Application Report
SPRA721 - December 2000

H.263 Encoder: TMS320C6000 Implementation

Hiroshi Miyazawa — *Digital Signal Processing Solutions*

ABSTRACT

This application report describes the implementation of the International Telecommunications Union (ITU)-T H.263 decoder on the TMS320C6000™ DSP. The H.263 encoder does not, at the time of print, meet all of the baseline requirements to be TMS320™ DSP Algorithm Standard compliant; future revisions, however, will be fully eXpressDSP™ compliant. The following document describes the basics of the standard, and proceeds to more technical aspects of the software.

Contents

List of Figures

List of Tables

1 Introduction

The TMS320C6000 implementation of H.263 encoder has the following features.

- It satisfies the minimal requirement defined in the *ITU-T H.263 specification*. None of the annexes have been implemented.
- It is fully compliant, at the time of printing, with the *TMS320 DSP Algorithm Standard*. Please refer to appropriate documentation for more information.
- The code has been tested extensively on TMS320C6201 EVM and TMS320C6211/C6711 DSK.
- The encoder has very little device specific code; executing the encoder on any of the TMS320C6000 devices can be achieved simply by defining the device type at build time.
- Hooks to allow the use of RTP are partially in place. A structure to store necessary information is defined, and can be allocated with appropriate flags at build time, but no code has been implemented.
- The encoder is structured to provide as much flexibility as possible, so that it can run under different system configurations. Please see the appendices for more information on changes that the users can make to suit their systems.

Although “TMS320C64x” is mentioned throughout this application note, this does not by any means imply that any source codes and/or object files are included in the release that the users will receive. Whether or not the TMS320C64x specific codes are released depends on the availability and the specifics of the agreement the user has signed. Please refer to the sales representatives for more information.

2 Encoder Implementation

2.1 Directory Structure

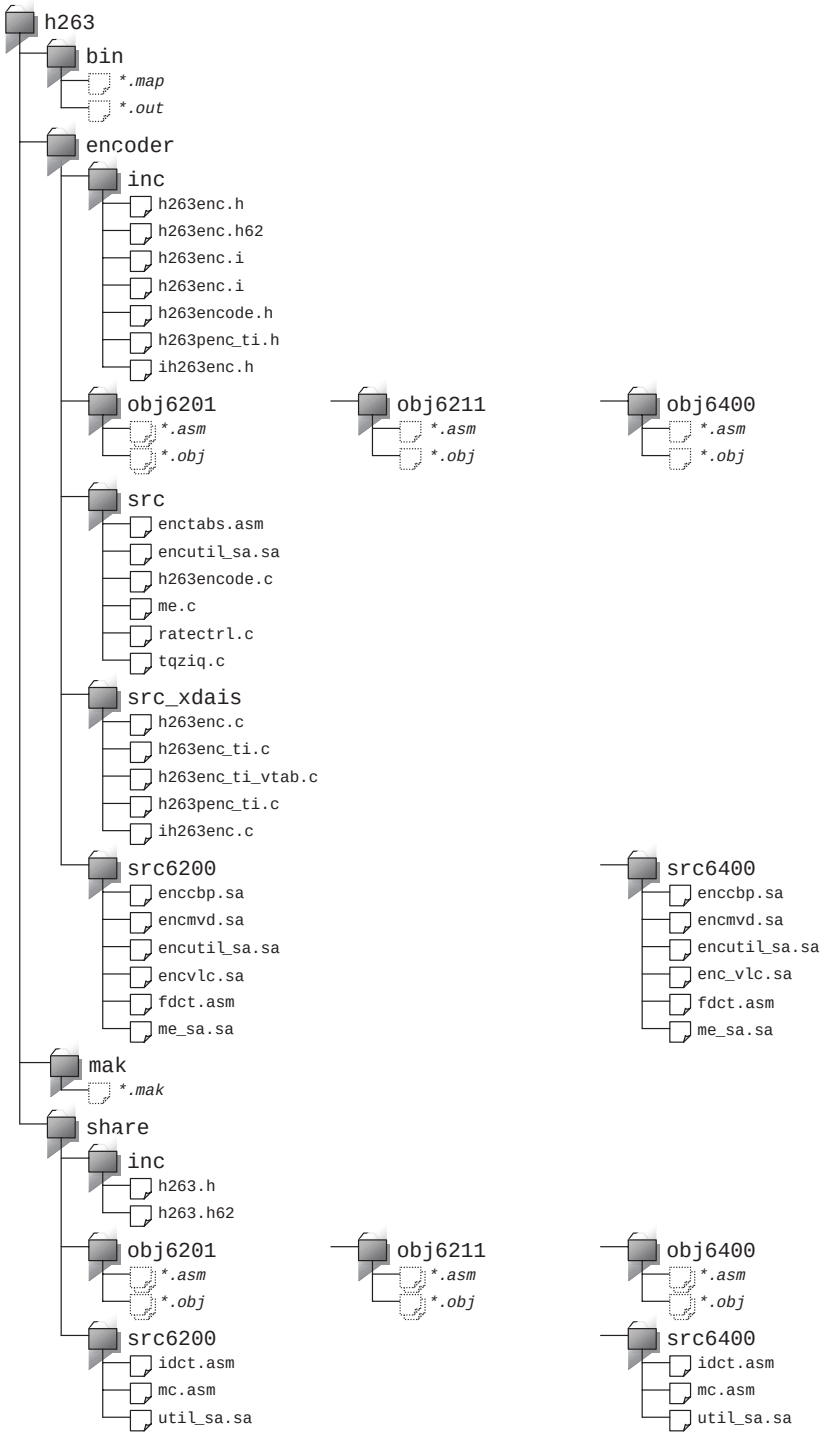


Figure 1. Encoder Directory Structure

h263: H.263 root directory.

bin: Encoder COFF and map files.

encoder: Encoder root directory.

inc: Include files used by the encoder source codes.

obj6201: Intermediate assembly and object files for TMS320C6201.

obj6211: Intermediate assembly and object files for TMS320C6211.

obj6400: Intermediate assembly and object files for TMS320C64x.

src: Encoder source files (device independent)

src_xdais: TMS320 DSP Algorithm Standard specific source files (device independent)

src6200: Source files specifically designed for TMS320C6200.

src6400: Source files specifically designed for TMS320C64x.

mak: Makefiles.

share: Root directory for shared files.

inc: Include files shared by H.263 encoder and decoder source codes.

obj6201: Intermediate assembly and object files for TMS320C6201.

obj6211: Intermediate assembly and object files for TMS320C6211.

obj6400: Intermediate assembly and object files for TMS320C64x.

src: Shared source files.

src6200: Source files specifically designed for TMS320C6200.

src6400: Source files specifically designed for TMS320C64x.

Although the default location for intermediate ASM and OBJ files for TS320C6201 is `obj6201`, the user may choose to create and assign a different directory.

Please note that depending on the specifics of the agreement, directories `obj6400` and `src6400` may not be included in the release. Please refer to the sales representative for more information.

2.2 H.263 Encoder Objects

The current implementation of the H.263 encoder defines a parent object `H263PENC_TI_Obj` (defined in `h263penc_ti.h`) that is used to hold a single copy of the encoder tables (look-up tables, VLC tables, etc.), since they are common to all H.263 encoder child instances. Each child instance, once created, stores pointers to appropriate sections of the tables. By using a parent instance to hold these tables, child instances do not have to retain their own copies of the tables, thereby reducing the amount of memory required by each child instance. This arrangement is shown in the diagram below.

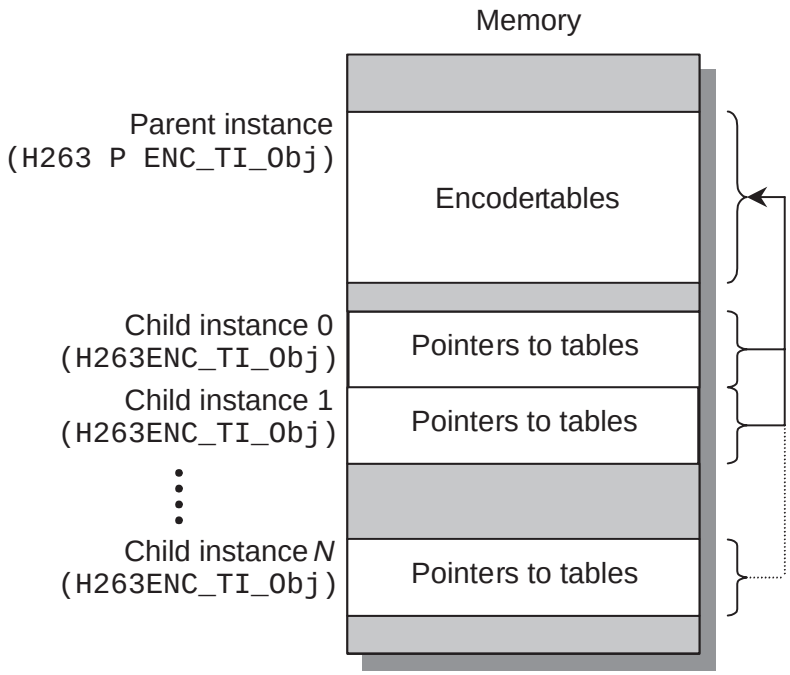


Figure 2. Using Parent and Child Instances

While the aforementioned configuration is probably the most desirable, this may not be practical in some systems. For example, if a particular system is designed to execute several different algorithms that all require their own tables, there may not be sufficient internal memory to hold every single parent instance for each algorithm. One solution for this scenario is to swap in and out the parent instances as needed. Alternatively, the system can swap in and out whichever child instance that has to execute. This is more suited for systems equipped with large external memories, since each child instance is allowed to keep its own copy of the tables. The H.263 encoder is designed so as to allow the user to select which configuration is more suitable. This arrangement is illustrated in the figure below.

