MI SYS API

Version 2.16

REVISION HISTORY

Revision No.	Description	Date
2.03	• Initial release	11/12/2018
2.04	• Added MI_SYS_GetFd note description • Added MI_SYS_CloseFd note description	02/21/2019
2.05	• Added API MI_SYS_ConfigDevChnPrivateMMAHeap	03/18/2019
2.06	• Added DMA API	04/02/2019
2.07	• Renamed MI_SYS_ConfigDevChnPrivateMMAHeap to MI_SYS_ConfigPrivateMMAPool and refined api	04/09/2019
2.08	• Removed E_MI_SYS_BUFDATA_META Renamed MI_SYS_MetaData_t to MI_SYS_PerFrameMetaBuf_t Added MI_SYS_PerFrameMetaBuf_t into MI_SYS_BufInfo_t	04/10/2019
2.09	• Added E_MI_SYS_PER_CHN_PORT_OUTPUT_POOL and MI_SYS_PerChnPortOutputPool_t for ouput port private pool	04/19/2019
2.10	• Added MI_SYS_FrameIspInfoType_e and MI_SYS_FrameIspInfo_t • Removed MI_SYS_MetaDataConfig_t	04/29/2019
2.11	• Added MI_SYS_WindowRect_t into MI_SYS_FrameData_t	06/11/2019
2.12	• Update frame rate control information.	06/11/2019
2.13	• Update Error Code	06/28/2019
2.14	• Added chroma alignment into MI_SYS_FrameBufExtraConfig_t	08/28/2019
2.15	• Added meta data:E_MI_SYS_BUFDATA_META and MI_SYS_MetaDataConfig_t • Renamed MI_SYS_PerFrameMetaBuf_t to MI_SYS_MetaData_t	11/29/2019
2.16	• Added API: MI_SYS_EnableChnOutputPortLowLatency	04/16/2020

TABLE OF CONTENTS

1. 概述

1.1. 模块说明

MI_SYS 是整个 MI 系统的基础模块，它给其他 MI 模块的运行提供了基础。

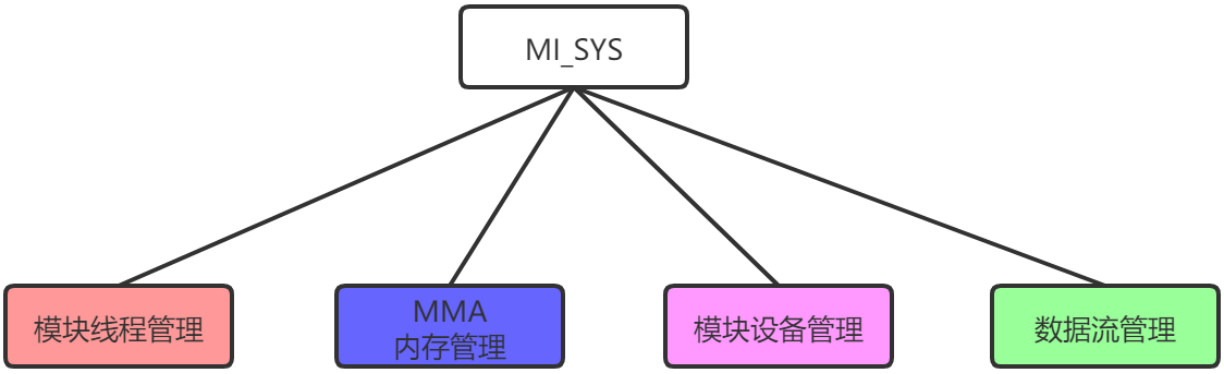

图 1-1 MI_SYS 系统框架图

如图 1-1 所示，MI_SYS 主要功能概述如下：

- 实现 MI 系统初始化、MMA 内存缓冲池管理。
- 提供各模块注册设备节点，proc 系统的建立的通用接口。
- 提供各模块建立绑定关系的接口，管理各模块之间数据流动。
- 提供各模块申请 MMA 连续物理内存的接口，管理内存分配，映射虚拟地址，内存回收。
- 提供各模块建立工作线程的接口，管理各模块线程的创建，运行，销毁。

如图 1-2 所示，MI_SYS 的代码结构主要分为四层：impl 层、internal 层、ioctl 层、api 层。

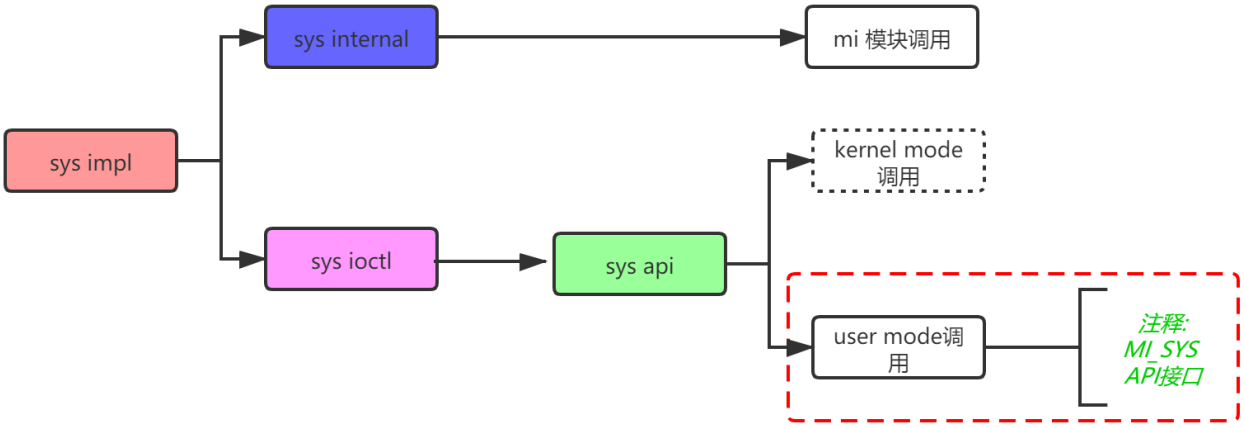

图 1-2 MI_SYS 代码结构图

Sys impl 层：MI_SYS 的最底层，基本上 sys 的功能都在这里实现。
Sys internal 层：Sys impl 层对外接口在 kernel mode 封装，提供给其他 MI 模块创建 Device、申请内存、创建线程、管理内存等功能。
Sys ioctl 层：Sys impl 层对外接口的封装，组成 ioctl 的格式，提供给 sys api 调用。
Sys api 层：Sys api 层会在 kernel mode 和 user mode 各编译一次，但目前只开放了 user mode 的调用，即最后开放给客户使用的 API 接口。具体功能请查阅 API 接口说明。

Tips:
Sys impl 层、Sys internal 层、Sys ioctl 层都实现在"mi_sys.ko"， Sys api 层实现在"libmi_sys.so"",头文件为（mi_sys.h、mi_sys_datatype.h、mi_sys.h）。

1.2. 文档格式约束

正体：用于文档正文内容的书写，其中代码段的书写需要用等宽字体。
正体 + 加粗：用于文档正文中重要内容的书写。
斜体：用于文档中 Tips 部分的书写。
斜体 + 加粗：用于文档中 Tips 部分重要内容的书写。

1.3. 关键字说明

ID: Identity document 的缩写，表示唯一编码。
MI: SStar SDK Middle Interface 的缩写，本文中，如果出现"MI_SYS"类似的结构表示的是 MI 的 SYS 模块，如果是单纯的"MI"，泛指整个 SDK。
Hex: 十六进制。
Kernl Mode: 指的是工作在 Kernel 环境下的代码，代码具有直接操作硬件的控制权限，比如 ko 中的函数和线程等。
User Mode: 指的是工作在 User 环境下的，比如客户的应用程序，系统调用等。
APP: Application 的缩写，此处主要指调用 MI API 的应用程序。
API: Application Programming Interface，应用程序接口。
NVR: 全称 Network Video Recorder，即网络视频录像机。
IPC: 全称 IP Camera，即网络摄像机。
HW: 全称 hardware，即硬件。
Dev: 全称 Device，本文中表示 MI 模块设备，1.4.1 有详细解释。
Pass: 本文中表示 MI 模块设备的设备的工作流程，1.4.1 有详细解释。
Chn: 全程 Chanel，本文中表示 MI 模块设备的某个通道，1.4.1 有详细解释。
Port: 本文中表示 MI 模块设备通道中的某个端口，1.4.1 有详细解释。

补充说明：

- 如无单独说明，本文中 MI_SYS 和 SYS，MI_DISP 和 DISP 是相同意思，其余模块名称类似。
- 本文中出现的所有模块名称可以在 MI_ModuleId_e 中查阅其功能描述。

1.4. 流程框图

1.4.1 Dev/Pass/Chn/Port 的关系

一个典型的 MI 模块都会有 Dev/Pass/Port 三级结构如图 1-3。

Dev

一个 MI 模块会有一个或多个 Dev，一般来说不同的 Dev 表示该模块设备需要调用不同的 HW 资源，或者工作在不同的工作模式。比如说，VENC 在编码 H264/H265 时和编码 Jpeg 时，需要调用不同的 HW 资源，这个时候 Dev 有要分开了。

Chn

一个 Dev 会有一个或多个 Chn，Chn 是通道的意思，一般来说不同的 Chn 表示该通道虽然和同 Dev 下的其他 Chn 共享 HW 资源，但是数据来源或者工作模式是不一样的。比如码流来源不同，Chn 一般也不一样。

Port

一个 Chn 会有一个或多个 Port，Port 是端口的意思，由 InputPort 和 OutputPort 组成。一般来说不同的 Port 表示该通道虽然和同 Dev 同 Chn 下的其他 Port 共享 HW 资源和数据来源，但是需要设置的参数是不一样的，比如分辨率不同，Port 一般也不一样。

一般来说，Port 才是客户操作 MI 模块的最小独立单位，因为它明确了所有的信息：**HW 资源、数据来源、参数属性**。

每个 Port 都由 InputPort 和 OutputPort 组成，InputPort 是数据流入的端口，OutputPort 是数据流出的端口。但需要注意的是，一个 Port 并不是总是必须有 InputPort 和 OutputPort 的，这样取决于该模块的行为。比如 Disp 这样的模块就只需要 InputPort，无需 OutputPort，它直接把结果显示到了 Panel 上。Vdec 这样的模块，数据来源可以是用户调用 Vdec 接口直接送入，不经过 InputPort，这样他就没有 OutputPort 了。

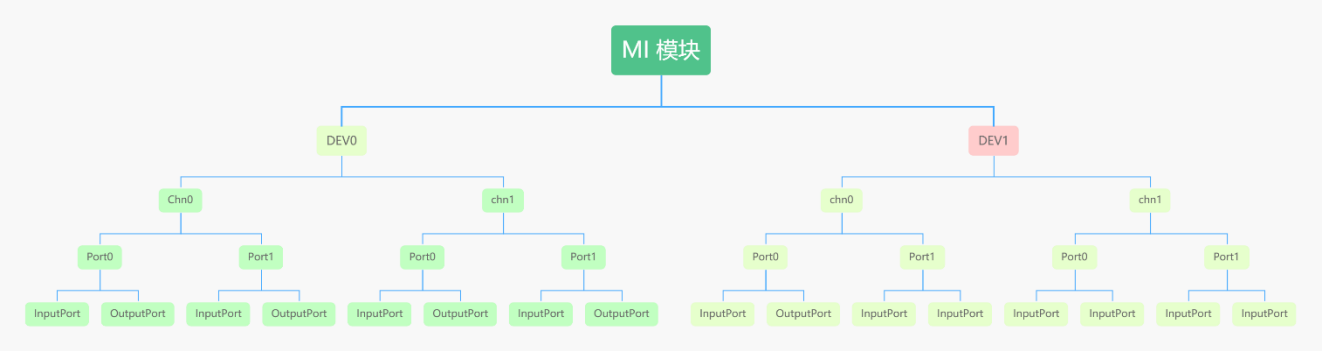

图 1-3 MI 模块的三级结构

Tips:

1. *Chn 和 Port 的界限并不是总是能区分清楚，如果某模块 API 文档的解释和上文的解释有差异，调用该模块请以它的 API 文档为准。*
2. *一个 Port 的 InputPort/OutpuPort 通常只有一个或零个，但某些特殊的模块例外。比如 Vpe 模块，他的结构关系如图所示。也就是说，它有一个 InputPort 和多个 OutputPort。这表示 Vpe 共享一个同一个数据原，但是又必须要有不同规格的输出格式。*

Pass

Pass 则是 MI_SYS V2.0 及以上版本引入的新概念。上文我们说过，Dev 一般对应着对 HW 资源的访问，也就是工作线程本来应该也是创建在这一层的。**但是对于有些功能复杂的模块，则需要使用多组 HW 资源，分阶段的完成功能。这些阶段可以并行工作，但彼此之间有先后顺序，前一阶段的输出是后一阶段的输入，这时，就需要把 Dev 工作的过程分成不同的 Pass 线程来处理了。**

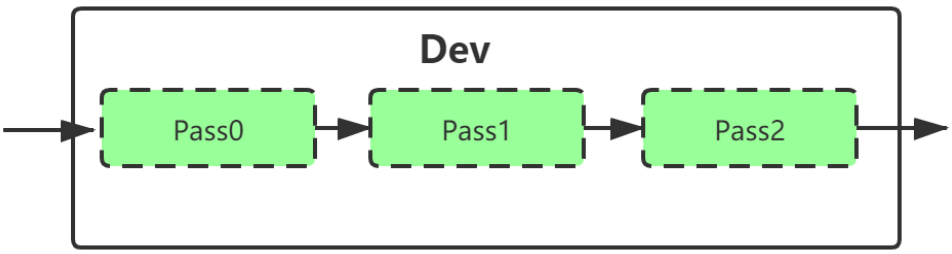

图 1-4 Pass 与 Dev 的关系

Tips:

1. *有些模块的 Dev 其实只有一个 Pass，有些则有多个，一个 Pass 对于着一个工作线程，一次有效的执行表示"一次数据流入->HW 处理->数据流出"的过程。*

2. 需要注意的是 Pass 和 Chn/Port 并不算是从属关系，它是属于 Dev 的一个子模块。它的功能在 MI 内部实现，使用 API 接口无需过于关注。

1.4.2 一个典型的 NVR 数据流

图 1-5 是一个典型的 NVR 数据流模型。流动过程如下：

1. 建立 Vdec->Divp->Disp 的绑定关系；
2. 用户写入一段码流到 Vdec 的 InputPort；
3. Vdec 解码，写入解码后的数据到 Vdec OutpuPort 申请的内存，送入下一级；
4. Divp 接收数据，送到 Divp HW 进行处理，写入 Divp OutputPort，送入下一级；
5. Disp 将接收到的数据显示出来。

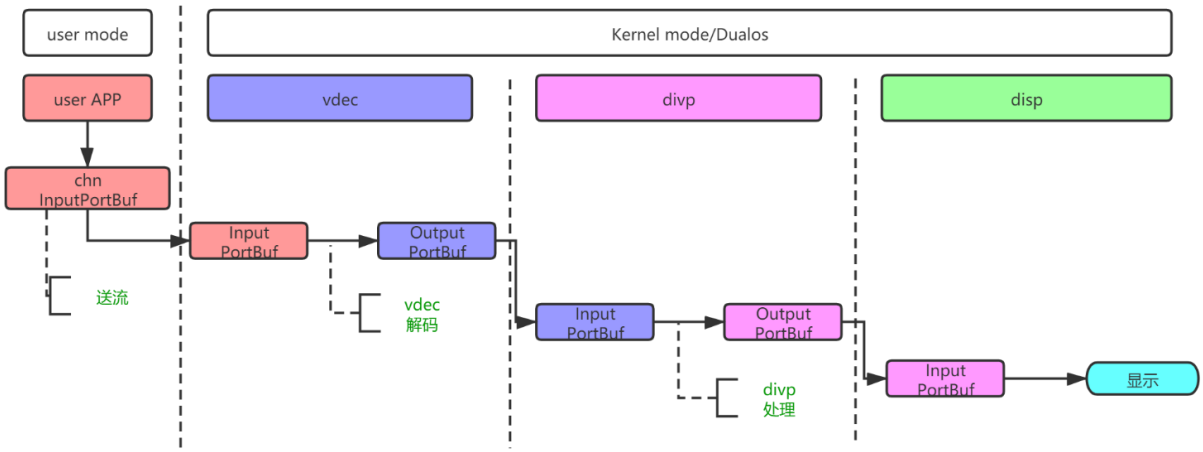

图 1-5 一个典型的 NVR 数据流模型

Tips:

