**CMDQ特性设计规格文档**

**Version 2.0**

**Prepared by <marvin.chen>**

**<last modified date>**

© 2017 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* | *CMDQ SW 设计* | *2016/12/22* | *Marvin.chen* |  |
| *2.0* | *CMDQ SW 设计：update internal interface* | *2017/3/15* | *Marvin.chen* |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

TABLE OF CONTENT

[REVISION HISTORY i](#_Toc477422406)

[TABLE OF CONTENT 1](#_Toc477422407)

[LIST OF TABLES 2](#_Toc477422408)

[LIST OF FIGURES 2](#_Toc477422409)

[1. 介绍 4](#_Toc477422410)

[1.1. 目的 4](#_Toc477422411)

[1.2. 简介 4](#_Toc477422412)

[2. 整体描述 5](#_Toc477422413)

[2.1. 系统描述 5](#_Toc477422414)

[2.2. 特性列表 6](#_Toc477422415)

[2.2.1. Command的格式 6](#_Toc477422416)

[2.2.2. DMA mode 6](#_Toc477422417)

[2.2.3. CMDQ的Work mode 8](#_Toc477422418)

[2.3. 假设和依赖 9](#_Toc477422419)

[2.4. 内部限制 9](#_Toc477422420)

[3. 功能需求 11](#_Toc477422421)

[3.1. basic mode use case 11](#_Toc477422422)

[3.1.1. Use Case: GE draw a graph 11](#_Toc477422423)

[3.2. multi-window下use case 11](#_Toc477422424)

[3.2.1. Use Case: 播放第一路DTV或者MM 11](#_Toc477422425)

[3.2.2. Use Case:更新command queue 13](#_Toc477422426)

[3.2.3. Use Case: 添加一路DTV或者MM 14](#_Toc477422427)

[3.2.4. Use Case: 删除一路DTV或者MM 15](#_Toc477422428)

[3.2.5. Use Case:交换两个window的layer 16](#_Toc477422429)

[3.2.6. Use Case: 一路DTV或者MM退出 17](#_Toc477422430)

[4. 非功能性需求设计 19](#_Toc477422431)

[4.1. 可靠性需求设计 19](#_Toc477422432)

[4.2. CMDQ debug需求设计 19](#_Toc477422433)

[4.2.1. CMDQ的log 19](#_Toc477422434)

[4.2.2. 判断command queue是否被更新 19](#_Toc477422435)

[4.2.3. 暂停CMDQ debug单个window 19](#_Toc477422436)

[5. 接口描述 20](#_Toc477422437)

[5.1. 枚举定义 20](#_Toc477422438)

[5.1.1. 外部使用的枚举定义 20](#_Toc477422439)

[5.1.2. 内部使用的枚举定义 20](#_Toc477422440)

[5.2. 结构体定义 24](#_Toc477422441)

[5.2.1. 外部使用的结构体定义 24](#_Toc477422442)

[5.2.2. 内部使用结构体定义 24](#_Toc477422443)

[5.3. 接口定义 26](#_Toc477422444)

[5.3.1. 外部使用接口定义 26](#_Toc477422445)

[5.3.2. 内部使用接口定义 27](#_Toc477422446)

LIST OF TABLES

[Table 1缩略语清单 3](#_Toc477422447)

[Table 2 CMDQ的command格式 6](#_Toc477422448)

[Table 3 Feature list 8](#_Toc477422449)

LIST OF FIGURES

[Figure 1 The block diagram of CMDQ(basic mode) 5](#_Toc477422450)

[Figure 2 The block diagram of CMDQ(multi-window mode) 5](#_Toc477422451)

[Figure 3 Direct Mode 7](#_Toc477422452)

[Figure 4 Increment mode 7](#_Toc477422453)

[Figure 5 ring-buffer mode 8](#_Toc477422454)

[Figure 6 window cmd set 9](#_Toc477422455)

[Figure 7 command queue 9](#_Toc477422456)

[Figure 8 command queue执行示意图 9](#_Toc477422457)

[Figure 9 add the first window 12](#_Toc477422458)

[Figure 10 add window cmd set 14](#_Toc477422459)

[Figure 11 add a window 15](#_Toc477422460)

[Figure 12 delete window 16](#_Toc477422461)

[Figure 13 change window layer 17](#_Toc477422462)

[Figure 14 delete the last window 18](#_Toc477422463)

[Figure 15 set CMDQ buffer flow 26](#_Toc477422464)

[Figure 16 Set DMA mode 28](#_Toc477422465)

[Figure 17 set command address 30](#_Toc477422466)

[Figure 18 set write pointer 31](#_Toc477422467)

[Figure 19 Add command in basic mode 33](#_Toc477422468)

[Figure 20 set CMDQ work mode 34](#_Toc477422469)

[Figure 21 set frame done 36](#_Toc477422470)

[Figure 22 get running queue ID 37](#_Toc477422471)

[Figure 23 add command in multi-window mode 38](#_Toc477422472)

[Figure 24 enable/disable CMDQ flow 39](#_Toc477422473)

Table 1缩略语清单

|  |  |
| --- | --- |
| **缩略语** | **描述** |
| HW | Hardware |
| CMDQ | Command queue HW |
| PQ | Picture quality |
| cmd set | 一个window的一个frame需要执行的command的集合 |
| Command queue | 多个window要在同一个vsync中执行的要执行的cmd set组合 |
| IP | Intellectual property |
| FRC | Frame rate control |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

# 介绍

## 目的

本文主要介绍专门为Scaler，MVOP，PQ新增的这套CMDQ的软件架构设计，用于指导后续LLD和PV测试。

## 简介

CMDQ可以完成一些register的设定，可以监视某些信号和register状态的变化，能够按照顺序执行memory中已经添加的正确格式的commands，由于可以受到HW严格控制，能够比SW的控制更加精准，因此，可以用来实现对设定register时序要求更高的flow设计。

# 整体描述

本文主要描述专门为Scaler，MVOP，PQ新增的这套CMDQ的SW架构设计，描述CMDQ的基本功能及在multi-window下的应用，CMDQ的code和scaler的code放在同一个lib中。

## 系统描述

CMDQ主要是用来执行添加到memory中的commands，按照相应的时序完成相关的设定。当CMDQ工作在basic mode下时，其他module可以设定CMDQ的DMA mode，然后添加command，最后trigger CMDQ就会执行添加到memory中的commands，这些commands可以等待某些信号，或者监视register状态，也可以写register，通过这些command的动作，可以实现对指定HW IP的控制，如figure 1所示。CMDQ也可以工作在multi-window mode下，工作方式如figure 2所示，这个时候CMDQ会循环执行各个command queue中的所有commands，从而实现对多个window的精准控制。在增加window，删除window或者window layer发生变化时，MWS会call Overlap模块。每个OP Vsync，上层MWS都会call Script模块造MVOP，PQ和scaler的剧本，Script造完剧本之后，会向CMDQ添加相应的commands，在一个window的commands全部添加完之后会call CMDQ的window done接口告知CMDQ这个window的command添加结束，在每个OP Vysnc要执行的各个window的command全部添加完之后，FRC会call CMDQ的frame done接口，CMDQ会将新添加的command根据window layer排序之后更新到command queue中。



Figure 1 The block diagram of CMDQ(basic mode)



Figure 2 The block diagram of CMDQ(multi-window mode)

## 特性列表

本节主要描述CMDQ的command格式，DMA mode和工作方式（work mode）。

### Command的格式

CMDQ的command是64 bit的command，主要有NULL，wait，write，polling\_eq，polling\_neq五种，具体格式如下表所示：

Table 2 CMDQ的command格式



* Null: 不做任何动作，继续执行下一条command，NULL command的bit60~63的值为0，其他bit

是无意义的.

