# MTK内存管理篇

MTK 手机用的操作系统是 nucleus, 这是一个RTOS(实时操作系统)，这个RTOS本身是不带内存管理功能，所以MTK自己写的内存管理。(nucleus在系统初始化完毕时，会调用Application\_Initialize，参数就是可使用内存的起始地址)

大体上来分，MTK内存可以分为3种：

* **control buffer**

平常使用的OslMalloc就是这个内存。

这个内存内部实现是按块来划分的(pool)，具体的配置可custom\_config.c 文件里的custom\_config\_ctrl\_buff\_info()里看到里面的size 指定了这个块的大小，no\_of\_buff 指定了有多少个这样的块。

按块来管理内存，可以有效地控制内存的碎片，管理也相对简单，可能会造成内存的一些浪费对于手机这种需要长时间运行不重起的设备来说，还是很有必要的。

MTK 默认最大块的大小为 2048byte，也就是2k。这就是说用OslMalloc 分配内存默认最大能分配到2k，这个可以看custom\_config\_ctrl\_buff\_info()配置可以看到。可以通过修改里面的配置来改变这个值，不过一般不这么做，因为MTK提供了其他的内存管理方式

* **system buffer**

system buffer 平时我们用不到，听名字也是系统使用的。

主要是提供 run-time usage,是一块 semi-static memory(什么意思？)比如 block of task, task stack ,control block of control buffer ,buffer pool等等在 custom\_config.c里面配置 ，主要有两个宏 GLOBAL\_MEM\_SIZE  和 GLOBAL\_DEBUG\_MEM\_SIZE两个 static 数组 static kal\_uint32 System\_Mem\_Pool[GLOBAL\_MEM\_SIZE/sizeof(kal\_uint32)];  和static kal\_uint32 Debug\_Mem\_Pool[GLOBAL\_DEBUG\_MEM\_SIZE/sizeof(kal\_uint32)];  
为了满足时间要求，也就是要求快速分配，系统内存又分为 internal system memory 和 system memory前者link 到 internal SRAM ,后者link 到 external SRAM

* **app buffer**

app的内存是使用通过MTK 提供的一种ADM(application dynamic memory)机制来实现，ADM 主要的功能是通过管理一个数组来实现内存的分配。

ADM 也是通过内存块(pool)来实现的，具体无法看到其代码。app通过这个adm这个机制，可以更加灵活的使用内存，比如分配大内存(大于2k)等等

**主要函数**  
 创建 adm  
 KAL\_ADM\_ID kal\_adm\_create(void \*mem\_addr, kal\_uint32 size, kal\_uint32 \*subpool\_size, kal\_bool islogging);  
  
 删除函数  
 kal\_status kal\_adm\_delete(KAL\_ADM\_ID adm\_id);  
  
 分配函数   
 extern void \*kal\_adm\_internal\_alloc(KAL\_ADM\_ID adm\_id, kal\_uint32 size, char \*filename, kal\_uint32 line);  
 #define kal\_adm\_alloc(adm\_id, size) kal\_adm\_internal\_alloc(adm\_id, size, \_\_FILE\_\_, \_\_LINE\_\_)  
  
 释放函数  
 extern void kal\_adm\_free(KAL\_ADM\_ID adm\_id, void \*mem\_addr);

在 [MTK 内存管理简单总结](http://blog.csdn.net/yanwuxufeng/archive/2010/07/12/5730092.aspx) 中，大体说了MTK的三种内**存分配方式**，对于第三种，也就是app buffer，是比较丰富的一种。

在 MTK 平台中也有许多具体的实现。在代码里搜索一下 kal\_adm\_create 就可以发现有许多地方使用了。

看一个比较典型的使用：

在文件app\_mem.c里，有两个memory pool，一个是用于应用之间共享内存，另一个是用于屏幕内存。

第一种内存，主要是用于各种应用之间共享内存(以下简称ASM)，这样可以节省内存，MTK实现了一种机制，可以在多个应用之间共享内存，当当前应用想获得的共享内存不足时，MTK会通知后台应用释放相应的内存。这套机制在AppMemMgr.c里面实现。

先看一下初始化该内存次池函数

void applib\_mem\_ap\_init(void (\*stop\_finish\_callback\_by\_MMI)(kal\_uint32 app\_id, kal\_uint32 string\_id, kal\_bool result))

这个函数带有一个参数，这个参数是一个函数指针，该回调函数有3个参数，app\_id,(应用id),string\_id 和 result。

这个回调函数比较特别，是当一个后台应用 被 要求释放内存，释放完毕后调用的。

为什么要搞这么一个函数，因为一些应用比较复杂，释放内存的同时需要关闭一些资源，而这些动作是异步的，等这些异步发的操作多完成时，调用一些函数，告诉ASM，内存释放完毕。

具体实现：通过 调用 kal\_adm\_create 来创建一个内存池，然后保存了一些回调函数，没有什么特别的地方，内存池的大小 是 APPLIB\_MEM\_AP\_POOL\_SIZE 来确定的，可以通过修改 app\_asm\_pool\_union 来修改内存池的大小。

应用分配内存

void \*applib\_mem\_ap\_alloc(kal\_uint32 app\_id, kal\_uint32 mem\_size)

应用通过上面的函数来获得ASM的内存，参数一 app\_id，是当前分配内存的id，这个id需要自己定义，并且注册(下文说明)，

参数二是实际需要分配的内存大小。

具体实现：先mem\_size 进行了处理，让其四字节对齐。然后通过 kal\_adm\_alloc 获得内存，不过这个内存加上了一个头结构和尾结构，(头和尾都加入了特殊字符，再释放时进行检查，这个可以判断内存是否越界)。然后把这个内存插入到list的头部。

昨天说到了内存的分配。下面看一下**内存释放**

主要进行了3步：

static void applib\_mem\_ap\_free\_int(void \*mem\_ptr)

{

/\*----------------------------------------------------------------\*/

/\* Local Variables

/\*----------------------------------------------------------------\*/

applib\_mem\_header\_struct \*header, \*prev\_node, \*remove\_node;

applib\_mem\_footer\_struct \*footer;

/\*----------------------------------------------------------------\*/

/\* Code Body

/\*----------------------------------------------------------------\*/

if (g\_applib\_mem\_cntx.app\_pool\_id) /\* Normal mode \*/

{

ASSERT(mem\_ptr && APPLIB\_MEM\_ALIGNED\_4(mem\_ptr));

header = ((applib\_mem\_header\_struct\*) mem\_ptr) - 1;

footer = (applib\_mem\_footer\_struct\*) (((char\*)mem\_ptr) + header->chunk\_size);

ASSERT(APPLIB\_MEM\_COMP\_PATTERN(header->guard\_pattern, APPLIB\_MEM\_HEADER\_PATTERN1) &&

APPLIB\_MEM\_COMP\_PATTERN(footer->guard\_pattern, APPLIB\_MEM\_FOOTER\_PATTERN1));

/\*

\* Remove the block from linked list

\*

\* It is not a fast algorithm, we can improve it by using double linked list,

\* but we choose simpler design because

\* 1. Typically total allocation count is small

\* 2. We don't want to increase space overheads

\* 3. We don't want to access KAL ADM internal data structure

\*/

prev\_node = &g\_applib\_mem\_cntx.app\_head;

ASSERT(prev\_node->next);

for (remove\_node = prev\_node->next;

remove\_node;

prev\_node = remove\_node, remove\_node = prev\_node->next)

{

if (remove\_node == header)

{

break;

}

}

ASSERT(remove\_node);

prev\_node->next = remove\_node->next;

/\* Set guard pattern \*/

APPLIB\_MEM\_SET\_PATTERN(header->guard\_pattern, APPLIB\_MEM\_HEADER\_PATTERN2);

APPLIB\_MEM\_SET\_PATTERN(footer->guard\_pattern, APPLIB\_MEM\_FOOTER\_PATTERN2);

/\* Release the block \*/

#ifdef APPLIB\_MEM\_USE\_ADM

kal\_adm\_free(g\_applib\_mem\_cntx.app\_pool\_id, header);

#else

free(header);

#endif

ASSERT(g\_applib\_mem\_cntx.app\_alloc\_count > 0);

g\_applib\_mem\_cntx.app\_alloc\_count--;

}

else /\* Full pool mode \*/

{

ASSERT(mem\_ptr == g\_applib\_mem\_ap\_pool && g\_applib\_mem\_cntx.app\_alloc\_count == 1);

g\_applib\_mem\_cntx.app\_alloc\_count = 0;

g\_applib\_mem\_cntx.app\_id\_of\_full\_pool = APPLIB\_MEM\_AP\_ID\_DUMMY; /\* 0 \*/

#ifdef APPLIB\_MEM\_USE\_ADM

g\_applib\_mem\_cntx.app\_pool\_id = kal\_adm\_create(

g\_applib\_mem\_ap\_pool,

APPLIB\_MEM\_AP\_POOL\_SIZE,

(kal\_uint32\*) g\_applib\_mem\_pool\_chunk\_size,

KAL\_FALSE);

#else /\* APPLIB\_MEM\_USE\_ADM \*/

g\_applib\_mem\_cntx.app\_pool\_id = APPLIB\_DUMMY\_POOL\_ID;