该数据流是原来的做法，现在的 Vdec 模块已经提供单独的接口给客户，可以把数据直接写到 Vdec 的私有内存上，不需要通过啊 SYS 的接口做多余的拷贝。也就是说，Vdec 已经不需要 InputPort 了。

1.4.3 一个典型的 IPC 数据流

图 1-6 是一个典型的 IPC 数据流模型，流动过程如下：

1. 建立 Vif->Vpe->Venc 的绑定关系；
2. Sensor 将数据送入 vif 处理；
3. Vif 将处理后的数据写入 Output Port 申请的内存，送入下一级；
4. Vpe 接收数据，分别送入 Pass0(ISP/SCL0)、Pass1(LDC)、Pass2(SCL1)进行处理，将处理的数据写入 Output Port 申请的内存，送入下一级；
5. Venc 接收数据，送入编码器进行编码处理，将编码后的数据写入 RingPool 内存区；
6. 用户调用 Venc 的接口取流，送入用户业务层 App。

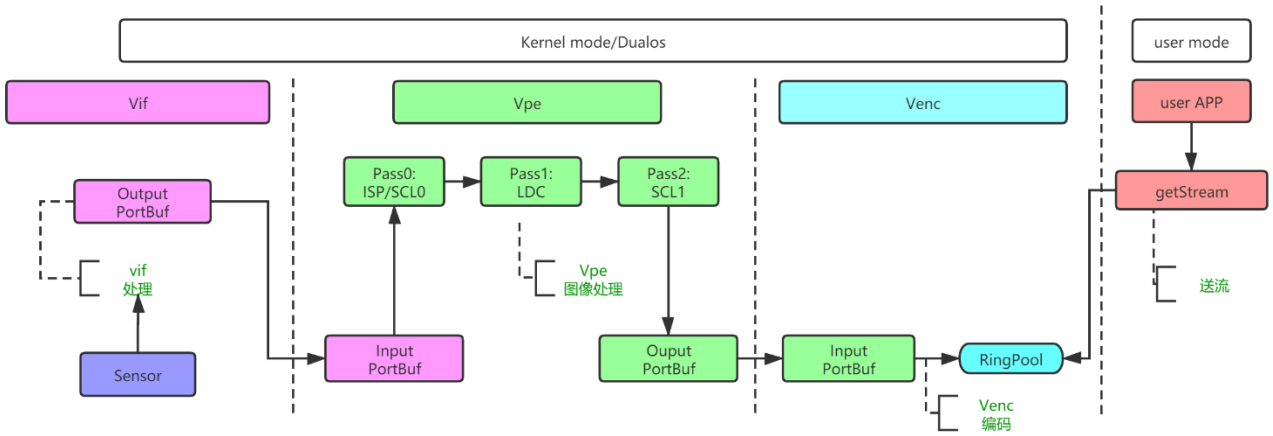

图 1-6 一个典型的 IPC 数据流模型

Tips:

Vpe 内部的 3 个 Pass 之间的数据流动实际比图中要复杂一些。但这都 MI 内部的处理逻辑，使用 API 时无需过度关注。

2. API 参考

2.1. API 描述格式说明

本手册使用 9 个参考域描述 个参考域描述 API 的相关信息，它们作用如 表 1-1 所表示。

标签	功能
功能	简要描述 API 的主要功能。
语法	列出调用 API 应包括的头文件以及 API 的原型声明。
参数	列出 API 的参数、参数说明及参数属性。
返回值	列出 API 所有可能的返回值及其含义。
依赖	列出 API 包含的头文件和 API 编译时要链接的库文件。
注意	列出使用 API 时应注意的事项。
举例	列出使用 API 的实例。
相关主题	调用上下文中有关联的接口。

2.2. 功能模块 API 列表

如前文所述，我们可以粗略的把 MI_SYS 的 API 分为三大类：系统功能类、数据流类、内存管理类。

API 名	功能
系统功能类	
MI_SYS_Init	初始化 MI_SYS 系统
MI_SYS_Exit	析构 MI_SYS 系统
MI_SYS_GetVersion	获取 MI 的系统版本号
MI_SYS_GetCurPts	获取 MI 系统当前时间戳
MI_SYS_InitPtsBase	初始化 MI 系统基准时间戳
MI_SYS_SyncPts	同步 MI 系统时间戳
MI_SYS_SetReg	设置寄存器的值，调试用
MI_SYS_GetReg	获取寄存器的值，调试用
MI_SYS_ReadUuid	获取 Chip 的 Unique ID
数据流类	
MI_SYS_BindChnPort	数据源输出端口到接收者输入端口的绑定
MI_SYS_BindChnPort2	数据源输出端口到接收者输入端口的绑定，需要指定工作模式
MI_SYS_UnBindChnPort	数据源输出端口到接收者输入端口的去绑定
MI_SYS_GetBindbyDest	查询数据接收者输入端口的对应的源输出端口
MI_SYS_ChnInputPortGetBuf	获取通道 inputPort 的 buf
MI_SYS_ChnInputPortPutBuf	将通道 inputPort 的 buf 加到待处理队列
MI_SYS_ChnOutputPortGetBuf	获取通道 outputPort 的 buf
MI_SYS_ChnOutputPortPutBuf	释放通道 outputPort 的 buf
MI_SYS_SetChnOutputPortDepth	设置通道 OutputPort 的深度
MI_SYS_ChnPortInjectBuf	向模块通道 inputPort 注入 outputPort Buf 数据
MI_SYS_GetFd	获取当前通道等待事件的文件描述符
MI_SYS_CloseFd	关闭当前通道的文件描述符
内存管理类	
MI_SYS_SetChnMMAConf	设置模块设备通道输出端口默认分配内存的 MMA 池名称
MI_SYS_GetChnMMAConf	获取模块设备通道输出端口默认分配内存的 MMA 池名称
MI_SYS_ConfDevPubPools	配置并初始化模块公共缓冲池
MI_SYS_ReleaseDevPubPools	释放模块公共缓冲池
MI_SYS_ConfGloPubPools	配置并初始化 MI 全系统默认 VB 缓存池
MI_SYS_ReleaseGloPubPools	释放 MI 全系统默认 VB 缓存池
MI_SYS_MMA_Alloc	应用程序从 MMA 内存管理池申请物理连续内存
MI_SYS_MMA_Free	在用户态释放 MMA 内存管理池中分配到的内存
MI_SYS_Mmap	映射物理内存到 CPU 虚拟地址
MI_SYS_Munmap	取消物理内存到虚拟地址的映射
MI_SYS_FlushInvCache	Flush cache CPU 虚拟地址
MI_SYS_ConfigPrivateMMAPool	为模快配置私有 MMA Heap

API 名	功能
MI_SYS_PrivateDevChnHeapAlloc	从模块通道私有 MMA Pool 申请内存
MI_SYS_PrivateDevChnHeapFree	从模块通道私有 MMA pool 释放内存

2.3. 系统功能类 API

2.3.1 MI_SYS_Init

➢ 功能

MI_SYS 初始化，MI_SYS 模块为系统内其它 MI 模组提供基础支援，需要早于系统内其它 MI 模组初始化，否则其它内其它 stream 类型的模组初始化时会失败。

➢ 语法

MI_S32 MI_SYS_Init(void);

➢ 形参

N/A

➢ 返回值

- 0 成功。
- 非 0 失败，参照错误码。

➢ 依赖

- 头文件：mi_sys_datatype.h、mi_sys.h
- 库文件：libmi_sys.a / libmi_sys.so

※ 注意

- MI_SYS_Init 需要早于其他 MI 模组的 Init 函数调用。
- 可以重复调用 MI_SYS_Init，但必须和 MI_SYS_Exit 成对使用，否则会报错。
- 系统在内核启动参数内，需要配置好 MMA 内存堆的配置参数。

➢ 举例

```
MI_S32 ST_Sys_Init(void)
{
    MI_SYS_Version_t stVersion;
    MI_U64 u64Pts = 0;

    STCHECKRESULT(MI_SYS_Init());

    memset(&stVersion, 0x0, sizeof(MI_SYS_Version_t));
    STCHECKRESULT(MI_SYS_GetVersion(&stVersion));
    ST_INFO("u8Version:%s\n", stVersion.u8Version);

    STCHECKRESULT(MI_SYS_GetCurPts(&u64Pts));
    ST_INFO("u64Pts:0x%llx\n", u64Pts);

    u64Pts = 0xF1237890F1237890;
    STCHECKRESULT(MI_SYS_InitPtsBase(u64Pts));

    u64Pts = 0xE1237890E1237890;
    STCHECKRESULT(MI_SYS_SyncPts(u64Pts));
```

```
    return MI_SUCCESS;
}

MI_S32 ST_Sys_Exit(void)
{
    STCHECKRESULT(MI_SYS_Exit());

    return MI_SUCCESS;
}
```

系统 Init 调用 Sample

Tips:

本示例适用于: MI_SYS_Init / MI_SYS_Exit / MI_SYS_GetVersion / MI_SYS_GetCurPts / MI_SYS_InitPtsBase / MI_SYS_SyncPts / MI_SYS_ReadUuid .

➢ 相关主题

MI_SYS_Exit

2.3.2 MI_SYS_Exit

➢ 功能

MI_SYS 初始化， 调用 MI_SYS_Exit 之前，需要确保系统内所有其他模组都已经完成去初始化，所有 VBPOOL 都已经销毁，否则 MI_SYS_Exit 会返回失败。

➢ 语法

MI_S32 MI_SYS_Exit (void);

➢ 形参

N/A

➢ 返回值

- 0 成功。
- 非 0 失败，参照错误码。

➢ 依赖

- 头文件：mi_sys_datatype.h、mi_sys.h
- 库文件：libmi_sys.a / libmi_sys.so

※ 注意

- MI_SYS_Exit 调用前需要确保系统内所有其他模组都已经完成去初始化。
- MI_SYS_Exit 调用前需要确保系统内所有创建的 VBPOOL 已经成功销毁。

➢ 举例

参见 系统 Init 调用 Sample 举例。

➢ 相关主题

MI_SYS_Init

2.3.3 MI_SYS_GetVersion

➢ 功能

获取 MI 的系统版本号。

➢ 语法

MI_S32 MI_SYS_GetVersion (MI_SYS_Version_t *pstVersion);

➢ 形参

参数名称	描述	输入/输出
pstVersion	系统版本号返回数据结构指针	输出

➢ 返回值

- 0 成功。
- 非 0 失败，参照错误码。

➢ 依赖

- 头文件：mi_sys_datatype.h、mi_sys.h
- 库文件：libmi_sys.a / libmi_sys.so

※ 注意

N/A

➢ 举例

参见 系统 Init 调用 Sample 举例。

➢ 相关主题

N/A

2.3.4 MI_SYS_GetCurPts

➢ 功能

获取 MI 系统当前时间戳。

➢ 语法

MI_S32 MI_SYS_GetCurPts (MI_U64 *pu64Pts);

➢ 形参

参数名称	描述	输入/输出
pu64Pts	系统当前时间戳返回地址	输出

➢ 返回值
- 0 成功。
- 非 0 失败，参照错误码。

➢ 依赖
- 头文件：mi_sys_datatype.h、mi_sys.h
- 库文件：libmi_sys.a / libmi_sys.so

※ 注意
N/A

➢ 举例
参见 系统 Init 调用 Sample 举例。

➢ 相关主题
N/A

2.3.5 MI_SYS_InitPtsBase

➢ 功能
初始化 MI 系统时间戳基准。

➢ 语法
MI_S32 MI_SYS_InitPtsBase (MI_U64 u64PtsBase);

➢ 形参

参数名称	描述	输入/输出
u64PtsBase	设置的系统时间戳基准	输入

➢ 返回值
- 0 成功。
- 非 0 失败，参照错误码。

➢ 依赖
- 头文件：mi_sys_datatype.h、mi_sys.h
- 库文件：libmi_sys.a / libmi_sys.so

※ 注意
N/A

➢ 举例
参见 系统 Init 调用 Sample 举例。

➢ 相关主题
N/A

2.3.6 MI_SYS_SyncPts

➢ 功能

微调同步 MI 系统时间戳。

➢ 语法

MI_S32 MI_SYS_SyncPts (MI_U64 u64Pts);

➢ 形参

参数名称	描述	输入/输出
u64Pts	微调后的系统时间戳基准	输入

➢ 返回值

- 0 成功。
- 非 0 失败，参照错误码。

➢ 依赖

- 头文件：mi_sys_datatype.h、mi_sys.h
- 库文件：libmi_sys.a / libmi_sys.so

※ 注意

N/A

➢ 举例

参见 系统 Init 调用 Sample 举例。

➢ 相关主题

N/A

2.3.7 MI_SYS_SetReg

➢ 功能

设置寄存器的值，调试用。

➢ 语法

MI_S32 MI_SYS_SetReg (MI_U32 u32RegAddr, MI_U16 u16Value，MI_U16 u16Mask);

➢ 形参

参数名称	描述	输入/输出
u32RegAddr	Register 总线之地址	输入
u16Value	待写入之 16bit 寄存器值	输入
u16Mask	本次写入寄存器值之 Mask 遮挡栏位	输入

➢ 返回值
- 0 成功。
- 非 0 失败，参照错误码。

➢ 依赖
- 头文件：mi_sys_datatype.h、mi_sys.h
- 库文件：libmi_sys.a / libmi_sys.so

※ 注意
N/A

➢ 举例
N/A

➢ 相关主题
N/A

2.3.8 MI_SYS_GetReg

➢ 功能

读取寄存器的值，调试用。

➢ 语法

MI_S32 MI_SYS_GetReg (MI_U32 u32RegAddr, MI_U16 *pu16Value);

➢ 形参

参数名称	描述	输入/输出
u32RegAddr	Register 总线之地址	输入
pu16Value	待读回 16bit 寄存器值返回地址	输出

➢ 返回值
- 0 成功。
- 非 0 失败，参照错误码。

➢ 依赖
- 头文件：mi_sys_datatype.h、mi_sys.h
- 库文件：libmi_sys.a / libmi_sys.so

※ 注意
N/A

➢ 举例
N/A

➢ 相关主题
N/A

2.3.9 MI_SYS_ReadUuid

➢ 功能

获取 Chip 的 Unique ID

➢ 语法

MI_S32 MI_SYS_ReadUuid (MI_U64 *u64Uuid);

➢ 形参

参数名称	描述	输入/输出
u64Uuid	获取 chip unique ID 值的指针	输出

➢ 返回值

- 0 成功。
- 非 0 失败，参照错误码。

➢ 依赖

- 头文件：mi_sys_datatype.h、mi_sys.h
- 库文件：libmi_sys.a / libmi_sys.so

※ 注意

N/A

➢ 举例

参见 系统 Init 调用 Sample 举例。

2.3.10 MI_SYS_EnableChnOutputPortLowLatency

➢ 功能

打开或者关闭通道 output 端口的低延迟输出功能

➢ 语法

MI_S32 MI_SYS_EnableChnOutputPortLowLatency(MI_SYS_ChnPort_t *pstChnPort,MI_BOOL bEnable , MI_U32 u32Param);