* Write: 根据mask写data到指定的地址.
* Wait: 等待某个trigger bus的信号到来，然后执行下一条command.
* Polling\_eq: 读取指定地址的值并检查指定bit，如果相等则执行下一条command
* Polling\_neq: 读取指定地址的值并检查指定bit，如果不相等则执行下一条command.
* **Command的mask和通常XC的register的mask不同，mask bit的值为1时，表示忽略该bit，值为0时有效。在我们的SW中，会在CMDQ中对的mask做取反动作，让其他module使用CMDQ mask的方式和scaler的一致，mask bit的值为0时，表示忽略该bit，值为1时有效。**

### DMA mode

Memory中需要执行的一组command的集合称之为command set，CMDQ执行memory中的commands的方式（DMA mode）有三种：direct mode，increment mode 和 ring-buffer mode。在basic mode下，CMDQ可以工作在这3种DMA mode下，在multi-window mode下，CMDQ的DMA方式固定使用direct mode。

Direct mode下command在memory中的排布方式如figure 3所示，每次需要设定CMDQ执行的comand set的start/end address，然后给CMDQ发送地址变化的信息（trigger），CMDQ收到trigger信号之后，会给MIU发送request从指定的memory中抓取commands数据，然后执行这些commands。CMDQ在执行commands的过程中，如果有多次修改start/end address和设定多个trigger信号，CMDQ只会记录最后一个。因此，在direct mode下，为保障CMDQ安全有序的执行commands，如果有改变commands地址的需求时，最好先查询CMDQ的状态，在idle状态下才重设star/end address和发送trigger信号。另外需要注意的是end pointer并不是read command的最后一个地址，end pointer – 1才是最后一条command的地址。



Figure 3 Direct Mode

Increment mode下，需要设定CMDQ执行的command set的start address和offset，command在memory中的排布如figure 4所示，CMDQ在收到地址信息变化的trigger信号之后，可以通过计算知道需要执行的commands数量。在CMDQ执行commands的过程中如果start address没有变化，可以接收到多个offset变化的信号之后，会将offset的值进行计算，记录最后的offset，在执行完指定的commands之后会立即执行后面添加的commands。由于CMDQ的memory是有限的，所以在increment mode下，添加完一段时间的commands之后可能就是用完memory了，这样就需要变更start address。如果commands的start address发生改变，需要发送地址信息变化的trigger信号和offset变化的trigger信号给CMDQ。



Figure 4 Increment mode

ring-buffer mode下，commands被写入一块指定的memory中，如figure 5所示，有一个HW读指针和一个SW写指针，两个指针在设定的commands memory区间内来回移动。一旦SW将commands写完到command memory中，应立即设置写指针并发送地址信息变化的trigger信号通知CMDQ，CMDQ将会执行SW写完的这些commands，当HW读指针和SW写指针相等时，CMDQ会自动停下来。



Figure 5 ring-buffer mode

### CMDQ的Work mode

CMDQ有两种工作方式，即basic mode和multi-window mode。在basic mode下，可以完成register的设定，以实现对某些HW IP的初始化，开关等操作；在multi-window mode下可以让CMDQ一直处于运行状态，只是不断地添加commands，精准的实现对multi-window的flow控制。

Table 3 Feature list

|  |  |
| --- | --- |
| ID | Feature |
| 1 | 添加和执行command，可以监视某些trigger bus的状态变化或者register的状态变化，也可以完成HW IP的初始化，开关等动作。 |
| 2 | 循环执行command queue，在时序上精准的控制Scaler的分时复用，实现multi-window功能。 |

在multi-window mode下，CMDQ的DMA方式使用direct mode，每个OP Vysnc每个window要执行的command的集合称之为cmd set，如figure 6所示。多个window在同一个OP Vysnc的cmd set和clock的设定一起组成一个command queue，如figure 7所示。在CMDQ中可能有多个command queue，CMDQ会循环执行这些command queue，有4条command queue的示意图如figure 8 所示。每个OP Vysnc会执行一个command queue，执行完一个window的cmd set之后跳到另一个window的cmd set继续执行，当一个command queue的command全部被执行完之后，跳到另一个command queue。由于使用的DMA mode是direct mode，在执行完一个window的cmd set之后，需要跳转到下一个window的cmd set，因此，每个cmd set的最后3条是重设command的start/end address和trigger point。一般在每个OP Vsync，其他module都会向CMDQ添加新的command，更新command queue。



Figure 6 window cmd set



Figure 7 command queue



Figure 8 command queue执行示意图

## 假设和依赖

1 multi-window mode下，每个window一个frame的所有commands add完之后，Script模块需要call CMDQ的接口通知CMDQ该window的commands已经添加完成。

2 multi-window mode下，CMDQ的DMA mode设定为Direct mode。

3 multi-window mode下，CMDQ一直循环执行各个command queue，当SW运行出现performance问题，未及时添加更新command时，仍会继续执行已经执行过的command。

4 multi-window mode下，在一个OP Vsync到来前要执行的多个window的command全部添加完成之后，FRC模块要通知CMDQ模块要更新哪些window的cmd set到command queue中，将这些cmd set排序之后添加到command queue中。

## 内部限制

在multi-window mode下，CMDQ一直处于运行状态，其他module不能像basic mode那样使用CMDQ，如果需要通过CMDQ来完成一些工作，必须将这些command按照window和stage添加到window的cmd set中，然后更新到command queue中去实现相关的功能。

# 功能需求

本节主要描述CMDQ的use case，包括basic mode和multi-window mode下的use case。本节将重点描述multi-window mode下的use case。

## basic mode use case

Basic use case是CMDQ的基本功能，其他module可以设定CMDQ的工作方式，添加command，控制CMDQ执行command，实现对GE，GOP，MenuLoad，DS，AutoDownload等IP完成相关的设定。

### Use Case: GE draw a graph

|  |  |
| --- | --- |
| **UCID** | 1 |
| **Use case名字** | GE画图GOP抓图 |
| **简介** | 客户要求通过GE画图存入memory中，通过GOP从memory中抓图送给后端显示 |
| **前置条件** | 客户正确配置config |
| **预期结果** | (Success) GOP抓到正确的图  (Exception) GE画图或者GOP抓图失败 |
| **触发条件** | 在某些场景下用户按下遥控按钮需要画OSD |
| **基本场景** | 1. 设置GE，GOP，RIU和CMDQ的register。 2. 添加command到CMDQ的memory中。 3. 设置CMDQ trigger，CMDQ开始动作。 4. 监控CMDQ的中断即可知道CMDQ是否完成。 |
| **可选场景** | 无 |
| **异常场景** | Command的排序错误，画图或者抓图失败 |
| **备注** | 无 |

## multi-window下use case

在multi window mode下，第一路video起播前， vdec提早解码和造剧本，将需要CMDQ执行的command添加到CMDQ，先填满command queue后再开启CMDQ。因此CMDQ的command queue会有缓存，每次添加的新的command queue不一定会在下一个Vsync执行。需要注意，PQ adaptive部分的设定要在每个OP Vsync去更新下一个OP Vsync到来前要load的设定，HW并没有这样的机制，因此需要Script模块更新完剧本之后去设定dummy register，然后更新CMDQ下一个OP Vsync要执行的command queue，如果状态不正确就一直等待Script更新PQ adaptive的设定。

### Use Case: 播放第一路DTV或者MM

|  |  |
| --- | --- |
| **UCID** | 1 |
| **Use case名字** | 播放第一路DTV或者MM |
| **简介** | 各个module通过CMDQ控制HW IP完成各项设定播放video |
| **前置条件** | 1. 系统正常起来。 2. UI Setting播放视频MM或DTV。 |
| **预期结果** | 成功：播放第一路视频，能正常播放。  失败：播放视频时出现花屏，抖动，闪garbage等现象 |
| **触发条件** | 按TV的UI，播放第一路DTV或者MM |
| **基本场景** | 1. MWS获取scaler的buffer queue number。 2. Vdec flip一帧，MVOP，PQ，Scaler会call Script模块在造好剧本之后，Script会add command给CMDQ，每一个frame的command放在同一个cmd set节点中。每造好一个frame的剧本，script会call一次CMDQ的set window done，FRC会call CMDQ的frame done，CMDQ将该cmd set节点添加到一个command queue中。 3. 在command queue都填满之后，FRC call CMDQ的接口enable CMDQ，在OP Vsync到来时CMDQ开始执行command queue 0的cmd set。 4. 保存最后一个被更新的command queue的下一个command queue的ID。 5. CMDQ循环执行各个command queue。 6. 在每个frame done到来时会更新最先执行完且未更新的那条command queue。 |
| **可选场景** | N/A |
| **异常场景** | N/A |
| **备注** | N/A |