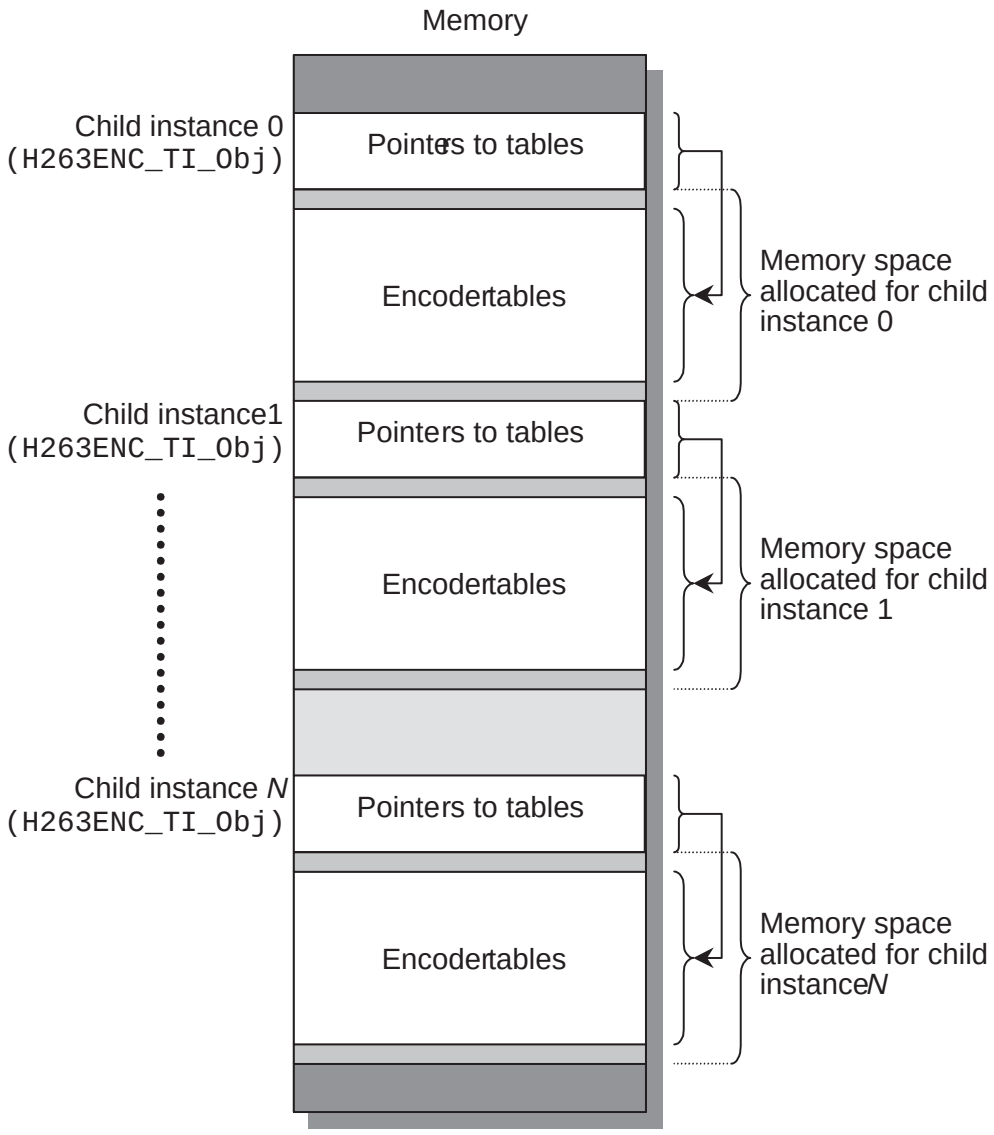


Figure 3. Using Only Child Instances

Please see section 3, *Building H.263 Encoder* for more information on how to use this configuration.

2.3 Encoder Default Parameters

The encoder by default uses a set of creation parameters such as target bitrate, target frame rate, source format, etc. to allocate required memory space and initialise rate control parameters. The `IH263ENC_Params` structure (defined in `ih263enc.h`) is organised as shown in Figure 4.

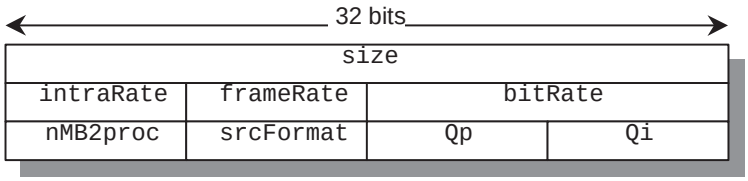


Figure 4. Encoder Creation Parameters – `IH263ENC_Params` (Little Endian)

The table below describes the fields in detail.

Table 1. Encoder Creation Parameters – `IH263ENC_Params`

Name	Description
`size`	Size of `IH263ENC_Params`
`bitRate`	Target bitrate in kbps (default = 512)
`frameRate`	Target frame rate (default = 30)
`intraRate`	Rate of I-frame update (default = 30)
`Qi`	Initial Q value for I-frames (default = 10)
`Qp`	Initial Q value for P-frames (default = 10)
`srcFormat`	Source format (default = `H263_SRCFMT_QCIF`)
`nMB2proc`	Number of MBs to process per call to `h263EncMB` (default = 1)

One can change the default values shown in the table above by editing the structure `IH263ENC_PARAMS` defined in `ih263enc.c`.

2.4 APIs and Example Code

Shown below is how the IALG functions structure `IH263ENC_Fxns` (defined in `ih263enc.h`) looks like.

```
typedef struct IH263ENC_Fxns
{
    IALG_Fxns   ialg;    /* IH263DEC extends IALG */

    void        (*control)(IH263ENC_Handle  handle,
                           IH263ENC_Cmd     cmd,
                           void            *input);

    void        (*encode) (IH263ENC_Handle  handle,
                           uchar           *in[3],
                           uint            *out);
} IH263ENC_Fxns;
```

ialg: This is the default IALG functions. Please refer to appropriate TMS320 Algorithm Standard documents for more information.

control: This function is used to obtain updated status from the encoder.

encode: Executes the H.263 encoder.

Shown below is an example code, in which one parent instance and one child instance are created.

Please refer to *TMS320 DSP Algorithm Standard Rules and Guidelines* for more information on TMS320 Algorithm Standard specific function APIs.

```
void main()
{
    H263PENC_TI_Obj *encParent; /* encoder parent handle   */
    IH263ENC_Handle  encHandle; /* encoder child handle    */
    IH263ENC_Status  encStatus; /* encoder status          */
    unsigned char   *in[3];     /* input frame (Y, Cb, Cr) */
    unsigned int    *out;       /* output bitstream        */

    /* create encoder parent instance */
    encParent = (H263PENC_TI_Obj *)ALG_create((IALG_Fxns *)&H263PENC_TI_IALG,
                                              NULL,
                                              (IALG_Params *)NULL);
    /* create encoder child instance */
    encHandle = (H263ENC_TI_Obj *)ALG_create((IALG_Fxns *)&H263ENC_TI_IH263ENC,
                                             encParent,
                                             (IALG_Params *)NULL);
    /* clear encoder status structure */
    H263ENC_TI_IH263ENC.control((IH263ENC_Handle)encHandle,
                                IH263ENC_CLR_STATUS,
                                &encStatus);
    while(1)
    {
        /* get pointer to input video frame -> in        */
        /* get pointer to output bitstream buffer -> out */
        /* execute H.263 encoder                         */
        H263ENC_TI_IH263ENC.encode((IH263ENC_Handle)encHandle,
                                   (uchar **)&in,
                                   out);
        /* get encoder status */
        H263ENC_TI_IH263ENC.control((IH263ENC_Handle)encHandle,
                                    IH263ENC_GET_STATUS,
                                    &encStatus);
    }
}
```

2.5 H.263 Encoder Structures

The figure below shows how the parent object `H263PENC_TI_Obj` (defined in `h263penc_ti.h`) and the encoder tables (defined in `enctabs.asm`) are structured.

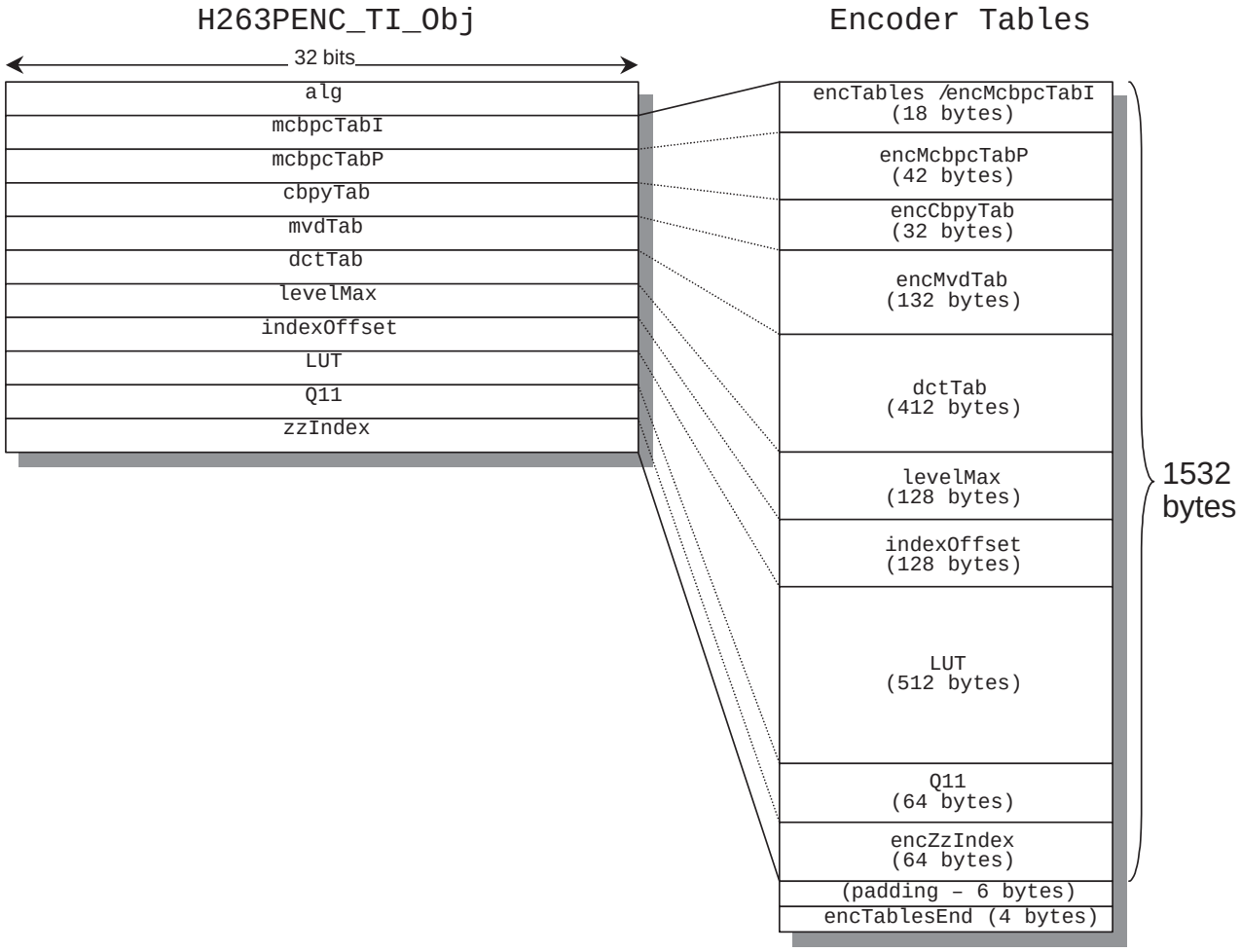


Figure 5. Parent Object – `H263PENC_TI_Obj`

Table 2. Parent Object – `H263PENC_TI_Obj`

Table Name	Description
`alg`	Default IALG object
`mcbpcTabI`	VLC table for MCBPC (MB type and CBP for chroma for INTRA MB)
`mcbpcTabP`	VLC table for MCBPC (MB type and CBP for chroma for INTER MB)
`cbpyTab`	VLC table for CBPY (CBP for luma)
`mvdTab`	VLC table for MVD (Motion Vector Difference)
`dctTab`	
`levelMax`	
`indexOffset`	
`LUT`	Loop-up table used by Motion Estimation to avoid division
`Q11`	Look-up table used for quantisation to avoid division
`zzIndex`	Zigzag index table

The figure below shows how `H263ENC_TI_Obj` (defined in `h263enc_ti.c`) is structured. The shaded field (`rtpParam`) is optional.

