**PLY-T32Scripts手册指南**

(V0.4)

**Pizer.Fan**

目录

[前言 2](#_Toc68091167)

[1 概述 3](#_Toc68091168)

[1.1 ”PLY-T32Scripts”能做什么？ 3](#_Toc68091169)

[1.2 ”PLY-T32Scripts”哪里下载？ 3](#_Toc68091170)

[1.3 启动 ”PLY-T32Scripts” 3](#_Toc68091171)

[1.4 配置自动启动 ”PLY-T32Scripts” 5](#_Toc68091172)

[1.5 Script list 6](#_Toc68091173)

[2 启动入口 10](#_Toc68091174)

[3 图形界面 11](#_Toc68091175)

[3.1 工具栏 11](#_Toc68091176)

[3.2 ARM菜单 11](#_Toc68091177)

[3.2.1 实例: 推导调用栈 12](#_Toc68091178)

[3.2.2 实例: 代码段是否破坏？ 13](#_Toc68091179)

[3.2.3 实例: 显示当前MPU的配置 14](#_Toc68091180)

[3.3 TX菜单 15](#_Toc68091181)

[3.3.1 实例: 显示当前所有就绪和执行状态的线程 15](#_Toc68091182)

[3.3.2 实例: 定时器是否激活？ 17](#_Toc68091183)

[3.3.3 实例: 栈溢出？ 17](#_Toc68091184)

[3.4 Task Monitor菜单 19](#_Toc68091185)

[3.5 OSA/SCI over TX菜单 20](#_Toc68091186)

[3.5.1 实例: HEAP内存破坏 20](#_Toc68091187)

[3.6 Statistics菜单 22](#_Toc68091188)

[3.7 Auto Analyze菜单 23](#_Toc68091189)

[3.8 LOG菜单 24](#_Toc68091190)

[4 自动检查是否升级 26](#_Toc68091191)

[5 后台自动化分析 27](#_Toc68091192)

# 前言

”PLY-T32Scripts”脚本套件是我发起“解放平台人”的运动之一，其目的当然是给通信等应用部门提供更开放、更易用、更便捷的分析工具链，同时解决平台人繁重也不太必要的普通支持工作。

前辈们大致提供过一些简单且单一的过程脚本，并没有成整体、成体系地提供完整的脚本工具链。”PLY-T32Scripts”脚本套件应运而生，其在设计上有着如下要求：

1. 成系统化，提供的是一个完整的工具包，包括诸如：应用栈的运行推导、消息队列的数据分析、内存泄漏的TOP排行，而基础的也包括RTOS的KERNEL OBJECT链条分析等。除此，还提供自动化分析。
2. 各个脚本解耦合，第三方可以基于此进行拼接，类似搭积木一样，完成自己想要的功能，实现再编程。

为了解耦合，一个是功能上独立，一个是制定一定的命名规则：例如\_arm\_\*用于arm芯片的基础分析、\_tx\_\*用于 THREADX的分析、\_osa\_\*用于 osa适配层的分析、\_log\_\*用于应用LOG子系统的分析、\_auto\_\*用于自动化分析，以及 \_tm\_\*用于RTOS LOG的分析。

1. 另外，”PLY-T32Scripts”脚本套件也提供人机交互（文本），希望这些交互的文本能够清晰、易懂。同时，极力极少交互的次数，减少不必要的交互。

我希望”PLY-T32Scripts”脚本套件能够帮助到平台人、通信部门等研发同事，未来也希望该套件用到offline的自动化分析当中。

# 概述

## ”PLY-T32Scripts”能做什么？

”PLY-T32Scripts”脚本套件在什么场景下使用？一般在出现ARM异常情况下，包括中断、ABORT、以及主动ASSERT 等等，而主动 ASSERT可能是“分配内存失败”、“操作RTOS API不当”等等。

”PLY-T32Scripts”脚本套件，给我们提供 ARM 芯片相关联的操作，RTOS（TX）内核对象链的分析，RTOS 的LOG分析（Task Monitor打点），OSA/SCI层面的内存调试头部信息分析，以及自动化ASSERT分析等等，常见的包括：

1. ARM异常还原，CPSR/SPSR的各个bit分析；
2. 列出RTOS层面的线程状态、信号量、互斥体、消息队列、byte pool、block pool，以及定时器等等状态。
3. RTOS的任务切换记录，中断进出记录等等。
4. 栈溢出分析
5. 列出管理内存的状态，发现可能的内存泄漏
6. 各类主动ASSERT分析，包括诸如：消息队列满、获取或者释放信号量/互斥体失败，等等。
7. RTOS适配层，对线程，信号量、互斥体、消息队列、byte pool、以及block pool的深度分析，包括消息队列中的内容，block pool的内容，以及block pool里释放过的内容，等等
8. V3 LOG编码层缓冲、SIPC传输通道内存等数据的分析和DUMP；

## ”PLY-T32Scripts”哪里下载？

内网：<http://shsvn02/svn/Tools/trunk/Tools/PLY-T32Scripts>

外网：<http://shexsvn02/svn/STRD/Dept/BasicSoftwareDesign/RTOS/DEBUG_TOOL/PLY-T32Scripts>

