

PLY-T32Scripts 手册指南

(V0.4)

Pizer.Fan

目录

前言.....	2
1 概述.....	3
1.1 “PLY-T32Scripts”能做什么？	3
1.2 “PLY-T32Scripts”哪里下载？	3
1.3 启动 “PLY-T32Scripts”	3
1.4 配置自动启动 “PLY-T32Scripts”	5
1.5 Script list.....	6
2 启动入口.....	10
3 图形界面.....	11
3.1 工具栏.....	11
3.2 ARM 菜单.....	11
3.2.1 实例: 推导调用栈.....	12
3.2.2 实例: 代码段是否破坏？	13
3.2.3 实例: 显示当前 MPU 的配置.....	14
3.3 TX 菜单.....	15
3.3.1 实例: 显示当前所有就绪和执行状态的线程	15
3.3.2 实例: 定时器是否激活？	17
3.3.3 实例: 栈溢出？	17
3.4 Task Monitor 菜单.....	19
3.5 OSA/SCI over TX 菜单	20
3.5.1 实例: HEAP 内存破坏	20
3.6 Statistics 菜单	22
3.7 Auto Analyze 菜单	23
3.8 LOG 菜单	24
4 自动检查是否升级.....	26
5 后台自动化分析.....	27

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。

- (3) 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。

1 概述

1.1 “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；

1.2 “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

1.3 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：

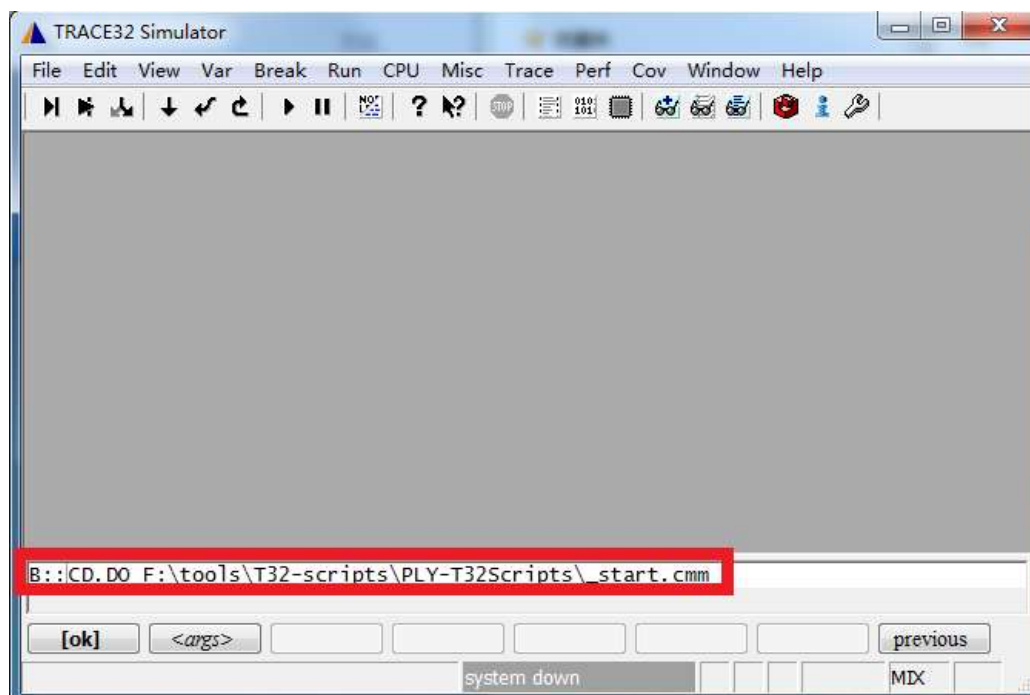


图 1 首次启动

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

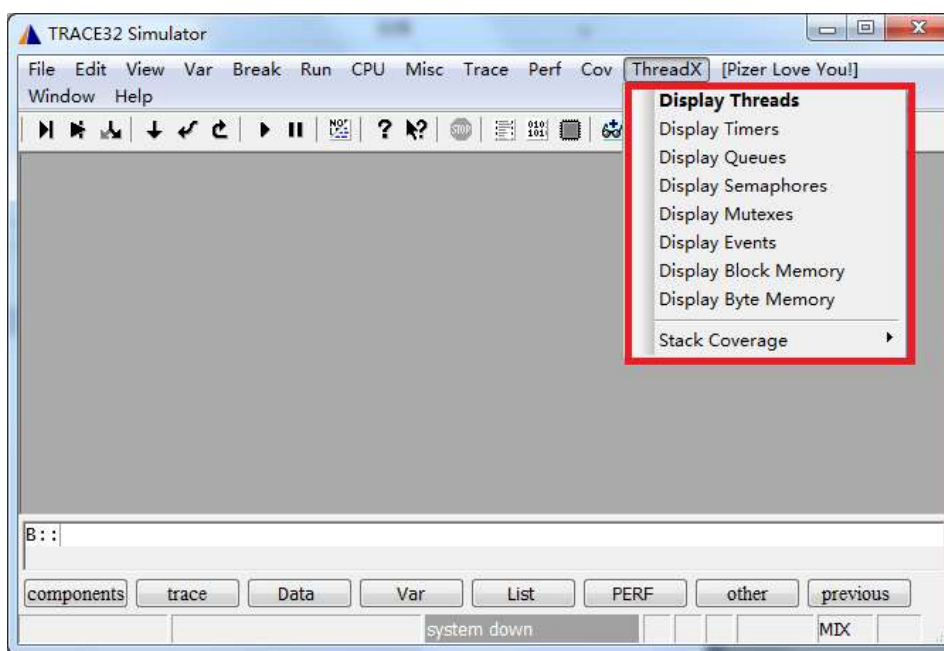


图 2 ThreadX 菜单

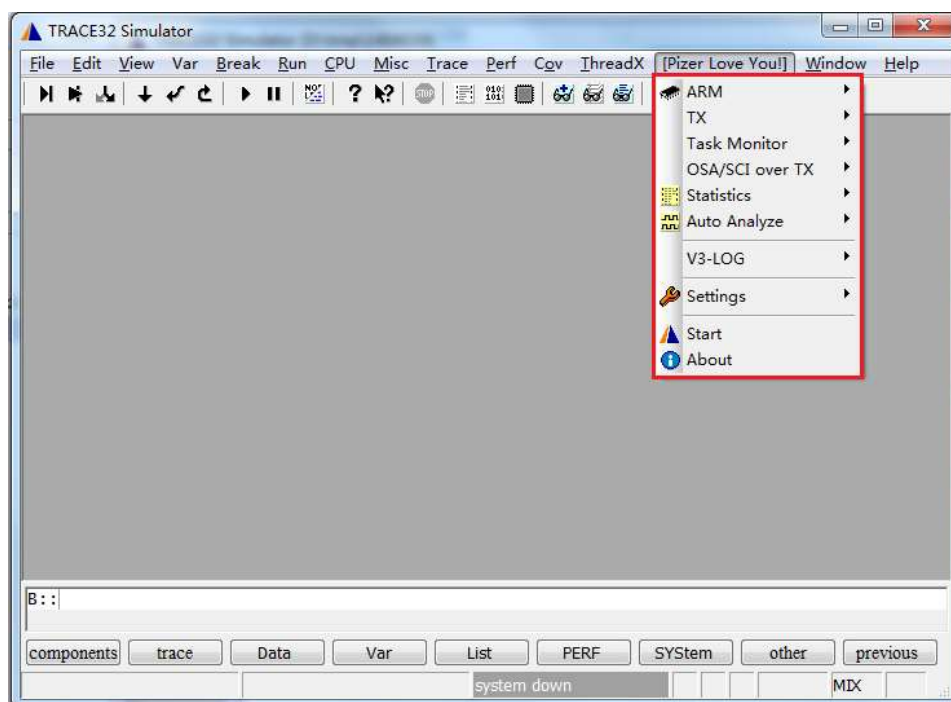


图 3 PLY 菜单

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”。

1.4 配置自动启动 “PLY-T32Scripts”