#endif /\* APPLIB\_MEM\_USE\_ADM \*/

}

}

1. 取得内存的头部和尾部，(调试版本可以判断内存是否越界)
2. 从链表中删除这个节点
3. 调用 kal\_adm\_free 释放内存

在[MTK 内存管理简单总结 2](http://blog.csdn.net/yanwuxufeng/archive/2010/07/13/5733138.aspx) 提到调用 applib\_mem\_ap\_alloc 分配内存是需要一个应用id，这个id是需要自己增加，

而且在调用这个函数之前必须 调用 applib\_mem\_ap\_register 注册这个id。需要注意的是最后一个参数，是一个回调函数，这个回调函数是在共享内存不够使用时，ASM会调用这个函数，告诉应用需要释放共享内存，供其它应用使用。增加 id  在 app\_mem.h 的 applib\_mem\_ap\_id\_enum 里面，只要添加一个id就可以。

同样 屏幕内存也是通过ADM来管理，屏幕内存 是用来 创建 layer 用的，在MTK的某个版本开始，创建layer的内存是有要求的，

需要applib\_mem\_screen\_alloc 分配的内存。

在 [MTK 内存管理简单总结 4](http://blog.csdn.net/yanwuxufeng/archive/2010/07/16/5740999.aspx) 提到，当内存不足时，可以调用 mmi\_frm\_appmem\_prompt\_to\_release\_mem 来显示后台应用，让用户去选择关闭后台应用释放内存，下面看一下具体流程：当选者某一后台应用，选择stop 按键时，会触发 mmi\_frm\_appmem\_prompt\_screen\_stop\_highlighted 函数 (触发过程跟MTK的UI事件机制想关，以后分析)。

http://www.cnblogs.com/Images/OutliningIndicators/ContractedBlock.gifmmi\_frm\_appmem\_prompt\_screen\_stop\_highlighted  
该函数没有做什么，只是通过选择list 选项 来获得应该被stop的后台应用id。接着进入到确认界面 ( mmi\_frm\_appmem\_entry\_stop\_confirm\_screen ), 当用户选择yes后，进入到mmi\_frm\_appmem\_stop\_confirm\_screen\_yes 函数，这个函数执行真正的释放内存动作

static void mmi\_frm\_appmem\_stop\_confirm\_screen\_yes(void)

{

g\_mmi\_frm\_appmem\_prompt.stop\_state = MMI\_FRM\_APPMEM\_STOP\_STATE\_WAITING;

//调用相应应用注册的函数,通知应用却释放内存

//这个就是为什么应用要用共享内存前,需要调用 applib\_mem\_ap\_register 来注册的原因

applib\_mem\_ap\_stop\_app\_by\_MMI\_task(g\_mmi\_frm\_appmem\_prompt.app\_id\_confirm\_to\_stop);

MMI\_TRACE(MMI\_FW\_TRC\_G1\_FRM, MMI\_FRM\_ASM\_STOP\_CONFIRM\_YES,

g\_mmi\_frm\_appmem\_prompt.app\_id\_confirm\_to\_stop,

g\_mmi\_frm\_appmem\_prompt.stop\_state);

//判断释放是否完成，如果没有完成，那么显示正在处理界面

//这里不直接认为回调完应用的就直接认为内存释放了，是因为

//应用stop的过程有可能是一个异步操作，这一点需要特别注意

if (g\_mmi\_frm\_appmem\_prompt.stop\_state != MMI\_FRM\_APPMEM\_STOP\_STATE\_FINISHED)

{

/\* mmi\_frm\_appmem\_stop\_finished\_handler() was not invoked inside

applib\_mem\_ap\_stop\_app\_by\_MMI\_task(), we display progressing

screen to wait for mmi\_frm\_appmem\_stop\_finished\_handler() \*/

mmi\_frm\_appmem\_entry\_progressing\_screen();

}

}

注意点：

相应的app 在释放完毕内存后，一定要applib\_mem\_ap\_notify\_stop\_finished 来通知 ASM内存管理器，内存释放成功或者失败。

该函数最后会调用 mmi\_frm\_appmem\_stop\_finished\_handler 来通知 内存管理器，内存释放结果。

这套机制，回调函数调来调去，一开始让人有点晕，不过这么设计主要是考虑应用的异步性，有些应用停止是一些异步动作，需要等到异步动作结束才能知道最后结果。

# MTK MMI event总结

在MTK MMI 里面有各种event，最常见的跟交互相关的 按键  event，触摸屏 event。跟各种具体事件比如电话event，短信event，电量event，信号量event，timer event等等。

MTK 都有相应的处理方式，跟交互相关的按键(KeyBrd.c)和触摸(TouchScreen.c) 各有一套自己的机子，timer event 已经在 [MTK timer 小结 3](http://blog.csdn.net/yanwuxufeng/archive/2010/07/20/5751255.aspx) 介绍过，还有一套机制就是处理各种其他事件，它主要分为普通事件interrupt event，interrupt event 主要用于需要弹出的对话框的event。

常用事件机制。做过MTK开发的童鞋应该知道要接受 MTK L4 层的消息，用一个注册函数 SetProtocolEventHandler，注册一个event 的 处理函数。比如短信，电话，电池电量，信号量等等都是通过 这个函数来注册消息。当L4 层，处理完封装事件后，就会把这个event 发送到MMI task 里来(消息发送可以看 [MTK task 小结 5](http://blog.csdn.net/yanwuxufeng/archive/2010/07/28/5770303.aspx) )，然后MMI task 通过这个 event 机制，找到相应的处理函数，进行处理。实现这种机制，也是考虑到零活性和扩张性。如果都在MMI task 里面，用 switch case 来处理，那就很疯狂了，长度不说，写个应用，定义个消息，都要去改MMI task，所以这个机制实现虽然比较简单，但是还是很有必要的。

昨天也说到，这个机制的 event 主要有两种，普通event 和 intrrupt event(中断事件)，这些主要是一些需要中断当前应用的事件(主要是看那些弹出框)，比如电话，有些应用需要提前处理这个消息，还要处理 这个事件处理完毕后的处理。还是看代码吧

在MMI task 最后，会调用函数 ProtocolEventHandler，这个函数是就是找相应的event 的相应处理函数(平台不一样，可能实际的函数名字有些区别，但是流程基本上一样的，我这里被 #define 到 mmi\_frm\_execute\_current\_protocol\_handler)。

// MsgStruct 是 具体事件的 消息体

void mmi\_frm\_execute\_current\_protocol\_handler(U16 eventID, void \*MsgStruct, int mod\_src, void \*peerBuf)

{

U16 count = 0;

PsExtPeerFuncPtr currFuncPtr = NULL;

U8 interrup\_result = MMI\_FALSE; /\* False not handle interrupt, True will handle \*/

MMI\_BOOL query\_result = MMI\_FALSE, execute\_result = MMI\_FALSE;

interrupt\_event\_hdlr int\_func = NULL, post\_int\_func = NULL;

mmi\_frm\_int\_event\_type current\_frm\_int\_event;

{

// 遍历 protocolEventHandler 查找 是否有event 注册了回调函数

// 虽然这个方法感觉比较笨，就一个数组,实际上对速度没有什么影响

for (count = 0; count < maxProtocolEvent; count++)

{

if (protocolEventHandler[count].eventID == eventID)

{

//找到 处理函数

currFuncPtr = (PsExtPeerFuncPtr) protocolEventHandler[count].entryFuncPtr;

// 这个地方时，一个优化

// 处理的事情就是 把刚才现在处理的 event id 往前移动一个单位

// 这是考虑到 程序的局部性原理， 这个优化还是很有必要的

if (count > 0)

{

protocolEventHandler[count].eventID = protocolEventHandler[count - 1].eventID;

protocolEventHandler[count].entryFuncPtr = protocolEventHandler[count - 1].entryFuncPtr;

protocolEventHandler[count - 1].eventID = eventID;

protocolEventHandler[count - 1].entryFuncPtr = (PsFuncPtr) currFuncPtr;

}

break;

}

}

}

// 这个就是查找是否是中断事件

current\_frm\_int\_event = mmi\_frm\_interrupt\_event\_converter(eventID, MsgStruct);

if (current\_frm\_int\_event > 0)

{

// 查看改 中断事件是否有 注册 提前处理函数

query\_result = mmi\_frm\_query\_interrupt\_event\_information(current\_frm\_int\_event, &int\_func, &post\_int\_func);

}

// 如果有 调用该函数

if (query\_result && int\_func)

{

/\* New interruption mechanism \*/

execute\_result = (\*int\_func)(current\_frm\_int\_event);

}

//根据 event 相应的回调函数 以及 前面处理的结果，来觉得是否处理该事件

// 要注意的是，如果如果 interrput 提前处理函数 返回 true，那么 这里就不会执行

if ((currFuncPtr) && (!interrup\_result) && (!execute\_result))

{

(\*currFuncPtr) (MsgStruct, mod\_src, peerBuf);

}

else

{

MMI\_TRACE(MMI\_FW\_TRC\_G1\_FRM, MMI\_FRM\_INFO\_EVENT\_EXECURPTO\_NO\_HDLR, eventID);

}

// interrupt event 最后处理函数

if (query\_result && post\_int\_func)