## 启动 ”PLY-T32Scripts”

将套件中的 \_start.cmm 、或者 \_menu\_PLY.cmm 脚本拖拽到 TRACE32 SIM的命令行中：

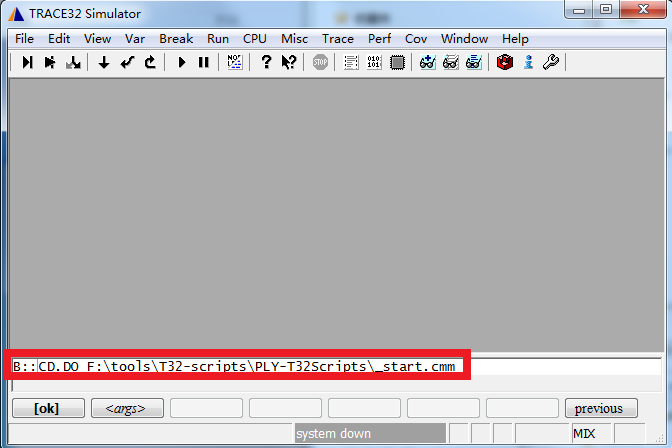


图 1 首次启动

回车一下（将会执行上述脚本），菜单将会出现THREADX、“PLY”的菜单：

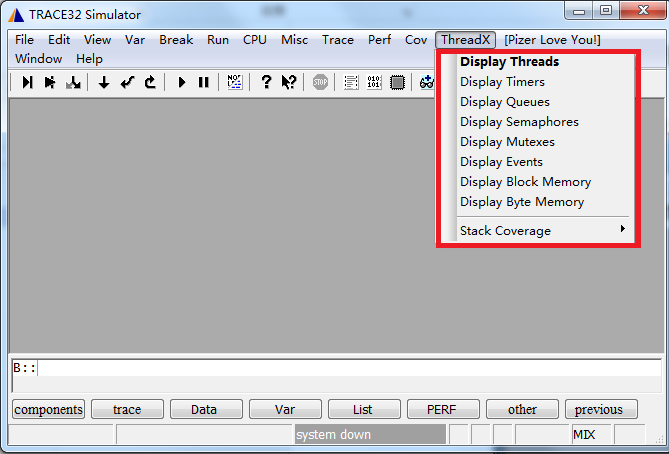


图 2 ThreadX菜单

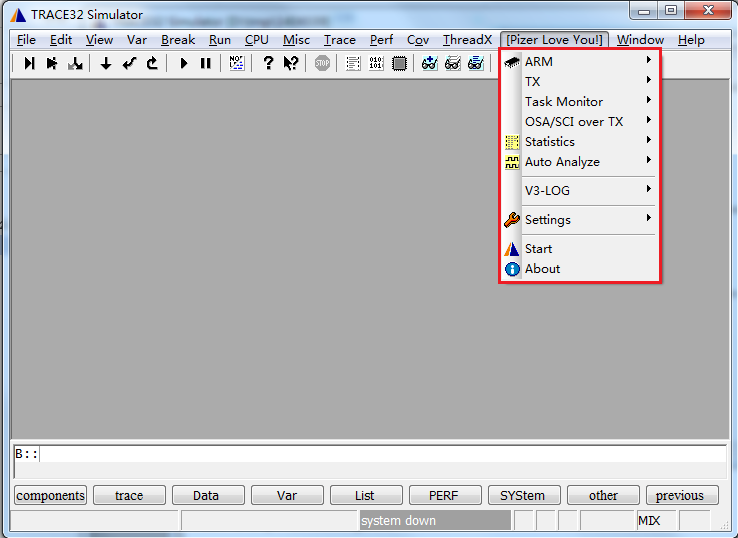


图 3 PLY菜单

细心一点，你会发现工具栏也多了一些按钮：

你可以直接运行 \_start.cmm 脚本开始你的分析，也可以点击“PLY” 菜单中的“Start”子菜单项开始你的分析之旅：根据人机交互界面，将指引你来到待分析的“问题现场“。

## 配置自动启动 ”PLY-T32Scripts”

配置TRACE32 Simulator启动时候，自动加载”PLY-T32Scripts”：

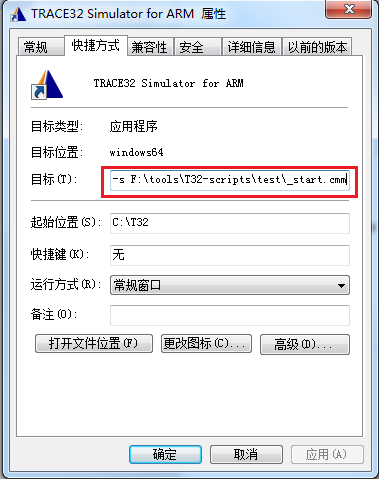
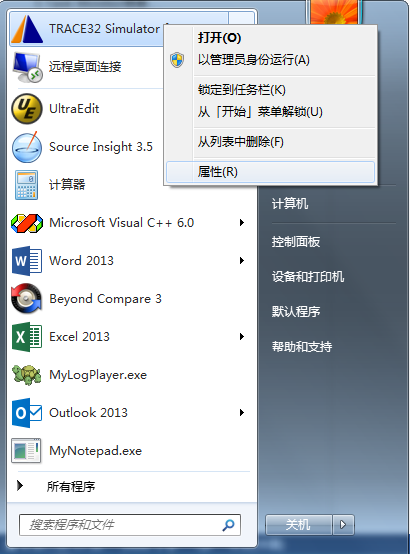


图 4 PLY菜单自动加载配置

将“目标”（即执行的命令）做一定的修改，假设先前是： C:\T32\bin\windows64\t32marm.exe -c C:\T32\configsim.t32 ，在这个基础上增加一个-s参数:

* 譬如：-s F:\tools\T32-scripts\test\\_start.cmm（假设PLY脚本\_start.cmm的路径是在F:\tools\T32-scripts\test目录），修改后的“目标”变更为：C:\T32\bin\windows64\t32marm.exe -c C:\T32\configsim.t32 -s F:\tools\T32-scripts\test\\_start.cmm
* 如果启动不希望直接开始分析，那么可以配置仅仅加载PLY-T32Script菜单，譬如：-s F:\tools\T32-scripts\test\\_menu\_PLY.cmm，修改后的“目标”变更为：C:\T32\bin\windows64\t32marm.exe -c C:\T32\configsim.t32 -s F:\tools\T32-scripts\test\\_menu\_PLY.cmm

## Script list

Script list:

Start

The starting point to analyze.

\_start.cmm

This script is used to startup analyzing.

Supported products:

\_product\_cfg\_cpu\_orca.cmm

\_product\_cfg\_cpu\_sharkle.cmm

\_product\_cfg\_cpu\_w317.cmm

ARM

The submenus related to ARM, such as recovering registers.

\_arm\_recover\_stack.cmm

This script is used to recover the registers from the memory variables.

It also try back to the previous mode when exception is data abort.

\_arm\_recover\_v3\_stack.cmm

This script is used to recover the registers from the memory variables, and only used on the V3 modem.

It also try back to the previous mode when exception is data abort.

\_arm\_show\_core\_state.cmm

This script is used to show the core state (such as: running threads)

\_arm\_analyze\_exception\_stacks.cmm

This script is used to analyze the stack frames of exception modes.

\_arm\_analyze\_app\_stack.cmm

This script is used to analyze the stack frame of application thread.

\_arm\_dump\_code\_regions.cmm

This script is used to dump the code regions (execution) from memory into host file.

Sometimes, the code region in DDR (not flash) was demaged by another master, so let's

dump and compare it with the binary file for flash.

Common

Modem version, Modem assert information, etc.

TX

Kernel object information, such as mutexes, semaphores, timers, queues, etc.

\_tx\_export\_mutex.cmm

This script is used to export all the mutexes into the host file, it also check the link between them.

\_tx\_export\_sem.cmm

This script is used to export all the semaphores into the host file, it also check the link between them.

\_tx\_export\_event\_flags.cmm

This script is used to export all the event flags into the host file, it also check the link between them.

\_tx\_export\_queue.cmm

This script is used to export all the message queues into the host file, it also check the link between them.

\_tx\_export\_timer.cmm

This script is used to export all the timers into the host file, it also check the link between them.

Task Monitor

The kernel log data, such as IDLE, TASK, ENTER IRQ, etc.

\_tm\_show\_version.cmm

This script is used to show the version of the task monitor

\_tm\_export\_data.cmm

This script is used to export the task monitor buffer into the HOST file.

OSA/SCI over TX

Kernal adapter (middle-ware) information, especially memory allocations.

\_sci\_tx\_export\_signal\_queue\_data.cmm

This script is used to export the signals in the queue into the host file.

\_sci\_tx\_export\_mem.cmm

This script is used to export the allocations into the host file, only for SCI allocations.

\_osa\_tx\_export\_mem.cmm

This script is used to export the allocations into the host file, for the allocations based on osa, such as SCI-osa/osa/SDI-osa.

\_osa\_tx\_show\_mem\_top\_5.cmm

This script is used to show the top 5 of the allocations based on osa, it requires the python application.

\_sdi\_tx\_export\_msg\_queue\_data.cmm

This script is used to export the messages in the queue into the host file.

LOG

The application log data.

The following scripts are used to analyze the APP-LOG subsystem, and dump the SIPC LOG buffer.

\_log\_chan\_buf\_list.cmm

\_log\_chan\_buf\_tracks.cmm

\_log\_pool\_shmem\_list.cmm

\_log\_ring\_shmem\_list.cmm

\_log\_last\_one\_frame.cmm

Statistics

The Statistics of system, ... etc

\_stat\_system\_statistics.cmm

Auto Analyze

Auto analyze scripts.

The following scripts are used to auto analyze.