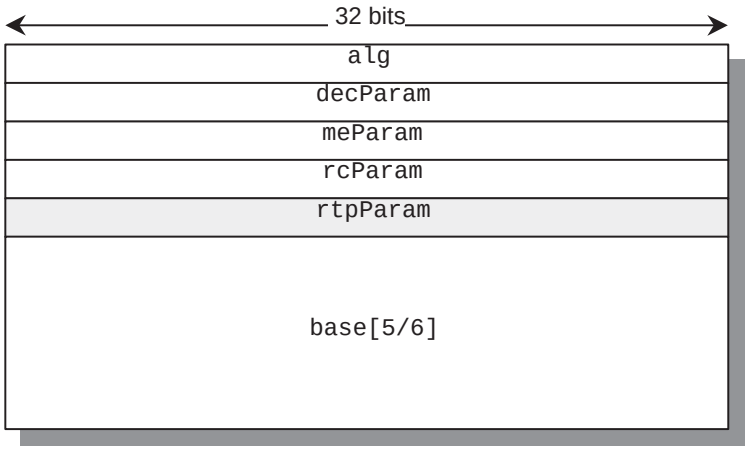


Figure 6. Child Object – `H263ENC_TI_Obj` (Little Endian)

Table 3 describes the fields in detail.

Table 3. Encoder Object – `H263ENC_TI_Obj`

Name	Description
`alg`	Default IALG object
`encParam`	Pointer to encoder parameter structure `H263EncParam` (see below)
`meParam`	Pointer to motion estimation parameter structure H263MEParam (see below)
`rcParam`	Pointer to rate control structure H263RCParam (see below)
`rtpParam`	RTP parameter structure (`H263RTPParam`). This is valid only when RTP flag is used at build time. Please see Appendix D, *Real-time Transport Protocol (RTP)* for more information.
`base[5/6]`	Base addresses for all memory spaces allocated for a particular child instance; 5 when used with a parent instance, and 6 without.

The `H263EncParam` structure (defined in `h263encode.h`) is the main structure that the encoder uses to store important information about the current frame that it is reconstructing. The following lists and describes each field in the structure. The structure fields between `tr` and `qp`, inclusive, are information extracted from the bitstream. Please refer to H.263 specification for more detail.

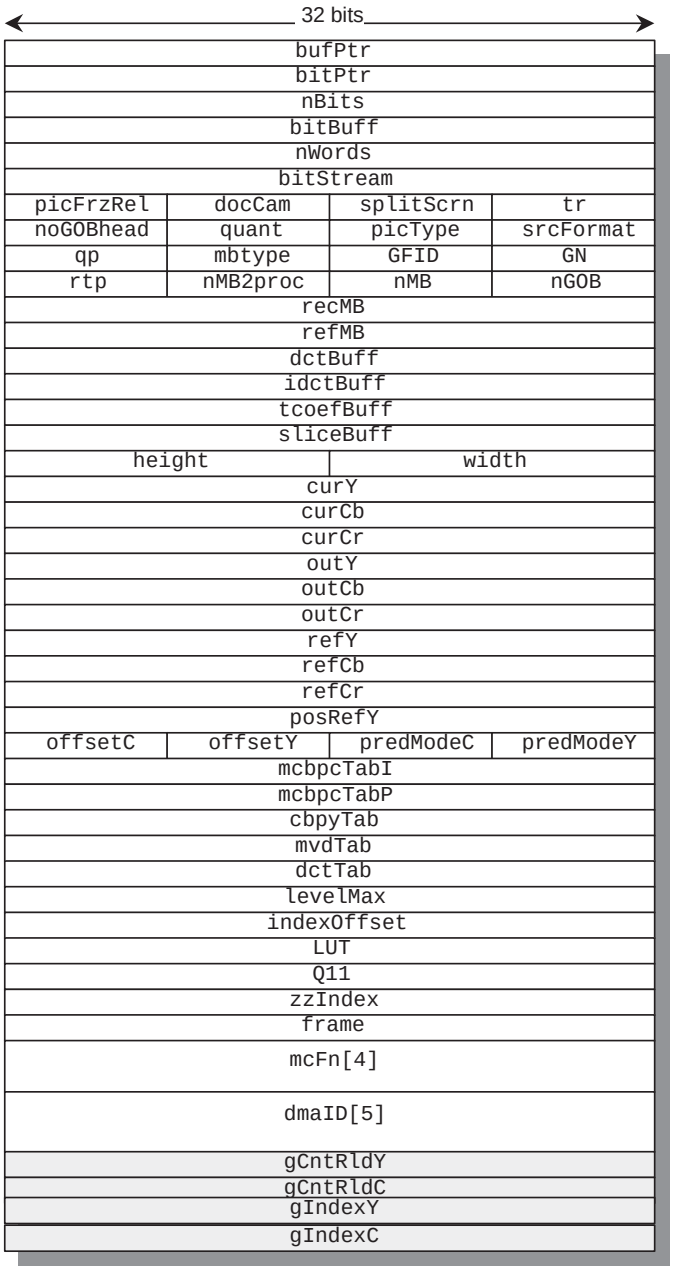


Figure 7. Encoder Parameters – `H263EncParam` (Little Endian)

Table 4. Encoder Parameters – H263EncParam

Name	Description
bufPtr	Points to the current 32-bit word of the H.263 bitstream
bitPtr	Bit position, or number of remaining bits in the current 32-bit word; MSB=32; LSB=1.
nBits	Number of bits produced for current frame
bitBuff	Points to temporary storage for bitstream
nWords	Number of words produced for entire sequence
bitStream	Points to memory space allocated by application to store bitstream for entire frame
tr	Temporal reference
splitScrn	Split screen indicator
docCam	Document camera indicator
picFrzRel	Full picture freeze release
srcFormat	Source format
picType	Picture coding type
quant	Picture quant
noGOBhead	“No GOB header” indicator
GN	Group number
GFID	GOB frame ID
mbtype	MB type
qp	Quantiser information
nGOB	Number of GOB per frame
nMB	Number of MB per GOB
nMB2proc	Number of MBs to process per call to the h263EncMB function. Please see section 2.5.1, *Number of MBs to Process (nMB2proc)*
rtp	’1’ for RTP mode; ’0’ for non-RTP
recMB	Points to reconstructed MB buffer
refMB	Points to reference MB buffer
dctBuff	Points to DCT buffer
idctBuff	Points to IDCT buffer
tcoefBuff	Points to TCOEF buffer used by the tqziq function
sliceBuff	Points to buffer used to hold portions of both input and output frames. Please see section 2.5.1, *Number of MBs to Process (nMB2proc)*
width	Width of image
height	Height of image

Name	Description
curY curCb curCr	Pointers to current frame buffer (luma, Cb, and Cr, respectively)
outY outCb outCr	Pointers to output frame buffer (luma, Cb, and Cr, respectively)
refY refCb refCr	Pointers to reference frame buffer (luma, Cb, and Cr, respectively)
posRefY	Position of reference luma inside refMB
predModeY predModeC	Half-pel modes for luma and chroma, respectively
offsetY offsetC	Offsets inside reference MB for luma and chroma, respectively. Please see section 2.5.2, *Reference Offsets (offsetY and offsetC)* for more information.
mcbpcTabI mcbpcTabP cbpyTab mvdTab dctTab levelMax indexOffset LUT Q11 zzIndex	Pointers to VLC tables for MCBPC (I-frame), MCBPC (P-frame), CBPY, MVD, and tables used to encode TCOEF.
frame	Number of frames encoded
mcFn[4]	Array of function pointers for MC kernels. Please see section 2.5.3, *Motion Compensation Kernels (mcFn_t mcFn[4])* for more information.
dmaID[5]	DMA/EDMA ID's. Please see section2.5.4, *DMA/EDMA ID's (dmaID[5])* for more information.
gCndRldY	Handle to a global count reload register used by luma transfers. This is valid only when CHIP_6201 and CSLDMA flags are specified at build time.
gCndRldC	Handle to a global count reload register used by chroma transfers. This is valid only when CHIP_6201 and CSLDMA flags are specified at build time.
gIndexY	Handle to a global index register used by luma transfers. This is valid only when CHIP_6201 and CSLDMA flags are specified at build time.
gIndexC	Handle to a global index register used by chroma transfers. This is valid only when CHIP_6201 and CSLDMA flags are specified at build time.

The H263MEParam structure (defined in h263encode.h) is the structure used to store information returned by the motion estimation routine. The following lists and describes each field in the structure.

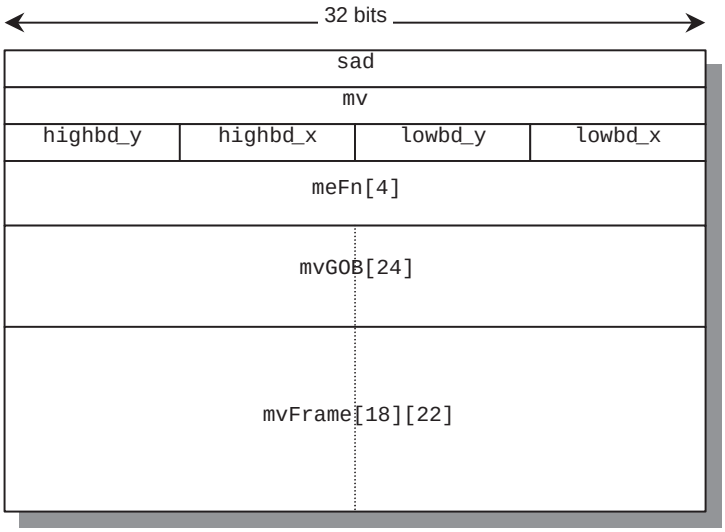


Figure 8. Motion Estimation Parameter – `MEParam` (Little Endian)

Table 5. Motion Estimation Parameter – `MEParam`

Name	Description
`sad`	SAD returned by Motion Estimation
`mv`	Motion vector for current MB
`lowbd_x` `lowbd_y`	Low bounds for motion search
`highbd_x` `highbd_y`	High bounds for motion search
`meFn`	Function pointers for SAD functions
`mvGOB`	Motion vectors for previous/current GOB
`mvFrame`	Motion vectors for previous/current frame

The `H263RCParam` structure (defined in `h263encode.h`) is used by the rate control routine. The following lists and describes each field in the structure.

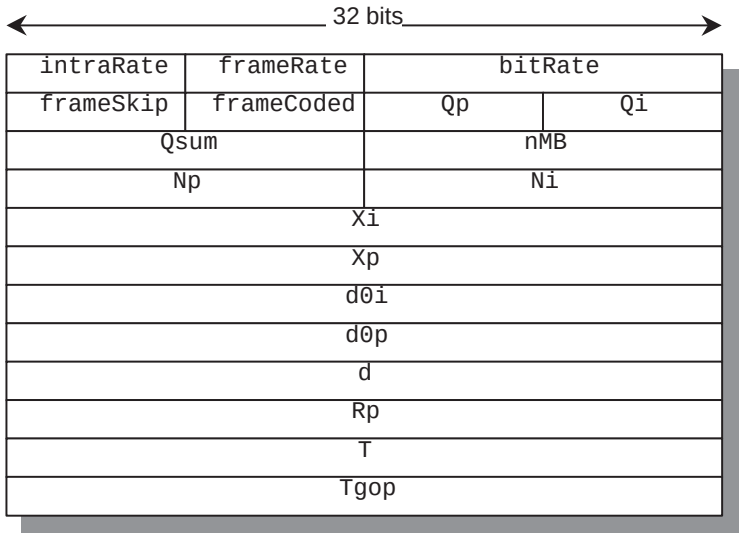


Figure 9. Rate Control Parameter – `RCParam` (Little Endian)

Table 6. Rate Control Parameter – `RCParam`

Name	Description
`bitRate`	Target bitrate
`frameRate`	Target frame rate
`intraRate`	Rate of I-frame update
`Qi` `Qp`	Initial Q values for I-frames and P-frames
`frameCoded`	Number of coded frames
`frameSkip`	Number of skipped frames
`nMB`	Number of MBs in a frame
`Qsum`	Sum of Q values used
`Ni` `Np`	Number of I-frames and P-frames
`Xi` `Xp`	Picture complexities for I-frames and P-frames

`d0i`
`d0p`
`d`
	Virtual bitstream buffer fullness for I-frames and P-frames

`Rp`	Reaction parameter
`T` `Tgop`	Target number of bits for a frame and a group of pictures

Shown below is how the H.263 encoder status structure `IH263ENC_Status` (defined in `ih263enc.h`) is organised.

