VPU Reference Software

User's Guide

Version 1.0.0

VPU Reference Software: User's Guide

Version 1.0.0

Revision History

Date	Revision	Change
2019/4/10	0.9.0	Initial version
2019/4/17	1.0.0	Release version generated

Proprietary Notice

Address and Phone Number

Chips&Media
V&S Tower, 11/12/13th FL, 891-46, Daechi-dong, Gangnam-gu, 135-280
Seoul, Korea
Tel: +82-2-568-3767
Fax: +82-2-568-3767

Homepage: *http://www.chipsnmedia.com*

Table of Contents

List of Figures

List of Tables

Preface

This preface introduces the VPU Reference Software User's Guide and its reference documentation. It contains the following sections:

1. About this document

This document is the user's guide of VPU Reference Software also called sample_v2.

1.1. Intended audience

This document has been written for application engineers who want to install the VPU reference software package, develop, and verify multimedia applications by using VPU reference software Components.

1.2. Document Scope

This document introduces the VPU reference software package provided by Chips&Media. It details key concepts of component-based architecture applied for the encode and decode sample and explains each of the sample components.

1.3. Typographical Conventions

The following typographical conventions are used in this document:

bold	Highlights signal names within text, and interface elements such as menu names. May also be used for emphasis in descriptive lists where appropriate.
italic	Highlights cross-references in blue, file names, and citations.
`typewriter`	Denotes example source codes and dumped character or text.

2. Further Reading

This section lists documents which are related to this product.

2.1. Other Documents

- *VPU API Reference Manual*

Chapter 1
Overview

We provide customer with reference software called `sample_v2`, which is an implementation of encode and decode software component running on VPU(Video Processing Unit). The VPU reference software realizes a relatively high level of abstraction by adopting some part of the OpenMAX component structure and abstracting major parts of the sample program. It separates the VPU control code from other test code, making it easier for users to understand what the VPU controls are.

Components support not only thread, but also the way that each component is executed in a round-robin fashion without threads. User can test the multi-instance functionality of the VPU in environments that do not support threads. Using the sample components, user can simply test the basic encode/decode operation of VPU, add new test codes, or implement video application program.

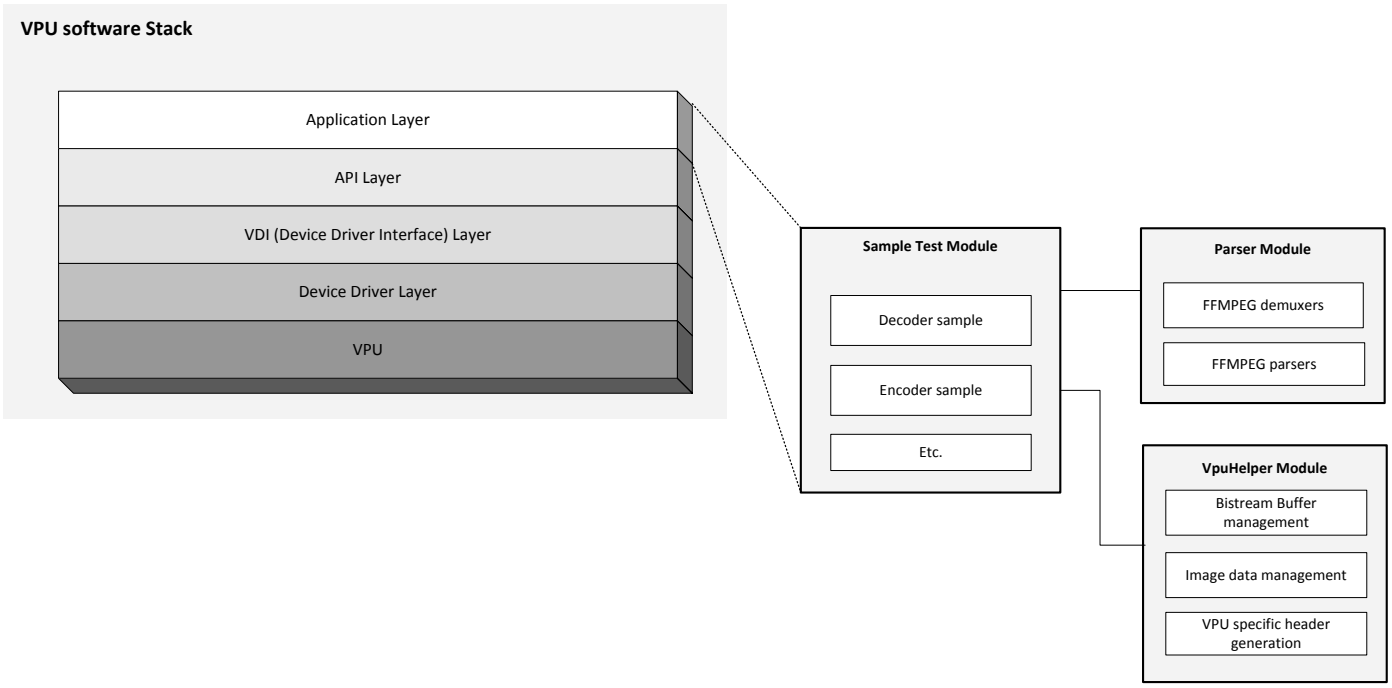

Figure 1.1. Reference Software Structure

The VPU reference software is placed in the `sample_v2` directory of the ReferenceSWpackage we provide. It includes component source and header files, enoder/decoder sample applications, and other stuff files.

```
./component
./component_decoder
./component_encoder
./helper
component_list_all.h
component_list_decoder.h
component_list_encoder.h
main_coda960_dec_test.c
main_coda960_enc_test.c
main_coda980_dec_test.c
main_coda980_enc_test.c
main_dec_test.c
main_enc_test.c
```

component
Component core source files

component_decoder
Decoder components source files

component_encoder
Encoder components source files

helper
A collection of functions needed to handle YUV and bitstream and to encode/decode

component_list_all.h
A group of structure variables for decoder and encoder components

component_list_decoder.h
A group of structure variables for decoder component

component_list_encoder.h
A group of structure variables for encoder component

main_coda960_dec_test.c
Decoder sample program for CODA960 IP

main_coda960_enc_test.c
Encoder sample program for CODA960 IP

main_coda980_dec_test.c
Decoder sample program for CODA980 IP

main_coda_980_enc_test.c
Encoder sample program for CODA980 IP

main_dec_test.c
Decoder sample program for WAVE5 series IP

main_enc_test.c
Encoder sample program for WAVE5 series IP

Chapter 2
Reference Software Architecture

In this chapter, we describe key vocabulary used for the sample application such as CNMApp, CNMTask, and the important functions. It also introduces component structure, specific component variables and methods, and how to write and register a component.

2.1. CNMApp

CNMApp is an object that manages CNMTask instances.

2.1.1. Important Functions

void CNMAppInit(void)
Initializes internal variables that manage CNMTask.

void CNMAppAdd(CNMTask task)
Registers CNMTask instance.

void CNMAppRun(void)
Runs the registered CNMTask instances. It is blocked until all the CNMTask instances are terminated.

void CNMAppStop(void)
Stops all running CNMTask instances and unblocks CNMAppRun(). If you need to terminate a CNMApp under certain conditions, you can call it.

2.1.2. Example Code

The following example code demonstrates the basic usage of CNMApp functions.

```
int main(void)
{
    CNMTask decoder;
    CNMTask encoder;

    decoder = CNMTaskCreate();
    decoder->Add(Component_Create(...)); // Source component
    decoder->Add(Component_Create(...)); // Filter component
    decoder->Add(Component_Create(...)); // Sink copmonent

    encoder= CNMTaskCreate();
    encoder->Add(Component_Create(...)); // Source component
    encoder->Add(Component_Create(...)); // Filter component
    encoder->Add(Component_Create(...)); // Sink copmonent

    CNMAppInit();
    CNMAppAdd(decoder);
    CNMAppAdd(encoder);

    return CNMAppRun();
}
```

2.2. CNMTask

CNMTask stands for a work unit consisting of one or more components.

2.2.1. Important Functions

CNMTask CNMTaskCreate(void)
Creates a task instance.

void CNMTaskAdd(CNMTask task, Component component)
Registers the component instances in the CNMTask. Registered component instances are identified in the order of registration:

- source component
- filter component
- sink component
- isolated component

The registered components are connected to each other by task, and data are delivered from source component to sink component. This is called tunneling. More details on the Component are covered in the *Section 2.3, "Component"* chapter below.

2.3. Component

A component is a functional component required to perform a specific task. A component is divided into four types according to the connection method with other components.

2.3.1. Component Type

2.3.1.1. Source Component

A Source Component is a component that generates data by itself without any input from other components and provides data to the component connected to the output. In *Figure 2.1, "Source Component"*, the Component A becomes the source component.

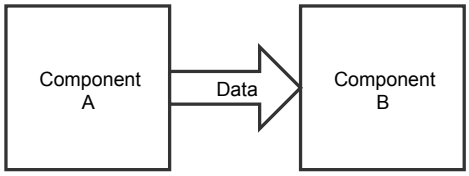

Figure 2.1. Source Component

There is only one source component among the connected components.

2.3.1.2. Sink Component

A Sink Component is responsible for receiving, processing, and destroying data from the component connected to the input. The Component B is a sink component in *Figure 2.1, "Source Component"*.

2.3.1.3. Filter Component

A Filter Component is a component that receives input data and provides data to the component connected to the output after processing. In *Figure 2.2, "Filter Component"*, the Component B and Component C are filter components.

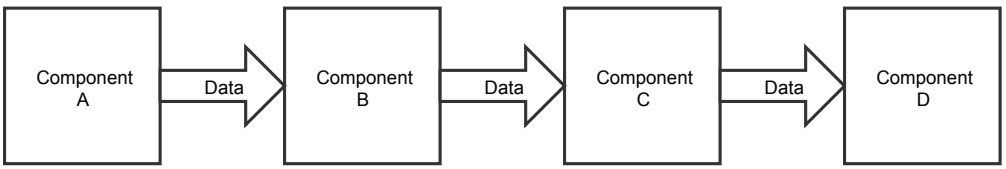

Figure 2.2. Filter Component

Among the many connected components, all are filter components except for the source component and sink component.

2.3.1.4. Isolated Component

An Isolated Component refers to a component that is not connected to any other components.

2.3.2. Component Class

The Component structure is represented by borrowing a class from C++.