\_auto\_analyze\_assert.cmm

This script is used to auto analyze assert, and it depends to:

\_assert\_is\_assert\_mode.cmm

\_assert\_get\_assert\_info.cmm

\_assert\_analyze\_assert\_info.cmm

\_assert\_analyze\_assert\_info.cmm"

\_assert\_analyze\_assert\_info\_case\_no\_assert\_info.cmm

\_assert\_analyze\_assert\_info\_case\_osa\_threadx\_error\_handling\_exp\_osa\_abort.cmm

\_assert\_analyze\_assert\_info\_case\_osa\_threadx\_itc\_take\_mutex\_failed.cmm

\_assert\_analyze\_assert\_info\_case\_osa\_threadx\_itc\_take\_sem\_failed.cmm

\_assert\_analyze\_assert\_info\_case\_sdi\_msg\_c\_enqueue\_failed.cmm

\_assert\_analyze\_assert\_info\_case\_threadx\_assert\_c\_exp\_abort\_except.cmm

\_assert\_analyze\_assert\_info\_case\_threadx\_assert\_c\_file\_exp\_0.cmm

\_assert\_analyze\_assert\_info\_case\_others.cmm

\_assert\_get\_abort\_info.cmm

Settings

The settings scripts.

\_setting\_show\_export\_dir.cmm

This script is used to show the exporting directory.

\_setting\_input\_export\_dir.cmm

This script is used to input the exporting directory.

Basic

The basic scripts.

\_trace32\_area\_delete.cmm

\_trace32\_data\_list.cmm

\_trace32\_print\_var\_frame.cmm

\_trace32\_symbol\_function.cmm

\_trace32\_symbol\_info.cmm

\_trace32\_var\_frame.cmm

\_trace32\_var\_frame\_ex.cmm

# 启动入口

前面提到：你可以直接运行 \_start.cmm 脚本开始你的分析，也可以点击“PLY” 菜单中的“Start”子菜单项开始你的分析之旅。

\_start.cmm 或者 Start 菜单，就像是 C 语言中 main 函数，是一个启动的开始入口。脚本系统将指引你选择 product，因为不同的 product 有着不同的 memory layout，如果错误，那么内存文件内容将会被加载到可能不正确的内存中，从而导致你看到可能是错误的内容，例如：

* 像 sharkle项目，内存文件存储的是目标某个起始内存之后的连续内容。
* 想 ORCA项目，内存文件可能存储的是目标某个起始内存之后的连续内容，也可能是不同内存段的数据——这要求多次分段加载到内存当中去。

实际上，对我们的 DUMP 机制的设计是有更高的要求的：

* 例如设计格式内存文件，其头部记录分段信息，这样有分段的信息，避免用户可能做出错误的选择。
* 而现状的裸内存文件，则需要用户输入起始内存地址，如果是多段则需要输入知道每一个段的起始地址

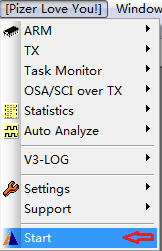


图 5 启动入口菜单



图 6 工具栏中的启动入口

# 图形界面

## 工具栏

PLY工具栏：

: 等效 “Auto Analyze Assert”菜单

: 等效 “Start”菜单

: 等效 “About”菜单

## ARM菜单

ARM菜单功能，提供ARM芯片息息相关的功能，例如寄存器的现场恢复，异常模式，异常模式下栈空间分析，用户栈空间分析等等。

如果启动入口无法找到对应的产品的时候，怎么办？

首先手工完成启动部分：

sys.cpu XXXXX

sys.up

d.load \*.axf

d.load.binary \*.mem <address> /noclear

然后操作 ARM 菜单，进行现场恢复和分析。

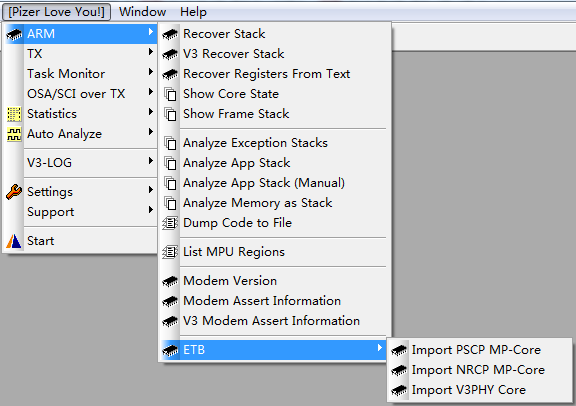


图 7 ARM菜单

**“Recover Stack”** : 从内存恢复到复现场的模式（寄存器）。

**“V3 Recover Stack”** : 从内存恢复到复现场的模式（寄存器），限V3（ORCA）。

**“Recover Registers From Text”** : 用于从文本中，提取Registers值，快速恢复现场——所有寄存器信息：CPSR/SPSR/R0～R12/R13(SP)/R14(LR)/R15(PC)。

含有寄存器的文本信息，可以从.ass文件中获取，也可以是Register窗口的文本拷贝（To Clipboard）。

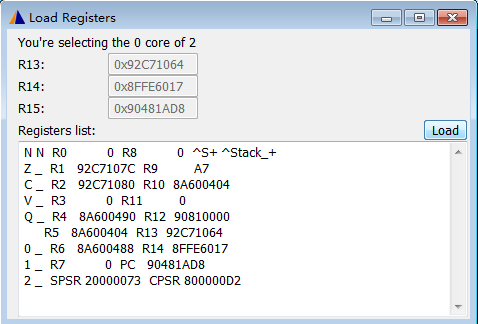


图 8 Recover Registers From Text

**“Show Core State”** : 显示当前子系统的CORE的基本状态，例如CPSR，以及运行的现场。

**“Show Frame Stack”** : 显示当前子系统运行的现场。

**“Analyze Exception Stacks”** : 分析ARM各个异常模式下STACK的调用情况。

**“Analyze App Stack”** : 分析指定线程的STACK的调用情况。

**“Analyze Memory as Stack”** : 分析指定内存作为STACK的调用情况，主要分析可能的STACK OVERFLOW造成对目标内存的覆盖分析。

**“Dump Code to File”** : 将内存的CODE段导出到HOST 文件，通过与原始映像的BIN进行对比，可以分析是否存在CODE段被修改的可能情况。

**“List MPU Regions”** : 显示当前MPU的配置。

**“Modem Version”** : 用于获得Modem制作的版本信息，和制作的时间。也可以通过时间，对比ASS文件的中时间，用于判断当前的AXF和MEMORY是否匹配。

**“Modem Assert Information”** : 获取ASSERT 模式的信息。

**“V3 Modem Assert Information”** : 获取ASSERT 模式的信息，限V3（ORCA）。

**“ETB”** : 用于分析ETB数据，限V3（ORCA）。

### 实例: 推导调用栈

首先需要知道待推导调用栈的所属线程，执行菜单：【PLY】>>【TX】>>【Export Thread to File】

===============Ready Thread Info===============

No, Addr, State, Prio, Runcount, Name

8 8274DE8C READY 248 21940 MTA Task

20 8274EB70 READY 249 30624 DSP\_LOG\_Task

48 8274CCF8 READY 180 3184 T\_TFT\_HAL\_MAIN\_HANDLER

83 82752608 EXECUTING 13 60881 LGRANT Task

===============Thread Info===============

No, Addr, State, Prio, Runcount, Name

1 81C86880 SUSPENDED 2 10969 System Timer Thread

2 8274D784 QUEUE\_SUSP 9 1 RTOS\_Manage

3 8274D8B0 SEMAPHORE\_SUSP 0 122893 hisr0

4 8274D9DC SEMAPHORE\_SUSP 1 1441 hisr1

5 8274DB08 SEMAPHORE\_SUSP 2 188 hisr2

6 8274DC34 SEMAPHORE\_SUSP 1 1 mbox\_send\_thread

…

例如选择“hisr0”，其线程控制块地址：0x8274D8B0，然后执行菜单：【PLY】>>【ARM】>>【Analyze App Stack (Manual)】，输入 0x8274D8B0地址之后，将弹出Analyze App Stack对话框（如下）：