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

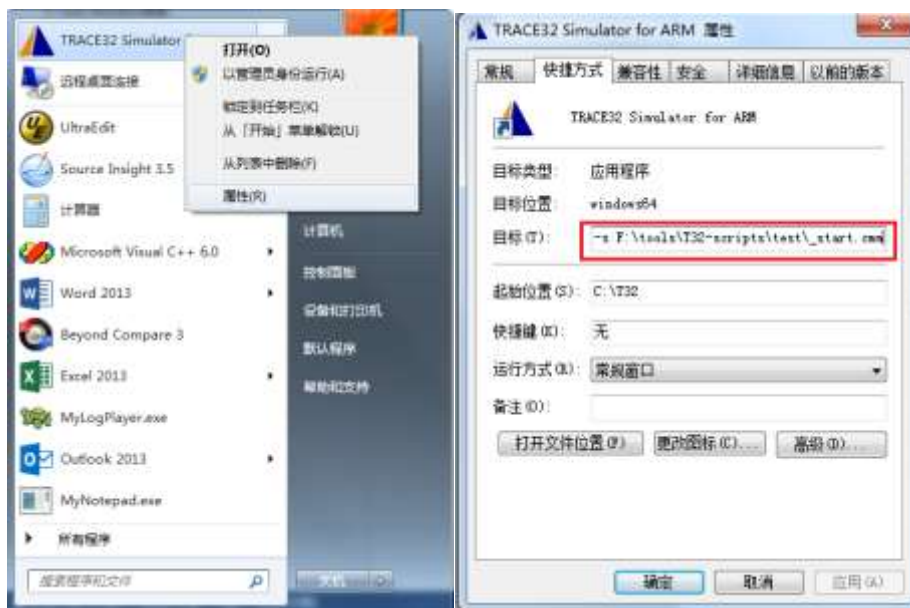


图 4 PLY 菜单自动加载配置

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`

1.5 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 damaged 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
```

2 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：

- 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

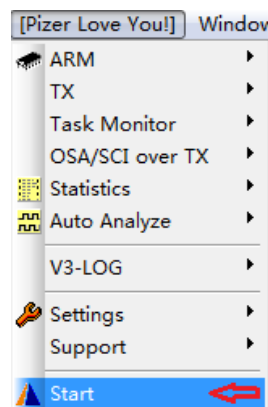


图 5 启动入口菜单





图 6 工具栏中的启动入口


3 图形界面

3.1 工具栏

PLY 工具栏: 

: 等效 “Auto Analyze Assert”菜单

: 等效 “Start”菜单

: 等效 “About”菜单

3.2 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。

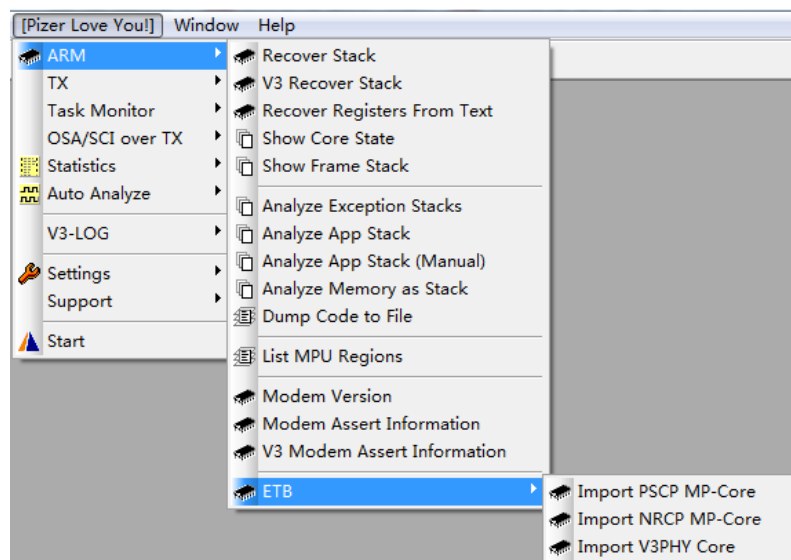


图 7 ARM 菜单

“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）。

3.2.1 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	hisor0
4	8274D9DC	SEMAPHORE_SUSP	1	1441	hisor1
5	8274DB08	SEMAPHORE_SUSP	2	188	hisor2
6	8274DC34	SEMAPHORE_SUSP	1	1	mbox_send_thread
...					

例如选择“hisor0”，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）：

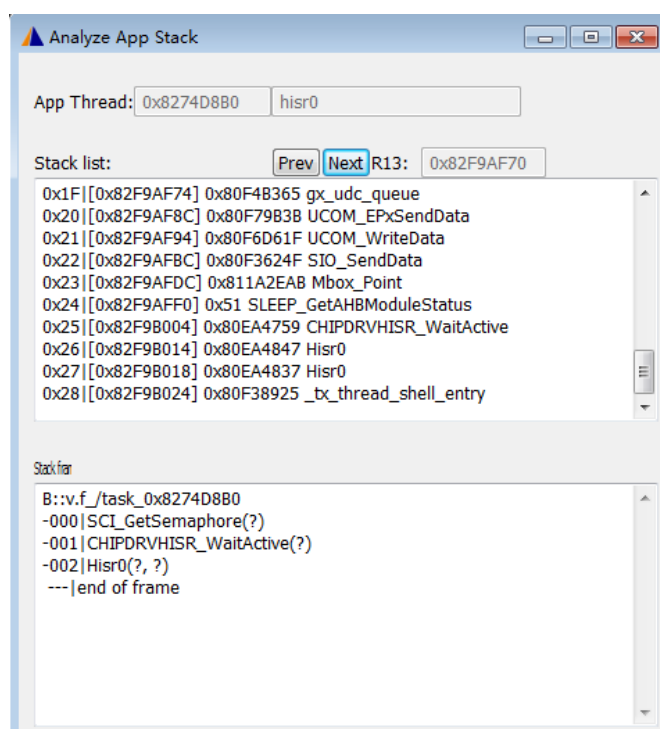


图 9 实例：推导调用栈

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？

3.2.2 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:

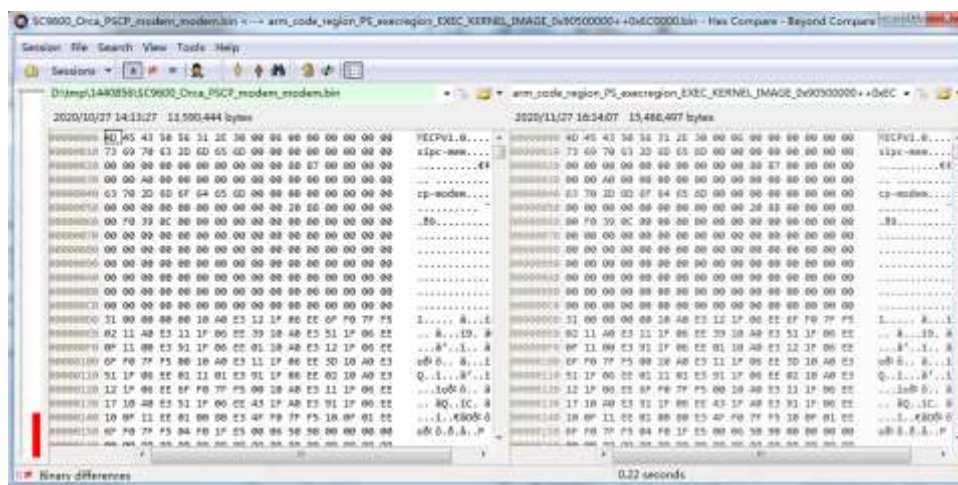


图 11 对比运行前后的代码段

3.2.3 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。

number	baseaddr	(endaddr)	size	(rgn size)	ap, attr, enable
1	00000000	(80000000)	0000003C	(80000000)	0x100 (RW), 0x0F (WRITE-BACK, WRITE-ALLOCATE SHAREABLE), 0x1
2	00000000	(00001000)	00000016	(00001000)	0x100 (RW), 0x0F (WRITE-BACK, WRITE-ALLOCATE SHAREABLE), 0x1
3	00000000	(00000800)	00000014	(00000800)	0x500 (RD), 0x0F (WRITE-BACK, WRITE-ALLOCATE SHAREABLE), 0x1
4	88000000	(90000000)	00000034	(08000000)	0x100 (RW), 0x0C (NO-CACHE SHAREABLE), 0x1
