

***MPEG4 Video Simple Profile***  
***Encoder User's Manual***

---

*C64 & C64+ platform – version 1.1*  
*Non-XDAIS version*

***Mecoso Technogy Inc.***  
***2008***

## Contents

<b>1 Introduction.....</b>	<b>3</b>
<b>2 Header files.....</b>	<b>5</b>
<b>3 API.....</b>	<b>7</b>
<b>4 Example.....</b>	<b>14</b>
<b>5 Differences between ver 1.0 and 1.1 .....</b>	<b>16</b>

# 1 Introduction

## 1.1 Overview

This document describes Mecoso's MPEG4 Simple Profile Video Encoder library for TI's C64x platforms. The library is *not* XDAIS-compatible and is provided for customers who need to write applications that don't follow XDAIS guidelines.

Questions regarding general features of MP4V Encoder library please send to [info@mecoso.com](mailto:info@mecoso.com). For bug reports/integration support please use [support@mecoso.com](mailto:support@mecoso.com)

## 1.2 Performance and memory requirement

Mecoso's MP4V encoder delivers high performance yet requires minimal resource. Code size is about 40KBytes. Scratch memory (on chip SRAM) requirement is moderate (depends on the video size, for examples with D1 resolution is 12K for nodma version and 32K for DMA version). On DM642 (600MHz), encoding a D1 resolution image (720\*480) at moderate quality takes around 12-15ms. This means two encoding channels could be run in parallel on a single chip. On DM6437 the performance is better and 3 channels could be run on a single chip if the video content and the speed control is appropriate. These figures are for DMA versions of the encoder.

Code size: ~ 40 KB  
Data size: ~ 10 KB

## 1.3 Encoder libraries

The MP4V encoder library consists of 4 libraries:

### C64 libraries:

- *mp4ve\_a64\_nonxdais\_nodma.lib*            nodma version
- *mp4ve\_a64\_nonxdais\_dat.lib*            dma version

### C64+ libraries:

- *mp4ve\_a64p\_nonxdais\_nodma.lib*            nodma version
- *mp4ve\_a64p\_nonxdais\_acpy3.lib*            dma version

Evaluation libraries have the suffix "eval" to their names.

The reasons for four libraries is:

- Binaries compiled for C64+ core are in general more efficient than for C64 core. Therefore we should use C64+ libraries for C64+ core and C64 libraries for C64 core.
- The DMA mechanism on C64 core (EDMA2) and C64+ core (EDMA3) are different. Thus the way a C64 library make calls to DMA functions (based on CSL) are also different from the way a C64+ library make calls to its DMA functions (based on DMAN3)
- When developers have difficulty with DMA versions (e.g. codec hang when using DMAN3) they can always fall back to the nodma versions of the library, which is slower but more stable.

## 2 Header files

There are two header files:

- *mp4ve\_nonxdais\_nodma.h*
- *mp4ve\_nonxdais\_dma.h*

Platform	Library	Use header file
C64	<i>mp4ve_a64_nonxdais_nodma.lib</i>	<i>mp4ve_nonxdais_nodma.h</i>
C64	<i>mp4ve_a64_nonxdais_dat.lib</i>	<i>mp4ve_nonxdais_dma.h</i>
C64+	<i>mp4ve_a64p_nonxdais_nodma.lib</i>	<i>mp4ve_nonxdais_nodma.h</i>
C64+	<i>mp4ve_a64p_nonxdais_acpy3.lib</i>	<i>mp4ve_nonxdais_dma.h</i>

These two header files defined just 3 memory size parameters:

```
MP4VE_INSTANCE_SIZE
MP4VE_SCRATCH_SIZE
MP4VE_FRAMEBUF_SIZE
```

,but the value of the last two parameters are different depend on the encoded video resolution and the library we're using. The header files will calculate the required memory sizes based on the previously defined MAXWIDTH and MAXHEIGHT. An encoder instance with memory define by MAXWIDTH and MAXHEIGHT will work will all video having the same or lower resolutions. Thus these two parameters needed to be defined before reading the header file. For example:

```
#define MAXWIDTH 720
#define MAXHEIGHT 480
#include <mp4ve_nonxdais_dma.h>
unsigned char my_mp4ve[MP4VE_INSTANCE_SIZE];
unsigned char my_scratch[MP4VE_SCRATCH_SIZE];
unsigned char encoder_framebuf[MP4VE_FRAMEBUF_SIZE];
...
```

```
// create an encoder instance  
  
mp4ve_create(&my_mp4ve,my_scratch,encoder_framebuf,720,480,10);
```

This encoder instance defined here will be able to encode all D1 (720x480) and smaller resolution video.