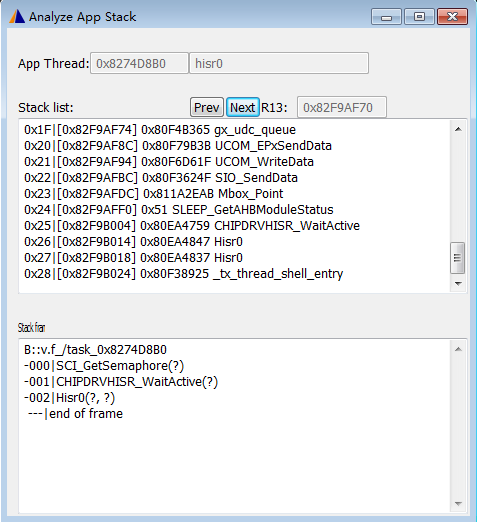


图 9实例: 推导调用栈

只需要点击 “Next” ，并观察 Stack Frame 的变化，是否可以back trace到栈底位置的线程entry？

### 实例: 代码段是否破坏？

执行菜单：【PLY】>>【ARM】>>【Dump Code to File】，如果成功则会自动打开导出的bin文件：

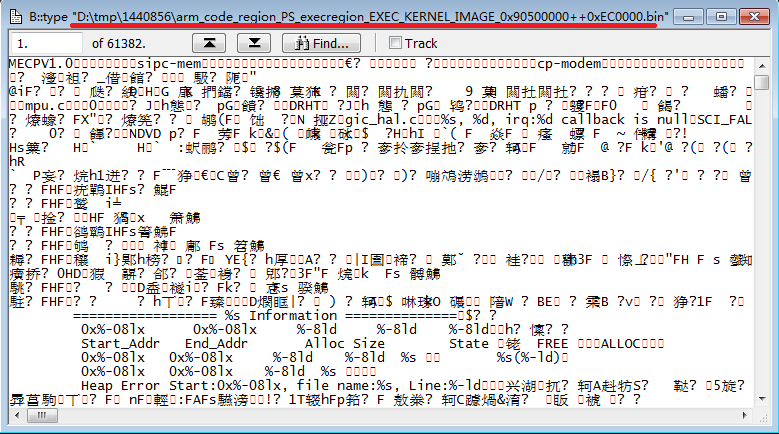


图 10 保存代码段到文件

通过比较工具（例如beyond compare）对比该bin文件和构建版本时候输出的bin文件:

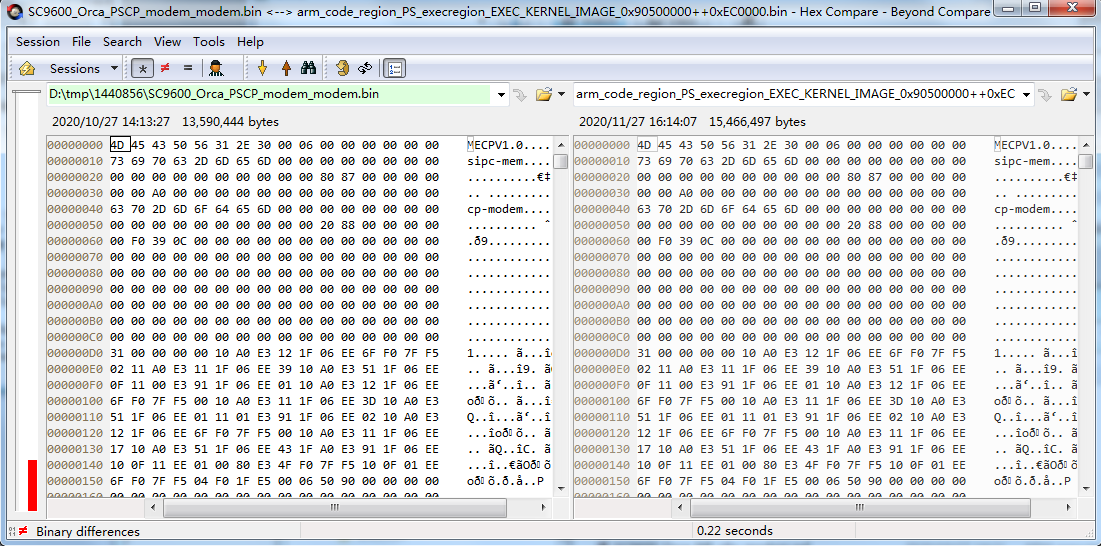


图 11 对比运行前后的代码段

### 实例: 显示当前MPU的配置

PLY MENU>>ARM>> List MPU Regions: 可以显示MPU的配置，用于查看诸如指定内存access permission——只读/读写/可执行；以及内存的attr——是否cachable，bufferable，以及cache miss策略诸如write back/write allocate等。

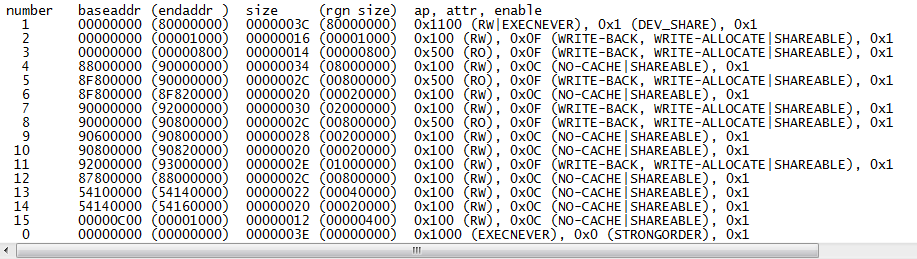


图 12显示当前MPU的配置

## TX菜单

TX菜单功能，提供THREADX操作系统之上的各种分析，包括各个KERNEL OBJECT链条的分析，例如链条是否断了或者破坏了？链条上的各个数据是否有异常？又例如有哪些活动的应用定时器，分配的内存等等？

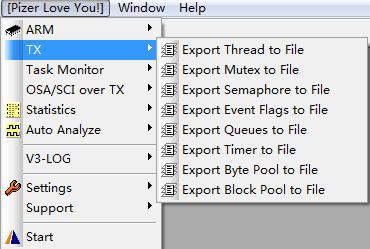


图 13 TX菜单

**“Export Thread to File”** : 将系统的各个线程信息导出到 HOST 文件，并显示。

**“Export Mutex to File”** : 将系统的各个互斥体信息导出到 HOST 文件，并显示。

**“Export Semaphores to File”** : 将系统的各个信号量信息导出到 HOST 文件，并显示。

**“Export Event Flags to File”** : 将系统的各个事件组导出到 HOST 文件，并显示。

**“Export Queues to File”** : 将系统的各个消息队列信息导出到 HOST 文件，并显示。其中包括单独显示有数据的消息队列，以及可能的队列满的情况。

**“Export Timer to File”** : 将系统的各个定时器导出到 HOST 文件，并显示。其中包括单独显示正在工作的定时器。

**“Export Byte Pool to File”** : 将系统的各个字节内存池导出到 HOST 文件，并显示。

**“Export Block Pool to File”** : 将系统的各个块内存池导出到 HOST 文件，并显示。

### 实例: 显示当前所有就绪和执行状态的线程

执行菜单：【PLY】>>【TX】>>【Export Thread to File】，例如单CPU子系统中，如下可以看有3个就绪状态的线程，以及一个正在执行的线程：