Figure 9 add the first window

### Use Case:更新command queue

|  |  |
| --- | --- |
| **UCID** | 3 |
| **Use case名字** | 播放多路DTV或者MM时更新command queue |
| **简介** | Script模块造完剧本之后，需要add command到CMDQ |
| **前置条件** | 1已有的多路视频播放正常。  2 Script正确的add command。 |
| **预期结果** | 成功：多路视频正常播放。  失败：更新节点后播放视频时出现花屏，抖动，闪garbage等现象。 |
| **触发条件** | 上层call Script模块造剧本 |
| **基本场景** | 1找到各个window对应的cmd set节点  2接收Script模块add进来的所有command，按stage存储到空节点中。  3在script call window done后，停止向该节点中添加command，并将该节点添加到每个window自己的temp cmd set链表的最后。如果在一个OP Vsync的时间内有造同一个window的多个剧本，会产生多个cmd set节点。  4在FRC call frame done时，从需要造剧本的各个window的 cmd set链表中拿出最早添加的cmd set节点按照Overlap设定的window order进行排序，并将该这些cmd set节点添加到上次保存的command queue ID对应的command queue中。  5从window的cmd set链表中移除已经取走的节点。  6保存下一个将要被更新的command queue的ID |
| **可选场景** | N/A |
| **异常场景** | N/A |
| **备注** | N/A |



Figure 10 add window cmd set

### Use Case: 添加一路DTV或者MM

在添加一路DTV或者MM时，对CMDQ架构不会有影响，只是在会更新的command queue中新增一个该window对应的cmd set节点，如图11所示，该节点生效的时间会滞后，滞后的时间为执行完已经更新单尚未执行的command queue的时间。

|  |  |
| --- | --- |
| **UCID** | 4 |
| **Use case名字** | 添加一路DTV或者MM |
| **简介** | Script模块造完剧本之后，需要add command到CMDQ |
| **前置条件** | 1已有的多路视频播放正常。  2 Script正确的add command。 |
| **预期结果** | 成功：增加的一路视频正常播放。  失败：播放视频时出现花屏，抖动，闪garbage等现象 |
| **触发条件** | UI上多开播放一路video |
| **基本场景** | 1创建一个cmd set节点  3接收Script模块add进来的所有command，按stage存储到新创建的节点中。  4在script call window done后，停止向该节点中添加command，并将该节点添加到每个window自己的temp cmd set链表的最后。如果在一个OP Vsync的时间内有造同一个window的多个剧本，会产生多个cmd set节点。  5在FRC call frame done时，从需要造剧本的各个window的 cmd set链表中拿出最早添加的cmd set节点按照Overlap设定的window order进行排序，并将该这些cmd set节点添加到上次保存的command queue ID对应的command queue中。  6从window的cmd set链表中移除已经取走的节点。  7保存下一个将要被更新的command queue的ID |
| **可选场景** | N/A |
| **异常场景** | N/A |
| **备注** | N/A |



Figure 11 add a window

### Use Case: 删除一路DTV或者MM

在删除一路DTV或者MM时，对CMDQ架构不会有影响，只是在更新的command queue中删除一个该window对应的cmd set节点，而且已经更新完的command queue中不会删除该window的cmd set节点，因此，删除的一路视频的时间会有delay，delay的时间为执行完已经add的command queue的时间。

|  |  |
| --- | --- |
| **UCID** | 5 |
| **Use case名字** | 删除一路DTV或者MM |
| **简介** | Script模块造完剧本之后，需要add command到CMDQ |
| **前置条件** | 1已有的多路视频播放正常。  2 Script正确的add command。 |
| **预期结果** | 成功：多路视频正常播放。  失败：删除一路视频后出现花屏，抖动，闪garbage等现象 |
| **触发条件** | 上层关闭一路video |
| **基本场景** | 1上层call Overlap重新设置window order  2 上层call destroy window，CMDQ删除该window所有未加入command queue的cmd set节点  3在FRC call frame done时，从需要造剧本的各个window的 cmd set链表中拿出最早添加的cmd set节点按照Overlap设定的window order进行排序，并将该这些cmd set节点添加到上次保存的command queue ID对应的command queue中。  4从window的cmd set链表中移除已经取走的节点。  5删除temp cmd set链表中该window的所有cmd set节点  6保存下一个将要被更新的command queue的ID |
| **可选场景** | N/A |
| **异常场景** | N/A |
| **备注** | N/A |



Figure 12 delete window

### Use Case:交换两个window的layer

在交换两个window的layer时，对CMDQ架构不会有影响，只是在更新command queue时，cmd set链表排序会与已更新的command queue不同，而且已经更新完的command queue中cmd set节点的顺序不会被改变，因此，交换两个window layer的时间会滞后3个Vsync。

|  |  |
| --- | --- |
| **UCID** | 6 |
| **Use case名字** | 交换两个window的layer |
| **简介** | Script模块造完剧本之后，需要add command到CMDQ |
| **前置条件** | 1多路视频播放正常。  2 Script正确的add command。 |
| **预期结果** | 成功：多路视频正常播放。  失败：播放视频时出现花屏，抖动，闪garbage等现象 |
| **触发条件** | 上层交换两个window的顺序 |
| **基本场景** | 1上层call Overlap模块的接口重新设置window order  2创建新的cmd set节点  3接收Script模块add进来的所有command，按stage存储到新创建的节点中。  4在script call window done后，停止向改节点中添加command，并将该节点添加到每个window自己的temp cmd set链表的最后。如果在一个OP Vsync的时间内有造同一个window的多个剧本，会产生多个cmd set节点。  5在FRC call frame done时，从需要造剧本的各个window的 cmd set链表中拿出最早添加的cmd set节点按照Overlap设定的window order进行排序，并将该这些cmd set节点添加到上次保存的command queue ID对应的command queue中。  6保存下一个将要被更新的command queue的ID |
| **可选场景** | N/A |
| **异常场景** | N/A |
| **备注** | N/A |



Figure 13 change window layer

### Use Case: 一路DTV或者MM退出

在只有一路DTV或者MM时，CMDQ仍然工作。当唯一的一路DTV或者MM退出后CMDQ停止工作，需要将4个command queue中的节点归还到memory control list中，设置各个command queue的头节点指向尾结点。

|  |  |
| --- | --- |
| **UCID** | 7 |
| **Use case名字** | 退出最后一路DTV或者MM |
| **简介** | 上层用户想退出DTV或者MM播放 |
| **前置条件** | 1路DTV或者MM播放正常。 |
| **预期结果** | 成功：退出视频播放。  失败：退出视频播放失败。 |
| **触发条件** | MWS call CMDQ停止工作 |
| **基本场景** | 1FRC disable CMDQ。  2删除该window未添加到command queue中的所有节点。  3重置command queue。  4重置command start/end address为buffer address。  5保存下一个将要被更新的command queue的ID为command queue 0。 |
| **可选场景** | N/A |
| **异常场景** | N/A |
| **备注** | N/A |



Figure 14 delete the last window

# 非功能性需求设计

## 可靠性需求设计

1所有window的cmd set必须在两个OP Vsync的时间内（16.7ms）执行完。

2所有需要造cmd set的window都必须提前将cmd set添加到CMDQ，如果所有缓冲的command queue都已经执行完，CMDQ会重复执行已添加的command queue。

