYSVIEW_TickCnt.
// void SysTick_Handler(void) {
// #if OS_PROFILE
//   SEGGER_SYSVIEW_TickCnt++;  // <<-- Increment SEGGER_SYSVIEW_TickCnt asap.
// #endif
//   OS_EnterNestableInterrupt();
//   OS_TICK_Handle();
//   OS_LeaveNestableInterrupt();
// }
//
extern unsigned int SEGGER_SYSVIEW_TickCnt;

/*********************************************************************
*
*       Defines, fixed
*
**********************************************************************
*/
#define SCB_ICSR
  (*(volatile U32*)  (0xE000ED04uL)) // Interrupt Control State Register
#define SCB_ICSR_PENDSTSET_MASK      (1UL << 26)     // SysTick pending bit
#define SYST_RVR
  (*(volatile U32*)  (0xE000E014uL)) // SysTick Reload Value Register
#define SYST_CVR
  (*(volatile U32*)  (0xE000E018uL)) // SysTick Current Value Register
/*********************************************************************
```

```
*
*       SEGGER_SYSVIEW_X_GetTimestamp()
*
* Function description
*   Returns the current timestamp in ticks using the system tick
*   count and the SysTick counter.
*   All parameters of the SysTick have to be known and are set via
*   configuration defines on top of the file.
*
* Return value
*   The current timestamp.
*
* * Additional information
*   SEGGER_SYSVIEW_X_GetTimestamp is always called when interrupts are
*   disabled. Therefore locking here is not required.
*/
U32 SEGGER_SYSVIEW_X_GetTimestamp (void) {
 U32 TickCount;
 U32 Cycles;
 U32 CyclesPerTick;
 //
 // Get the cycles of the current system tick.
 // SysTick is down-counting, subtract the current value from the number of cycles per
 //
 CyclesPerTick = SYST_RVR + 1;
 Cycles =  (CyclesPerTick - SYST_CVR);
 //
 // Get the system tick count.
 //
 TickCount = SEGGER_SYSVIEW_TickCnt;
 //
 // If a SysTick interrupt is pending, re-read timer and adjust result
 //
 if  ((SCB_ICSR & SCB_ICSR_PENDSTSET_MASK)  != 0) {
   Cycles =  (CyclesPerTick - SYST_CVR);
   TickCount++;
 }
 Cycles += TickCount * CyclesPerTick;

 return Cycles;
}
```

7.3.2 Cortex-M0中断ID

当前活动的中断可以直接通过读取Cortex-M的ICSR[5..0]寄存器获取，ICSR[5..0]是中断控制状态寄存器ICSR的活动向量表区域。

配置

```
//
// Get the interrupt Id by reading the Cortex-M ICSR[5:0]
//
#define SEGGER_SYSVIEW_GET_INTERRUPT_ID() ((*(U32 *)(0xE000ED04)) & 0x3F)1234
```

7.3.3 Cortex-M0 SystemView锁定和解锁

锁定和接收SystemView和RTT是相同的。

配置

```
//
// RTT locking for GCC toolchains in SEGGER_RTT_Conf.h
//
#define SEGGER_RTT_LOCK ()       {
    \
                                 unsigned int LockState;
  \
                                 __asm volatile  ("mrs   %0, primask  \n\t"
    \
                                                  "mov   r1, $1     \n\t"
  \
                                                  "msr   primask, r1  \n\t"
  \
                                                   : "=r"  (LockState)
     \
                                                   :
    \
                                                   : "r1"
   \
                                                  );

#define SEGGER_RTT_UNLOCK ()       __asm volatile  ("msr   primask, %0
\n\t"     \

                                                   :
    \
                                                   : "r"  (LockState)
    \
                                                   :
    \
                                                  );
   \
                                }

//
// Define SystemView locking in SEGGER_SYSVIEW_Conf.h
//
#define SEGGER_SYSVIEW_LOCK ()   SEGGER_RTT_LOCK ()
#define SEGGER_SYSVIEW_UNLOCK () SEGGER_RTT_UNLOCK ()
```

7.3.4 Cortex-M0示例配置

SEGGER_SYSVIEW_Conf.h

```
/*********************************************************************
*               (c) SEGGER Microcontroller GmbH & Co. KG             *
*********************************************************************
```

```
4  ---------------------------- END-OF-HEADER -----------------------------
5
6  File        : SEGGER_SYSVIEW_Conf.h
7  Purpose     : SEGGER SysView configuration for Cortex-M0, Cortex-M0+,
8                and Cortex-M1
9  */
10
11 #ifndef SEGGER_SYSVIEW_CONF_H
12 #define SEGGER_SYSVIEW_CONF_H
13
14 /*********************************************************************
15 *
16 *       SysView timestamp configuration
17 */
18 // Retrieve a system timestamp via user-defined function
19 #define SEGGER_SYSVIEW_GET_TIMESTAMP ()      SEGGER_SYSVIEW_X_GetTimestamp
   ()
20 // number of valid bits low-order delivered by
   SEGGER_SYSVIEW_X_GetTimestamp()
21 #define SEGGER_SYSVIEW_TIMESTAMP_BITS       32
22
23 /*********************************************************************
24 *
25 *       SysView Id configuration
26 */
27 // Default value for the lowest Id reported by the application.
28 // Can be overridden by the application via SEGGER_SYSVIEW_SetRAMBase().
29 #define SEGGER_SYSVIEW_ID_BASE         0x20000000
30 // Number of bits to shift the Id to save bandwidth.
31 // (i.e. 2 when all reported Ids (pointers) are 4 byte aligned)
32 #define SEGGER_SYSVIEW_ID_SHIFT        0
33
34 /*********************************************************************
35 *
36 *       SysView interrupt configuration
37 */
38 // Get the currently active interrupt Id. (read Cortex-M ICSR[8:0]
39 = active vector)
40 #define SEGGER_SYSVIEW_GET_INTERRUPT_ID ()    ((*(U32 *)(0xE000ED04)) &
   0x3F)
41
42 /*********************************************************************
43 *
44 *       SysView locking
45 */
46 // Lock SysView (nestable)
47 #define SEGGER_SYSVIEW_LOCK ()             SEGGER_RTT_LOCK ()
48 // Unlock SysView (nestable)
49 #define SEGGER_SYSVIEW_UNLOCK ()           SEGGER_RTT_UNLOCK ()
50 #endif
51
52 /************************** End of file ****************************/
```

SEGGER_SYSVIEW_Config_embOS_CM0.c

```
/*********************************************************************
*               (c) SEGGER Microcontroller GmbH & Co. KG             *
*                       The Embedded Experts                         *
*                          www.segger.com                            *
**********************************************************************

-------------------------- END-OF-HEADER -----------------------------

File    : SEGGER_SYSVIEW_Config_embOS_CM0.c
Purpose : Sample setup configuration of SystemView with embOS
         on Cortex-M0/Cortex-M0+/Cortex-M1 systems which do not
         have a cycle counter.
Revision: $Rev: 7750 $

Additional information:
 SEGGER_SYSVIEW_TickCnt has to be defined in the module which handles
 the SysTick and must be incremented in the SysTick_Handler.