### 3 API

#### MP4V Encoder functions:

Initialization functions:

*mp4ve\_create()*

*mp4ve\_get\_dman3()*

*mp4ve\_release\_man3*

for C64+ DMA version only

for C64+ DMA version only

Encoding function:

*mp4ve\_encode()*

Controlling function:

*mp4ve\_setspeed()*

**Name:** `mp4ve_create()`

**Synopsis:** `int mp4ve_create(void *instance_mem,  
unsigned char *scratch_buf,unsigned char *frame_buf,  
int width, int height, int startq)`

**Argument:** `void *mp4ve_instance  
unsigned char *scratch_buf,unsigned char *frame_buf,  
int width,int height`

**Return value:** `int (should be zero)`

**Description:** This is the first function to call to create an instance of the mp4ve encoder.

- `instance_mem` is a pointer to the (pre-allocated) memory used to keep all parameters related to this instance of the encoder. Each instance of the encoder needs a separated memory space.
- `scratch_buf` is a pointer to the on chip (L2 SRAM) memory used by the encoder as DMA and processing buffer. Image data will be DMAed into this buffer, then all the encoding will happen here. For this reason we need it to be in a fast memory space (on-chip L2 memory). All encoder instances and other algorithm instances running at the same priority can and should share the same scratch memory to save precious on-chip memory resources.
- `frame_buf` is a pointer to the memory space used by the encoder to store its reconstructed frame buffers.
- As we don't assume the direct control of memory allocation functions, these three memory space should be allocated by the application before calling `mp4ve_create`. Their requirement sizes are defined in `mp4ve_params.h`
- `width` and `height` are the size of input video frames. Input video frames are always in YUV420 format (other video formats will need to convert to YUV420 externally)
- `startq` is the starting value for `qscale`. Its value is [1-31]

When we don't need an encoder instance anymore, just re-use it's instance memory space for other tasks. A scratch memory space can only be released if all algorithm instances that share it don't run any more.

**Example:**

This example creates 3 separated encoder instances. Each one has its own instance memory and frame buffer. All of them share a single scratch memory:

```
#define MAXWIDTH 720
#define MAXHEIGHT 480
#include <mp4ve_nonxdais_nodma.h>

// space for encoder instance
unsigned char mp4ve_instance[3][MP4VE_INSTANCE_SIZE];
```

```
// space for scratch memory
#pragma DATA_SECTION(my_scratch, ".ISRAM") // assume ISRAM is onchip memory
#pragma DATA_ALIGN(my_scratch, 128)
unsigned char my_scratch[MP4VE_SCRATCH_SIZE];

// space for frame buffers
unsigned char mp4ve_framebuf[3][MP4VE_FRAMEBUF_SIZE];

...

// create 3 encoder instances, all share the
// same scratch memory on my_scratch

// first instance running at 720*480 resolution, starting qscale=10
mp4ve_create(&mp4ve_instance[0][0], &my_scratch, &mp4ve_framebuf[0][0],
            720, 480,
            10);

// second instance running at 480*320 resolution, starting qscale=6
mp4ve_create(&mp4ve_instance[1][0], &my_scratch, &mp4ve_framebuf[1][0],
            480, 320,
            6);

// third instance running at 320*240 resolution, starting qscale=12
mp4ve_create(&mp4ve_instance[2][0], &my_scratch, &mp4ve_framebuf[2][0],
            320, 240,
            12);

...
```

**Name:** `mp4ve_get_dman3()`

**Synopsis:** `int mp4ve_get_dman3(void *instance_mem)`

**Argument:** `void *instance_instance`

**Return value:** `int (should be zero)`

**Description:**

This function is only used for *mp4ve\_a64p\_nonxdais\_acpy3.lib*.

The C64+ DMA version (*mp4ve\_a64p\_nonxdais\_acpy3.lib*) needs to access and use the DMA resource. We will use DMAN3 and ACPY3 for this purpose.

After the encoder instance is initialized, this function is called to allocate DMAN3 resources to the encoder. Later on when `mp4ve_encode()` is called, the encoder instance will use these DMAN3 resources through calls to ACPY3 library.

If the return value is non-zero there is an error in the allocation of DMAN3 resources (not enough memory or not enough free EDMA3 channels)

When we don't use an encoder instance any more, the DMAN3 resources allocated to it should be returned to the DMA manager, this is done by calling `mp4ve_release_dman3()`

**Name:** `mp4ve_release_dman3()`

**Synopsis:** `int mp4ve_release_dman3(void *instance_mem)`

**Argument:** `void *instance_instance`

**Return value:** `int (should be zero)`

Note that the C64 DMA version (*mp4ve\_a64\_nonxdais\_dat.lib*) also use DMA resources but this is done though CSL DAT module. We just need to have DAT module initialized at the beginning of the program and there is no need for a function equivalent to `mp4ve_get_dman3()` for *mp4ve\_a64\_nonxdais\_dat.lib*

**Name:** `mp4ve_setspeed()`

**Synopsis:** `void mp4ve_setspeed(void *encoder_instance,  
int speed)`

**Argument:** `void *encoder_instance  
int speed`

**Return value:** `void`

**Description:**

This is a function to control some trade-off between speed and quality inside the encoder.