{

execute\_result = (\*post\_int\_func)(current\_frm\_int\_event);

}

}

看完这个，其实觉得挺简单的，对吧。就是一个数组，数组里关联了 event id 和 对应的处理函数。

还有就是增加了一种 intrrupt event。可以注册这种event 的 pre\_handler  and post\_handler。这样如果需要，可以提前做处理或者不响应该事件。

普通 event 的执行过程。中间还有几个函数没有介绍，这里简单介绍一下

// 这个函数就是简单把 L4c 的interrupt消息，转换成 MMI 层interrupt的消息

static mmi\_frm\_int\_event\_type mmi\_frm\_interrupt\_event\_converter(U16 event\_id, void \*msg)

{

mmi\_frm\_int\_event\_type frm\_interrupt\_event = 0;

switch (event\_id)

{

// GPIO 消息

case PRT\_EQ\_GPIO\_DETECT\_IND:

{

mmi\_eq\_gpio\_detect\_ind\_struct \*gpio\_detect\_ind = (mmi\_eq\_gpio\_detect\_ind\_struct \*)msg;

switch (gpio\_detect\_ind->gpio\_device)

{

// 翻盖关闭

case EXT\_DEV\_CLAM\_CLOSE:

{

frm\_interrupt\_event = MMI\_FRM\_INT\_CLAM\_CLOSE;

break;

}

// 翻盖打开

case EXT\_DEV\_CLAM\_OPEN:

{

frm\_interrupt\_event = MMI\_FRM\_INT\_CLAM\_OPEN;

break;

}

// 耳机插入 拔出

case EXT\_DEV\_EARPHONE:

{

if (gpio\_detect\_ind->on\_off == 1)

{

frm\_interrupt\_event = MMI\_FRM\_INT\_EARPHONE\_PLUG\_IN;

}

else

{

frm\_interrupt\_event = MMI\_FRM\_INT\_EARPHONE\_PLUG\_OUT;

}

break;

}

}

break;

}

// 充电信息，

case PRT\_BATTERY\_STATUS\_IND:

{

mmi\_eq\_battery\_status\_ind\_struct \*battery\_status\_ind = (mmi\_eq\_battery\_status\_ind\_struct\*)msg;

frm\_interrupt\_event = mmi\_frm\_get\_frm\_int\_event\_macro(

battery\_status\_ind->battery\_status,

mmi\_frm\_int\_event\_battery\_table,

sizeof(mmi\_frm\_int\_event\_battery\_table) / sizeof(mmi\_frm\_int\_event\_battery\_table[0]));

/\* For low battery indication \*/

if (battery\_status\_ind->battery\_status == PMIC\_VBAT\_STATUS)

{

if (battery\_status\_ind->battery\_voltage == BATTERY\_LOW\_WARNING)

{

frm\_interrupt\_event = MMI\_FRM\_INT\_BATTERY\_LOW\_WARNING;

}

else if (battery\_status\_ind->battery\_voltage == BATTERY\_LOW\_TX\_PROHIBIT)

{

frm\_interrupt\_event = MMI\_FRM\_INT\_BATTERY\_LOW\_TX\_PROHIBIT;

}

}

break;

}

case MSG\_ID\_TIMER\_EXPIRY:

case MSG\_ID\_MMI\_EQ\_POWER\_ON\_IND:

case MSG\_ID\_MMI\_EQ\_KEYPAD\_DETECT\_IND:

{

break;

}

default:

{

//其他 interrupt event，主要是弹出框相关，短信报告等等

frm\_interrupt\_event = mmi\_frm\_get\_frm\_int\_event\_macro(

event\_id,

mmi\_frm\_int\_event\_convert\_table,

sizeof(mmi\_frm\_int\_event\_convert\_table) / sizeof(mmi\_frm\_int\_event\_convert\_table[0]));

break;

}

}

return frm\_interrupt\_event;

}

这个函数就是把 L4C 的消息转换到 MMI 的消息，然后通过 mmi\_frm\_query\_interrupt\_event\_information 这个函数去查找，是否有相应的注册。这个函数比较简单，就是数组里去比较。

还有一个常用的函数是 SetProtocolEventHandler，也就是注册一个消息的地回调函数。

具体也没有什么可说的，无非就是循环查找是否有这个 event。

1，如果有，看func 是否为空，为空表示去掉这个event 的注册，不为空，就直接修改 这个event 的 func

2，没有，就增加一个event。

这里注意，由一个小小的优化，就是判断，event 数组的最后是否为无效的 event，如果无效，那么减少总注册event 的个数。这个样可以减少 循环查询的次数。

今天就饥饿着总结MMI event，今天打算说说key event。MTK 的 key envent 管理其实跟 [MTK MMI event 小结 2](http://blog.csdn.net/yanwuxufeng/archive/2010/07/29/5775323.aspx) 说的protocol event 机制也差不了多少，简单来说，就是app 通过注册某一个key 的回调函数，当这个key事件产生是，该机制就去找到相应的回调函数，进行回调。

先说说key 的 event 有哪些：

1 key down 按键按下 KEY\_EVENT\_DOWN

2 key up    按键弹起 KEY\_EVENT\_UP

3 key long press 按键长按  KEY\_LONG\_PRESS。 默认好像是长按1.5 记不清楚了，可以自己设置，函数Kbd\_SetLongPressTime，不过除非必不得已，不要去随便修改，可能会引起其他应用出问题，要修改，也要 先获得当前值，退出应用时设置回原来的值。

4 key repeat 按键按下一直不放，发送该消息 KEY\_REPEAT。 举个简单例子，pc 上当输入时，如果长按 a 键，那么回不停的增加a，这个就是repeat动作。这个的时间好像是0.5s，也就是按住键 0.5 s 之后，每隔0.5s，就会发送一个repeat 消息。设置函数Kbd\_SetRepeatTime

还有两个不常用的 KEY\_HALF\_PRESS\_DOWN和KEY\_HALF\_PRESS\_UP，也就是说的 2step 可以。

先讲一下按键的处理流程。

当 按键下按键之后，MMI task 会收到 MSG\_ID\_MMI\_EQ\_KEYPAD\_DETECT\_IND这个消息，在系统初始化的时候，已经注册了这个消息的处理函数 mmi\_frm\_key\_handle。也就是说 mmi\_frm\_key\_handle 是处理按键过程的函数。这个函数 mmi\_frm\_key\_handle 收到这个消息后，回去一个 通过 该消息带的函数，去取当前发生的按键事件(实际这个函数就是一个按键缓存队列里去第一个，用一个按键缓存队列，可以防止按键大量的丢失，可以起到同步驱动 和MMI层，驱动产生按键事件很快而MMI有可能比较慢，可以简单的理解为生产者和消费者的关系，跑题了)。 取到当前按键后，看当前是否可以进行操作，比如是否有触摸笔按下等等。

昨天说了一下key event 的基本情况，今天直接从代码开始吧

先看执行key event 函数 mmi\_frm\_key\_handle

void mmi\_frm\_key\_handle(void \*paraBuff)

{

kbd\_data k;

U32 msg\_count;

mmi\_eq\_keypad\_detect\_ind\_struct \*p;

// 判断参数是否为空

// 在 初始化时，系统就注册了 MSG\_ID\_MMI\_EQ\_KEYPAD\_DETECT\_IND 的回调函数为 mmi\_frm\_key\_handle

// 当收到这个消息是，表示有按键事件要处理，

// 这个paraBuff 是 keyTask 传过来的参数，里面的内容是获取按键信息的函数指针

if (paraBuff != NULL)

{

drv\_get\_key\_func new\_key\_ptr;

p = (mmi\_eq\_keypad\_detect\_ind\_struct\*) paraBuff;

new\_key\_ptr = (drv\_get\_key\_func) (p->func);

//判断是否是新的函数指针，如果是的话，需要进行一些清理工作

if (new\_key\_ptr != keypad\_ptr)

{

// 新的函数指针，则进行清理，并且判断是否有按键需要处理

if (!mmi\_kbd\_process\_keyptr\_change((void \*)new\_key\_ptr))

{

return;

}

//清理 按键 事件缓存buffer

ClearKeyEvents();

keypad\_ptr = new\_key\_ptr;

}

}

// 通过 while 1 来 不停的从 按键缓存里获取按键信息

// 当然这里不会无限死循环，后面会进行相应处理

while (1)

{

//1)判读是否是挂起，2)是否key需要处理

if ((g\_mmi\_suspend\_key\_flag == MMI\_FALSE) && keypad\_ptr &&((\*(keypad\_ptr))(&k) == MMI\_TRUE))

{

if (k.Keydata[0] != KEY\_INVALID)

{

#if defined(\_\_MMI\_TOUCH\_SCREEN\_\_) || defined(\_\_MMI\_HANDWRITING\_PAD\_\_)

kal\_bool is\_pen\_enabled, is\_pen\_down;

// 获得触摸屏状态

mmi\_pen\_get\_state(&is\_pen\_enabled, &is\_pen\_down);

#endif /\* defined(\_\_MMI\_TOUCH\_SCREEN\_\_) || defined(\_\_MMI\_HANDWRITING\_PAD\_\_) \*/

#if defined(\_\_MMI\_TOUCH\_SCREEN\_\_) || defined(\_\_MMI\_HANDWRITING\_PAD\_\_)