 This configuration can be adopted for any other OS and device.
*/
#include "RTOS.h"
#include "SEGGER_SYSVIEW.h"
#include "SEGGER_SYSVIEW_embOS.h"

//
// SystemcoreClock can be used in most CMSIS compatible projects.
// In non-CMSIS projects define SYSVIEW_CPU_FREQ directly.
//
extern unsigned int SystemCoreClock;

//
// SEGGER_SYSVIEW_TickCnt has to be defined in the module which
// handles the SysTick and must be incremented in the SysTick
// handler before any SYSVIEW event is generated.
//
// Example in embOS RTOSInit.c:
//
// unsigned int SEGGER_SYSVIEW_TickCnt; // <<-- Define
SEGGER_SYSVIEW_TickCnt.
// void SysTick_Handler(void) {
// #if OS_PROFILE
//   SEGGER_SYSVIEW_TickCnt++;                  //
<<-- Increment SEGGER_SYSVIEW_TickCnt before calling
OS_EnterNestableInterrupt.
// #endif
//   OS_EnterNestableInterrupt();
//   OS_TICK_Handle();
//   OS_LeaveNestableInterrupt();
// }
//
extern unsigned int SEGGER_SYSVIEW_TickCnt;

/*********************************************************************
*
*       Defines, fixed
```

```
*
**********************************************************************
*/
#define SCB_ICSR
  (*(volatile U32*)  (0xE000ED04uL)) // Interrupt Control State Register
#define SCB_ICSR_PENDSTSET_MASK      (1UL << 26)     // SysTick pending bit
#define SYST_RVR
  (*(volatile U32*)  (0xE000E014uL)) // SysTick Reload Value Register
#define SYST_CVR
  (*(volatile U32*)  (0xE000E018uL)) // SysTick Current Value Register

/*********************************************************************
*
*       Defines, configurable
*
**********************************************************************
*/
// The application name to be displayed in SystemViewer
#ifndef   SYSVIEW_APP_NAME
 #define SYSVIEW_APP_NAME          "Demo Application"
#endif

// The target device name
#ifndef   SYSVIEW_DEVICE_NAME
 #define SYSVIEW_DEVICE_NAME       "Cortex-M0"
#endif

// Frequency of the timestamp. Must match SEGGER_SYSVIEW_Conf.h
#ifndef   SYSVIEW_TIMESTAMP_FREQ
 #define SYSVIEW_TIMESTAMP_FREQ   (SystemCoreClock)
#endif

// System Frequency. SystemcoreClock is used in most CMSIS compatible
projects.
#ifndef   SYSVIEW_CPU_FREQ
 #define SYSVIEW_CPU_FREQ         (SystemCoreClock)
#endif

// The lowest RAM address used for IDs (pointers)
#ifndef   SYSVIEW_RAM_BASE
 #define SYSVIEW_RAM_BASE         (0x20000000)
#endif

#ifndef   SYSVIEW_SYSDESC0
 #define SYSVIEW_SYSDESC0        "I#15=SysTick"
#endif

//#ifndef   SYSVIEW_SYSDESC1
//  #define SYSVIEW_SYSDESC1      ""
//#endif

//#ifndef   SYSVIEW_SYSDESC2
//  #define SYSVIEW_SYSDESC2      ""
//#endif

```

```
/*********************************************************************
*
*       _cbSendSystemDesc()
*
*  Function description
*    Sends SystemView description strings.
*/
static void _cbSendSystemDesc (void) {
 SEGGER_SYSVIEW_SendSysDesc ("N=" SYSVIEW_APP_NAME ",O=embOS,D="
SYSVIEW_DEVICE_NAME);
#ifdef SYSVIEW_SYSDESC0
 SEGGER_SYSVIEW_SendSysDesc (SYSVIEW_SYSDESC0);
#endif
#ifdef SYSVIEW_SYSDESC1
 SEGGER_SYSVIEW_SendSysDesc (SYSVIEW_SYSDESC1);
#endif
#ifdef SYSVIEW_SYSDESC2
 SEGGER_SYSVIEW_SendSysDesc (SYSVIEW_SYSDESC2);
#endif
}

/*********************************************************************
*
*       Global functions
*
**********************************************************************
*/
void SEGGER_SYSVIEW_Conf (void) {
 SEGGER_SYSVIEW_Init (SYSVIEW_TIMESTAMP_FREQ, SYSVIEW_CPU_FREQ,
                      &SYSVIEW_X_OS_TraceAPI, _cbSendSystemDesc);
 SEGGER_SYSVIEW_SetRAMBase (SYSVIEW_RAM_BASE);
 OS_SetTraceAPI (&embOS_TraceAPI_SYSVIEW);    // Configure embOS to use
SYSVIEW.
}

/*********************************************************************
*
*       SEGGER_SYSVIEW_X_GetTimestamp()
*
* Function description
*   Returns the current timestamp in ticks using the system tick
*   count and the SysTick counter.
*   All parameters of the SysTick have to be known and are set via
*   configuration defines on top of the file.
*
* Return value
*   The current timestamp.
*
* Additional information
*   SEGGER_SYSVIEW_X_GetTimestamp is always called when interrupts are
*   disabled. Therefore locking here is not required.
*/
U32 SEGGER_SYSVIEW_X_GetTimestamp (void) {
 U32 TickCount;
 U32 Cycles;
```

```
161   U32 CyclesPerTick;
162   //
163   // Get the cycles of the current system tick.
164   // SysTick is down-counting, subtract the current value from the number of
      cycles per
165   //
166   CyclesPerTick = SYST_RVR + 1;
167   Cycles =  (CyclesPerTick - SYST_CVR);
168   //
169   // Get the system tick count.
170   //
171   TickCount = SEGGER_SYSVIEW_TickCnt;
172   //
173   // If a SysTick interrupt is pending, re-read timer and adjust result
174   //
175   if  ((SCB_ICSR & SCB_ICSR_PENDSTSET_MASK)  != 0) {
176     Cycles =  (CyclesPerTick - SYST_CVR);
177     TickCount++;
178   }
179   Cycles += TickCount * CyclesPerTick;
180
181   return Cycles;
182 }
183
184 /************************** End of file ***************************/
```

7.4 Cortex-A / Cortex-R

录制模式	是否支持
连续模式	支持/否
单次模式	支持
死后分析	支持

略

7.5 Renesas RX

录制模式	是否支持
连续模式	支持
单次模式	支持
死后分析	支持

略

7.6 其他CPU

录制模式	是否支持

录制模式	是否支持
连续模式	否
单次模式	支持
死后分析	支持

在没有被sections覆盖的cpu上，SystemView可以使用在singleshot模式中。
要正确运行SystemView，必须配置与之前那些CPU同样的项目：

- 获取事件时间戳。
- 获取活动中断的中断Id。
- 锁定和解锁SystemView，以防止记录传输被中断

章节8 支持的操作系统 - Segger SystemView使用手册（译文）

章节8 支持的操作系统

下面的章节描述了哪些（实时）操作系统已经验证过可以使用SystemView，和如果配置它们。

8.1 embOS

SEGGER embOS（V4.12a或者更高版本）能够为SystemView和其他记录实现（当分析功能使能时）产生跟踪事件。

8.1.1 配置embOS

分析功能是在OS_LIBMODE_SP、OS_LIBMODE_DP和OS_LIBMODE_DT的库配置中使能的(详细信息请参阅embOS用户手册UM01001)。
除了SYSTEMVIEW和RTT核心模块之外，还需要包含以下文件：
对于Cortex-M0、Cortex-M1、Cortex-M3和Cortex-M4，需要包含
SEGGER_SYSVIEW_Config_embOS.c。
该文件还为SystemView提供了额外的功能，并允许修改配置从而与目标系统相匹配，比如定义应用程序名称、目标设备和目标核心频率。它初始化SYSTEMVIEW模块并配置embOS来将跟踪事件发送到SYSTEMVIEW。参阅第56页的SystemView系统信息配置的例子。

在main函数的开始，当目标板被初始化之后，必须调用SEGGER_SYSVIEW_Conf()来启用SystemView。现在，当应用程序运行时，SystemView可以连接到目标并开始记录事件。当SystemView连接或SEGGER_SYSVIEW_Start()被调用时，记录所有的任务、中断和OS调度器活动，以及embOS API调用。

8.2 uC/OS-III

SystemView可以用于Micrium公司的uC/OS-III，用来记录任务、中断和调度情况。

配置uC/OS-III

除了SYSTEMVIEW和RTT核心模块之外，还必须包括在应用程序项目中的以下文件：

SEGGER_SYSVIEW_Config_uCOSIII.c 提供了SystemView所需的功能，并允许修改配置从而与目标系统相匹配，比如定义应用程序名称、目标设备和目标核心频率。SystemView包中的示例配置文件被配置为与大多数Cortex-M3、Cortex-M4和Cortex-M7目标一起使用。请参阅第56页的SystemView系统信息配置的例子。

SEGGER_SYSVIEW_uCOSIII.c和os_trace_events.h提供uC/OS-III和SystemView之间的接口。它们通常不需要修改。

os_cfg_trace.h是SystemView实现uC/OS跟踪最小配置的需要的文件。如果项目已经包含该文件，请确保内容适合该应用程序。该文件包含两个定义，用于配置要管理的任务的最大数量和在SystemView记录中管理和命名的最大资源数。