3在每个window的一个frame完成时，必须call window done，才能将cmd set节点添加到cmd set链表中。

4在每个OP Vsync到来时必须call CMDQ的frame done，才能将新增的cmd set链表更新到command queue中。

## CMDQ debug需求设计

CMDQ在 multi-window下一直处于运行状态，HW会一直动作，SW很难介入，因此需要借助dummy register来debug。

### CMDQ的log

CMDQ的log分3个档次：

Level 0：设置CMDQ的BK00\_7F\_[15]=1, CMDQ的打印err message。

Level 1：设置设置CMDQ的BK00\_7F\_[8]=1, debug level被设置成info，会打印相关的debug信息。

Level 2：设置设置CMDQ的BK00\_7F\_[9]=1, debug level被设置成ALL，会打印每一个函数的输入输出参数信息。

Dump command：设置设置CMDQ的BK00\_7F\_[10]=1, 会打印出添加到command queue中的所有command。

### 判断command queue是否被更新

在CMDQ的debug level设置成level 0的时候，使用dummy register来表示每个command queue是否被更新，当CMDQ执行完这个command queue之后，将对应的dummy register bit设置为0，在SW更新改command queue后设置为1，可以通过查看register的值得知command queue是否被更新。（需要使用dummy register的bit数量等于command queue的数量）

### 暂停CMDQ debug单个window

在CMDQ的BK00\_7F\_[11]= 1的时候，在创建window的cmd set时添加debug的command，读取BK00\_7F\_[3:0]=window-2，在这个window的cmd set最后增加一条polling\_neq command去检查BK00\_7F\_[3:0]不等于window-2，这样就能暂停在该window。可以debug这个window的register设定，可以dump register和踩register。

# 接口描述

本章主要描述CMDQ模块提供的接口，结构体和枚举的定义和使用。这个部分包含内部使用和外部使用的接口定义。

## 枚举定义

### 外部使用的枚举定义

|  |  |
| --- | --- |
| 枚举类型名称 | 说明 |
| EN\_XC\_CMDQ\_ID | CMDQ的ID，代表使用的是哪套CMDQ HW，目前只有一套专为multi window使用的CMDQ。 |

**1 EN\_XC\_CMDQ\_ID**

/// Define CMDQ type

typedef enum

{

E\_XC\_CMDQ\_MULTI\_WINDOW=0, ///CMDQ in scaler for multi-window

E\_XC\_CMDQ\_MAX ///CMDQ NUMBER

}EN\_XC\_CMDQ\_ID;

### 内部使用的枚举定义

|  |  |
| --- | --- |
| 枚举类型名称 | 说明 |
| EN\_CMDQ\_DMA\_MODE | CMDQ抓取执行command的方式 |
| EN\_CMDQ\_WORK\_MODE | CMDQ的工作模式：basic mode和multi window mode，basic mode下可以实现CMDQ的基本功能；multi window mode下CMDQ一直处于busy状态，不断处理多个window的command，无法用来实现其他功能。 |
| EN\_CMDQ\_STATUS | CMDQ的状态：idle or busy。如果需要使用CMDQ去完成一些工作时，要先检测CMDQ的状态，当CMDQ处于busy状态时，有些操作可能被忽略，例如CMDQ工作在basic mode，且DMA方式为direct mode时，如果CMDQ处于busy状态，连续更新多次command的start/end address，只有最后一次设定的有效，前面的将被忽略。 |
| EN\_CMDQ\_COMMAND\_TYPE | Command的type |
| EN\_CMDQ\_ADD\_COMMAND\_MODE | 添加command的方式 |
| EN\_CMDQ\_TRIGGER\_BUS\_ID | CMDQ可以监听的trigger bus的ID |
| EN\_CMDQ\_STAGE\_TYPE | Multi window下，添加的command需要在哪个stage执行 |

**1 EN\_CMDQ\_DMA\_MODE**

//define CMDQ DMA mode

typedef enum

{

E\_CMDQ\_DMA\_DIRECT\_MODE, ///(0) direct mode

E\_CMDQ\_DMA\_INCREMENT\_MODE,///(1) increment mode

E\_CMDQ\_DMA\_RING\_BUFFER\_MODE,///(2) ring buffer mode

E\_CMDQ\_DMA\_DMA\_MAX,///(3) DMA mode number

}EN\_CMDQ\_DMA\_MODE;

**2 EN\_CMDQ\_WORK\_MODE**

//define CMDQ work mode

typedef enum

{

E\_CMDQ\_WORK\_BASIC\_MODE, ///(0) basic mode:

E\_CMDQ\_WORK\_MULTI\_WINDOW\_MODE,///(1) multi-window mode

E\_CMDQ\_WORK\_MAX,///(2) work mode number

}EN\_CMDQ\_WORK\_MODE;

**3 EN\_CMDQ\_STATUS**

//define CMDQ status

typedef enum

{

E\_CMDQ\_IDLE,

E\_CMDQ\_BUSY,

}EN\_CMDQ\_STATUS;

**4 EN\_CMDQ\_COMMAND\_TYPE**

//define CMDQ command type

typedef enum

{

E\_CMDQ\_COMMAND\_NULL, ///(0) NULL command: do nothing and excute next command

E\_CMDQ\_COMMAND\_WRITE, ///(1) write command:write register

E\_CMDQ\_COMMAND\_WAIT, ///(2) wait command:wait trigger bus signal

E\_CMDQ\_COMMAND\_POLLING\_EQ, ///(3) polling\_eq command: watch a register value

E\_CMDQ\_COMMAND\_POLLING\_NEQ, ///(4) polling\_neq command: watch a register value

E\_CMDQ\_COMMAND\_MAX, ///(5) command type number

}EN\_CMDQ\_COMMAND\_TYPE;

**5 EN\_CMDQ\_ADD\_COMMAND\_MODE**

//define add command mode

typedef enum

{

E\_CMDQ\_ADD\_MULTI\_WINDOW\_COMMAND,///(1) add a command of one window in multi-window mode

E\_CMDQ\_ADD\_NEXT\_FRAME\_SETTING\_COMMAND,///(2)update PQ adaptive command

E\_CMDQ\_ADD\_COMMAND\_MAX,///(3) add commnad mode number

}EN\_CMDQ\_ADD\_COMMAND\_MODE;

6 **EN\_CMDQ\_TRIGGER\_BUS\_ID**

//define trigger Bus ID, wait command signal definition

typedef enum

{

E\_CMDQ\_TRIGGER\_BUS\_GE2CMDQ\_TRI,///(0) GE become idle, and send a pulse to CMDQ

E\_CMDQ\_TRIGGER\_BUS\_INT\_BDMA\_0,///(1) Rising Bdma data completed(0)

E\_CMDQ\_TRIGGER\_BUS\_INT\_BDMA\_1,///(2) Rising Bdma data completed(1)

E\_CMDQ\_TRIGGER\_BUS\_GPU2CMDQ\_TRIG,///(3) GPU completed MIU access and bridage is idle, then rise CMDQ trigger

E\_CMDQ\_TRIGGER\_BUS\_GOP\_CMDQ\_INT\_0,///(4) GOP0 destination VDE falling edge, GOP0 frame start point, double buffer register.

E\_CMDQ\_TRIGGER\_BUS\_GOP\_CMDQ\_INT\_1,///(5) GOP1 destination VDE falling edge, GOP1 frame start point, double buffer register.

E\_CMDQ\_TRIGGER\_BUS\_GOP\_CMDQ\_INT\_2,///(6) GOP2 destination VDE falling edge, GOP2 frame start point, double buffer register.

E\_CMDQ\_TRIGGER\_BUS\_GOP\_CMDQ\_INT\_3,///(7) GOP3 destination VDE falling edge, GOP3 frame start point, double buffer register.

E\_CMDQ\_TRIGGER\_BUS\_SC2CMDQ\_TRIG\_0,///(8) Sub2 IP vsync/vfde rising or falling, Scaler recieve video start or end.

E\_CMDQ\_TRIGGER\_BUS\_SC2CMDQ\_TRIG\_1,///(9) Sub IP vsync/vfde rising or falling, Scaler recieve video start or end.

E\_CMDQ\_TRIGGER\_BUS\_SC2CMDQ\_TRIG\_2,///(10) Main IP vsync/vfde rising or falling, Scaler recieve video start or end.

E\_CMDQ\_TRIGGER\_BUS\_SC2CMDQ\_TRIG\_3,///(11) OP vsync/vfde rising or falling, Scaler sending video to display start or end.

