**IPA Pool Notify特性设计规格文档**

**Version 1.1**

**Prepared by <Samson.Huang>**

**<2015-10-23>**

© 2015 MStar Semiconductor, Inc. All rights reserved.

MStar Semiconductor makes no representations or warranties including, for example but not limited to, warranties of merchantability, fitness for a particular purpose, infringement of any intellectual property right or the accuracy or completeness of this document, and reserves the right to make changes without further notice to any products herein to improve reliability, function or design. No responsibility is assumed by MStar Semiconductor arising out of the application or user of any product or circuit described herein; neither does it convey any license under its patent rights, nor the rights of others.

MStar is a trademark of MStar Semiconductor, Inc. Other trademarks or names herein are only for identification purposes only and owned by their respective owners.

REVISION HISTORY

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Rev.** | **Description** | **Date** | **Author** | **Approval** |
| *1.0* | *SRS/HLD档合二为一的初始HLD* | *2015/09/16* | *samson.huang* |  |
| *1.1* | *正式版* | *2015/10/23* | *samson.huang* |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

TABLE OF CONTENT

[REVISION HISTORY i](#_Toc433355597)

[TABLE OF CONTENT 1](#_Toc433355598)

[LIST OF TABLES 2](#_Toc433355599)

[LIST OF FIGURES 2](#_Toc433355600)

[1. 介绍 4](#_Toc433355601)

[1.1. 目的 4](#_Toc433355602)

[1.2. 简介 4](#_Toc433355603)

[1.3. 参考文档 4](#_Toc433355604)

[2. 整体描述 5](#_Toc433355605)

[2.1. 系统描述 5](#_Toc433355606)

[2.2. 架构描述 6](#_Toc433355607)

[2.3. 特性列表 11](#_Toc433355608)

[2.5. 假设和依赖 12](#_Toc433355609)

[2.6. 内部限制 12](#_Toc433355610)

[3. 功能需求 13](#_Toc433355611)

[3.1. 类别1 13](#_Toc433355612)

[3.1.1. 类别1的关键设计说明 13](#_Toc433355613)

[3.1.2. Use Case: clientA注册回调 13](#_Toc433355614)

[3.2. 类别2 13](#_Toc433355615)

[3.2.1. 类别2的关键设计说明 13](#_Toc433355616)

[3.2.2. Use Case: client B timeout alloc 未和client A冲突 14](#_Toc433355617)

[3.2.3. Use Case: client B timeout alloc 和client A冲突 14](#_Toc433355618)

[3.2.4. Use Case: client A的IPAPool\_Release 15](#_Toc433355619)

[3.3. 移植功能的use case 16](#_Toc433355620)

[4. 非功能性需求设计 17](#_Toc433355621)

[5. 功能交互性分析 18](#_Toc433355622)

[6. 关键性能分析（可选） 19](#_Toc433355623)

[7. 需求分解 20](#_Toc433355624)

[7.1.1. utopia IPA Pool Notify需求 20](#_Toc433355625)

[7.1.2. Kernel IPA Pool Notify需求 20](#_Toc433355626)

[8. 接口描述 21](#_Toc433355627)

[8.1.1. 接口一 21](#_Toc433355628)

[8.1.2. 接口二 21](#_Toc433355629)

LIST OF TABLES

[表格 1缩略语清单Table 3](#_Toc433355592)

[表格 2 Feature list 11](#_Toc433355593)

LIST OF FIGURES

[图表 1 IPA Pool Notify整体思路 5](#_Toc433355585)

[图表 2 MApi\_IPA\_Pool\_GetMem\_Timeout流程 6](#_Toc433355586)

[图表 3 MApi\_IPA\_Pool\_Register\_Notify流程 7](#_Toc433355587)

[图表 4 polling thread流程 8](#_Toc433355588)

[图表 5 MApi\_IPA\_Pool\_Release里新增改动 9](#_Toc433355589)

[图表 6 waitqueue的sleep和wakeup 10](#_Toc433355590)

[图表 7 冲突检测机制示意图 11](#_Toc433355591)

