SystemView 介绍与移植

> **version：** *v1.0　　「2022.7.20」　第一版*
>
> **author：** *Y.Z.T.*
>
> **摘要：** *介绍systemview，以及移植过程*
>
> **简介：** *前几天在看RM官方代码(**RoboRTS-Firmware-icra2021**)时，看到其中对SystemView的应用，稍微花了点时间研究了一　　下，感觉挺有用的，记录一下*

前言

在使用FreeRTOS进行应用开发时经常会遇到普通的方式难以调试的问题，如栈内存不足等等，同时也希望对运行的多个Task进行实时的

性能及资源占用的分析，通常的调试手段在这里就变的心有余而力不足了。以FreeRTOS为例，如何在长时间的运行过程中收集调试数据

进行分析，以及如何调试不同的组件（如Queue，Notification，Semaphore等等）？这个时候就需要Trace工具帮忙了。针对RTOS的

Trace需求，接下来将针对SEGGER开发的SystemView Trace工具进行介绍。

什么是SystemView?

SystemView是SEGGER开发的针对嵌入式系统的trace工具，支持多种RTOS，也支持自定义OS的移植（需实现trace API，参见User

Manual）。其核心基于**SEGGER RTT**，一个**Host-Target**间的通信框架，可通过多种方式连接，除J-LINK之外还可以使用串口及TCP-IP协

议，对非商业用途免费且无功能限制。

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。

官方介绍

SystemView的原理简单来讲是这样的：首先上位机PC端有一个处理程序，下位机ARM上需增加部分代码，用于记录嵌入式系统的一些数

据，通过连接的仿真器接口将数据传输到上位机，然后PC端的处理程序处理这些数据，由于数据传输使用的是**SEGGER J-link**的实时传输

技术（RTT），所以**SystemView**可以实时分析和展示数据。分析的内容包括中断、任务、软件定时器执行的时间，切换的时间，以及发

生的事件等，这些分析都是实时的，丝毫不影响下位机的运行。这样就可以验证我们设计的整个嵌入式系统是否按照我们的预期在工作，

比如任务的切换逻辑，中断的触发等。它可以应用于带有RTOS的系统，也支持裸机，对于下位机资源的消耗量也是比较少的。

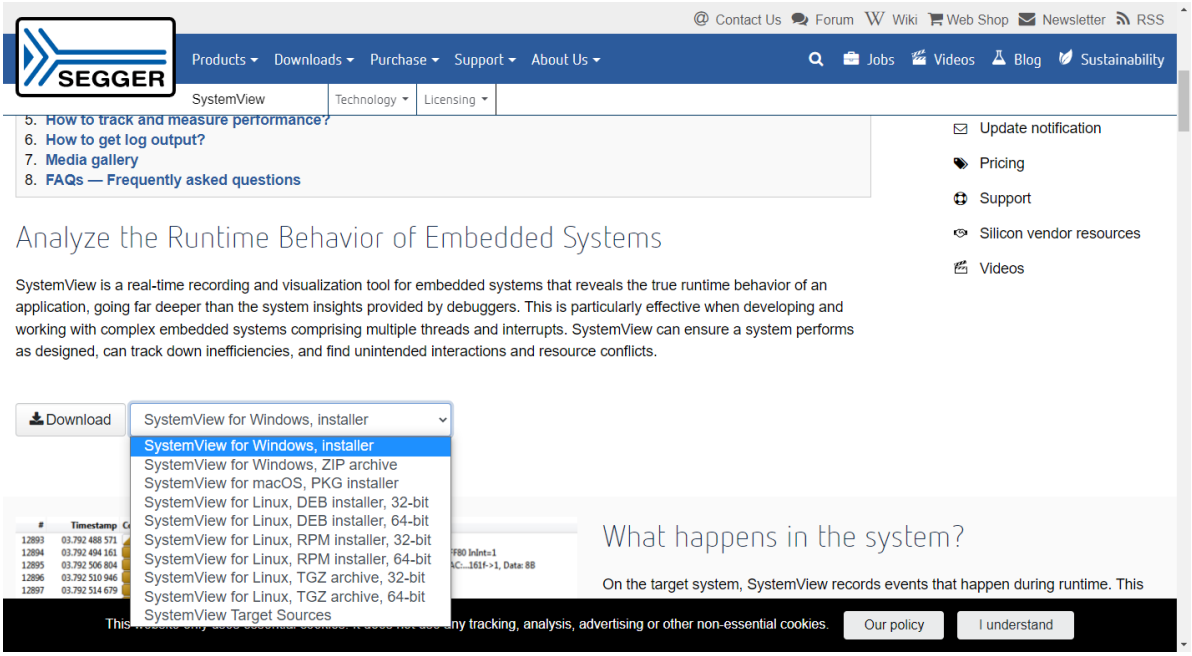
测试环境

> **开发板：** *ACE实验室H7通用开发板 // ACE实验室哨兵H7云台板*
>
> > **主控芯片：** *STM32H750VBT6*
> >
> > **CPU：** *Cortex-M7*
> >
> > **最高主频：** 480MHZ
>
> **测试程序：** *[2022_sentryH750_v1.3.5] / [systemview_test]*
>
> > **FreeRTOS版本：** *v10.3.1*
> >
> > **FreeRTOS说明：** *FreeRTOS为STM32CubeMX配置的未修改版本。*
>
> **开发平台：** *VSCODE、MDK-ARM*
>
> > **程序编辑：** *VSCODE*
> >
> > **程序调试：** *MDK-ARM [v 5.35.0.2]*
>
> **调试设备：** *J-Link仿真器： [ACE - Sentry-哨兵] / [Sentry]*
>
> **Systemview 程序包：** *SystemView_Windows_V332_x86*
>
> > 下载链接