```
#define TRACE_CFG_MAX_TASK 16u
#define TRACE_CFG_MAX_RESOURCES 16u
```

使能记录
可以在os_cfg.h中配置用于uC/OS-III的记录。
定义 OS_CFG_TRACE_EN 为 1，以使用基本的记录。
当 OS_CFG_TRACE_API_ENTER_EN 定义为1，API 函数调用也会被记录。
当 OS_CFG_TRACE_API_EXIT_EN 定义为1，API函数退出也会被记录。
在应用程序开始时，且系统已经初始化完成后，调用 TRACE_INIT()：

```
   [...]
    BSP_Init (); /* Initialize BSP functions       */
    CPU_Init (); /* Initialize the uC/CPU services */
#if  (defined (OS_CFG_TRACE_EN) &&  (OS_CFG_TRACE_EN > 0u))
    /* Initialize uC/OS-III Trace. Should be called after initializing the system. */
    TRACE_INIT ();
#endif
   [...]
```

8.3 uC/OS-II

SystemView可以用于Micrium公司的uC/OS-II，用来记录任务、中断和调度情况。SystemView支持uC/OS-II v2.92.13版本。
关于如何配置uC/OS-II以使用SystemView，请查看引导文章：< https://doc.micrium.com/display/osiidoc/SEGGER+SystemView>

*8.3.1 配置uC/OS-II

除了SYSTEMVIEW和RTT核心模块之外，还必须包括在应用程序项目中的以下文件：

SEGGER_SYSVIEW_Config_uCOSII.c 提供了SystemView所需的功能，并允许修改配置从而与目标系统相匹配，比如定义应用程序名称、目标设备和目标核心频率。SystemView包中的示例配置文件被配置为与大多数Cortex-M3、Cortex-M4和Cortex-M7目标一起使用。请参阅第56页的SystemView系统信息配置的例子。

SEGGER_SYSVIEW_uCOSII.c和os_trace_events.h提供uC/OS-III和SystemView之间的接口。它们通常不需要修改。

os_cfg_trace.h是SystemView实现uC/OS跟踪最小配置的需要的文件。如果项目已经包含该文件，请确保内容适合该应用程序。该文件包含两个定义，用于配置要管理的任务的最大数量和在SystemView记录中管理和命名的最大资源数。

```
#define TRACE_CFG_MAX_TASK 16u
#define TRACE_CFG_MAX_RESOURCES 16u
```

使能记录
可以在os_cfg.h中配置用于uC/OS-III的记录。
定义 OS_CFG_TRACE_EN 为 1，以使用基本的记录。
当 OS_CFG_TRACE_API_ENTER_EN 定义为1，API 函数调用也会被记录。
当 OS_CFG_TRACE_API_EXIT_EN 定义为1，API函数退出也会被记录。
在应用程序开始时，且系统已经初始化完成后，调用 TRACE_INIT() ：

```
   [...]
    BSP_Init (); /* Initialize BSP functions       */
    CPU_Init (); /* Initialize the uC/CPU services */
#if  (defined (OS_CFG_TRACE_EN) &&  (OS_CFG_TRACE_EN > 0u))
    /* Initialize uC/OS-III Trace. Should be called after initializing the
system. */
    TRACE_INIT ();
#endif
   [...]
```

8.4 Micrium OS 内核

SystemView可以用于Micrium公司的uC/OS-II，用来记录任务、中断和调度情况。

8.4.1 配置Micrium OS内核

除了SYSTEMVIEW和RTT核心模块之外，还必须包括在应用程序项目中的以下文件：

SEGGER_SYSVIEW_Config_MicriumOSKernel.c 提供了SystemView所需的功能，并允许修改配置从而与目标系统相匹配，比如定义应用程序名称、目标设备和目标核心频率。SystemView包中的示例配置文件被配置为与大多数Cortex-M3、Cortex-M4和Cortex-M7目标一起使用。请参阅第56页的SystemView系统信息配置的例子。

SEGGER_SYSVIEW_MicriumOSKernel.c和os_trace_events.h提供uC/OS-III和SystemView之间的接口。它们通常不需要修改。

os_cfg_trace.h是SystemView实现uC/OS跟踪最小配置的需要的文件。如果项目已经包含该文件，请确保内容适合该应用程序。该文件包含两个定义，用于配置要管理的任务的最大数量和在SystemView记录中管理和命名的最大资源数。

```
#define TRACE_CFG_MAX_TASK 16u
#define TRACE_CFG_MAX_RESOURCES 16u
```

使能记录
可以在os_cfg.h中配置用于uC/OS-III的记录。
定义 OS_CFG_TRACE_EN 为 1，以使用基本的记录。
当 OS_CFG_TRACE_API_ENTER_EN 定义为1，API 函数调用也会被记录。
当 OS_CFG_TRACE_API_EXIT_EN 定义为1，API函数退出也会被记录。
在应用程序开始时，且系统已经初始化完成后，调用 TRACE_INIT()：