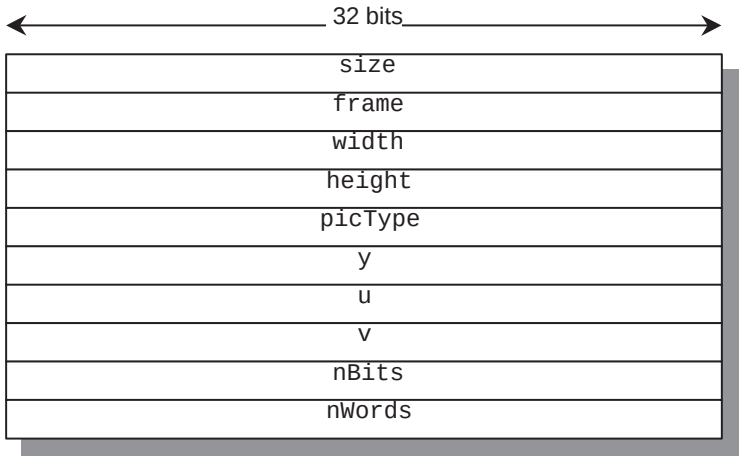


Figure 10. Encoder Status – `IH263ENC_Status`

Table 7. Encoder Status – `IH263ENC_Status`

Name	Description
`size`	Default
`frame`	Number of frames encoded
`width`	Width of encoded frame
`height`	Height of encoded frame
`picType`	Picture type of encoded frame
`y`	Address of encoded frame (luma)
`u`	Address of encoded frame (Cb)
`v`	Address of encoded frame (Cr)
`nBits`	Number of bits produced by current frame
`nWords`	Number of words produced by entire sequence

2.5.1 Number of MBs to Process (`nMB2proc`)

On TMS320C620x, the internal data memory is equal to, or greater than, 64KB, which is sufficiently large to hold an entire CIF GOB for the encoder to process. On TMS320C6211 (and TMS320C6711), however, both the program and data must share 64KB of L2 memory. In most cases, the recommended use of L2 memory is 16KB SRAM and 48KB 3-way cache for optimal performance. Since the amount of on-chip memory is no longer large enough to allocate a whole GOB, the encoder must process each GOB in pieces. The current implementation of the H.263 encoder allows the user to tell the encoder how many MBs it can process at any one time; for the recommended use of L2 memory, this will be one.

This feature can also be used in TMS320C620x as well, in case the system is not able to allocate the required memory to process the whole GOB.

The diagram below illustrates how the encoder uses `nMB2proc` to allocate `sliceBuff`.

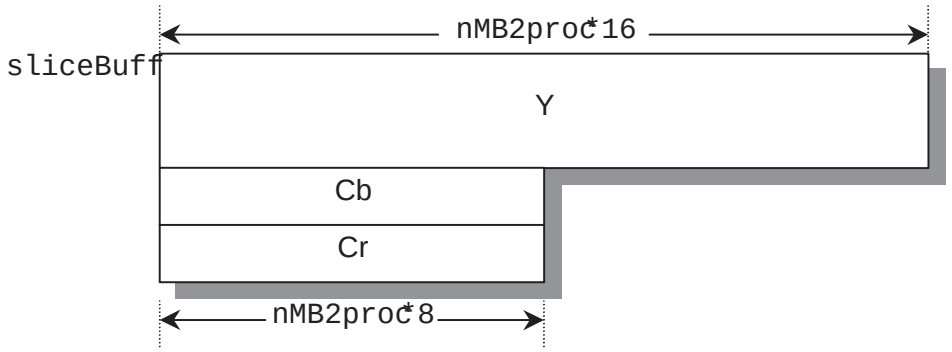


Figure 11. How nMB2proc is Used to Allocate SliceBuff

Please note that if one assigns an invalid number to nMB2proc (a number less than one, or larger than the number of MBs per GOB in a specified format), the encoder will limit this value to the valid range [1, number of MBs/GOB].

2.5.2 Reference Offsets (offsetY and offsetC)

Half-pel motion compensation requires one 17x17 luma block and two 9x9 chroma blocks from the reference frame buffer. To ensure best performance, all data transfers are done in 32-bit words, i.e. every element count must be a multiple of four bytes. This means that one 20x17 luma block and two 12x9 chroma blocks have to be read from the reference frame buffer.

offsetY refers to the number of bytes from the left edge of the 20x17 block, where the required reference MB exists. Similarly, offsetC refers to the number of bytes from the left edge of 12x9 block, where the required block exists.

Figure below illustrates this concept.

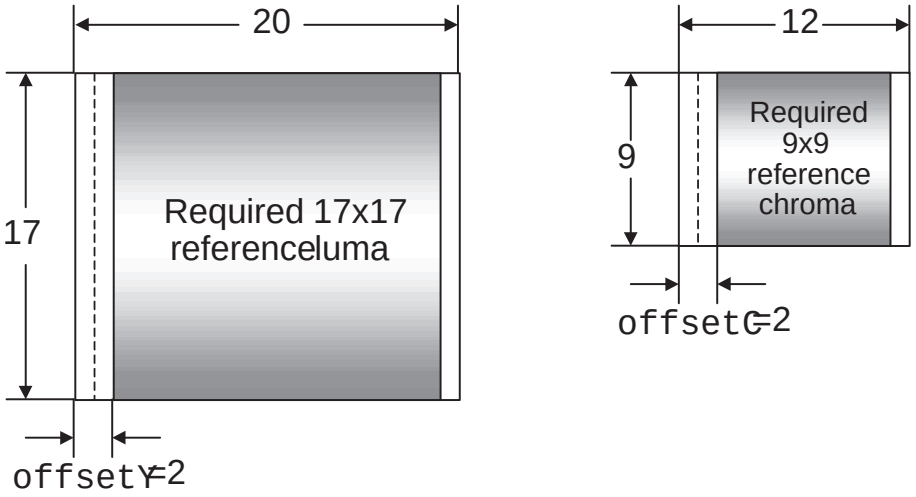


Figure 12. Example of Using offsetY and offsetC

2.5.3 Motion Compensation Kernels (mcFn_t mcFn[4])

In the encoder motion compensation, there are four half-pel modes, requiring four separate functions. Due to the complexity involved in determining which functions to use, an array of function pointers mcFn is used to store addresses for all four motion compensation kernels. The use of this array eliminates the need for nested if-then-else and switch-case statements, thereby improving both the performance and overall code size.

All four functions have the following API (defined in h263.h).

```
typedef void (*mcFn_t)(uchar  *src,    /* source address w/o offset                */
                       uchar *dst,     /* destination address                      */
                       uchar  offset,  /* offset within 20x17 and 12x9 blocks      */
                       int    sWidth,  /* source pitch (default=20 & 12)           */
                       int    dWidth,  /* destination pitch (default=16 & 8)       */
                       int    rc,      /* rounding control; ignored by mcA & mcAi  */
                       short *idct);   /* IDCT coefs; ignored by mcA, mcB, mcC & mcD */
```

During the initialization stage (H263ENC_TI_initObj defined in h263enc_ti.c), the array is set up as shown below. Note that the structure ep is of the type H263EncParam.

```
ep–>mcFn[0] = mcA;
ep–>mcFn[1] = mcB;
ep–>mcFn[2] = mcC;
ep–>mcFn[3] = mcD;
```

Please see section 2.8.3, *Motion Compensation (h263EncMC)* for more information on how the motion compensation is applied to a MB.

2.5.4 DMA/EDMA IDs (dmaID[5])

The current implementation of the encoder only supports the use of the DAT module in the Chip Support Library (CSL).

The encoder needs to keep track of five sets of data transfers into and out of sliceBuff (in and out), refMB (luma and chroma), and bitBuff. When a request is issued via the DAT module, a unique request ID that is associated with that particular request is returned. The dmaID array is used to store separate ID's for each buffer.

- dmaID[0] is the request ID associated with sliceBuff (input).
- dmaID[1] is the request ID associated with refMB (luma).
- dmaID[2] is the request ID associated with refMB (chroma).
- dmaID[3] is the request ID associated with sliceBuff (output).
- dmaID[4] is the request ID associated with bitBuff.

Please refer to *TMS320C6000 Chip Support Library API Reference Guide* for more information on the DAT module.

2.6 Memory Requirements

The memory requirements for the H.263 encoder are shown below.

Table 8. H.263 Encoder Code Sizes (Bytes)

	TMS320C6201	TMS320C6211
Core encoder code	29,408	28,928
Shared code	8,384	8,384
Total	37,792	37,312

The term “shared code” refers to a set of functions that the encoder shares with the decoder.

The difference in code size between TMS320C6201 and TMS320C62111 is due mostly to the scheduler and its ability to take advantage of the different architectures.

Please note that the encoder code size may change depending on which compilation flags are used and which release of the code generation tools are used to build the source codes.

The following tables show the memory requirement for both internal and external memory spaces. The tables for internal memory requirements assume the use of the parent instance to hold encoder tables. If one decides not to use the parent instance, but instead to allow each child instance to keep its own copy of the tables, the internal memory requirement for each child instance will equal to the two totals at the bottom of the table.

Table 9. Internal Memory Requirements (TMS320C6200)

	Size (Bytes)		Alignment
	Parent	**Child**	
FDCT buffer (`dctBuff`)	0	768	16 bytes (0x10)
IDCT buffer	0	896	16 bytes (0x10)
Reconstructed MB buffer (`recMB`)	0	384	16 bytes (0x10)
Reference MB Buffer (`refMB`)	0	2520	16 bytes (0x10)
TCOEF buffer (`tcoefBuff`)	0	768	16 bytes (0x10)
Slice buffer (`sliceBuff`)	0	384 x *N*	16 bytes (0x10)
Bitstream buffer (`bitBuff`)	0	1064 x *N*	16 bytes (0x10)
Encoder parent object (`H26ENC_TI_Obj`)	1536	0	16 bytes (0x10)
Encoder child object (`H263ENC_TI_Obj`)	0	1132	16 bytes (0x10)
Stack	0	456	N/A
Total	0	6468	
	1536	1448x*N*	–

N = nMB2proc

Table 10. External Memory Requirements

	Size (Bytes)	Alignment
Encoder tables (original)	1,532	16 bytes (0x10)
Frame buffer 0	152,096	16 bytes (0x10)
Frame buffer 1	152,096	16 bytes (0x10)
Total	305,724	–

Please note that the encoder neither imposes nor assumes the placement of these buffers. However, for optimal performance, it is recommended that the buffers be placed as indicated above. One should not encounter any cache coherency problems when running the encoder on TMS320C6211/TMS320C6711/TMS320C64x.