➢ 形参

参数名称	描述	输入/输出
pstChnPort	指向模块通道之 output 端口的指针	输入
bEnable	TRUE：打开 FALSE:关闭	输入
u32Param	Low Latency 参数，其含义由具体模块决定 VPE 用于设置 Line Count，表示写完多少条 Line 后就 Output 给 User 或后级	输入

➢ 返回值

- 0 成功。
- 非 0 失败，参照错误码。

➢ 依赖
- 头文件：mi_sys_datatype.h、mi_sys.h
- 库文件：libmi_sys.a / libmi_sys.so

※ 注意
当 bEnable 为 true 时，u32Param 必须大于 0，否则设置无效

➢ 举例

```
MI_SYS_ChnPort_t stChnPort;
stChnPort.eModId = E_MI_MODULE_ID_VPE;
stChnPort.u32DevId = 0;
stChnPort.u32ChnId = 0;
stChnPort.u32PortId = 0;
//打开 VPE 通道 1 output 端口 0 的低延迟输出功能,设定完成 100 行即可输出
MI_SYS_EnableChnOutputPortLowLatency(&stChnPort,TRUE , 100);

//关闭 VPE 通道 1 output 端口 0 的低延迟输出功能,
MI_SYS_EnableChnOutputPortLowLatency(&stChnPort,FALSE , 0);
```

2.4. 数据流类 API

2.4.1 MI_SYS_BindChnPort

➢ 功能
数据源输出端口到数据接收者输入端口的绑定。

➢ 语法
MI_S32 MI_SYS_BindChnPort(MI_SYS_ChnPort_t *pstSrcChnPort, MI_SYS_ChnPort_t *pstDstChnPort, ,MI_U32 u32SrcFrmrate, MI_U32 u32DstFrmrate);

➢ 形参

参数名称	参数含义	输入/输出
pstSrcChnPort	源端口配置信息数据结构指针	输入
pstDstChnPort	目标端口配置信息数据结构指针	输入
u32SrcFrmrate	源端口配置的帧率	输入
u32DstFrmrate	目标端口配置的帧率	输入

➢ 返回值
- 0 成功。
- 非 0 失败，参照错误码。

➢ 依赖
- 头文件：mi_sys_datatype.h、mi_sys.h
- 库文件：libmi_sys.a / libmi_sys.so

※ 注意

- 源端口必须是通道输出端口。
- 目标端口必须是通道输入端口。
- 源和目标端口必须之前没有被绑定过。
- 本接口只支持按照 E_MI_SYS_BIND_TYPE_FRAME_BASE 方式绑定模块，在 Version 2.0 以上版本不推荐使用，请使用 MI_SYS_BindChnPort2 替代。

➢ 举例

```
MI_SYS_ChnPort_t stSrcChnPort;
MI_SYS_ChnPort_t stDstChnPort;
MI_U32 u32SrcFrmrate;
MI_U32 u32DstFrmrate;

stSrcChnPort.eModId = E_MI_MODULE_ID_VPE;
stSrcChnPort.u32DevId = 0;
stSrcChnPort.u32ChnId = 0;
stSrcChnPort.u32PortId = 0;
stDstChnPort.eModId = E_MI_MODULE_ID_VENC;
stDstChnPort.u32DevId = 0;
stDstChnPort.u32ChnId = 0;
stDstChnPort.u32PortId = 0;
u32SrcFrmrate = 30;
u32DstFrmrate = 30;
MI_SYS_BindChnPort(&stSrcChnPort, &stDstChnPort, u32SrcFrmrate, u32DstFrmrate);
```

➢ 相关主题

MI_SYS_BindChnPort2、MI_SYS_UnBind_ChnPort。

2.4.2 MI_SYS_BindChnPort2

➢ 功能

数据源输出端口到数据接收者输入端口的绑定，需要额外指定工作模式。

➢ 语法

MI_S32 MI_SYS_BindChnPort2(MI_SYS_ChnPort_t *pstSrcChnPort, MI_SYS_ChnPort_t *pstDstChnPort,MI_U32 u32SrcFrmrate, MI_U32 u32DstFrmrate, MI_SYS_BindType_e eBindType, MI_U32 u32BindParam);

➢ 形参

参数名称	参数含义	输入/输出
pstSrcChnPort	源端口配置信息数据结构指针。	输入
pstDstChnPort	目标端口配置信息数据结构指针。	输入
u32SrcFrmrate	源端口配置的帧率	输入
u32DstFrmrate	目标端口配置的帧率	输入
eBindType	源端口与目标端口连接的工作模式，参考 MI_SYS_BindType_e	输入
u32BindParam	不同工作模式需带入的额外参数。	输入

➢ 返回值

- 0 成功。
- 非 0 失败，参照错误码。

➢ 依赖

- 头文件：mi_sys_datatype.h、mi_sys.h
- 库文件：libmi_sys.a / libmi_sys.so

※ 注意

- 源端口必须是通道输出端口。
- 目标端口必须是通道输入端口。
- 源和目标端口必须之前没有被绑定过
- 旧版本 MI SYS 不提供此接口，如没有找到，不用设置。
- 各种 eBindType 使用场景如下：

eBindType	适用场景
E_MI_SYS_BIND_TYPE_SW_LOW_LATENCY	u32BindParam 表示低时延值，单位 ms
E_MI_SYS_BIND_TYPE_HW_RING	u32BindParam 表示 ring buffer depth，目前是只有 vpe 和 venc（h264/h265）支持这种模式，只支持一路
E_MI_SYS_BIND_TYPE_REALTIME	u32BindParam 未使用，jpe imi 会走这种模式，只支持一路
E_MI_SYS_BIND_TYPE_FRAME_BASE	u32BindParam 未使用，默认是走这种 frame mode

➢ 举例

```
MI_SYS_ChnPort_t stSrcChnPort;
MI_SYS_ChnPort_t stDstChnPort;
MI_U32 u32SrcFrmrate;
MI_U32 u32DstFrmrate;
MI_SYS_BindType_e eBindType;
MI_U32 u32BindParam;

// a.  vpe 与 venc 连接方式为 E_MI_SYS_BIND_TYPE_FRAME_BASE，代码如下：
stSrcChnPort.eModId = E_MI_MODULE_ID_VPE;
```

```
    stSrcChnPort.u32DevId = 0;
    stSrcChnPort.u32ChnId = 0;
    stSrcChnPort.u32PortId = 0;
    stDstChnPort.eModId = E_MI_MODULE_ID_VENC;
    stDstChnPort.u32DevId = 0;
    stDstChnPort.u32ChnId = 0;
    stDstChnPort.u32PortId = 0;
    u32SrcFrmrate = 30;
    u32DstFrmrate = 30;
    eBindType = E_MI_SYS_BIND_TYPE_FRAME_BASE;
    u32BindParam = 0;
      STCHECKRESULT(MI_SYS_BindChnPort2(&stSrcChnPort, &stDstChnPort, u32SrcFrmrate, u32DstFrmrate, e
BindType, u32BindParam));

     // b.  vpe 与 jpe 连接方式为 E_MI_SYS_BIND_TYPE_REALTIME，代码如下：
    stSrcChnPort.eModId = E_MI_MODULE_ID_VPE;
    stSrcChnPort.u32DevId = 0;
    stSrcChnPort.u32ChnId = 0;
    stSrcChnPort.u32PortId = 0;
    stDstChnPort.eModId = E_MI_MODULE_ID_VENC;
    stDstChnPort.u32DevId = 1;
    stDstChnPort.u32ChnId = 0;
    stDstChnPort.u32PortId = 0;
    u32SrcFrmrate = 30;
    u32DstFrmrate = 30;
    eBindType = E_MI_SYS_BIND_TYPE_REALTIME;
    u32BindParam = 0;
      STCHECKRESULT(MI_SYS_BindChnPort2(&stSrcChnPort, &stDstChnPort, u32SrcFrmrate, u32DstFrmrate, e
BindType, u32BindParam));

     // c.  vpe 与 venc 连接方式为 E_MI_SYS_BIND_TYPE_HW_RING，代码如下：
    tSrcChnPort.eModId = E_MI_MODULE_ID_VPE;
    stSrcChnPort.u32DevId = 0;
    stSrcChnPort.u32ChnId = 0;
    stSrcChnPort.u32PortId = 0;
    stDstChnPort.eModId = E_MI_MODULE_ID_VENC;
    stDstChnPort.u32DevId = 0;
    stDstChnPort.u32ChnId = 0;
    stDstChnPort.u32PortId = 0;
    u32SrcFrmrate = 30;
    u32DstFrmrate = 30;
    eBindType = E_MI_SYS_BIND_TYPE_HW_RING;
    u32BindParam = 1080;//假设 vpe output resolution 为 1920*1080，设置 ring buffer depth 为 1080
    STCHECKRESULT(MI_SYS_BindChnPort2(&stSrcChnPort, &stDstChnPort, u32SrcFrmrate, u32DstFrmrate, eBi
ndType, u32BindParam));
```

➢ 相关主题
MI_SYS_BindChnPort2、MI_SYS_UnBind_ChnPort。

2.4.3 MI_SYS_UnBind_ChnPort

➢ 功能
数据源输出端口到数据接收者输入端口之间的去绑定。

➢ 语法
MI_S32 MI_SYS_UnBindChnPort(MI_SYS_ChnPort_t *pstSrcChnPort, MI_SYS_ChnPort_t *pstDstChnPort);

➢ 形参

参数名称	描述	输入/输出
pstSrcChnPort	源端口配置信息数据结构指针。	输入
pstDstChnPort	目标端口配置信息数据结构指针。	输入

➢ 返回值
- 0 成功。
- 非 0 失败，参照错误码。

➢ 依赖
- 头文件：mi_sys_datatype.h、mi_sys.h
- 库文件：libmi_sys.a / libmi_sys.so

※ 注意
- 源端口必须是通道输出端口。
- 目标端口必须是通道输入端口。
- 源和目标端口之间之前必须已经被绑定过

➢ 举例
N/A

➢ 相关主题
MI_SYS_Bind_ChnPort、MI_SYS_BindChnPort2。

2.4.4 MI_SYS_GetBindbyDest

➢ 功能
查询数据接收者输入端口的对应的源输出端口。

➢ 语法
MI_S32 MI_SYS_GetBindbyDest (MI_SYS_ChnPort_t *pstDstChnPort, MI_SYS_ChnPort_t *pstSrcChnPort);

➢ 形参

参数名称	描述	输入/输出
pstDstChnPort	目标端口配置信息数据结构指针。	输入
pstDstChnPort	源端口配置信息数据结构指针。	输出

➢ 返回值

- 0 成功。
- 非 0 失败，参照错误码。

➢ 依赖

- 头文件：mi_sys_datatype.h、mi_sys.h
- 库文件：libmi_sys.a / libmi_sys.so

※ 注意

- 目标端口必须是通道输入端口。
- 目标端口之前必须已经被绑定过

➢ 举例

N/A

➢ 相关主题

N/A

2.4.5 MI_SYS_ChnInputPortGetBuf

➢ 功能

分配通道 input 端口对应的 buf object。

➢ 语法

MI_S32 MI_ SYS_ChnInputPortGetBuf (MI_SYS_ChnPort_t *pstChnPort，MI_SYS_BufConf_t *pstBufConf, MI_SYS_BufInfo_t *pstBufInfo, MI_SYS_BUF_HANDLE *phHandle , MI_S32 s32TimeOutMs);

➢ 形参

参数名称	描述	输入/输出
pstChnPort	指向模块通道之 input 端口的指针	输入
pstBufConf	待分配内存配置信息	输入
pstPortBuf	返回之 buf 指针	输出
phHandle	获取的 intputPort Buf 的 Idr handle	输出
s32TimeOutMs	等待超时的毫秒数	输出

➢ 返回值

- 0 成功。
- 非 0 失败，参照错误码。

➢ 依赖

- 头文件：mi_sys_datatype.h、mi_sys.h
- 库文件：libmi_sys.a / libmi_sys.so

※ 注意

N/A

➢ 举例

```
MI_SYS_ChnPort_t stVpeChnInput;
MI_SYS_BUF_HANDLE hHandle = 0;
MI_SYS_BufConf_t stBufConf;
MI_SYS_BufInfo_t stBufInfo;
struct timeval stTv;
MI_U16 u16Width = 1920, u16Height = 1080;
FILE *fp = NULL;

memset(&stVpeChnInput, 0x0, sizeof(MI_SYS_ChnPort_t));
memset(&stBufConf, 0x0, sizeof(MI_SYS_BufConf_t));
memset(&stBufInfo, 0x0, sizeof(MI_SYS_BufInfo_t));

stVpeChnInput.eModId = E_MI_MODULE_ID_VPE;
stVpeChnInput.u32DevId = 0;
stVpeChnInput.u32ChnId = 0;
stVpeChnInput.u32PortId = 0;

fp = fopen("/mnt/vpeport0_1920x1080_pixel0_737.raw","rb");
if(fp == NULL)
{
    printf("file %s open fail\n", "/mnt/vpeport0_1920x1080_pixel0_737.raw");
    return 0;
}

while(1)
{
    stBufConf.eBufType = E_MI_SYS_BUFDATA_FRAME;
    gettimeofday(&stTv, NULL);
    stBufConf.u64TargetPts = stTv.tv_sec*1000000 + stTv.tv_usec;
    stBufConf.stFrameCfg.eFormat = E_MI_SYS_PIXEL_FRAME_YUV422_YUYV;
    stBufConf.stFrameCfg.eFrameScanMode = E_MI_SYS_FRAME_SCAN_MODE_PROGRESSIVE;
    stBufConf.stFrameCfg.u16Width = u16Width;
    stBufConf.stFrameCfg.u16Height = u16Height;

    if(MI_SUCCESS  == MI_SYS_ChnInputPortGetBuf(&stVpeChnInput,&stBufConf,&stBufInfo,&hHandle,0))
    {
        if(fread(stBufInfo.stFrameData.pVirAddr[0], u16Width*u16Height*2, 1, fp) <= 0)
        {
```

```
            fseek(fp, 0, SEEK_SET);
        }

        MI_SYS_ChnInputPortPutBuf(hHandle,&stBufInfo, FALSE);
    }
}
```

MI_SYS_ChnInputPortGetBuf 调用 Sample

➢ 相关主题

MI_SYS_ChnInputPortPutBuf / MI_SYS_ChnPortInjectBuf

2.4.6 MI_SYS_ChnInputPortPutBuf

➢ 功能

把通道 input 端口对应的 buf object 加到待处理队列。

➢ 语法

MI_S32 MI_SYS_ChnInputPortPutBuf (MI_SYS_BUF_HANDLE hHandle ， MI_SYS_BufInfo_t *pstPortBuf，MI_BOOL bDropBuf);

➢ 形参

参数名称	描述	输入/输出
hHandle	当前 buf 的 Idr Handle	输入
pstPortBuf	待提交之 buf 指针	输入
bDropBuf	直接放弃对 buf 的修改不提交	输入

➢ 返回值

- 0 成功。
- 非 0 失败，参照错误码。

➢ 依赖

- 头文件：mi_sys_datatype.h、mi_sys.h
- 库文件：libmi_sys.a / libmi_sys.so

※ 注意

N/A

➢ 举例

参见 MI_SYS_ChnInputPortGetBuf 调用 Sample 举例。

➢ 相关主题

MI_SYS_ChnInputPortGetBuf / MI_SYS_ChnPortInjectBuf

2.4.7 MI_SYS_ChnOutputPortGetBuf

➢ 功能

分配通道 input 端口对应的 buf object。

➢ 语法

MI_S32 MI_SYS_ChnOutputPortGetBuf (MI_SYS_ChnPort_t *pstChnPort, MI_SYS_BufInfo_t *pstBufInfo , MI_SYS_BUF_HANDLE *phHandle);

➢ 形参

参数名称	描述	输入/输出
pstChnPort	指向模块通道之 input 端口的指针	输入
pstBufInfo	返回之 buf 指针	输出
phHandle	获取的 outputPort Buf 的 Idr handle	输出