移植过程

PC端程序包下载：

在官网上下载Systemview程序包



window端的程序有两种

安装包

从官网下载最新的DEB包或者RPM包并执行。安装向导会指引完成安装。

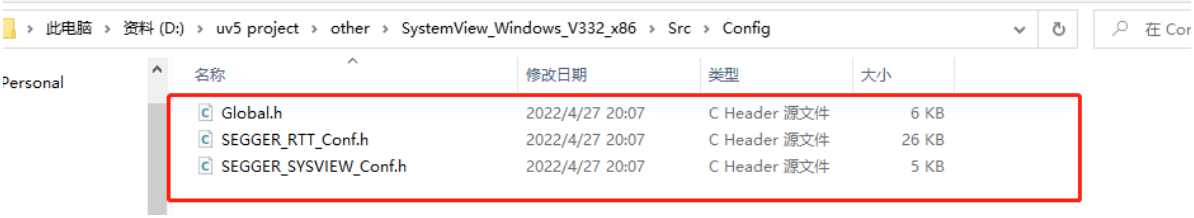
可移植压缩包

从官网下载最新的存档，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。

代码移植

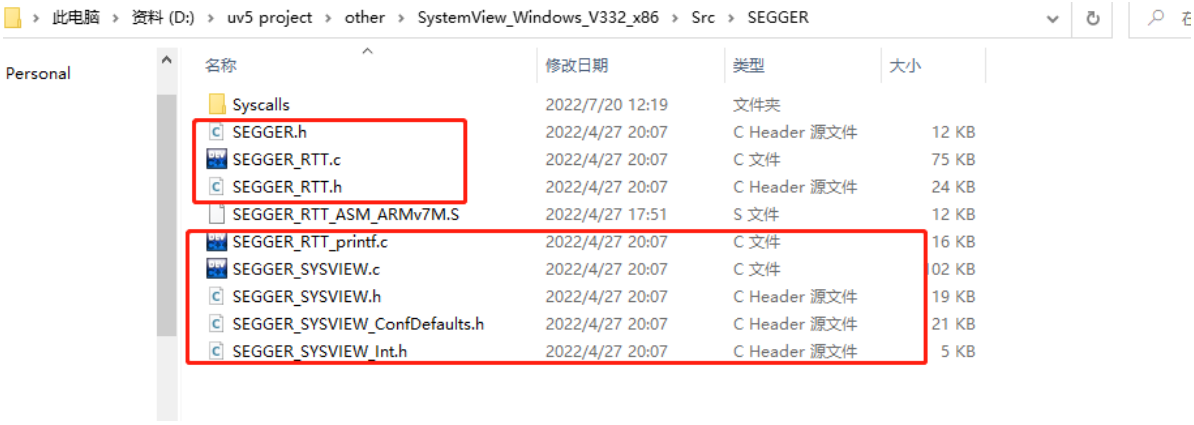
将程序包内`SEGGER`目录下代码以及`Sample`中的**FreeRTOS代码**添加到项目中。

1. 首先是`Config`文件夹下面的所有文件。

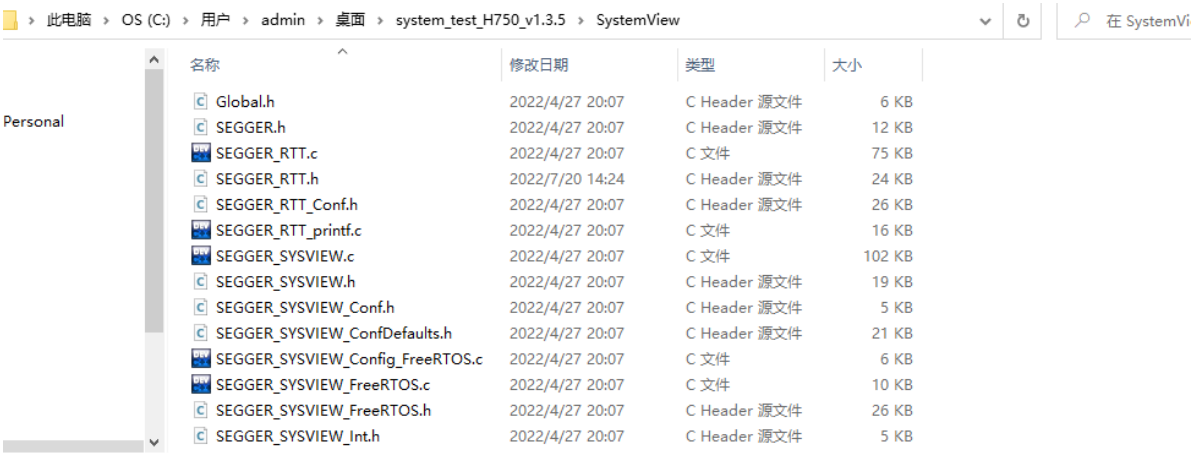


2. 根据自己的RTOS版本选择系统配置文件（我的是 *FreeRTOS v10.3.1* 选择用 FreeRTOSV10）

3. RTT相关文件



4. 放进项目里面的systemview文件夹



5. 程序添加

1. 在`FreeRTOS.h`中添加`#include "SEGGER_SYSVIEW_FreeRTOS.h"`

```
SEGGER_SYSVIEW_Conf();
```

```
/* Application specific configuration options. */
#include "FreeRTOSConfig.h"
#include "SEGGER_SYSVIEW_FreeRTOS.h"
/* Basic FreeRTOS definitions. */
```

2. 在`main.c`里面添加`#include "SEGGER_SYSVIEW.h"`

```
#include "SEGGER_SYSVIEW.h"
```

```
/* Private includes ----------------------------------------------
/* USER CODE BEGIN Includes */
#include "SysInit.h"
#include "SEGGER_SYSVIEW.h"

/* USER CODE END Includes */
```

3. *然后还需要在系统启动前添加segger systemview的初始化*

```
    /*  systemview 初始化 */
    SEGGER_SYSVIEW_Conf();
```

```
    /* 底盘云台初始化选择 */
    task_init();

    /*  systemview 初始化 */
    SEGGER_SYSVIEW_Conf();
```

