



版本: 1.0.1 2015 年 03 月

Sword37 状态机应用 接口说明

声 明

本手册的版权归安凯技术公司所有，受相关法律法规的保护。未经安凯技术公司的事先书面许可，任何人不得复制、传播本手册的内容。

本手册所涉及的知识产权归属安凯技术公司所有（或经合作商授权许可使用），任何人不得侵犯。

本手册不对包括但不限于下列事项担保：适销性、特殊用途的适用性；实施该用途不会侵害第三方的知识产权等权利。

安凯技术公司不对由使用本手册或执行本手册内容而带来的任何损害负责。

本手册是按当前的状态提供参考，随附产品或本书内容如有更改，恕不另行通知。

联 系 方 式

安凯（广州）微电子有限公司

地址：广州科学城科学大道 182 号创新大厦 C1 区 3 楼

电话: (86)-20-3221 9000

传真: (86)-20-3221 9258

邮编: 510663

销售热线:

(86)-20-3221 9499

电子邮箱:

sales@anyka.com

主页:

<http://www.anyka.com>

版本变更说明

以下表格对于本文档的版本变更做一个简要的说明。版本变更仅限于技术内容的变更，不包括版式、格式、句法等的变更。

版本	说明	完成日期
V1.0.0	正式发布	2011 年 12 月
V1.0.1	增加态机机制内容	2015 年 03 月

文档内容说明

本文档为《Sword37 用户开发手册》配套使用文档，主要介绍了状态机应用的详细接口说明。

目录

1	模块介绍	5
1.1	功能概述	5
1.2	依赖模块关系	5
2	状态机机制	5
2.1	EXCEL配置表的维护	5
2.1.1	普通状态机	5
2.1.2	特殊状态机	7
2.2	状态机源文件的维护	8
2.2.1	添加新的普通状态机	8
2.2.2	删除已有普通状态机	9
2.2.3	特殊状态机	9
2.2.4	注意事项	9
2.3	事件跳转流程	12
2.3.1	状态机内部跳转关系	12
2.3.2	系统状态跳转	15
3	接口说明	16
3.1	子模块功能概述	16
3.2	术语定义	16
3.3	数据结构定义	17
3.3.1	vT_EvtCode	17
3.3.2	vT_EvtParam	17
3.3.3	M_STATESTRUCT	17
3.3.4	_fHandle	18
3.3.5	_feHandle	18
3.3.6	_GetNextState	18
3.3.7	_fVoid	20
3.3.8	M_TRANSTYPE	20
3.4	接口函数列表	21
3.4.1	m_initStateHandler	21
3.4.2	m_mainloop	21
3.4.3	m_triggerEvent	22
3.4.4	m_regSuspendFunc	22
3.4.5	m_regResumeFunc	23

3.4.6	SM_GetCurrentSM	23
3.4.7	SM_GetStackMaxDepth.....	24
3.4.8	SM_CalcStackBufByMaxDepth	24
3.4.9	SM_GetEventMaxEntries	24
3.4.10	SM_CalcEventBufferByMaxEntries.....	25
4	状态机库的特性配置	25
5	依赖接口列表	25
5.1	状态机管理库配置	25
5.1.1	SM_GetStateArray.....	26
5.1.2	SM_GetfHande	26
5.1.3	SM_GeteHandle.....	26
5.1.4	SM_GetStackBuffer	26
5.1.5	SM_GetEventQueueBuffer.....	27
5.1.6	SM_GetEvtReturn.....	27
5.1.7	SM_GetEvtNoNext.....	27
5.1.8	SM_GetPreProcID	28
5.1.9	SM_GetPostProcID.....	28
5.2	目标平台调试接口	28

1 模块介绍

1.1 功能概述

本模块提供状态机的事件驱动切换框架，支持由用户定义状态机，定义事件切换逻辑，提供状态机事件分发/触发的核心逻辑。本模块可由外部配置状态机栈的存贮缓冲区，事件的缓冲区。

1.2 依赖模块关系

本模块依赖于平台调试模块。

2 状态机机制

状态机是用于定义系统所有可能的状态及负责状态维护、状态间跳转的调度等核心程序，通过状态切换和事件跳转来显示不同的界面。

状态机设计的思想是将应用功能分割成小状态（状态机）。每一个小状态尽可能简单和独立。由于状态机是以状态栈的形式进行管理的，按照先入后出的原则进行状态机跳转。在任何情况下，状态栈里面至少有一个状态机（standby 状态机），栈顶状态机（当前状态）只有一个，只有栈顶状态机能收到事件队列的事件，所谓状态跳转是状态栈最顶上的状态机变化。

2.1 Excel配置表的维护

系统处于不同的状态，各个状态之间的跳转通过 statelist.xls 对状态机进行维护。statelist.xls 依据模块的不同分成若干个工作表，按模块划分出了状态机的跳转定义。

2.1.1 普通状态机

表格的第一行固定为标题，不存放任何状态机跳转定义；

表格的每一行非空行定义一个特定的状态机跳转，如下图所示。

A	B	C	D	E	F
Module Name	ID	Current State	Description	Event	Next State
radio		s_radio_player	radio player	M_EVT_MENU	+> s_radio_menu
				M_EVT_VOLUME	+> s_radio_adjust_volume
				M_EVT_RETURN_ROOT	+> _return(STACKROOT)
				M_EVT_RETURN	_return
				M_EVT_EXIT	_return
		s_radio_menu	radio menu	M_EVT_RADIO_RECORD	-> s_audio_record
				M_EVT_RETURN_ROOT	+> _return(STACKROOT)
				M_EVT_EXIT	_return
				M_EVT_RETURN	_return(2)
				M_EVT_TIMEOUT	_return
		s_radio_adjust_volume	change radio volume	M_EVT_RETURN_ROOT	+> _return(STACKROOT)
				M_EVT_EXIT	_return
				M_EVT_RETURN	_return(2)
				M_EVT_TIMEOUT	_return

图 2-1 普通状态机定义

表格中各列的名称定义如下。

名称	定义
Module Name	(可选) 状态机所属的模块名
ID	(可选) 保留, 无用
Current State	(必填) 当前状态机名称, 该名称必须符合 C 关键字语法, 建议小写
Description	(可选) 描述、注释用
Event	(必填) 触发该次跳转的事件
Next State	(必填) 状态跳转的目标状态机; 该字段的书写规则为: JumpMode NextStateName (定义见下两行)
Jump Mode	-> 弹栈式跳转
	+> 压栈式跳转
	_ 返回式跳转
NextStateName	跳转的目标状态机名称