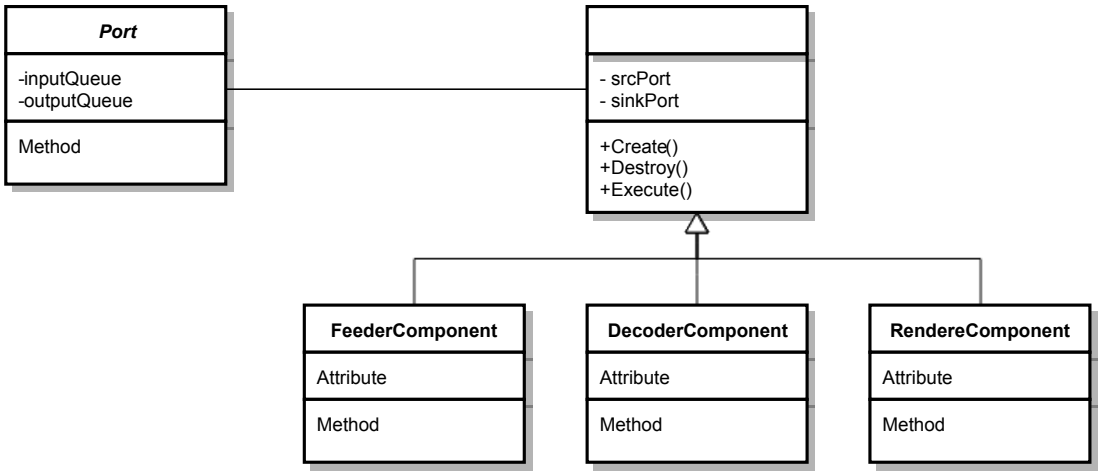

Figure 2.3. Component Class

2.3.3. Component Structure

The code below is a component structure. The function pointer acts as a member method of the class.

```
typedef void*   Component;
typedef struct ComponentImpl {
  char*                 name;
  void*                 context;
  Port                  sinkPort;
  Port                  srcPort;
  Uint32                containerSize;
  Uint32                numSinkPortQueue;
  Component             (*Create)(struct ComponentImpl*, CNMComponentConfig*);
  CNMComponentParamRet (*GetParameter)(struct ComponentImpl*, struct ComponentImpl*, GetParameterCMD, void*);
  CNMComponentParamRet (*SetParameter)(struct ComponentImpl*, struct ComponentImpl*, SetParameterCMD, void*);
  BOOL                  (*Prepare)(struct ComponentImpl*, BOOL*);
  BOOL                  (*Execute)(struct ComponentImpl*, PortContainer*, PortContainer*);
  void                  (*Release)(struct ComponentImpl*);
  BOOL                  (*Destroy)(struct ComponentImpl*);
  BOOL                  success;
  osal_thread_t         thread;
  ComponentState        state;
  BOOL                  terminate;
  ComponentListener     listeners[MAX_NUM_LISTENERS];
  Uint32                numListeners;
  Queue*                usingQ;  /*<<! NOTE: DO NOT USE Enqueue() AND Dequeue() IN YOUR COMPONENT.
                                                    BUT, YOU CAN USE Peek() FUNCTION.
                                 */
  CNMComponentType      type;
  Uint32                updateTime;
  Uint32                Hz; /*Use clock signal ex) 30Hz, A component will receive 30 clock signals per second.*/
  void*                 internalData;
  BOOL                  pause;
  BOOL                  portFlush;
} ComponentImpl;
```

2.3.3.1. Member variables of Component

Table 2.1. Member variables of Component

Member variables	Description
`char* name`	The name of component It must be unique.
`void* context`	Pointer to the internal context variable which is created by `ComponentImpl::Create()`
`Port sinkPort`	The port used to output data. When a component is created, sinkPort is created together.
`Port srcPort`	The port pointing to the sink pointer of the source component when the components are tunneled together.
`Uint32 containerSize`	The size of container stored in the sink port
`Uint32 numSinkPortQueue`	The number of containers that can be stored in the sink port
`BOOL success`	TRUE - the component has finished successfully. FALSE - the component has failed.
`osal_thread_t thread`	Thread handle pointer obtained when each component is thread based running

`ComponentState state`	Information about the status of component It can have one of the followings:
	
	`CNM_COMPONENT_STATE_CREATED`
	`CNM_COMPONENT_PREPARED`
	`CNM_COMPONENT_EXECUTED`

Member variables	Description
	CNM_COMPONENT_TERMINATED
BOOL terminate	TRUE - The component to terminate has been indicated.
ComponentListener listeners[MAX_NUM_LISTENERS]	An array containing the listeners registered via ComponentRegisterListener()
Queue* usingQ	This is a temporary storage area for the PortContainer in the internal operation of component.
CNMComponentType type	Represents a component role among connected components. CNM_COMPONENT_TYPE_ISOLATION - plays a role that has not been connected with any other component. CNM_COMPONENT_TYPE_SOURCE - plays a source component role. CNM_COMPONENT_TYPE_FILTER - plays a filter component role. CNM_COMPONENT_TYPE_SINK - plays a sink component role.
Uint32 updateTime	Time information used inside the component
Uint32 Hz	Determines how many times per second time information will be given. If Hz is 1, PortContainerClock comes in every second in the second argument of Component::Execute().
void* internalData	Data used inside the component
BOOL pause	TRUE - indicates pause status. It is used for internal use.
BOOL portFlush	TRUE - flushes the source port. It is used for internal use.

2.3.3.2. Member methods of Component

Table 2.2. Member methods of Component

Member methods	Description
Component (*Create)(struct ComponentImpl*, CNMComponentConfig*)	The member method, which acts as a constructor, creates the resources needed for the component to work.
CNMComponentParamRet (*GetParameter)(struct ComponentImpl* from, struct ComponentImpl* to, GetParameterCMD cmd, void* val)	• from: indicates the component that calls GetParameter member method. • to: indicates the component itself. • cmd: indicates the GetParameter which is defined in GetParameterCMD of component.h. • val: space to store appropriate data for the cmd
CNMComponentParamRet (*SetParameter)(struct ComponentImpl* from, struct ComponentImpl* to, SetParameterCMD cmd, void*val)	• from: indicates the component that calls GetParameter member method. • to: indicates the component itself. • cmd: the passed command • val: the argument of the passeed command
BOOL (*Prepare)(struct ComponentImpl*, BOOL*)	Defines the necessary operations before the Excute () method is executed. • ComponentImpl*: points to the component itself. • BOOL*: a pointer to the end of the operation. At the end of the operation, assign TRUE to the value of the pointer, otherwise FALSE. The caller who called the Prepare () method should call it repeatedly until the value of the second argument is TRUE. If an error occurs during the Prepare() process, you should return FALSE.
BOOL (*Execute)(struct ComponentImpl*, PortContainer*, PortContainer*)	Indicates a function that defines what the Component should do.

Member methods	Description
	• `ComponentImpl*`: points to the component itself. • `PortContainer*`: a pointer to a `PortContainer` structure with input data. When extracting data from this pointer structure, you must change the reuse member variable to FALSE. Otherwise, the same data will continue to be used when the next `Execute()` is called. • `PortContainer*`: a pointer to a `PortContainer` that will be passed to the sink component after processing the input data and saving them. Likewise, if you store and pass data to this structure, you must change the `reuse` variable to FALSE. Otherwise, it is not delivered to the sink component.
`void (*Release)(struct ComponentImpl*)`	Indicates a function that defines the task of releasing allocated resources.
`BOOL (*Destroy)(struct ComponentImpl*)`	Indicates the function that defines the task of releasing the internal context information. Do not free the memory of the first argument.

2.3.3.3. Important Functions(Abstract layer functions)

2.3.3.3.1. ComponentCreate()

This function creates an instance of the component registered with the first argument componentName. The second argument is passed when the instance is created.

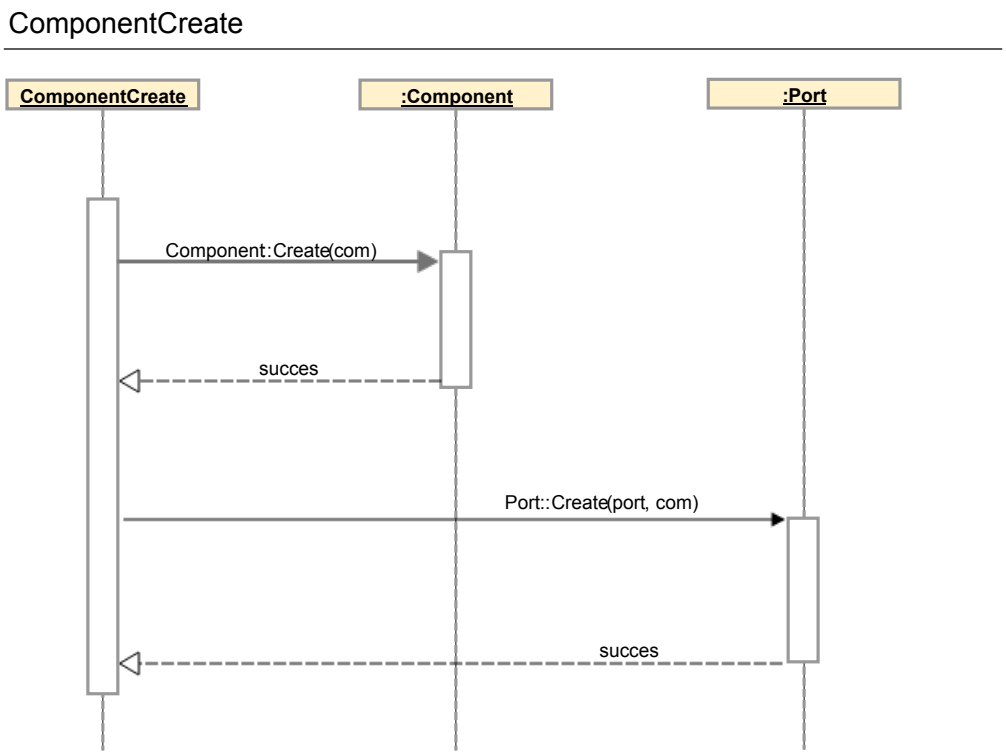

Figure 2.4. ComponentCreate

2.3.3.3.2. ComponentExecute()

This function is responsible for running component::Execute().

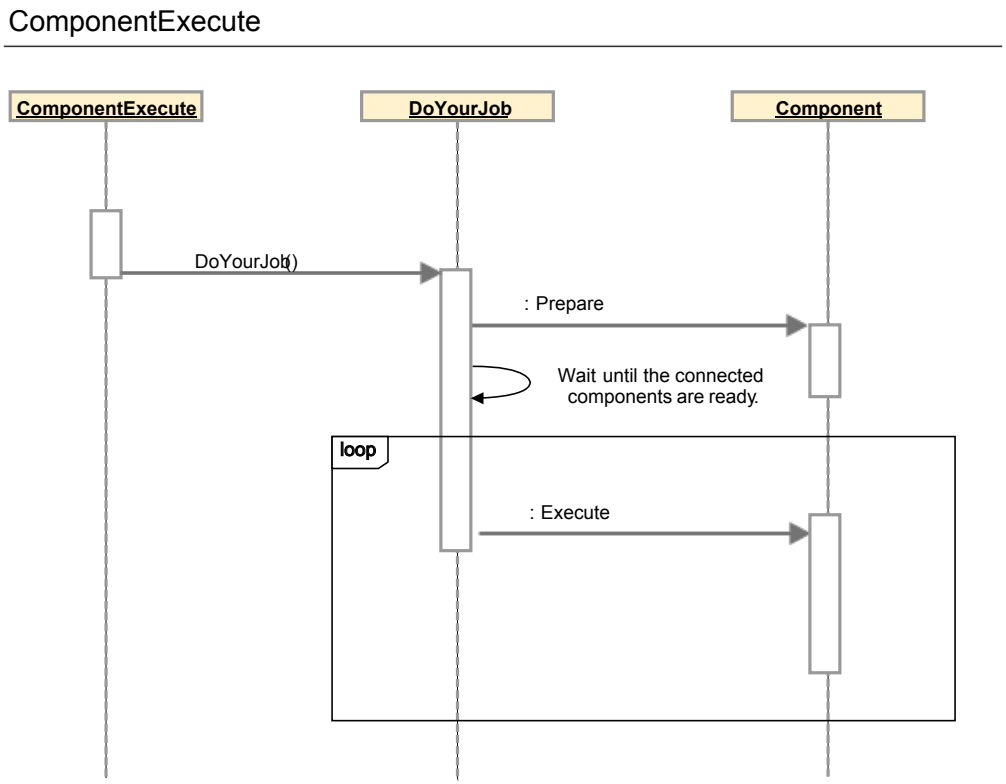

Figure 2.5. ComponentExecute

2.3.3.3.3. ComponentStop()

This function changes the component to the exit state and ends it.