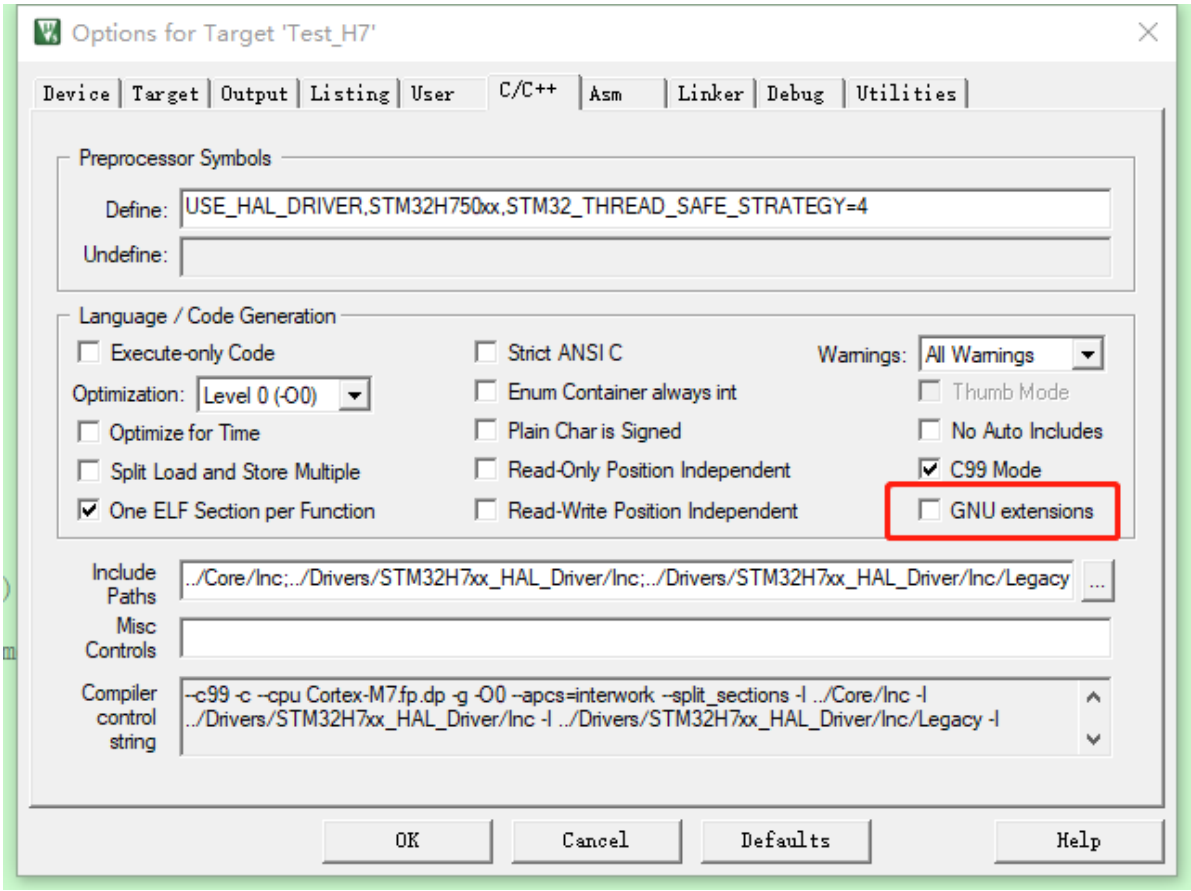
4. 在 `FreeRTOSConfig.h` 中添加宏定义

```
#define INCLUDE_xTaskGetIdleTaskHandle  1
#define INCLUDE_pxTaskGetStackStart     1
```

```
/* USER CODE BEGIN Defines */
/* Section where parameter definitions can be added (for instance, to override default ones in FreeRTOS.h) *
#define INCLUDE_xTaskGetIdleTaskHandle  1
#define INCLUDE_pxTaskGetStackStart     1
/* USER CODE END Defines */
```

5. 取消勾选keil的GNU勾选模式

[不然会报错]



环境问题（未解决）

[ps: 取消之后目前已知问题，会导致print 重定义出问题]

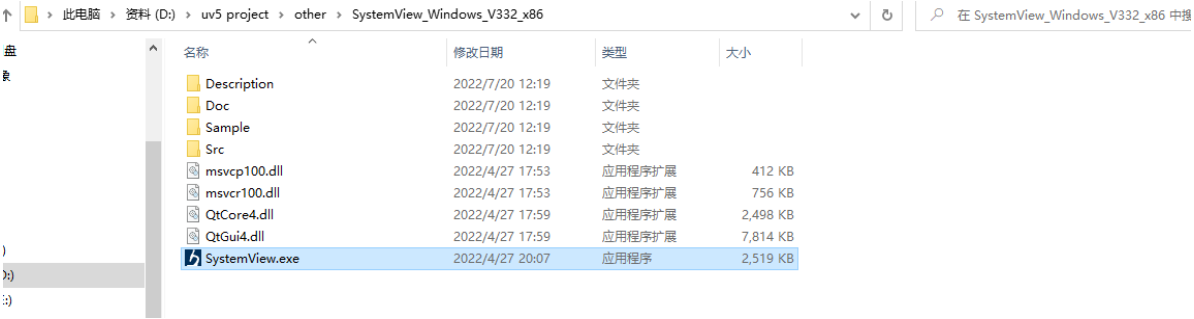
```c
/* USER CODE BEGIN 1 */
#if 0
#pragma import(__use_no_semihosting)  //确保没有从 C 库链接使用半主机的函数

_sys_exit(int x)     //定义_sys_exit()以避免使用半主机模式

{ x = x; }

struct __FILE  //标准库需要的支持函数

{

    int handle;

};

/* FILE is typedef'  d in stdio.h. */

FILE __stdout;

int fputc(int ch, FILE *f)
{
    uint8_t temp[1]={ch};
    HAL_UART_Transmit(&huart2,temp,1,2);         //UartHandle是串口的句柄
    return ch;
}

///重定向c库函数scanf到串口DEBUG_USART，重写向后可使用scanf、getchar等函数
int fgetc(FILE *f)
{
    int ch;
    HAL_UART_Receive(&huart3, (uint8_t *)&ch, 1, 1000);
    return (ch);
}
#endif
```