5	8F800000	(90000000)	0000002C	(00800000)	0x500 (RD), 0x0F (WRITE-BACK, WRITE-ALLOCATE SHAREABLE), 0x1
6	8F800000	(8F820000)	00000020	(00020000)	0x100 (RW), 0x0C (NO-CACHE SHAREABLE), 0x1
7	90000000	(92000000)	00000030	(02000000)	0x100 (RW), 0x0F (WRITE-BACK, WRITE-ALLOCATE SHAREABLE), 0x1
8	90000000	(90800000)	0000002C	(00800000)	0x500 (RD), 0x0F (WRITE-BACK, WRITE-ALLOCATE SHAREABLE), 0x1
9	90600000	(90800000)	00000028	(00200000)	0x100 (RW), 0x0C (NO-CACHE SHAREABLE), 0x1
10	90800000	(90820000)	00000020	(00020000)	0x100 (RW), 0x0C (NO-CACHE SHAREABLE), 0x1
11	92000000	(93000000)	0000002E	(01000000)	0x100 (RW), 0x0F (WRITE-BACK, WRITE-ALLOCATE SHAREABLE), 0x1
12	87800000	(88000000)	0000002C	(00800000)	0x100 (RW), 0x0C (NO-CACHE SHAREABLE), 0x1
13	54100000	(54140000)	00000022	(00040000)	0x100 (RW), 0x0C (NO-CACHE SHAREABLE), 0x1
14	54140000	(54160000)	00000020	(00020000)	0x100 (RW), 0x0C (NO-CACHE SHAREABLE), 0x1
15	00000000	(00001000)	00000012	(00000400)	0x100 (RW), 0x0C (NO-CACHE SHAREABLE), 0x1
0	00000000	(00000000)	0000003E	(00000000)	0x1000 (EXECNEVER), 0x0 (STRONGORDER), 0x1

图 12 显示当前 MPU 的配置

3.3 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？

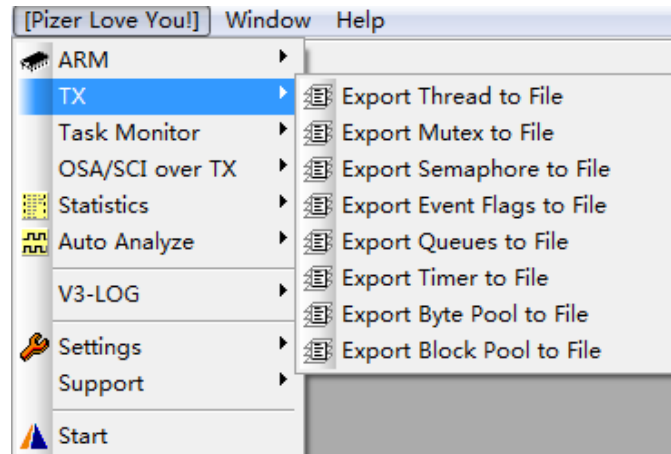


图 13 TX 菜单

“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。

3.3.1 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，如下可以查看有 3 个就绪状态的线程，以及一个正在执行的线程：

No.	Addr.	State	Prio.	Runcount.	Name
8	8274DE8C	READY	248	21940	MTA Task
20	8274EB70	READY	249	30624	DSP_LOG_Task
48	8274CCFB	READY	180	3184	T_TFT_HAL_MAIN_HANDLER
83	82752608	EXECUTING	13	60861	LGRANT Task

No.	Addr.	State	Prio.	Runcount.	Name
1	81C86880	SUSPENDED	2	10969	System Timer Thread
2	8274D784	QUEUE_SUSP	9	1	RTOS_Manage
3	8274D880	SEMAPHORE_SUSP	0	122893	hiss0