表格 1缩略语清单Table

|  |  |
| --- | --- |
| **缩略语** | **描述** |
| TC | Technical Center大陆技术中心部门，大陆RD |
| SRS | Software Requirement Specification 软件需求规格书 |
| HLD | High Level Design 概要设计 |
| LLD | Low Level Design 详细设计 |
| SIT | SIT System Integration Test系统整合测试 |
| MT | Module Test 模块测试 |
| UT | Unit Test 单元测试 |
| PV | Product Validation 产品验证 |
| PMB | Project Management Board 项目管理委员会 |
| SE | System Engineer 系统工程师 |
| RR | Requirement Request 需求请求 |
| RMS | Requirement Management System 需求管理系统 |
| RME | Requirement Management Engineer 需求管理工程师 |
| PM | Project Manager 项目经理，在RD一般指G type项目的项目经理 |
| TL | Techincal Leader 技术项目经理，具体一个特性或solution的开发负责人，负责该特性或solution开发涉及若干人人的开发过程管理和技术管理，为该特性或solution的开发质量负责。 |
|  |  |

# 介绍

## 目的

本文描述的是IPA Pool Notify功能的实现设计，为不同的client使用IPA Pool产生的冲突提供通知机制，至于冲突的memory range 能否协调成功IPA Pool Notify并不负责，协调的实现由各个client自主决定。通过冲突通知机制和协调实现，降低get memory fail的机率。

## 简介

*IPA Pool notify功能是实现在IPA Pool应用中，不同的client在get memory冲突时，冲突的一种通知(notify)机制。*

*在 client B alloc memory，由于client A/client C/client D/…等已经占用了其要alloc的memory中的一部分还未free，导致client B alloc memory fail时，需要通知到A/C/D其哪些memory和B冲突了，是否采取解决冲突的措施以及如何采取由A/C/D/…自己分别各自决定，B只是通知到位。*

## 参考文档

*【1】samson.huang，“***关于IPA相关API使用***”，v4.3，2015-7-31*

# 整体描述

## 系统描述

IPA pool Notify整体思路如图1示意，在使用IPA Pool时，在utopia 层的不同process的alloc memory可能产生冲突，借助kernel 的waitqueue和event处理，把冲突信息返回给utopia，并由process自主协调冲突。



图表 1 IPA Pool Notify整体思路

## 架构描述

（1）如图2示意，utopia code里面新增一个接口MApi\_IPA\_Pool\_GetMem\_Timeout，这个接口是带timeout的get memory接口。该接口和原有的MApi\_IPA\_Pool\_GetMem接口很类似，但有一点差别，差别就是在ioctl之前把timeout的值赋值给ioctl的入口参数，这样以便kernel知道alloc fail需要delay之后再次alloc的话，要delay多长时间。MApi\_IPA\_Pool\_GetMem\_Timeout具体实现的主体是设置timeout的值给kernel和通过ioctl去alloc memory。发生冲突后的delay timeout 和try alloc again都是放在utopia code做，而不是放在kernel code里面做（注：后来实测在kernel code里面做，不放utopia code做麻烦些，有些问题要解决，所以最终是采用了放在utopia code做）。delay timeout 和try alloc again（这时try alloc again时不把timeout信息给kernel了）实做时可以拆分timeout值为实测的经验值，分成多次的delay和try alloc again，直到alloc success了或者总的timeout时间到了。



图表 2 MApi\_IPA\_Pool\_GetMem\_Timeout流程

在增加了MApi\_IPA\_Pool\_GetMem\_Timeout后，原有的MApi\_IPA\_Pool\_GetMem本身也要做适当修改：需要MApi\_IPA\_Pool\_GetMem里的ioctl之前设置ipa\_alloc\_args.timeout = 0;这里的0表示的是不会考虑timeout和after fail alloc again这些事情。

（2）如图3示意，utopia code里新增第二个接口MApi\_IPA\_Pool\_Register\_Notify，在这个接口里去做create polling thread的事情(借助pthread\_create函数)。然后返回。



图表 3 MApi\_IPA\_Pool\_Register\_Notify流程

关于MApi\_IPA\_Pool\_Register\_Notify函数的参数（也即polling thread里的入口参数）：pool handle id和call back组成一个pair，在一个process中可能会有好几个pair。这个可以用带mutex lock的global array存储，也可以用局部变量（这样就不用存储pool handle id）。