暂时是注释解决

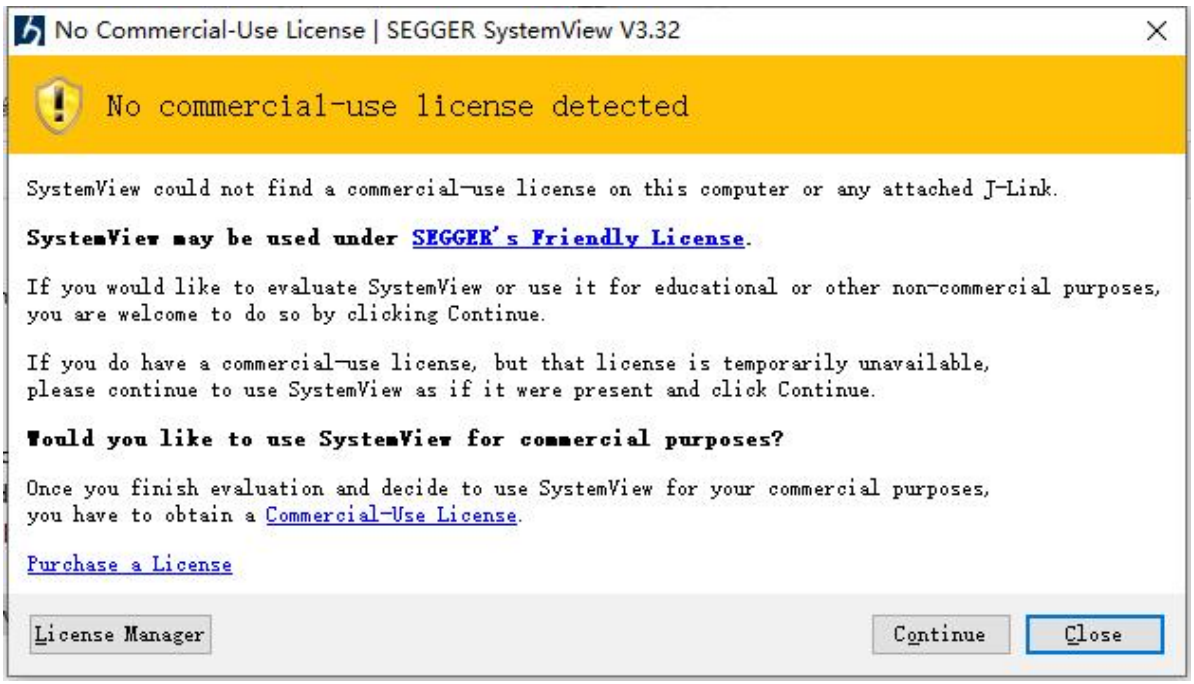
到这里就完成了在STM32上的程序移植。

使用过程

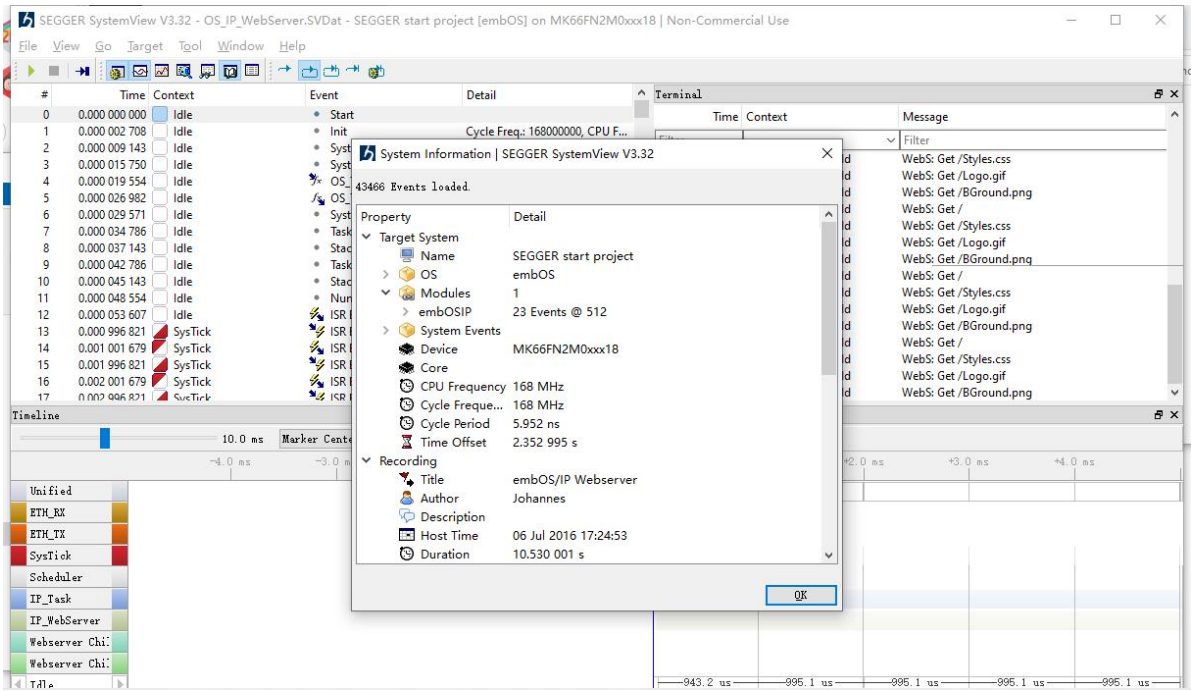
1. 启动SystemView



2. 可能会弹出警告，说没有商业许可；不管他，暂时没发现问题

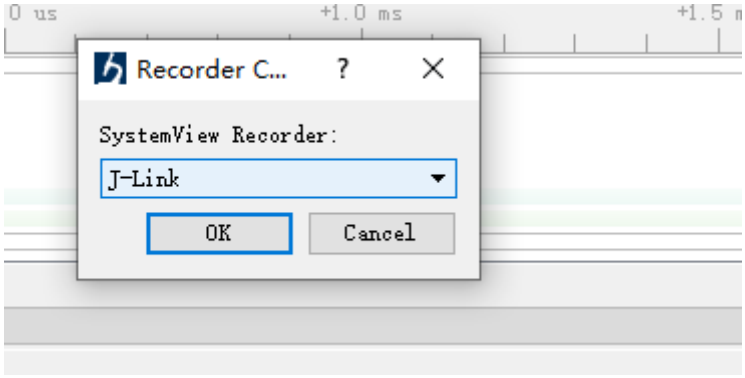


3. SystemView将加载并分析这些数据，展示加载的记录的系统信息

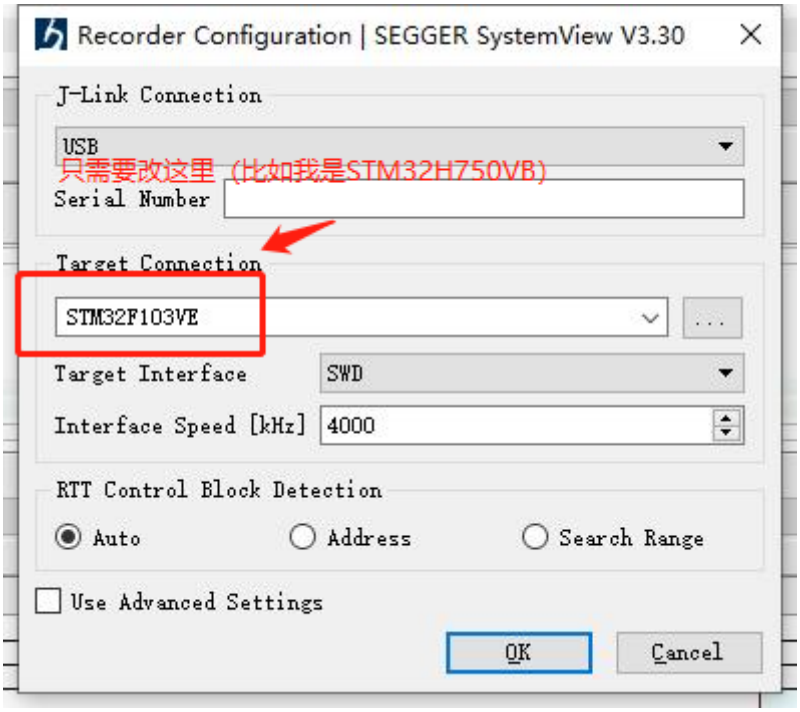