JumpMode 为 “_” 时, **NextStateName** 必须为 return(n) 格式, n 可以是 ≥ 0 的数字, 当 n 为数字时, 指示状态机返回跳转时, 返回几次。

以同一个状态机为源状态机的跳转应写成一组，第一个跳转定义的 **Current State** 字段写为源状态机名，后续跳转定义行的 **Current State** 字段应保持为空。不同源的状态机跳转之间用空行隔开，具体的跳转示例见源代码包中的 `statelist.xls` 表格。

弹栈式跳转就是将状态栈栈顶的转态机替换成新的状态机，换句话说，就是原来的栈顶状态机，先通过调用该状态机的退出(`exit`)函数，移出状态栈，新的状态机再通过调用它的初始化(`init`)函数移入状态栈，成为栈顶的转态机（当前状态）。

压栈式跳转（入栈式跳转）就是叠加 1 个状态机到状态栈上，也就是说，原来处于栈顶的状态机不移出状态栈，而是新的状态机通过调用初始化(`init`)函数，叠加到原来的栈顶状态机上，成为新的栈顶状态机。

返回式跳转是从状态栈移出一个或是多个状态机，移出的状态机都是通过调用各自的 `exit` 函数出来的。

如要需要回到 `standby` 状态，用户则可以在 `statelist.xl` 定义这样的事件：
`M_EVT_RETURN_ROOT`, Next State 为 `+>_return(STACKROOT)`

2.1.2 特殊状态机

在 `statelist.xls` 表中，存在两个特殊的状态机：前处理状态机(`preProc`)和后处理状态机(`postProc`)，如下图所示。

- 2、接着，必须创建 **Current State** 相关的.c 文件以及在.c 文件中添加相关代码，用以实现需要的功能，每个状态机代码必须包括 `init()`，`exit()`，`paint()`以及 `handle()`四个函数，函数名是前缀名加上 **Current State** 名去掉 `s_`以后生成的名字。如新添加的状态机的名字为 `s_set_repeat`，那么 `s_set_repeat.c` 中的四个函数的名字分别为：`initset_repeat()`，`exitset_repeat()`，`paintset_repeat()`和 `handleset_repeat()`。

状态机四个函数的功能定义如下表所示：

函数	功能
<code>init()</code>	状态机的初始化函数，用于资源初始化，指针内存的分配，调用底层模块的初始化等；
<code>exit()</code>	状态机的退出函数，用于内存的释放，底层模块函数的释放等；
<code>handle()</code>	事件处理函数，用于处理进入到当前状态机的所有事件；
<code>paint()</code>	状态机的绘图函数，用于界面的显示。

2.2.2 删除已有普通状态机

删除已有普通状态机步骤如下：

- 1、在源码中删除状态机的实现*.c 文件以及修改好与其所关联的代码。
- 2、在 `statelist.xls` 表中删除源为该状态机的所有状态跳转定义，删除所有可以跳转到该状态机的其它状态机中的事件触发。

2.2.3 特殊状态机

- 1、在 `statelist.xls` 表中在 `preProc/postProc` 新增一条事件处理函数。
- 2、在 `preProc/postProc` 对应源码目录源文件中，增加对应事件处理函数的实现。

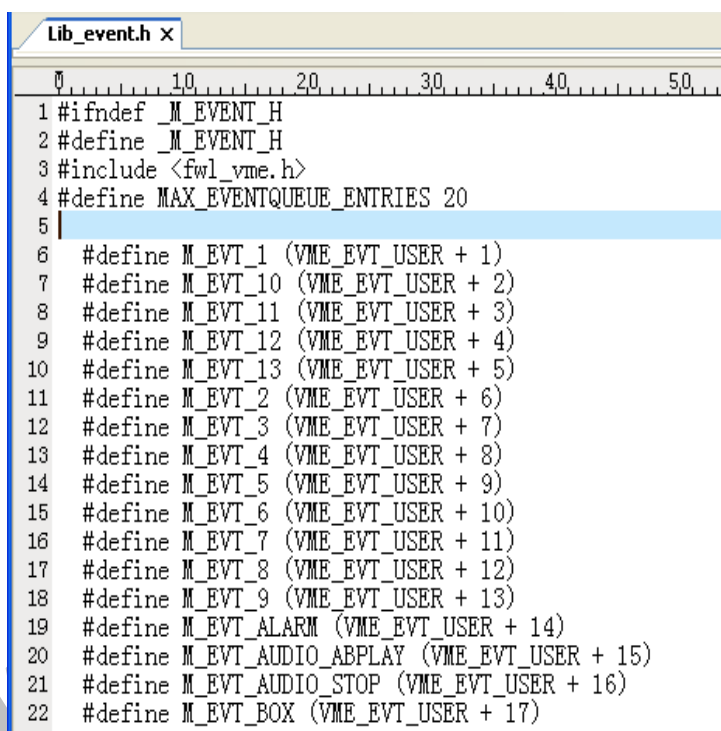
2.2.4 注意事项

- 1、加入或删除状态机后，需要更新状态机的定义文件，重新执行 `make2makee2.pl` 文件生成 `Lib_event.h`、`Lib_state.h` 和 `Lib_state.c`，然后执行 `make` 命令重新编译。

- `Lib_event.h` 定义状态机跳转的所有事件值，如图 2-3 所示。

- Lib_state.h 定义所有的状态机事件处理函数 ID，预处理和后处理的 function 事件处理函数 ID，状态机 ID，如图 2-4 所示。
- Lib_state.c 定义所有的状态机初始化、退出、状态机事件处理函数数组、事件处理函数数组、状态机结构数组，状态机 next 函数，如图 2-5 所示。