注意从流程里可以看出来是一个pool handle id（即一个client）唯一对应一个polling thread的。如果要调用MApi\_IPA\_Pool\_Register\_Notify则一定要在init之后调用。否则MApi\_IPA\_Pool\_Register\_Notify之前可能的notify，就没办法处理。（有notify需要的有callback函数的client的utopia的调用者自己做，如果client没有这个需求，utopia的调用者就不用做，做不做都是utopia的调用者自己决定的。）。如果utopia调用者会调用MApi\_IPA\_Pool\_Register\_Notify，就不要多个模块share同一个pool handle id，否则可能会出错。每个模块自己init pool，获取自己的pool id，这样做也减少了模块之间的耦合性。然后再针对各个poolhandle id做各自的MApi\_IPA\_Pool\_Register\_Notify，然后得到对应的polling thread的polling id 。一个pool handle id跟唯一一个polling id绑定（polling id会记录下来，在需要DeleteTask的时候会用到他）。在polling thread里的入口参数是含有callback function的信息的，该callback function的内部的具体实现只是对这个pool handle id会做处理（很容易理解，因为其拿到的event就是该pool handle id自己的event），别的pool handle id的event对应的所需的callback操作他是不会管的，他也管不了，这样就做到各自管各自的事情。

(3)polling thread函数对应的参数里只有指针表征的pool handle id,至于对应的会用到的其专属的回调函数在MApi\_IPA\_Pool\_Register\_Notify函数里提前存储在IPAPool\_Info全局变量的信息里。一个client对应的是一个polling thread。具体流程如图4示意：



图表 4 polling thread流程

(a).在polling thread里会做一个while(1)操作，while里面的内容是先ioctl去获得event。如果没有event，那么ioctl就睡眠不返回；如果event是退出的event，那么就break退出while循环；如果event是conflict的event，那么就调用对应的callback做对应的工作，callback函数的具体实现由utopia的调用者自己实现，IPA Pool notify不负责。callback做完之后进入到while是下一轮循环，继续进行ioctl…

(b). while循环退出后,return之前 ,client自身信息的结构体里置个DeleteTask标志（client init的时候该DeleteTask标志默认设置成未设的清除状态）。

然后在MApi\_IPA\_Pool\_Release函数(utopia code)里需要增加code检查是否有polling thread（即是否有绑定polling id），如果没有那么MApi\_IPA\_Pool\_Release里就不用做额外的操作。如果有绑定polling id，那么需要判断DeleteTask flag是否有置,一直等到标志被置了，才在MApi\_IPA\_Pool\_Release里释放polling thread（即调用pthread\_join，前文有create task过程就必须要有对应的delete task过程。）。这段过程如图5示意。注意这段过程相对于原有MApi\_IPA\_Pool\_Release函数过程而言，是要放在IPA\_POOL\_IOC\_DEINIT的iotcl之后做的。



图表 MApi\_IPA\_Pool\_Release里新增改动

(c) 然后是polling thread函数return。

（4）polling thread里while(1)里面的ioctl的具体实现（现在转入了kernel code）：

如图6的part1过程的示意，设置client的一个flag。该flag是“表征client已注册polling thread的flag”。在这个client的这个ioctl第一次进入到kernel code后，去设置client对应的这个flag。而client在初始init的时候，默认的都是会提前先clear掉这个flag。如果没有注册过polling thread，那么整个IPA Pool Notify相关的工作都不用做，也无从做起。一定要牢记，该flag是否设置过的判断是后续很多sleep/send event/wakeup相关Notify改动的最基本的先决条件。

然后检查消息计数，如果计数为0就睡眠（调用interruptible\_sleep\_on），等在别处被唤醒wakeup后就返回；非0的话拿消息数据的第一个，并wakeup后返回。



图表 6 waitqueue的sleep和wakeup

上面提到了消息计数，之所以要用到消息计数，是因为与client A冲突的可能有多个比如client B和client C,比如client B的event还未开始处理或者已处理但还未完成，这时client C的event 已经来了。那么这2个event都是要处理的，把他们利用消息数组和消息计数去存储到client A的waitqueue里去。