➢ 返回值

- 0 成功。
- 非 0 失败，参照错误码。

➢ 依赖

- 头文件：mi_sys_datatype.h、mi_sys.h
- 库文件：libmi_sys.a / libmi_sys.so

※ 注意

N/A

➢ 举例

```
MI_SYS_ChnPort_t stChnPort;
MI_SYS_BufInfo_t stBufInfo;
MI_SYS_BUF_HANDLE stBufHandle;
MI_S32 s32Ret = MI_SUCCESS;
MI_S32 s32Fd = 0;
fd_set read_fds;
struct timeval TimeoutVal;
char szFileName[128];
int fd = 0;
MI_U32 u32GetFramesCount = 0;
MI_BOOL _bWriteFile = TRUE;

stChnPort.eModId = E_MI_MODULE_ID_DIVP;
stChnPort.u32DevId = 0;
stChnPort.u32ChnId = DIVP_CHN_FOR_VDF;
stChnPort.u32PortId = 0;

s32Ret = MI_SYS_GetFd(&stChnPort, &s32Fd);
if(MI_SUCCESS != s32Ret)
{
```

```
    ST_ERR("MI_SYS_GetFd 0, error, %X\n", s32Ret);
    return NULL;
}
s32Ret = MI_SYS_SetChnOutputPortDepth(&stChnPort, 2, 3);
if (MI_SUCCESS != s32Ret)
{
    ST_ERR("MI_SYS_SetChnOutputPortDepth err:%x, chn:%d,port:%d\n", s32Ret,
        stChnPort.u32ChnId, stChnPort.u32PortId);
    return NULL;
}

sprintf(szFileName, "divp%d.es", stChnPort.u32ChnId);
printf("start to record %s\n", szFileName);
fd = open(szFileName, O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
if (fd < 0)
{
    ST_ERR("create %s fail\n", szFileName);
}

while (1)
{
    FD_ZERO(&read_fds);
    FD_SET(s32Fd, &read_fds);

    TimeoutVal.tv_sec  = 1;
    TimeoutVal.tv_usec = 0;

    s32Ret = select(s32Fd + 1, &read_fds, NULL, NULL, &TimeoutVal);

    if(s32Ret < 0)
    {
        ST_ERR("select failed!\n");
        //  usleep(10 * 1000);
        continue;
    }
    else if(s32Ret == 0)
    {
        ST_ERR("get divp frame time out\n");
        //usleep(10 * 1000);
        continue;
    }
    else
    {
        if(FD_ISSET(s32Fd, &read_fds))
        {
            s32Ret = MI_SYS_ChnOutputPortGetBuf(&stChnPort, &stBufInfo, &stBufHandle);
```

```
            if(MI_SUCCESS != s32Ret)
            {
                //ST_ERR("MI_SYS_ChnOutputPortGetBuf err, %x\n", s32Ret);
                continue;
            }

            // save one Frame YUV data
            if (fd > 0)
            {
                if(_bWriteFile)
                {
                    write(fd, stBufInfo.stFrameData.pVirAddr[0], stBufInfo.stFrameData.u16Height * stBufInf
o.stFrameData.u32Stride[0] +
                        stBufInfo.stFrameData.u16Height * stBufInfo.stFrameData.u32Stride[1] /2);
                }

            }

            ++u32GetFramesCount;
            printf("channelId[%u] u32GetFramesCount[%u]\n", stChnPort.u32ChnId, u32GetFramesCount
);

            MI_SYS_ChnOutputPortPutBuf(stBufHandle);
        }
    }
}

if (fd > 0)
{
    close(fd);
    fd = -1;
    }

MI_SYS_SetChnOutputPortDepth(&stChnPort, 0, 3);
printf("exit record\n");
return NULL;
```

MI_SYS_ChnOutputPortGetBuf 调用 Sample

- 相关主题

  MI_SYS_ChnOutputPortPutBuf / MI_SYS_ChnPortInjectBuf

2.4.8 MI_SYS_ChnOutputPortPutBuf

➢ 功能

释放通道 output 端口对应的 buf object。

➢ 语法

MI_S32 MI_**S**YS_ChnOutPortPutBuf (MI_SYS_BUF_HANDLE hBufHandle);

➢ 形参

参数名称	描述	输入/输出
hBufHandle	待提交之 buf 的 Idr handle	输入

➢ 返回值

- 0 成功。
- 非 0 失败，参照错误码。

➢ 依赖

- 头文件：mi_sys_datatype.h、mi_sys.h
- 库文件：libmi_sys.a / libmi_sys.so

※ 注意

N/A

➢ 举例

参见 MI_SYS_ChnOutputPortGetBuf 调用 Sample 举例。

➢ 相关主题

MI_SYS_ChnOutputPortGetBuf / MI_SYS_ChnPortInjectBuf

2.4.9 MI_SYS_SetChnOutputPortDepth

➢ 功能

设置通道 output 端口对应的系统 buf 数量和用户可以拿到的 buf 数量。

➢ 语法

MI_S32 MI_SYS_SetChnOutputPortDepth(MI_SYS_ChnPort_t *pstChnPort, MI_U32 u32UserFrameDepth, MI_U32 u32BufQueueDepth);

➢ 形参

参数名称	描述	输入/输出
pstChnPort	指向模块通道之 output 端口的指针	输入
u32UserFrameDepth	设置该 output 用户可以拿到的 buf 最大数量	输入
u32BufQueueDepth	设置该 output 系统 buf 最大数量	输入

➢ 返回值
- 0 成功。
- 非 0 失败，参照错误码。

➢ 依赖
- 头文件：mi_sys_datatype.h、mi_sys.h
- 库文件：libmi_sys.a / libmi_sys.so

※ 注意
假如使用完 OutputPortBuf（不再需要调用 MI_SYS_ChnOutputPortGetBuf / MI_ **S**YS_ChnOutPortPutBuf）之后，可以把 u32UserFrameDepth 设置为 0，这样就不会影响底层调用的速度。

➢ 举例
参见 MI_SYS_ChnOutputPortGetBuf 调用 Sample 举例。

➢ 相关主题
N/A

2.4.10 MI_SYS_ChnPortInjectBuf

➢ 功能
把获取的 outputPort buf 插到指定的 inputPort Buf Queue 中

➢ 语法
MI_S32 MI_SYS_ChnPortInjectBuf(MI_SYS_BUF_HANDLE hHandle ,MI_SYS_ChnPort_t *pstChnInputPort);

➢ 形参

参数名称	描述	输入/输出
pstChnPort	指向模块通道之 input 端口的指针	输入
hHandle	获取的 outputPort Buf 的 Idr handle	输出

➢ 返回值
- 0 成功。
- 非 0 失败，参照错误码。

➢ 依赖
- 头文件：mi_sys_datatype.h、mi_sys.h
- 库文件：libmi_sys.a / libmi_sys.so

※ 注意
N/A

➢ 举例
参见 MI_SYS_ChnOutputPortGetBuf 调用 Sample 举例。

➢ 相关主题

MI_SYS_ChnInputPortGetBuf / MI_SYS_ChnInputPortPutBuf / MI_SYS_ChnOutputPortGetBuf / MI_SYS_ChnOutputPortPutBuf

2.4.11 MI_SYS_GetFd

➢ 功能

获取当前 output Port 等待事件的文件描述符

➢ 语法

MI_S32 MI_SYS_GetFd(MI_SYS_ChnPort_t *pstChnPort ， MI_S32 *ps32Fd);

➢ 形参

参数名称	描述	输入/输出
pstChnPort	端口信息结构体指针	输入
ps32Fd	等待事件的文件描述符	输出

➢ 返回值

- 0 成功。
- 非 0 失败，参照错误码。

➢ 依赖

- 头文件：mi_sys_datatype.h、mi_sys.h
- 库文件：libmi_sys.a / libmi_sys.so

※ 注意

- 需要与 MI_SYS_ClosdFd 成对使用。
- 推荐使用使用 fd + select 的方式取数据，这样 MI_SYS 只有在 fd 对应的 port 口有数据的时候才会唤醒线程，效率比采用 while+sleep 循环取数据的效率要搞。

➢ 举例

参见 MI_SYS_ChnOutputPortGetBuf 调用 Sample 举例。

➢ 相关主题

MI_SYS_CloseFd

2.4.12 MI_SYS_CloseFd

➢ 功能

关闭当前通道的文件描述符

➢ 语法

MI_S32 MI_SYS_CloseFd(MI_S32 s32ChnPortFd);

➢ 形参

参数名称	描述	输入/输出
s32ChnPortFd	等待事件的文件描述符	输入

➢ 返回值

- 0 成功。
- 非 0 失败，参照错误码。

➢ 依赖

- 头文件：mi_sys_datatype.h、mi_sys.h
- 库文件：libmi_sys.a / libmi_sys.so

※ 注意

需要与 MI_SYS_GetFd 成对使用。

➢ 举例

参见 MI_SYS_ChnOutputPortGetBuf 调用 Sample 举例。

➢ 相关主题

MI_SYS_GetFd

2.5. 内存管理类 API

2.5.1 MI_SYS_SetChnMMAConf

➢ 功能

设置模块设备通道输出默认分配内存的 MMA 池名称。

➢ 语法

MI_S32 MI_SYS_SetChnMMAConf (MI_ModuleId_e eModId, MI_U32 u32DevId, MI_U32 u32ChnId, MI_U8 *pu8MMAHeapName);

➢ 形参

参数名称	描述	输入/输出
eModId	待配置的模块 ID	输入
u32DevId	待配置的设备 ID	输入
u32ChnId	带配置的通道号	输入
pu8MMAHeapName	MMA heap name	输入

➢ 返回值

- 0 成功。
- 非 0 失败，参照错误码。

➢ 依赖

- 头文件：mi_sys_datatype.h、mi_sys.h
- 库文件：libmi_sys.a / libmi_sys.so

※ 注意

N/A

➢ 举例

```
MI_ModuleId_e eVifModeId = E_MI_MODULE_ID_VIF;
MI_VIF_DEV vifDev = 0;
MI_VIF_CHN vifChn = 0;

MI_SYS_SetChnMMAConf(eVifModeId, vifDev, vifChn, "mma_heap_name0");
```

➢ 相关主题

N/A

2.5.2 MI_SYS_GetChnMMAConf

➢ 功能

获取模块设备通道输出端口默认分配内存的 MMA 池名称。

➢ 语法

MI_S32 MI_SYS_GetChnMMAConf (MI_ModuleId_e eModId, MI_U32 u32DevId, MI_U32 u32ChnId, void *data, MI_U32 u32Length);

➢ 形参

参数名称	描述	输入/输出
eModId	待配置的模块 ID	输入
u32DevId	待配置的设备 ID	输入
u32ChnId	带配置的通道号	输入
pu8MMAHeapName	MMA heap name	输入

➢ 返回值

- 0 成功。
- 非 0 失败，参照错误码。

➢ 依赖

- 头文件：mi_sys_datatype.h、mi_sys.h
- 库文件：libmi_sys.a / libmi_sys.so

※ 注意

N/A

➢ 举例

N/A

➢ 相关主题

MI_SYS_SetChnMMAConf

2.5.3 MI_SYS_ConfDevPubPools

➢ 功能

配置并初始化模块公共缓冲池。

➢ 语法

MI_S32 MI_SYS_ConfDevPubPools(MI_ModuleId_e eModule, MI_U32 u32DevId, MI_VB_PoolListConf_t stPoolListConf);

➢ 形参

参数名称	描述	输入/输出
eModule	目标模块 ID	输入
u32DevId	Dev ID	输入
stPoolListConf	模块公共缓冲池队列配置	输入

➢ 返回值

- 0 成功。
- 非 0 失败，参照错误码。

➢ 依赖

- 头文件：mi_sys_datatype.h、mi_sys.h
- 库文件：libmi_sys.a / libmi_sys.so

※ 注意

Version 2.0 及以上版本默认不开启，推荐使用 MI_SYS_ConfigPrivateMMAPool。

➢ 举例

N/A

➢ 相关主题

N/A

2.5.4 MI_SYS_ReleaseDevPubPools

➢ 功能

释放全局的公共缓冲池。

➢ 语法

MI_S32 MI_SYS_ReleaseDevPubPools(MI_ModuleId_e eModule, MI_U32 u32DevId);

➢ 形参

参数名称	描述	输入/输出
eModule	目标模块 ID	输入
u32DevId	Dev ID	输入

➢ 返回值
- 0　　成功。
- 非 0　　失败，参照错误码。

➢ 依赖
- 头文件：mi_sys_datatype.h、mi_sys.h
- 库文件：libmi_sys.a / libmi_sys.so

※ 注意

N/A

➢ 举例

Version 2.0 及以上版本默认不开启，推荐使用 MI_SYS_ConfigPrivateMMAPool。

➢ 相关主题

N/A

2.5.5 MI_SYS_ConfGloPubPools

➢ 功能

配置并初始化系统全局公共缓冲池。

➢ 语法

MI_S32 MI_SYS_ConfGloPubPools(MI_VB_PoolListConf_t stPoolListConf);

➢ 形参

参数名称	描述	输入/输出
stPoolListConf	模块公共缓冲池队列配置	输入

➢ 返回值
- 0　　成功。
- 非 0　　失败，参照错误码。

➢ 依赖
- 头文件：mi_sys_datatype.h、mi_sys.h
- 库文件：libmi_sys.a / libmi_sys.so

※ 注意

Version 2.0 及以上版本默认不开启，推荐使用 MI_SYS_ConfigPrivateMMAPool。

➢ 举例

N/A

➢ 相关主题

N/A

2.5.6 MI_SYS_ReleaseGloPubPools

➢ 功能

释放全局的公共缓冲池。

➢ 语法

MI_S32 MI_VB_ ReleaseGloPubPools (void);

➢ 形参

N/A

➢ 返回值

- 0 成功。
- 非 0 失败，参照错误码。

➢ 依赖

- 头文件：mi_sys_datatype.h、mi_sys.h
- 库文件：libmi_sys.a / libmi_sys.so

※ 注意

N/A

➢ 举例

Version 2.0 及以上版本默认不开启，推荐使用 MI_SYS_ConfigPrivateMMAPool。

➢ 相关主题

N/A

2.5.7 MI_SYS_MMA_Alloc

➢ 功能

直接向 MMA 内存管理器申请分配内存。

➢ 语法

MI_S32 MI_SYS_MMA_Alloc(MI_U8 *pstMMAHeapName, MI_U32 u32BlkSize ,MI_PHY *phyAddr);

➢ 形参

参数名称	描述	输入/输出
pstMMAHeapName	目标 MMA heapname	输入
u32BlkSize	待分配的块字节大小	输入
phyAddr	返回的内存块物理地址	输出

➢ 返回值

- 0 成功。
- 非 0 失败，参照错误码。

➢ 依赖
- 头文件：mi_sys_datatype.h、mi_sys.h
- 库文件：libmi_sys.a / libmi_sys.so

※ 注意
N/A

➢ 举例

```
MI_PHY phySrcBufAddr = 0;
void *pVirSrcBufAddr = NULL;
MI_U32 srcBuffSize = 1920 * 1980 * 3 / 2;
srcBuffSize = ALIGN_UP(srcBuffSize, 4096);

ret = MI_SYS_MMA_Alloc(NULL, srcBuffSize, &phySrcBufAddr);
if(ret != MI_SUCCESS)
{
    printf("alloc src buff failed\n");
    return -1;
}

ret = MI_SYS_Mmap(phySrcBufAddr, srcBuffSize, &pVirSrcBufAddr, TRUE);
if(ret != MI_SUCCESS)
{
    MI_SYS_MMA_Free(phySrcBufAddr);
    printf("mmap src buff failed\n");
    return -1;
}

memset(pVirSrcBufAddr, 0, srcBuffSize);
MI_SYS_FlushInvCache(pVirSrcBufAddr,srcBuffSize);
MI_SYS_Munmap(pVirSrcBufAddr, srcBuffSize);
MI_SYS_MMA_Free(phySrcBufAddr);
```