4. 点击开始记录

5. 选择JLink

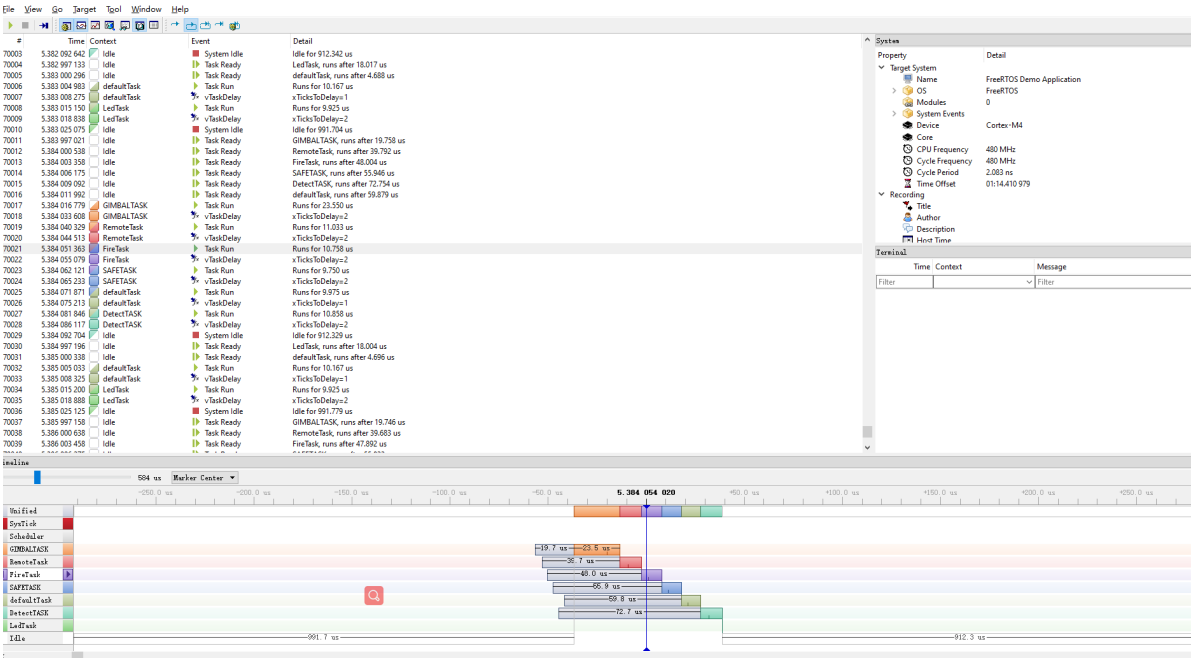


6.选择目标板的信息（这里我的以及搞过了，没弹出来，用的网图）



7.进入实时监视

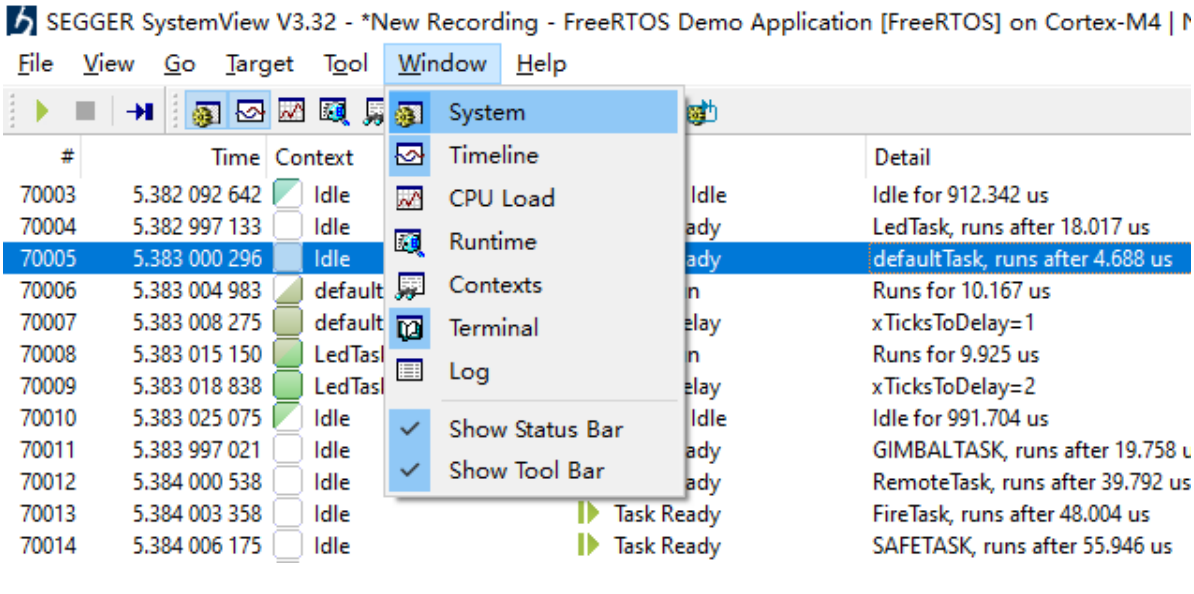
部分JLINK会因为被检测到盗版而无法使用



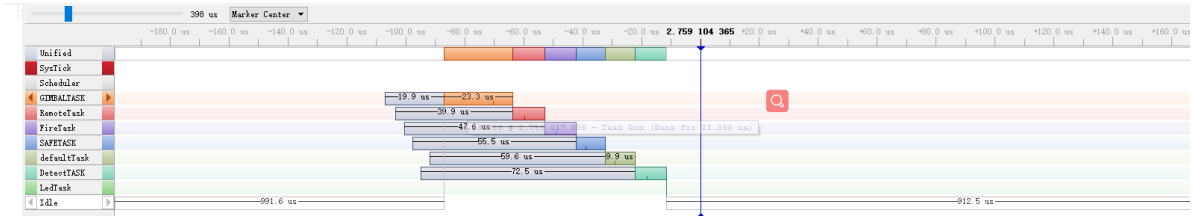
窗口介绍

总共分大概以下几种窗口：

时间轴窗口（Timeline）、事件窗口（Events）、终端窗口(Terminal)、CPU占用窗口（CPU Load）、不知道怎么命名窗口（context）