4	8274D90C	SEMAPHORE_SUSP	1	1441	hiss1
5	8274D808	SEMAPHORE_SUSP	2	188	hiss2
6	8274DC34	SEMAPHORE_SUSP	1	1	mbox_send_thread
7	8274DD60	EVENT_FLAG	223	1	MS_MTA_RECV
8	8274DE8C	READY	248	21940	MTA Task
9	8274DFB8	QUEUE_SUSP	220	20	T_USB_VBUS_HANDLER
10	8274E0E4	QUEUE_SUSP	220	7	T_USBSERVICE_HANDLER
11	8274E210	QUEUE_SUSP	4	1	T_P_CHNG_FREQ
12	8274E33C	QUEUE_SUSP	75	340	CHGNG_SRV
13	8274E468	QUEUE_SUSP	58	123	THERMAL_SRV
14	8274E594	QUEUE_SUSP	228	1	T_IMG_PROC_CTRL
15	8274E6C0	QUEUE_SUSP	231	1	T_IMG_PROC_PROCESS
16	82747640	QUEUE_SUSP	228	1	TASK_ISP_SERVICE
17	8274E7EC	QUEUE_SUSP	229	1	T_IMG_DEC

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，如下可以看见 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：

No.	Addr.	State	Prio.	Runcount.	Name
16	9089EC7C	READY	248	516776	sblock_send_thread
25	9089C90C	EXECUTING(0)	12	56568	nrpal_hw

No.	Addr.	State	Prio.	Runcount.	Name
1	91700124	SUSPENDED	8	113943	System Timer Thread
2	90890A1C	QUEUE_SUSP	9	1	RTOS_Manage
3	90890B6C	SEMAPHORE_SUSP	0	1	hiss0
4	90890C8C	SEMAPHORE_SUSP	6	59063	hiss1
5	90890E0C	SEMAPHORE_SUSP	10	1	hiss2
6	90890F5C	EVENT_FLAG	242	14966	RamDisk
7	90890E4C	QUEUE_SUSP	242	11	RamDisk_CoreSync
8	9089E1FC	QUEUE_SUSP	243	2088	T_REFNOTIFY
9	9089E34C	SEMAPHORE_SUSP	0	1	mbox_send_thread
10	9089E49C	EVENT_FLAG	135	4	ctrl_0
11	9089E56C	EVENT_FLAG	130	1	DUAL_SIN_HOTPLUG_MAIN_HANDLER
12	9089E73C	EVENT_FLAG	5	3	sio_r0_0
13	9089E88C	QUEUE_SUSP	5	3	sio_w0_0
14	9089E9DC	QUEUE_SUSP	127	3	ppp_send_thread
15	9089EB2C	EVENT_FLAG	19	2	sblock_0_5
16	9089EC7C	READY	248	516776	sblock_send_thread
17	9089EDCC	EVENT_FLAG	115	78696	sbuf_smux
18	9089EF1C	EVENT_FLAG	223	1	MS_MTA_RECV
19	9089F06C	QUEUE_SUSP	248	1	MTA Task

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：