以上这三个文件在状态机维护均可不用修改，只维护 statelist.xls 文件即可，三个文件的内容完全由 perl 脚本运行时解析自动生成。如果在状态机的实现代码内需要引用事件定义值，请使用 include 包含 Lib_event.h 文件。



```

1 #ifndef _M_EVENT_H
2 #define _M_EVENT_H
3 #include <fwl_vme.h>
4 #define MAX_EVENTQUEUE_ENTRIES 20
5
6 #define M_EVT_1 (VME_EVT_USER + 1)
7 #define M_EVT_10 (VME_EVT_USER + 2)
8 #define M_EVT_11 (VME_EVT_USER + 3)
9 #define M_EVT_12 (VME_EVT_USER + 4)
10 #define M_EVT_13 (VME_EVT_USER + 5)
11 #define M_EVT_2 (VME_EVT_USER + 6)
12 #define M_EVT_3 (VME_EVT_USER + 7)
13 #define M_EVT_4 (VME_EVT_USER + 8)
14 #define M_EVT_5 (VME_EVT_USER + 9)
15 #define M_EVT_6 (VME_EVT_USER + 10)
16 #define M_EVT_7 (VME_EVT_USER + 11)
17 #define M_EVT_8 (VME_EVT_USER + 12)
18 #define M_EVT_9 (VME_EVT_USER + 13)
19 #define M_EVT_ALARM (VME_EVT_USER + 14)
20 #define M_EVT_AUDIO_ABPLAY (VME_EVT_USER + 15)
21 #define M_EVT_AUDIO_STOP (VME_EVT_USER + 16)
22 #define M_EVT_BOX (VME_EVT_USER + 17)

```

图 2-3 Lib_event.h

```

Lib_state.h x
0 10 20 30 40 50
1 #ifndef _M_STATE_H
2 #define _M_STATE_H
3 #include "Lib_event.h"
4
5
6 #define MAX_STACK_DEPTH 21
7
8 typedef enum
9 {
10     eM_s_init_system = 0,
11     eM_s_init_power_on = 1,
12     eM_s_stdb_standby = 2,
13     eM_s_audio_root = 3,
14     eM_s_audio_fetch_song = 4,
15     eM_s_audio_list_curply = 5,
16     eM_s_audio_list = 6,
17     eM_s_audio_list_menu = 7,
18     eM_s_audio_add_to_mylist = 8,
19     eM_s_audio_add_list_to_mylist = 9,
20     eM_s_audio_delete_cnfm = 10,
21     eM_s_audio_delete_all_cnfm = 11,
22     eM_s_audio_player = 12,

```

图 2-4 Lib_state.h

在 Lib_state.c 中每个状态机都会自动生成一个 GetNext 函数，预处理有 m_postprocGetNext，后处理有 m_preprocGetNext，一般的状态机，如状态机 s_audio_config，有 m_s_audio_configGetNext 等。GetNext 函数用于判断事件到来时，状态机的跳转及函数处理。

```

Lib_state.c x
0 10 20 30 40 50 60
818 static const M_STATESTRUCT m_postproc =
819 {
820     (void *)0,
821     (void *)0,
822     (void *)0,
823     m_postprocGetNext
824 };
825 static const M_STATESTRUCT m_preproc =
826 {
827     (void *)0,
828     (void *)0,
829     (void *)0,
830     m_preprocGetNext
831 };
832 static const M_STATESTRUCT m_s_audio_add_list_to_mylist =
833 {
834     initaudio_add_list_to_mylist,
835     exitaudio_add_list_to_mylist,
836     paintaudio_add_list_to_mylist,
837     m_s_audio_add_list_to_mylistGetNext
838 };

```

图 2-5 Lib_state.c

2、preProc 和 postProc 状态机是框架预置的两个状态机，这两个状态机必须存在，不能删除。postProc 状态机专门处理模块消息。如果当前状态机，处理完某个消息后，返回 1。系统内核就会跳转到 postProc 去处理这个消息。

3、事件的值定义由 perl 脚本扫描 Excel 配置表里的事件自动生成在 Lib_event.h 文件中。

注意：不同事件的个数不能超过 255 个。

2.3 事件跳转流程

为用户更好的理解状态机的处理机制及状态机事件的处理流程，现用框图来说明。

2.3.1 状态机内部跳转关系

状态机内部的四个函数 init()、exit()、paint()和 handle()的执行流程，如下图所示。

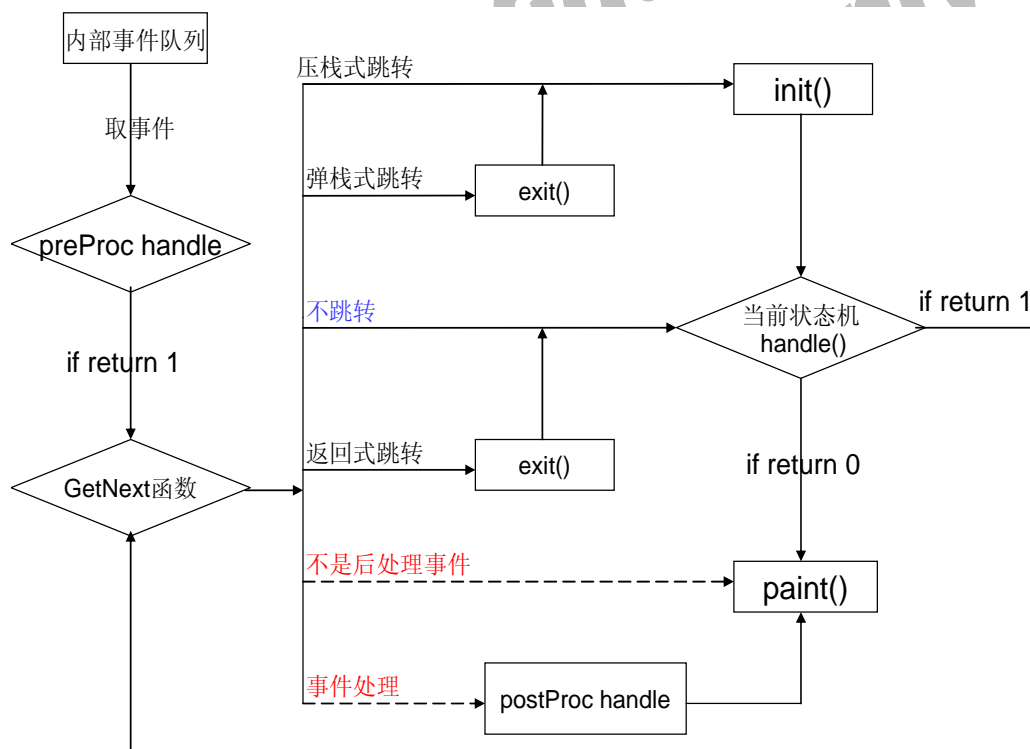


图 2-6 状态机内部跳转流程

现对上图说明如下，该流程图省略了预处理状态机的 GetNext 函数处理：

1、系统从内部事件队列获取事件，将取到的事件传入预处理状态机。如果预处理的事件处理函数的结果 `return 1`，就将事件传入当前状态机的 `GetNext` 函数（`GetNext` 函数是在 `Lib_state.c` 中自动生成的，用于判断事件传入时，状态机是什么类型的跳转方式）。