```
   [...]
    BSP_Init (); /* Initialize BSP functions       */
    CPU_Init (); /* Initialize the uC/CPU services */
#if  (defined (OS_CFG_TRACE_EN) &&  (OS_CFG_TRACE_EN > 0u))
    /* Initialize uC/OS-III Trace. Should be called after initializing the
system. */
    TRACE_INIT ();
#endif
   [...]
```

8.5 FreeRTOS

FreeRTOS可以为SystemView生成跟踪事件，并且而不需要修改就允许一些基本但有用的分析。
对于更详细的分析，如调度器活动和中断，FreeRTOS源码和使用的接口必须稍微修改。

8.5.1 配置FreeRTOS

除系统视图和RTT核心模块外，
SEGGER_SYSVIEW_Config_FreeRTOS.c需要包含在应用程序中。该文件为SystemView提供了额外的功能，并允许配置与目标系统相匹配，比如定义应用程序名称、目标设备和目标核心频率。请参阅第56页的SystemView系统信息配置的例子。
SEGGER_SYSVIEW_FreeRTOS.h头必须包含在FreeRTOSConfig.h文件的末尾。或者在每一个引用FreeRTOS.h头文件的上方引用这个文件。它定义了跟踪宏来创建SYSTEMVIEW事件。
为了得到最好的结果，FreeRTOSConfig.h中的 INCLUDE_xtaskgetidletaskhandle 和 INCLUDE_pxTaskGetStackStart 应该定义为1。
补丁文件Sample/FreeRTOSV8/Patch/FreeRTOSV8.2.3_Core.patch显示了FreeRTOS 8.2.3源码和GCC / ARM_CM4F端口所需的修改。当使用另一个版本FreeRTOS或其他FreeRTOS的接口时，它可以作为参考。也就是说，如果使用的是另一个接口，而不是使用GCC / ARM_CM4F，则需要相应地添加traceISR_ENTER()、traceISR_EXIT()和traceISR_EXIT_TO_SCHEDULER()调用。
补丁文件Sample/FreeRTOSV9/Patch/FreeRTOSV9_Core.patch是用于FreeRTOS V9版本的补丁。

8.6 其他操作系统

其他的OSes还没有正式的验证。
如果您想要在其他操作系统中使用SystemView，请与SEGGER或OS供应商联系。操作系统工具也可以在接下来的章节中完成。

8.7 无操作系统

SystemView可以用于无操作系统或者其他未验证的操作系统的情况，用来记录中断和用户事件。

8.7.1 配置应用程序

除了SYSTEMVIEW和RTT核心模块之外，
SEGGER_SYSVIEW_Config_NoOS.c需要包含在应用程序中。该文件提供了SystemView所需函数的基本配置，可以修改为适合系统。请参阅第56页的SystemView系统信息配置的例子。
可以在SEGGER_SYSVIEW_Init中传递一个额外的SEGGER_SYSVIEW_OS_API指针，以提供关于程序的系统时间或"任务"的信息。
有关如何在系统中记录中断的描述，请参阅第117页。

章节9 性能和资源使用 - Segger SystemView使用手册（译文）

章节9 性能和资源使用

本章将介绍SystemView的性能和资源使用情况。它包含了关于典型系统内存需求的信息，这些信息可以用来获得对大多数目标系统的充分评估。

9.1 内存需求

取决于使用的操作系统集成、目标配置和编译器优化，因此内存需求可能不同。
为了实现性能和内存使用的平衡，建议为SystemView和RTT模块设置编译器优化级别。对于SystemView和RTT模块，即使在调试状态时，也应该始终打开编译器优化。

9.1.1 ROM需求

下表列出了组件的SystemView的ROM使用情况。对于使用智能链接器的IDE，只有使用的函数会被包含在应用程序中。

描述	ROM
最小核心代码需求	~920字节
基本记录函数（用于应用程序、OS和模块的事件）	~380字节
操作系统相关的记录函数	~360字节
中间层模块相关的记录函数	~120字节
完整的SystemView模块	~1.8K字节

下表列出了不同配置的SystemView的ROM使用情况

描述	配置	ROM
SystemView模块	平衡优化，无静态缓冲区	~1.8K字节
SystemView模块	平衡优化，静态缓冲区	~2.1K字节

描述	配置	ROM
SystemView模块	平衡优化，无静态缓冲区，死后模式	~1.4K字节
SystemView模块	平衡优化，静态缓冲区，死后模式	~1.7K字节
RTT模块	平衡优化	~0.5K字节

9.1.2 静态RAM需求

下表列出了不同配置的SystemView的RAM使用情况

描述	配置	RAM
SystemView模块	无静态缓冲区	~70字节+通道缓冲区
SystemView模块	静态缓冲区	~280字节+通道缓冲区
SystemView模块	无静态缓冲区，死后模式	~60字节+通道缓冲区
SystemView模块	静态缓冲区，死后模式	~180字节+通道缓冲区
RTT模块		~30字节+通道缓冲区

9.1.3 堆栈RAM需求

SystemView要求堆栈在应用程序中记录时间的每个上下文中记录事件（*brj2016注：这句话不知道该怎么理解*）。这通常包括调度器使用的系统堆栈、中断堆栈和任务堆栈。

由于SystemView处理系统描述和任务信息的传入请求，所以堆栈上必须有足够的空闲空间来记录事件并发送记录另一个事件的系统描述。

可以将SystemView配置为在较低的堆栈使用或更少的静态RAM使用之间进行选择。

描述	最大堆栈
无静态缓冲区，用于事件产生和编码	~230字节
静态缓冲区，用于事件产生和编码	~510字节
无静态缓冲区，用于事件产生和编码，死后模式	~150字节
静态缓冲区，用于事件产生和编码，死后模式	~280字节

章节10 集成向导 - Segger SystemView使用手册（译文）

章节10 集成向导

> 本节介绍如何将SEGGER SystemView集成到操作系统或中间件模块中，以记录其执行情况。

10.1 在操作系统中集成SEGGER SystemView

SEGGER SystemView可以集成在任何(实时)操作系统中，用于获取关于任务执行、OS内部活动(如调度程序)和OS API调用的信息。对于以下的RTOS，已经完成了这个集成，并且可以在盒子外使用。

- SEGGER embOS（V4.12a 或者更高版本）
- Micrium uC/OS-III（即将到来的V3.06）
- FreeRTOS（使用V8.2.3测试）

要集成到其他操作系统中，请与操作系统分销商联系，或者按照本节中的说明进行集成。

本节中的示例是用来说明何时调用特定的SystemView函数的伪代码。为了让跟踪检测工具对这些函数进行通用集成，可以将这些函数集成为函数宏，或者通过可配置的跟踪API来集成。

集成到操作系统核心
为了能够记录任务执行和上下文切换，操作系统核心必须被检测用来在适当的核心函数中生成SystemView事件。
在大多数情况下，中断执行也是由操作系统处理的。这允许在输入和退出中断的情况下对操作系统进行检测，否则将在应用程序中对每个ISR进行处理。
操作操作系统核心的第三个方面是提供运行时信息，以便进行更详细的分析。该信息包括系统时间，允许SystemView显示相对于应用程序开始时的时间戳，而不是开始记录的时间，并且任务列表被SystemView使用，以显示任务名称、堆栈信息和按优先级排序任务。

10.1.1 记录任务活动

SystemView可以记录系统和操作系统活动主要信息(如任务执行)的预定义系统事件集合。这些事件应该由操作系统在相应的函数中生成。
预定义的事件有：

事件	描述	SystemView API
Task Create	创建了新任务	SEGGER_SYSVIEW_OnTaskCreate
Task Start Ready	任务被标记为准备好启动或者恢复执行	SEGGER_SYSVIEW_OnTaskStartReady
Task Start Exec	任务被激活（启动或者恢复执行）	SEGGER_SYSVIEW_OnTaskStart
Task Stop Ready	任务被阻塞或暂停	SEGGER_SYSVIEW_OnTaskStopReady
Task Stop Exec	任务终止	SEGGER_SYSVIEW_OnTaskStopExec
System Idle	没有任务执行，系统进入空闲状态	SEGGER_SYSVIEW_OnIdle

10.1.1.1 Task Create任务创建

任务创建事件发生在系统创建一个任务时。
在任务创建事件调用带有新任务的Id的SEGGER_SYSVIEW_OnTaskCreate()函数。此外，建议使用SEGGER_SYSVIEW_SendTaskInfo()记录新任务的任务信息。

示例

```
void OS_CreateTask(TaskFunc* pF, unsigned Prio, const char* sName, void*
pStack) {
    SEGGER_SYSVIEW_TASKINFO Info;
    OS_TASK* pTask; // Pseudo struct to be replaced
    [OS specific code ...]
    SEGGER_SYSVIEW_OnTaskCreate((unsigned)pTask);
    memset(&Info, 0, sizeof(Info));
    //
    // Fill elements with current task information
    //
    Info.TaskID = (U32)pTask;
    Info.sName = pTask->Name;
    Info.Prio = pTask->Priority;
    Info.StackBase = (U32)pTask->pStack;
    Info.StackSize = pTask->StackSize;
    SEGGER_SYSVIEW_SendTaskInfo(&Info);
}
```

10.1.1.2 Task Start Ready 任务准备好启动或者恢复执行

当任务的延迟时间结束，或者当任务等待的资源可用，或者当事件触发时，任务开始准备事件可以发生。
在使用Task Start Ready事件时，将调用带有已准备就绪的任务的Id的SEGGER_SYSVIEW_OnTaskStartReady()函数。

示例

```
int OS_HandleTick(void) {
    int TaskReady = 0; // Pseudo variable indicating a task is ready
    [OS specific code ...]
    if (TaskReady) {
        SEGGER_SYSVIEW_OnTaskStartReady((unsigned)pTask);
    }
}
```

10.1.1.3 Task Start Exec

Task Start Exec事件发生发生在上下文即将被切换到激活任务。这通常是调度程序在有准备好的任务时完成的。
调用带有将要执行的任务的Id的SEGGER_SYSVIEW_OnTaskStartExec()函数，创建Task Start Exec事件。

示例