//如果触摸屏按下，那么保存状态

if (is\_pen\_down && (k.Keyevent == WM\_KEYPRESS))

{

U16 KeyMapIndex;

// 不允许 按键处理

is\_allow\_key\_action = MMI\_FALSE;

// 把驱动的按键和MMI 按键进行转换

KeyMapIndex = mmi\_frm\_get\_idx\_from\_device\_key\_code(k.Keydata[0]);

if (nKeyPadStatus[KeyMapIndex] == KEY\_EVENT\_UP)

{

KEYBRD\_MESSAGE KeyBrdMsg;

KeyBrdMsg.nKeyCode = nKeyPadMap[KeyMapIndex].nMMIKeyCode;

if (mmi\_frm\_is\_2step\_keyCode(KeyBrdMsg.nKeyCode))

{

nKeyPadStatus[KeyMapIndex] = KEY\_HALF\_PRESS\_DOWN;

pressKey = HALF\_DOWN\_STATUS;

key\_is\_pressing\_count++;

}

else

{

nKeyPadStatus[KeyMapIndex] = KEY\_EVENT\_DOWN;

pressKey = FULL\_DOWN\_STATUS;

key\_is\_pressing\_count++;

}

}

}

else

#endif /\* defined(\_\_MMI\_TOUCH\_SCREEN\_\_) || defined(\_\_MMI\_HANDWRITING\_PAD\_\_) \*/

//如果按键是弹起，那么还原原来按键状态，也就是弹起状态

//

if ((k.Keyevent == WM\_KEYRELEASE) && (is\_allow\_key\_action == MMI\_FALSE))

{

#if defined(\_\_MMI\_TOUCH\_SCREEN\_\_) || defined(\_\_MMI\_HANDWRITING\_PAD\_\_)

/\* Reset allow key flag and Update key status even if pen is down\*/

U16 KeyMapIndex;

KeyMapIndex = mmi\_frm\_get\_idx\_from\_device\_key\_code(k.Keydata[0]);

if ((nKeyPadStatus[KeyMapIndex] == KEY\_EVENT\_DOWN)

|| (nKeyPadStatus[KeyMapIndex] == KEY\_HALF\_PRESS\_DOWN)

|| (nKeyPadStatus[KeyMapIndex] == KEY\_LONG\_PRESS)

|| (nKeyPadStatus[KeyMapIndex] == KEY\_REPEAT))

{

nKeyPadStatus[KeyMapIndex] = KEY\_EVENT\_UP;

key\_is\_pressing\_count--;

}

prevKeyMapIndex = prevKeyMapIndex;

#endif /\* defined(\_\_MMI\_TOUCH\_SCREEN\_\_) || defined(\_\_MMI\_HANDWRITING\_PAD\_\_) \*/

is\_allow\_key\_action = MMI\_TRUE;

}

// 按键是否可以处理

if (is\_allow\_key\_action == MMI\_TRUE)

{

if (((k.Keyevent == WM\_KEYPRESS) && (mmi\_kbd\_get\_key\_is\_pressing\_count() == 0))

|| k.Keyevent == DRV\_WM\_KEYLONGPRESS || k.Keyevent == DRV\_WM\_KEYREPEATED || k.Keyevent == DRV\_WM\_KEYFULLPRESS)

{

#if defined(\_\_MMI\_TOUCH\_SCREEN\_\_) || defined(\_\_MMI\_HANDWRITING\_PAD\_\_)

//关闭触摸屏

mmi\_pen\_disable();

#endif

}

//真正处理按键事件

mmi\_frm\_convert\_process\_key\_event(k.Keyevent, k.Keydata[0]);

if ((k.Keyevent == WM\_KEYRELEASE) && (mmi\_kbd\_get\_key\_is\_pressing\_count() == 0))

{

#if defined(\_\_MMI\_TOUCH\_SCREEN\_\_) || defined(\_\_MMI\_HANDWRITING\_PAD\_\_)

//重新打开触摸屏

mmi\_pen\_enable();

#endif

}

}

}

// 获得MMI task 消息队列消息个数

msg\_get\_ext\_queue\_info(mmi\_ext\_qid, &msg\_count);

//如果外部有消息或者内部消息，则跳出该循环

if (msg\_count > 0 || OslNumOfCircularQMsgs() > 0)

{

//表示还有key需要处理，在MMI task 里会直接调用这个函数进行再处理

g\_keypad\_flag = MMI\_TRUE;

break;

}

}

else

{

// 没有key消息要处理

g\_keypad\_flag = MMI\_FALSE;

break;

}

} /\* while(1) \*/

MMI\_TRACE(MMI\_FW\_TRC\_G1\_FRM, MMI\_FRM\_KEY\_HDLR\_END);

}

1） 这里key 用while (1)来处理，不停的从key 消息buffer里取出按键信息，然后处理，这么做可以防止task中不停的发送消息，可以理解为共享内存。同样当有其他消息来时，需要跳出来处理其他消息，处理完毕后还需要再来处理。

2）这里跟触摸屏的事件进行了冲突处理，也就是按键和触摸屏不能同时工作，这两者也没有优先级。

3）实际真正处理按键 是在 mmi\_frm\_convert\_process\_key\_event里面，下次在分析

在 [MTK MMI event 小结 5](http://blog.csdn.net/yanwuxufeng/archive/2010/08/06/5792234.aspx) 中，提到了event 处理函数 mmi\_frm\_key\_handle，这个函数主要作用是判断是否需要处理按键，从按键缓存里面持续的读取按键信息，然后调用 mmi\_frm\_convert\_process\_key\_event 进行处理。这个函数没有什么可说的，最多是在屏幕旋转的情况下，把 导航键 转换一下，接着它调用了 ProcessKeyEvent， 这个函数主要是对于一些状态的处理，防止key down 和up 不成对，出现混乱。

void ProcessKeyEvent(U32 MsgType, U16 DeviceKeyCode)

{

MMI\_BOOL isKeyPaired;

U16 KeyMapIndex;

/\*----------------------------------------------------------------\*/

/\* Code Body \*/

/\*----------------------------------------------------------------\*/

// 按键影射，把驱动的按键码，转换成MMI 的 按键消息

KeyMapIndex = mmi\_frm\_get\_idx\_from\_device\_key\_code(DeviceKeyCode);

if (KeyMapIndex >= MAX\_KEYS)

{

return;

}

// 处理各种按键事件，没有什么可以多说的，

// 主体结构都一样，

// 1 判断状态是否正常

// 2 正常 则调用 KeyEventHandler 处理， 否则忽略该事件

if (MsgType == WM\_KEYPRESS)

{

// 这里处理 多案件同时按下的情况。这里需要硬件支持

if ((KeyMapIndex != prevKeyMapIndex) && (g\_kbd\_concurrent\_key\_mode == CONCURRENT\_KEY\_MODE\_1\_KEY))

{

isKeyPaired = (nKeyPadStatus[prevKeyMapIndex] == KEY\_EVENT\_UP);

prevKeyMapIndex = KeyMapIndex;

}

//判断案件状态是否正常，防止不匹配

if (nKeyPadStatus[KeyMapIndex] == KEY\_EVENT\_UP)

{

KEYBRD\_MESSAGE KeyBrdMsg;

KeyBrdMsg.nKeyCode = nKeyPadMap[KeyMapIndex].nMMIKeyCode;

if (mmi\_frm\_is\_2step\_keyCode(KeyBrdMsg.nKeyCode))

{

nKeyPadStatus[KeyMapIndex] = KEY\_HALF\_PRESS\_DOWN;

key\_is\_pressing\_count++;

KeyBrdMsg.nMsgType = KEY\_HALF\_PRESS\_DOWN;

}

else

{

nKeyPadStatus[KeyMapIndex] = KEY\_EVENT\_DOWN; /\* same with KEY\_FULL\_PRESS\_DOWN \*/

key\_is\_pressing\_count++;

KeyBrdMsg.nMsgType = KEY\_EVENT\_DOWN;

}

// 处理按键事件

KeyEventHandler((KEYBRD\_MESSAGE\*) & KeyBrdMsg);

}

else

{

/\* Ignore the event \*/

}

}

else if (MsgType == WM\_KEYRELEASE)

{

if ((nKeyPadStatus[KeyMapIndex] == KEY\_EVENT\_DOWN)

|| (nKeyPadStatus[KeyMapIndex] == KEY\_LONG\_PRESS)

|| (nKeyPadStatus[KeyMapIndex] == KEY\_REPEAT) || (nKeyPadStatus[KeyMapIndex] == KEY\_HALF\_PRESS\_DOWN))

{

KEYBRD\_MESSAGE KeyBrdMsg;

nKeyPadStatus[KeyMapIndex] = KEY\_EVENT\_UP;

key\_is\_pressing\_count--;

KeyBrdMsg.nMsgType = KEY\_EVENT\_UP;

KeyBrdMsg.nKeyCode = nKeyPadMap[KeyMapIndex].nMMIKeyCode;

KeyEventHandler((KEYBRD\_MESSAGE\*) & KeyBrdMsg);

}

else

{

/\* Ignore the event \*/

}

}

/\* ++Robin, modified by Max Chen \*/

else if (MsgType == DRV\_WM\_KEYLONGPRESS)