E\_CMDQ\_TRIGGER\_BUS\_AESDMA\_FINISHED,///(12) AESDMA done.

E\_CMDQ\_TRIGGER\_BUS\_DVB\_VDE,///(13) (1st) Main MVOP output Vertical Frame DE

E\_CMDQ\_TRIGGER\_BUS\_DVBSUB\_VDE,///(14)(2nd) Sub MVOP output Vertical Frame DE

E\_CMDQ\_TRIGGER\_BUS\_UNUSED,///(15) unsed

}EN\_CMDQ\_TRIGGER\_BUS\_ID;

**7 EN\_CMDQ\_STAGE\_TYPE**

//define multi window command belongs to which stage

typedef enum

{

E\_CMDQ\_STAGE\_CLK\_ON,///(0)Open mvop/XC clock

E\_CMDQ\_STAGE\_SW\_RESET,///(1)do multi-window reset

E\_CMDQ\_STAGE\_ADL,///(2)auto\_download,restore pre-sram

E\_CMDQ\_STAGE\_LDR,///(3)load register,include MVOP,restore pre-register

E\_CMDQ\_STAGE\_WAIT\_ADL\_DONE,///(4)Wait auto\_download done

E\_CMDQ\_STAGE\_WAIT\_LDR\_DONE,///(5)Wait load register done

E\_CMDQ\_STAGE\_NEXT\_FRAME\_SETTING,///(6)Load pq adaptive register by mload

E\_CMDQ\_STAGE\_LOAD\_NEW\_SETTING,///(7)Load new setting register by DS

E\_CMDQ\_STAGE\_FIRE\_TGEN,///(8)video processing start

E\_CMDQ\_STAGE\_WAIT\_WINDOW\_DONE,///(9)window done,output buffer have new data

E\_CMDQ\_STAGE\_AUL,///(1)auto\_upload,store SRAM status

E\_CMDQ\_STAGE\_STR,///(11)store register

E\_CMDQ\_STAGE\_WAIT\_AUL\_DONE,///(12)Wait auto\_upload done

E\_CMDQ\_STAGE\_WAIT\_STR\_DONE,///(13)Wait store register done

E\_CMDQ\_STAGE\_CLK\_OFF,///(14)Close MVOP and XC clock

E\_CMDQ\_STAGE\_WAIT\_OP\_VSYNC,///(15)Wait next op vsync

E\_CMDQ\_STAGE\_MAX,///(16)

}EN\_CMDQ\_STAGE\_TYPE;

## 结构体定义

### 外部使用的结构体定义

|  |  |
| --- | --- |
| 数据结构体名称 | 说明 |
| ST\_XC\_CMDQ\_INIT | CMDQ init的传入参数 |

**1 ST\_XC\_CMDQ\_INIT**

//define CMDQ init parameters

typedef struct

{

MS\_U16 u16DataInfo\_Length; ///<Length of this structure, u16Length=sizeof(ST\_XC\_CMDQ\_INIT\_PARAM)

MS\_PHY u32BufferStartAddress;

MS\_U32 u32BufferSize;

}ST\_XC\_CMDQ\_INIT;

### 内部使用结构体定义

|  |  |
| --- | --- |
| 数据结构体名称 | 说明 |
| ST\_CMDQ\_COMMAND | Basic mode下command的定义 |
| ST\_CMDQ\_MULTI\_WINDOW\_COMMAND | Multi-window下command的定义 |
| ST\_CMDQ\_STAGE\_COMMAND\_SET | Multi window下各个stage的cmd set |
| ST\_CMDQ\_WINDOW\_COMMAND\_SET | 一个window的一个frame的cmd set |

**1 ST\_CMDQ\_COMMAND**

//define CMDQ command info

typedef struct DLL\_PACKED

{

EN\_CMDQ\_COMMAND\_TYPE enCommandType;/// command type:write,wait,polling\_eq,polling\_neq,null

MS\_U32 u32Bank;/// bank address

MS\_U32 u32Addr;/// 16 bit register address

MS\_U16 u16Value;/// value

MS\_U16 u16Mask;/// mask

}ST\_CMDQ\_COMMAND;

**2 ST\_CMDQ\_MULTI\_WINDOW\_COMMAND**

//define CMDQ multi-window command info

typedef struct DLL\_PACKED

{

SCALER\_WIN enWindow;/// window id

EN\_CMDQ\_STAGE\_TYPE enStageType;//window stage type

ST\_CMDQ\_COMMAND stCommand;/// command

}ST\_CMDQ\_MULTI\_WINDOW\_COMMAND;

**3 ST\_CMDQ\_STAGE\_COMMAND\_SET**

typedef struct DLL\_PACKED

{

MS\_U32 u32CmdNum;///command number

MS\_PHY u32StartAddrPtr;///physical address : command start address pointer

}ST\_CMDQ\_STAGE\_COMMAND\_SET;

**4 ST\_CMDQ\_WINDOW\_COMMAND\_SET**

//define set window order parameter

typedef struct DLL\_PACKED

{

SCALER\_WIN enWindow;///(in)scaler window id

ST\_CMDQ\_STAGE\_COMMAND\_SET astStageCmdSet[E\_CMDQ\_STAGE\_MAX];

}ST\_CMDQ\_WINDOW\_COMMAND\_SET;

## 接口定义

### 外部使用接口定义

|  |  |
| --- | --- |
| 接口名称 | 说明 |
| MApi\_XC\_CMDQ\_Init | 初始化CMDQ HW。 |
| MApi\_XC\_CMDQ\_DeInit | 反初始化CMDQ |

**1 E\_APIXC\_ReturnValue SYMBOL\_WEAK MApi\_XC\_CMDQ\_Init(EN\_XC\_CMDQ\_ID enCMDQID, ST\_XC\_CMDQ\_INIT\* pstParams)**

**功能说明：**

初始化CMDQ模块。

**入参说明：**

enCMDQID：CMDQ ID，表示使用哪套CMDQ HW，目前Scaler这边只有为multi-window增加的这套CMDQ HW，因此，该参数的值为E\_XC\_CMDQ\_MULTI\_WINDOW。如果后面有增加CMDQ的数量，可以扩展这个变量的传入参数值。

pstParams->u32StartAddress：CMDQ buffer的起始地址。

pstParams->u32BufferSize：CMDQ buffer的大小。

**出参说明：**

E\_APIXC\_ReturnValue 返回执行该接口的结果。



Figure 15 set CMDQ buffer flow

**2 E\_APIXC\_ReturnValue SYMBOL\_WEAK MApi\_XC\_CMDQ\_DeInit(EN\_XC\_CMDQ\_ID enCMDQID)**

**功能说明：**

CMDQ模块反初始化。

**入参说明：**

enCMDQID：CMDQ ID，表示使用哪套CMDQ HW，目前Scaler这边只有为multi-window增加的这套CMDQ HW，因此，该参数的值为E\_XC\_CMDQ\_MULTI\_WINDOW。如果后面有增加CMDQ的数量，可以扩展这个变量的传入参数值。