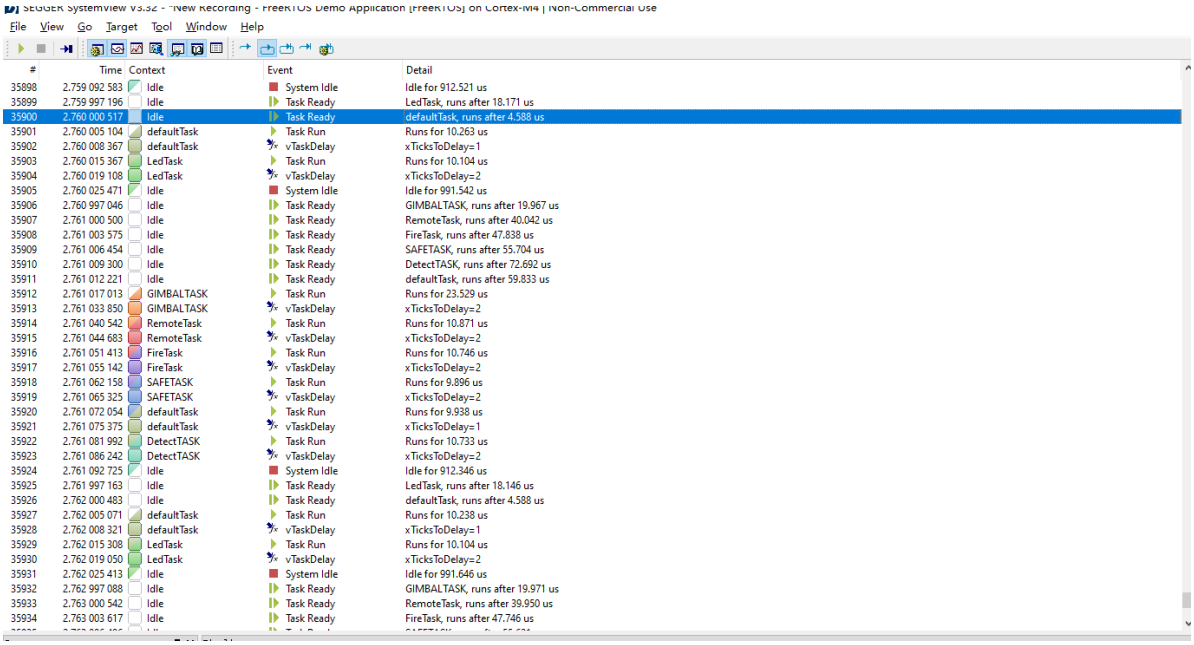
时间轴窗口（Timeline）



用于显示各任务占用的时间和逻辑顺序

- 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。

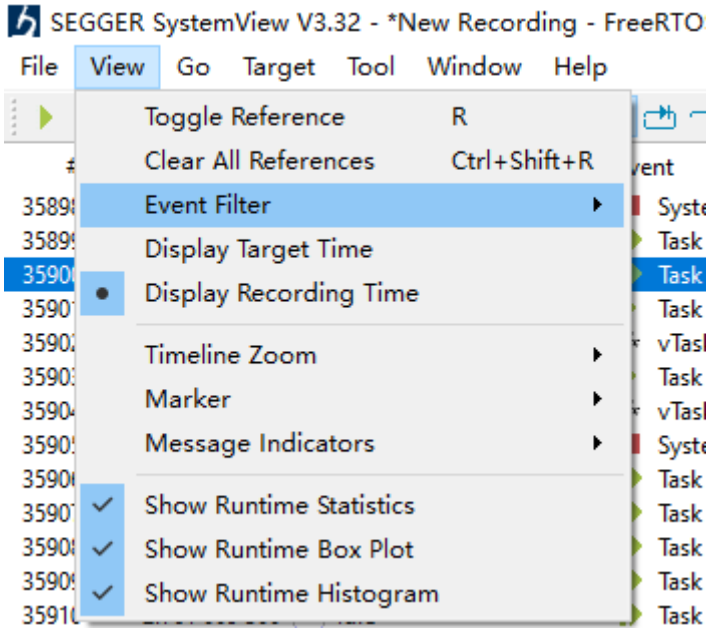
事件窗口（Events）



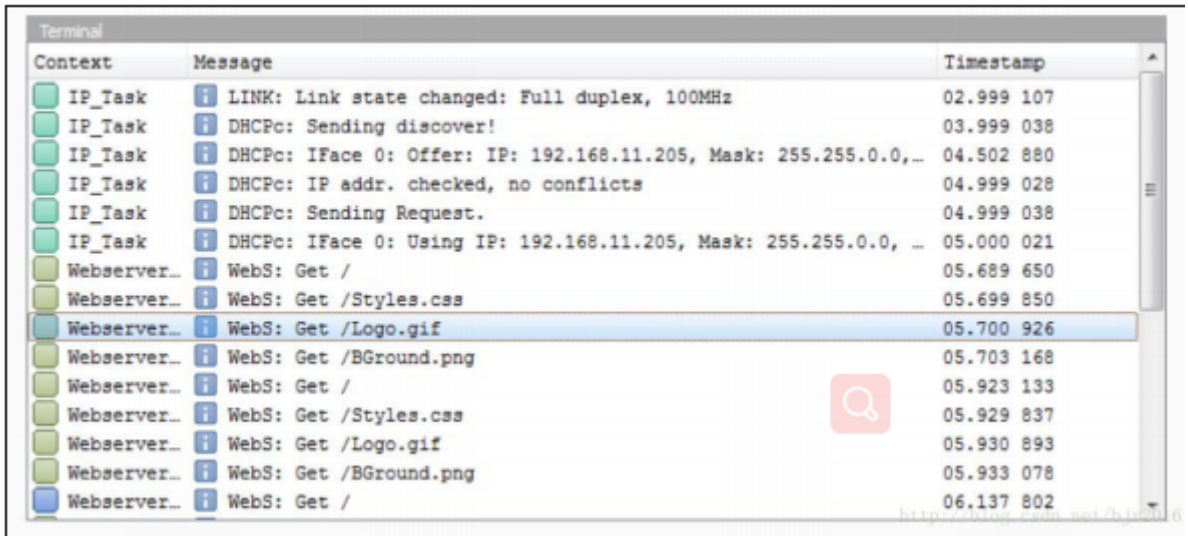
Events窗口显示系统发送的所有事件，并显示它们的信息。每个事件都有以下几项：