{

if (nKeyPadStatus[KeyMapIndex] == KEY\_EVENT\_DOWN)

{

KEYBRD\_MESSAGE KeyBrdMsg;

nKeyPadStatus[KeyMapIndex] = KEY\_LONG\_PRESS;

KeyBrdMsg.nMsgType = KEY\_LONG\_PRESS;

KeyBrdMsg.nKeyCode = nKeyPadMap[KeyMapIndex].nMMIKeyCode;

KeyEventHandler((KEYBRD\_MESSAGE\*) & KeyBrdMsg);

}

else

{

/\* Ignore the event \*/

}

}

else if (MsgType == DRV\_WM\_KEYREPEATED)

{

if ((nKeyPadStatus[KeyMapIndex] == KEY\_LONG\_PRESS) || (nKeyPadStatus[KeyMapIndex] == KEY\_REPEAT))

{

KEYBRD\_MESSAGE KeyBrdMsg;

nKeyPadStatus[KeyMapIndex] = KEY\_REPEAT;

KeyBrdMsg.nMsgType = KEY\_REPEAT;

KeyBrdMsg.nKeyCode = nKeyPadMap[KeyMapIndex].nMMIKeyCode;

KeyEventHandler((KEYBRD\_MESSAGE\*) & KeyBrdMsg);

}

else

{

/\* Ignore the event \*/

}

}

else if (MsgType == DRV\_WM\_KEYFULLPRESS)

{

/\*

\* Only in two-stage key will have KEY\_FULL\_PRESS\_DOWN, and it followed after KEY\_HALF\_PRESS\_DOWN

\*/

if (nKeyPadStatus[KeyMapIndex] == KEY\_HALF\_PRESS\_DOWN)

{

KEYBRD\_MESSAGE KeyBrdMsg;

nKeyPadStatus[KeyMapIndex] = KEY\_EVENT\_DOWN;

KeyBrdMsg.nMsgType = KEY\_EVENT\_DOWN;

KeyBrdMsg.nKeyCode = nKeyPadMap[KeyMapIndex].nMMIKeyCode;

KeyEventHandler((struct KEYBRD\_MESSAGE\*)&KeyBrdMsg);

}

else

{

/\* Ignore the event \*/

}

}

else if ((MsgType == DRV\_WM\_ENABLE\_TWOKEY\_DETECTION) || (MsgType == DRV\_WM\_ENABLE\_THREEKEY\_DETECTION) ||

(MsgType == DRV\_WM\_DISABLE\_MULTIKEY\_DETECTION))

{

/\* Ignore the event \*/

}

else

{

MMI\_TRACE(MMI\_FW\_TRC\_G6\_FRM\_DETAIL, MMI\_FRM\_ERROR\_PROC\_KEYEVENT\_HDLR);

MMI\_ASSERT(0);

}

}

KeyEventHandler 函数主要判断是否要真的处理该事件，可以看成是一个按键事件的拦截，比如应用切换过程中，需要一个切换动画，而这个动画工程中，需要处理忽略这些按键。就需要特殊的处理。

static void KeyEventHandler(KEYBRD\_MESSAGE \*eventKey)

{

MMI\_BOOL is\_hdlr\_enabled = MMI\_TRUE;

// 主要处理一些特殊相应：屏幕背光，屏幕锁定，按键声音

mmi\_kbd\_app\_key\_hdlr(eventKey);

// 判断是否有前置处理函数

if (g\_mmi\_frm\_cntx.kbd\_pre\_func)

{

is\_hdlr\_enabled = g\_mmi\_frm\_cntx.kbd\_pre\_func(eventKey);

}

// 根据前置处理函数结果，判断是否要处理该 key event

if (is\_hdlr\_enabled)

{

//处理 案件事件

ExecuteCurrKeyHandler((S16) eventKey->nKeyCode, (S16) eventKey->nMsgType);

}

// 是否有后置处理函数，可以进行一些监视

if (g\_mmi\_frm\_cntx.kbd\_post\_func)

{

g\_mmi\_frm\_cntx.kbd\_post\_func(eventKey);

}

}

接下来是 ExecuteCurrKeyHandler 这个函数就是根据 按键事件，获得处理函数，进行处理。

void ExecuteCurrKeyHandler(S16 keyCode, S16 keyType)

{

FuncPtr currFuncPtr = NULL;

// 重新设定 键盘锁和屏保timer

mmi\_idle\_restart\_keypad\_lock\_timer();

mmi\_idle\_restart\_screensaver\_timer();

frm\_p->currKeyCode = keyCode;

frm\_p->currKeyType = keyType;

// 对电话状态下,挂电话键的特殊处理.

if (frm\_p->currKeyType == KEY\_EVENT\_DOWN &&

isInCall() && !GetWapCallPresent() &&

IsBitReset(g\_mmi\_frm\_cntx.end\_key\_flag, frm\_p->currKeyType) )

{

RegisterEndKeyHandlerInCall();

}

// 获得按键处理函数

currFuncPtr = currKeyFuncPtrs[keyCode][keyType];

// 导航的选择键，默认和左功能键的效果一样。

if (keyCode == KEY\_ENTER && currFuncPtr == NULL)

{

if (currKeyFuncPtrs[KEY\_ENTER][KEY\_EVENT\_UP] == NULL &&

currKeyFuncPtrs[KEY\_ENTER][KEY\_EVENT\_DOWN] == NULL &&

currKeyFuncPtrs[KEY\_ENTER][KEY\_HALF\_PRESS\_DOWN] == NULL &&

currKeyFuncPtrs[KEY\_ENTER][KEY\_REPEAT] == NULL && currKeyFuncPtrs[KEY\_ENTER][KEY\_LONG\_PRESS] == NULL)

{

currFuncPtr = currKeyFuncPtrs[KEY\_LSK][keyType];

}

}

// 处理按键 消息

if (currFuncPtr)

{

(\*currFuncPtr)();

}

}

// 重置状态

if (keyType == KEY\_EVENT\_UP)

{

frm\_p->currKeyCode = KEY\_INVALID;

frm\_p->currKeyType = MAX\_KEY\_TYPE;

}

}

从整个按键事件的处理流程来看，也没有什么特殊的地方，就是收到消息后，从按键buffer里取出按键事件，然后处理。如果有有别的事件 要处理，那么就break出来，等到处理完这个消息后，在 MMI task 调用 mmi\_frm\_key\_handle 继续处理剩下的按键事件。

这里无非多了很多的判断，是否要进行按键处理，MTK的代码函数名字上虽然看不出什么东西，但是一层一层函数，每一层函数的功能还是比较独立的，所以也不是很难看懂。

到这，key event 的处理基本上就算完成了。其他几个常用的函数注册key 处理函数了。了解了按键event的处理过程之后(其实也是比较简单的)，注册函数就很好了解了，就是把相应key 和 event 的数组里放入一个回调函数的指针。比如

void SetKeyHandler(FuncPtr funcPtr, U16 keyCode, U16 keyType) 抛开其他语句，就是一句简单的实现currKeyFuncPtrs[keyCode][keyType] = funcPtr; 还有可以设置一组按键的处理函数，

void SetGroupKeyHandler(FuncPtr funcPtr, PU16 keyCodes, U8 len, U16 keyType)，这个很好理解，keyCodes 就是一个按键数组，len是这个数组的长度。实现也很简单，就是一个for循环，把这些keyCodes的回调函数都写成 funcPtr。这里就不再说了(其实也没有什么可以说得了)

其他还有：

void mmi\_frm\_kbd\_reg\_pre\_key\_hdlr(PsKeyProcessCBPtr func) ; //注册按键预处理函数，[MTK MMI event 小结 6](http://blog.csdn.net/yanwuxufeng/archive/2010/08/08/5797645.aspx) 解释过

void mmi\_frm\_kbd\_set\_tone\_state(mmi\_frm\_kbd\_tone\_state\_enum state); // 设置按键音。需要说明一点，在开发一个应用程序的时候，有些手机状态一般先保存一下进入程序前的状态，在自己的app 退出后还原状态。比如这个按键音，背光长亮等等

接下来简单说一下触摸屏事件，其实触摸屏的实现跟key event 几乎一样。也是通过pen event 缓存队列来获得pen 事件。来看几个代码.