（5）上文提到了在何时会睡眠，对应的唤醒的时机的code有2处（在kernel code里）：

(a) IPA\_pool\_get\_mem函数里，这个函数是为client分配memory。这个过程如图6的part2过程的示意。会去pool\_allocation\_tree里寻找是否和别的client的已alloc的冲突。原有的IPA\_pool\_get\_mem函数code逻辑是如果client B在get\_mem时，发现了和client A冲突，那么立即返回-EBUSY宣告分配失败。现在需要做的是寻找所有和client B冲突的含有polling thread的clients（比如client A/client C/ client D/…）以及其各自的范围。各自的范围是指与B有交叉的各个子范围的串连。如图7示意，如A取A1的开始，取A2的结束；D取D1的开始，取D2的结束。



图表 7 冲突检测机制示意图

寻找所有和client B冲突的clients（比如client A/client C/ client D/…）以及其各自的范围时，先要判断哪些client有置“表征client已注册polling thread的flag”，没置的clients的冲突范围就不会记录下来，就不会做Notify的相关操作了，有置的clients才会进一步的操作。进一步的操作是对这些clients都发IPA\_EVENT\_CONFLICT信息到各自的client的waitqueue，即目的是要通知这些clients需要请其释放刚找到的其各自的范围。

最后是唤醒冲突的各个clients的各自的polling thread。

(b) IPA\_pool\_deinit函数(kernel code)里（注意实测之后的建议是放在IPA\_pool\_deinit函数，而不是IPAPool\_Release函数），这个时候整个pool都要被deinit了。首先，依然是要先判断是否有置“表征client已注册polling thread的flag”，没有的话，就不用notify相关的操作。只要有置flag，就需要将先前的event（如果有的话）全清掉，放入no wait event，然后唤醒（调用wake\_up函数）。上面这些过程放在原有的IPA\_pool\_deinit函数的最开始的过程。这个过程如图6的part3过程的示意。

## 特性列表

*如下表2示意*

表格 Feature list

|  |  |
| --- | --- |
| ID | Feature |
| 1 | Notify   * MApi\_IPA\_Pool\_Register\_Notify：polling thread的工作和callback的实施 |
| 2 | Get memory   * MApi\_IPA\_Pool\_GetMem\_Timeout：带timeout的get memory |

## 假设和依赖

*（1）同一个client A只会处理自己的event，不会去处理别的client B的event。尽管client A的event可能是client B 在做相关操作时引起触发的。*

## 内部限制

*(1)* 应用场景只是需要IPA提供一个通知机制，即不同process可以知道相互间是否有冲突，至于冲突的memory range能否协调成功IPA Pool Notify并不负责。

*（2）*B在和A发生confict然后 wakeup A的polling thead后只是单纯的schedule\_timeout，A的callback处理完以后不去通知B。简单的B schedule  out之后再次尝试B alloc容易处理些，如果再增加A通知B自己已经处理好的逻辑的话，需要再评估下复杂性。建议暂时先不考虑。因为冲突的case毕竟是偶尔出现，不影响大局。

# 功能需求

## 类别1

### 类别1的关键设计说明

类别1主要是pooling thread的创建，以及回调函数相关。

### Use Case: clientA注册回调

|  |  |
| --- | --- |
| **UCID** | 1 |
| **Use case名字** | 注册回调 |
| **简介** | 通过调用utopia接口注册对应的回调函数 |
| **前置条件** | 使用IPA Pool;client对应的callback有具体实现 |
| **预期结果** | (Success) 注册callback过程成功  (Exception) 注册callback过程失败 |
| **触发条件** | pool init完成之后，适当的时机调用register notify的utopia接口 |
| **基本场景** | 1.register notify里主体是create polling thread。  2. polling thread里会*ioctl的方式去设置*“表征client已注册polling thread的flag”。  3. *然后，ioctl的方式去sleep，等待client memory conflict或者no wait的event发生，抛出event时，需要将冲突范围转换到冲突pool内的offset和length，发生并抛出event后wakeup。*  *4.wakeup后 ,如果是 conflict event的话，执行callback的主体；no wait event的话，则无需执行callback。*  *5.然后是根据event类型的循环或者返回。* |
| **可选场景** | 无 |
| **异常场景** | 1. *ioctl fail* 2. *callback fail* |
| **备注** | 参考“图4 polling thread流程” |