2.3.3.3.4. ComponentRelease()

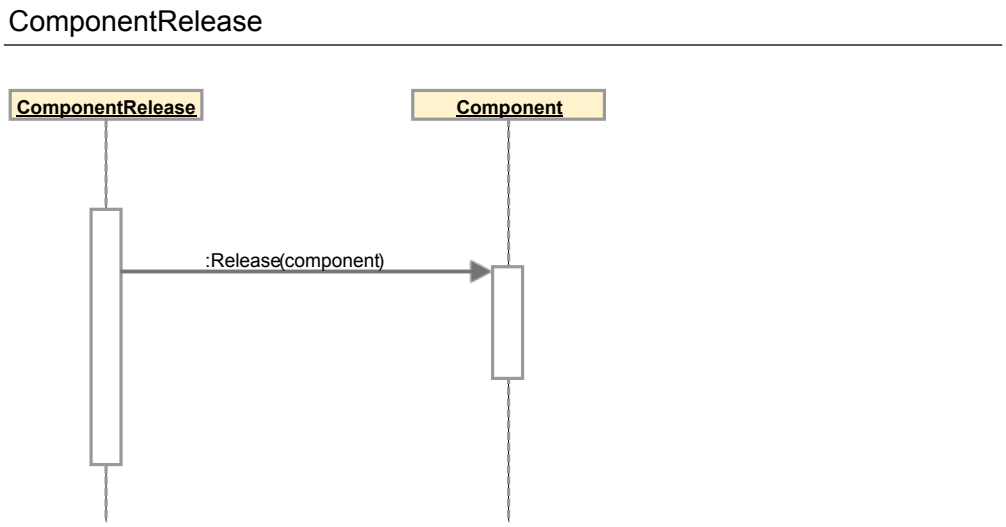

Figure 2.6. ComponentRelease

2.3.3.3.5. ComponentDestroy()

This function destroys the component instance.

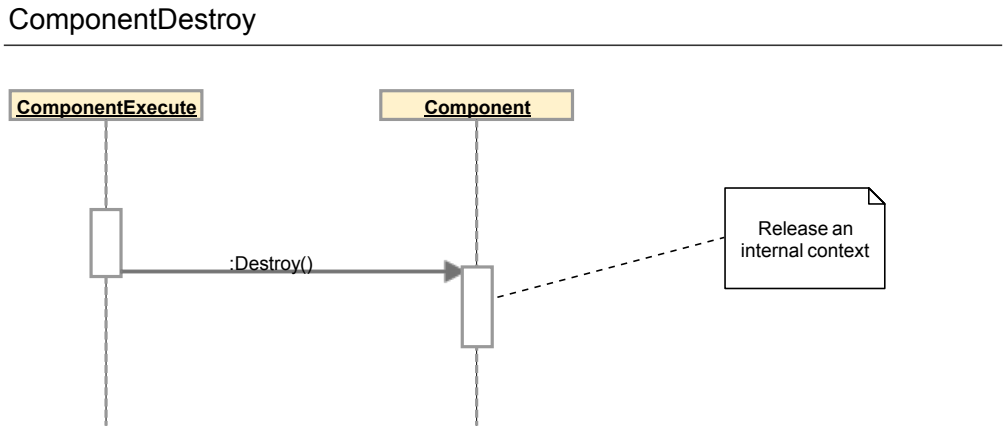

Figure 2.7. ComponentDestroy

2.3.3.3.6. ComponentRegisterListener()

This function is used to call a registered function when a specific event occurs in the desired component.

- `component` : The desired component
- `event` : Oring with interesting events is possible. Events are defined in component.h.
- `func` : The function pointer that is called when events occur.
- `context` : The argument of the function to be called.

The data type for the third argument of `ComponentListenerFunc` can be changed according to the event occurred. For example, when `COMPONENT_EVENT_DEC_GET_OUTPUT_INFO` event occurs, the third argument is a pointer to the `CNMComListenerDecDone` structure variable. That is, the data required when an event occurs can be obtained from the third argument.

Component authors can add a desired event in the component.h if there is no such event. The following code shows an example of decoder component.

```
typedef struct {
    Uint32 val;
} ListenerContext lsnCtx;
static void Listener(Component com, Uint32 event, void* data, void* context)
{
    CNMComListenerDecDone* p = (CNMComListenerDecDone*)data;
    ListenerContext* c = (ListenerContext*)context;


    ...
}


int main()
{
    decoder = ComponentCreate("decoder", &config)
    ComponentRegisterListener(decoder, COMPONENT_EVENT_DEC_GET_OUTPUT_INFO, Listener, (void*)&lsnCtx);
}
```

2.3.3.3.7. ComponentDestroy()

This function waits for the created thread to terminate, then call the `Destory()` of Concrete component to free all allocated resources.

2.3.3.3.8. ComponentSetParameter()

This function sends a specific signal to a component so that the component immediately executes an action. If the component has nothing to do for the signal, the component passes the signal to the connected component . For example, Components A, B, and C are implemented as follows.

`SET_PARAM_XXX` is implemented only in the component C. The component A, B, and C are tunneled.

```
A <---> B <---> C
```

In this case, the `SET_PARAM_XXX` signal is passed to the component B as shown below, but the actual processing is done in the component C.

```
ComponentSetParam(Null, componentB, SET_PARAM_XXX, arg);
```

2.3.3.3.9. ComponentGetParameter()

This function is used to obtain status information from a specific component. The behavior is the same as `ComponentSetParameter()`.

```
static void ExecuteRenderer(ComponentImpl* me)
{
    ...
    ComponentGetParameter(me, me->srcPort.connectedComponent, GET_PARAM_HANDLE, &handle);
    ...
}
```

2.3.4. Writing and Registering A Component

Here we describe the process of creating and registering a dummy component that does nothing.

1. Define functions to be linked with function pointers in the structure of `ComponentImpl`. The code below is a simple example.

```
component_dummy.c
#include <string.h>
#include "component.h"

typedef struct {
    Uint32  data;
} DummyFilterContext;

static CNMComponentParamRet GetParameterDummyFilter(ComponentImpl* from, ComponentImpl* com,
GetParameterCMD commandType, void* data)
{
    return CNM_COMPONENT_PARAM_SUCCESS;
}

static CNMComponentParamRet SetParameterDummyFilter(ComponentImpl* from, ComponentImpl* com,
SetParameterCMD commandType, void* data)
{
    return CNM_COMPONENT_PARAM_SUCCESS;
}

static BOOL ExecuteDummyFilter(ComponentImpl* com, PortContainer* in, PortContainer* out)
{
    if (TRUE == in->last) com->terminate = TRUE;

    return TRUE;
}

static BOOL DestroyDummyFilter(ComponentImpl* com)
{
    DummyFilterContext* ctx = (DummyFilterContext*)com->context;

    osal_free(ctx);

    return TRUE;
}

static Component CreateDummyFilter(ComponentImpl* com, CNMComponentConfig* componentParam)
{
    DummyFilterContext* ctx;

    ctx = (DummyFilterContext*)osal_malloc(sizeof(DummyFilterContext));
    osal_memset(ctx, 0, sizeof(DummyFilterContext));

    com->context = (void*)ctx;

    return (Component)com;
}

static BOOL PrepareDummyFilter(ComponentImpl* com, BOOL* done)
{
    *done = TRUE;

    return TRUE;
}

static void ReleaseDummyFilter(ComponentImpl* com)
{
}

ComponentImpl dummyFilterComponentImpl = {
    "dummyFilter",
    NULL,
    {0,},
    {0,},
    sizeof(PortContainerDisplay),   /* OUTPUT */
    5,
    CreateDummyFilter,
    GetParameterDummyFilter,
    SetParameterDummyFilter,
    PrepareDummyFilter,
    ExecuteDummyFilter,
    ReleaseDummyFilter,
    DestroyDummyFilter
};
```

2. As shown in the above code, the defined functions are assigned to function pointers of `dummyFilterComponentImpl`. And declare it in component.h.

```
component.h
extern ComponentImpl feederComponentImpl;
extern ComponentImpl decoderComponentImpl;
extern ComponentImpl rendererComponentImpl;
extern ComponentImpl dummyFilterComponentImpl;
```

3. Finally, register it in the componentList array in component_list.h. Execution of Wave5xxDecV2.mak/Wave5xxEncV2.mak generates the component_list_decoder.h/component_list_encoder.h and component_list_all.h. Your components should be registered in the component_list.h properly whether it works for encoder or decoder.

```
static ComponentImpl* componentList[] = {
    &feederComponentImpl,
    &decoderComponentImpl,
    &rendererComponentImpl,
    &dummyFilterComponentImpl,
    NULL
};
```

4. This allows dummyFilterComponentImpl to be created via the `ComponentCreate()` function with the name "dummyFilter".

```
Component com = ComponentCreate("dummyFilter");
```

2.4. Port

A port is a gate for passing data between components. A port has two internal queues: one is the data tranfer side (outputQ) and the other is the side that receives the transferred data (inputQ). *Figure 2.8, "Port"* illustrates the port consisting of inputQ and outputQ.

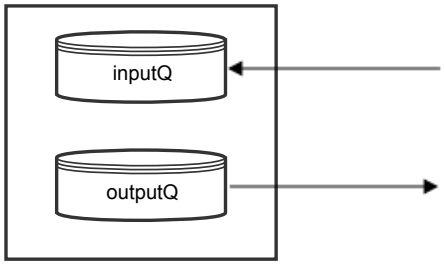

Figure 2.8. Port

One port is created in the `ComponentCreate()` process. The component has two ports: source port and sink port. The port created by `ComponentCreate()` is the sink port. The source port points to the sink port of the source component when tunneling.

- source port - A pointer to the port of the source component.
- sink port - A port that has data to send to the sink component.

Data transfer between ports is done through the `PortContainer` structure. For passing data, data should be carried in the `PortContainer`. The correspoding component retrieves the `PortContainer`, extracts the data, and returns the `PortContainer`.

Chapter 3
Encoder Sample

Figure 3.1, "Encoder Sample" depicts the encoder application sample consisting of Feeder Component, Encoder Component, and Reader Component. Data communications with components is always directed to a specific component port.

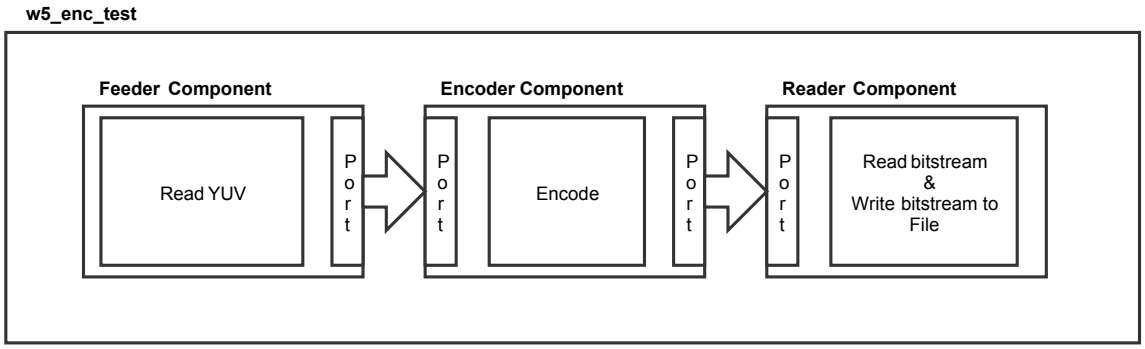

Figure 3.1. Encoder Sample

This chapter details each of component in terms of component duty, task function and flow, and data passed between the components.