```
void OS_Switch(void) {
    [OS specific code ...]
    //
    // If a task is activated
    //
    SEGGER_SYSVIEW_OnTaskStartExec((unsigned)pTask);
    //
    // Else no task activated, go into idle state
    //
    SEGGER_SYSVIEW_OnIdle()
}
```

10.1.1.4 Task Stop Ready

任务被阻塞或暂停。
当任务被暂停或阻塞时， Task Stop Ready事件发生。例如因为它延迟了一个特定的时间，或者当它试图声明一个资源正在被另一个任务使用，或者当它等待事件发生时。当任务被暂停或阻塞时，调度器将激活另一个任务或进入空闲状态。
调用SEGGER_SYSVIEW_OnTaskStopReady()函数，并传入被阻塞的任务的Id，以及一个可以指示为什么任务被阻塞的理由，来创建Task Stop Ready事件。

示例

```
void OS_Delay(unsigned NumTicks) {
    [OS specific code ...]
    SEGGER_SYSVIEW_OnTaskStopReady(OS_Global.pCurrentTask, OS_CAUSE_WAITING);
}
```

10.1.1.5 Task Stop Exec

任务终止。
当任务最终停止执行时，例如当它完成任务并终止执行时，Task Stop Exec事件发生。
在Task Stop Ready事件中，调用SEGGER_SYSVIEW_OnTaskStopExec()来记录当前任务的停止状态。

示例

```
void OS_TerminateTask(void) {
    [OS specific code ...]
    SEGGER_SYSVIEW_OnTaskStopExec();
}
```

10.1.1.6 System Idle

没有任务执行时，系统进入空闲状态。
System Idle事件是当一个任务被暂停或停止时，而没有其他任务准备好时发生的。系统可以切换到一个空闲状态以节省电力，等待中断或任务准备就绪。
在某些操作系统中，空闲是由另一个任务处理的。在这种情况下，当空闲任务被激活时，建议记录系统空闲事件。
在SystemView中，空闲状态时间被显示为不是CPU负载。
在系统空闲事件中调用SEGGER_SYSVIEW_OnIdle()。

示例

```
void OS_Switch(void) {
    [OS specific code ...]
    //
    // If a task is activated
    //
    SEGGER_SYSVIEW_OnTaskStartExec((unsigned)pTask);
    //
    // Else no task activated, go into idle state
    //
    SEGGER_SYSVIEW_OnIdle()
}
```

10.1.2 记录中断

SystemView可以记录进入和退出中断服务例程(ISRs)。SystemView API为这些事件提供了函数，当它提供用于标记中断执行的函数时，该函数应该添加到操作系统中。
当OS调度程序被中断控制时，例如SysTick中断，退出中断事件应该区分恢复正常执行还是切换到调度器，并调用适当的SystemView函数。

10.1.2.1 进入中断

当操作系统提供一个函数来通知OS，中断代码正在执行时，在中断服务函数(ISR)开始时调用，OS函数应该调用SEGGER_SYSVIEW_RecordEnterISR()来记录进入中断事件。
当操作系统没有提供进入中断函数，或者ISR没有调用它时，用户有必要调用SEGGER_SYSVIEW_RecordEnterISR()来记录中断执行。
segger_sysview_conf. h中的SEGGER_SYSVIEW_RecordEnterISR()函数通过SEGGER_SYSVIEW_GET_INTERRUPT_ID()函数宏自动检索中断ID。

示例

```
void OS_EnterInterrupt(void) {
    [OS specific code ...]
    SEGGER_SYSVIEW_RecordEnterISR();
}
```

10.1.2.2 Exit Interrupt

当操作系统提供一个函数来通知OS，中断代码已经执行完时，该函数应该在中断服务函数结尾调用，OS函数应该调用：

- SEGGER_SYSVIEW_RecordExitISR()。 当系统将恢复正常执行时
- SEGGER_SYSVIEW_RecordExitISRToScheduler() 当中断引发一个上下文切换时。

示例

```
void OS_ExitInterrupt(void) {
    [OS specific code ...]
    //
    // If the interrupt will switch to the Scheduler
    //
    SEGGER_SYSVIEW_RecordExitISRToScheduler();
    //
    // Otherwise
    //
    SEGGER_SYSVIEW_RecordExitISR();
}
```

10.1.2.3 中断服务器函数举例

下面的两个例子展示了如何用OS中断处理和不使用OS中断来记录中断执行。

使用OS处理的例子

```
void Timer_Handler(void) {
    //
    // Inform OS about start of interrupt execution
    // (records SystemView Enter Interrupt event).
    //
    OS_EnterInterrupt();
    //
    // Interrupt functionality could be here
    //
    APP_TimerCnt++;
    //
    // Inform OS about end of interrupt execution
    // (records SystemView Exit Interrupt event).
    //
    OS_ExitInterrupt();
}
```

不使用OS处理的例子

```
void ADC_Handler(void) {
    //
    // Explicitly record SystemView Enter Interrupt event.
    // Should not be called in high-frequency interrupts.
    //
    SEGGER_SYSVIEW_RecordEnterISR();
    //
    // Interrupt functionality could be here
    //
    APP_ADCValue = ADC.Value;
    //
    // Explicitly record SystemView Exit Interrupt event.
    // Should not be called in high-frequency interrupts.
    //
    SEGGER_SYSVIEW_RecordExitISR();
}
```

10.1.3 记录运行时的信息

SystemView可以记录更详细的运行时的信息，如系统时间和任务信息。当SystemView运行时，记录开始并周期性地请求这些信息时，记录这些信息。
为了请求信息，在初始化时可以将带有操作系统特定函数的SEGGER_SYSVIEW_OS_API结构传递给SystemView。
SEGGER_SYSVIEW_OS_API设置是可选的，但建议允许SystemView显示更详细的信息。

SEGGER_SYSVIEW_OS_API

```
typedef struct {
    U64 (*pfGetTime) (void);
    void (*pfSendTaskList) (void);
} SEGGER_SYSVIEW_OS_API;
```

参数

参数	描述
pfGetTime	指向返回系统时间的函数
pfSendTaskList	指向记录整个任务列表的函数

10.1.3.1 pfGetTime

描述
获取系统时间，也就是自系统启动后经过的时间，单位us。
如果pfGetTime为空，SystemView可以显示相对于记录开始时的时间戳。
原型

> U64 (*pfGetTime) (void);

10.1.3.2 pfSendTaskList

描述
通过SEGGER_SYSVIEW_SendTaskInfo()记录整个任务列表。
如果pfSendTaskList为空，SystemView可能只会获取在记录时新创建任务的任务信息。当SystemView连接到当前任务列表时，将定期调用pfSendTaskList。
原型

> void (*pfSendTaskList) (void);
> **例子**

```
void cbSendTaskList(void) {
    SEGGER_SYSVIEW_TASKINFO Info;
    OS_TASK* pTask;
    OS_EnterRegion(); // Disable scheduling to make sure the task list does not change.
    for (pTask = OS_Global.pTask; pTask; pTask = pTask->pNext) {
        //
        // Fill all elements with 0 to allow extending the structure
        // in future version without breaking the code.
        //
        memset(&Info, 0, sizeof(Info));
```

```
        //
        // Fill elements with current task information
        //
        Info.TaskID = (U32)pTask;
        Info.sName = pTask->Name;
        Info.Prio = pTask->Priority;
        Info.StackBase = (U32)pTask->pStackBot;
        Info.StackSize = pTask->StackSize;
        //
        // Record current task information
        //
        SEGGER_SYSVIEW_SendTaskInfo(&Info);
    }
    OS_LeaveRegion(); // Enable scheduling again.
}
```

10.1.4 记录操作系统API调用

除了操作系统核心工具之外，SystemView还可以记录应用程序中的OS API调用。API函数可以像操作系统的核心一样被检测。
当传递简单参数时，或使用适当的SEGGER_SYSVIEW_EncodeXXX()函数创建SystemView事件并调用SEGGER_SYSVIEW_SendPacket()来记录它时，可以使用现成的SEGGER_SYSVIEW_RecordXXX()函数来完成记录API事件。
示例