MI_SYS_MMA_Alloc 调用 Sample

➢ 相关主题
MI_SYS_MMA_Free / MI_SYS_Mmap / MI_SYS_FlushCache / MI_SYS_UnMmap

2.5.8 MI_SYS_MMA_Free

➢ 功能
直接向 MMA 内存管理器释放之前分配的内存。

➢ 语法
MI_S32 MI_SYS_MMA_Free(MI_U64 phyAddr);

➢ 形参

参数名称	描述	输入/输出
phyAddr	待释放之内存的物理地址	输入

➢ 返回值

- 0 成功。
- 非 0 失败，参照错误码。

➢ 依赖

- 头文件：mi_sys_datatype.h、mi_sys.h
- 库文件：libmi_sys.a / libmi_sys.so

※ 注意

N/A

➢ 举例

参见 MI_SYS_MMA_Alloc 调用 Sample 举例。

➢ 相关主题

MI_SYS_MMA_Alloc / MI_SYS_Mmap / MI_SYS_FlushCache / MI_SYS_UnMmap

2.5.9 MI_SYS_Mmap

➢ 功能

映射任意物理内存到当前用户态进程的 CPU 虚拟地址空间。

➢ 语法

MI_S32 MI_SYS_Mmap(MI_U64 u64PhyAddr, MI_U32 u32Size , void *pVirtualAddress ， MI_BOOL bCache);

➢ 形参

参数名称	描述	输入/输出
u64PhyAddr	待映射的物理地址	输入
u32Size	待映射的物理地址长度	输入
pVirtualAddress	CPU 虚拟地址指针	输入
bCache	是否 map 成 cache 还是 un-cache	输入

➢ 返回值

- 0 成功。
- 非 0 失败，参照错误码。

➢ 依赖

- 头文件：mi_sys_datatype.h、mi_sys.h
- 库文件：libmi_sys.a / libmi_sys.so

※ 注意
- 物理地址必须 4KByte 对齐。
- 物理地址是 SStar 内存控制器地址， 非 CPU bridge 之地址。
- 物理地址长度必须 4KByte 对齐。
- 物理内存必须完整的落在 MMA 管理的内存范围内或者 linux kenrel 管理的内存之外。

➢ 举例
参见 MI_SYS_MMA_Alloc 调用 Sample 举例。

➢ 相关主题
MI_SYS_MMA_Alloc / MI_SYS_MMA_Free / MI_SYS_FlushCache / MI_SYS_UnMmap

2.5.10 MI_SYS_FlushInvCache

➢ 功能
Flush cache。

➢ 语法
MI_S32 MI_SYS_FlushCache(MI_VOID *pVirtualAddress, MI_U32 u32Size);

➢ 形参

参数名称	描述	输入/输出
pVirtualAddress	之前 MI_SYS_Mmap 返回的 CPU 虚拟地址	输入
u32Size	待 flush 的 cache 长度	输入

➢ 返回值
- 0 成功。
- 非 0 失败，参照错误码。

➢ 依赖
- 头文件：mi_sys_datatype.h、mi_sys.h
- 库文件：libmi_sys.a / libmi_sys.so

※ 注意
- 待 flush cache 的物理地址必须 4KByte 对齐。
- 待 flush cache 的映射长度必须 4KByte 对齐。
- 待 flush cache 的映射内存范围必须是之前通过 MI_SYS_Mmap API 获取到的。
- 待 flush cache 的映射内存应该是 MI_SYS_Mmap 时采用 cache 方式进行的，nocache 的方式无需 flush cache。

➢ 举例
参见 MI_SYS_MMA_Alloc 调用 Sample 举例。

➢ 相关主题
MI_SYS_MMA_Alloc / MI_SYS_MMA_Free / MI_SYS_Mmap / MI_SYS_UnMmap

2.5.11 MI_SYS_Munmap

➢ 功能

取消物理内存到虚拟地址的映射。

➢ 语法

MI_S32 MI_SYS_UnMmap(MI_VOID *pVirtualAddress, MI_U32 u32Size);

➢ 形参

参数名称	描述	输入/输出
pVirtualAddress	之前 MI_SYS_Mmap 返回的 CPU 虚拟地址	输入
u32Size	待取消映射的映射长度	输入

➢ 返回值

- 0 成功。
- 非 0 失败，参照错误码。

➢ 依赖

- 头文件：mi_sys_datatype.h、mi_sys.h
- 库文件：libmi_sys.a / libmi_sys.so

※ 注意

- 待取消映射的虚拟地址必须 4KByte 对齐。
- 待取消映射的映射长度必须 4KByte 对齐
- 待取消的映射内存范围必须是之前通过 MI_SYS_Mmap API 获取到的。

➢ 举例

参见 MI_SYS_MMA_Alloc 调用 Sample 举例。

➢ 相关主题

MI_SYS_MMA_Alloc / MI_SYS_MMA_Free / MI_SYS_Mmap / MI_SYS_FlushCache

2.5.12 MI_SYS_MemsetPa

➢ 功能

通过 DMA 硬件模块，填充整块物理内存.

➢ 语法

MI_S32 MI_SYS_MemsetPa(MI_PHY phyPa, MI_U32 u32Val, MI_U32 u32Lenth);

➢ 形参

参数名称	参数含义	输入/输出
phyPa	填充的物理地址	输入
u32Val	填充值	输入
u32Lenth	填充大小，单位为 byte	输入

➢ 返回值
- 0　　成功。
- 非 0　　失败，参照错误码。

➢ 依赖
- 头文件：mi_sys_datatype.h、mi_sys.h
- 库文件：libmi_sys.a / libmi_sys.so

※ 注意
N/A

➢ 举例

```
MI_PHY phySrcBufAddr = 0;
MI_PHY phyDstBufAddr = 0;
void *pVirSrcBufAddr = NULL;
void *pVirDstBufAddr = NULL;
MI_U32 buffSize = 1920 * 1980 * 3 / 2;
buffSize = ALIGN_UP(buffSize, 4096);

ret = MI_SYS_MMA_Alloc(NULL, buffSize, &phySrcBufAddr);
if(ret != MI_SUCCESS)
{
    printf("alloc src buff failed\n");
    return -1;
}

ret = MI_SYS_MMA_Alloc(NULL, buffSize, &phyDstBufAddr);
if(ret != MI_SUCCESS)
{
    MI_SYS_MMA_Free(phySrcBufAddr);
    printf("alloc dts buff failed\n");
    return -1;
}

MI_SYS_MemsetPa(phySrcBufAddr, 0xff, buffSize);
MI_SYS_MemsetPa(phyDstBufAddr, 0x00, buffSize);
MI_SYS_MemcpyPa(phyDstBufAddr, phySrcBufAddr, buffSize);
MI_SYS_MMA_Free(phySrcBufAddr);
MI_SYS_MMA_Free(phyDstBufAddr);
```

MI_SYS_MemsetPa 调用 Sample

➢ 相关主题
MI_SYS_MemcpyPa

2.5.13 MI_SYS_MemcpyPa

➢ 功能

通过 DMA 硬件模块，把源内存数据拷贝到目标内存上.

➢ 语法

MI_S32 MI_SYS_MemcpyPa(MI_PHY phyDst, MI_PHY phySrc, MI_U32 u32Lenth);

➢ 形参

参数名称	参数含义	输入/输出
phyDst	目的物理地址	输入
phySrc	源物理地址	输入
u32Lenth	拷贝大小，单位为 byte	输入

➢ 返回值

- 0 成功。
- 非 0 失败，参照错误码。

➢ 依赖

- 头文件：mi_sys_datatype.h、mi_sys.h
- 库文件：libmi_sys.a / libmi_sys.so

※ 注意

N/A

➢ 举例

参见 MI_SYS_MemsetPa 举例。

➢ 相关主题

MI_SYS_MemsetPa

2.5.14 MI_SYS_BufFillPa

➢ 功能

通过 DMA 硬件模块，填充部分物理内存.

➢ 语法

MI_S32 MI_SYS_BufFillPa(MI_SYS_FrameData_t *pstBuf, MI_U32 u32Val, MI_SYS_WindowRect_t *pstRect);

➢ 形参

参数名称	参数含义	输入/输出
pstBuf	填充的帧数据描述之结构体	输入
u32Val	填充值	输入
pstRect	填充的数据范围	输入

➢ 返回值
- 0 成功。
- 非 0 失败，参照错误码。

➢ 依赖
- 头文件：mi_sys_datatype.h、mi_sys.h
- 库文件：libmi_sys.a / libmi_sys.so

※ 注意
- pstRect 数据范围是以 pstBuf 所描述的首地址为（0，0）点开始，宽高都是以 pstBuf 中的 ePixelFormat 来计算每一个 pixel 所移动的内存数据大小来进行部分填充。
- pstBuf 中 u16Width、u16Height、phyAddr、u32Stride、ePixelFormat 为必填，其余数值无意义。

➢ 举例

```
MI_S32 ret = 0;
MI_SYS_WindowRect_t rect;
MI_SYS_FrameData_t stSysFrame;
memcpy(&stSysFrame, &buf->stFrameData, sizeof(MI_SYS_FrameData_t));

if(pRect) {
    rect.u16X = pRect->left;
    rect.u16Y = pRect->top;
    rect.u16Height = pRect->bottom-pRect->top;
    rect.u16Width = pRect->right-pRect->left;
} else {
    rect.u16X = 0;
    rect.u16Y = 0;
    rect.u16Height = stSysFrame.u16Height;
    rect.u16Width = stSysFrame.u16Width;

}

DBG_INFO("rect %d %d %d %d \n", rect.u16X, rect.u16Y
        , rect.u16Width, rect.u16Height);
ret = MI_SYS_BufBlitPa(&stSysFrame, u32ColorVal, &rect);

return ret;
```

➢ 相关主题
MI_SYS_BufBlitPa

2.5.15 MI_SYS_BufBlitPa

➢ 功能
通过 DMA 硬件模块，把源内存数据上的部分区域拷贝到目标内存上的部分区域.

➢ 语法

MI_S32 MI_SYS_BufBlitPa(MI_SYS_FrameData_t *pstDstBuf, MI_SYS_WindowRect_t *pstDstRect, MI_SYS_FrameData_t *pstSrcBuf, MI_SYS_WindowRect_t *pstSrcRect);

➢ 形参

参数名称	参数含义	输入/输出
pstDstBuf	目标内存物理首地址	输入
pstDstRect	目标内存拷贝的区域	输入
pstSrcBuf	源内存物理首地址	输入
pstSrcRect	源内存拷贝的区域	输入

➢ 返回值

- 0　　成功。
- 非 0　　失败，参照错误码。

➢ 依赖

- 头文件：mi_sys_datatype.h、mi_sys.h
- 库文件：libmi_sys.a / libmi_sys.so

※ 注意

- pstDstRect/pstSrcRect 数据范围是以 pstDstBuf/pstSrcBuf 所描述的首地址为（0，0）点开始，宽高都是以其中的 ePixelFormat 来计算每一个 pixel 所移动的内存数据大小来进行部分填充。
- pstDstBuf/pstSrcBuf 中 u16Width、u16Height、phyAddr、u32Stride、ePixelFormat 为必填，其余数值无意义。
- 源内存或者目标内存的区域部分超过了其本来的范围，则只会拷贝其未超过部分上的数据。

➢ 举例

```
MI_S32 ret = MI_SUCCESS;
vdisp_copyinfo_plane_t *plane;
MI_SYS_FrameData_t stSrcFrame, stDstFrame;
MI_SYS_WindowRect_t stSrcRect, stDstRect;

plane = &copyinfo->plane[0];
stSrcFrame.ePixelFormat = E_MI_SYS_PIXEL_FRAME_I8;
stSrcFrame.phyAddr[0] = plane->src_paddr;
stSrcFrame.u16Width =  plane->width;
stSrcFrame.u16Height = plane->height;
stSrcFrame.u32Stride[0] = plane->src_stride;

stDstFrame.ePixelFormat = E_MI_SYS_PIXEL_FRAME_I8;
stDstFrame.phyAddr[0] = plane->dst_paddr;
stDstFrame.u16Width =  plane->width;
stDstFrame.u16Height = plane->height;
stDstFrame.u32Stride[0] = plane->dst_stride;
```

```
stSrcRect.u16X = 0;
stSrcRect.u16Y = 0;
stSrcRect.u16Height = stSrcFrame.u16Height;
stSrcRect.u16Width = stSrcFrame.u16Width;

stDstRect.u16X = 0;
stDstRect.u16Y = 0;
stDstRect.u16Height = stDstFrame.u16Height;
stDstRect.u16Width = stDstFrame.u16Width;

ret = MI_SYS_BufBlitPa(&stDstFrame, &stDstRect, &stSrcFrame, &stSrcRect);

return ret;
```

➢ 相关主题

MI_SYS_BufFillPa

2.5.16 MI_SYS_ConfigPrivateMMAPool

➢ 功能

为模块配置私有 MMA Heap.

➢ 语法

MI_S32 MI_SYS_ConfigPrivateMMAPool(MI_SYS_GlobalPrivPoolConfig_t *pstGlobalPrivPoolConf);

➢ 形参

参数名称	参数含义	输入/输出
pstGlobalPrivPoolConf	配置私有 Mma Pool 结构体指针	输入

➢ 返回值

- 0 成功。
- 非 0 失败，参照错误码。

➢ 依赖

- 头文件：mi_sys_datatype.h、mi_sys.h
- 库文件：libmi_sys.a / libmi_sys.so

※ 注意

- **设备私有 MMA Heap 与通道私有 MMA Heap 不能共存。**
- 建议在 MI_SYS_Init 后就为各模块创建好私有 MMA heap。
- 当 pstGlobalPrivPoolConf->bCreate 为 TRUE 时，创建私有 POOL，为 FALSE 时，销毁私有 POOL
- 各种 eConfigType 使用场景如下：

eConfigType	适用场景
E_MI_SYS_VPE_TO_VENC_PRIVATE_RING_POOL	为绑定为 E_MI_SYS_BIND_TYPE_HW_RIN 模式的 vpe 与 venc 端口设置私有 ring heap pool
E_MI_SYS_PRE_CHN_PRIVATE_POOL	为模块通道设置私有 heap pool
E_MI_SYS_PRE_DEV_PRIVATE_POOL	为模块设备设置私有 heap pool
E_MI_SYS_PER_CHN_PORT_OUTPUT_POOL	为模块 output port 设置 heap pool，设置后该 port 口的 ouput buf 优先从其中分配

➢ 举例

假设场景为：IPC 下两路流，一路 Main Stream H265 最大分辨率 1920*1080,一路 Sub Stream H264 最大分辨率 720*576。

VENC 创建 MMA Heap:

```
//Main Sream:
//为通道 1 创建大小为 38745760 的私有 MMA heap
MI_SYS_GlobalPrivPoolConfig_t stConfig;
memset(&stConfig , 0 ,sizeof(stConfig));
stConfig.eConfigType = E_MI_SYS_PER_CHN_PRIVATE_POOL;
stConfig.bCreate = TRUE;
stConfig.uConfig.stPreChnPrivPoolConfig.eModule = E_MI_MODULE_ID_VENC;
stConfig.uConfig.stPreChnPrivPoolConfig.u32Devid =  0;
stConfig.uConfig.stPreChnPrivPoolConfig.u32Channel= 1;
stConfig.uConfig.stPreChnPrivPoolConfig.u32PrivateHeapSize = 38745760;
MI_SYS_ConfigDevChnPrivateMMAHeap(&stConfig);

//Sub Stream:
//为通道 2 创建大小为 805152 的私有 MMA Heap
stConfig.eConfigType = E_MI_SYS_PER_CHN_PRIVATE_POOL;
stConfig.bCreate = TRUE;
stConfig.uConfig.stPreChnPrivPoolConfig.eModule = E_MI_MODULE_ID_VENC;
stConfig.uConfig.stPreChnPrivPoolConfig.u32Devid =  0;
stConfig.uConfig.stPreChnPrivPoolConfig.u32Channel= 2;
stConfig.uConfig.stPreChnPrivPoolConfig.u32PrivateHeapSize = 805152;
MI_SYS_ConfigDevChnPrivateMMAHeap(&stConfig);
```