```

core_name: PS
the current mode is 0x60000003: svc mode {i:0x1,f:0x1,t:0x0}.
the previous mode is 0x60000073: svc mode {i:0x0,f:0x1,t:0x1}.
Core 0x0 exception type is 0x0
the current mode is svc mode.
the current mode is 0x80000002: irq mode {i:0x1,f:0x1,t:0x0}.
the previous mode is 0x531: svc mode {i:0x0,f:0x1,t:0x0}.
Core 0x1 exception type is 0x0
the current mode is irq mode.
ETB is enabled, but not choosed for the current subsys(PS) !
Current threads in TX:
{
  _tx_thread_current_ptr[0x0]: 0x9089C90C nrpal_hw
  _tx_thread_execute_ptr[0x0]: 0x9089C90C nrpal_hw
  _tx_thread_system_state[0x0]: 0x0
  _tx_thread_current_ptr[0x1]: 0x0
  _tx_thread_execute_ptr[0x1]: 0x9089EC7C sblock_send_thread
  _tx_thread_system_state[0x1]: 0x1
  Seems IRQ is breaking the thread: sblock_send_thread
}
  
```

图 16 查询 CORE 的状态

3.3.2 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：

Type	Addr	Kernal	Func	Param	Next	Prev	Name
ATIM	0x91759878	2677	0x90043093	91759878	917598D8	917598B0	PERIODIC_PHR_TIMER
ATIM	0x917598D0	2677	0x90043093	917598D0	9167F9E8	91759880	PROHIBIT_PHR_TIMER
THRD	0x9167F99C	2780	0x9058C1ED	9167F99C	917597D0	917598D0	tcpip_clock
ATIM	0x917597C8	2897	0x90043093	917597C8	917611C4	9167F9E8	RETX_BSR_TIMER
ATIM	0x9176118C	8882	0x90043093	9176118C	91761484	917597D0	OSAL_tmr
ATIM	0x9176147C	12632	0x90043093	9176147C	923E0F6C	917611C4	MSG_ID_NLR_RRC_MEASURE_PERIOD
ATIM	0x923E0F64	12649	0x90043093	923E0F64	9172BA58	91761484	PHONE_LIDSP_Timer
ATIM	0x9172BA50	24522	0x90043093	9172BA50	91759510	923E0F6C	SIM_CONTEXT_TIMER
ATIM	0x91759508	2679319	0x90043093	91759508	91729068	9172BA58	SIM_ST_validityTimer_Ptr
ATIM	0x91729060	2832613	0x90043093	91729060	91756490	91759510	NVM_FLUSH_TIMER
ATIM	0x91756488	2874627	0x90043093	91756488	91757460	91729068	OSAL_tmr
ATIM	0x91757458	53240745	0x90043093	91757458	91758438	91756490	CleanForbiddenTAI_list_Timer
ATIM	0x91758430	143076778	0x90043093	91758430	9174AB20	91757460	rlf or hf valid timer
ATIM	0x9174AB18	566272617	0x90043093	9174AB18	91759EB0	91758438	OSAL_tmr
ATIM	0x91759EAB	4294967295	0x90043093	91759EAB	91759880	9174AB20	T_timeAlignment

Type	Addr	Remain	Func	Param	Next	Prev	Name
ATIM	0x923DEA38	600000	0x90043093	923DEA38	00000000	00000000	Modem_Assert_Timer
ATIM	0x923E0778	100	0x90043093	923E0778	00000000	00000000	LogFlush
ATIM	0x91728088	1	0x90043093	91728088	00000000	00000000	Inject_Timer
ATIM	0x91729060	2832613	0x90043093	91729060	91756490	91759510	NVM_FLUSH_TIMER

图 17 查询定时器是否激活

3.3.3 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：

No.	Addr	Stack	Start	-End	State	Prio	Runcount	Name
101	9089765C	913AE09C	00000000	EFEFEFEF	READY	224	69680	T_P_ATC

No.	Addr	Stack	Start	-End	State	Prio	Runcount	Name
1	91804354	9180442B	EFEFEFEF	EFEFEFEF	READY	8	422919	System Timer Thread
2	90890A1C	9180637C	EFEFEFEF	EFEFEFEF	QUEUE_SUSP	9	1	RTOS_Manage
3	90890B6C	912FD038	EFEFEFEF	EFEFEFEF	SEMAPHORE_SUSP	0	1	hisr0
4	90890C8C	912FE06C	EFEFEFEF	EFEFEFEF	SEMAPHORE_SUSP	6	37227	hisr1
5	90890EDC	912FF040	EFEFEFEF	EFEFEFEF	SEMAPHORE_SUSP	10	1	hisr2
6	90890F5C	91300004	EFEFEFEF	EFEFEFEF	READY	242	808	RamDisk
7	90890DAC	91301908	EFEFEFEF	EFEFEFEF	QUEUE_SUSP	242	17	RamDisk_CoreSync
8	908911FC	91763E4C	EFEFEFEF	EFEFEFEF	QUEUE_SUSP	243	884	T_REFNOTIFY
9	9089134C	91749018	EFEFEFEF	EFEFEFEF	SEMAPHORE_SUSP	0	1	mbox_send_thread
10	9089E49C	917550CC	EFEFEFEF	EFEFEFEF	EVENT_FLAG	135	3	sctrl_0
11	9089E5EC	91755544	EFEFEFEF	EFEFEFEF	EVENT_FLAG	130	2	DUAL_SIM_HOTPLUG_MAI
12	9089E73C	9176CD50	EFEFEFEF	EFEFEFEF	EVENT_FLAG	5	3	sio_rdr_0
13	9089E88C	9175590C	EFEFEFEF	EFEFEFEF	QUEUE_SUSP	5	2	sio_wr_0
14	9089E9DC	91755E34	EFEFEFEF	EFEFEFEF	QUEUE_SUSP	127	3	ppp_send_thread
15	9089EB2C	91303160	EFEFEFEF	EFEFEFEF	EVENT_FLAG	19	2	sblock_0_3
16	9089EC7C	917562AC	EFEFEFEF	EFEFEFEF	READY	248	1205087	sblock_send_thread
17	9089EDCC	91303968	EFEFEFEF	EFEFEFEF	EVENT_FLAG	115	32071	sbuff_smux
18	9089EF1C	908C6960	EFEFEFEF	EFEFEFEF	EVENT_FLAG	223	1	MS_MTA_RECV
19	9089F06C	90946880	EFEFEFEF	EFEFEFEF	QUEUE_SUSP	248	1	MTA_Task

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_ [ b1 0x8F8610A0 ; mta_chec
  [0x913AB68C] 0x8FFEDDE5 LOG_Print_ [ b1 0x8F8610A0 ; mta_chec
  [0x913AB694] 0x8FFEDDE5 LOG_Print_ [ b1 0x8F8610A0 ; mta_chec
  [0x913AB75C] 0x8FAC413B nrrc_get_module_state [ b1 0x8FFEDDC8
  [0x913AB760] 0x8FAC4514 nrrc_common_handle_store_info
  [0x913AB7A4] 0x8FA22AE9 nrrcng_is_ue_stay_in_nr_cell [ b1 0x8FFEDDC28
  [0x913AB7A8] 0x8FA22D84 nrrcng_nr_temperature_control
  ...
  [0x913B053C] 0x8F85FEBF sdi_msg_send_and_trace_ex9 [ b1 0x8F85FDBA
  [0x913B0574] 0x8F85BE5 sdi_msg_send [ b1 0x8F85FE96 ; sdi_ms
  [0x913B058C] 0x805 SCI_SIM_EnableDeepSleep
  [0x913B0594] 0x8FAACE87 osa_tx_free_buff [ b1 0x8F830C78 ; _t
  [0x913B05A4] 0x8FAACDC7 osa_int_release_buffer [ b1 0x8FAACE56
  [0x913B05C4] 0x8FFE4F6B sdi_blk_mem_int_free [ b1 0x8FAACC58
  [0x913B05CC] 0x8FFD852D mnrsf_crsm_callback_function
  [0x913B060C] 0x8FFA1EBF ATC_List_Expected_Event [ b1 0x8FFE92E2
  [0x913B0614] 0x8FFA1F7C ATC_List_Expected_Event
  [0x913B064C] 0x8FFA2D67 ATC_Add_Expected_Event [ b1 0x8FFA1E20
  [0x913B0674] 0x8FC0456D ATC_ProcessCRSM [ b1 0x8FFA1F94 ; ATC
```