## 类别2

### 类别2的关键设计说明

*类别2主要是event的发送和*pooling thread的wakeup以及pooling thread的后续操作相关。

### Use Case: client B timeout alloc 未和client A冲突

|  |  |
| --- | --- |
| **UCID** | 2 |
| **Use case名字** | client B timeout alloc 未和client A冲突 |
| **简介** | 在alloc memory的clientB所需的memory ,clientA目前没有在占用,两者没有发生冲突。 |
| **前置条件** | 1.client A有init。  2.client A有注册callback。  3.client B有init。 |
| **预期结果** | 成功： client B alloc 成功。  失败： client B alloc 失败(非和client A冲突而是别的原因导致的alloc fail)。 |
| **触发条件** | client B timeout alloc memory |
| **基本场景** | 1.client B timeout alloc 调用常规的alloc。  2.检查是否有和别的client冲突,如果有冲突，且有置“表征client已注册polling thread的flag”且timeout的值大于0，那么会做相应的处理，否则就不会涉及任何冲突相关的处理操作。如果有冲突的话会做相应的处理,后面的**Use case**里会展开讲。这里我们当前的这个use case是client B没有和clientA冲突。  3.如果没有任何冲突的话，或者有冲突但是没有置“表征client已注册polling thread的flag”的话,按原本原有的“常规的alloc”的code继续下去，最后的结果可能是alloc成功，也可以是别的一些原因alloc失败。 |
| **可选场景** | N/A |
| **异常场景** | N/A |
| **备注** | 参考“图6 waitqueue的sleep和wakeup” |

### Use Case: client B timeout alloc 和client A冲突

|  |  |
| --- | --- |
| **UCID** | 3 |
| **Use case名字** | client B timeout alloc 和client A冲突 |
| **简介** | 在alloc memory的clientB所需的memory ,clientA有在占用（占用其中一块或者多块）,两者有冲突。 |
| **前置条件** | 1.client A有init。  2.client A有注册callback。  3.client B有init。 |
| **预期结果** | 先失败： client B alloc 失败(和client A冲突或者和别的client冲突导致的fail)。  再成功或者再失败：本use case下会尝试2次alloc（在utopia code里实际的调用2轮alloc,其中第二轮是针对第一轮的fail后的再次尝试），第一次是必然失败的，第二个则成功或者失败都有可能。 |
| **触发条件** | client B timeout alloc memory |
| **基本场景** | 1.client B timeout alloc 调用常规的alloc。  2.检查是否有和别的client冲突,如果有的话会做相应的处理，我们当前的这个use case是有和clientA冲突的，需要先判断是否有置“表征client已注册polling thread的flag”且timeout的值大于0，是的话才会处理，不是的话就会转向走原有的常规逻辑。  3. 需要处理的话，client A会发memory conflict的event到utopia的调用者，并把client A的sleep的polling thread给wakeup,由utopia的调用者去做clientA的callback。  4.上面提到了，client A会发memory conflict的event到utopia的调用者，要free的start和length也会同时计算出来发给utopia的调用者。  5.如果类似clientA,还有别的clientC/clientD/…等等都和clientB冲突。那么别的clients都需要做和clientA类似的操作。  6.接着在utopia code角度，会delay timeout的值对应的一段指定的时间。  7. client B 再次尝试分配所需的那块memory，即在utopia code角度try alloc again。  8. 再次尝试分配之后，不管是成功还是失败，都结束。  9.实做时也可把上面的6、7、8三步拆分成多次，直到总的timeout时间到了或者某次alloc success了。 |
| **可选场景** | N/A |
| **异常场景** | N/A |
| **备注** | 参考“图6 waitqueue的sleep和wakeup” |