The value for speed is 0 to 5 (other values will be clamped to this range).

Speed = 0: slowest speed and highest quality

Speed = 5: highest speed and lowest quality.

When an encoder instance is created it has the default value for speed as 0.

If we want to change the encoding speed from the default value we can call this function any time after `mp4ve_create()`. In general, `speed=2` is a good value (faster than default speed but doesn't make quality much worse). For each application and video content we can experiment to see which value is best for speed.

**Name:** `mp4ve_encode()`

**Synopsis:** `int mp4ve_encode(void *encoder_instance,  
unsigned char *frame_ptr[3], unsigned char *bitstream_buf,  
int encode_params)`

**Argument:** `void *encoder_instance  
unsigned char *frame_ptr[3], unsigned char *bitstream_buf,  
int encode_params`

**Return value:** `int` (returning value is the length of writing bitstream)

**Description:** Encode function. Call to encode a frame that have addresses stored in `frame_ptr[]`, using parameters in `encode_params`. The result is written out at `bitstream_buf`. The length of the writing bitstream is returned from the function.

- encoder instance: one encoder instance that has been created using `mp4ve_create()`
- `frame_ptr` is a set of 3 pointers to input frame data
  - `frame_ptr[0]` is Y pointer
  - `frame_ptr[1]` is U pointer
  - `frame_ptr[2]` is V pointer
- `bitstream_buf`: output bitstream buffer
- `encode_params`: takes value as defined in `mp4ve_params.h`. For example:
  - Encode the current frame as an I frame using the current qscale:  
`encode_params = MP4VE_IFRAME`
  - Encode the current frame as an I frame using a new qscale:  
`encode_params = MP4VE_IFRAME + MP4VE_SETQ + qscale`
  - Encode the current frame as a P frame using the current qscale:  
`encode_params = 0`
  - Encode the current frame as an P frame using a new qscale:  
`encode_params = MP4VE_SETQ + qscale`
  - After encode the frame as a P frame and seeing that generated bitstream length is too large, we can “re-encode” it using a new value of qscale:  
`encode_params = MP4VE_REENCODE + MP4VE_SETQ + qscale`

Note: Re-encoding using `(MP4VE_SETQ + qscale)` will result in not correct data, because after the previous encoding call, the encoder has changed the order of its internal buffers.

- If qscale is changed it will be used for the current and all subsequent frames until a new qscale is set. Therefore we'll use a constant qscale for the whole frame. If we keep a constant qscale for the whole sequence, we'll have constant Q, constant quality, variable bit rate (VBR) video stream. If we need constant bit rate (CBR), variable quality video stream, we can change the qscale of each frame based on the variability of the current bit rate, the GOP structure and the video content. Rate control policy in this case has been taken out of the encoder. A simple rate control algorithm for CBR (with source code) is provided as a starting example (rate\_control.c in mp4v\_a64\_loopback\_cbr project). Users can use it as-is or can replace it by better and more sophisticated rate control algorithms.

Note:

a/ - Output buffer size:

The encoding function assumes that the output buffer has enough memory to store all the bitstream content of the current frame. No check is done about buffer overflow. Therefore the designer has to allocate enough memory to the output buffer for the maximum possible length of the bitstream.

b/ For DMA-based versions: it's DMA in, direct-write out:

This function uses DMA to transfer video frame data from external memory into on-chip scratch memory. Encoding functions are done in scratch memory. The result will be written out directly into output bitstream buffer (external or internal). Thus before calling encoding function, we have to make sure that the input frame data memory is cache-cleaned. One way to meet this requirement is to call CSL function CACHE\_clean() (for C64 version) or BCACHE\_wbInv() (for C64+ version) for this block of memory if other processing tasks have directly accessed it before encoding.

There is no need to clean the cache for output bitstream buffer. But if after the encoding there are other processing tasks that access the output bitstream buffer by DMA these tasks will have to clean cache for the output buffer.

c/ In C64 dma version (*mp4ve\_a64\_nonxdais\_dat.lib*), data is dma-in by calling DAT functions from CSL library. Therefore applications using this encoding library has to be linked with either CSL library or one of its subset containing DAT module and the DAT module needs to be initialized by calling DAT\_open().

In C64+ dma version (*mp4ve\_a64p\_nonxdais\_acpy3.lib*), data is dma-in by calling ACPY3 functions that work in the framework of DMAN3 manager. Therefore we need to call mp4ve\_get\_dman3() after each mp4ve\_create() to allocate the necessary DMAN3 resources for the newly created instance.