**出参说明：**

E\_APIXC\_ReturnValue 返回执行该接口的结果。

### 内部使用接口定义

|  |  |
| --- | --- |
| 接口名称 | 说明 |
| MDrv\_CMDQ\_SetDMAMode | 设定CMDQ的DMA mode |
| MDrv\_CMDQ\_GetDMAMode | 获取CMDQ当前的DMA mode |
| MDrv\_CMDQ\_SetWaitTime | 设置CMDQ 轮询等待时间 |
| MDrv\_CMDQ\_GetWaitTime | 获取CMDQ 轮询等待时间 |
| MDrv\_CMDQ\_SetTimerRatio | 设置CMDQ 轮询计时器比 |
| MDrv\_CMDQ\_GetTimerRatio | 获取CMDQ 轮询计时器比 |
| MDrv\_CMDQ\_SetCommandAddress | 设置basic mode下，DMA mode为direct mode或increment mode时command的起始地址。 |
| MDrv\_CMDQ\_GetCommandAddress | 获取basic mode下，DMA mode为direct mode或increment mode时添加的command的起始地址。 |
| MDrv\_CMDQ\_SetWritePointer | 设置ring buffer mode下command的写指针。 |
| MDrv\_CMDQ\_GetWritePointer | 获取ring buffer mode下command的写指针。 |
| MDrv\_CMDQ\_GetWaitTriggerBusCommand | 获取等待trigger信号的command |
| MDrv\_CMDQ\_GetStatus | 获取CMDQ的状态 |
| MDrv\_CMDQ\_ResetInterrupt | 重置CMDQ的中断 |
| MDrv\_CMDQ\_AddCommand | 在basic mode下，添加command到CMDQ的memory中 |
| MDrv\_CMDQ\_SetWorkMode | 设置CMDQ的工作方式，CMDQ可以工作在basic mode和multi window mode |
| MDrv\_CMDQ\_GetWorkMode | 获取CMDQ当前的工作方式 |
| MDrv\_CMDQ\_SetWindowDone | Script模块通知CMDQ当前window的command已经全部添加完 |
| MDrv\_CMDQ\_SetFrameDone | FRC模块通知CMDQ当前frame需要造剧本的window都已经造完剧本，CMDQ可以将这些window的cmd set添加到command queue中 |
| MDrv\_CMDQ\_GetRunningQueueID | 获取当前正在执行的command queue的ID |
| MDrv\_CMDQ\_AddMultiWindowCommand | 在multi window mode下，添加command到CMDQ的memory中 |
| MDrv\_CMDQ\_DestroyWindow | 通知CMDQ某个window被destroy |
| MDrv\_ CMDQ\_Enable | 开关CMDQ |

**1 MS\_BOOL MDrv\_CMDQ\_SetDMAMode(void\* pInstance, EN\_XC\_CMDQ\_ID enCMDQID, EN\_CMDQ\_DMA\_MODE enDMAMode)**

**功能说明：**

设置CMDQ执行command的方式，basic mode下可以使用E\_XC\_CMDQ\_DMA\_DIRECT\_MODE, E\_XC\_CMDQ\_DMA\_INCREMENT\_MODE和E\_XC\_CMDQ\_DMA\_RING\_BUFFER\_MODE,但是multi-window mode固定使用E\_XC\_CMDQ\_DMA\_DIRECT\_MODE。

**入参说明：**

enCMDQID：CMDQ ID，表示使用哪套CMDQ HW，目前Scaler这边只有为multi-window增加的这套CMDQ HW，因此，该参数的值为E\_XC\_CMDQ\_MULTI\_WINDOW。如果后面有增加CMDQ的数量，可以扩展这个变量的传入参数值。

enDMAMode：CMDQ的DMA mode。

**返回值说明：**

TRUE：成功设置CMDQ的DMA mode，FALSE：设置CMDQ的DMA mode失败。



Figure 16 Set DMA mode

**2 MS\_BOOL MDrv\_CMDQ\_GetDMAMode(void\* pInstance, EN\_XC\_CMDQ\_ID enCMDQID, EN\_CMDQ\_DMA\_MODE\* penDMAMode)**

**功能说明：**

获取CMDQ的DMA mode。

**入参说明：**

enCMDQID：CMDQ ID，表示使用哪套CMDQ HW，目前Scaler这边只有为multi-window增加的这套CMDQ HW，因此，该参数的值为E\_XC\_CMDQ\_MULTI\_WINDOW。如果后面有增加CMDQ的数量，可以扩展这个变量的传入参数值。

**出参说明：**

penDMAMode：CMDQ的DMA mode，应该为direct mode，increment mode和ring-buffer mode中的一种。。

**返回值说明：**

TRUE：成功获取CMDQ的DMA mode，FALSE：获取CMDQ的DMA mode失败。

**3 MS\_BOOL MDrv\_CMDQ\_SetWaitTime(void\* pInstance, EN\_XC\_CMDQ\_ID enCMDQID, MS\_U16 u16Time)**

**功能说明：**

设置CMDQ的轮询等待时间poll\_wait\_time。相关接口MDrv\_CMDQ\_SetTimerRatio，实际的轮询等待时间为poll\_wait\_time \* (poll\_timer\_ratio + 1)。。

**入参说明：**

enCMDQID：CMDQ ID，表示使用哪套CMDQ HW，目前Scaler这边只有为multi-window增加的这套CMDQ HW，因此，该参数的值为E\_XC\_CMDQ\_MULTI\_WINDOW。如果后面有增加CMDQ的数量，可以扩展这个变量的传入参数值。

u16Time：CMDQ的轮询等待时间。

**返回值说明：**

TRUE：成功设置CMDQ的轮询等待时间，FALSE：设置CMDQ的轮询等待时间失败。

**4 MS\_BOOL MDrv\_CMDQ\_GetWaitTime(void\* pInstance, EN\_XC\_CMDQ\_ID enCMDQID, MS\_U16\* pu16Time)**

**功能说明：**

获取CMDQ的轮询等待时间。

**入参说明：**

enCMDQID：CMDQ ID，表示使用哪套CMDQ HW，目前Scaler这边只有为multi-window增加的这套CMDQ HW，因此，该参数的值为E\_XC\_CMDQ\_MULTI\_WINDOW。如果后面有增加CMDQ的数量，可以扩展这个变量的传入参数值。

**出参说明：**

pu16Time：获取到的CMDQ的轮询等待时间。

**出参说明：**

TRUE：成功获取CMDQ的轮询等待时间，FALSE：获取CMDQ的轮询等待时间失败。

**5 MS\_BOOL MDrv\_CMDQ\_SetTimerRatio(void\* pInstance, EN\_XC\_CMDQ\_ID enCMDQID, MS\_U16 u16Ratio)**

**功能说明：**

设置CMDQ的轮询计时器比poll\_timer\_ratio。相关接口MDrv\_CMDQ\_SetWaitTime，真正的轮询等待时间为poll\_wait\_time \* (poll\_timer\_ratio + 1)。

**入参说明：**

enCMDQID：CMDQ ID，表示使用哪套CMDQ HW，目前Scaler这边只有为multi-window增加的这套CMDQ HW，因此，该参数的值为E\_XC\_CMDQ\_MULTI\_WINDOW。如果后面有增加CMDQ的数量，可以扩展这个变量的传入参数值。

u16Ratio：CMDQ的轮询计时器比，取值范围0~127。

**返回值说明：**

TRUE：成功设置CMDQ的轮询计时器比，FALSE：设置CMDQ的轮询计时器比失败。

**6 MS\_BOOL MDrv\_CMDQ\_GetTimerRatio(void\* pInstance, EN\_XC\_CMDQ\_ID enCMDQID, MS\_U16\* pu16Ratio)**

**功能说明：**

获取CMDQ的轮询计时器比。

**入参说明：**

enCMDQID：CMDQ ID，表示使用哪套CMDQ HW，目前Scaler这边只有为multi-window增加的这套CMDQ HW，因此，该参数的值为E\_XC\_CMDQ\_MULTI\_WINDOW。如果后面有增加CMDQ的数量，可以扩展这个变量的传入参数值。

**出参说明：**

pu16Ratio：获取到的CMDQ的轮询计时器比。

**返回值说明：**

TRUE：成功获取CMDQ的轮询轮询计时器比，FALSE：获取CMDQ的轮询计时器比失败。

**7 MS\_BOOL MDrv\_CMDQ\_SetCommandAddress(void\* pInstance, EN\_XC\_CMDQ\_ID enCMDQID, MS\_PHY u32StartAddress, MS\_PHY u32EndAddress)**

**功能说明：**

在basic mode下，DMA mode为direct mode和increment mode时，用于设置CMDQ的起始地址。在direct mode下，如果CMDQ正在执行某个command set，如果多次设定和trigger CMDQ，只有最后一次设定的地址有效，因此，为了安全使用CMDQ，最好先查询CMDQ的状态为 idle是再去设置command的其实地址和trigger CMDQ。在increment mode下，如果正在执行的command set的地址与新设定的地址是连续的，多次设定和trigger CMDQ都有效，但是如果正在执行的command set的地址与新设定的地址不是连在一起的，而且CMDQ在busy状态的时候设置起始地址会失败。