The IDCT code for TMS320C6200 requires its input buffer to have extra 128 bytes (or 64 shorts) to be used as scratch memory, hence 896 bytes, rather than 768 bytes. The code for TMS320C64x, however processes its input data completely in place, and needs no extra space.

In the default configuration, the original encoder tables are located in external memory, and during initialisation, the tables are copied to internal memory for optimal performance. This is done by the parent instance only once, and all child instances are then set up to access the same copy. The exception is when the user decides not to use the parent instance, in which case each child instance copies its own copy of the tables into its memory space.

The encoder requires at least two frame buffers to correctly reconstruct the frames: one for reference, and the other for current (being reconstructed), which becomes the reference frame for the next frame. A single CIF frame occupies 152,064 bytes, but the encoder requires 32 additional bytes due to the way a 48x48 reference block is transferred into refMB. Each data transfer reads in 48 pixels wide, for optimal DMA performance. If this reference frame buffer is somehow placed at the edge of a memory space, the first two, or the last two, transfers will not read the data properly – the DMA/EDMA will attempt to access invalid memory space. The extra padding (16 bytes on either end of the frame buffer) ensures that the transfers occur correctly no matter where the buffer is located.

2.6.1 Memory Maps

The figure below shows the memory map used for TMS320C6201 EVM and TMS320C6211 DSK. Please refer to *TMS320C6000 Peripherals Reference Guide* for more information on how to configure L2 memory for TMS320C6211. The size of the external memory on TMS320C6211 DSK may differ; please refer to the documentation provided with the board for more information.

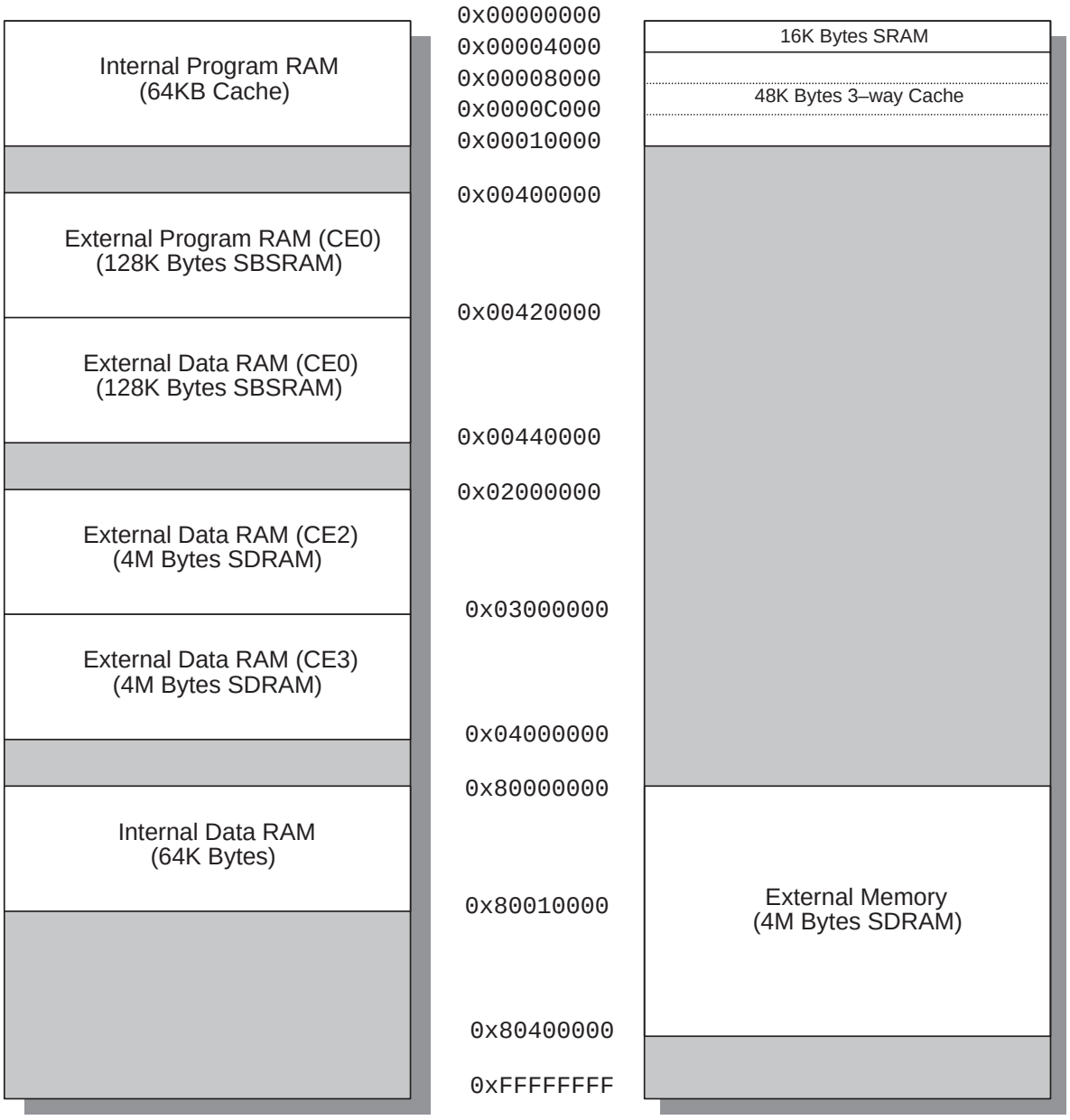


Figure 13. TMS320C6201 EVM & TMS320C6211 DSK Memory Maps

2.7 H.263 Encoder Functions

The table below shows the encoder functions, what they return, and where they are defined. Please refer to individual source code for more information on APIs etc.

Table 11. H.263 Encoder Functions

Function	Description	Returns	Source File
bytealign	Byte aligns the bitstream	void	encutil_sa.sa
calcDFD	Calculates SAD of a MB inside refMB	int	me.c
diffMB	Calculates the difference between current MB and motion compensated MB	void	encutil_sa.sa
enccbp	Encodes MCBPC and CBPY	void	enccbp.sa
encmvd	Encodes MVD	void	encmvd.sa
encvlcI encvlcP	Encodes VLC for INTRA and INTER MB	void	encvlc.sa
fdct	Applies forward DCT	void	fdct.asm
gradVec	Refines best motion vector with half-pel accuracy	void	me.c
h263EncMB	Encodes nMB2proc MBs	void	h263encode.c
h263EncMC	Applies half-pel motion compensation to a MB	void	h263encode.c
h263Encode	Encodes one frame	void	h263encode.c
mesad_a mesad_b mesad_c mesad_d	Calculates SAD of a MB inside refMB in half–pel modes A, B, C, and D	int	me_sa.sa
mesadavg	Calculates SAD of MB and its average pixel values	int	me_sa.sa
putbits	Merges new bits with the rest of the bitstream	void	encutil_sa.sa
rcAllocBits	Bit allocation	void	ratectrl.c
rcCalcQuant	Calculates new quant value	int	ratectrl.c
rcInit	Initialises rate control structure H263RCParam	void	ratectrl.c
rcUpdateFrame	Update rate control information at end of a frame	void	ratectrl.c
rcUpdateGOP	Update rate control information at end of a GOP	void	ratectrl.c
rcUpdateMB	Update rate control information at end of a MB	void	ratectrl.c
rdCurBuff	Reads current frame	void	encutil.c
rdRefC	R eads reference MB (chroma)	void	encutil.c
rdRefY	Reads reference MB (luma)	void	encutil.c
tqziq	Quantise, zigzag scan, and inverse quantise; returns number of symbols and CBP	int	tqziq.c
unpackmb	Unpacks 8-bit values to 16-bit values	void	encutil_sa.sa
wrBits	Writes encoded bitstream in bitBuff to bitStream	void	encutil.c
wrRecMB	Writes reconstructed MB(s) to output	void	encutil.c

NOTE: “uint” and “ushort” are “unsigned int” and “unsigned short”, respectively.

2.8 Code Flow

The following sections describe the order in which the encoder processes a captured frame.

2.8.1 Main Encoder Function (h263Encode)

The figure below shows the high-level code flow.

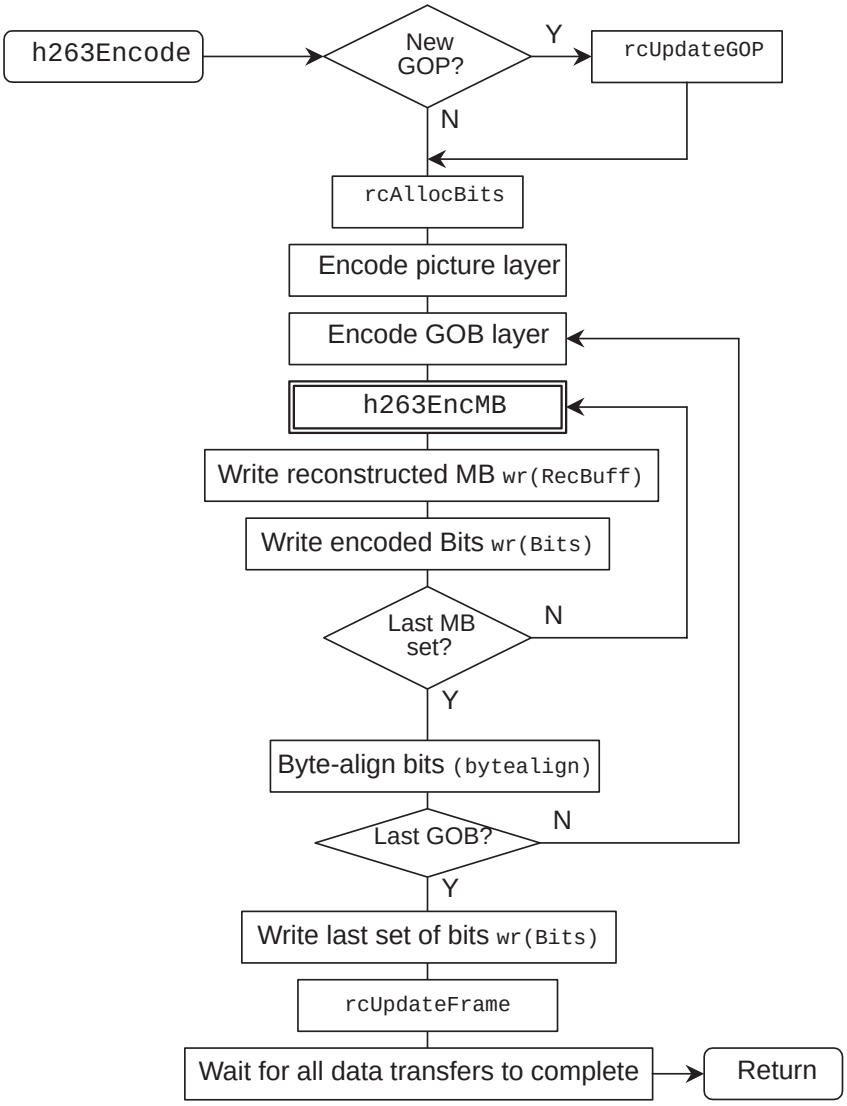


Figure 14. Code Flow – h263Encode

For each frame, the encoder is provided with pointers to the three colour planes and the output bitstream buffer, into which it will copy the encoded bits. The function `h263Encode` starts by allocating bits for entire group of pictures (GOP), and for the current frame, by calling the `rcUpdateGOP` and `rcAllocBits`, respectively. After the bit allocation, the encoder codes the picture layer.