先看一下初始化函数：mmi\_pen\_init

void mmi\_pen\_init(void)

{

g\_pen\_initialized = KAL\_TRUE;

// 看到这两个函数指针，是不是感觉很熟悉？对，就是跟key 一样，前置处理和后置处理函数

// 这个看多了，不用细看代码就能猜到了。

g\_pen\_cntx.pre\_pen\_callback\_fp = NULL;

g\_pen\_cntx.post\_pen\_callback\_fp = NULL;

// 设置 手写输入法的区域，手写输入法在 触摸屏这里需要特殊处理。

mmi\_pen\_stop\_capture\_strokes();

// 设置触摸屏采样时间间隔，参数一是在普通状态下，采样时间间隔，参数二是手写输入法情况下采样时间间隔

mmi\_pen\_config\_sampling\_period(MMI\_PEN\_SAMPLING\_PERIOD\_1, MMI\_PEN\_SAMPLING\_PERIOD\_2);

// 设置超时时间间隔

mmi\_pen\_config\_timeout\_period(MMI\_PEN\_LONGTAP\_TIME, MMI\_PEN\_REPEAT\_TIME, MMI\_PEN\_STROKE\_LONGTAP\_TIME);

// 设置move 的间距，也就是两个点之间多少个pixel 算move 事件。

mmi\_pen\_config\_move\_offset(

MMI\_PEN\_SKIP\_MOVE\_OFFSET,

MMI\_PEN\_SKIP\_STROKE\_MOVE\_OFFSET,

MMI\_PEN\_SKIP\_LONGTAP\_OFFSET,

MMI\_PEN\_SKIP\_STROKE\_LONGTAP\_OFFSET);

// 设置drv 的回调函数，也就是当产生事件时，会回调这个函数

touch\_panel\_cb\_registration(mmi\_pen\_touch\_panel\_sendilm, NULL);

// 设置MMI 层消息 MSG\_ID\_TP\_EVENT\_IND 的处理函数

SetProtocolEventHandler(mmi\_pen\_touch\_panel\_event\_ind, MSG\_ID\_TP\_EVENT\_IND);

}

在驱动drv 这一层，触摸屏有一个task-- MOD\_TP\_TASK，在处理 事件，MOD\_TP\_TASK 会回调用 touch\_panel\_cb\_registration 注册的函数，有pen event 事件产生了，这个去pen event 缓存里去，这里注册的是：mmi\_pen\_touch\_panel\_sendilm 函数，这个函数就是简单的向MMI task 发送一个消息 MSG\_ID\_TP\_EVENT\_IND，让MMI task 来处理。这里这么做的原因也很简单，drv 的task 优先级是很高的，要处理的东西也很多，所以希望回调函数能很快处理完这个事情（发送一个消息很快），这里可以想一下 pc 上的中断，这两者差不多。当然中断还有更多其他原因，比如内核态和用户态等等，需要回调函数尽可能快处理，发送消息到消息队列(扯远了。。。)。

     还有一点需要说明的是，现在触摸屏的手机越来越普遍，那么该善触摸屏的体验是非常重要的，可惜的是MTK 的又不是多点触摸，处理起来很费劲。这个时候就需要充分利用好这里的几个设置，采样时间间隔，以及move 事件距离。需要时，让自己app可以捕获尽可能多的点，然后自己进行计算，来判断用户行为。这对改善体验很有帮助。当然这个也是比较耗电的，不需要时，一 定要设置回普通状态。

在 [MTK MMI event 小结 7](http://blog.csdn.net/yanwuxufeng/archive/2010/08/09/5800121.aspx) 中，说到，init 时候， 注册了触摸屏消息MSG\_ID\_TP\_EVENT\_IND 的处理函数 mmi\_pen\_touch\_panel\_event\_ind，

static void mmi\_pen\_touch\_panel\_event\_ind(void \*param )

{

// 这个用于控制 只有一个 MSG\_ID\_TP\_EVENT\_IND 在 消息队列里

g\_pen\_is\_driver\_indication\_in\_queue = 0;

// 判断是否启动了timer 在定时读取 pen event 缓存

if (!g\_pen\_polling\_timer\_started)

{

//真正的处理函数

mmi\_pen\_poll\_hdlr();

}

}

这个函数其实没有做什么真正的处理，只是简单的判断的了一下是否要处理这个消息。这里需要注意的是， 跟按键事件不同的是，这里通过timer来控制处理 pen event 消息缓存里剩余的消息， 按键事件 是在 MMI task 消息循环里，直接调用 按键事件的处理函数。从要实现的效果上没有什么太多的区别。

static void mmi\_pen\_poll\_hdlr(void)

{

TouchPanelEventStruct data;

MMI\_BOOL is\_stroke\_move = MMI\_FALSE;

MMI\_BOOL has\_incoming\_event = MMI\_FALSE;

MMI\_BOOL delay\_polling\_timer = MMI\_FALSE;

MMI\_BOOL pen\_abort\_happen = MMI\_FALSE;

U16 interval = 0;

g\_pen\_polling\_timer\_started = MMI\_FALSE;

ResetBit(g\_input\_msg\_in\_queue, MMI\_DEVICE\_PEN);

// 判断 pen event 是否开启，按键和pen 是冲突的。

if (!g\_pen\_cntx.is\_enabled)

{

return;

}

// 判断是否还有数据要处理

// delay\_polling\_timer 表示是否启动了timer，来稍后处理这些pen event 事件，开始时初始化为 false

while (!delay\_polling\_timer && mmi\_pen\_lookahead\_buffer\_get\_event(&data))

{

mmi\_pen\_point\_struct pos;

// 检测背光情况，是否要开启

mmi\_idle\_key\_event\_backlight\_check();

// 重置 屏幕保护和键盘锁的时间

mmi\_idle\_restart\_screensaver\_timer();

mmi\_idle\_restart\_keypad\_lock\_timer();

// 获取 x，y。坐标点

pos.x = (S16) data.x\_adc;

pos.y = (S16) data.y\_adc;

has\_incoming\_event = MMI\_TRUE;

#ifdef \_\_MMI\_TOUCH\_PANEL\_SHORTCUT\_\_

if(!mmi\_pen\_check\_tp\_shortcut(&pos, data.event))

#endif /\* \_\_MMI\_TOUCH\_PANEL\_SHORTCUT\_\_ \*/

{

// 判断 是否是 stroke move 事件结束

// is\_stroke\_move 表示前一个pen event 事件是否是 stroke move

if (is\_stroke\_move && (data.event != STROKE\_MOVE))

{

is\_stroke\_move = MMI\_FALSE;

// 这个函数眼熟吧

if (g\_pen\_stroke\_post\_move)

{

g\_pen\_stroke\_post\_move();

}

}

// 保存现在状态

g\_pen\_cntx.pen\_current\_pos = pos;

g\_pen\_cntx.pen\_event = data.event;

switch (data.event)

{

case PEN\_DOWN:

g\_pen\_cntx.is\_pen\_down = 1;

g\_pen\_cntx.is\_in\_pen\_handler = 1;

// 计算最近两次 点击事件的 事件间隔

// 这里这么判断是当时间溢出 short的时候，需要特殊处理

if (g\_pen\_cntx.previous\_pen\_down\_time > data.time\_stamp)

{

interval = 0XFFFF - g\_pen\_cntx.previous\_pen\_down\_time + data.time\_stamp;

}

else

{

interval = data.time\_stamp - g\_pen\_cntx.previous\_pen\_down\_time;

}

//判断两次pen down 的时间是否足够短，间隔距离足够短，从而判断是否是 双击

if ((interval <= MMI\_PEN\_DOUBLE\_CLICK\_THRESHOLD) &&

(mmi\_pen\_get\_distance\_square(g\_pen\_cntx.pen\_down\_pos, pos) < MMI\_PEN\_SKIP\_DOUBLE\_CLICK\_OFFSET\_SQUARE))

{

if (g\_pen\_event\_table[MMI\_PEN\_EVENT\_DOUBLE\_CLICK] && g\_pen\_cntx.is\_enabled)

{

// 前置处理函数

MMI\_PEN\_EXECUTE\_PRE\_CALLBACK\_FUNC(MMI\_PEN\_EVENT\_DOUBLE\_CLICK);

// 回调double click 注册函数

(g\_pen\_event\_table[MMI\_PEN\_EVENT\_DOUBLE\_CLICK]) (pos);

// 后置处理函数

MMI\_PEN\_EXECUTE\_POST\_CALLBACK\_FUNC(MMI\_PEN\_EVENT\_DOUBLE\_CLICK);

}

g\_pen\_cntx.previous\_pen\_down\_time = 0;

}

else

{

g\_pen\_cntx.previous\_pen\_down\_time = data.time\_stamp;

}

g\_pen\_cntx.pen\_down\_pos = pos;

if (g\_pen\_event\_table[MMI\_PEN\_EVENT\_DOWN] && g\_pen\_cntx.is\_enabled)

{

// 前置处理函数

MMI\_PEN\_EXECUTE\_PRE\_CALLBACK\_FUNC(MMI\_PEN\_EVENT\_DOWN);

// 回调pen down注册函数

(g\_pen\_event\_table[MMI\_PEN\_EVENT\_DOWN]) (pos);

// 后置处理函数

MMI\_PEN\_EXECUTE\_POST\_CALLBACK\_FUNC(MMI\_PEN\_EVENT\_DOWN);

}

break;

case PEN\_UP:

g\_pen\_cntx.is\_pen\_down = 0;

if (g\_pen\_event\_table[MMI\_PEN\_EVENT\_UP] && g\_pen\_cntx.is\_enabled)

{

MMI\_PEN\_EXECUTE\_PRE\_CALLBACK\_FUNC(MMI\_PEN\_EVENT\_UP);

(g\_pen\_event\_table[MMI\_PEN\_EVENT\_UP]) (pos);

MMI\_PEN\_EXECUTE\_POST\_CALLBACK\_FUNC(MMI\_PEN\_EVENT\_UP);

}

// 判断是否要进行pen event reset

if (g\_pen\_cntx.reset\_stroke\_on\_pen\_up)

{

// 判读是否有 stroke event

if (g\_pen\_stroke\_max\_points > 0)