**入参说明：**

enCMDQID：CMDQ ID，表示使用哪套CMDQ HW，目前Scaler这边只有为multi-window增加的这套CMDQ HW，因此，该参数的值为E\_XC\_CMDQ\_MULTI\_WINDOW。如果后面有增加CMDQ的数量，可以扩展这个变量的传入参数值。

u32StartAddress：设置的command的首地址。

u32EndAddress：设置的command的结束地址。

**返回值说明：**

TRUE：成功设置CMDQ的command的起始地址，FALSE：设置CMDQ的command的起始地址失败。

multi-window mode下不允许上层指定command写的地址，由utopia统一管理，在ring-buffer mode下，将写command的memory固定于设定的buffer相同，在direct mode和increment mode下才允许上层设定command的地址和command set的跳转。



Figure 17 set command address

**8 MS\_BOOL MDrv\_CMDQ\_GetCommandAddress(void\* pInstance, EN\_XC\_CMDQ\_ID enCMDQID, MS\_PHY\* pu32StartAddress, MS\_PHY\* pu32EndAddress)**

**功能说明：**

在basic mode下，DMA mode为direct mode和increment mode时，获取上次获取command的起始地址之后到目前为止添加的command的起始地址。为了使每次获取的地址都是安全有效的，会将地址按照MIU bus宽度对齐，在末尾添加NULL command。

**入参说明：**

enCMDQID：CMDQ ID，表示使用哪套CMDQ HW，目前Scaler这边只有为multi-window增加的这套CMDQ HW，因此，该参数的值为E\_XC\_CMDQ\_MULTI\_WINDOW。如果后面有增加CMDQ的数量，可以扩展这个变量的传入参数值。

**出参说明：**

pu32StartAddress：新添加的command的首地址。

pu32EndAddress：新添加的command的结束地址。

**返回值说明：**

TRUE：成功获取新添加到CMDQ的command的起始地址，FALSE：获取新添加到CMDQ的command的起始地址失败。

**9 MS\_BOOL MDrv\_CMDQ\_SetWritePointer(void\* pInstance, EN\_XC\_CMDQ\_ID enCMDQID, MS\_PHY u32WritePointer)**

**功能说明：**

在ring buffer mode下，设置command的写指针到CMDQ，CMDQ在被设置command的写指针并trigger之后，CMDQ开始执行command，当HW的读指针等于写指针的时候，CMDQ会停下来。

**入参说明：**

enCMDQID：CMDQ ID，表示使用哪套CMDQ HW，目前Scaler这边只有为multi-window增加的这套CMDQ HW，因此，该参数的值为E\_XC\_CMDQ\_MULTI\_WINDOW。如果后面有增加CMDQ的数量，可以扩展这个变量的传入参数值。

u32WritePointer：command的写指针。

**返回值说明：**

TRUE：成功设置command的写指针到CMDQ，FALSE：设置command的写指针到CMDQ失败。



Figure 18 set write pointer

**10 MS\_BOOL MDrv\_CMDQ\_GetWritePointer(void\* pInstance, EN\_XC\_CMDQ\_ID enCMDQID, MS\_PHY\* pu32WritePointer)**

**功能说明：**

在ring buffer mode下，获取当前添加command的写指针。

**入参说明：**

enCMDQID：CMDQ ID，表示使用哪套CMDQ HW，目前Scaler这边只有为multi-window增加的这套CMDQ HW，因此，该参数的值为E\_XC\_CMDQ\_MULTI\_WINDOW。如果后面有增加CMDQ的数量，可以扩展这个变量的传入参数值。

**出参说明：**

pu32WritePointer：传回当前添加command的写指针。

**返回值说明：**

TRUE：成功获取command的写指针，FALSE：获取command的写指针失败。

**11 MS\_BOOL MDrv\_CMDQ\_GetWaitTriggerBusCommand(EN\_XC\_CMDQ\_ID enCMDQID, EN\_CMDQ\_TRIGGER\_BUS\_ID enTriggerBusId, MS\_BOOL bRisingEdge, ST\_CMDQ\_COMMAND\* pstCmd);**

**功能说明：**

获取指定信号的对应的command，其他module在需要等待这些trigger信号时，并不需要command的具体格式，只需要知道trigger bus的ID，就可以从CMDQ直接获取该command，然后将command添加到相应的位置即可。

**入参说明：**

enCMDQID：CMDQ ID，表示使用哪套CMDQ HW，目前Scaler这边只有为multi-window增加的这套CMDQ HW，因此，该参数的值为E\_XC\_CMDQ\_MULTI\_WINDOW。如果后面有增加CMDQ的数量，可以扩展这个变量的传入参数值。

enTriggerBusId：需要等待的信号的trigger bus ID。

bRisingEdge：指定等待信号的上升沿还是下降沿。

**出参说明：**

pstCmd：传回指定的信号的command。

**返回值说明：**

TRUE：成功获取指定信号对应的command，FALSE：获取指定信号对应的command失败。

**12 MS\_BOOL MDrv\_CMDQ\_ResetInterrupt(void\* pInstance, EN\_XC\_CMDQ\_ID enCMDQID)**

**功能说明：**

重置CMDQ的中断。

**入参说明：**

enCMDQID：CMDQ ID，表示使用哪套CMDQ HW，目前Scaler这边只有为multi-window增加的这套CMDQ HW，因此，该参数的值为E\_XC\_CMDQ\_MULTI\_WINDOW。如果后面有增加CMDQ的数量，可以扩展这个变量的传入参数值。

**返回值说明：**

TRUE：成功重置CMDQ的中断，FALSE：重置CMDQ的中断失败。

**13 MS\_BOOL MDrv\_CMDQ\_GetStatus(void\* pInstance, EN\_XC\_CMDQ\_ID enCMDQID, EN\_CMDQ\_STATUS\* penStatus)**

**功能说明：**

获取CMDQ的状态，在basic mode下该函数的返回CMDQ的真实状态。在multi window mode，如果CMDQ被开启之后，传回的状态一直为busy，直到CMDQ被disable。

**入参说明：**

enCMDQID：CMDQ ID，表示使用哪套CMDQ HW，目前Scaler这边只有为multi-window增加的这套CMDQ HW，因此，该参数的值为E\_XC\_CMDQ\_MULTI\_WINDOW。如果后面有增加CMDQ的数量，可以扩展这个变量的传入参数值。

**出参说明：**

**penStatus**：传回CMDQ的状态。

**返回值说明：**

TRUE：成功获取CMDQ的状态，FALSE：获取CMDQ的状态失败。

**14 MS\_BOOL MDrv\_CMDQ\_AddCommand(void\* pInstance, EN\_XC\_CMDQ\_ID enCMDQID, MS\_U32 u32CmdNum, ST\_CMDQ\_COMMAND\* pstCmd)**

**功能说明：**

CMDQ工作在basic mode时，添加command到CMDQ。

**入参说明：**

enCMDQID：CMDQ ID，表示使用哪套CMDQ HW，目前Scaler这边只有为multi-window增加的这套CMDQ HW，因此，该参数的值为E\_XC\_CMDQ\_MULTI\_WINDOW。如果后面有增加CMDQ的数量，可以扩展这个变量的传入参数值。

pstCmd：添加的command参数。

**返回值说明：**

TRUE：成功获取CMDQ的状态，FALSE：获取CMDQ的状态失败。



Figure 19 Add command in basic mode

**15 MS\_BOOL MDrv\_CMDQ\_SetWorkMode(void\* pInstance, EN\_XC\_CMDQ\_ID enCMDQID, EN\_CMDQ\_WORK\_MODE enWorkMode)**

**功能说明：**

设置CMDQ的工作方式，该接口的主要flow如图20所示。

**入参说明：**

enCMDQID：CMDQ ID，表示使用哪套CMDQ HW，目前Scaler这边只有为multi-window增加的这套CMDQ HW，因此，该参数的值为E\_XC\_CMDQ\_MULTI\_WINDOW。如果后面有增加CMDQ的数量，可以扩展这个变量的传入参数值。

enWorkMode：CMDQ的工作方式，当CMDQ工作在basic mode时，其他module才能自由的使用CMDQ，当CMDQ工作在multi window mode时，CMDQ这套HW会被multi window一直使用，添加command时必须按照按照window和stage添加才能被执行。