For each GOB, the encoder may or may not encode GOB layer header, depending on whether it is the first GOB or not.

During the creation stage, the encoder is set up to encode one or more MBs per call to the h263EncMB function. The loop is set up to correctly execute this function as often as it needs to for the entire GOB. For example, if the source format is QCIF, and nMB2proc is set to three, then for each GOB, the function h263EncMB will be called four times to process 3, 3, 3, and 2 MBs each time.

After every call to the h263EncMB function, the reconstructed MB(s) is/are written to the output frame by calling the wrRecBuff function, followed by a call to the wrBits function to write out the bitstream. Once the encoder has processed all the MBs in a GOB, the bytealign function is used so that the next GBSC is aligned on a byte boundary.

When all of the GOBs in the current frame has been encoded, the last remaining bits are written to the bitstream buffer, followed by a call to the rcUpdateFrame function to update the information in the rate control.

The final step involves simply making sure that all the data transfer requests have completed.

2.8.2 Enconding MB (h263EncMB)

The first thing the function needs to know is if the current frame is an I-frame or P-frame. In the case of an I-frame, each MB is unpacked using the unpackmb function, forward DCT is applied, followed by quantisation, zigzag scan, and inverse quantisation.

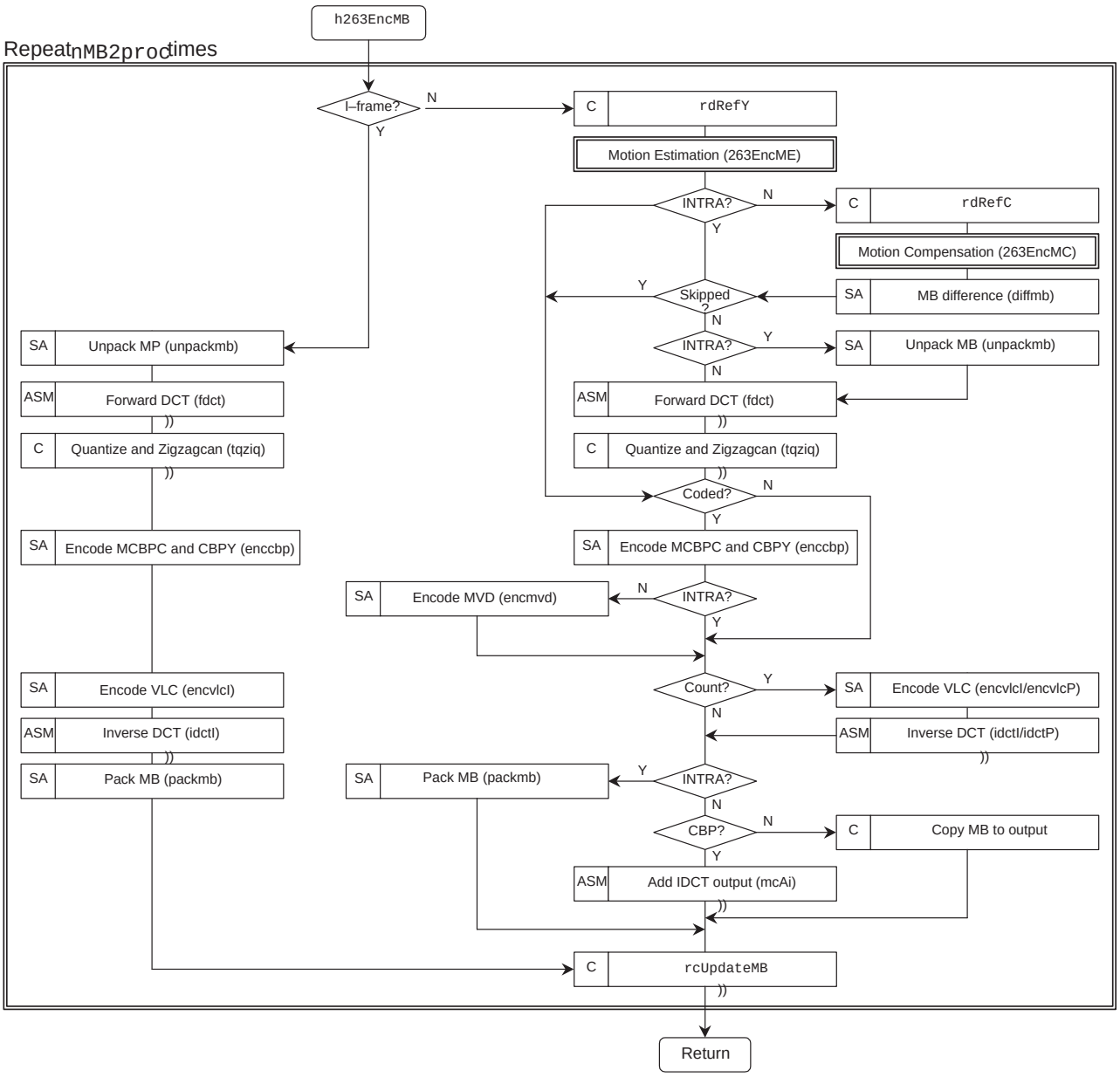


Figure 15. Code Flow – h263EncMB

The figure below shows the bit fields of a value returned by the `tqziq` function.

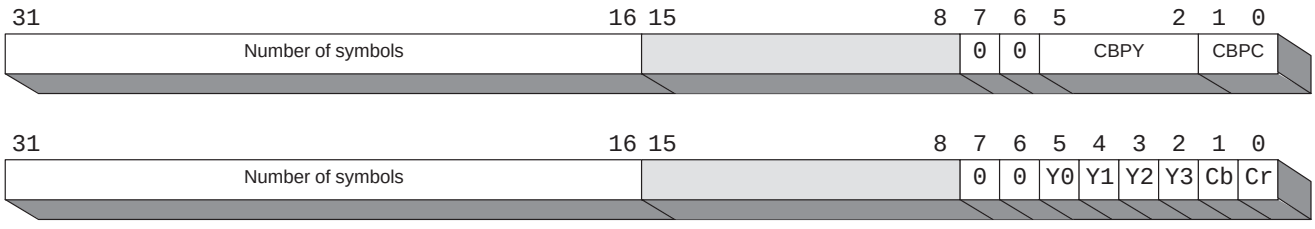


Figure 16. Bit Fields of Value Returned by `tqziq`

The output from the `tqziq` function, including CBP and quantised data are passed to the `enccbp` and `encvlcI` functions to produce the bitstream; the IDCT is applied to the inverse quantised data, followed by the `packmb` function to reconstruct the MB.

Encoding a P-frame involves additional steps, most of which are part of motion estimation. The particular type of motion estimation currently implemented in the encoder uses a hybrid matching gradient technique [8].

IDCT kernels for TMS320C6200 and TMS3206400 have been designed specifically for these families of devices to fully exploit their architectures. For this reason, two separate versions exist; they are not bit-exact, but are IEEE-1180 compliant.

The TMS320C6200 version of the IDCT, because of its highly optimised nature, has two flavours: one for INTRA MB (`idctI`) and another for INTER MB (`idctP`). The only difference between these two is the final precision setting. The `idctI` function outputs unsigned 8-bit values as signed 16-bit values, with an offset of 128, so the encoder must call the `packmb` function to pack and adjust the offsets of the IDCT output.

The TMS320C64x version, however, outputs the results in the identical manner for both INTRA and INTER MB. Unlike the TMS320C6200 version, a final saturation stage is required for INTRA MB, before packing the results.

2.8.3 Motion Compensation (h263EncMC)

The code responsible for applying half-pel motion compensation is shown below.

```
void h263EncMC(H263EncParam *ep, uchar *d[6])
{
    int      posRefY, predModeY, predModeC;
    uchar    offsetY, offsetC, *refMBY, *refMBCb, *refMBCr;
    mcFn_t   mcfY0, mcfY1, mcfY2, mcfY3, mcfCb, mcfCr;

    uchar   *s0,   *s1,   *s2,   *s3,   *s4,   *s5;
    posRefY   = ep->posRefY;
    predModeY = ep->predModeY;  predModeC = ep->predModeC;
    offsetY   = ep->offsetY;    offsetC   = ep->offsetC;
    refMBY    = ep->refMB;
    refMBCb   = ep->refMB+(48*48);
    refMBCr   = ep->refMB+(48*48)+(12*9);

    mcfY0 = ep->mcFn[predModeY];  s0 = refMBY + posRefY +   0;
    mcfY1 = ep->mcFn[predModeY];  s1 = refMBY + posRefY +   8;
    mcfY2 = ep->mcFn[predModeY];  s2 = refMBY + posRefY + 384;
    mcfY3 = ep->mcFn[predModeY];  s3 = refMBY + posRefY + 392;
    mcfCb = ep->mcFn[predModeC];  s4 = refMBCb;
    mcfCr = ep->mcFn[predModeC];  s5 = refMBCr;

    mcfY0(s0, d[0], offsetY, 48, 8, 0, NULL);
    mcfY1(s1, d[1], offsetY, 48, 8, 0, NULL);
    mcfY2(s2, d[2], offsetY, 48, 8, 0, NULL);
    mcfY3(s3, d[3], offsetY, 48, 8, 0, NULL);
    mcfCb(s4, d[4], offsetC, 12, 8, 0, NULL);
    mcfCr(s5, d[5], offsetC, 12, 8, 0, NULL);
}
```

For each block, the required motion compensation kernel is selected based on its prediction mode. Once the appropriate kernels are selected, the code simply calls these kernels. Since all the kernels have the same API, and any excess arguments are ignored accordingly, they can be called in an identical manner.

Consider the following example.