```
/*********************************************************************
*
* OS_malloc()
*
* Function description
* API function to allocate memory on the heap.
*/
void OS_malloc(unsigned Size) {
    SEGGER_SYSVIEW_RecordU32(ID_OS_MALLOC, // Id of OS_malloc (>= 32)
    Size // First parameter
    );
    [OS specific code...]
}
```

为了记录API函数执行时间并记录其返回值的时间，API函数的返回也可以通过调用SEGGER_SYSVIEW_RecordEndCall来只记录返回值或者调用SEGGER_SYSVIEW_RecordEndCallReturnValue来记录退出事件以及返回值。

10.1.5 操作系统描述文件

为了使SystemView正确识别API调用，它需要在SystemView的 /description/ 目录中显示一个描述文件。文件的名称必须是SYSVIEW_<操作系统名>.txt，其中<操作系统名>是系统描述中发送的名称。

10.1.5.1 API函数描述

描述文件包括所有可以由操作系统记录的API函数。文件中的每一行都是一个函数，格式如下：

```
<EventID> <FunctionName> <ParameterDescription> | <ReturnValueDescription>1
```

EventID是为API函数记录的Id。取值范围：32~511。
FunctionName是API函数的名称，显示在SystemView的事件列中。它可以不包含空格。
ParameterDescription是用API函数记录的参数的描述字符串。
ReturnValueDescription是返回值的描述字符串，可以用SystemView进行记录。
ReturnValueDescription是可选的。

参数显示可以通过一组修饰符来配置：

- % b - 显示参数为二进制。
- %B - 参数显示为十六进制字符串(例如00 AA FF...)。
- %d - 显示参数为有符号的十进制整数。
- %D - 显示参数为时间值。
- %I - 显示参数作为资源名，如果资源id已知为SystemView。
- %p - 显示参数为4字节十六进制整数(如0xAABBCCDD)。
- %s - 显示参数为字符串。
- %t - 显示参数作为任务名称，如果任务id已知为SystemView。
- %u - 显示参数为无符号十进制整数。
- %x - 显示参数为十六进制整数

例子
下面的例子显示了SYSVIEW_embOS.txt的部分内容

```
35      OS_CheckTimer           pGlobal=%p
42      OS_Delay                Delay=%u
43      OS_DelayUntil           Time=%u
44      OS_setPriority          Task=%t Pri=%u
45      OS_WakeTask             Task=%t
46      OS_CreateTask           Task=%t Pri=%u Stack=%p Size=%u
```

除了默认的修饰符，描述文件还可以定义NamedTypes来将数值映射到字符串，例如，可以用来显示枚举的文本值或错误代码。
NamedTypes有以下格式：

```
NamedType <TypeName> <Key>=<Value> [<Key1>=<Value1> ...]1
```

NamedTypes可以在参数描述和返回值描述中使用。
示例

```
#
# Types for parameter formatters
#
NamedType OSErr 0=OS_ERR_NONE
NamedType OSErr 10000=OS_ERR_A 10001=OS_ERR_ACCEPT_ISR
NamedType OSErr 12000=OS_ERR_C 12001=OS_ERR_CREATE_ISR
NamedType OSErr 13000=OS_ERR_D 13001=OS_ERR_DEL_ISR
NamedType OSFlag 0=FLAG_NONE 1=FLAG_READ 2=FLAG_WRITE 3=FLAG_READ_WRITE
#
# API Functions
#
34 OSFunc Param=%OSFlag | Returns %OSErr
```

10.1.5.2 任务状态描述

当一个任务暂停执行时，它的状态被记录在SystemView事件中。
这个任务状态可以被转换为SystemView中的文本表示，并使用任务状态解析。
TaskState以下格式：

```
TaskState <Mask> <Key>=<Value>, [<Key1>=<Value1>, ...]
```

示例

```
#
# Task States
#
TaskState 0xFF 0=Ready, 1=Delayed or Timeout, 2=Pending, 3=Pending with
Timeout,
4=Suspended, 5=Suspended with Timeout, 6=Suspended and Pending, 7=Suspended
and
Pending with Timeout, 255=Deleted
```

10.1.5.3 Option 描述

在描述文件中也可以设置操作系统特定选项来配置SystemView。
目前可以在描述文件中插入的选项为:
Option ReversePriority：更高的任务优先级值等于较低的任务优先级。

10.1.6 操作系统集成示例

下面的代码展示了如何在基于伪代码片段的操作系统中集成SystemView，并可作为参考使用。

```
/*********************************************************************
*               (c) SEGGER Microcontroller GmbH & Co. KG             *
*                       The Embedded Experts                         *
*                          www.segger.com                            *
**********************************************************************
------------------------- END-OF-HEADER ------------------------------

Purpose : Pseudo-code OS with SEGGER SystemView integration.
*/

/*********************************************************************
```

```
*
*       OS_CreateTask()
*
*  Function description
*    Create a new task and add it to the system.
*/
void OS_CreateTask (TaskFunc* pF, unsigned Prio, const char* sName, void*
pStack) {
  SEGGER_SYSVIEW_TASKINFO Info;
  OS_TASK*                pTask;  // Pseudo struct to be replaced

   [OS specific code ...]

  SEGGER_SYSVIEW_OnTaskCreate ((unsigned)pTask);
  memset (&Info, 0, sizeof(Info));
  //
  // Fill elements with current task information
  //
  Info.TaskID     =  (U32)pTask;
  Info.sName      = pTask->Name;
  Info.Prio       = pTask->Priority;
  Info.StackBase  =  (U32)pTask->pStack;
  Info.StackSize  = pTask->StackSize;
  SEGGER_SYSVIEW_SendTaskInfo (&Info);
}

/*********************************************************************
*
*       OS_TerminateTask()
*
*  Function description
*    Terminate a task and remove it from the system.
*/
void OS_TerminateTask (void) {

   [OS specific code ...]

  SEGGER_SYSVIEW_OnTaskStopExec ();
}

/*********************************************************************
*
*       OS_Delay()
*
*  Function description
*    Delay and suspend a task for the given time.
*/
void OS_Delay (unsigned NumTicks) {

   [OS specific code ...]

  SEGGER_SYSVIEW_OnTaskStopReady (OS_Global.pCurrentTask,
OS_CAUSE_WAITING);
}

```

```c
/*********************************************************************
*
*       OS_HandleTick()
*
*  Function description
*    OS System Tick handler.
*/
int OS_HandleTick (void) {
  int TaskReady = 0;   // Pseudo variable indicating a task is ready

   [OS specific code ...]

  if  (TaskReady) {
    SEGGER_SYSVIEW_OnTaskStartReady ((unsigned)pTask);
  }
}

/*********************************************************************
*
*       OS_Switch()
*
*  Function description
*    Switch to the next ready task or go to idle.
*/
void OS_Switch (void) {

   [OS specific code ...]

  //
  // If a task is activated
  //
  SEGGER_SYSVIEW_OnTaskStartExec ((unsigned)pTask);
  //
  // Else no task activated, go into idle state
  //
  SEGGER_SYSVIEW_OnIdle ()
}

/*********************************************************************
*
*       OS_EnterInterrupt()
*
*  Function description
*    Inform the OS about start of interrupt execution.
*/
void OS_EnterInterrupt (void) {

   [OS specific code ...]