2、当前状态机的 `GetNext` 函数如图 2-7 所示，如果 `*pTrans` 为 `eM_TYPE_SCALL`，表示状态机不跳转；如果 `*pTrans` 为 `eM_TYPE_RETURN`，表示状态机要发生返回式跳转；如果 `*pTrans` 为 `eM_TYPE_RETURN_ROOT`，表示状态机要发生返回式跳转，状态机要跳转到 `standby` 状态，这种情况状态机根据状态栈的深度，要从状态栈中移出多个状态机；如果 `*pTrans` 为 `eM_TYPE_EXIT`，状态机要发生弹栈式跳转；如果 `*pTrans` 为 `eM_TYPE_IRPT`，状态机要发生压栈式跳转。`eM_TYPE_SCALL`（图 2-6 中蓝色字体显示）是普通状态机特有的，后处理或预处理状态机没有。

3、如果不需要跳转，就执行当前栈顶状态机的 `handle()` 函数。

4、如果 `handle()` 函数 `return 0`，就执行当前栈顶状态机的 `paint()` 函数，不会再把事件传入后处理状态机，然后回到第 1 步开始执行；如果 `handle()` 函数 `return 1`，就将事件传入后处理状态机的 `GetNext` 函数，跳转到第 10 步执行。

5、如果是返回式跳转，会根据返回的级数，执行 `exit()` 函数几次。

6、如果返回一级，会调用当前栈顶状态机的 `exit()` 函数，该状态机从状态栈中移出。然后执行当前栈顶状态机的 `handle()` 函数，然后从第 4 步开始执行。

7、如果返回 `n` 级，会依次从状态栈分别调用各自状态机的 `exit()` 函数，将 `n` 个状态机从状态栈中移出。然后执行当前栈顶状态机的 `handle()` 函数，然后从第 4 步开始执行。

8、如果是弹栈式跳转，会执行栈顶状态机的 `exit()` 函数，使状态机从状态栈中移出。然后执行新的栈顶状态机的 `init()` 函数，然后执行栈顶状态机的 `handle()`，然后从第 4 步开始执行。

9、如果是压栈式跳转，就执行新的栈顶状态机的 `init()` 函数，然后执行栈顶状态机的 `handle()`，然后从第 4 步开始执行。

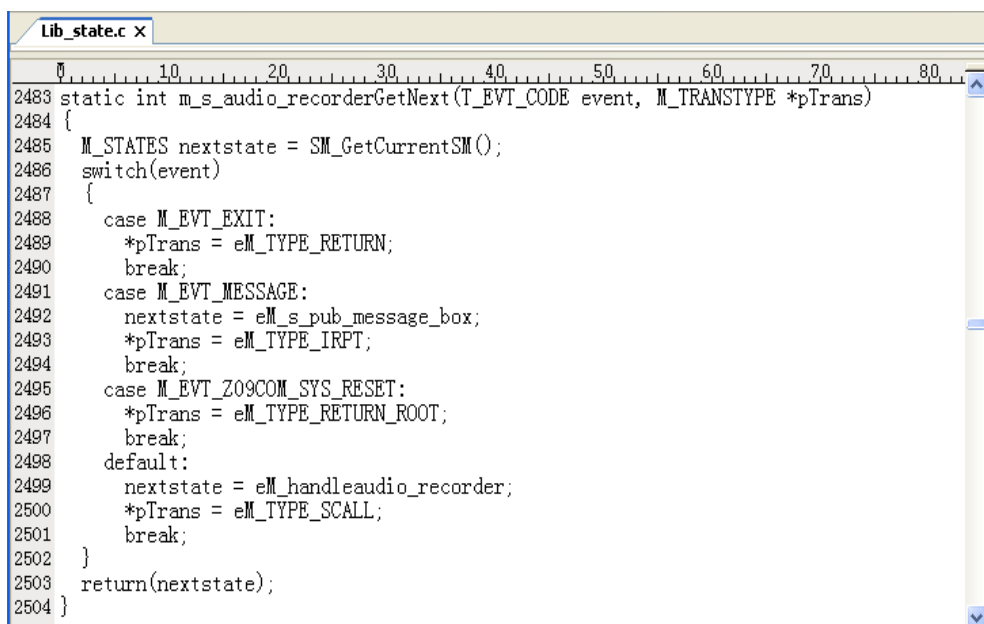
10、后处理的 `GetNext` 函数为 `m_postprocGetNext`（在 `Lib_state.c` 中自动生成的，用于判断事件传入时，状态机是跳转还是事件函数处理，或者事件不是后处理状态机要处理的事件），如图 2-8 所示。

11、如果 `*pTrans` 为 `eM_TYPE_IRPT`，状态机要发生压栈式跳转；如果 `*pTrans` 为 `eM_TYPE_ECALL`，表示状态机要执行事件处理函数；当后处理或预处理状态机不处理该事件时，`*pTrans` 为 `eM_TYPE_NONE`；`eM_TYPE_ECALL` 和 `eM_TYPE_NONE`（图 2-6 中红色字体显示）为后处理或预处理状态机特有的。