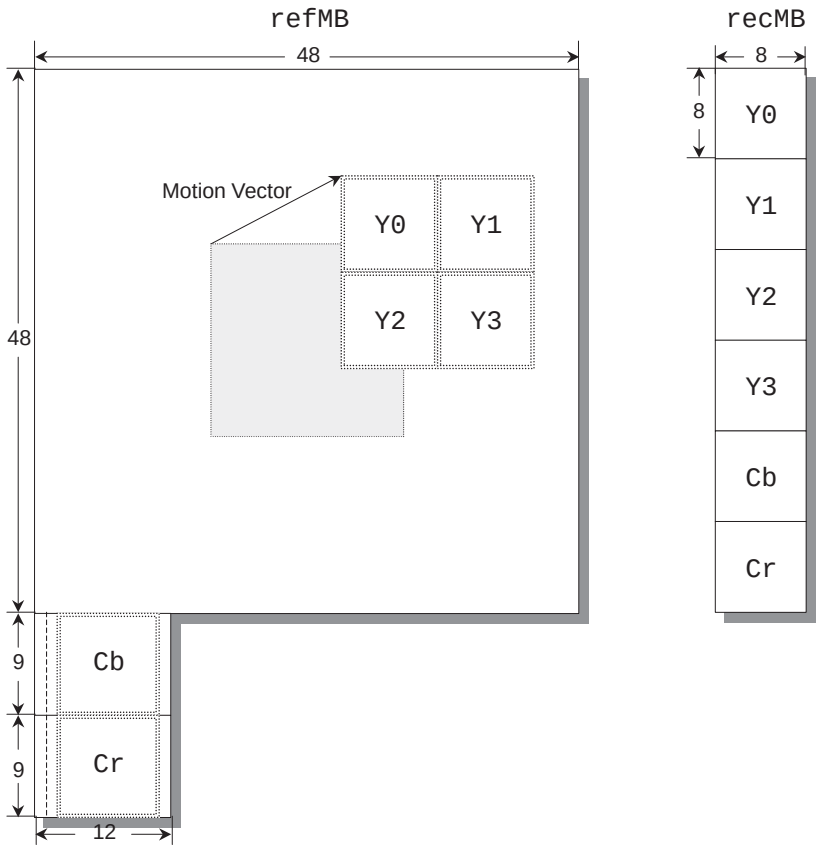


Figure 17. Example of Motion Compensation

The source addresses of `Y0`, `Y1`, `Y2`, and `Y3` are based on the motion vector determined by the `h263EncME` function; the chroma blocks are read in by the `rdRefC` function, based, also on the motion vector, and are calculated accordingly.

The motion compensated blocks are stored contiguously, as shown above.

2.9 Data Flow

The following sections describe how the encoder moves and processes data.

2.9.1 Frame Buffer

The following figure shows how a frame buffer looks for CIF and QCIF, respectively. 16 bytes of padding space is allocated at the beginning of the buffer, as well as at the end, to ensure that DMA/EDMA requests issued by the `rdRefY` function occur properly, regardless of where the buffer is placed inside valid memory space. The pixels are stored contiguously, so for QCIF, the pixels occupy the first 38,016 bytes of the buffer, after the 16-byte padding.

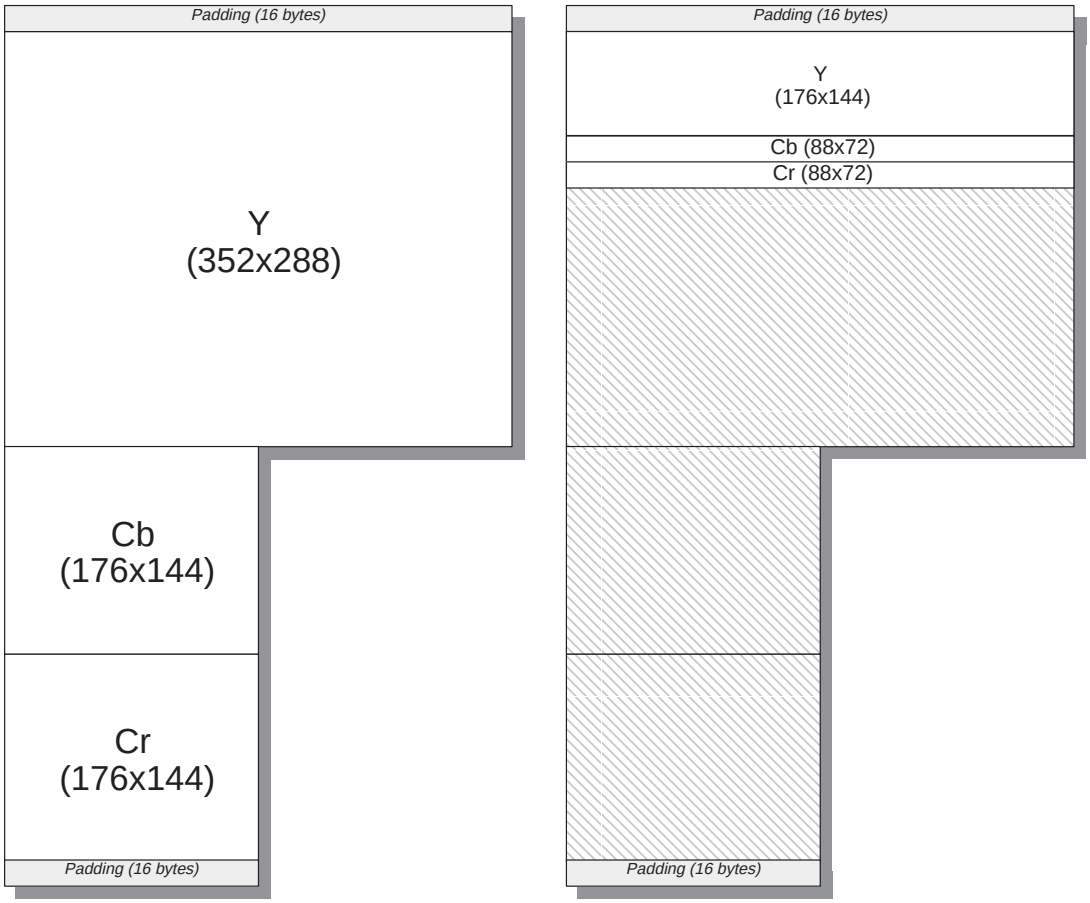


Figure 18. Frame Buffer for CIF and QCIF

2.9.2 Reading Current Data (`rdCurBuff`)

As mention previously, the `h263EncMB` function is capable of processing multiple MBs per GOB. The diagram below illustrates an example of encoding a CIF frame, with `nMB2proc` equal to six, and the `h263Encode` function is bringing in the second set of MBs in the second GOB, prior to calling the `h263EncMB` function.

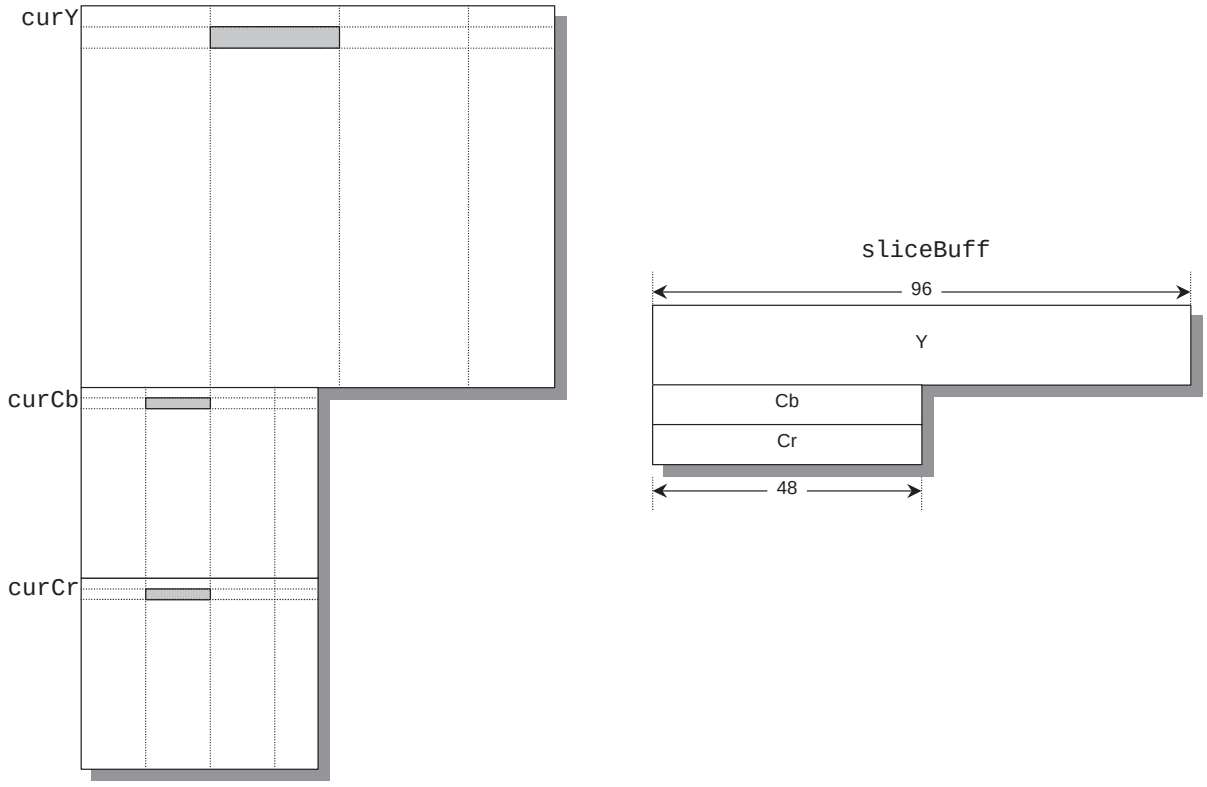


Figure 19. Reading Current Data (nBM2proc = 6)

2.9.3 Reading Reference Luma (rdRefY)

The rdRefY function transfers the 48x48 luma data required by the motion estimation from the reference frame buffer to refMB.

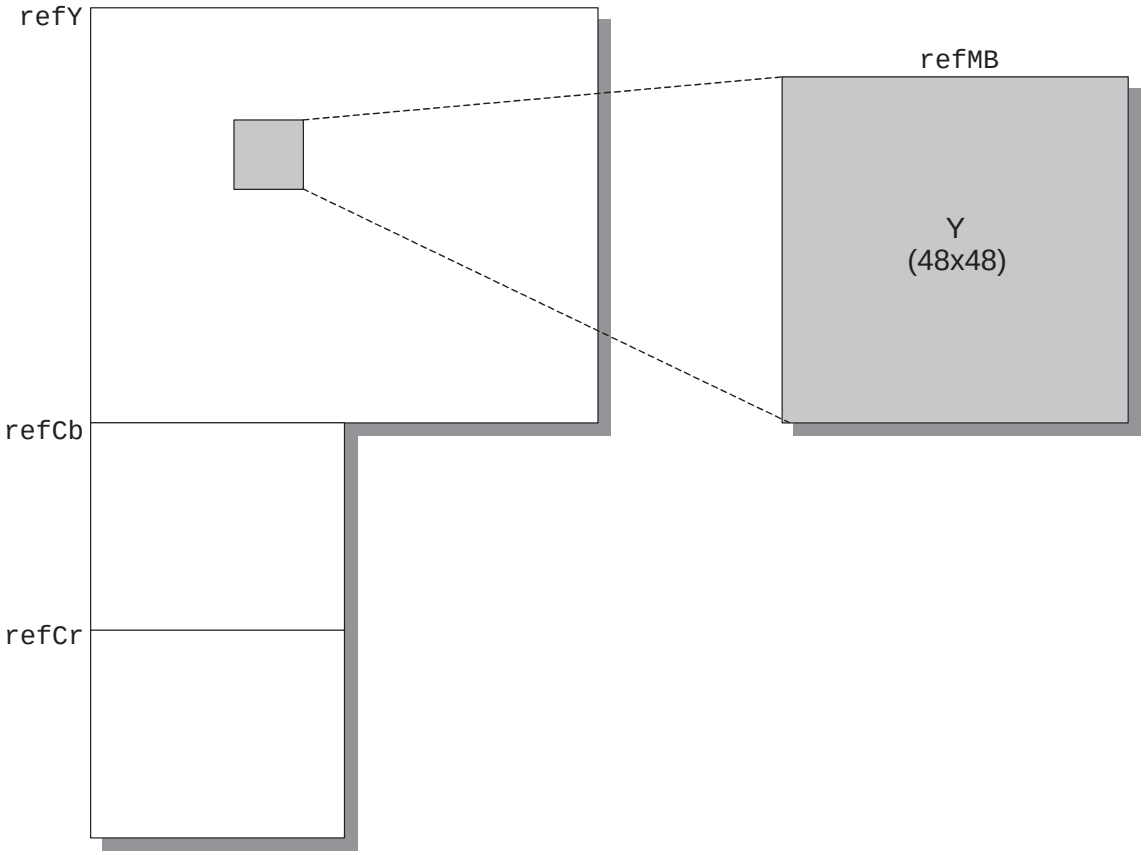


Figure 20. Reading Reference Luma

2.9.4 Reading Reference Chroma (`rdRefC`)

The `rdRefC` function transfers the chroma data (whose locations are determined from motion vector return by the motion estimation) from the reference frame buffer to `refMB+(48x48)`.