3.1. YUV Feeder Component

3.1.1. Componenet Role

The YUV Feeder Component allocates framebuffers. It reads one frame from YUV file, loads it in the allocated framebuffer, and passes the framebuffer information to encoder component.

`PrepareYuvFeeder()`
To allocate the reconstruction framebuffer and the source framebuffer, this method obtains the information of each necessary framebuffer from the encoder component. *Figure 3.2, "PrepareYuvFeeder"* below is the process of assigning reconstruction framebuffer and source framebuffer in the YuvFeeder:Prepare() phase.

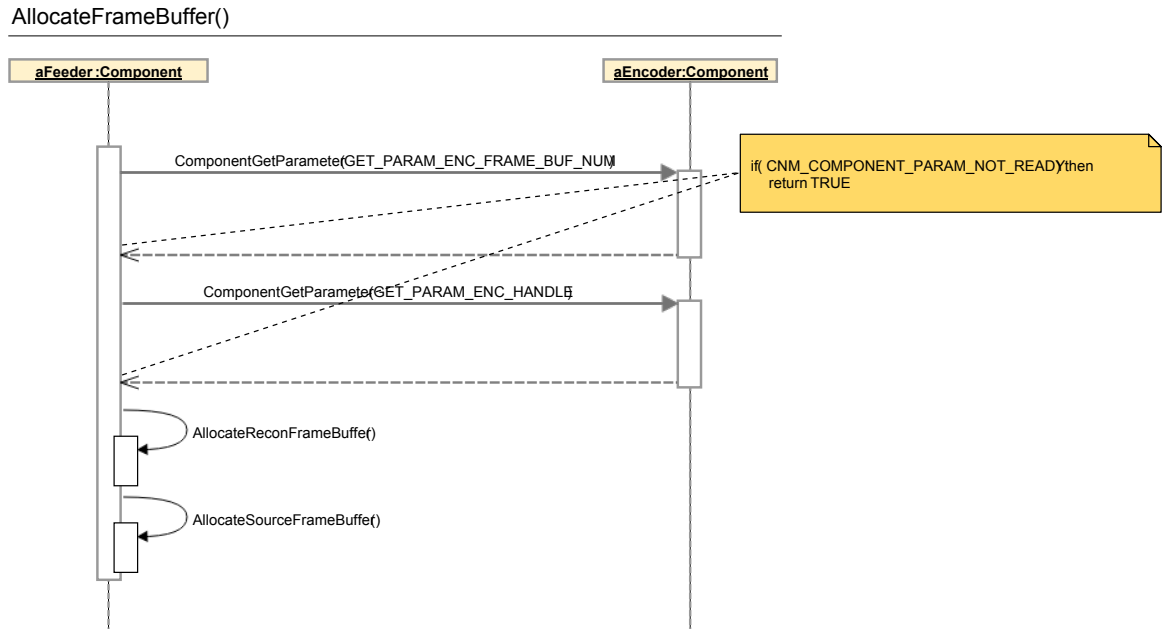

Figure 3.2. PrepareYuvFeeder

`ExecuteYuvFeeder()`

The main function of the YUV feeder component is to read YUV frame by frame, store it in the framebuffer, and pass the framebuffer information to the encoder component. *Figure 3.3, "ExecuteYuvFeeder"* is a schematic flow chart for `ExecuteYuvFeeder()`.

The `out` variable, the `PortContainer` data type pointer, is a container that is passed to the encoder component. It must not pass YUV data to the encoder component until the framebuffer is verified to be registered with the VPU. Through `ComponentGetParameter(GET_PARAM_ENC_BUF_REGISTERED)`, we can see if the encoder has registered a framebuffer with the VPU. If the framebuffer has not been registered yet, set `out->reuse` to TRUE to prevent YUV data from being passed to the encoder component so that you can reuse the container on the next call. If the encoder component is ready, you can load YUV into the framebuffer by calling `YuvFeeder_Feed()`.

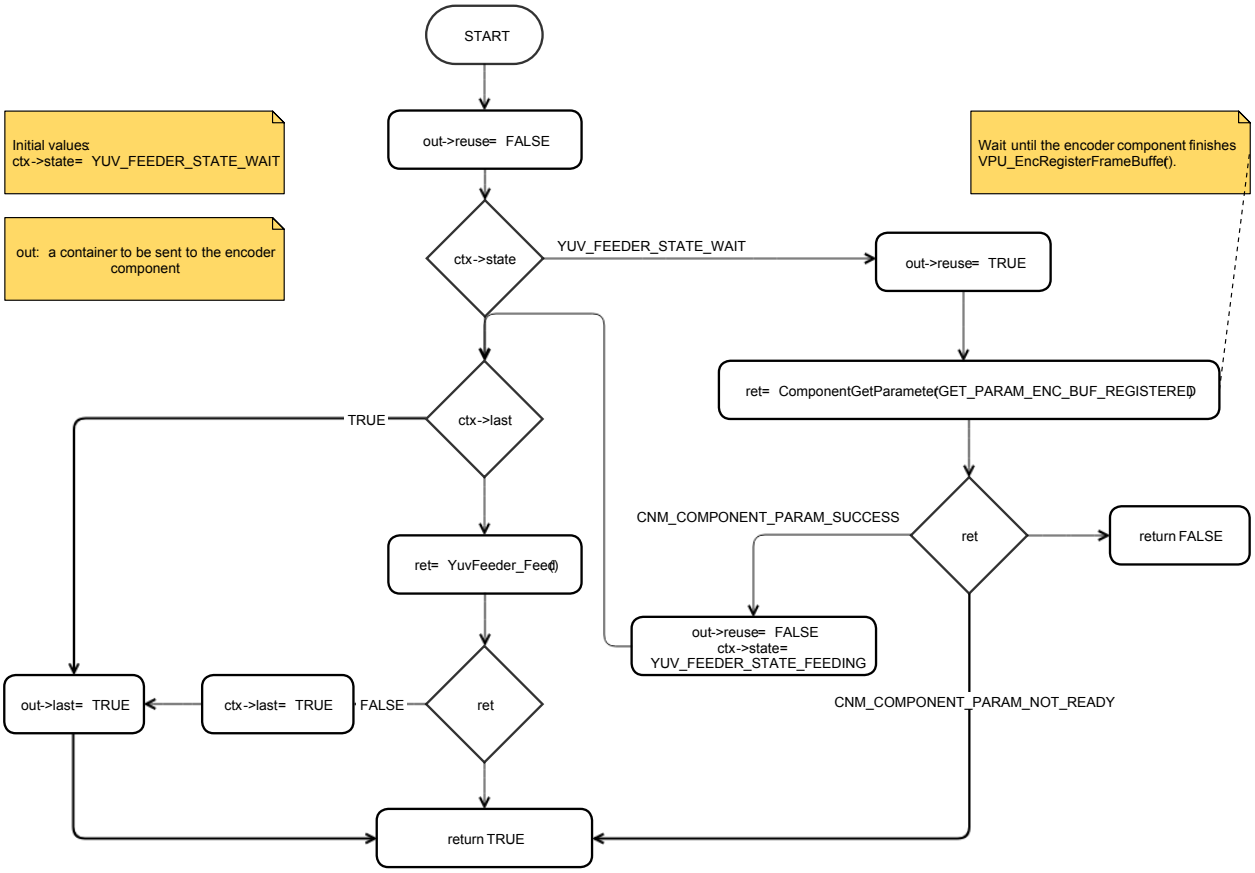

Figure 3.3. ExecuteYuvFeeder

3.1.2. Listener Events

None

3.1.3. GetParameter

`GET_PARAM_YUVFEEDER_FRAME_BUF`
passes the informaion on framebuffer.

3.1.4. SetParameter

None

3.2. Encoder Component

The Encoder Component is responsible for encoding YUV received from the feeder and passing the encoded stream to the Reader Component. This component carries out a four-step operation except for the initialization and termination steps.

1. Open a encoder instance
2. Encode a stream header
3. Register framebuffers

4. Encode a frame

The three initial steps are performed only once, and the last "Encode a frame" is repeated every time the stream comes in.

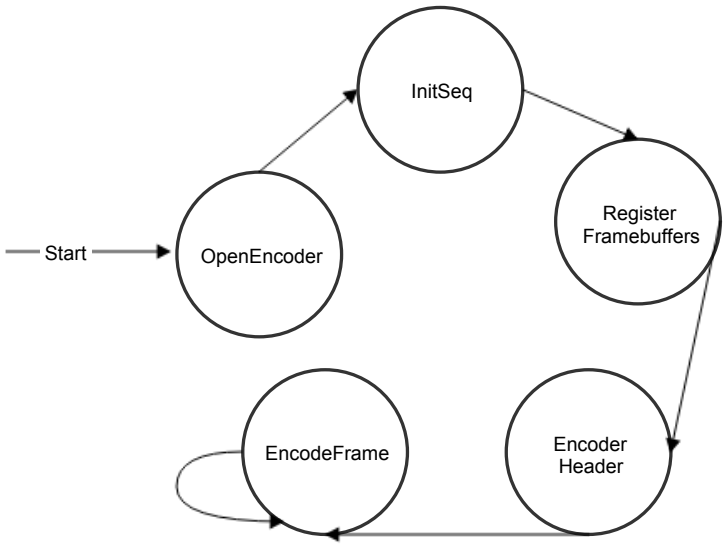

Figure 3.4. Encoder State Transition

The code below is for the above state transition.

```
ExecuteEncoder
switch (ctx->state) {
case ENCODER_STATE_OPEN:
    ret = OpenEncoder(com);
    if (ctx->stateDoing == FALSE) ctx->state = ENCODER_STATE_INIT_SEQ;
    break;
case ENCODER_STATE_INIT_SEQ:
    ret = SetSequenceInfo(com);
    if (ctx->stateDoing == FALSE) ctx->state = ENCODER_STATE_REGISTER_FB;
    break;
case ENCODER_STATE_REGISTER_FB:
    ret = RegisterFrameBuffers(com);
    if (ctx->stateDoing == FALSE) ctx->state = ENCODER_STATE_ENCODE_HEADER;
    break;
case ENCODER_STATE_ENCODE_HEADER:
    ret = EncodeHeader(com);
    if (ctx->stateDoing == FALSE) ctx->state = ENCODER_STATE_ENCODING;
    break;
case ENCODER_STATE_ENCODING:
    if (in)  in->reuse  = FALSE;
    if (out) out->reuse = FALSE;
    ret = Encode(com, (PortContainerYuv*)in, (PortContainerES*)out);
    break;
default:
    ret = FALSE;
    break;
}
```

3.2.1. OpenEncoder()

This method calls VPU_EncOpen() to get an encoder handle. It also raises an event if there is a listener registered in the `COMPONENT_EVENT_ENC_OPEN` event.