12、如果压栈式跳转，就执行第 9 步。

13、如果类型为事件处理，就执行 postproc handle 函数，然后执行当前状态机的 paint() 函数，然后回到第 1 步开始执行。

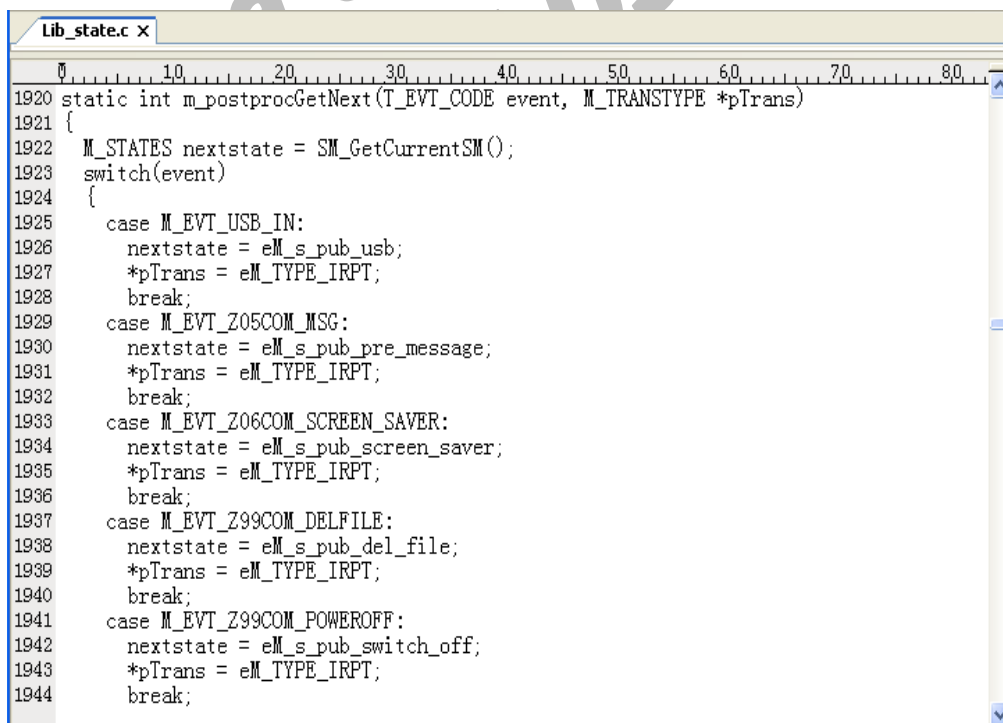
14、如果该事件不是后处理事件，执行当前状态机的 paint()函数，然后回到第 1 步开始执行。



```

Lib_state.c x
2483 static int m_s_audio_recorderGetNext(T_EVT_CODE event, M_TRANSTYPE *pTrans)
2484 {
2485     M_STATES nextstate = SM_GetCurrentSM();
2486     switch(event)
2487     {
2488     case M_EVT_EXIT:
2489         *pTrans = eM_TYPE_RETURN;
2490         break;
2491     case M_EVT_MESSAGE:
2492         nextstate = eM_s_pub_message_box;
2493         *pTrans = eM_TYPE_IRPT;
2494         break;
2495     case M_EVT_Z09COM_SYS_RESET:
2496         *pTrans = eM_TYPE_RETURN_ROOT;
2497         break;
2498     default:
2499         nextstate = eM_handleaudio_recorder;
2500         *pTrans = eM_TYPE_SCALL;
2501         break;
2502     }
2503     return(nextstate);
2504 }
    
```

图 2-7 当前状态机的 getNext 函数



```

Lib_state.c x
1920 static int m_postprocGetNext(T_EVT_CODE event, M_TRANSTYPE *pTrans)
1921 {
1922     M_STATES nextstate = SM_GetCurrentSM();
1923     switch(event)
1924     {
1925     case M_EVT_USB_IN:
1926         nextstate = eM_s_pub_usb;
1927         *pTrans = eM_TYPE_IRPT;
1928         break;
1929     case M_EVT_Z05COM_MSG:
1930         nextstate = eM_s_pub_pre_message;
1931         *pTrans = eM_TYPE_IRPT;
1932         break;
1933     case M_EVT_Z06COM_SCREEN_SAVER:
1934         nextstate = eM_s_pub_screen_saver;
1935         *pTrans = eM_TYPE_IRPT;
1936         break;
1937     case M_EVT_Z99COM_DELFILE:
1938         nextstate = eM_s_pub_del_file;
1939         *pTrans = eM_TYPE_IRPT;
1940         break;
1941     case M_EVT_Z99COM_POWEROFF:
1942         nextstate = eM_s_pub_switch_off;
1943         *pTrans = eM_TYPE_IRPT;
1944         break;
    
```