3.4 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。

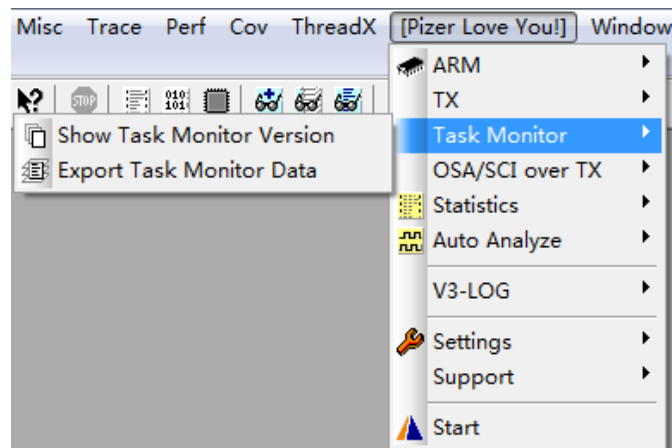


图 20 Task Monitor 菜单

“**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。

3.5 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。

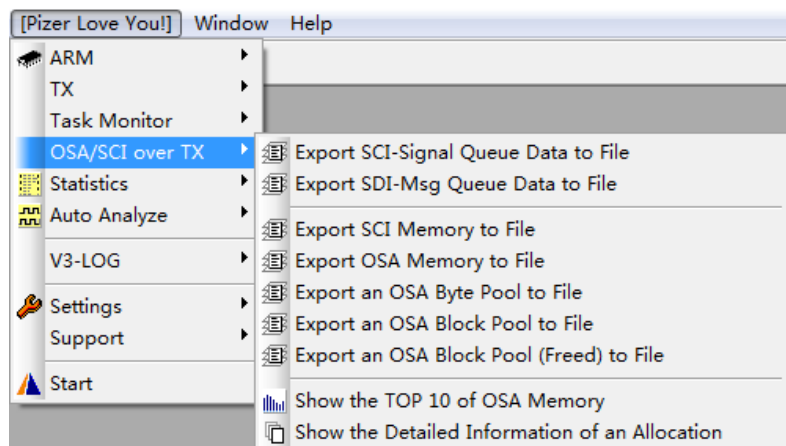


图 21 OSA/SCI over TX 菜单

“**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。

3.5.1 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 0x8CBF7C4C 内存发现被破坏，通过 PLY MENU>>OSA/SCI over TX >> “**Show the Detailed Information of an Allocation**”：