- 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。

也可以通过事件滤波器查看对应的事件



终端窗口



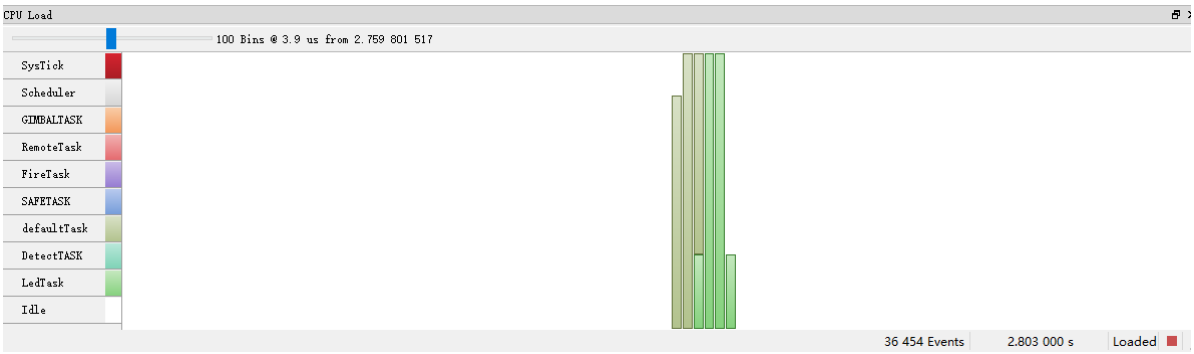
终端窗口显示来自目标应用程序的printf输出，以及发出log输出的任务上下文，以及发送消息时的时间戳。双击消息，可以显示出该消息在事件列表中的所有信息。

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。

可通过systemview的API接口，打印信息

```
/* 打印一个由SystemView格式化的字符串. */
SEGGER_SYSVIEW_PrintfHost("system tick:%u.\r\n", xTaskGetTickCount() );
```

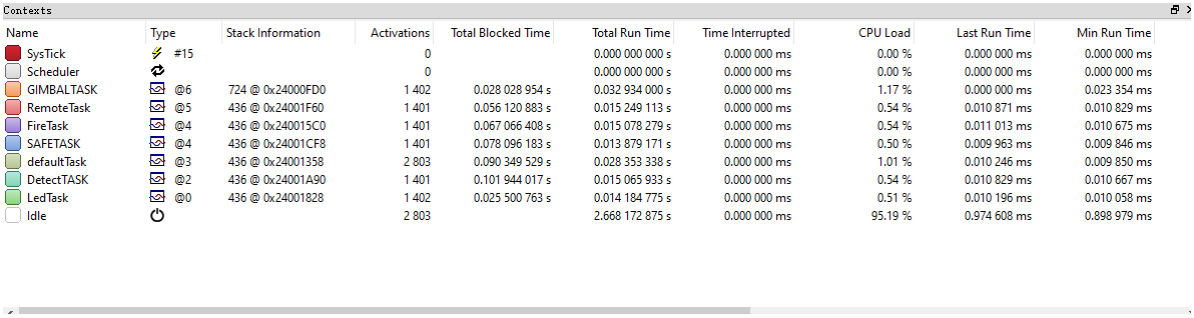
CPU占用窗口（CPU Load）



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。

Context窗口


Contexts

Name	Type	Stack Information	Activations	Total Blocked Time	Total Run Time	Time Interrupted	CPU Load	Last Run Time	Min Run Time
SysTick	#15		0		0.000 000 000 s	0.000 000 ms	0.00 %	0.000 000 ms	0.000 000 ms
Scheduler			0		0.000 000 000 s	0.000 000 ms	0.00 %	0.000 000 ms	0.000 000 ms
GIMBALTASK	@6	724 @ 0x24000FD0	1 402	0.028 028 954 s	0.032 934 000 s	0.000 000 ms	1.17 %	0.000 000 ms	0.023 354 ms
RemoteTask	@5	436 @ 0x24001F60	1 401	0.056 120 883 s	0.015 249 113 s	0.000 000 ms	0.54 %	0.010 871 ms	0.010 829 ms
FireTask	@4	436 @ 0x240015C0	1 401	0.067 066 408 s	0.015 078 279 s	0.000 000 ms	0.54 %	0.011 013 ms	0.010 675 ms
SAFETASK	@4	436 @ 0x24001CF8	1 401	0.078 096 183 s	0.013 879 171 s	0.000 000 ms	0.50 %	0.009 963 ms	0.009 846 ms
defaultTask	@3	436 @ 0x24001358	2 803	0.090 349 529 s	0.028 353 338 s	0.000 000 ms	1.01 %	0.010 246 ms	0.009 850 ms
DetectTASK	@2	436 @ 0x24001A90	1 401	0.101 944 017 s	0.015 065 933 s	0.000 000 ms	0.54 %	0.010 829 ms	0.010 667 ms
LedTask	@0	436 @ 0x24001828	1 402	0.025 500 763 s	0.014 184 775 s	0.000 000 ms	0.51 %	0.010 196 ms	0.010 058 ms
Idle			2 803		2.668 172 875 s	0.000 000 ms	95.19 %	0.974 608 ms	0.898 979 ms

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%。

其他使用

事件记录

systemview默认的事件有

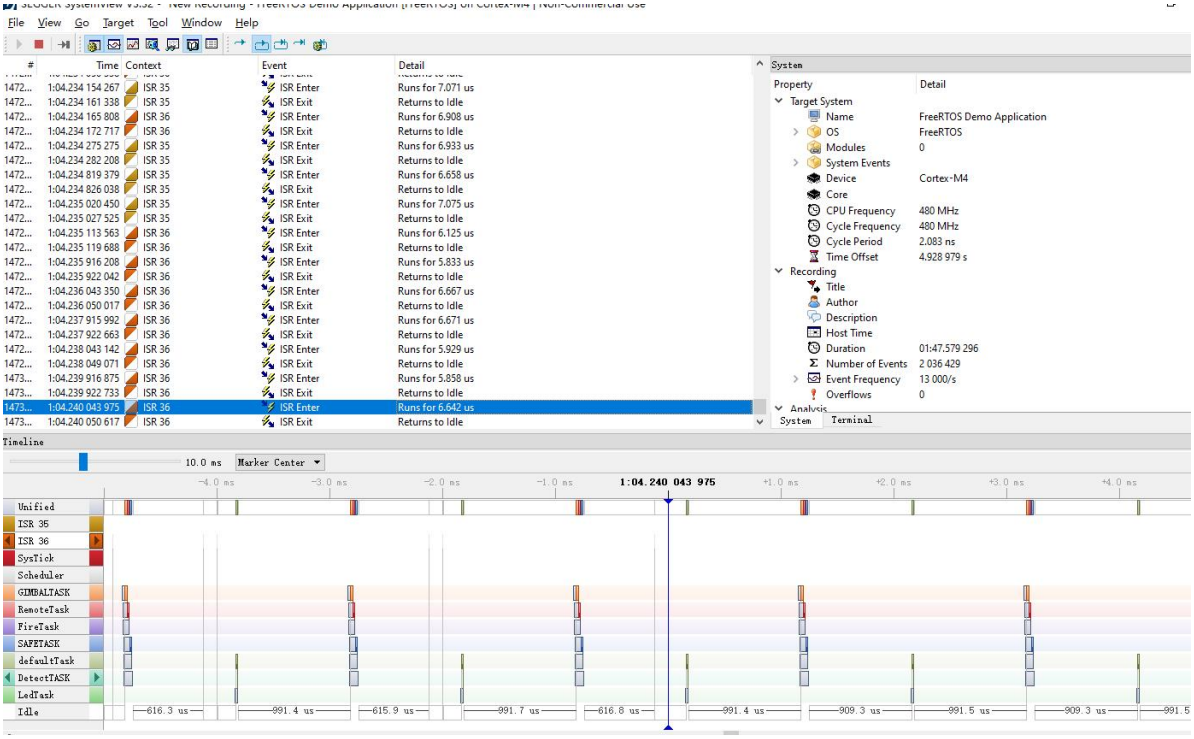
事件	描述	SystemView API
Task Create	创建了新任务	SEGGER_SYSVIEW_OnTaskCreate
Task Start Ready	任务被标记为准备好启动或者恢复执行	SEGGER_SYSVIEW_OnTaskStartReady
Task Start Exec	任务被激活（启动或者恢复执行）	SEGGER_SYSVIEW_OnTaskStart
Task Stop Ready	任务被阻塞或暂停	SEGGER_SYSVIEW_OnTaskStopReady
Task Stop Exec	任务终止	SEGGER_SYSVIEW_OnTaskStopExec
System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