图 2-8 后处理的 getNext 函数

2.3.2 系统状态跳转

整个系统的事件处理流程如下图所示。

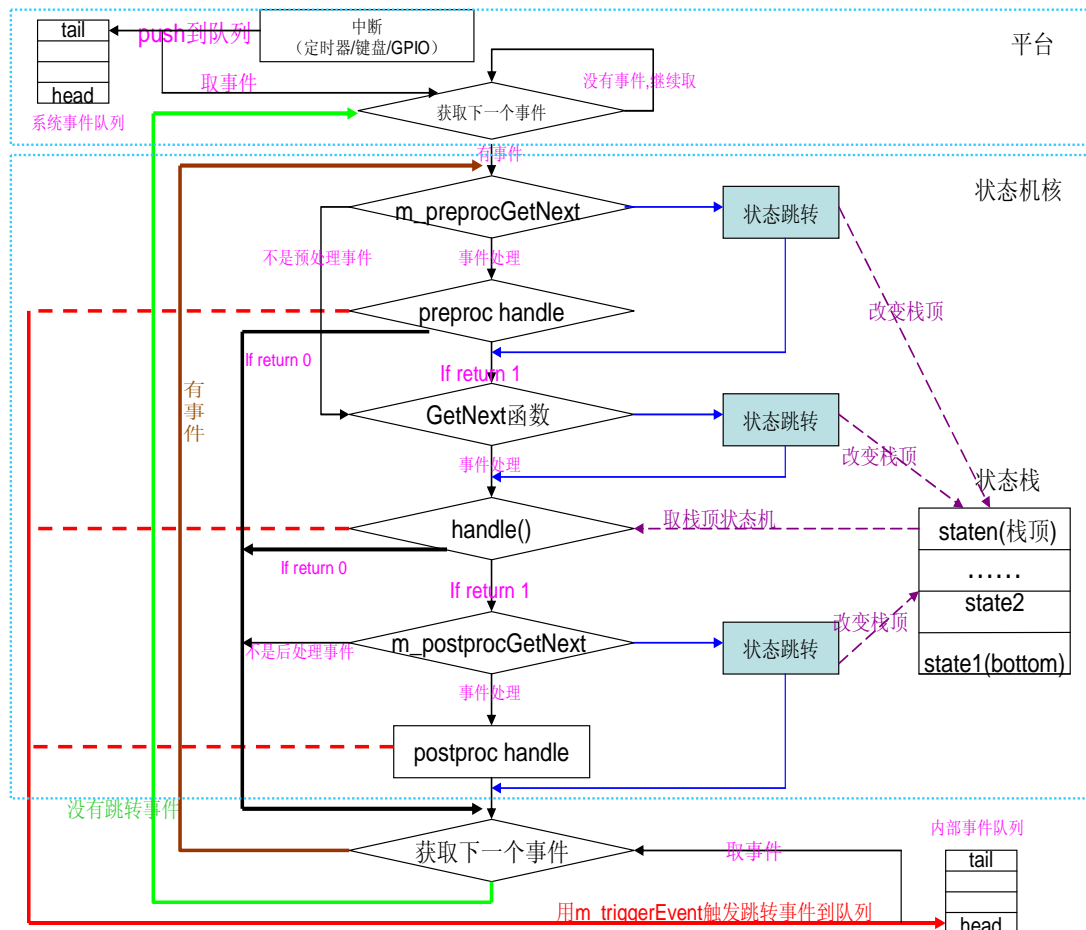


图 2-9 事件处理流程图

系统维护的两个事件队列分别是：系统事件队列和状态跳转事件队列（内部事件队列）。系统事件队列是由中断服务程序（如定时器、键盘、GPIO 等）或应用层的状态机产生的。状态跳转事件队列是用于状态机的跳转的，状态机会通过 `m_triggerEvent` 方式将状态跳转事件添加到队列。

下面详细说明整个系统的事件处理流程：

1、平台系统从系统事件队列取事件，如果取到系统事件，该事件将作为入口事件进入到状态机核，这部分流程见图 2-9 的最上面蓝色虚线框部分。状态机核的处理部分见中间蓝色虚线框部分。

2、进入到状态机核部分，首先特殊状态机 `preproc`（预处理）处理该事件。如果预处理处理完该事件后，`return 0`，就会执行第 5 步；如果预处理处理完该事件后，`return 1`，该事件会被传入当前栈顶状态机。

3、栈顶状态机对事件的处理与预处理状态机一样。如果栈顶状态机处理完该事件后，`return 0`，就会执行第 5 步；如果栈顶状态机处理完该事件后，`return 1`，该事件会被传入下一个特殊状态机 `postproc`（后处理）。

4、经过 `postproc`（后处理）状态机处理后，该系统事件的处理结束。

5、状态机核从内部事件队列获取下一个事件。如果能从内部事件队列取到事件，就从第 2 步开始执行；如果内部事件队列没有事件，就从第 1 步开始执行。

3 接口说明

3.1 子模块功能概述

详见“1.1 功能概述”。

3.2 术语定义

- **状态机 ID**

状态机的 ID 是唯一标识状态机的一个数值，在本模块的要求里，该 ID 为 `SM_GetStateArray` 返回状态机结构数组的下标值。

- **状态机**

平台接收事件的单位，以栈的形式管理状态机之间的切换，用状态机来实现每个不同的状态功能，如上网状态、待机状态。

- **事件处理 ID**

状态机的 `next` 函数在 `pTrans` 参数返回 `eM_TYPE_ECALL` 类型时，框架会调用用户实现的 `SM_GeteHandle` 函数，从其返回的数组里选择一个事件处理函数进行调用，事件处理 ID 也就是唯一标识事件处理函数的一个数值。在 `SMCore` 模块的要求里，该 ID 为 `SM_GeteHandle` 函数返回的事件处理数组下标值。

- **状态机事件处理 ID**

当 SMCore 调用状态机的 next 函数并传入一特殊事件 VME_EVT_USER 时，SMCore 要求状态机的 next 函数返回一个 ID，SMCore 用这个 ID 为数组下标到 SM_GetfHande 返回的状态机事件数组中找到状态机的事件处理函数调用。

● 状态机挂起

挂起就是压栈式栈顶状态机切换时，栈顶有新的状态机 B 入栈，原来的栈顶状态机 A 由栈顶 非栈顶的状态，对于 A 状态机来说，A 被挂起了。由于只有处于栈顶的状态机才能得到事件，因此被挂起就意味着不能再收到任何事件。

● 状态机唤醒

唤醒就是弹栈式栈顶状态机切换时，栈顶的状态机 A 出栈，原来处于栈顶下一位置的 B 状态机变为栈顶，对于 B 状态机来说，B 被唤醒。由于只有处于栈顶的状态机才能得到事件，因此状态机被唤醒就意味着能重新再收到事件。

3.3 数据结构定义

3.3.1 vT_EvtCode

16 位数据：因为库中以 16 位数据引用其，因此要求目标平台在包含接口头文件时将该类型定义为 16 位类型。

3.3.2 vT_EvtParam

指针：因为库中只会以 void* 指针引用其指针，因此目标平台在包含接口头文件时可以将该类型定义为任何的指针类型

3.3.3 M_STATESTRUCT

typedef struct

{

_fVoid pInit; //状态机的初始化函数的地址指针

_fVoid pExit; //状态机的退出函数的地址指针

_fVoid pPaint; //状态机的画图函数的地址指针

_fGetNextState pNext; //取将要转换到的状态机的函数的地址指针

```
} M_STATESTRUCT;
```

本数据结构用于用户定义其状态机的实现，SMCore 框架要求每个状态机提供如上 4 个函数，Init、Exit、Paint 函数调用的时刻见框架文档。

3.3.4 _fHandle

原 型	typedef unsigned char (*_fHandle)(vT_EvtCode event, vT_EvtParam* pEventParm);	
功能概述	状态机事件处理函数	
参数说明	Event	传入的事件值
	pParam	传入事件参数的地址指针
返回值说明	返回 1 则将事件交由下一状态处理，每一个事件的处理顺序为 preProcState→StackTopStack→postProcState	
注意事项	该函数指针被 SMCore 框架调用	

3.3.5 _feHandle

原 型	typedef unsigned char (*_feHandle)(vT_EvtCode* event, vT_EvtParam** pEventParm);	
功能概述	状态机事件处理函数	
参数说明	Event	传入的事件值
	pParam	传入事件参数的地址指针
返回值说明	返回 1 则将事件交由下一状态处理，每一个事件的处理顺序为 preProcState→StackTopStack→postProcState	
注意事项	该函数指针被 SMCore 框架调用	