## 4 Example

Examples in the evaluation suite are in two groups:

- Examples for C64 libraries:

These examples run on DM642EVM. The code is good for ver1 and ver2 of the EVM. For ver3 of the EVM (TI video encoder chip) you can replace the video header, video parameter files and the vport library and recompile. All examples are built under CCS 2.21 for maximum compatibility.

- Examples for C64+ libraries:

These examples run on DM6437EVM. All examples are built under CCS 3.3.

Both set of examples for C64 and C64+ have the same structure and (equivalent) files like following:

- *doc* documentation
- *examples*
  - *common* common files used for examples
  - *mp4ve\_nonxdais\_test* simple project to test the functionality of the encoding library
  - *mp4v\_nonxdais\_loopback* simple project to test the encoder working together with decoder
- *inc* header files
- *lib* library files

Evaluation suite content:

We will describe the files in the evaluation suite example for C64 core (running on DM642EVM) here but the same is true for the files in the C64+ suite examples.

### 4.1/ mp4ve\_a64\_nonxdais\_test:

In this folder, there are two projects:

- *mp4ve\_a64\_nonxdais\_dat.prj* test using dma library
- *mp4ve\_a64\_nonxdais\_nodma.prj* test using nodma library

They are nearly the same with the exception of linking to different headers and libraries.

Each project encode 3 frames of video data (raw frame data is provided in file bus03.asm) and write the result to output file test.m4v.

In “common” folder there is a win32 utility called mp4vd.exe. This can be used to decode the new bitstream file. In a Command window, run this:

```
mp4vd test.m4v
```

and the output will look like this:

I Qi P Qp P Qp ....

where I is the I frame, P is the P frame, Qi and Qp are the Q value that was used to encode the corresponding frame. Each line is a GOP.

At the same time, decoded frames are written out as test0000.yuv, test0001.yuv, ....

Each file is a raw YUV420 file with the size as (width\*height\*3/2). An image viewer that can view YUV420 raw file can be used to view these file. For example the free viewer Xnview at <http://www.xnview.com>

## 4.2/ mp4ve\_a64\_nonxdais\_loopback:

In this folder there are four projects:

- *mp4v\_a64\_loopback\_vbr\_nodma.prj*                      loopback test in constant Q, variable bit rate mode, using nodma library
- *mp4v\_a64\_loopback\_vbr\_dat.prj*                      loopback test in constant Q, variable bit rate mode, using dma library
- *mp4v\_a64\_loopback\_cbr\_nodma.prj*                      loopback test in constant bit rate mode, using nodma library
- *mp4v\_a64\_loopback\_cbr\_dat.prj*                      loopback test in constant bit rate mode, using dma library

The constant bit rate example is built based on a simple bit allocation algorithm (rate\_control.c). You can design and test your own rate control algorithm here.

In each example, after compiling and loading, we should open the following window:

- Dsp/Bios Statistics View                              shows the related statistics, eg. encoder and decoder cycles
- Dsp/Bios Cpu Load Graph                              shows Cpu load
- Dsp/Bios Message Log                                  shows realtime bitrate in "trace"

For the vbr examples also open View Watch window. Watch and change these variables and see the effects:

- *cur\_gop*    change gop size (any value >=1), gop=1 means only I-frame
- *cur\_quant*    change Q (1-31)
- *cur\_speed*    change speed (0-5)

## 5. Differences between version 1.0 and 1.1

### 5.1/ Number of libraries

In version 1.0, the encoder is supplied as two libraries:

- *mp4ve\_lib.lib* contains functions that are used only by encoder
- *mp4v\_common.lib* contains functions that are used by both encoder and decode

In version 1.1, all the common parts are included in the single encoder library, but instead of one library for C64, there are 4 libraries for C64 and C64+. Each platform (C64 and C64+) has two libraries: nodma-based version and dma-based version. Dma-based versions are usually faster than non-dma based versions. C64+ version are often faster (more efficient code) than C64 version.

#### **C64 libraries:**

- *mp4ve\_a64\_nonxdais\_nodma.lib* nodma version
- *mp4ve\_a64\_nonxdais\_dat.lib* dma version

#### **C64+ libraries:**

- *mp4ve\_a64p\_nonxdais\_nodma.lib* nodma version
- *mp4ve\_a64p\_nonxdais\_acpy3.lib* dma version

5.2/ The scratch memory size has been reduced significantly. This allows the encoder to work with larger resolutions that weren't supported in version 1.0. For example, D1 resolution requires about 90KB of scratch memory in version 1.0. In version 1.1 it's 12KB (nodma version) and 32K (dma version).

5.3/ An additional function is introduced to control the trade-off between speed and quality of the encoder. The new function is *mp4ve\_setspeed()*. We can set speed from 0 (slowest speed, highest quality) to 5 (fastest speed, lowest quality)