  SEGGER_SYSVIEW_RecordEnterISR ();
}
/*********************************************************************
*
*       OS_ExitInterrupt()
*
```

```
*  Function description
*    Inform the OS about end of interrupt execution and switch to
*    Scheduler if necessary.
*/
void OS_ExitInterrupt (void) {

   [OS specific code ...]
  //
  // If the interrupt will switch to the Scheduler
  //
  SEGGER_SYSVIEW_RecordExitISRToScheduler ();
  //
  // Otherwise
  //
  SEGGER_SYSVIEW_RecordExitISR ();
}
```

10.2 在中间层模块集成SEGGER SystemView

SEGGER SystemView还可以集成到中间件模块，甚至是应用程序模块中，以获取关于这些模块的执行的信息，比如API调用或中断触发事件。此集成用于在SEGGER embOS/IP中使用，以监视通过IP和SEGGER emFile发送和接收数据包，以记录API调用。
要集成到其他模块，请与您的分销商联系，或者按照本节中的说明进行集成。

10.2.1 注册模块

为了能够记录中间件模块事件，模块必须通过SEGGER_SYSVIEW_RegisterModule()在SystemView进行注册。
该模块传递一个SEGGER_SYSVIEW_MODULE 结构体指针，该指针包含有关模块的信息，并接收模块可以生成的事件id的事件偏移量。
在注册时，必须在SEGGER_SYSVIEW_MODULE结构中设置sDescription和NumEvents。也可以设置pfSendModuleDesc。
当SEGGER_SYSVIEW_RegisterModule()返回时，SEGGER_SYSVIEW_MODULE结构中的EventOffset被设置为模块可能生成的最低的事件Id，而pNext将指向下一个注册模块，以创建一个链表。因此，SEGGER_SYSVIEW_MODULE结构必须是可写的，并且可能不会在堆栈上分配。

SEGGER_SYSVIEW_MODULE

```
struct SEGGER_SYSVIEW_MODULE {
        const char*                  sModule;
              U32                    NumEvents;
              U32                    EventOffset;
              void                    (*pfSendModuleDesc)(void);
              SEGGER_SYSVIEW_MODULE*  pNext;
};
```

参数

参数	描述
sModule	指向包含模块名称和模块事件描述的字符串的指针。
NumEvents	模块需要注册的事件数

参数	描述
EventOffset	事件id的偏移量。输出参数，由这个函数设置。调用此函数后不修改
pfSendModuleDesc	回调函数指针，向SystemView发送更详细的模块描述。
pNext	指向下一个注册模块的指针。输出参数，由这个函数设置。调用此函数后不修改

示例

```
SEGGER_SYSVIEW_MODULE IPModule = {
  "M=embOSIP, " \
  "0 SendPacket IFace=%u NumBytes=%u, " \
  "1 ReceivePacket Iface=%d NumBytes=%u", // sModule
  2,                                      // NumEvents
  0,
  // EventOffset, Set by SEGGER_SYSVIEW_RegisterModule()
  NULL,
  // pfSendModuleDesc, NULL: No additional module description
  NULL,
  // pNext, Set by SEGGER_SYSVIEW_RegisterModule()
};

static void _IPTraceConfig (void) {
 //
 // Register embOS/IP at SystemView.
 // SystemView has to be initialized before.
 //
 SEGGER_SYSVIEW_RegisterModule (&IPModule);
}
```

10.2.2 记录模块活动

为了能够记录模块的活动，模块必须被检测以在适当的函数中生成SystemView事件。
通过对SystemView函数进行直接集成，可以通过可配置的宏函数或API结构来实现对模块的检测，这些功能可以通过SystemView来填充和设置。
当传递简单参数时，或使用适当的SEGGER_SYSVIEW_EncodeXXX()函数创建SystemView事件，并调用SEGGER_SYSVIEW_SendPacket()来记录该事件时，可以使用随时可用的SEGGER_SYSVIEW_RecordXXX()函数来完成记录事件。

示例

```
int SendPacket (IP_PACKET *pPacket) {
 //
 // The IP stack sends a packet.
 // Record it according to the module description of SendPacket.
 //
 SEGGER_SYSVIEW_RecordU32x2 (
 // Id of SendPacket (0) + Offset for the registered module
                             ID_SENDPACKET + IPModule.EventOffset,
 // First parameter  (displayed as event parameter IFace)
                             pPacket->Interface,
 // Second parameter (displayed as event parameter NumBytes)
```

```
                              pPacket->NumBytes
                             );

  [Module specific code...]
}
```

有关更多信息，请参阅第108页上的OS API调用和第117页的API引用。
与操作系统一样，可以在描述文件中使用模块的名称(M=的值)提供中间件模块描述。参见第108页的OS描述文件。

10.2.3 提供模块描述

SEGGER_SYSVIEW_MODULE.sModule指向一个包含注册模块的基本信息的字符串，该模块是一个逗号分隔的列表，可以包含以下内容：

项	标识	举例
模块名称	M	“M=embOSIP”
模块令牌	T	“T=IP”
描述	S	“S=’embOS/IP V12.09’”
模块事件		“0 SendPacket IFace=%u NumBytes=%u”

字符串长度不能超过SEGGER_SYSVIEW_MAX_STRING_LEN，默认情况下是128。

要发送额外的描述字符串，并发送由模块所使用和记录的资源的名称，在注册模块时可以设置SEGGER_SYSVIEW_MODULE.fSendModuleDesc。

SEGGER_SYSVIEW_MODULE.pfSendModuleDesc在SystemView连接时被周期性调用。它可以调用SEGGER_SYSVIEW_RecordModuleDescription()和SEGGER_SYSVIEW_NameResource()。

示例

```
static void _cbSendIPModuleDesc (void) {
  SEGGER_SYSVIEW_NameResource ((U32)&(RxPacketFifo), "Rx FIFO");
  SEGGER_SYSVIEW_NameResource ((U32)&(TxPacketFifo), "Tx FIFO");
  SEGGER_SYSVIEW_RecordModuleDescription (&IPModule, "T=IP, S='embOS/IP V12.09'");
}