3.3.6 _GetNextState

原 型	typedef int (*_fGetNextState)(vT_EvtCode event, M_TRANSTYPE *pTrans);	
功能概述	状态机跳转查询函数	
参数说明	Event	传入的事件值

原 型	typedef int (*_fGetNextState)(vT_EvtCode event, M_TRANSTYPE *pTrans);	
	pParam	传入事件参数的地址指针
返回值说明	详见下	
注意事项	该函数指针被 SMCore 框架调用	

_fGetNextState 函数为状态机调度的核心函数，当 SMCore 框架接每次收到事件时，都会通过当前状态机的该函数进行下一操作的查询。

pTrans	返回值	SMCore 逻辑
eM_TYPE_SCALL		调用栈顶状态机对应的 handler 事件处理函数
eM_TYPE_ECALL	事件处理 ID	跟据事件处理 ID 调用 SM_GeteHandle 返回的事件处理函数集合中的对应 ID 事件处理函数
eM_TYPE_NONE	无意义	SMCore 会调用当前栈顶状态机的事件处理函数
eM_TYPE_EXIT	弹栈式状态机跳转的目标状态机 ID	SMCore 会进行弹栈式的 A->B 的状态机切换。依次调用 A 状态机的退出函数、B 状态机的初始化函数，B 状态机的 handler 函数
eM_TYPE_IRPT	压栈式状态机跳转的目标状态机 ID	SMCore 会进行弹栈式的 A+>B 的状态机切换。将 A 状态机休眠，B 状态放入栈顶并调用 B 状态机的初始化函数
eM_TYPE_RETURN_ROOT	无意义	SMCore 会进行状态机的弹栈处理，栈顶 S1 到栈底 Sn，SMCore 会将 S1 到 Sn-1 依次弹栈，只剩下栈底 Sn。每个状态弹栈之前，SMCore 会调用状态机的退出函数。所有 S1-Sn-1 均弹栈完毕后，会调用栈底状态机 Sn 的事件处理函数

pTrans	返回值	SMCore 逻辑
eM_TYPE_RETURN 或 >eM_TYPE_RETURN	无意义	SMCore 会进行状态机的弹栈处理，栈顶 S1 到栈底 Sn，SMCore 从 S1 开始，将 pTrans-eM_TYPE_RETURN 个状态机依次弹栈，每个状态弹栈之前，SMCore 会调用状态机的退出函数。以上都弹栈完毕后，SMCore 会调用弹栈后栈顶状态机的事件处理函数

3.3.7 _fVoid

```
typedef void (*_fVoid)(void);
```

状态机初始化、退出、绘制、挂起、唤醒函数原型

3.3.8 M_TRANSTYPE

```
typedef enum //在状态机的调度中，根据 type 的值来进行相应的动作
```

```
{
```

```
// all function call types
```

```
eM_TYPE_SCALL, //当 type 为此值时，可调用某状态机里的控制程序
```

```
eM_TYPE_ECALL, //当 type 为此值时，可调用常用的那些控制程序
```

```
eM_TYPE_NONE, //当 type 为此值时，不进行任何调用
```

```
// all transistion types
```

```
eM_TYPE_EXIT, //当 type 为此值时，栈顶的状态机先 pop 出去，然后一个状态机被 push 进来，成为栈顶状态机
```

```
eM_TYPE_IRPT, //当 type 为此值时，一个状态机被 push 进堆栈里，成为新的栈顶状态机，原栈顶状态机被保存起来，堆栈指针加 1,栈深度加 1
```

```
// Do not change after here!
```

```
eM_TYPE_RETURN_ROOT, //当 type 为此值时，状态机堆栈的堆栈指针回到栈底
```

```
eM_TYPE_RETURN      //当 type 为此值时，栈顶状态机被 pop 出去，堆栈指针减 1，
                      栈深度减 1
} M_TRANSTYPE;
```

3.4 接口函数列表

3.4.1 m_initStateHandler

原 型	void m_initStateHandler(void);
功能概述	初始化 SMCore 模块
参数说明	-
返回值说明	-
注意事项	主要对状态机管理模块的特性进行一些配置初始化(事件队列空间、状态机栈空间)等；该初始化动作必须在使用库其它接口函数之前进行
调用示例	m_initStateHandler();

3.4.2 m_mainloop

原 型	void m_mainloop(vT_EvtCode Event, vT_EvtParam *pParam)	
功能概述	状态机管理模块的主循环	
参数说明	Event	触发事件
	pParam	事件参数的地址指针
返回值说明	-	
注意事项	-	

原 型	void m_mainloop(vT_EvtCode Event, vT_EvtParam *pParam)
调用示例	<pre> vT_EvtCode event = 0; vT_EvtParam evtParm; while(1) { if(GetInterruptEvent(event, &evtParm)) { m_mainloop(mailbox.event, &mailbox.param); } } //GetInterruptEvent 为从系统中断的事件队列里取事件的同步函数 </pre>

3.4.3 m_triggerEvent

原 型	void m_triggerEvent(vT_EvtCode event, vT_EvtParam* pEventParm);	
功能概述	把触发事件加进事件队列中	
参数说明	event	触发事件
	pEventParm	事件参数的地址指针
返回值说明	-	
注意事项	-	
调用示例	-	

3.4.4 m_regSuspendFunc

原 型	void m_regSuspendFunc(_fVoid pfSuspend);	
功能概述	注册挂起函数	
参数说明	pfSuspend	挂起函数的地址指针
返回值说明	-	
注意事项	在压栈式栈顶状态机切换的情况下，调度栈顶状态机，这个时候 SMCore 模块会调用由用户通过本函数注册的 pfSuspend 指针所指向的处理函数，通过该函数通知状态机被挂起。	

原 型	<code>void m_regSuspendFunc(_fVoid pfSuspend);</code>
调用示例	<pre>void s_stdb_standby_Suspend (void) { puts("s_stdb_standby machine Suspend!"); } m_regSuspendFunc (s_stdb_standby_Suspend);</pre>