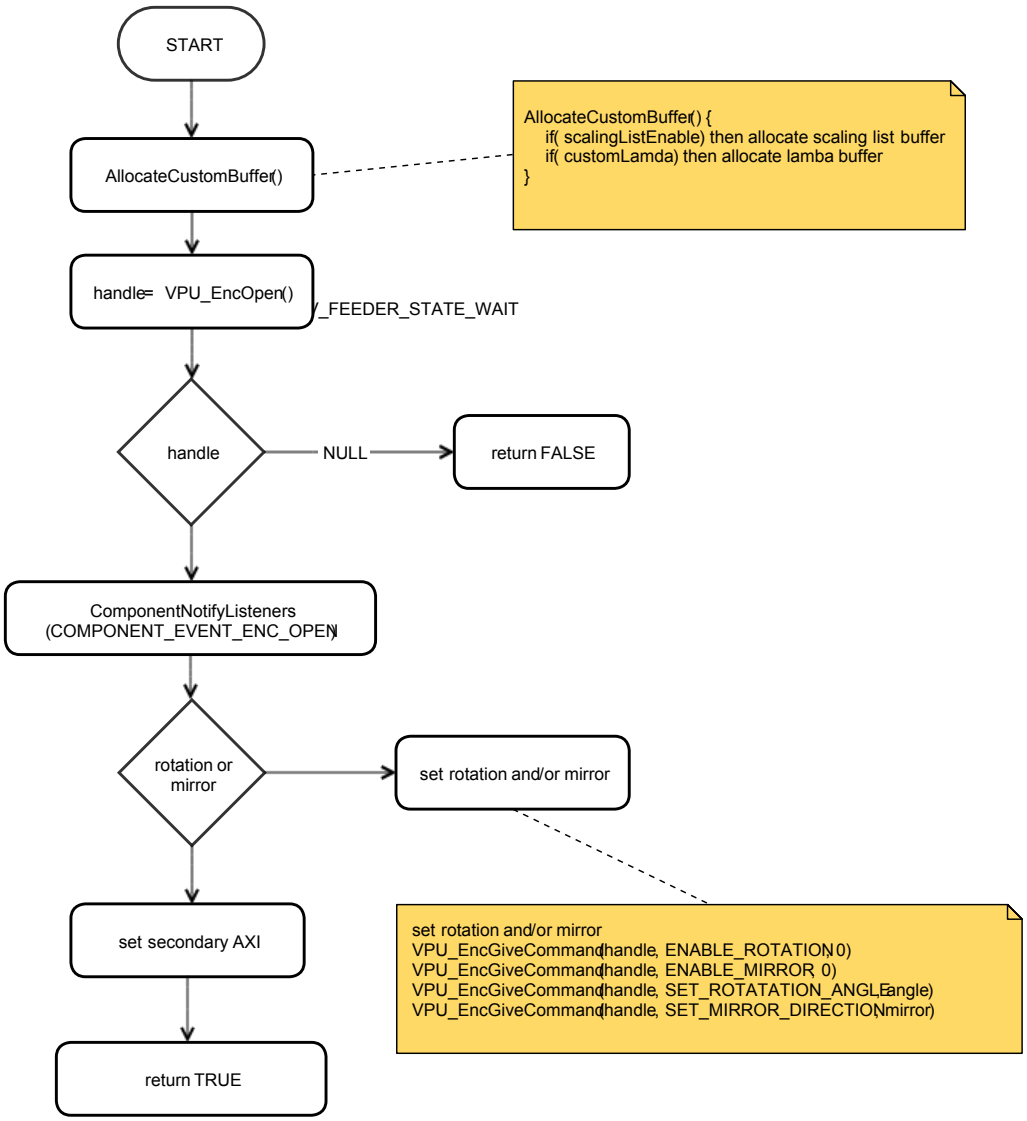

Figure 3.5. OpenEncoder()

3.2.2. SetSequenceInfo()

This method passes encoder parameter information to VPU so that the VPU can compose of video sequence information. *Figure 3.6, "SetSequenceInfo()"* is a flow chart of `SetSequenceInfo()`. `SetSequenceInfo()` goes to the next step if `ctx-> stateDoing` is FALSE as shown in the code for the above state transition. That is, `ctx-> stateDoing` means that `SetSequenceInfo()` has not yet finished.

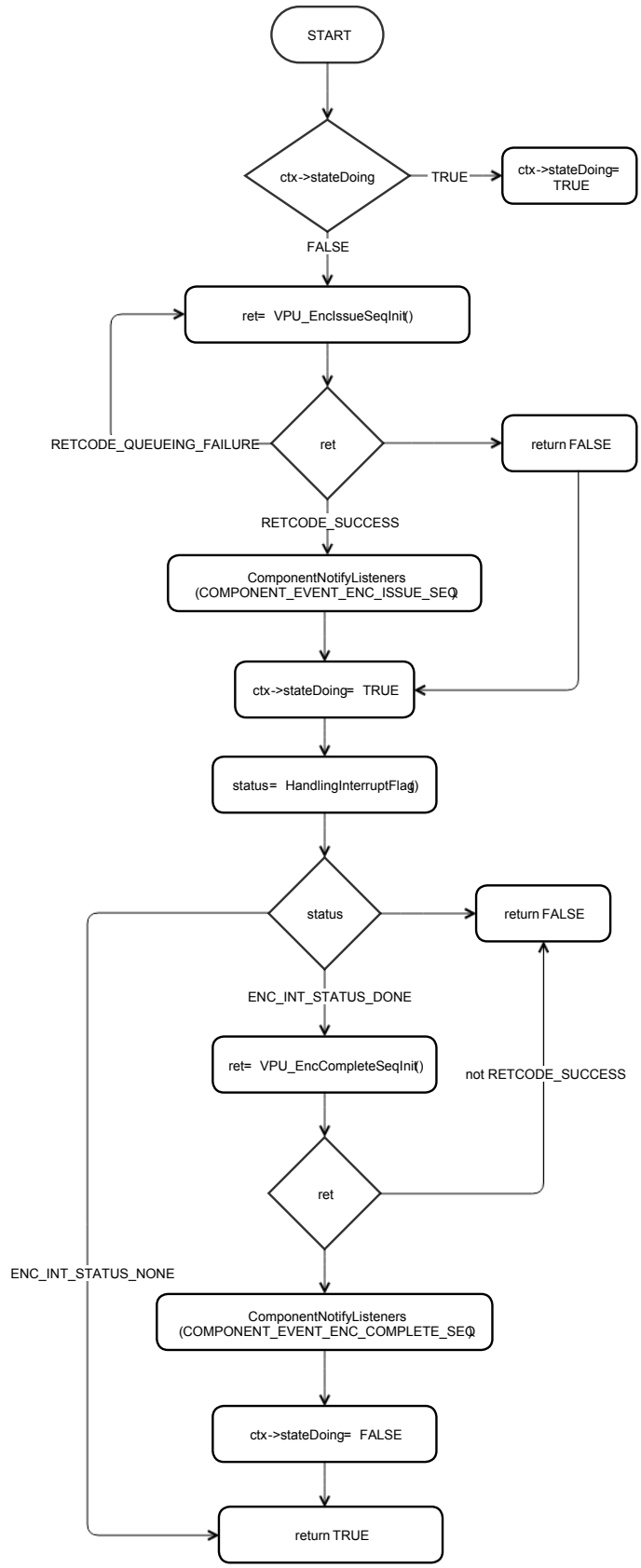

Figure 3.6. SetSequenceInfo()

3.2.3. RegisterFrameBuffer()

This method obtains the information of reconstruction framebuffer and source framebuffer from the YUV feeder component and registers the reconstruction framebuffers for VPU by using `VPU_EncRegisterFrameBuffer()` function.

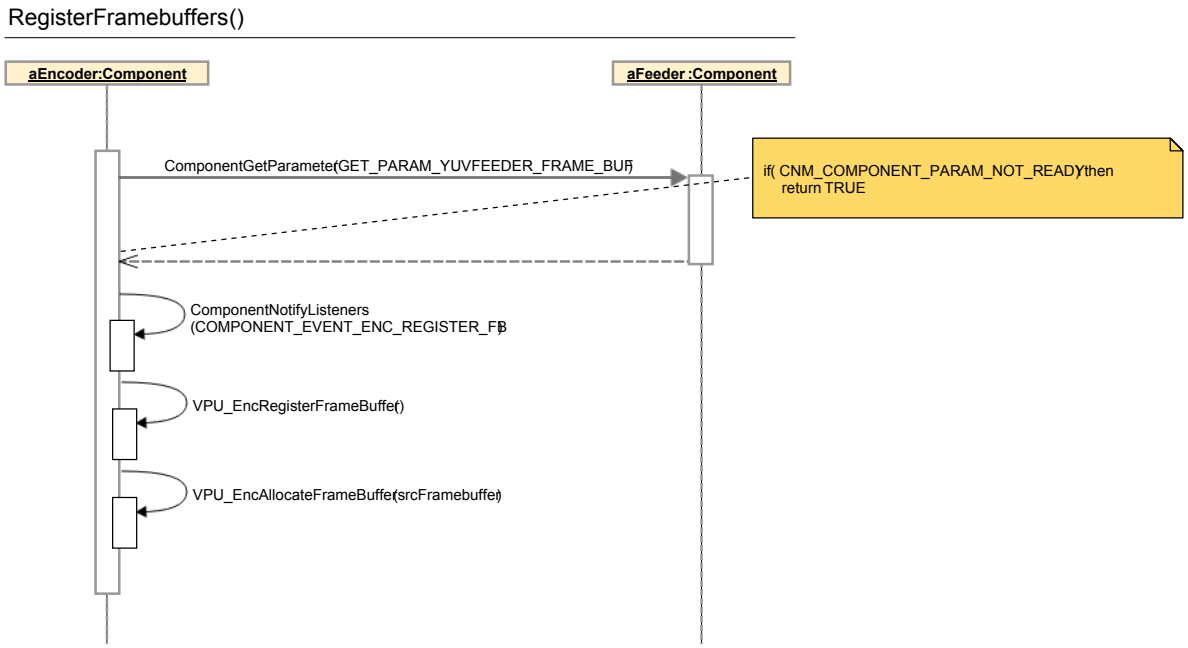

Figure 3.7. RegisterFrameBuffer()

3.2.4. EncodeHeader()

This method creates the video sequence information according to the codec with the video parameter information delivered to the VPU via `SetSequenceInfo()`.

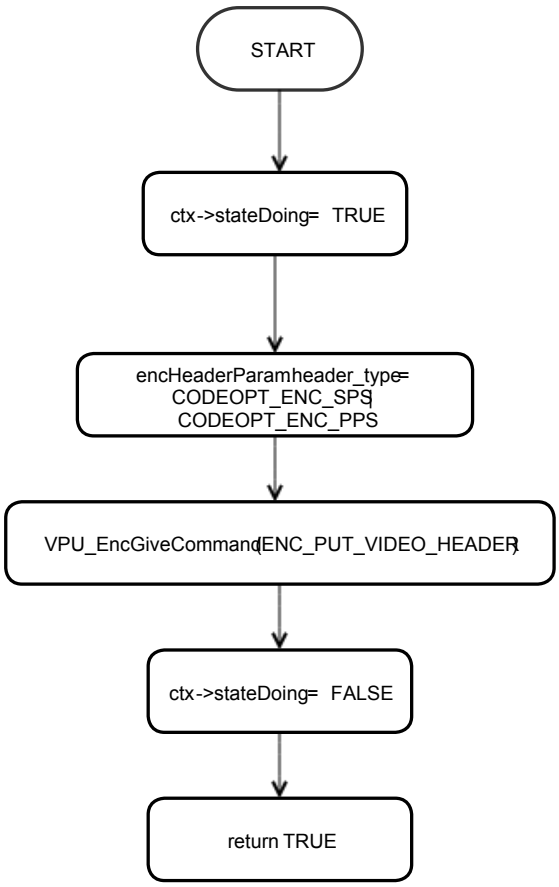

Figure 3.8. EncodeHeader()

3.2.5. Encode()

Figure 3.9, "Encode()" is a flowchart of the `encoder()` function. The encoding process is divided into three phases for convenience of explanation.