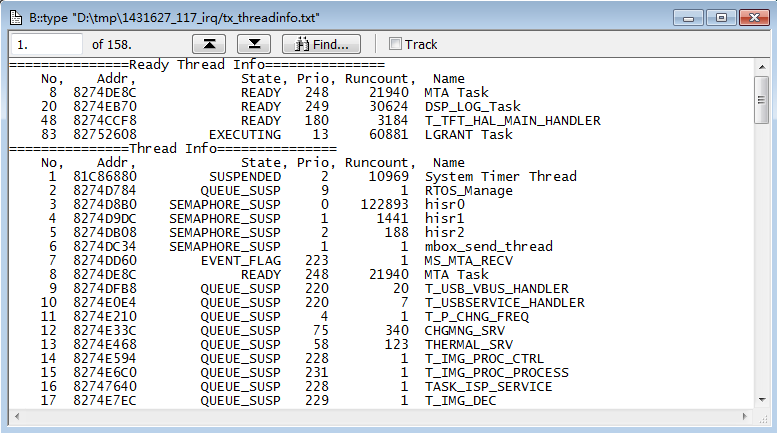


图 14 单CPU子系统中就绪和执行状态的线程

例如双CPU子系统中，如下可以看有1个就绪状态的线程，以及一个正在执行CORE 0 的线程，那么另外一个核心（CORE 1）在做什么？由于存在就绪状态的线程，那么推测CORE 1处于中断异常模式：

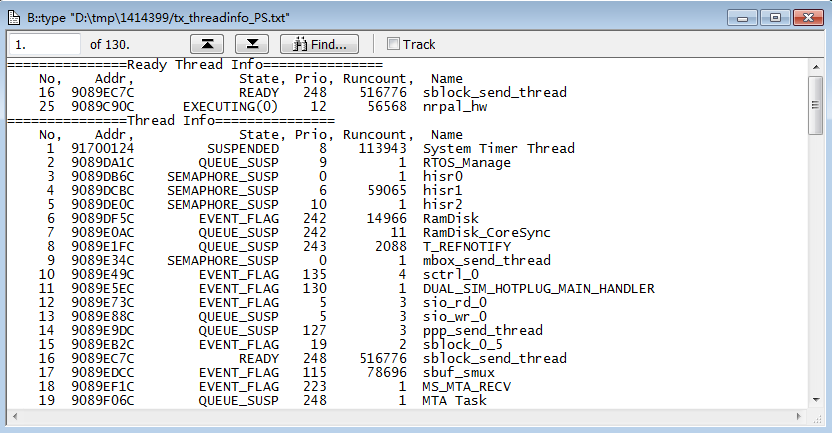


图 15 双CPU子系统中就绪和执行状态的线程

若想确定，查看CORE的状态，执行菜单：【PLY】>>【ARM】>>【Show Core State】，如下显示就绪的sblock\_send\_thread线程所在的CORE 1被中断:

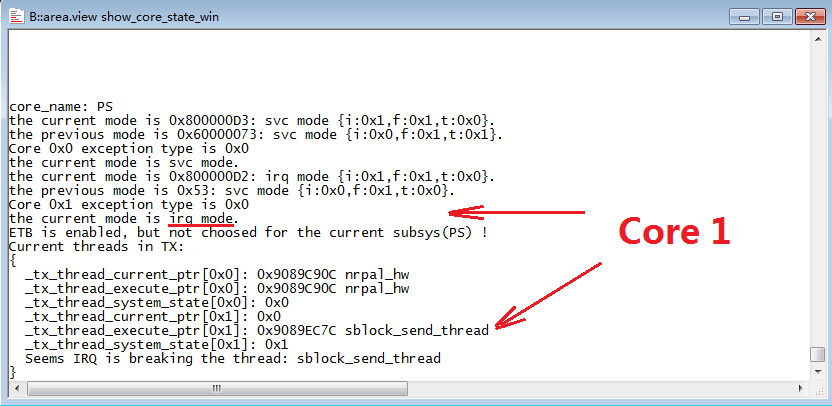


图 16 查询CORE的状态

### 实例: 定时器是否激活？

执行菜单：【PLY】>>【TX】>>【Export Timer to File】，可以获取所有正在执行状态的定时器，通过匹配定时器的名称，或者操作句柄（地址），可以知道目标定时器是否激活：