### Use Case: client A的IPAPool\_Release

|  |  |
| --- | --- |
| **UCID** | 4 |
| **Use case名字** | client A的IPA\_pool\_deinit |
| **简介** | client A的IPA\_pool\_deinit，需要发NO WAIT event 和唤醒 polling thread和结束polling thread。 |
| **前置条件** | 1.client A有init。  2.client A有注册callback。 |
| **预期结果** | 成功：成功的发event和实现polling thread相关操作；以及原本有的成功的实现pool deinit |
| **触发条件** | clientA IPA\_pool\_deinit |
| **基本场景** | 1. 依然是首先要判断是否有置“表征client已注册polling thread的flag”，没有置的话，后续步骤什么都不用做的。  2. 然后，如果waitqueue为空，则不需做什么event和wakeup，直接返回。  3. 如果waitqueue不为空，则清空waitqueue，并设置NO WAIT event。  4. 唤醒 polling thread。  5. 然后从kernel返回到utopia的调用者，在utopia的调用者里结束polling thread：需要kernel返回后， utopia的调用者自身的返回之前先设置需要delete polling task的flag（该flag会在utopia的调用者的MApi\_IPA\_Pool\_Release里判断后去做delete polling task动作）。 |
| **可选场景** | N/A |
| **异常场景** | N/A |
| **备注** | 参考“图6 waitqueue的sleep和wakeup” |

## 移植功能的use case

*无内容*

# 非功能性需求设计

*此章无内容。*

# 功能交互性分析

IPA Pool notify只是在IPA Pool里新增了notify功能，不会影响到其他模块。IPA Pool notify的增加会同时改到IPA Pool里别的函数（比如MApi\_IPA\_Pool\_GetMem/ MApi\_IPA\_Pool\_Release等），要求相关的函数做相应的一些必要的很基本的修改，对IPA Pool模块本身的其他功能影响极小，但为了保险起见，还是有必要IPA Pool的其他功能也注意测试下。

# 关键性能分析（可选）

无内容

# 需求分解

### utopia IPA Pool Notify需求

*IPA Pool Notify需求中，utopia需要完成新增2个接口，一个是带timeout的get memory；另一个是建个polling thread，在里面循环的去等待event happen和针对event做callback或者不做callback而结束polling thread。**参看2.2节的”*架构描述”一节了解更多。

### Kernel IPA Pool Notify需求

*IPA Pool Notify需求中，kernel需要在适当的场合让polling thread睡眠，然后在一些适当的场合发event和唤醒polling thread。参看2.2节的”*架构描述”一节了解更多。

# 接口描述

### 接口一

MS\_BOOL \_\_attribute\_\_((weak)) MApi\_IPA\_Pool\_GetMem\_Timeout(struct IPA\_Pool\_GetMem\_Param\* get\_param,MS\_U32 timeout)

功能说明：这个接口描述的是带timeout 的alloc memory接口。

入参说明：

struct IPA\_Pool\_GetMem\_Param\* get\_param，含义同原有的MApi\_IPA\_Pool\_GetMem函数的入口参数。

MS\_U32 timeout：timeout的数值，单位是毫秒。

出参说明：MS\_BOOL型，false表示失败，true表示成功。

### 接口二

typedef void (\*callback)(MS\_U32 pool\_handle\_id,MS\_U64 start,MS\_U64 length);

功能说明：实现针对event消息的回调，具体实现是utopia的调用者的事。

入参说明：

pool\_handle\_id为client对应的pool handle id,

start为event消息传递给的冲突的起始位置，

length为event消息传递给的冲突的长度。

出参说明：无出参

struct IPA\_Pool\_Polling\_Param

{

MS\_U32 pool\_handle\_id;//pool handle

void (\*polling\_callback)(MS\_U32 pool\_handle\_id,MS\_U64 start,MS\_U64 length);

};

MS\_BOOL \_\_attribute\_\_((weak)) MApi\_IPA\_Pool\_Register\_Notify(struct IPA\_Pool\_Polling\_Param \*polling\_param)功能说明：polling thread的工作和callback的实施

入参说明：polling\_callback为该client对应的回调函数，pool\_handle\_id为client对应的pool handle id

出参说明：MS\_BOOL型，false表示失败，true表示成功。