VENC 销毁 MMA Heap:

```
// Main Sream:
//销毁通道 1 的私有 MMA heap
stConfig.eConfigType = E_MI_SYS_PER_CHN_PRIVATE_POOL;
stConfig.bCreate = FALSE;
stConfig.uConfig.stPreChnPrivPoolConfig.eModule = E_MI_MODULE_ID_VENC;
stConfig.uConfig.stPreChnPrivPoolConfig.u32Devid =  0;
stConfig.uConfig.stPreChnPrivPoolConfig.u32Channel= 1;
MI_SYS_ConfigDevChnPrivateMMAHeap(&stConfig);
```

```
//Sub Stream:
//销毁通道 2 的私有 MMA Heap
stConfig.eConfigType = E_MI_SYS_PER_CHN_PRIVATE_POOL;
stConfig.bCreate = FALSE;
stConfig.uConfig.stPreChnPrivPoolConfig.eModule = E_MI_MODULE_ID_VENC;
stConfig.uConfig.stPreChnPrivPoolConfig.u32Devid =  0;
stConfig.uConfig.stPreChnPrivPoolConfig.u32Channel= 2;
MI_SYS_ConfigDevChnPrivateMMAHeap(&stConfig);
```

VPE 创建 MMA Heap:

```
//为设备 0 创建大小为 0x4f9200 的私有 MMA Heap
stConfig.eConfigType = E_MI_SYS_PER_DEV_PRIVATE_POOL;
stConfig.bCreate = TRUE;
stConfig.uConfig.stPreDevPrivPoolConfig.eModule = E_MI_MODULE_ID_VPE;
stConfig.uConfig.stPreDevPrivPoolConfig.u32Devid =  0;
MI_SYS_ConfigDevChnPrivateMMAHeap(&stConfig);
```

VPE 销毁 MMA Heap:

```
//销毁设备 0 的私有 MMA Heap
stConfig.eConfigType = E_MI_SYS_PER_DEV_PRIVATE_POOL;
stConfig.bCreate = FALSE;
stConfig.uConfig.stPreDevPrivPoolConfig.eModule = E_MI_MODULE_ID_VPE;
stConfig.uConfig.stPreChnPrivPoolConfig.u32Devid =  0;
MI_SYS_ConfigDevChnPrivateMMAHeap(&stConfig);
```

创建 VPE 与 VENC 之间的 ring MMA Heap：

```
stConfig.eConfigType = E_MI_SYS_VPE_TO_VENC_PRIVATE_RING_POOL;
stConfig.bCreate = TRUE;
stConfig.uConfig. stPreVpe2VencRingPrivPoolConfig.u32VencInputRingPoolStaticSize =  8*1024*1024;
MI_SYS_ConfigDevChnPrivateMMAHeap(&stConfig);
```

销毁 VPE 与 VENC 之间的 ring MMA Heap：

```
stConfig.eConfigType = E_MI_SYS_VPE_TO_VENC_PRIVATE_RING_POOL;
stConfig.bCreate = FALSE;
MI_SYS_ConfigDevChnPrivateMMAHeap(&stConfig);
```

不同分辨率或开启不同功能 size 大小都不一样，详细参数请根据使用的场景和规格计算。

- 相关主题
  MI_SYS_PrivateDevChnHeapAlloc / MI_SYS_PrivateDevChnHeapFree

2.5.17 MI_SYS_PrivateDevChnHeapAlloc

➢ 功能

从模块通道私有 MMA Pool 申请内存.

➢ 语法

MI_S32 MI_SYS_PrivateDevChnHeapAlloc(MI_ModuleId_e eModule, MI_U32 u32Devid, MI_S32 s32ChnId, MI_U8 *pu8BufName, MI_U32 u32blkSize, MI_PHY *pphyAddr, MI_BOOL bTailAlloc);

➢ 形参

参数名称	参数含义	输入/输出
eModule	模块 ID	输入
u32Devid	模块的设备 ID	输入
s32ChnId	模块的通道 ID	输入
pu8BufName	申请私有内存的名字，传 Null 时，使用"app-privAlloc"作为默认的名字	输入
u32blkSize	申请私有内存的大小	输入
pphyAddr	分配出的私有内存的物理地址	输出
bTailAlloc	是否从通道私有 pool 的尾部申请 0-否，1-是	输入

➢ 返回值

- 0 成功。
- 非 0 失败，参照错误码。

➢ 依赖

- 头文件：mi_sys_datatype.h、mi_sys.h
- 库文件：libmi_sys.a / libmi_sys.so

※ 注意

使用该接口时，请确定已经先使用 MI_SYS_ConfigPrivateMMAPool 申请了 E_MI_SYS_PRE_CHN_PRIVATE_POOL 类型的私有内存池。

➢ 举例

```
MI_PHY *pphyAddr = NULL;
MI_SYS_GlobalPrivPoolConfig_t stConfig;
Memset(&stConfig , 0 ,sizeof(stConfig));
stConfig.eConfigType = E_MI_SYS_PER_CHN_PRIVATE_POOL;
stConfig.bCreate = TRUE;
stConfig.uConfig.stPreChnPrivPoolConfig. eModule = E_MI_MODULE_ID_VENC;
stConfig.uConfig.stPreChnPrivPoolConfig. u32Devid =  0;
stConfig.uConfig.stPreChnPrivPoolConfig. u32Channel= 1;
stConfig.uConfig.stPreChnPrivPoolConfig. u32PrivateHeapSize = 38745760;
MI_SYS_ConfigDevChnPrivateMMAHeap(&stConfig);
```

```
ret = MI_SYS_PrivateDevChnHeapAlloc(E_MI_MODULE_ID_VENC, 0, 1, NULL, 4096, pphyAddr, FALSE);
if(ret != MI_SUCCESS)
{
    MI_SYS_ConfigDevChnPrivateMMAHeap(&stConfig);
    printf("alloc buff from chn private heap failed\n");
    return -1;
}

//do something...

MI_SYS_PrivateDevChnHeapFree(E_MI_MODULE_ID_VENC, 0, 1, *pphyAddr);
MI_SYS_ConfigDevChnPrivateMMAHeap(&stConfig);
```

MI_SYS_PrivateDevChnHeapAlloc 调用 Sample

- 相关主题
  MI_SYS_ConfigPrivateMMAPool / MI_SYS_PrivateDevChnHeapFree

2.5.18 MI_SYS_PrivateDevChnHeapFree

- 功能
  从模块通道私有 MMA Pool 释放内存.

- 语法
  MI_S32 MI_SYS_PrivateDevChnHeapFree(MI_ModuleId_e eModule, MI_U32 u32Devid, MI_S32 s32ChnId, MI_PHY phyAddr);

- 形参

参数名称	参数含义	输入/输出
eModule	模块 ID	输入
u32Devid	模块的设备 ID	输入
s32ChnId	模块的通道 ID	输入
phyAddr	需要释放的内存对应的物理地址	输入

- 返回值
  - 0 成功。
  - 非 0 失败，参照错误码。

- 依赖
  - 头文件：mi_sys_datatype.h、mi_sys.h
  - 库文件：libmi_sys.a / libmi_sys.so

※ 注意
使用该接口时，请确定已经先使用 MI_SYS_ConfigPrivateMMAPool 申请了 E_MI_SYS_PRE_CHN_PRIVATE_POOL 类型的私有内存池。

➢ 举例
参见 MI_SYS_PrivateDevChnHeapAlloc 调用 Sample 举例。

➢ 相关主题
MI_SYS_ConfigPrivateMMAPool / MI_SYS_PrivateDevChnHeapAlloc

3. 数据类型

3.1. 数据结构描述格式说明

本手册使用 5 个参考域描述数据类型的相关信息，它们作用如 个参考域描述数据类型的相关信息，它们作用如表 3-1 所示。

标签	功能
说明	简要描述数据类型的主要功能。
定义	列出数据类型的定义语句
成员	列出数据结构的成员及含义。
注意事项	列出使用数据类型时应注意的事项。
相关数据类型及接口	列出与本数据类型相关联的其他数据类型和接口。

表 3-1 数据结构描述格式说明

3.2. 数据结构列表

相关数据类型、数据结构定义如下：

MI_ModuleId_e	定义模块 ID 枚举类型
MI_SYS_PixelFormat_e	定义像素枚举类型
MI_SYS_CompressMode_e	定义压缩方式枚举类型
MI_SYS_FrameTileMode_e	定义 Tile 格式枚举类型
MI_SYS_FieldType_e	定义 Field 枚举类型
MI_SYS_BufDataType_e	定义模块 ID 枚举类型
MI_SYS_ChnPort_t	定义模块设备通道结构体
MI_SYS_MetaData_t	定义码流 MetaData 的结构体
MI_SYS_RawData_t	定义码流 RawData 的结构体
MI_SYS_WindowRect_t	定义 Window 坐标的结构体
MI_SYS_FrameData_t	定义码流 FrameData 的结构体
MI_SYS_BufInfo_t	Buf 信息结构体
MI_SYS_BufFrameConfig_t	Frame buf 配置信息结构体
MI_SYS_BufRawConfig_t	Raw buf 配置信息结构体
MI_SYS_BufConf_t	配置 Buf 信息结构体
MI_SYS_Version_t	Sys 版本信息结构体
MI_VB_PoolListConf_t	描述 VB Pool 链表信息的结构体
MI_SYS_BindType_e	定义前后级工作模式的枚举类型
MI_SYS_FrameData_PhySignalType	描述 frame data 隶属的 buffer 类型的枚举类型
MI_SYS_MetaDataConfig_t	Meta buf 配置信息结构体

表 3-2 数据结构汇总

3.2.1 MI_ModuleId_e

- 说明

  定义模块 ID 枚举类型。

- 定义

```
typedef enum
{
    E_MI_MODULE_ID_IVE    = 0,
    E_MI_MODULE_ID_VDF    = 1,
    E_MI_MODULE_ID_VENC   = 2,
    E_MI_MODULE_ID_RGN    = 3,
    E_MI_MODULE_ID_AI     = 4,
    E_MI_MODULE_ID_AO     = 5,
    E_MI_MODULE_ID_VIF    = 6,
    E_MI_MODULE_ID_VPE    = 7,
    E_MI_MODULE_ID_VDEC   = 8,
    E_MI_MODULE_ID_SYS    = 9,
    E_MI_MODULE_ID_FB     = 10,
    E_MI_MODULE_ID_HDMI   = 11,
    E_MI_MODULE_ID_DIVP   = 12,
    E_MI_MODULE_ID_GFX    = 13,
    E_MI_MODULE_ID_VDISP  = 14,
    E_MI_MODULE_ID_DISP   = 15,
    E_MI_MODULE_ID_OS     = 16,
    E_MI_MODULE_ID_IAE = 17,
    E_MI_MODULE_ID_MD = 18,
    E_MI_MODULE_ID_OD = 19,
    E_MI_MODULE_ID_SHADOW = 20,
    E_MI_MODULE_ID_WARP = 21,
    E_MI_MODULE_ID_UAC = 22,
    E_MI_MODULE_ID_LDC = 23,
    E_MI_MODULE_ID_SD = 24,
    E_MI_MODULE_ID_PANEL = 25,
    E_MI_MODULE_ID_CIPHER = 26,
    E_MI_MODULE_ID_SNR = 27,
    E_MI_MODULE_ID_WLAN =28,
    E_MI_MODULE_ID_IPU = 29,
    E_MI_MODULE_ID_MIPITX = 30,
    //E_MI_MODULE_ID_SED  = 29,
    E_MI_MODULE_ID_MAX,
} MI_ModuleId_e;
```

➢ 成员

模块 ID	模块 ID HEX 值	成员名称	模块
0	0x00	E_MI_MODULE_ID_IVE	图像智能算子 IVE 的模块 ID
1	0x01	E_MI_MODULE_ID_VDF	视频智能算法框架模块 VDF 的模块 ID
2	0x02	E_MI_MODULE_ID_VENC	视频编码模块 VPE 的模块 ID
3	0x03	E_MI_MODULE_ID_RGN	OSD 叠加和遮挡模块 REG 的模块 ID
4	0x04	E_MI_MODULE_ID_AI	音频输入模块 AI 的模块 ID
5	0x05	E_MI_MODULE_ID_AO	音频输出模块 AO 的模块 ID
6	0x06	E_MI_MODULE_ID_VIF	视频输入 VIF 的模块 ID
7	0x07	E_MI_MODULE_ID_VPE	视频图像处理模块 VPE 的模块 ID
8	0x08	E_MI_MODULE_ID_VDEC	视频处理 VPE 的模块 ID
9	0x09	E_MI_MODULE_ID_SYS	系统模块 SYS 的模块 ID
10	0x0A	E_MI_MODULE_ID_FB	SStar UI 显示 FrameBuffer Device 模块的模块 ID
11	0x0B	E_MI_MODULE_ID_HDMI	HMDI 模块 ID
12	0x0C	E_MI_MODULE_ID_DIVP	视频 DI 及后处理模块 DIVP 的模块 ID
13	0x0D	E_MI_MODULE_ID_GFX	2D 图形处理加速模块 GFX 的模块 ID
14	0x0E	E_MI_MODULE_ID_VDISP	图像拼图模块 VDISP 的模块 ID
15	0x0F	E_MI_MODULE_ID_DISP	视频显示模块 DISP 的模块 ID
16	0x10	E_MI_MODULE_ID_OS	RTOS 系统模块 ID
17	0x11	E_MI_MODULE_ID_IAE	音频智能算子 IAE 的模块 ID
18	0x12	E_MI_MODULE_ID_MD	移动侦测模块 MD 的模块 ID
19	0x13	E_MI_MODULE_ID_OD	遮挡侦测模块 OD 的模块 ID
20	0x14	E_MI_MODULE_ID_SHADOW	虚拟 MI 框架 SHADOW 的模块 ID
21	0x15	E_MI_MODULE_ID_WARP	图像畸形校正算法 WARP 的模块 ID
22	0x16	E_MI_MODULE_ID_UAC	USB Aduio Class 专用 ALSA 设备的模块 ID
23	0x17	E_MI_MODULE_ID_LDC	图像畸形校正算法 LDC 的模块 ID
24	0x18	E_MI_MODULE_ID_SD	图像缩放功能的模块 ID
25	0x19	E_MI_MODULE_ID_PANEL	屏显示 PANEL 的模块 ID
26	0x1A	E_MI_MODULE_ID_CIPHER	芯片加密 CIPHER 的模块 ID
27	0x1B	E_MI_MODULE_ID_SNR	Sensor 传感器模块 ID
28	0x1C	E_MI_MODULE_ID_WLAN	无线通信网络 WLAN 的模块 ID
29	0x1D	E_MI_MODULE_ID_IPU	图像识别处理 IPU 的模块 ID
30	0x1E	E_MI_MODULE_ID_MIPITX	使用MIPI协议发送数据模块ID