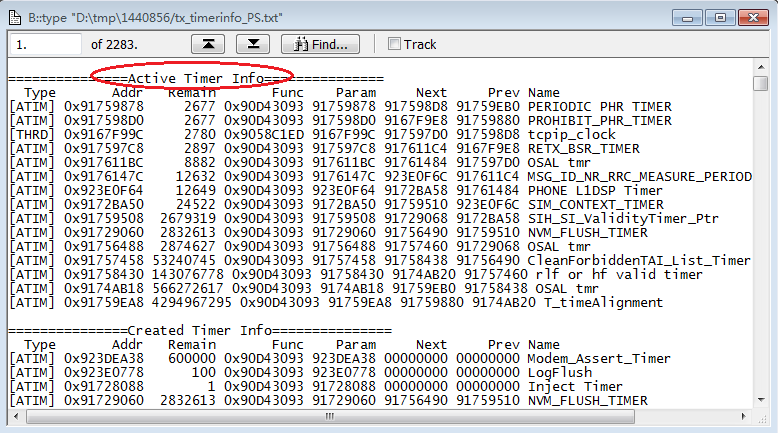


图 17查询定时器是否激活

### 实例: 栈溢出？

执行菜单：【PLY】>>【TX】>>【Analyze Threads Stack Overflow】，可以知道是否有线程存在栈溢出：

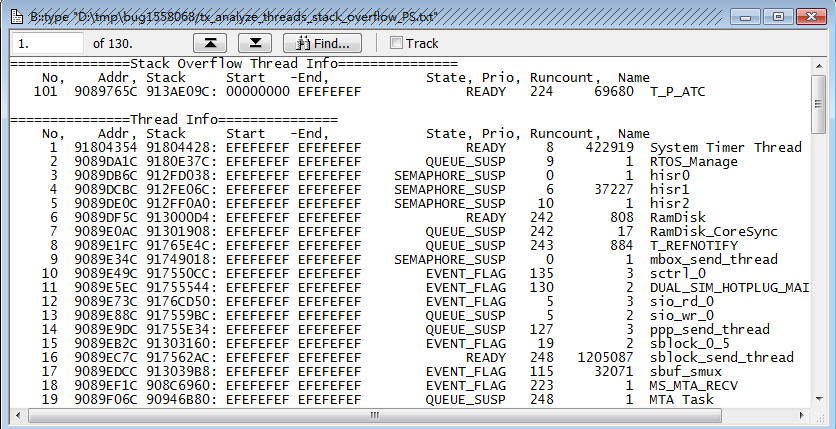


图 18查询是否存在栈溢出？

例如某BUGZILLA BUG分析，发现线程T\_P\_ATC存在栈溢出（Start：栈顶；End：为栈底；其中栈顶防破坏标记被修改）：

===============Stack Overflow Thread Info===============

No,    Addr, Stack     Start   -End, State, Prio, Runcount,  Name

101  9089765C 913AE09C: 00000000 EFEFEFEF   READY 224   69680  T\_P\_ATC

如果我们想进一步分析破坏的现场信息，从上述我们得到栈顶位置：913AE09C，那么可以分析该位置附近，诸如：0x913AB630++0x10000

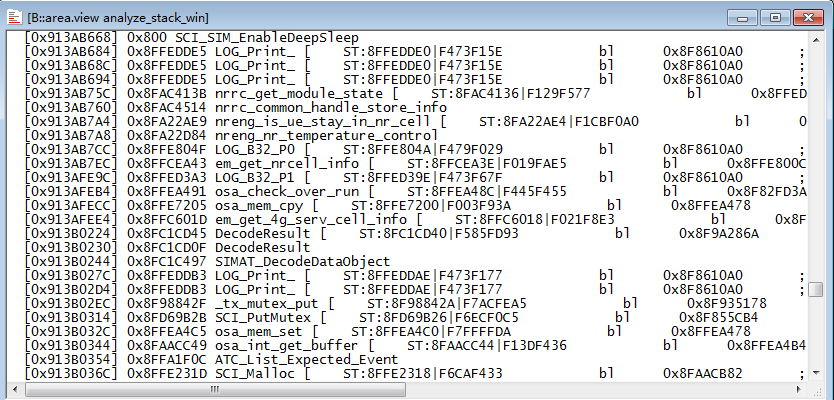


图 19分析数据栈溢出现场

Please enter the address of stack: 913AB630

Please enter the length of stack: 0x10000

{

  [0x913AB64C] 0x800 SCI\_SIM\_EnableDeepSleep

  [0x913AB668] 0x800 SCI\_SIM\_EnableDeepSleep

  [0x913AB684] 0x8FFEDDE5 LOG\_Print\_ [   bl      0x8F8610A0       ; mta\_chec

  [0x913AB68C] 0x8FFEDDE5 LOG\_Print\_ [   bl      0x8F8610A0       ; mta\_chec

  [0x913AB694] 0x8FFEDDE5 LOG\_Print\_ [   bl      0x8F8610A0       ; mta\_chec

  [0x913AB75C] 0x8FAC413B nrrc\_get\_module\_state [   bl      0x8FFEDC28

  [0x913AB760] 0x8FAC4514 nrrc\_common\_handle\_store\_info

  [0x913AB7A4] 0x8FA22AE9 nreng\_is\_ue\_stay\_in\_nr\_cell [   bl      0x8FFEDC28

  [0x913AB7A8] 0x8FA22D84 nreng\_nr\_temperature\_control

…

  [0x913B053C] 0x8F85FEBF sdi\_msg\_send\_and\_trace\_ex9 [   bl      0x8F85FDBA

  [0x913B0574] 0x8F855BE5 sdi\_msg\_send [   bl      0x8F85FE96       ; sdi\_ms

  [0x913B058C] 0x805 SCI\_SIM\_EnableDeepSleep

  [0x913B0594] 0x8FAACE87 osa\_tx\_free\_buff [   bl      0x8F830C78       ; \_t

  [0x913B05A4] 0x8FAACDC7 osa\_int\_release\_buffer [   bl      0x8FAACE56

  [0x913B05C4] 0x8FFE4F6B sdi\_blk\_mem\_int\_free [   bl      0x8FAACC58

  [0x913B05CC] 0x8FFD852D mnrsf\_crsm\_callback\_function

  [0x913B060C] 0x8FFA1EBF ATC\_List\_Expected\_Event [   bl      0x8FFE92E2

  [0x913B0614] 0x8FFA1F7C ATC\_List\_Expected\_Event

  [0x913B064C] 0x8FFA2067 ATC\_Add\_Expected\_Event [   bl      0x8FFA1E20

  [0x913B0674] 0x8FC0456D ATC\_ProcessCRSM [   bl      0x8FFA1F94       ; ATC

## Task Monitor菜单

Task Monitor菜单功能，提供RTOS 内核LOG的数据分析，俗称TM打点。我们可以获取任务的调度时间点、中断的进出时间点、以及应用的测试打点，从而分析大致的流程。TM打点的另外一个功能就是分析 CPU loading。

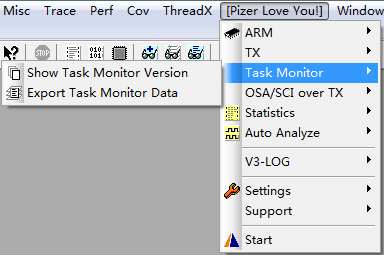


图 20 Task Monitor菜单

**“Show Task Monitor Version”** ：显示手机/目标中Task Monitor 的版本，例如：TASKMONITORBEGIN、TASKMONITORBEGIN2.00、TASKMONITORBEGIN2.10、TASKMONITORBEGIN2.20等等，还包括诸如TM的过滤设置值。

*Task Monitor Version: TASKMONITORV2.20*

*Cpu Number : 2*

*Blocks Counter : 7*

*Block Data Size : 4096*

*The TM switch settings:*

*tm\_pt\_mask=0x3F, tm\_pt\_bkgnd\_mask=0x3F*

*IDLE:Enabled, TASK:Enabled, IRQ:Enabled, SLEEP:Enabled*

**“Export Task Monitor Data”** ：导出当前class（subsystem）的Task Monitor 打点数据。

## OSA/SCI over TX菜单

OSA/SCI over TX菜单功能，主要是给 SCI 接口、OSA接口、SDI接口等提供分析，例如SCI SIGNAL队列中有哪些 SIGNAL？SDI MESSAGE有哪些MESSAGE？内存泄漏分析中有哪些内存分配（位置信息），以及内存分配的TOP排行以定位可能的内存泄漏位置。

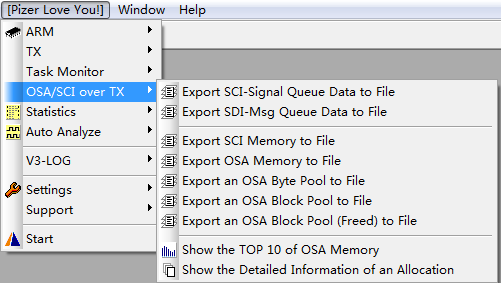


图 21 OSA/SCI over TX菜单

**“Export SCI-Signal Queue Data to File”** ：导出SCI Signal消息的队列数据。譬如你想知道队列里面有哪些消息？

**“Export SDI-Msg Queue Data to File”**：导出SDI message消息的队列数据。譬如你想知道队列里面有哪些消息，诸如消息ID、发送者模块、目标模块，以及消息其它内容等等？

**“Export SCI Memory to File”**：导出当前分配的内存信息到文件，基于TX/SCI版本。

**“Export OSA Memory to File”**：导出当前分配的内存信息到文件，基于TX/OSA/SCI(based on OSA)版本。

**“Export an OSA Byte Pool to File”**：导出指定BYTE内存池的分配信息到文件，基于TX/OSA版本。

**“Export an OSA Block Pool to File”**：导出指定block内存池的分配信息到文件，基于TX/OSA版本。

**“Export an OSA Block Pool (Freed) to File”** ：导出指定block内存池的曾经分配（但现场已经处于释放状态）信息到文件，基于TX/osa版本。

**“Show the TOP 10 of OSA Memory”**：显示分配中以分配位置为关键字的TOP 10排行，用于查看可能的内存泄漏。

**“Show the Detailed Information of an Allocation”**：显示指定内存的信息，例如分配位置、size，甚至释放的位置信息。

### 实例: HEAP内存破坏

HEAP内存破坏往往是对当前内存越界写，或者临近内存的越界写所致，所以一般情况下，需要对当前内存的长度和数据信息进行分析，同时也需要对临近的内存（或者分配状态或者释放状态）进行一个分析（主要是对其长度和数据信息进行分析）。

通过**“Export OSA Memory to File”**在dump既有的分配内存时候，可能发现破坏的内存。如果你已经知道某一块内存的地址，那么可以通过**“Show the Detailed Information of an Allocation”**获取其相关信息——属于哪个内存池？当然你也可以逐一对比内存池的范围，从而获取其所属的内存池。

例如0x8C8F7C4C内存发现被破坏，通过PLY MENU>>OSA/SCI over TX **>>“Show the Detailed Information of an Allocation”**：