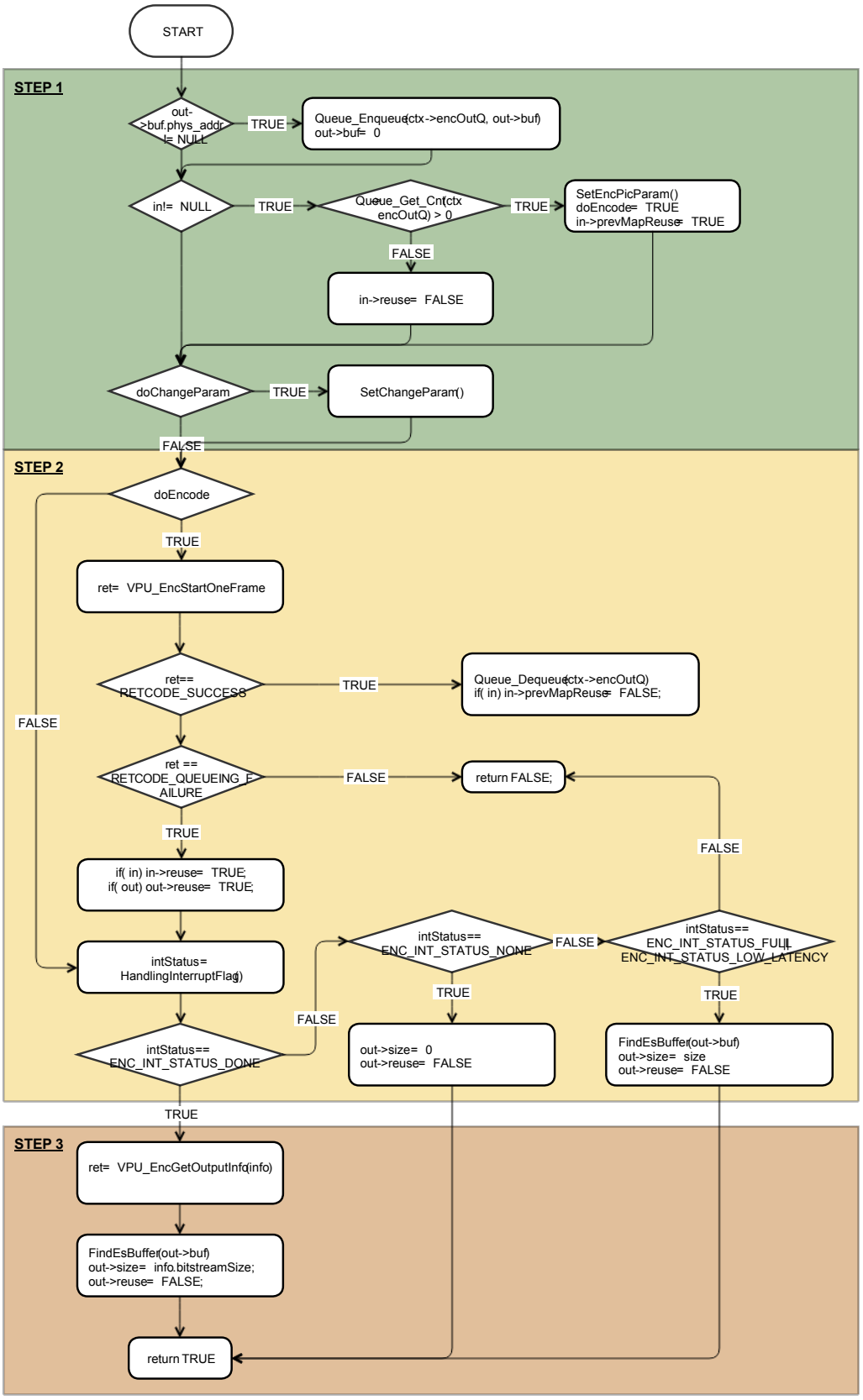

Figure 3.9. Encode()

STEP 1

The first step is to check the input container and output container and verify the encoding is possible. The variable `in`, which corresponds to the input container, contains YUV information. Therefore, if `in` is not null, set `doEncode` to TRUE. The variable `out` in the output container has bitstream buffer information. Put this information in the container internal variable `encOutQ`. The purpose of `encOutQ` is to have a usable bitstream buffer as container internal information. When one encode command uses one bitstream buffer, it prevents the encoding process from proceeding if the bitstream buffer is exhausted. Therefore, if encoding is possible `(doEncoder = TRUE)`, there must be an input container and a bitstream buffer available for encOutQ. If encoding is possible, call `SetEncPicParam()` to set encoding related parameters in the `EncParam` structure variable.

STEP 2

The second step is to issue the encoding command to the VPU and receive the interrupt. If the return value of `VPU_EncStartOneFrame()` is `RETCODE_SUCCESS`, it calls `Queue_Dequeue(encOutQ)` to indicate that the bitstream buffer has been consumed. `RETCODE_QUEUEING_FAILURE` does not call `Queue_Dequeue(encOutQ)`, because the command queue is full. Also, it changes the `reuse` field of the input/output container to TRUE so that you can use the same input and output container again for the next call.

Call `HandlingInterruptFlag()` to get interrupt flag information. If encoding is complete with `ENC_INT_STATUS_DONE`, go to the next step. Otherwise, it behaves according to each state.

STEP 3

The last step is to call the `VPU_EncGetOutputInfo()` that obtains the bitstream information, and to place the received bitstream buffer information in the output container, and pass it to the connected sink component.

3.2.6. Listener events

`COMPONENT_EVENT_ENC_OPEN`

Event for VPU_EncOpen() method call. The listener receives the CNMComListenerEncOpen structure information.

`COMPONENT_EVENT_ENC_ISSUE_SEQ`

Event for VPU_EncIssueSeqInit() method call. The listener does not receive information about the event, but a null pointer.

`COMPONENT_EVENT_ENC_COMPLETE_SEQ`

Event for VPU_EncCompleteSeqInit() method call. The listener receives the CNMComListenerEncCompleteSeq structure information.

`COMPONENT_EVENT_ENC_REGISTER_FB`

Event for VPU_EncRegisterFrameBuffer() method call. The listener does not receive information about the event, but a null pointer.

`COMPONENT_EVENT_ENC_READY_ONE_FRAME`

Event occurring before calling VPU_EncStartOneFrame() The listener does not receive information about the event, but a null pointer.

`COMPONENT_EVENT_ENC_START_ONE_FRAME`

Event for VPU_EncStartOneFrame() method call. The listener receives the CNMComListenerEncDone structure information.

3.2.7. GetParameter

`GET_PARAM_COM_IS_CONTAINER_CONUSUMED`

Informs whether the YUV framebuffer in the input container has been consumed. In case of command queue, it cannot return the container immediately to the source component because the frame is not encoded at the time the command is entered.

`GET_PARAM_ENC_HANDLE`
Obtains an encoder handle.

`GET_PARAM_ENC_FRAME_BUF_NUM`
Returns the required reconstruction framebuffer and the number of source framebuffer.

`GET_PARAM_ENC_FRAME_BUF_REGISTERED`
Informs VPU about whether the reconstruction framebuffer is registered.

3.2.8. SetParameter

None

3.3. Reader Component

3.3.1. Component Role

Allocates a bitstream buffer. Reads the bitstream information received from the encoder component and stores it in a file.

`PrepareReader()`
Allocate a bitstream buffer as much as the source port.

`ExecuteReader()`
Receives the bitstream buffer information from the input container, variable `in`, reads the bitstream, and stores it in the file.

3.3.2. Listener Events

None

3.3.3. GetParameter

`GET_PARAM_COM_IS_CONTAINER_CONSUMED`
The consumed field of conatiner is always TRUE, since there are no special restrictions.

`GET_PARAM_READER_BITSTREAM_BUF`
Obtains the information of allocated bistream buffer.

3.3.4. SetParameter

None

Chapter 4
Decoder Sample

Figure 4.1, "Decoder Sample" depicts the decoder application sample consisting of Feeder Component, Decoder Component, and Renderer Component. Data communications with components is always directed to a specific component port.

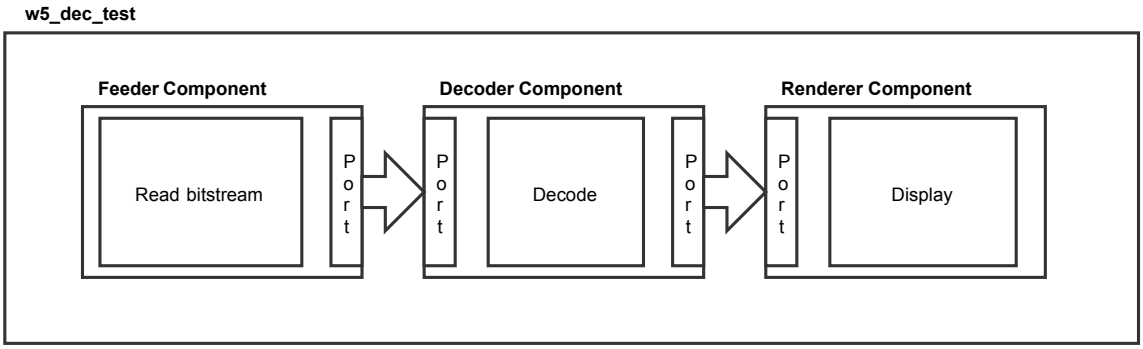

Figure 4.1. Decoder Sample

This chapter details each of component in terms of component duty, task function and flow, and data passed between the components.

4.1. Feeder Component

4.1.1. Componenet Role

The Feeder Component reads bitstream from a file, writes it to bitstream buffer, and passes the read pointer and writer pointer of the bitstream buffer to decoder component. It also allocates a bitstream buffer. In case of interrupt mode, one bitstream buffer is allocated. In pic-end mode, it allocates the bitstream buffers as muany as the number of sink port queues.

4.1.2. Listener Events

None

4.1.3. GetParameter

`GET_PARAM_COM_IS_CONTAINER_CONSUMED`
The fourth argument to `GetParameter()` is a pointer to the PortCotainer structure. You should always assign TRUE to `PortContainer::consumed`.

`GET_PARAM_FEEDER_BITSTREAM_BUF`
Obtains the address and size of the allocated bitstream buffer.

`GET_PARAM_FEEDER_EOS`
Gets the result if the stream has been consumed up.

4.1.4. SetParameter

SET_PARAM_COM_PAUSE
Depending on the value of the fourth argument, the operation of the component is temporarily stopped or restarted.

SET_PARAM_FEEDER_START_INJECT_ERROR
Enables bit-error injection which puts error bits in the stream to be passed to the decoder componenet.

SET_PARAM_FEEDER_STOP_INJECT_ERROR
Disables bit-error injection which puts error bits in the stream to be passed to the decoder componenet.

SET_PARAM_FEEDER_RESET
Initializes the Feeder job.