3.4.5 m_regResumeFunc

原 型	<code>void m_regResumeFunc(_fVoid pfResume);</code>	
功能概述	注册唤醒程序	
参数说明	pfResume	唤醒程序的地址
返回值说明	-	
注意事项	在弹栈式栈顶状态机切换的情况下，调度栈顶状态机，这个时候 SMCore 模块会调用由用户通过本函数注册的 pfResume 指针所指向的处理函数，通过该函数通知状态机被唤醒。	
调用示例	<pre>void s_stdb_standby_resume(void) { puts("s_stdb_standby machine resumed!"); } m_regResumeFunc(s_stdb_standby_resume);</pre>	

3.4.6 SM_GetCurrentSM

原 型	<code>M_STATES SM_GetCurrentSM(void);</code>	
功能概述	返回当前栈顶状态机 ID	
参数说明	-	
返回值说明	返回当前栈顶状态机 ID，即在状态机数组的下标值	
注意事项	-	
调用示例	<code>M_STATES nextstate = SM_GetCurrentSM();</code>	

3.4.7 SM_GetStackMaxDepth

原 型	unsigned int SM_GetStackMaxDepth(void);
功能概述	在当前状态机栈缓冲区配置(通过 SM_GetStackBuffer 接口配置)的情况下, 返回可用的最大的状态机栈深度
参数说明	-
返回值说明	-
注意事项	-
调用示例	-

3.4.8 SM_CalcStackBufByMaxDepth

原 型	unsigned int SM_CalcStackBufByMaxDepth(unsigned int maxDepth);	
功能概述	计算状态机栈深度为 maxDepth 时, 目标平台应提供的状态机栈缓冲区的大小	
参数说明	maxDepth	计算时使用的状态机栈深度值
返回值说明	状态机栈缓冲区大小, 以字节为单位	
注意事项	-	
调用示例	-	

3.4.9 SM_GetEventMaxEntries

原 型	unsigned int SM_GetEventMaxEntries(void);
功能概述	在当前事件缓冲区配置(通过 SM_GetEventQueueBuffer 接口配置)的情况下, 事件缓冲区可以容纳触发事件的最大数量
参数说明	-
返回值说明	返回可容纳的触发事件的最大数量
注意事项	-
调用示例	-

3.4.10 SM_CalcEventBufferByMaxEntries

原 型	unsigned int SM_CalcEventBufferByMaxEntries(unsigned int maxEntries);	
功能概述	计算事件数量最大值为 maxEntries 时的事件队列所需缓冲区大小	
参数说明	maxEntries	计算时使用的事件队列的事件数量
返回值说明	返回事件队列缓冲区大小，以字节为单位	
注意事项	-	
调用示例	-	

4 状态机库的特性配置

在平台集成状态机库依赖几个目标平台对状态机库特性的要求接口，以下函数都是在 m_initStateHandler 执行时由状态机库内部调用，由平台实现这些函数来配置状态机库状态机栈缓冲区、事件队列缓冲区，状态机库所管理的状态机集合、事件处理函数集合、状态机事件处理函数集合。

- SM_GetStateArray
- SM_GetfHande
- SM_GeteHandle
- SM_GetStackBuffer
- SM_GetEventQueueBuffer
- SM_GetPreProcID
- SM_GetPostProcID

接口实现说明详见“5依赖接口列表”。

5 依赖接口列表

5.1 状态机管理库配置

以下接口由使用状态机管理库的平台实现，供状态机管理库在初始化时配置其特性时调用。

5.1.1 SM_GetStateArray

原 型	const M_STATESTRUCT** SM_GetStateArray(void);
功能概述	取得状态机集合数组的首地址
参数说明	-
返回值说明	取得状态机集合数组的首地址
注意事项	-

5.1.2 SM_GetfHandle

原 型	const _fHandle* SM_GetfHandle(void);
功能概述	取得状态机事件处理函数集合数组的首地址
参数说明	-
返回值说明	返回取得状态机事件处理函数集合数组的首地址
注意事项	-

5.1.3 SM_GeteHandle

原 型	const _feHandle* SM_GeteHandle(void);
功能概述	取得事件处理函数集合数组的首地址
参数说明	-
返回值说明	返回事件处理函数集合数组的首地址
注意事项	-

5.1.4 SM_GetStackBuffer

原 型	void *SM_GetStackBuffer(unsigned int *bufSize);
功能概述	取得供状态机栈用的连续内存
参数说明	bufSize 输出参数，用于返回连续内存的字节大小
返回值说明	返回连续内存的首址
注意事项	-

5.1.5 SM_GetEventQueueBuffer

原 型	void *SM_GetEventQueueBuffer(unsigned int *bufSize);	
功能概述	取得状态机的事件队列存贮的连续内存	
参数说明	bufSize	输出参数，用于返回连续内存的字节大小
返回值说明	返回连续内存的首址	
注意事项	-	

5.1.6 SM_GetEvtReturn

原 型	vT_EvtCode SM_GetEvtReturn(void);	
功能概述	返回用户用于状态机弹栈的触发事件 ID，以便弹栈 B→A 时能否以该事件值通知 A 当前 A 被唤醒了	
参数说明	-	
返回值说明	-	
注意事项	为了兼容现有平台应用层逻辑而设，可以通过 m_regResumeFunc 注册唤醒回调实现	

5.1.7 SM_GetEvtNoNext

原 型	vT_EvtCode SM_GetEvtNoNext(void);	
功能概述	返回查询状态机事件处理函数 ID 的事件	
参数说明	-	
返回值说明	事件 ID	
注意事项	<p>SMCore 模块在需要调用状态机事件处理函数前，会调用状态机切换 next 函数时并入该接口返回事件值查询状态机事件处理函数的 ID</p> <p>因此，所有状态机的 next 函数实现要注意当传入的事件值为该值时，要返回状态机事件处理函数 ID</p>	

5.1.8 SM_GetPreProcID

原 型	M_STATES SM_GetPreProcID(void);
功能概述	取预处理状态机的 ID
参数说明	
返回值说明	返回预处理状态机的 ID
注意事项	预处理状态机详见其它文档

5.1.9 SM_GetPostProcID

原 型	M_STATES SM_GetPostProcID(void);
功能概述	取后处理状态机的 ID
参数说明	-
返回值说明	返回后处理状态机的 ID
注意事项	预处理状态机详见其它文档

5.2 目标平台调试接口

依赖于如下目标平台调试接口：

函 数	备注
T_VOID AkAssertDispMsg(T_pCSTR message, T_pCSTR filename, T_U32 line);	无
T_S32 Dbg_Trace(T_U32 mID, T_pCSTR format, ...);	无