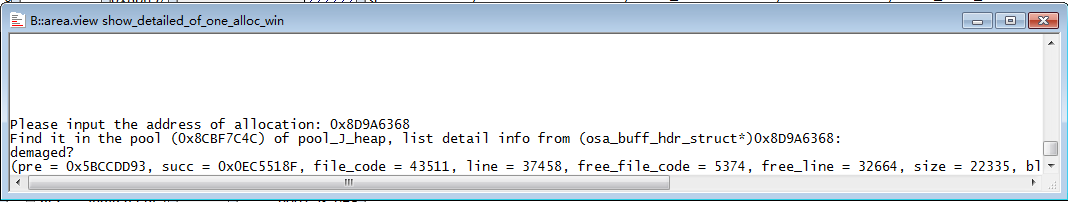


图 22 获取指定内存（分配状态）的信息

进而分析目标内存池“pool\_J\_heap”（是块内存），通过PLY MENU>>TX>>**“Export Block Pool to File”** :

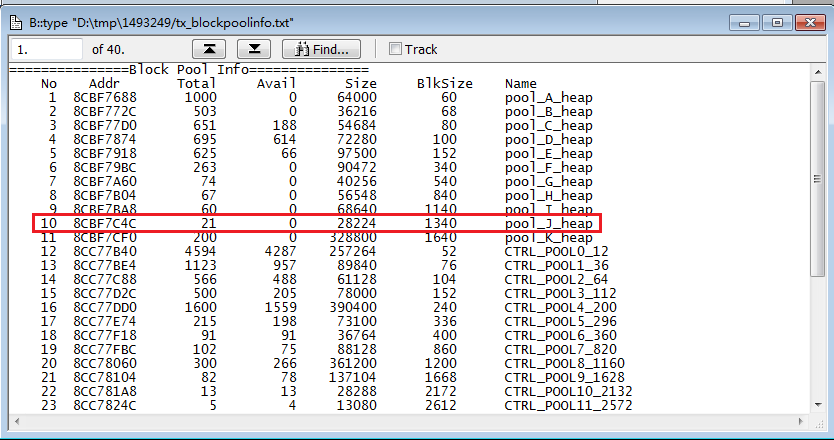


图 23 获取所有块内存池的信息

我们可以得到目标内存池“pool\_J\_heap”的Pool Addr：0x8CBF7C4C。进而可以详细分析该内存的相关数据信息，可以通过PLY MENU>> OSA/SCI over TX>>**“Export an OSA Block Pool to File”**：

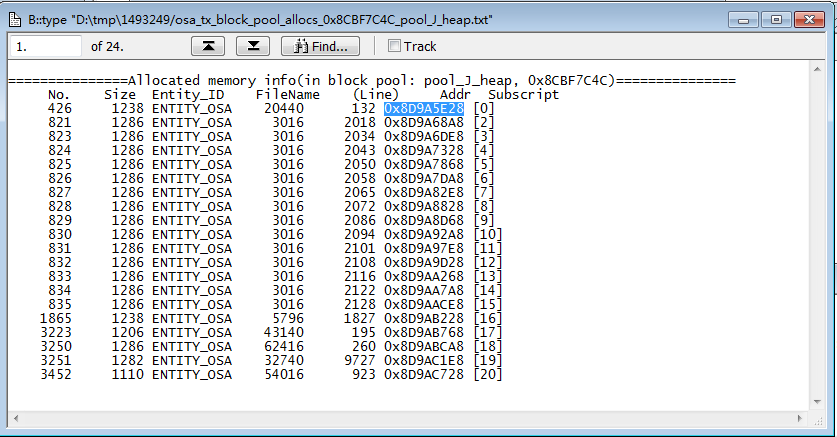


图 24 指定内存池的数据信息

我们发现前面的分配内存：0x8D9A5E28，通过PLY MENU>>OSA/SCI over TX **>>“Show the Detailed Information of an Allocation”**可以查看详细.

## Statistics菜单

Statistics菜单功能，用于统计系统的各种信息，可以是静态的，可以是运行时的。统计的信息可以是RTOS相关，也可以是框架的其它模块或者应用的信息。

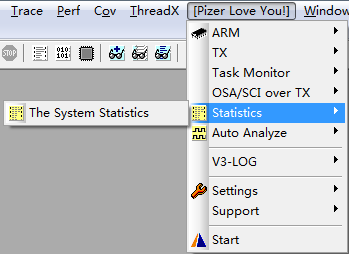


图 25 Statistics菜单

执行菜单：【PLY】>>【Statistics】>>【The System Statistics】，可以获取系统的各种统计信息，包括诸如有多少文件、行数、函数个数、类型数量、内存布局分段信息，以及运行时内核对象统计信息等等：

=============== Image Symbols Statistics ===============

files: 2495

lines: 1471425

functions: 69828

types: 162072

=============== Memory Config Regions ===============

Start Addr Size Name

0x88000000 207220736 mem\_size

0x90500000 15466496 EXEC\_KERNEL\_IMAGE

0x91600000 65536 SYS\_STACK

0x92490000 34603008 RAM\_RW

0x8A000000 7340032 RAM\_FIXNV

0x8A700000 9437184 RAM\_RUNNV

0x91610000 15204352 RAM\_HEAP

0x88000000 134217728 Share\_mem

0x90500000 67764224 PSCP\_mem

0x94800000 108068864 NRPHY\_mem

0x9B000000 33554432 V3PHY\_mem

0x88000000 352321536 MODEM\_Global

0x00000000 4096 PSCP\_AON\_IRAM

0x60004000 4096 V3PHY\_AON\_IRAM

0x60005000 4096 NRPHY\_AON\_IRAM

0x54100000 393216 PSCP\_LLRAM

0x48000000 1310720 NRPHY\_LLRAM

0x40020000 65536 V3PHY\_ATCM

0x40030000 65536 V3PHY\_BTCM

0x87800000 8388608 SIPC\_MEM

=============== OS Running Time Statistics ===============

threads: 120

queues: 125

timers: 2257

semaphores: 105

mutexes: 443

event flags: 288

byte pools: 7

block pools: 35

## Auto Analyze菜单

Auto Analyze菜单功能，主顾名思义就是自动化分析，根据既有发生的问题分析，进行逐一遍历分析：

1. 例如根据 assert info信息字眼，进一步分析assert大致种类，再根据辅助信息，给出准确定位。
2. 譬如内存分配失败，可能是RTOS的限制，如果是RTOS的限制，就会获取调用者的文件信息，函数信息等，用于自动流转；也可能是因为“内存泄漏”所致，就会获取TOP 10的分配情况，从而让APP判断可能的泄漏位置。
3. 譬如消息队列满，就会查找目标队列所属的线程，获取其状态是否就绪？或者阻塞？如果前者可能涉及优先级，以及队列SIZE，或者存在可能的循环，当前工具链会给出所有就绪且优先级不低于其的线程列表，也会给出队列中的消息列表；后者就需要进一步分析为什么阻塞，当前工具链会给出其阻塞时候的调用栈，并给出详细的文件信息，便于后台自动化转发到下一站的待分析模块。
4. 譬如内核对象操作失败，可能是RTOS的限制，也可能是非法的RTOS内核对象，这些都可以呈现给APP，并尽量发现潜在可疑的位置信息，用于自动流转。

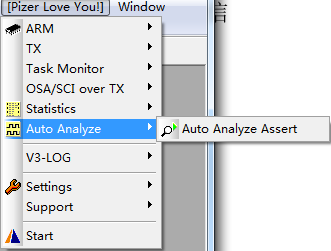


图 26 Auto Analyze菜单

例如某bug自动化分析，发现ASSERT，是因为内存分配失败，通过获取分配的TOP排行，发现文件号 2395 ，行号 8506 位置多达 1143处，存在可能的泄漏。