添加中断记录

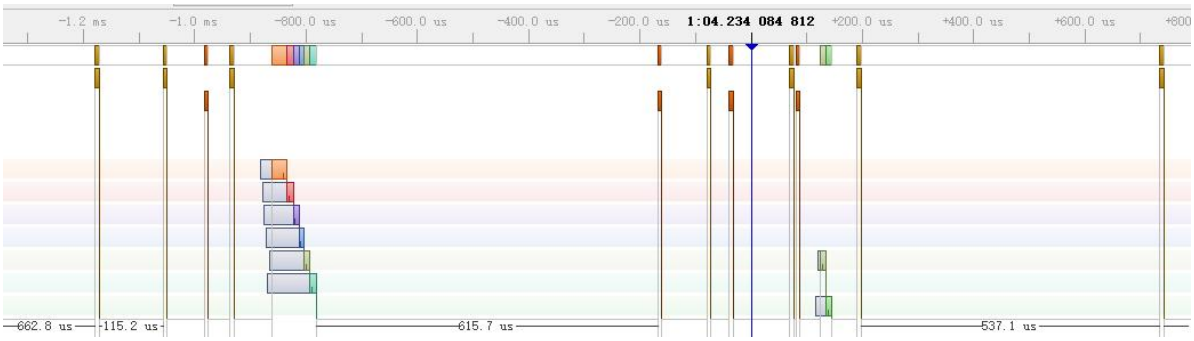
可以自己在中断函数中添加对应的API接口，来实现对中断进入的记录。

```
/**
 * @brief   HAl库can1的回调函数
 * @param   传入参数： CAN的句柄
 * @retval   none
 */
void HAL_FDCAN_RxFifo0Callback(FDCAN_HandleTypeDef *hfdcan, uint32_t RxFifo0ITs)
{

   /* 格式化并发送进入中断服务函数事件 */
  SEGGER_SYSVIEW_RecordEnterISR ();

  if (hfdcan == &hfdcan1)
  {
    if (CAN1_receive_callback != NULL)
      CAN1_receive_callback(&hfdcan1);
  }
  if (hfdcan == &hfdcan2)
    if (CAN2_receive_callback != NULL)
      CAN2_receive_callback(&hfdcan2);

    /* 格式化并发送退出中断服务函数事件 */
   SEGGER_SYSVIEW_RecordExitISR ();
}

/************************************can1接收处理函数************************************/

```

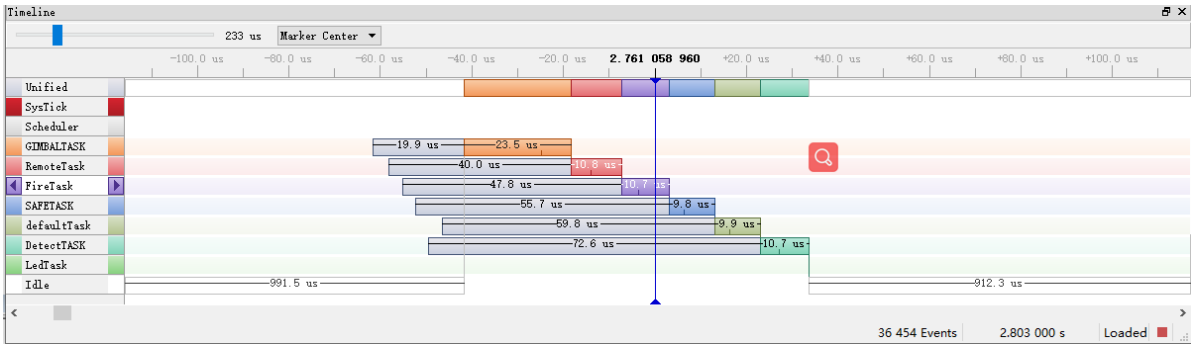
结果如图：



可以看到会定期触发can中断，且每次进入中断事件长短不同



分析示例



可以看到哨兵云台任务占用的时间最长，且可以看到我们现在的任务运行时间在整个周期中占用的都是非常短的。

存在的问题

> **问题描述：** *J-Link不同而导致被识别盗版而不能用的问题*
>
> - [x] **原因：** *J-Link被识别盗版*
> - [x] **解决：** 换用其他的J-Link仿真器，或利用串口进行收发（详见链接串口移植收发）

> **问题描述：** 出现报错：
>
> Error: L6218E: Undefined symbol SEGGER_SYSVIEW_X_GetInterruptId (referred from segger_sysview.o).
> Error: L6218E: Undefined symbol SEGGER_SYSVIEW_X_GetTimestamp (referred from segger_sysview.o).
>
> - [x] **原因：** systemview 在keil中与gnu扩展的不兼容
> - [x] **解决：** 在keil中选择不勾选GNU拓展（详见）
> - [ ] **隐患：** 其他部分代码可能会出现问题

> **拓展问题表述：** 串口 print重定义代码出现报错
>
> - [x] **原因**：因为没有了GNU拓展，导致这部分代码不可用
> - [ ] **解决：** 暂定先注释掉，待解决。

应用场景

- 用于查看每个进程的时间片，查看各个进程时间占用情况。
- 防止某个进程占用时间过长，而被操作系统强行进入堵塞状态。
- 查看中断进入情况，避免因传感器发送频率过高，而频繁进入中断。