※ 注意事项

N/A

➢ 相关数据接口及类型

N/A

3.2.2 MI_SYS_PixelFormat_e

➢ 说明

定义像素枚举类型。

➢ 定义

```
typedef enum
{
    E_MI_SYS_PIXEL_FRAME_YUV422_YUYV = 0,
    E_MI_SYS_PIXEL_FRAME_ARGB8888,
    E_MI_SYS_PIXEL_FRAME_ABGR8888,
    E_MI_SYS_PIXEL_FRAME_BGRA8888,

    E_MI_SYS_PIXEL_FRAME_RGB565,
    E_MI_SYS_PIXEL_FRAME_ARGB1555,
    E_MI_SYS_PIXEL_FRAME_ARGB4444,
    E_MI_SYS_PIXEL_FRAME_I2,
    E_MI_SYS_PIXEL_FRAME_I4,
    E_MI_SYS_PIXEL_FRAME_I8,

    E_MI_SYS_PIXEL_FRAME_YUV_SEMIPLANAR_422,
    E_MI_SYS_PIXEL_FRAME_YUV_SEMIPLANAR_420,
    E_MI_SYS_PIXEL_FRAME_YUV_MST_420,
    E_MI_SYS_PIXEL_FRAME_YUV422_UYVY,
    E_MI_SYS_PIXEL_FRAME_YUV422_YVYU,
    E_MI_SYS_PIXEL_FRAME_YUV422_VYUY,

    //vdec sigmastar private video format
    E_MI_SYS_PIXEL_FRAME_YC420_MSTTILE1_H264,
    E_MI_SYS_PIXEL_FRAME_YC420_MSTTILE2_H265,
    E_MI_SYS_PIXEL_FRAME_YC420_MSTTILE3_H265,

    E_MI_SYS_PIXEL_FRAME_RGB_BAYER_BASE,
    E_MI_SYS_PIXEL_FRAME_RGB_BAYER_NUM = E_MI_SYS_PIXEL_FRAME_RGB_BAYER_BASE + E_MI_SYS_D
ATA_PRECISION_MAX*E_MI_SYS_PIXEL_BAYERID_MAX-1,

    E_MI_SYS_PIXEL_FRAME_RGB888,
    E_MI_SYS_PIXEL_FRAME_BGR888,

    E_MI_SYS_PIXEL_FRAME_FORMAT_MAX,
} MI_SYS_PixelFormat_e;
```

➢ 成员

成员名称	描述
E_MI_SYS_PIXEL_FRAME_YUV422_YUYV	YUV422_YUYV 格式
E_MI_SYS_PIXEL_FRAME_ARGB8888	ARGB8888 格式
E_MI_SYS_PIXEL_FRAME_ABGR8888	ABGR8888 格式
E_MI_SYS_PIXEL_FRAME_BGRA8888	BGRA8888 格式
E_MI_SYS_PIXEL_FRAME_RGB565	RGB565 格式
E_MI_SYS_PIXEL_FRAME_ARGB1555	ARGB1555 格式
E_MI_SYS_PIXEL_FRAME_ARGB4444	ARGB4444 格式
E_MI_SYS_PIXEL_FRAME_I2	2 bpp 格式
E_MI_SYS_PIXEL_FRAME_I4	4 bpp 格式
E_MI_SYS_PIXEL_FRAME_I8	8 bpp 格式
E_MI_SYS_PIXEL_FRAME_YUV_SEMIPLANAR_422	YUV422 semi-planar 格式
E_MI_SYS_PIXEL_FRAME_YUV_SEMIPLANAR_420	YUV420 格式
E_MI_SYS_PIXEL_FRAME_YUV_MST_420	YUV420 packe（YYUYYV）格式
E_MI_SYS_PIXEL_FRAME_YC420_MSTTILE1_H264	内部自定义 TILE 格式
E_MI_SYS_PIXEL_FRAME_YC420_MSTTILE2_H265	内部自定义 TILE 格式
E_MI_SYS_PIXEL_FRAME_YC420_MSTTILE3_H265	内部自定义 TILE 格式
E_MI_SYS_PIXEL_FRAME_RGB_BAYER_BASE	RGB raw data 组合格式

※ 注意事项

N/A

➢ 相关数据类型及接口

N/A

3.2.3 MI_SYS_CompressMode_e

➢ 说明

定义压缩方式枚举类型。

➢ 定义

```
typedef enum
{
    E_MI_SYS_COMPRESS_MODE_NONE,//no compress
    E_MI_SYS_COMPRESS_MODE_SEG,//compress unit is 256 bytes as a segment
    E_MI_SYS_COMPRESS_MODE_LINE,//compress unit is the whole line
    E_MI_SYS_COMPRESS_MODE_FRAME,//compress unit is the whole frame
    E_MI_SYS_COMPRESS_MODE_BUTT, //number
}MI_SYS_CompressMode_e;
```

➢ 成员

成员名称	描述
E_MI_SYS_COMPRESS_MODE_NONE	非压缩的视频格式
E_MI_SYS_COMPRESS_MODE_SEG	段压缩的视频格式
E_MI_SYS_COMPRESS_MODE_LINE	行压缩的视频格式，按照一行为一段进行压缩
E_MI_SYS_COMPRESS_MODE_FRAME	帧压缩的视频格式，将一帧数据进行压缩
E_MI_SYS_COMPRESS_MODE_BUTT	视频压缩方式的数量

※ 注意事项

N/A

➢ 相关数据类型及接口

N/A

3.2.4 MI_SYS_FrameTileMode_e

➢ 说明

定义 Tile 格式枚举类型。

➢ 定义

```
typedef enum
{
    E_MI_SYS_FRAME_TILE_MODE_NONE = 0,
    E_MI_SYS_FRAME_TILE_MODE_16x16,     // tile mode 16x16
    E_MI_SYS_FRAME_TILE_MODE_16x32,     // tile mode 16x32
    E_MI_SYS_FRAME_TILE_MODE_32x16,     // tile mode 32x16
    E_MI_SYS_FRAME_TILE_MODE_32x32,     // tile mode 32x32
    E_MI_SYS_FRAME_TILE_MODE_MAX
} MI_SYS_FrameTileMode_e;
```

➢ 成员

成员名称	描述
E_MI_SYS_FRAME_TILE_MODE_NONE	None
E_MI_SYS_FRAME_TILE_MODE_16x16	16x16 mode
E_MI_SYS_FRAME_TILE_MODE_16x32	16x32 mode
E_MI_SYS_FRAME_TILE_MODE_32x16	32x16 mode
E_MI_SYS_FRAME_TILE_MODE_32x32	32x32 mode

※ 注意事项

N/A

➢ 相关数据类型及接口

N/A

3.2.5 MI_SYS_FieldType_e

➢ 说明

定义枚举类型。

➢ 定义

```
typedef enum
{
    E_MI_SYS_FIELDTYPE_NONE,      //< no field.
    E_MI_SYS_FIELDTYPE_TOP,       //< Top field only.
    E_MI_SYS_FIELDTYPE_BOTTOM,    //< Bottom field only.
    E_MI_SYS_FIELDTYPE_BOTH,      //< Both fields.
    E_MI_SYS_FIELDTYPE_NUM
} MI_SYS_FieldType_e;
```

➢ 成员

成员名称	描述
E_MI_SYS_FIELDTYPE_NONE	None
E_MI_SYS_FIELDTYPE_TOP	Top field only
E_MI_SYS_FIELDTYPE_BOTTOM	Bottom field only
E_MI_SYS_FIELDTYPE_BOTH	Both fields

※ 注意事项

N/A

➢ 相关数据类型及接口

N/A

3.2.6 MI_SYS_BufDataType_e

➢ 说明

定义模块 ID 枚举类型。

➢ 定义

```
typedef enum
{
    E_MI_SYS_BUFDATA_RAW = 0,
    E_MI_SYS_BUFDATA_FRAME,
    E_MI_SYS_BUFDATA_META,
} MI_SYS_BufDataType_e;
```

➢ 成员

成员名称	描述
E_MI_SYS_BUFDATA_RAW	Raw 数据类型
E_MI_SYS_BUFDATA_FRAME	Frame 数据类型
E_MI_SYS_BUFDATA_META	Meta 数据类型

※ 注意事项

N/A

➢ 相关数据类型及接口

N/A

3.2.7 MI_SYS_FrameIspInfoType_e

➢ 说明

定义 frame data 携带的 ISP info 枚举类型。

➢ 定义

```
typedef enum
{
    E_MI_SYS_FRAME_ISP_INFO_TYPE_NONE,
    E_MI_SYS_FRAME_ISP_INFO_TYPE_GLOBAL_GRADIENT
}MI_SYS_FrameIspInfoType_e;
```

➢ 成员

成员名称	描述
E_MI_SYS_FRAME_ISP_INFO_TYPE_NONE	NONE
E_MI_SYS_FRAME_ISP_INFO_TYPE_GLOBAL_GRADIENT	ISP 的全局梯度数据类型

※ 注意事项

N/A

➢ 相关数据类型及接口

N/A

3.2.8 MI_SYS_ChnPort_t

➢ 说明

定义模块设备通道结构体。

➢ 定义

```
typedef struct MI_SYS_ChnPort_s
{
    MI_ModuleId_e eModId;
    MI_U32  u32DevId;
    MI_U32 u32ChnId;
    MI_U32 u32PortId;

} MI_SYS_ChnPort_t;
```

➢ 成员

成员名称	描述
eModId	模块号
u32DevId	设备号
u32ChnId	通道号
u32PortId	端口号

※ 注意事项

N/A

➢ 相关数据类型及接口

N/A

3.2.9 MI_SYS_MetaData_t

➢ 说明

定义码流 MetaData 的结构体。

➢ 定义

```
typedef struct MI_SYS_MetaData_s
{
    void*  pVirAddr;
    MI_PHY phyAddr;//notice that this is miu bus addr,not cpu bus addr.

    MI_U32 u32Size;
    MI_U32 u32ExtraData;    /*driver special flag*/
    MI_ModuleId_e eDataFromModule;
} MI_SYS_MetaData_t;
```

➢ 成员

成员名称	描述
pVirAddr	数据存放虚拟地址。
phyAddr	数据存放物理地址
u32Size	数据大小
u32ExtraData	driver special flag
eDataFromModule	来自哪个模块的数据

※ 注意事项

N/A

➢ 相关数据类型及接口

N/A

3.2.10 MI_SYS_RawData_t

➢ 说明

定义码流 RawData 的结构体。

➢ 定义

```
typedef struct MI_SYS_RawData_s
{
    void*   pVirAddr;
    MI_PHY  phyAddr;//notice that this is miu bus addr,not cpu bus addr.
    MI_U32  u32BufSize;

    MI_U32  u32ContentSize;
    MI_BOOL bEndOfFrame;
    MI_U64  u64SeqNum;
} MI_SYS_RawData_t;
```

➢ 成员

成员名称	描述
pVirAddr	码流包的地址。
phyAddr	码流包的物理地址
u32BufSize	Buf 大小
u32ContentSize	数据实际占用 buf 大小
bEndOfFrame	当前帧是否结束。（预留参数）当前只支持帧模式下按帧传送。
u64SeqNum	当前帧的帧序号

※ 注意事项

- 当前只支持按帧传送数据，每次需完整传送一帧数据。
- 码流帧数据附带 PTS 时，解码后输出数据输出相同 PTS。当 PTS=-1 时，不参考系统时钟输出数据帧。

➢ 相关数据类型及接口

N/A

3.2.11 MI_SYS_WindowRect_t

➢ 说明

定义 window 坐标的结构体。

➢ 定义

```
typedef struct MI_SYS_WindowRect_s
{
    MI_U16 u16X;
    MI_U16 u16Y;
    MI_U16 u16Width;
    MI_U16 u16Height;
}MI_SYS_WindowRect_t;
```

➢ 成员

成员名称	描述
u16X	window 起始位置的水平方向的值。
u16Y	window 起始位置的垂直方向的值。
u16Width	window 宽度。
u16Height	window 高度。

※ 注意事项

N/A

➢ 相关数据类型及接口

N/A

3.2.12 MI_SYS_FrameData_t

➢ 说明

定义码流 FrameData 的结构体。

➢ 定义

```
//N.B. in MI_SYS_FrameData_t should never support u32Size,
//for other values are enough,and not support u32Size is general standard method.
typedef  struct  MI_SYS_FrameData_s
{
    MI_SYS_FrameTileMode_e eTileMode;
    MI_SYS_PixelFormat_e ePixelFormat;
    MI_SYS_CompressMode_e eCompressMode;
    MI_SYS_FrameScanMode_e eFrameScanMode;
    MI_SYS_FieldType_e eFieldType;
```

```
    MI_SYS_FrameData_PhySignalType ePhylayoutType;

    MI_U16 u16Width;
    MI_U16 u16Height;
//in case ePhylayoutType equal to REALTIME_FRAME_DATA, pVirAddr would be MI_SYS_REALTIME_MAGIC_PA
DDR and phyAddr would be MI_SYS_REALTIME_MAGIC_VADDR

    void* pVirAddr[3];
    MI_PHY phyAddr[3];//notice that this is miu bus addr,not cpu bus addr.
    MI_U32 u32Stride[3];
    MI_U32 u32BufSize;//total size that allocated for this buffer,include consider alignment.

    MI_U16 u16RingBufStartLine;//Valid in case RINGBUF_FRAME_DATA,  u16RingBufStartLine must be LGE tha
n 0 and less than u16Height
    MI_U16 u16RingBufRealTotalHeight;///Valid in case RINGBUF_FRAME_DATA,  u16RingBufStartLine must be
LGE than u16Height

    MI_SYS_FrameIspInfo_t stFrameIspInfo;//isp info of each frame
    MI_SYS_WindowRect_t stContentCropWindow;
} MI_SYS_FrameData_t;
```

- 成员

成员名称	描述
eTileMode	Tile 模式
ePixelFormat	像素格式
eCompressMode	压缩格式
eFrameScanMode	Frame scan 模块
eFieldType	File 类型
ePhylayoutType	frame data 隶属的 buffer 的类型
u16Width	Frame 宽度
u16Height	Frame 高度
pVirAddr	虚拟地址
phyAddr	物理地址
u32Stride	图片每行所占字节数
u32BufSize	Sys 分配给当前 frame 的实际 buf 大小
u16RingBufStartLine	ring mode 时，frame data 开始于 ring buffer 的行数
u16RingBufRealTotalHeight	ring mode 时，frame data 的真实高度
stFrameIspInfo	ISP info struct
stContentCropWindow	Crop info struct

- 注意事项

  N/A

➢ 相关数据类型及接口

N/A

3.2.13 MI_SYS_BufInfo_t

➢ 说明

定义码流信息的结构体。

➢ 定义

```
typedef struct MI_SYS_BufInfo_s
{
    MI_U64 u64Pts;
    MI_U64 u64SidebandMsg;
    MI_SYS_BufDataType_e eBufType;
    MI_BOOL bEndOfStream;
    MI_BOOL bUsrBuf;
    MI_U32 u32SequenceNumber;
    MI_BOOL bDrop;
    union
    {
        MI_SYS_FrameData_t stFrameData;
        MI_SYS_RawData_t stRawData;
        MI_SYS_MetaData_t stMetaData;
    };
} MI_SYS_BufInfo_t;
```

➢ 成员

成员名称	描述
eBufType	Buf 类型
u64Pts	时间戳
bEndOfStream	是否已发完所有信息

※ 注意事项

N/A

➢ 相关数据类型及接口

N/A

3.2.14 MI_SYS_FrameBufExtraConfig_t

➢ 说明

定义码流 Frame buffer 额外配置的结构体。

➢ 定义

```
typedef struct MI_SYS_FrameBufExtraConfig_s
{
  //Buf alignment requirement in horizontal
  MI_U16 u16BufHAlignment;
  //Buf alignment requirement in vertical
  MI_U16 u16BufVAlignment;
  //Buf alignment requirement in chroma
  MI_U16 u16BufChromaAlignment;
  //Clear padding flag
  MI_BOOL bClearPadding;
}MI_SYS_FrameBufExtraConfig_t;
```

➢ 成员

成员名称	描述
u16BufHAlignment	水平方向对齐值
u16BufVAlignment	垂直方向对齐值
u16BufChromaAlignment	色度 buffer size 对齐值
bClearPadding	是否对 buffer 的边缘进行涂黑