4.2. Decoder Component

4.2.1. Componenet Role

The Decoder Component is responsible for decoding bitstream received from the feeder and delivering the decoded result to the Renderer Component. This component carries out four stages of operation except for the initialization and termination steps.

1. Open a decoder instance
2. Decode a stream header
3. Register framebuffers
4. Decode a frame

The previous 3 steps are performed only once, and the last "Decode a frame" is repeated every time the stream comes in.

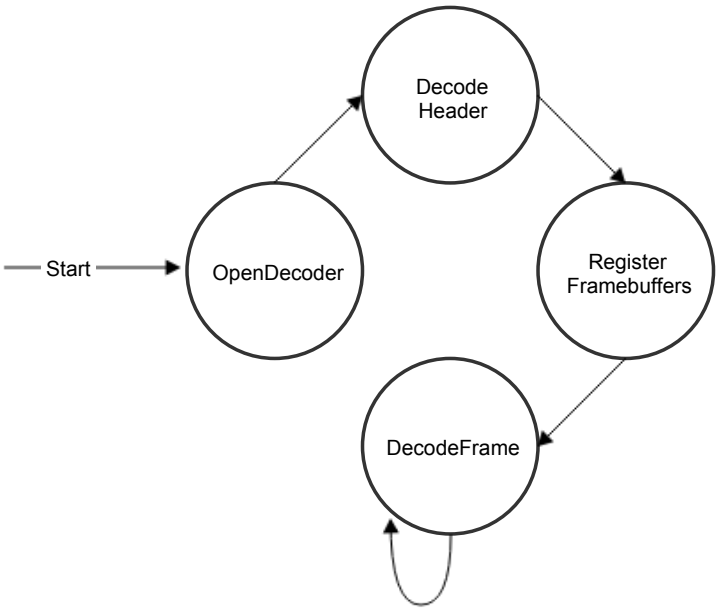

Figure 4.2. Decoder State Transition

The code below is for the above state transition.

```
ExecuteDecoder
static BOOL ExecuteDecoder(ComponentImpl* com, PortContainer* in, PortContainer* out)
{
...
if (ctx->state == DEC_STATE_INIT_SEQ || ctx->state == DEC_STATE_DECODING) {
    if (UpdateBitstream(ctx, (PortContainerES*)in) == FALSE) {
        return FALSE;
    }
    if (in) {
        // In ring-buffer mode, it has to return back a container immediately.
        if (bsMode == BS_MODE_PIC_END) {
            if (ctx->state == DEC_STATE_INIT_SEQ) {
                in->reuse = TRUE;
            }
            in->consumed = FALSE;
        }
        else {
            in->consumed = (in->reuse == FALSE);
        }
    }
}
...
switch (ctx->state) {
case DEC_STATE_OPEN_DECODER:
    ret = OpenDecoder(com);
    if (ctx->stateDoing == FALSE) ctx->state = DEC_STATE_INIT_SEQ;
    break;
case DEC_STATE_INIT_SEQ:
    ret = DecodeHeader(com);
    if (ctx->stateDoing == FALSE) ctx->state = DEC_STATE_REGISTER_FB;
    break;
case DEC_STATE_REGISTER_FB:
    ret = RegisterFrameBuffers(com);
    if (ctx->stateDoing == FALSE) {
      ctx->state = DEC_STATE_DECODING;
      DisplayDecodedInformation(ctx->handle, ctx->decOpenParam.bitstreamFormat, 0, NULL,
          ctx->testDecConfig.performance, 0);
    }
    break;
case DEC_STATE_DECODING:
    ret = Decode(com, (PortContainerES*)in, (PortContainerDisplay*)out);
    break;
default:
    ret = FALSE;
    break;
}
...
}
```

4.2.1.1. OpenDecoder()

`OpenDecoder()` gets the bitstream buffer from the feeder component and calls `VPU_DecOpen()` to get the decoder handle. If the feeder component has not yet allocated the bitstream buffer, it returns TRUE without changing state to `DEC_STATE_INIT_SEQ`. The bitstream buffer should be checked again in the next call, since the state change to `DEC_STATE_INIT_SEQ` has not been made.

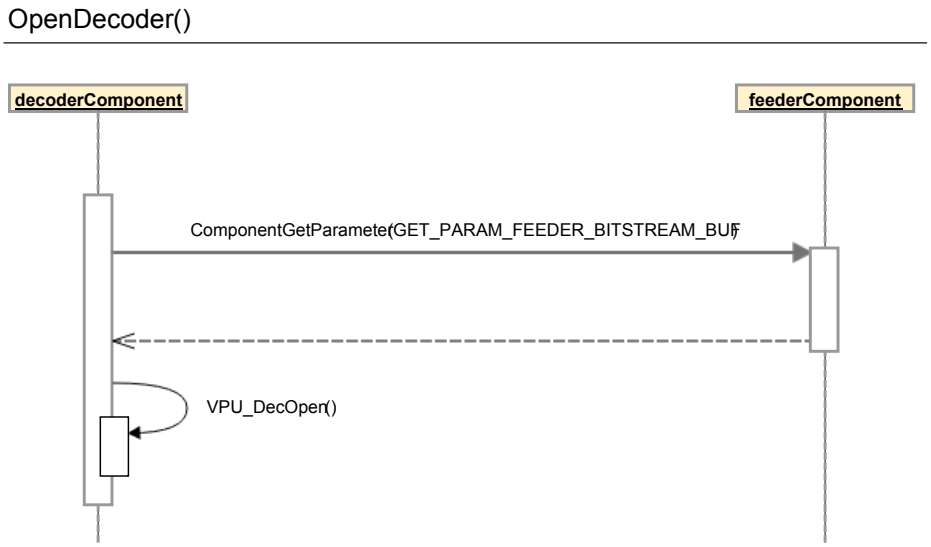

Figure 4.3. OpenDecoder

4.2.1.2. DecodeHeader()

`DecodeHeader()` parses the header of the video sequence.

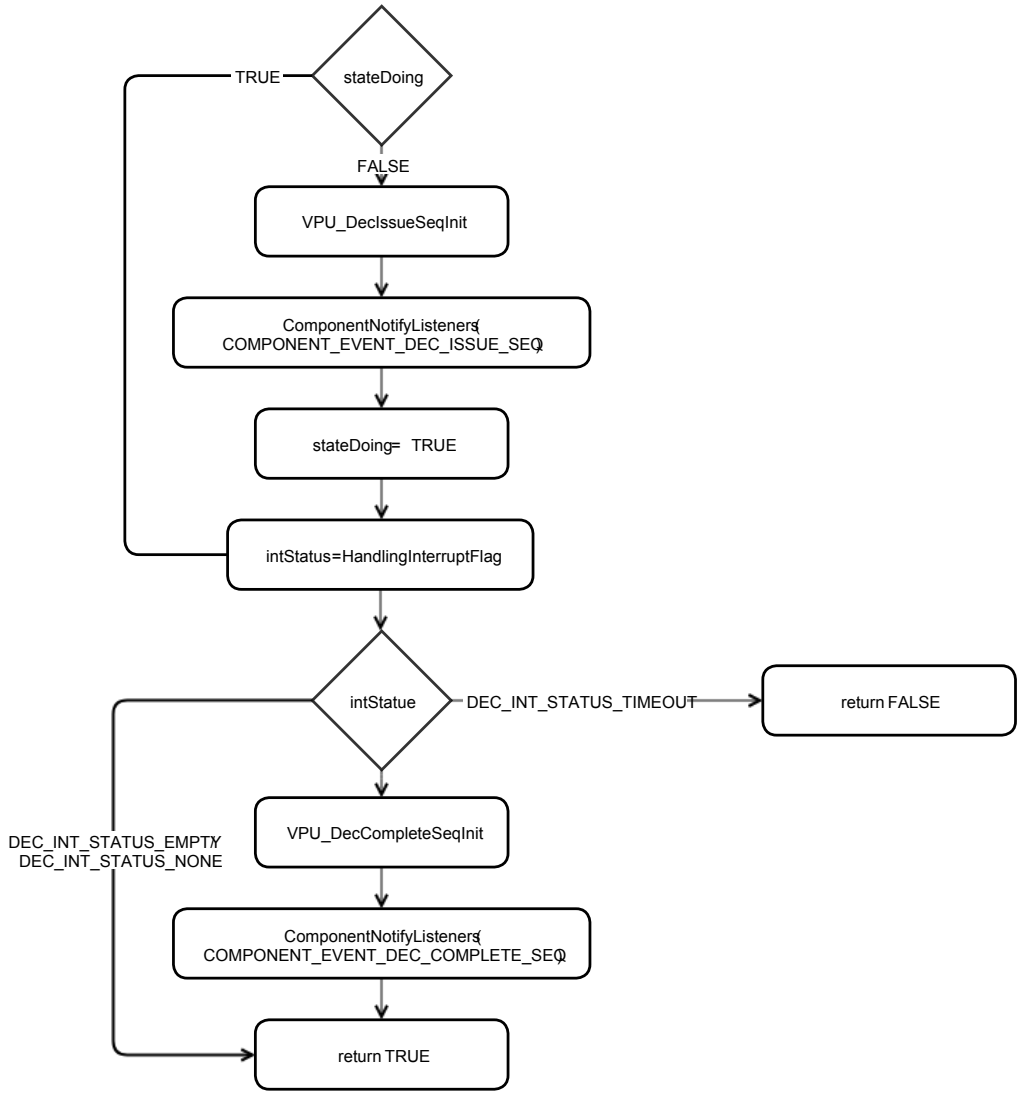

Figure 4.4. DecodeHeader

4.2.1.3. RegisterFrameBuffers()

The framebuffer is allocated by the renderer component. After passing the `GET_PARAM_RENDERER_FRAME_BUF` command to the Renderer component and getting the framebuffer, it calls `VPU_DecRegisterFrameBufferEx()`.