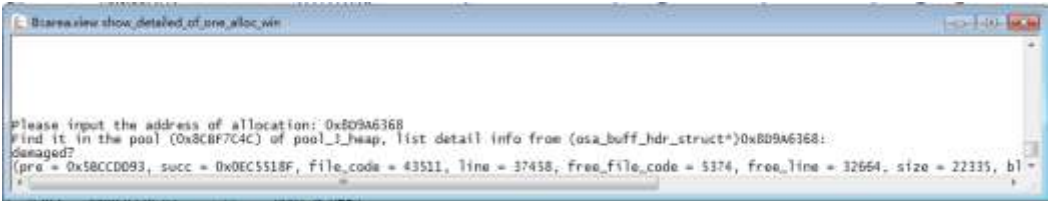


图 22 获取指定内存（分配状态）的信息

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**”：

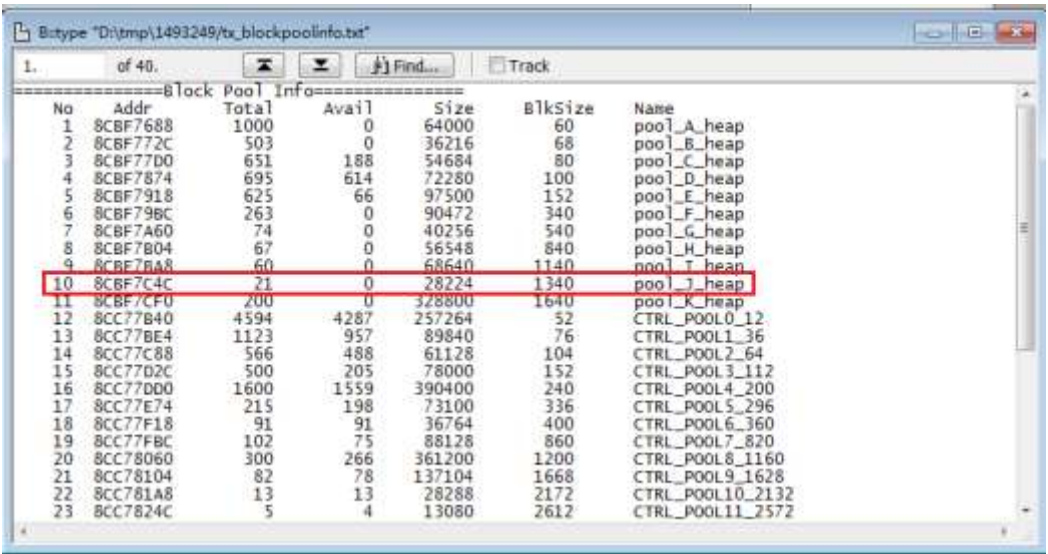


图 23 获取所有块内存池的信息

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**”：

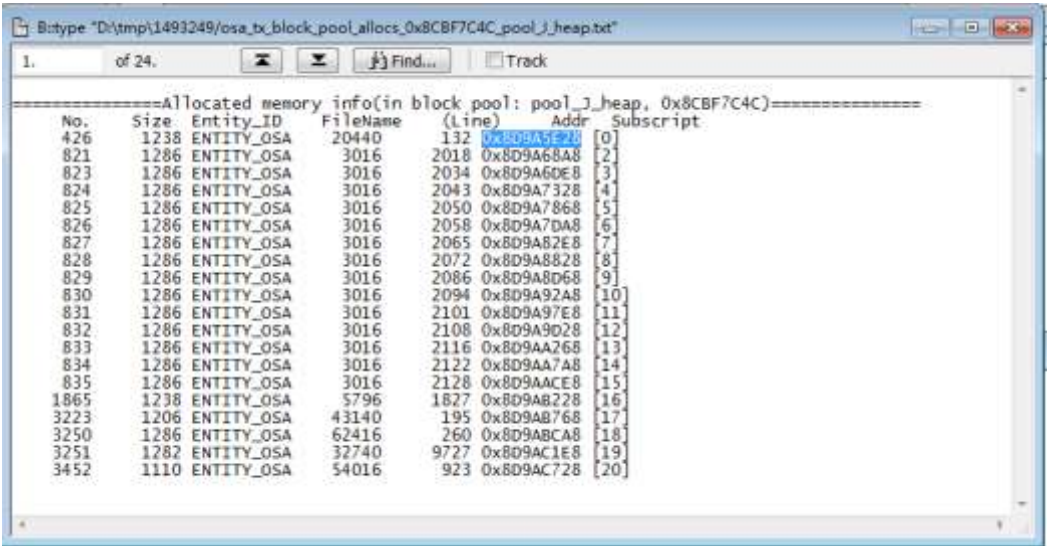


图 24 指定内存池的数据信息

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。

3.6 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。

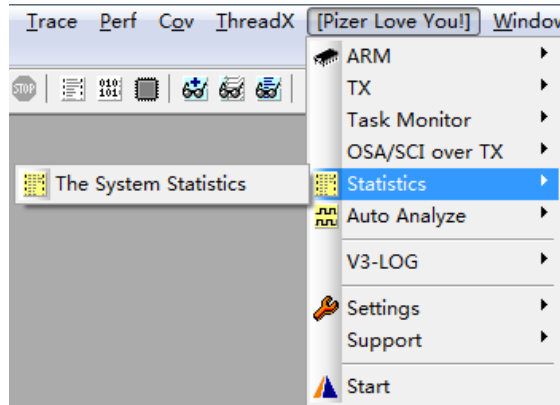


图 25 Statistics 菜单

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

```

3.7 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。

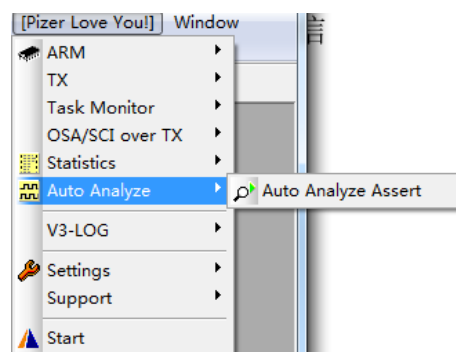


图 26 Auto Analyze 菜单

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。

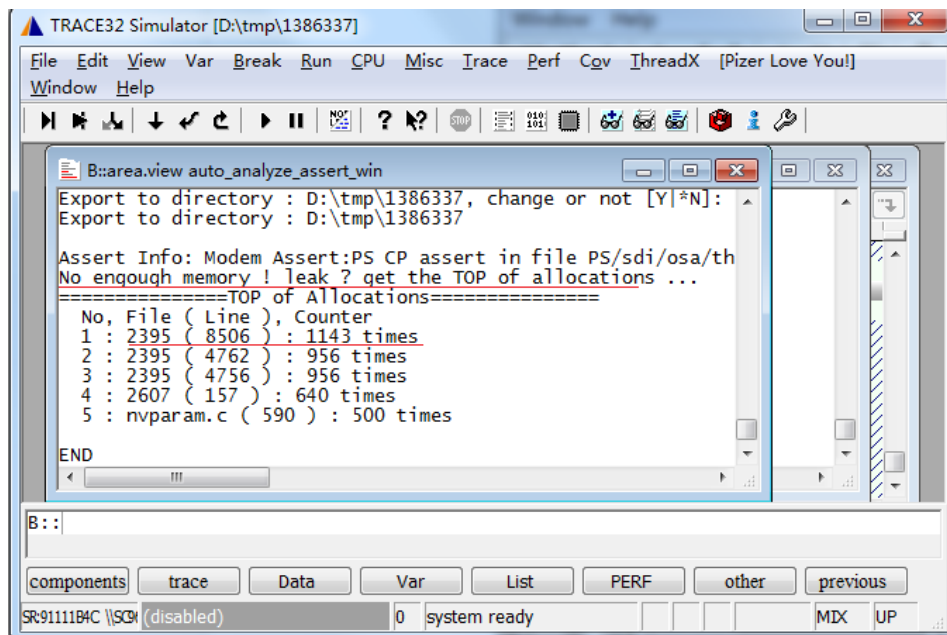


图 27 某 Auto Analyze Assert 的结果

3.8 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。

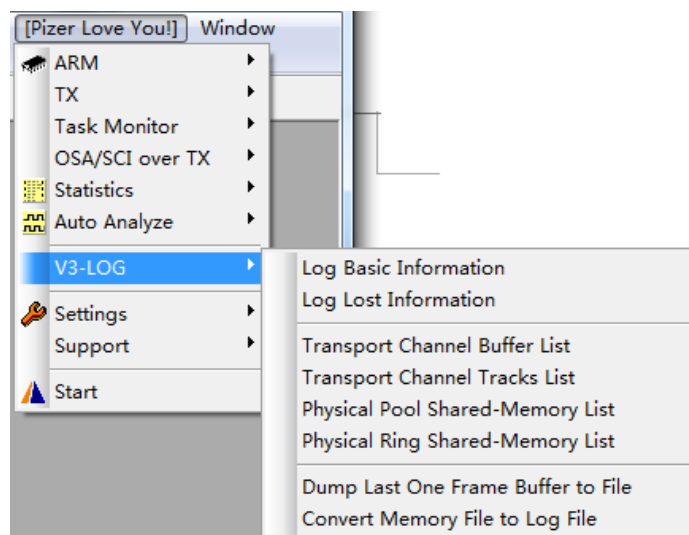


图 28 LOG 菜单

“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。