※ 注意事项

N/A

➢ 相关数据类型及接口

N/A

3.2.15 MI_SYS_BufFrameConfig_t

➢ 说明

定义码流 Frame buffer 配置的结构体。

➢ 定义

```
typedef struct MI_SYS_BufFrameConfig_s
{
    MI_U16 u16Width;
    MI_U16 u16Height;
    MI_SYS_FrameScanMode_e eFrameScanMode;//
    MI_SYS_PixelFormat_e eFormat;
    MI_SYS_FrameBufExtraConfig_t stFrameBufExtraConf;//set by MI_SYS internal
    //MI_U32 u32Size;//this value will be calculated through others values in this struct
}MI_SYS_BufFrameConfig_t;
```

➢ 成员

成员名称	描述
u16Width	Frame 宽度
u16Height	Frame 高度
eFrameScanMode	Frame Scan Mode
eFormat	Frame 像素格式
stFrameBufExtraConf	Frame 对齐的 pixel 大小（无需用户设置）

※ 注意事项

N/A

➢ 相关数据类型及接口

N/A

3.2.16 MI_SYS_BufRawConfig_t

➢ 说明

定义码流 raw buffer 配置的结构体。

➢ 定义

```
typedef struct MI_SYS_BufRawConfig_s
{
   MI_U32 u32Size;
}MI_SYS_BufRawConfig_t;
```

➢ 成员

成员名称	描述
u32Size	Buf 大小

※ 注意事项

N/A

➢ 相关数据类型及接口

N/A

3.2.17 MI_SYS_MetaDataConfig_t

➢ 说明

定义码流 meta buffer 配置的结构体。

➢ 定义

```
typedef struct MI_SYS_MetaDataConfig_s
{
   MI_U32 u32Size;
}MI_SYS_MetaDataConfig_t;
```

➢ 成员

成员名称	描述
u32Size	Buf 大小

※ 注意事项

N/A

➢ 相关数据类型及接口

N/A

3.2.18 MI_SYS_BufConf_t

➢ 说明

定义码流 Port Buf 配置的结构体。

➢ 定义

```
typedef struct MI_SYS_BufConf_s
{
   MI_SYS_BufDataType_e eBufType;
   MI_U32 u32Flags;   //0 or MI_SYS_MAP_VA
   MI_U64 u64TargetPts;
   union
   {
      MI_SYS_BufFrameConfig_t stFrameCfg;
      MI_SYS_BufRawConfig_t stRawCfg;
      MI_SYS_MetaDataConfig_t stMetaCfg;
   };
}MI_SYS_BufConf_t;
```

➢ 成员

成员名称	描述
eBufType	Buf type
u32Flags	Buf 是否 map kernel space va

※ 注意事项

N/A

➢ 相关数据类型及接口

N/A

3.2.19 MI_SYS_Version_t

➢ 说明

定义 Sys 版本信息结构体。

➢ 定义

```
typedef struct MI_SYS_Version_s
{
   MI_U8 u8Version[128];
}MI_SYS_Version_t;
```

➢ 成员

成员名称	描述
u8Version[128]	描述 sys 版本信息的字符串 buf

※ 注意事项

N/A

➢ 相关数据类型及接口

N/A

3.2.20 MI_VB_PoolListConf_t

➢ 说明

定义描述 VB Pool 链表信息的结构体。

➢ 定义

```
typedef struct MI_VB_PoolListConf_s
{
   MI_U32 u32PoolListCnt;
   MI_VB_PoolConf_t stPoolConf[MI_VB_POOL_LIST_MAX_CNT];
} MI_VB_PoolListConf_t;
```

➢ 成员

成员名称	描述
u32PoolListCnt	VB pool 链表成员数量
stPoolConf	VB pool 链表成员配置信息

※ 注意事项

N/A

➢ 相关数据类型及接口

N/A

3.2.21 MI_SYS_BindType_e

➢ 说明

定义前后级工作模式。

➢ 定义

```
typedef enum
{
  E_MI_SYS_BIND_TYPE_FRAME_BASE = 0x00000001,
  E_MI_SYS_BIND_TYPE_SW_LOW_LATENCY = 0x00000002,
  E_MI_SYS_BIND_TYPE_REALTIME = 0x00000004,
  E_MI_SYS_BIND_TYPE_HW_AUTOSYNC = 0x00000008,
  E_MI_SYS_BIND_TYPE_HW_RING = 0x00000010
}MI_SYS_BindType_e;
```

➢ 成员

成员名称	描述
E_MI_SYS_BIND_TYPE_FRAME_BASE	frame mode，默认工作方式
E_MI_SYS_BIND_TYPE_SW_LOW_LATENCY	低时延工作方式
E_MI_SYS_BIND_TYPE_REALTIME	硬件直连工作方式
E_MI_SYS_BIND_TYPE_HW_AUTOSYNC	前后级 handshake，buffer size 与图像分辨率一致
E_MI_SYS_BIND_TYPE_HW_RING	前后级 handshake，ring buffer depth 可以调小于图像分辨率高

※ 注意事项

旧版本 MI SYS 不提供此功能，如没有找到，不用设置。

➢ 相关数据类型及接口

N/A

3.2.22 MI_SYS_FrameData_PhySignalType

➢ 说明

描述 frame data 隶属的 buffer 类型的枚举类型。

➢ 定义

```
typedef enum
{
  REALTIME_FRAME_DATA,
  RINGBUF_FRAME_DATA,
  NORMAL_FRAME_DATA,
}MI_SYS_FrameData_PhySignalType;
```

➢ 成员

成员名称	描述
REALTIME_FRAME_DATA	以 E_MI_SYS_BIND_TYPE_REALTIME 模式产生的 frame data
RINGBUF_FRAME_DATA	以 E_MI_SYS_BIND_TYPE_HW_RING 模式产生的 frame data
NORMAL_FRAME_DATA	以 E_MI_SYS_BIND_TYPE_FRAME_BASE 模式产生的 frame data

※ 注意事项
旧版本 MI SYS 不提供此功能，如没有找到，不用设置。

➢ 相关数据类型及接口
N/A

3.2.23 MI_SYS_InsidePrivatePoolType_e

➢ 说明
描述创建的私有 MMA POOL 类型的枚举类型。

➢ 定义

```
typedef enum
{
  E_MI_SYS_VPE_TO_VENC_PRIVATE_RING_POOL = 0,
  E_MI_SYS_PER_CHN_PRIVATE_POOL=1,
  E_MI_SYS_PER_DEV_PRIVATE_POOL=2,
  E_MI_SYS_PER_CHN_PORT_OUTPUT_POOL=3,
}MI_SYS_InsidePrivatePoolType_e;
```

➢ 成员

成员名称	描述
E_MI_SYS_VPE_TO_VENC_PRIVATE_RING_POOL	VPE 与 VENC 的私有 Ring Pool 类型
E_MI_SYS_PER_CHN_PRIVATE_POOL	通道私有 Pool 类型
E_MI_SYS_PER_DEV_PRIVATE_POOL	设备私有 Pool 类型
E_MI_SYS_PER_CHN_PORT_OUTPUT_POOL	输出端口私有 Pool 类型

※ 注意事项
旧版本 MI SYS 不提供此功能，如没有找到，不用设置。

➢ 相关数据类型及接口
N/A

3.2.24 MI_SYS_PerChnPrivHeapConf_t

➢ 说明

定义描述通道私有 MMA Pool 的结构体。

➢ 定义

```
typedef struct MI_PerChnPrivHeapConf_s
{
    MI_ModuleId_e eModule;
    MI_U32 u32Devid;
    MI_U32 u32Channel;
    MI_U8 u8MMAHeapName[MI_MAX_MMA_HEAP_LENGTH];
    MI_U32 u32PrivateHeapSize;
}MI_SYS_PerChnPrivHeapConf_t;
```

➢ 成员

成员名称	描述
eModule	模块 ID
u32Devid	设备 ID
u32Channel	通道 ID
u8MMAHeapName	Mma heap name
u32PrivateHeapSize	私有 pool 大小

※ 注意事项

N/A

➢ 相关数据类型及接口

N/A

3.2.25 MI_SYS_PerDevPrivHeapConf_t

➢ 说明

定义描述设备私有 MMA Pool 的结构体。

➢ 定义

```
typedef struct MI_PerDevPrivHeapConf_s
{
    MI_ModuleId_e eModule;
    MI_U32 u32Devid;
    MI_U32 u32Reserve;
    MI_U8 u8MMAHeapName[MI_MAX_MMA_HEAP_LENGTH];
    MI_U32 u32PrivateHeapSize;
}MI_SYS_PerDevPrivHeapConf_t;
```

➢ 成员

成员名称	描述
eModule	模块 ID
u32Devid	设备 ID
u32Reserve	预留
u8MMAHeapName	Mma heap name
u32PrivateHeapSize	私有 pool 大小

※ 注意事项

N/A

➢ 相关数据类型及接口

N/A

3.2.26 MI_SYS_PerVpe2VencRingPoolConf_t

➢ 说明

定义描述 VPE 与 VENC 之间的私有 Ring MMA Pool 的结构体。

➢ 定义

```
typedef struct MI_SYS_PerVpe2VencRingPoolConf_s
{
    MI_U32 u32VencInputRingPoolStaticSize;
    MI_U8 u8MMAHeapName[MI_MAX_MMA_HEAP_LENGTH];
}MI_SYS_PerVpe2VencRingPoolConf_t;
```

➢ 成员

成员名称	描述
u32VencInputRingPoolStaticSize	私有 pool 大小
u8MMAHeapName	Mma heap name

※ 注意事项

N/A

➢ 相关数据类型及接口

N/A

3.2.27 MI_SYS_PerChnPortOutputPool_t

➢ 说明

定义描述 ouput port 的私有 MMA Pool 的结构体。

➢ 定义

```
typedef struct MI_SYS_PerChnPortOutputPool_s
{
    MI_ModuleId_e eModule;
    MI_U32 u32Devid;
    MI_U32 u32Channel;
    MI_U32 u32Port;
    MI_U8 u8MMAHeapName[MI_MAX_MMA_HEAP_LENGTH];
    MI_U32 u32PrivateHeapSize;
}MI_SYS_PerChnPortOutputPool_t;
```

➢ 成员

成员名称	描述
eModule	模块 ID
u32Devid	设备 ID
u32Channel	通道 ID
u32Port	端口 ID
u8MMAHeapName	Mma heap name
u32PrivateHeapSize	私有 pool 大小

※ 注意事项

N/A

➢ 相关数据类型及接口

N/A

3.2.28 MI_SYS_GlobalPrivPoolConfig_t

➢ 说明

定义描述配置私有 MMA Pool 的结构体。

➢ 定义

```
typedef struct MI_SYS_GlobalPrivPoolConfig_s
{
    MI_SYS_InsidePrivatePoolType_e eConfigType;
    MI_BOOL bCreate;
    union
    {
        MI_SYS_PerChnPrivHeapConf_t stPreChnPrivPoolConfig;
        MI_SYS_PerDevPrivHeapConf_t stPreDevPrivPoolConfig;
        MI_SYS_PerVpe2VencRingPoolConf_t stPreVpe2VencRingPrivPoolConfig;
        MI_SYS_PerChnPortOutputPool_t stPreChnPortOutputPrivPool;
    }uConfig;
}MI_SYS_GlobalPrivPoolConfig_t;
```

➢ 成员

成员名称	描述
eConfigType	私有 pool 类型
bCreate	是否创建私有 pool。true:创建 false:销毁
uConfig	不同类型私有 pool 的结构体

※ 注意事项

N/A

➢ 相关数据类型及接口

N/A

3.2.29 MI_SYS_FrameIspInfo_t

➢ 说明

定义 frame data 中 ISP info 的结构体。

➢ 定义

```
typedef struct MI_SYS_FrameIspInfo_s
{
   MI_SYS_FrameIspInfoType_e eType;
   union
   {
      MI_U32 u32GlobalGradient;
   }uIspInfo;
}MI_SYS_FrameIspInfo_t;
```

➢ 成员

成员名称	描述
eType	ISP info type
uIspInfo	ISP info union

※ 注意事项

N/A

➢ 相关数据类型及接口

N/A

4. 错误码

4.1. 错误码的组成

MI 返回的错误码为 4 字节，共 32bits，由表 4-1 所示五部分组成：

表 4-1 错误码的组成

位置	所占位数	描述
BIT [0-11]	12bits	错误类型，表示该错误码的具体错误含义。
BIT [12-15]	4bits	错误等级，固定返回 2。
BIT [16-23]	8bits	模块 ID，表示该错误码是属于哪个模块。
BIT [24-31]	8bits	固定为 0xA0。

Tips:

我们可以简单的把错误码看成两个双字节（16bits）做快速解读，以错误码 0xA0092001 为例：
"A009"：表示该错误发生在模块 ID 为 9 的模块，通过查看"MI_ModuleId_e"的定义可知为 MI_SYS 模块。
"2001"：表示该错误的错误类型是 1，即"设备 ID 超出合法范围"。

4.2. MI_SYS API 错误码表

MI_SYS 模块所有 API 返回的值都为 4 字节，0 表示执行成功，其它值表示执行失败，需要注意的是，所有非 0 的返回值都应该是表 4-2 中列出的，否则为非法值。

表 4-1 MI_SYS 模块 API 错误码

错误码	宏定义	描述
0xA0092001	MI_ERR_SYS_INVALID_DEVID	设备 ID 超出合法范围
0xA0092002	MI_ERR_SYS_INVALID_CHNID	通道组号错误或无效区域句柄
0xA0092003	MI_ERR_SYS_ILLEGAL_PARAM	参数超出合法范围
0xA0092004	MI_ERR_SYS_EXIST	重复创建已存在的设备、通道或资源
0xA0092005	MI_ERR_SYS_UNEXIST	试图使用或者销毁不存在的设备、通道或者资源
0xA0092006	MI_ERR_SYS_NULL_PTR	函数参数中有空指针
0xA0092007	MI_ERR_SYS_NOT_CONFIG	模块没有配置
0xA0092008	MI_ERR_SYS_NOT_SUPPORT	不支持的参数或者功能
0xA0092009	MI_ERR_SYS_NOT_PERM	该操作不允许，如试图修改静态配置参数
0xA009200C	MI_ERR_SYS_NOMEM	分配内存失败，如系统内存不足
0xA009200D	MI_ERR_SYS_NOBUF	分配缓存失败，如申请的数据缓冲区太大
0xA009200E	MI_ERR_SYS_BUF_EMPTY	缓冲区中无数据
0xA009200F	MI_ERR_SYS_BUF_FULL	缓冲区中数据满

错误码	宏定义	描述
0xA0092010	MI_ERR_SYS_NOTREADY	系统没有初始化或没有加载相应模块
0xA0092011	MI_ERR_SYS_BADADDR	地址非法
0xA0092012	MI_ERR_SYS_BUSY	系统忙
0xA0092013	MI_ERR_SYS_CHN_NOT_STARTED	通道没有开始
0xA0092014	MI_ERR_SYS_CHN_NOT_STOPED	通道没有停止
0xA0092015	MI_ERR_SYS_NOT_INIT	模块没有初始化
0xA0092016	MI_ERR_SYS_INITED	模块已经初始化
0xA0092017	MI_ERR_SYS_NOT_ENABLE	通道或端口没有 ENABLE
0xA0092018	MI_ERR_SYS_NOT_DISABLE	通道或端口没有 DISABLE
0xA0092019	MI_ERR_SYS_TIMEOUT	超时
0xA009201A	MI_ERR_SYS_DEV_NOT_STARTED	设备没有开始
0xA009201B	MI_ERR_SYS_DEV_NOT_STOPED	设备没有停止
0xA009201C	MI_ERR_SYS_CHN_NO_CONTENT	通道没有资料
0xA009201D	MI_ERR_SYS_NOVASAPCE	映射虚拟地址失败
0xA009201E	MI_ERR_SYS_NOITEM	RingPool 中没有 record 记录
0xA009201F	MI_ERR_SYS_FAILED	未明确定义的错误