SEGGER_SYSVIEW_MODULE IPModule = {
  "M=embOSIP, " \
  "0 SendPacket IFace=%u NumBytes=%u, " \
  "1 ReceivePacket Iface=%d NumBytes=%u", // sModule
  2,                                      // NumEvents
  0,                                      // EventOffset, Set by RegisterModule()
  _cbSendIPModuleDesc,                    // pfSendModuleDesc
  NULL,                                   // pNext, Set by RegisterModule()
};
```

章节11 API参考 - Segger SystemView使用手册（译文）

章节11 API参考

本节描述了SEGGER SystemView的公共API。

11.1 SEGGER SystemView API函数

可以使用以下函数将SEGGER SystemView添加到应用程序中，或者将SEGGER SystemView集成到操作系统和中间件模块中。
由应用程序调用的控制函数。

函数	描述
SEGGER_SYSVIEW_Init()	初始化SYSVIEW模块
SEGGER_SYSVIEW_Start()	开始录制SystemView事件
SEGGER_SYSVIEW_Stop()	停止录制SystemView事件
SEGGER_SYSVIEW_EnableEvents()	使能产生标准SystemView事件功能
SEGGER_SYSVIEW_DisableEvents()	禁止产生标准SystemView事件功能

应用程序系统调用的配置函数。通常包括在系统回调函数中。

函数	描述
SEGGER_SYSVIEW_SetRAMBase()	设置RAM基地址，这是为了节省带宽而从IDs中减去的地址
SEGGER_SYSVIEW_SendTaskList()	向主机发送所有任务描述
SEGGER_SYSVIEW_SendTaskInfo()	发送一个任务信息包，包含任务ID、任务优先级和任务名称
SEGGER_SYSVIEW_SendSysDesc()	向主机发送系统描述字符串
SEGGER_SYSVIEW_NameResource()	发送用于在SystemView中显示的资源名称

中间层模块注册和配置函数

函数	描述
SEGGER_SYSVIEW_RegisterModule()	注册一个中间层模块，以记录它的事件
SEGGER_SYSVIEW_RecordModuleDescription()	向主机发送已注册模块的详细信息

操作系统相关的事件记录函数

函数	描述
SEGGER_SYSVIEW_OnIdle()	记录一次空闲事件
SEGGER_SYSVIEW_OnTaskCreate()	记录任务创建事件
SEGGER_SYSVIEW_OnTaskStartExec()	记录任务开始执行事件
SEGGER_SYSVIEW_OnTaskStartReady()	记录任务准备好开始事件
SEGGER_SYSVIEW_OnTaskStopExec()	记录任务停止执行事件
SEGGER_SYSVIEW_OnTaskStopReady()	记录任务准备好停止事件
SEGGER_SYSVIEW_OnTaskTerminate()	记录任务终止事件

由操作系统或者模块调用的生成事件记录函数

函数	描述
SEGGER_SYSVIEW_RecordEndCallU32()	格式化并发送带有返回值的API调用结束事件
SEGGER_SYSVIEW_RecordEndCall()	格式化并发送无返回值的API调用结束事件
SEGGER_SYSVIEW_RecordEnterISR()	格式化并发送进入中断服务函数事件
SEGGER_SYSVIEW_RecordEnterTimer()	格式化并发送进入定时器函数事件
SEGGER_SYSVIEW_RecordExitISRToScheduler()	格式化并发送退出中断函数进入调度器事件
SEGGER_SYSVIEW_RecordExitISR()	格式化并发送中断服务函数退出事件
SEGGER_SYSVIEW_RecordExitTimer()	格式化并发送定时器退出事件
SEGGER_SYSVIEW_RecordString()	格式化并发送包含一串字符的SystemView包
SEGGER_SYSVIEW_RecordSystime()	格式化并发送包含单个U64或者U32类型的参数的SystemView 系统时间
SEGGER_SYSVIEW_RecordVoid()	格式化并发送不包含有效数据的SystemView包
SEGGER_SYSVIEW_RecordU32()	格式化并发送包含单个U32类型的有效参数的SystemView包
SEGGER_SYSVIEW_RecordU32X10()	格式化并发送包含10个U32类型的有效参数的SystemView包
SEGGER_SYSVIEW_RecordU32X2()	格式化并发送包含2个U32类型的有效参数的SystemView包
SEGGER_SYSVIEW_RecordU32X3()	格式化并发送包含3个U32类型的有效参数的SystemView包
SEGGER_SYSVIEW_RecordU32X4()	格式化并发送包含4个U32类型的有效参数的SystemView包
SEGGER_SYSVIEW_RecordU32X5()	格式化并发送包含5个U32类型的有效参数的SystemView包

函数	描述
SEGGER_SYSVIEW_RecordU32X6()	格式化并发送包含6个U32类型的有效参数的SystemView包
SEGGER_SYSVIEW_RecordU32X7()	格式化并发送包含7个U32类型的有效参数的SystemView包
SEGGER_SYSVIEW_RecordU32X8()	格式化并发送包含8个U32类型的有效参数的SystemView包
SEGGER_SYSVIEW_RecordU32X9()	格式化并发送包含9个U32类型的有效参数的SystemView包
SEGGER_SYSVIEW_SendPacket()	发送一个事件数据包
SEGGER_SYSVIEW_EncodeU32()	以可变长度格式编码一个U32类型数据
SEGGER_SYSVIEW_EncodeData()	以可变长度格式编码一个字节类型缓存
SEGGER_SYSVIEW_EncodeString()	以可变长度格式编码一个字符串
SEGGER_SYSVIEW_EncodeId()	以缩小的可变长度格式编码一个32位 Id
SEGGER_SYSVIEW_ShrinkId()	获取缩小的Id值用于将来处理，例如在SEGGER_SYSVIEW_NameResource()函数中.

在用户应用程序中调用的事件记录函数

函数	描述
SEGGER_SYSVIEW_OnUserStart()	发送用户事件开始，例如一个子函数开始
SEGGER_SYSVIEW_OnUserStop()	发送用户事件结束事件，例如从子函数返回
SEGGER_SYSVIEW_PrintfHostEx()	打印一个字符串，该字符串由SystemViewer以附加信息格式化。
SEGGER_SYSVIEW_PrintfTargetEx()	打印一个字符串，该字符串由目标板在发送给主机前以附加信息格式化
SEGGER_SYSVIEW_PrintfHost()	打印一个由SystemView格式化的字符串
SEGGER_SYSVIEW_PrintfTarget()	打印一个由目标系统在发送给主机前格式化的字符串
SEGGER_SYSVIEW_Print()	向主机打印一个字符串.
SEGGER_SYSVIEW_WarnfHost()	打印一个警告字符串，该字符串由SystemView在主机上格式化
SEGGER_SYSVIEW_WarnfTarget()	打印一个警告字符串，该字符串由目标系统在发给主机前格式化
SEGGER_SYSVIEW_Warn()	打印一个警告字符串发送给主机
SEGGER_SYSVIEW_ErrorfHost()	打印一个错误字符串，该字符串由SystemView在主机上格式化

函数	描述
SEGGER_SYSVIEW_ErrorfTarget()	打印一个错误字符串，该字符串由目标系统在发给主机前格式化
SEGGER_SYSVIEW_Error()	打印一个错误字符串发送给主机

应用程序提供的函数

函数	描述
SEGGER_SYSVIEW_Conf()	对于特定操作系统，初始化和配置SystemView
SEGGER_SYSVIEW_X_GetTimestamp()	用于SystemView获取时间戳的回调函数

11.1.1 SEGGER_SYSVIEW_Conf()

描述
可以与操作系统集成一起使用，以便更容易地初始化SystemView和OS SystemView接口。
该函数通常在使用的操作系统配置文件SEGGER_SYSVIEW_Config_[OS].c中提供。
原型

> void SEGGER_SYSVIEW_Conf(void);

实施举例

```
void SEGGER_SYSVIEW_Conf (void) {
 //
 // Initialize SystemView
 //
 SEGGER_SYSVIEW_Init (SYSVIEW_TIMESTAMP_FREQ,   // Frequency of the timestamp.
                      SYSVIEW_CPU_FREQ,         // Frequency of the system.
                      &SYSVIEW_X_OS_TraceAPI,
 // OS-specific SEGGER_SYSVIEW_OS_API
                      _cbSendSystemDesc
 // Callback for application-specific description
                     );
 SEGGER_SYSVIEW_SetRAMBase (SYSVIEW_RAM_BASE);
 // Explicitly set the RAM base address.
 OS_SetTraceAPI (&embOS_TraceAPI_SYSVIEW);
 // Configure embOS to use SystemView via the Trace-API.
}
```

章节12 常见问题 - Segger SystemView使用手册（译文）

章节12 常见问题

问：当我正在调试我的应用程序时，我可以使用SystemView应用程序吗?

可以。SystemView可以与调试器并行运行，并进行连续记录。为确保数据可以阅读速度不够快，请配置调试器连接到一个高速接口(≥4 MHz)。

问：我能在没有J-Link的情况下进行连续记录吗?
不能。连续记录需要J-Link实时传输技术(RTT)自动读取目标数据。但单次记录和死后记录可以用任何调试探针完成。

问：我可以在Cortex-A、Cortex-R 或 ARM7 上连续记录吗?

不能。RTT需要在目标运行时对目标进行内存访问。如果你有一个问题中的设备，只有通过单次记录可以完成。

问：在连续记录的情况下，会出现溢出事件。我该如何预防呢?

当SystemView RTT缓冲区满时溢出事件会发生。这可能有以下原因：

- J-Link一直忙于调试器，无法快速读取数据。
- 目标接口速度太低，无法快速读取数据。
- 应用程序生成了太多的事件以致于填满缓冲区。
  为了防止这种情况：
- 在目标运行时将调试器与J-Link的交互最小化。(即禁用实时监视（live watch）)
- 在与J-Link相关的所有实例中选择更高的接口速度(例如，调试器和SystemView)，以及选择更大的SystemView缓冲区。(1-4k字节)
- 在没有调试器的情况下独立运行SystemView。

问：SystemView不能找到RTT控制块，我如何配置它?

RTT控制块的自动检测只能在初始化后的已知RAM地址范围内进行。确保应用程序启动时已经运行。如果RTT控制块位于所选设备的已知范围之外，则选择“地址”并输入RTT控制块的确切地址或选择“地址范围”，并输入RTT控制块将在的地址范围。

问：我需要选择一个目标设备来开始记录吗?

是的。J-Link需要与目标设备连接起来。下拉列表列出了最近使用的设备。要选择另一个设备，只需输入它的名称。可以在这里找到支持的设备列表。

问：我的问题没有列在上面。我在哪里可以得到更多的信息?
A:想了解更多信息，请在SEGGER论坛https://forum.segger.com询问你的问题