**返回值说明：**

TRUE：成功设置CMDQ的工作方式，FALSE：设置CMDQ的工作方式失败。



Figure 20 set CMDQ work mode

**16 MS\_BOOL MDrv\_CMDQ\_GetWorkMode(void\* pInstance, EN\_XC\_CMDQ\_ID enCMDQID, EN\_CMDQ\_WORK\_MODE\* penWorkMode)**

**功能说明：**

获取CMDQ的工作方式。

**入参说明：**

enCMDQID：CMDQ ID，表示使用哪套CMDQ HW，目前Scaler这边只有为multi-window增加的这套CMDQ HW，因此，该参数的值为E\_XC\_CMDQ\_MULTI\_WINDOW。如果后面有增加CMDQ的数量，可以扩展这个变量的传入参数值。

**出参说明：**

penWorkMode：传回CMDQ当前的工作方式。

**返回值说明：**

TRUE：成功获取CMDQ的工作方式，FALSE：获取CMDQ的工作方式失败。

**17 MS\_BOOL MDrv\_CMDQ\_SetWindowDone(void\* pInstance, EN\_XC\_CMDQ\_ID enCMDQID, SCALER\_WIN enWindow, MS\_U32\* pu32QueueID)**

**功能说明：**

CMDQ工作在multi window mode下，通知CMDQ当前window的command已经全部添加完成。

**入参说明：**

enCMDQID：CMDQ ID，表示使用哪套CMDQ HW，目前Scaler这边只有为multi-window增加的这套CMDQ HW，因此，该参数的值为E\_XC\_CMDQ\_MULTI\_WINDOW。如果后面有增加CMDQ的数量，可以扩展这个变量的传入参数值。

SCALER\_WIN：当前已经添加完command的window ID。

**出参说明：**

pu32QueueID：传回当前更新的command queue的ID。

**返回值说明：**

TRUE：成功通知CMDQ的结束添加该window的command，FALSE：通知CMDQ失败。

**18 MS\_BOOL MDrv\_CMDQ\_SetFrameDone(void\* pInstance, EN\_XC\_CMDQ\_ID enCMDQID, MS\_U32 u32WinNum, SCALER\_WIN\* penWindow)**

**功能说明：**

通知CMDQ当前frame中需要造剧本的所有window的command都已经添加完。CMDQ会将这个frame的所有window的command set按照layer排序后更新到command queue中，该接口的主要flow如图21所示。

**入参说明：**

enCMDQID：CMDQ ID，表示使用哪套CMDQ HW，目前Scaler这边只有为multi-window增加的这套CMDQ HW，因此，该参数的值为E\_XC\_CMDQ\_MULTI\_WINDOW。如果后面有增加CMDQ的数量，可以扩展这个变量的传入参数值。

u32WinNum：当前frame要造剧本的window的数量。

penWindow：当前frame要造剧本的window ID的数组其实地址。

**返回值说明：**

TRUE：成功通知CMDQ更新command queue，FALSE：CMDQ更新command queue失败。

用于通知CMDQ这个OP Vsync的所有window的cmd set已经全部添加完成，CMDQ将临时cmd set链表中的节点按照window layer进行排序，再将这些节点添加到command queue中。



Figure 21 set frame done

**19 MS\_BOOL MDrv\_CMDQ\_GetRunningQueueID(void\* pInstance, EN\_XC\_CMDQ\_ID enCMDQID,MS\_U32\* pu32QueueID)**

**功能说明：**

获取CMDQ正在执行的command queue的ID，但是我们无法直接知道这个ID，只能从register获取当前正在执行的command的address之后，再去查找到该address所在的command queue的ID，最后返回下一个将要执行的command queue的ID。该接口的主要flow如图22所示。

**入参说明：**

enCMDQID：CMDQ ID，表示使用哪套CMDQ HW，目前Scaler这边只有为multi-window增加的这套CMDQ HW，因此，该参数的值为E\_XC\_CMDQ\_MULTI\_WINDOW。如果后面有增加CMDQ的数量，可以扩展这个变量的传入参数值。

**出参说明：**

pu32QueueID：传回CMDQ正在执行的command queue的ID。

**返回值说明：**

TRUE：成功获取CMDQ正在执行的command queue的ID，FALSE：获取CMDQ正在执行的command queue的ID失败。



Figure 22 get running queue ID

**20 MS\_BOOL MDrv\_CMDQ\_AddMultiWindowCommand(void\* pInstance, EN\_XC\_CMDQ\_ID enCMDQID, MS\_U32 u32CmdNum, EN\_CMDQ\_ADD\_COMMAND\_MODE enAddCmdMode, ST\_CMDQ\_MULTI\_WINDOW\_COMMAND\* pstCmd)**

**功能说明：**

CMDQ工作在multi window下，向CMDQ添加command或者更新下一个OP Vsync要执行的command queue的command。该接口的主要flow如图23所示。

**入参说明：**

enCMDQID：CMDQ ID，表示使用哪套CMDQ HW，目前Scaler这边只有为multi-window增加的这套CMDQ HW，因此，该参数的值为E\_XC\_CMDQ\_MULTI\_WINDOW。如果后面有增加CMDQ的数量，可以扩展这个变量的传入参数值。

u32CmdNum：添加的command的数量。

enAddCmdMode：添加command的方式，可以用于指定是正常添加下一个frame的command，或是指定更新下个OP Vsync要执行的command。

pstCmd：要添加的command的指针。

**返回值说明：**

TRUE：成功添加command，FALSE：添加command失败。



Figure 23 add command in multi-window mode

**21 MS\_BOOL MDrv\_CMDQ\_DistroyWindow(void\* pInstance, EN\_XC\_CMDQ\_ID enCMDQID, SCALER\_WIN enwindow)**

**功能说明：**

通知CMDQ某个window被销毁，CMDQ将移除该window还未添加到command queue的command set节点。

**入参说明：**

enCMDQID：CMDQ ID，表示使用哪套CMDQ HW，目前Scaler这边只有为multi-window增加的这套CMDQ HW，因此，该参数的值为E\_XC\_CMDQ\_MULTI\_WINDOW。如果后面有增加CMDQ的数量，可以扩展这个变量的传入参数值。

enwindow：被销毁的window的ID。

**返回值说明：**

TRUE：成功删除被销毁的window尚未加入command queue的command set节点，FALSE：删除该window未加入command queue的节点失败。

**22 MS\_BOOL MDrv\_CMDQ\_Enable(void\* pInstance, EN\_XC\_CMDQ\_ID enCMDQID, MS\_BOOL bEnable)**

**功能说明：**

开启或关闭CMDQ，CMDQ工作在不同的DMA mode下时，enable的设定也不同，在3种DMA mode下都需要设定DMA read enable，此外，在direct mode下，需要设定command的start/end address和地址信息变化的trigger信号；在increment mode下，如果command的start address没有变化，则只要设定offset变化的trigger信号，如果start address也有变化，需要设定地址和offset变化的两个trigger点，要先设定start address变化的trigger点，再设置offset变化的trigger点；在ring-buffer mode下，只需要设定SW写指针和trigger信号。该接口的主要flow如图24所示。

**入参说明：**

enCMDQID：CMDQ ID，表示使用哪套CMDQ HW，目前Scaler这边只有为multi-window增加的这套CMDQ HW，因此，该参数的值为E\_XC\_CMDQ\_MULTI\_WINDOW。如果后面有增加CMDQ的数量，可以扩展这个变量的传入参数值。

bEnable：start CMDQ时bEnable传TRUE，disable CMDQ时bEnable传FALSE。

**出参说明：**

TRUE成功开启CMDQ，打开CMDQ失败。



Figure 24 enable/disable CMDQ flow