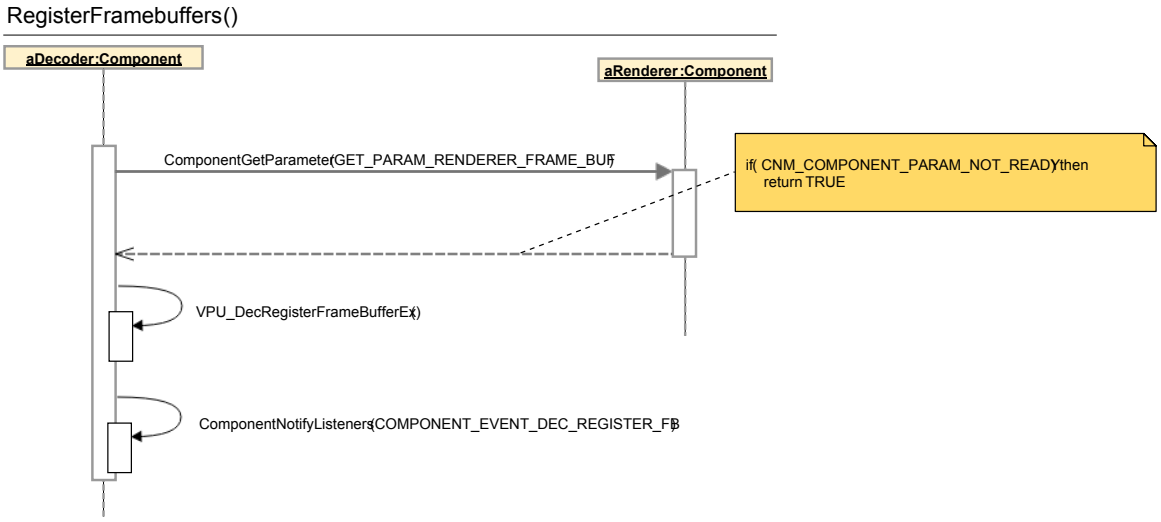

Figure 4.5. RegisterFrameBuffers

4.2.1.4. Decode()

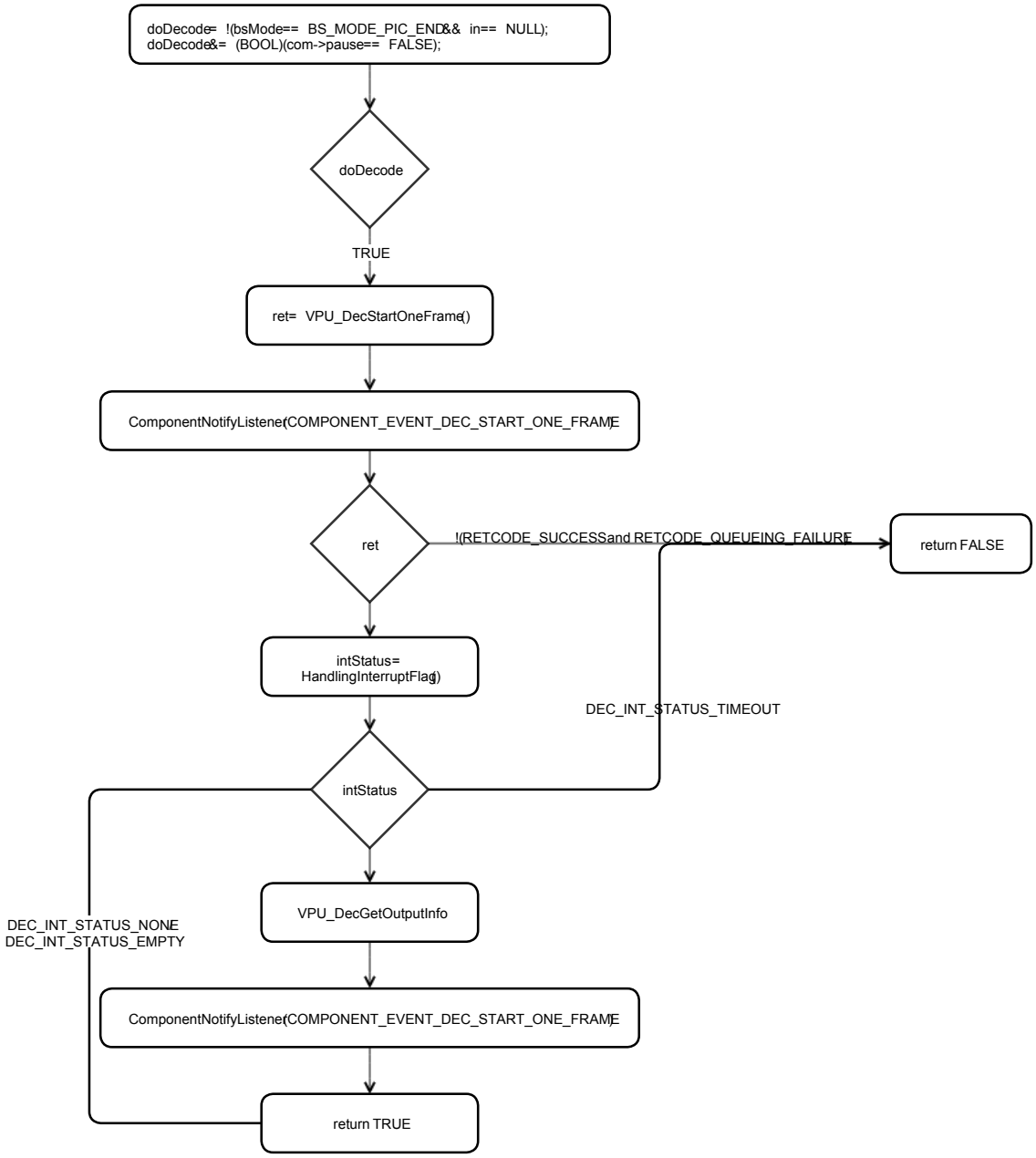

Figure 4.6. Decode

If `DecOutputInfo::indexFrameDisplay` is greater than or equal to 0 in the information obtained from `VPU_DecGetOutputInfo()`, data is loaded into the output container It changes the reuse field of the output container to FALSE.

```
Decode
static BOOL Decode(ComponentImpl* com, PortContainerES* in, PortContainerDisplay* out)
  ...
  if ((decOutputInfo.indexFrameDisplay >= 0) || (decOutputInfo.indexFrameDisplay == DISPLAY_IDX_FLAG_SEQ_END)) {
      ctx->numOutput++;
      out->last  = (BOOL)(decOutputInfo.indexFrameDisplay == DISPLAY_IDX_FLAG_SEQ_END);
      out->reuse = FALSE;
```

```
  }
  ...
}
```

4.2.2. Listener Events

COMPONENT_EVENT_DEC_OPEN
Event for VPU_DecOpen() method call. The event will occur even on failure.

COMPONENT_EVENT_DEC_ISSUE_SEQ
Event for VPU_DecIssueSeqInit() method call. The event will occur even on failure.

COMPONENT_EVENT_DEC_COMPLETE_SEQ
Event for VPU_DecCompleteSeqInit() method call. The event will occur even on failure.

COMPONENT_EVENT_DEC_REGISTER_FB
Event for VPU_DecRegisterFrameBuffferEx() method call. The event will occur even on failure.

COMPONENT_EVENT_DEC_READY_ONE_FRAME
Event raising before VPU_DecStartOneFrame() method call. The event will occur even on failure.

COMPONENT_EVENT_DEC_START_ONE_FRAME
Event for VPU_DecStartOneFrame() method call. The event will occur even on failure.

COMPONENT_EVENT_DEC_GET_OUTPUT_INFO
Event for VPU_DecGetOutputInfo() method call. The event will occur even on failure.

COMPONENT_EVENT_DEC_CLOSE
Event for VPU_DecClose() method call

COMPONENT_EVENT_RESET_DONE
The decoder component automatically enters reset process when there is no response from VPU. This event arises when the reset flow is over.

COMPONENT_EVENT_DEC_DECODED_ALL
This event arises when all frames have been decoded.

4.2.3. GetParameter

GET_PARAM_COM_IS_CONTAINER_CONSUMED
Checks if input stream in the container has been consumed or not. In case of using command queue, it is needed because input stream is not consumed immediately at the time the command is entered. If the input stream is consumed, it changes `PortContainer::consumed` to TRUE if comsumed.

GET_PARAM_DEC_HANDLE
Returns a decoder handle.

GET_PARAM_DEC_FRAME_BUF_NUM
Obtains the minimum required number of framebuffers.

GET_PARAM_DEC_BITSTREAM_BUF_POS
Returns the read pointer and write pointer of the current bitstream buffer.

GET_PARAM_DEC_CODEC_INFO
Returns the video sequence information in the DecGetOutputInfo structure.

GET_PARAM_DEC_QUEUE_STATUS
Obtains the status of command queue.

4.2.4. SetParameter

`SET_PARAM_COM_PAUSE`
Pauses temporarily or restart the component operation according to the value of the fourth argument.

`SET_PARAM_DEC_SKIP_COMMAND`
Sends a skip command.

`SET_PARAM_DEC_RESET`
Resets VPU.

4.3. Renderer Component

4.3.1. Component Role

The renderer component allocates the framebuffer with the information retrieved from the decoder component. It also displays the framebuffer which is passed from the decoder component. The framebuffer displayed is returned to VPU.

`AllocateFrameBuffer()`
Framebuffers are assigned in Component:Prepare(). It obtains required number of framebuffers from the decoder component.

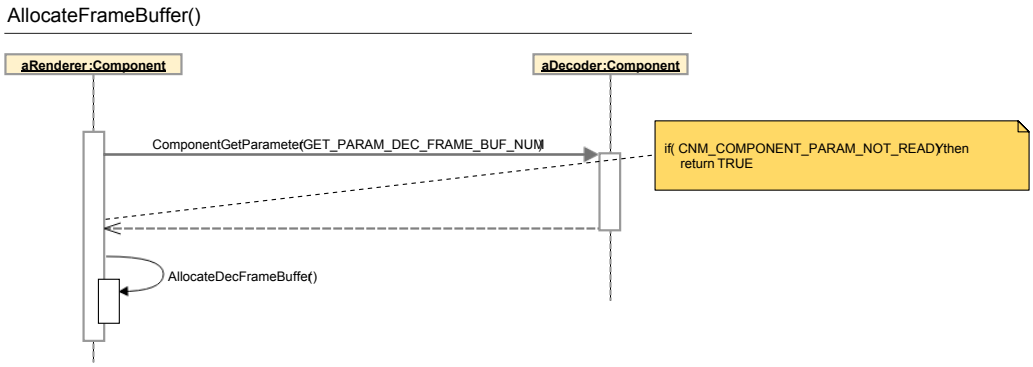

Figure 4.7. AllocateFramebuffer

`DisplayFrame()`
Returns the used framebuffer to VPU by using `VPU_DecClrDispFlag()`.

4.3.2. Listener events

None

4.3.3. GetParameter

`GET_PARAM_COM_IS_CONTAINER_CONSUMED`
The fourth argument of `GetParameter()` is a pointer to the PortCotainer structure. `PortContainer::consumed` should be always TRUE.

`GET_PARAM_RENDERER_FRAME_BUF`
Returns the allocated framebuffer information.

GET_PARAM_RENDERER_PPU_FRAME_BUF
If PPU(rotation, mirror) is supported, it returns the information of PPU framebuffer.

4.3.4. SetParameter

SET_PARAM_RENDERER_REALLOC_FRAMEBUFFER
Reassigns the framebuffer to a different size. It can be used when only certain frames such as VP9 are changed in size.

SET_PARAM_RENDERER_FREE_FRAMEBUFFERS
Releases all the framebuffer memory.

SET_PARAM_RENDERER_FLUSH
Extracts all the data from the renderer component and returns them to VPU immediately by calling `VPU_DecClrDispFlag()`.

SET_PARAM_RENDERER_ALLOC_FRAMEBUFFERS
Reallocates the framebuffer. You must first pass and execute `SET_PARAM_RENDERER_FREE_FRAMEBUFFER` before issuing `SET_PARAM_RENDERER_ALLOC_FRAMEBUFFERS`.

About Chips&Media

Chips&Media, Inc. is a leading video IP provider. Over the past decade, they have been doing high quality product design focusing on hardware implementation of high speed, low power, cost effective video solution for H.264, MPEG-4, H.263, MPEG-1/2, VC-1, AVS, MVC, VP8, Theora, Sorenson, and up to recent H.265/HEVC along with its remarkable bandwidth reduction technology.

They have a broad range of IP products from low-end 1080p HD mobile solutions to high-end 4K 60fps HEVC solutions or even brandnew HEVC/H.265, VP9, and AVS2 multi-decoder IP to fulfill target demands from various multimedia device markets.

Chips&Media's IPs are field-proven in over a hundred million of multimedia devices and by more than 60 top-tier semiconductor companies. The headquarter is located in Seoul, Korea (Republic of) with sales offices in Japan, China, and Taiwan. To find further information, please visit the company's web site at *http://www.chipsnmedia.com*