4 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）。

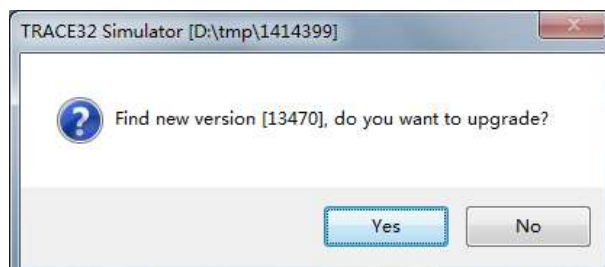


图 29 发现新版本，提示升级

如果暂时不想，可以点击否。之后可以在菜单中查看到：

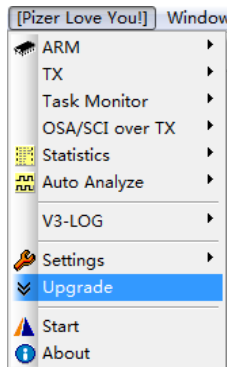


图 30 升级版本的菜单项

如果选择升级，就会看到 SVN 下载新版本：

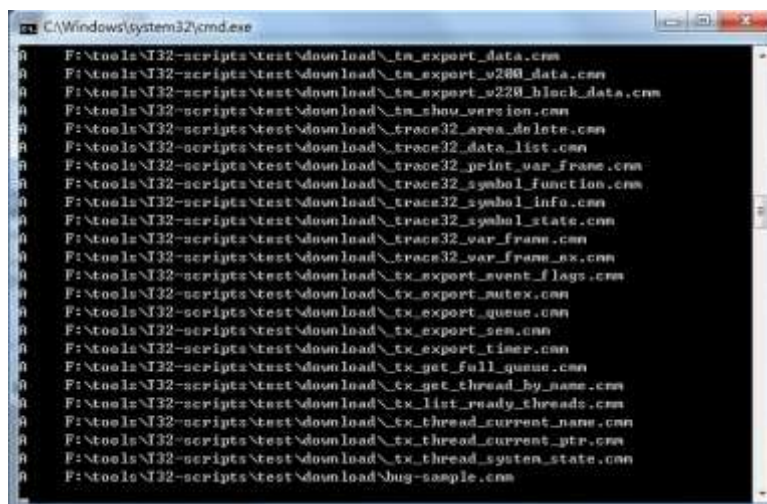


图 31 SVN 下载新版本

5 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  &_amp;global_cli_quiet  &_amp;global_cli_product  &_amp;global_cli_cpu_num  &_amp;global_cli_axf
&_global_cli_mem  &_amp;global_cli_begin_address  &_amp;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：

- (一) **_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。
- (二) **_global_cli_axf**：用于指明 target 的 ELF 执行程序
- (三) **_global_cli_mem**：用于指明 target 的 core dump 的 memory host file
- (四) **_global_cli_working_dir**：用于指明本地工作路径，存储中间结果
- (五) **do ..._start.cmm**：用于启动恢复 ASSERT 现场
- (六) **do ..._auto_analyze_assert.cmm**：用于自动化分析 ASSERT，并输出报告到指定文件中
- (七) **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。