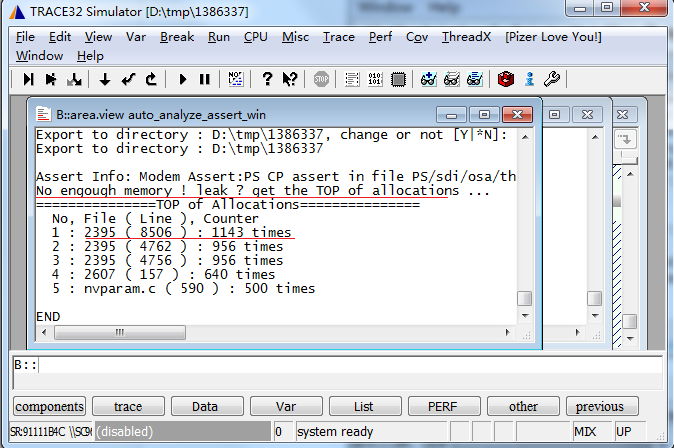


图 27 某Auto Analyze Assert的结果

## LOG菜单

LOG菜单功能，主要给APP LOG子系统本身提供分析的，例如导出通道里的LOG 数据到HOST 文件（俗称获取管道里的LOG数据，用于可能APP流程分析），有譬如LOG子系统的打点（俗称LOG的LOG）分析——用于对LOG子系统工作是否完好的分析。

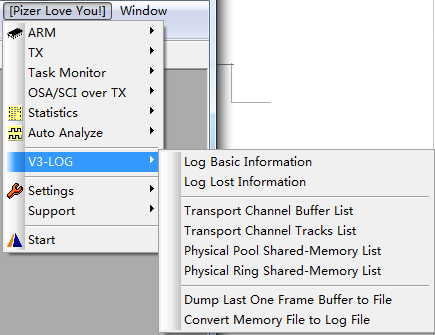


图 28 LOG菜单

**“Log Basic Information”** : MODEM LOG模块的基本信息。

**“Log Lost Information”** : MODEM LOG模块的LOG数据丢失信息。

**“Transport Channel Buffer List”** : MODEM LOG 传输模块 BUFFER LIST 信息。

**“Transport Channel Tracks List”** : MODEM LOG 传输模块的打点信息（即LOG模块的LOG）。

**“Physical Pool Shared-Memory List”** : SIPC LOG通路中已经传输完毕（空闲）的SBLOCK 块列表。

**“Physical Ring Shared-Memory List”** : SIPC LOG通路中正在传输中的SBLOCK 块列表。

**“Dump Last One Frame Buffer to File”** : 将MODEM LOG模块中最后一帧数据导出到HOST文件（主要涉及场景是：SIPC LOG传输通路阻塞时候）。

**“Convert Memory File to Log File”** : 从MODEM System Dump 的Memory File提取SIPC LOG通路中的LOG数据，保存到HOST文件。

# 自动检查是否升级

当执行\_start.cmm的时候，会自动检查是否有新版本，若发现新版本，会弹出对话提示（如下）。

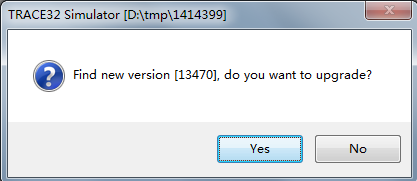


图 29 发现新版本，提示升级

如果暂时不想，可以点击否。之后可以在菜单中查看到：

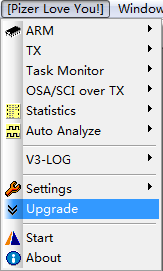


图 30 升级版本的菜单项

如果选择升级，就会看到SVN下载新版本：

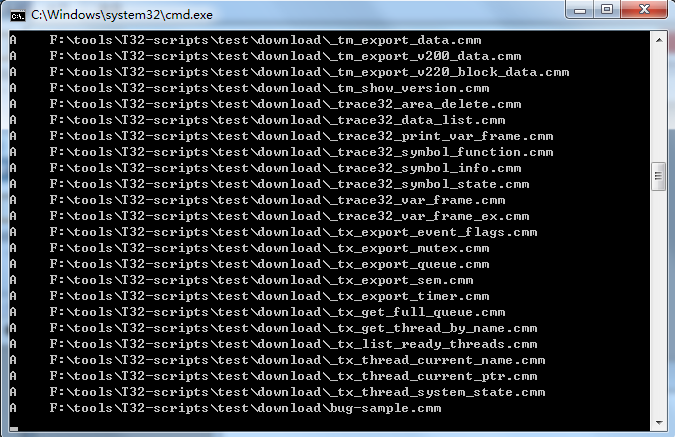


图 31 SVN下载新版本

# 后台自动化分析

后台自动化分析脚本，详细参考 bug-sample.cmm：

; product:

; 0: ORCA

; 1: SHARKLE/SHARKL5/SHARKL5PRO

; 2: W317/T117

global &\_global\_cli\_quiet &\_global\_cli\_product &\_global\_cli\_cpu\_num &\_global\_cli\_axf &\_global\_cli\_mem &\_global\_cli\_begin\_address &\_global\_cli\_working\_dir

&\_global\_cli\_quiet="1"

&\_global\_cli\_product="0"

&\_global\_cli\_axf="D:\tmp\1404339\SC9600\_Orca\_PSCP\_modem\_modem.axf"

&\_global\_cli\_mem="D:\tmp\1404339\arm\arm\_1.mem"

&\_global\_cli\_working\_dir="D:\tmp\1404339"

do "F:\tools\T32-scripts\PLY-T32Scripts\\_start.cmm"

do "F:\tools\T32-scripts\PLY-T32Scripts\\_auto\_analyze\_assert.cmm" "D:\tmp\1404339\a.log"

quit

其中重要的参数或过程：

1. **\_global\_cli\_product：**需要指明产品类型，主要是考虑的不同产品类型有着不同的 target memory layout（诸如不同region 的begin address、size），以及memory layout反映到core memory host file的host layout 格式。
2. **\_global\_cli\_axf：**用于指明target 的ELF执行程序
3. **\_global\_cli\_mem：**用于指明target 的core dump的memory host file
4. **\_global\_cli\_working\_dir：**用于指明本地工作路径，存储中间结果
5. **do …\_start.cmm：**用于启动恢复ASSERT现场
6. **do …\_auto\_analyze\_assert.cmm：**用于自动化分析ASSERT，并输出报告到指定文件中
7. **quit：**退出分析现场和程序

例如，我们创建一个 bug.cmm包含上述内容，再创建一个 bug.bat，其内容如下：

C:\T32\bin\windows64\t32marm.exe -c C:\T32\configsim.t32 -s D:\tmp\1404339\bug.cmm

然后便于在各个web等后台服务中调用执行bug.bat批处理程序，进而分析其输出的D:\tmp\1404339\a.log分析报告。

分析报告的一些关键字，包括：

* [Task doing: xxxx] ：指示当前正在执行的函数
* [Task entry: xxxx] ：指示当前线程的entry函数
* [Task name: xxxx] ：指示当前线程的名称
* [owner : xxxx] ：指示所属的owner，可以是 file path、queue name、或者 task name等。
* [Analyze : Memory leak] ：分析结论是内存泄漏
* [Analyze : Memory Overrun] ：分析结论是内存可能覆盖
* [Analyze : Seems] ：分析结论可能的情况
* [Analyze : Error] ：环境错误
* [Analyze : Ignore] ：可以忽略当前ASSERT，一般为其它子系统导致的。