{

// 结束 stroke event，注意，stroke event 会把所有的点push到一个数组里，供注册的程序使用，当然这个数组大小也是程序设定的。这里设置结束标志。

mmi\_pen\_end\_strokes\_of\_character();

mmi\_pen\_reset();

mmi\_pen\_begin\_strokes\_of\_character();

}

else

{

mmi\_pen\_reset();

}

}

// 一对 down move up 后，即时还有pen事件，通过timer 来取下一组信息

delay\_polling\_timer = MMI\_TRUE;

g\_pen\_cntx.is\_in\_pen\_handler = 0;

g\_pen\_cntx.pen\_event = TP\_UNKNOWN\_EVENT;

break;

case PEN\_MOVE:

if (g\_pen\_event\_table[MMI\_PEN\_EVENT\_MOVE] && g\_pen\_cntx.is\_enabled)

{

MMI\_PEN\_EXECUTE\_PRE\_CALLBACK\_FUNC(MMI\_PEN\_EVENT\_MOVE);

(g\_pen\_event\_table[MMI\_PEN\_EVENT\_MOVE]) (pos);

MMI\_PEN\_EXECUTE\_POST\_CALLBACK\_FUNC(MMI\_PEN\_EVENT\_MOVE);

}

break;

case PEN\_LONGTAP:

if (g\_pen\_event\_table[MMI\_PEN\_EVENT\_LONG\_TAP] && g\_pen\_cntx.is\_enabled)

{

MMI\_PEN\_EXECUTE\_PRE\_CALLBACK\_FUNC(MMI\_PEN\_EVENT\_LONG\_TAP);

(g\_pen\_event\_table[MMI\_PEN\_EVENT\_LONG\_TAP]) (pos);

MMI\_PEN\_EXECUTE\_POST\_CALLBACK\_FUNC(MMI\_PEN\_EVENT\_LONG\_TAP);

}

break;

case PEN\_REPEAT:

if (g\_pen\_event\_table[MMI\_PEN\_EVENT\_REPEAT])

{

MMI\_PEN\_EXECUTE\_PRE\_CALLBACK\_FUNC(MMI\_PEN\_EVENT\_REPEAT);

(g\_pen\_event\_table[MMI\_PEN\_EVENT\_REPEAT]) (pos);

MMI\_PEN\_EXECUTE\_POST\_CALLBACK\_FUNC(MMI\_PEN\_EVENT\_REPEAT);

}

break;

case PEN\_ABORT:

pen\_abort\_happen = MMI\_TRUE;

g\_pen\_drvq\_full\_abort = MMI\_TRUE;

mmi\_pen\_reset();

break;

case TP\_UNKNOWN\_EVENT:

MMI\_ASSERT(0);

break;

case STROKE\_MOVE:

// 这个主要是用于当只有一个手写区域时使用

if (g\_pen\_cntx.is\_pending\_stroke\_event)

{

// 判断是第一stroke move点是否符合标准（最小距离）

if (mmi\_pen\_get\_distance\_square(g\_pen\_cntx.pen\_down\_pos, pos) > g\_pen\_stroke\_min\_offset\_square)

{

g\_pen\_cntx.is\_pending\_stroke\_event = 0;

g\_pen\_stroke\_min\_offset\_square = 0;

g\_pen\_cntx.reset\_stroke\_on\_pen\_up = 0;

g\_pen\_cntx.pen\_event = PEN\_ABORT;

// 先产生 pen\_abort 事件，告诉应用普通的pen 事件被中断，也就是光有pen down 没有pen up事件了

if (g\_pen\_event\_table[MMI\_PEN\_EVENT\_ABORT]&& g\_pen\_cntx.is\_enabled)

{

MMI\_PEN\_EXECUTE\_PRE\_CALLBACK\_FUNC(MMI\_PEN\_EVENT\_ABORT);

(g\_pen\_event\_table[MMI\_PEN\_EVENT\_ABORT]) (g\_pen\_cntx.pen\_down\_pos);

MMI\_PEN\_EXECUTE\_POST\_CALLBACK\_FUNC(MMI\_PEN\_EVENT\_ABORT);

}

g\_pen\_cntx.pen\_event = STROKE\_DOWN;

// 把 stroke down 时间事件 保存到数组里

mmi\_pen\_push\_stroke\_point(g\_pen\_cntx.pen\_down\_pos.x, g\_pen\_cntx.pen\_down\_pos.y);

// 发送 stroke down 事件

if (g\_pen\_stroke\_table[MMI\_PEN\_STROKE\_DOWN] && g\_pen\_cntx.is\_enabled)

{

MMI\_PEN\_EXECUTE\_PRE\_CALLBACK\_FUNC(MMI\_PEN\_STROKE\_DOWN);

(g\_pen\_stroke\_table[MMI\_PEN\_STROKE\_DOWN]) (g\_pen\_cntx.pen\_down\_pos);

MMI\_PEN\_EXECUTE\_POST\_CALLBACK\_FUNC(MMI\_PEN\_STROKE\_DOWN);

}

g\_pen\_cntx.pen\_event = STROKE\_MOVE;

// 预处理

is\_stroke\_move = MMI\_TRUE;

if (g\_pen\_stroke\_pre\_move)

{

g\_pen\_stroke\_pre\_move();

}

// 发送 stroke move

mmi\_pen\_push\_stroke\_point(pos.x, pos.y);

if (g\_pen\_stroke\_table[MMI\_PEN\_STROKE\_MOVE] && g\_pen\_cntx.is\_enabled)

{

MMI\_PEN\_EXECUTE\_PRE\_CALLBACK\_FUNC(MMI\_PEN\_STROKE\_MOVE);

(g\_pen\_stroke\_table[MMI\_PEN\_STROKE\_MOVE]) (pos);

MMI\_PEN\_EXECUTE\_POST\_CALLBACK\_FUNC(MMI\_PEN\_STROKE\_MOVE);

}

}

}

else

{

// 判断在哪个区域里面

if (g\_pen\_num\_stroke\_area > 1)

{

mmi\_pen\_fix\_multi\_block\_pen\_position(&pos);

}

// 预处理

if (!is\_stroke\_move)

{

is\_stroke\_move = MMI\_TRUE;

if (g\_pen\_stroke\_pre\_move)

{

g\_pen\_stroke\_pre\_move();

}

}

// 保存 stroke move 并发送

mmi\_pen\_push\_stroke\_point(pos.x, pos.y);

if (g\_pen\_stroke\_table[MMI\_PEN\_STROKE\_MOVE] && g\_pen\_cntx.is\_enabled)

{

MMI\_PEN\_EXECUTE\_PRE\_CALLBACK\_FUNC(MMI\_PEN\_STROKE\_MOVE);

(g\_pen\_stroke\_table[MMI\_PEN\_STROKE\_MOVE]) (pos);

MMI\_PEN\_EXECUTE\_POST\_CALLBACK\_FUNC(MMI\_PEN\_STROKE\_MOVE);

}

}

break;

case STROKE\_DOWN:

g\_pen\_cntx.is\_stroke\_down = 1;

g\_pen\_cntx.is\_pen\_down = 1;

g\_pen\_cntx.pen\_down\_pos = pos;

g\_pen\_cntx.stroke\_down\_block\_index = mmi\_pen\_lookup\_handwriting\_block(pos.x, pos.y);

// 单个区域特殊处理

if (g\_pen\_stroke\_min\_offset\_square > 0)

{

g\_pen\_cntx.is\_pending\_stroke\_event = 1;

g\_pen\_cntx.pen\_event = PEN\_DOWN;

if (g\_pen\_event\_table[MMI\_PEN\_EVENT\_DOWN] && g\_pen\_cntx.is\_enabled)

{

MMI\_PEN\_EXECUTE\_PRE\_CALLBACK\_FUNC(MMI\_PEN\_EVENT\_DOWN);

(g\_pen\_event\_table[MMI\_PEN\_EVENT\_DOWN]) (g\_pen\_cntx.pen\_down\_pos);

MMI\_PEN\_EXECUTE\_POST\_CALLBACK\_FUNC(MMI\_PEN\_EVENT\_DOWN);

}

}

else

{

g\_pen\_cntx.is\_pending\_stroke\_event = 0;

g\_pen\_cntx.pen\_event = STROKE\_DOWN;

if (g\_pen\_stroke\_table[MMI\_PEN\_STROKE\_DOWN] && g\_pen\_cntx.is\_enabled)

{

MMI\_PEN\_EXECUTE\_PRE\_CALLBACK\_FUNC(MMI\_PEN\_STROKE\_DOWN);

(g\_pen\_stroke\_table[MMI\_PEN\_STROKE\_DOWN]) (pos);

MMI\_PEN\_EXECUTE\_POST\_CALLBACK\_FUNC(MMI\_PEN\_STROKE\_DOWN);

}

if (g\_pen\_cntx.is\_enabled && g\_pen\_cntx.is\_pen\_down && g\_pen\_stroke\_max\_points > 0)

{

mmi\_pen\_push\_stroke\_point(pos.x, pos.y);

}

}

break;

case STROKE\_UP:

MMI\_DBG\_ASSERT(g\_pen\_stroke\_max\_points > 0);

g\_pen\_cntx.is\_stroke\_down = 0;

g\_pen\_cntx.is\_pen\_down = 0;

if (g\_pen\_cntx.is\_pending\_stroke\_event)

{

g\_pen\_cntx.is\_pending\_stroke\_event = 0;

g\_pen\_cntx.reset\_stroke\_on\_pen\_up = 1;

g\_pen\_cntx.pen\_event = PEN\_UP;

if (g\_pen\_event\_table[MMI\_PEN\_EVENT\_UP] && g\_pen\_cntx.is\_enabled)

{

MMI\_PEN\_EXECUTE\_PRE\_CALLBACK\_FUNC(MMI\_PEN\_EVENT\_UP);

(g\_pen\_event\_table[MMI\_PEN\_EVENT\_UP]) (g\_pen\_cntx.pen\_down\_pos);

MMI\_PEN\_EXECUTE\_POST\_CALLBACK\_FUNC(MMI\_PEN\_EVENT\_UP);

}

delay\_polling\_timer = MMI\_TRUE;

}

else

{

g\_pen\_cntx.pen\_event = STROKE\_UP;

if (g\_pen\_num\_stroke\_area > 1)

{

mmi\_pen\_fix\_multi\_block\_pen\_position(&pos);

}

mmi\_pen\_push\_stroke\_end();

if (g\_pen\_stroke\_table[MMI\_PEN\_STROKE\_UP] && g\_pen\_cntx.is\_enabled)

{

MMI\_PEN\_EXECUTE\_PRE\_CALLBACK\_FUNC(MMI\_PEN\_STROKE\_UP);

(g\_pen\_stroke\_table[MMI\_PEN\_STROKE\_UP]) (pos);

MMI\_PEN\_EXECUTE\_POST\_CALLBACK\_FUNC(MMI\_PEN\_STROKE\_UP);

}

}

if (g\_pen\_cntx.reset\_stroke\_on\_pen\_up)

{

if (g\_pen\_stroke\_max\_points > 0)

{

mmi\_pen\_end\_strokes\_of\_character();

mmi\_pen\_reset();

mmi\_pen\_begin\_strokes\_of\_character();

}

else

{

mmi\_pen\_reset();

}

}

/\* leave pen handler procedure \*/

g\_pen\_cntx.is\_in\_pen\_handler = 0;

g\_pen\_cntx.pen\_event = TP\_UNKNOWN\_EVENT;

break;

default:

MMI\_ASSERT(0);

}

} /\* if(!mmi\_pen\_check\_tp\_shortcut(&pos, data.event)) \*/

// 把外部消息放入到内部循环消息队列

mmi\_frm\_fetch\_msg\_from\_extQ\_to\_circularQ();

OslCircularDump();

// 如果内部消息积压大于 MMI\_PENDING\_MSG\_THRESHOLD，就跳出循环，进行处理

if (OslNumOfCircularQMsgs() > MMI\_PENDING\_MSG\_THRESHOLD)

{

break;

}

}

if (is\_stroke\_move)

{

if (g\_pen\_stroke\_post\_move)

{

g\_pen\_stroke\_post\_move();

}

}

if (has\_incoming\_event)

{

if (!mmi\_shutdown\_is\_in\_power\_off\_period())

{

mmi\_idle\_key\_event\_backlight\_check();

}

}

if (g\_pen\_cntx.is\_in\_pen\_handler == 0)

{

g\_pen\_cntx.pen\_event = TP\_UNKNOWN\_EVENT;

}

// 启动定时期

if (delay\_polling\_timer)

{

StartTimer(PEN\_POLLING\_TIMER, MMI\_PEN\_DEBOUNCE\_POLLING\_DELAY, mmi\_pen\_poll\_hdlr);

g\_pen\_polling\_timer\_started = MMI\_TRUE;

SetBit(g\_input\_msg\_in\_queue, MMI\_DEVICE\_PEN);

}

else if (g\_pen\_cntx.is\_pen\_down || pen\_abort\_happen)

{

StartTimer(PEN\_POLLING\_TIMER, MMI\_PEN\_POLLING\_PERIOD, mmi\_pen\_poll\_hdlr);

g\_pen\_polling\_timer\_started = MMI\_TRUE;

SetBit(g\_input\_msg\_in\_queue, MMI\_DEVICE\_PEN);

if(pen\_abort\_happen)

{

pen\_abort\_happen = MMI\_FALSE;

}

}

}

这里其实也是比较简单：

1： 取pen event 事件

2： 根据时间类型处理事件

3：是否有额外消息要处理，是 退出，否继续 1

这里有点点复杂的是手写输入法，这个事件比较乱，全局变量一堆，分析起来比较头疼。

到这里 pen event 其实也就差不多了，说一下几个常用的函数：

mmi\_pen\_enable /mmi\_pen\_disable  //打开/关闭 pen 事件

mmi\_pen\_set\_calibration\_data / mmi\_pen\_read\_calibration\_data  //设置和读取校准数据，校准数据总共四个值 x = x\*x\_scale + x\_offset, y = y\*y\_scale + y\_offset

mmi\_pen\_start\_calibration  // 开始校准

mmi\_pen\_start\_capture\_strokes / mmi\_pen\_stop\_capture\_strokes  //在设定区域里 开始和关闭  捕捉输入法轨迹(stroke)点。具体可以参见 EntryEmPenTest 里面用法，有机会在分析MTK手写输入法

mmi\_pen\_config\_move\_offset / mmi\_pen\_config\_timeout\_period / mmi\_pen\_config\_sampling\_period  //几个配置函数，对于写一个好的触摸屏应用，还是很有必要的，相关介绍看 [MTK MMI event 小结 7](http://blog.csdn.net/yanwuxufeng/archive/2010/08/09/5800121.aspx)

mmi\_pen\_register\_move(up down repeat …)\_handler  //注册pen 事件处理函数。

对于触摸屏校准，做一个简单的说明：

当 调用 void mmi\_pen\_start\_calibration(kal\_uint16 num, const mmi\_pen\_point\_struct \*points) 函数时，就开始进入触摸屏校准状态，一旦调用这个函数，那么收到点将都是没有校准过的，直到校准完成。num 表示校准的个数，MTK是3个，points 分别就是理想点的位置(可能不好理解，就是屏幕上显示需要点击的3个点)

真正执行校准的是在 函数 touch\_excute\_cali里面

void touch\_excute\_cali(kal\_int16 x\_adc, kal\_int16 y\_adc)

{

ilm\_struct \*tp\_ilm;

tp\_cali\_done\_struct \*local\_para;

kal\_bool cali\_result;

module\_type owner=0;

// 校准是否被激活，调用 mmi\_pen\_start\_calibration 后，就为true

if(tp\_cali\_mode==KAL\_FALSE)

return;

// 校准第一个点

if(tp\_cali\_cnt==0)

{

tp\_cali\_cnt++;

// 记录获取的adc 点(实际物理点坐标)

cali\_point\_adc[0].x=x\_adc;

cali\_point\_adc[0].y=y\_adc;

}

// 第二个点

else if(tp\_cali\_cnt==1)

{

tp\_cali\_cnt++;

cali\_point\_adc[1].x=x\_adc;

cali\_point\_adc[1].y=y\_adc;

}

// 第三个点

else if(tp\_cali\_cnt==2)

{

cali\_point\_adc[2].x=x\_adc;

cali\_point\_adc[2].y=y\_adc;

tp\_cali\_mode=KAL\_FALSE;

tp\_cali\_cnt=0;

// 对校准进行计算

cali\_result=touch\_panel\_check\_cali();

if(cali\_result==KAL\_TRUE)

{

// 校准ok ,分别计算 x 和 y 的 slope 和 offset

touch\_panel\_tuning(cali\_point[0].x, cali\_point\_adc[0].x,

cali\_point[1].x, cali\_point\_adc[1].x,

&TPCali.x\_slope, &TPCali.x\_offset);

touch\_panel\_tuning(cali\_point[0].y, cali\_point\_adc[0].y,

cali\_point[1].y, cali\_point\_adc[1].y,

&TPCali.y\_slope, &TPCali.y\_offset);

}

// 发送消息告诉MMI , 校准的结果.

// 如果失败,MMI 需要重新校准

local\_para = (tp\_cali\_done\_struct \*)

construct\_local\_para(sizeof(tp\_cali\_done\_struct),TD\_UL);

local\_para->result=cali\_result;

owner=MOD\_MMI;

DRV\_BuildPrimitive(tp\_ilm,

MOD\_TP\_TASK,

owner,

MSG\_ID\_TP\_CALI\_DONE,

local\_para);

msg\_send\_ext\_queue(tp\_ilm);

}

}

其实校准是否正确 还是主要看 touch\_panel\_check\_cali 这个函数的结果。这个函数主要里有  3 个校准测试，只有都符合条件才符合。

stage 1 比例对比，也就是保证测试点1 和 点 2  倾斜一定的角度。不会与x 轴或者y轴平行

stage 2 保证第1 点 和 第 2 点  在 第3 点的 两端

stage 3 根据stage 1 和 stage 2 得出的slope 和 offset， 比较计算获得的第3点 和 希望获得第3点 误差小于 设定值。这几个函数就是几个数学公式，自己研究研究 就Ok 了。