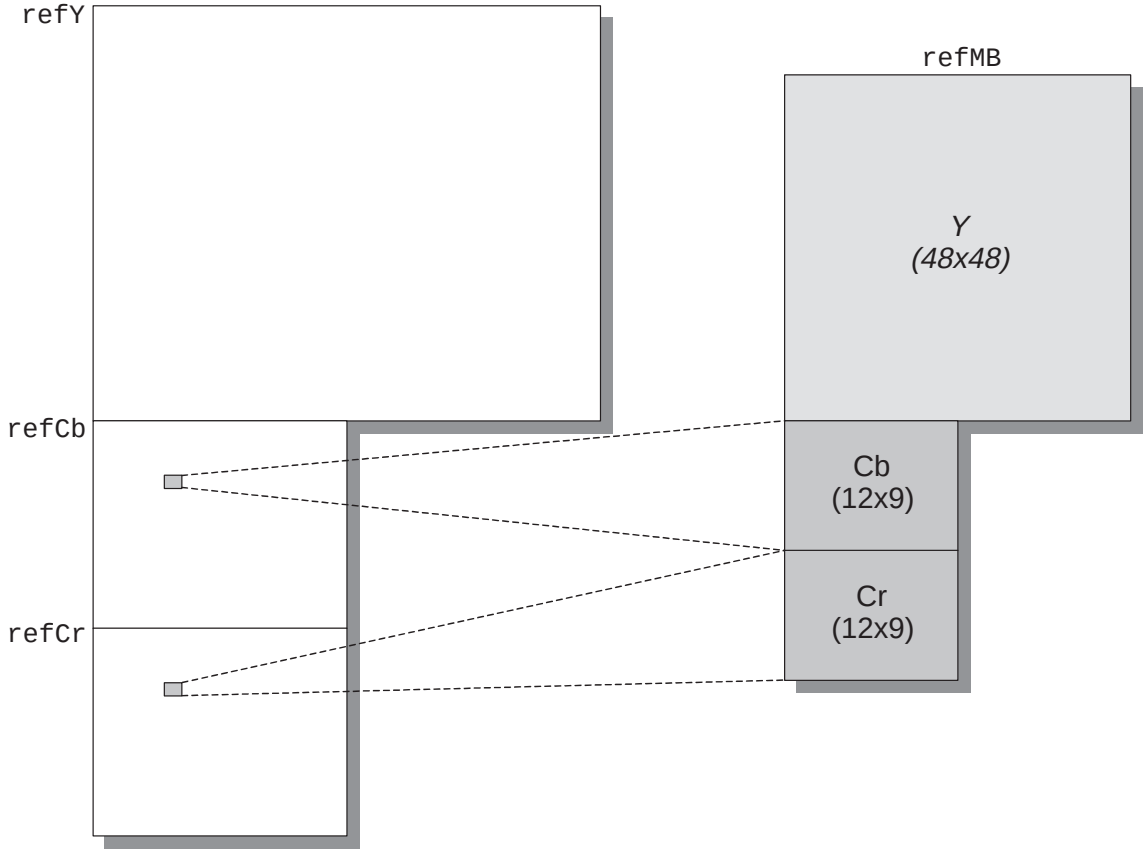
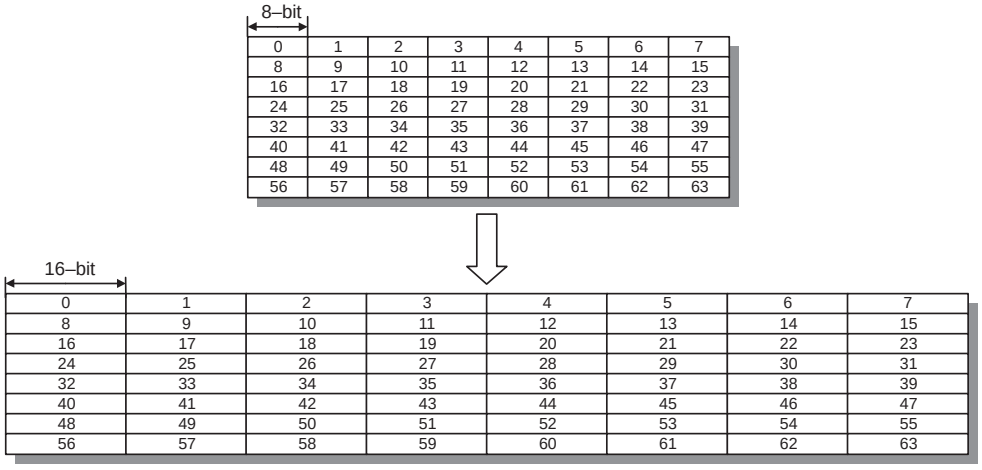


Figure 21. Reading Reference Chroma

2.9.5 Unpacking Current MB (unpackmb)

The FDCT requires its input data to be unsigned 16-bit values. The unpackmb function simply expands the byte data into halfwords and stores them contiguously, while at the same time rearranging the MBs as shown below.



0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

Figure 22. Processing one 8x8 block

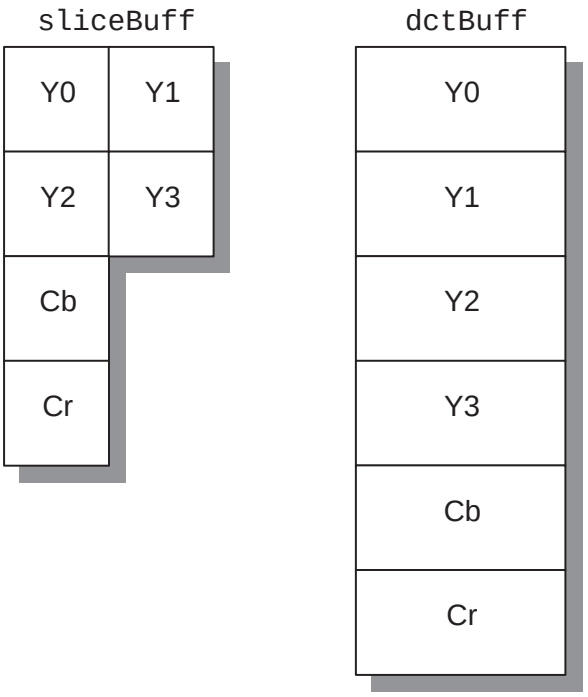


Figure 23. Processing One MB

2.9.6 IDCT (`idctBuff`)

The figures below show how `idctBuff` looks when IDCT coefficients for all six blocks in a MB are coded, and when coefficients for only three blocks are coded, for both TMS320C6200 and TMS320C64x.

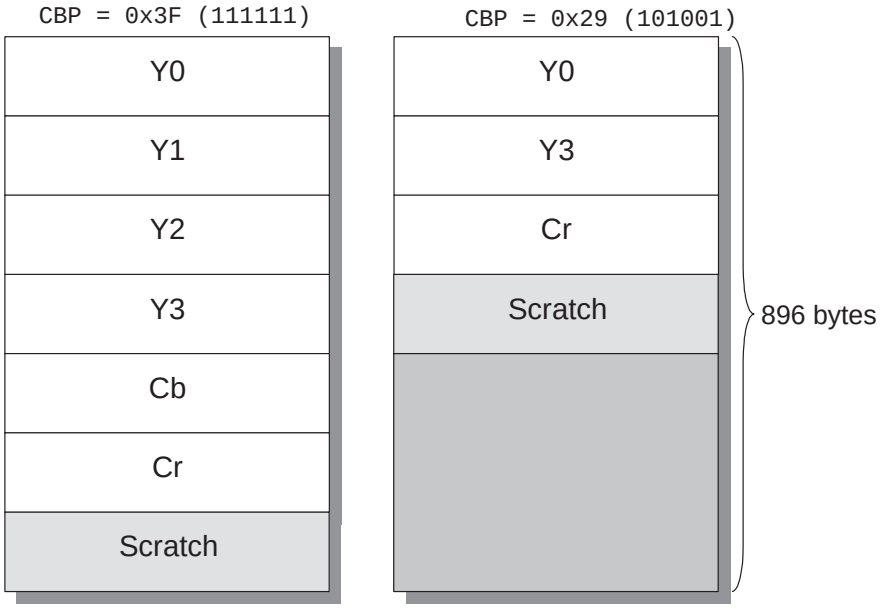


Figure 24. Examples of Using `idctBuff` (TMS320C6200)

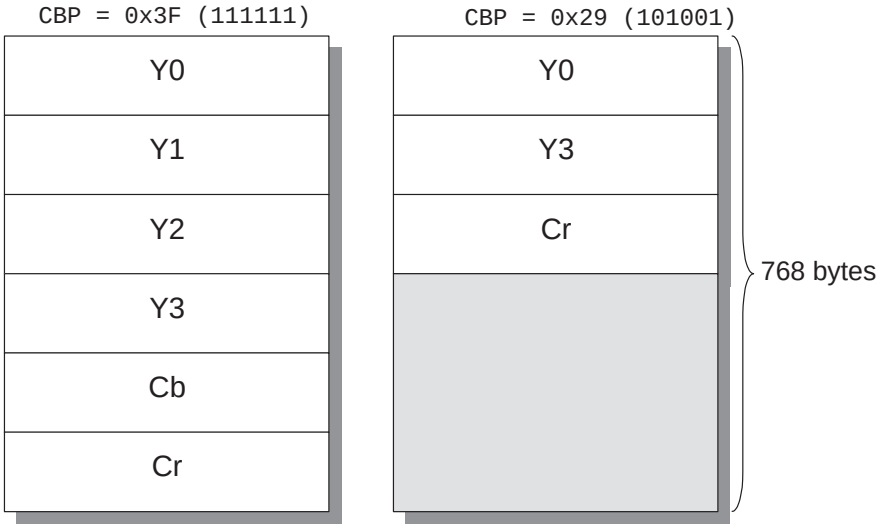


Figure 25. Examples of Using `idctBuff` (TMS320C64x)

2.9.7 Packing INTRA MB (`packmb`)

The encoder uses a highly optimised IDCT kernel that produces signed 16-bit results. For an INTRA MB, however, it produces signed 8-bit results (with an offset of 128) as signed 16-bit values that must be packed into unsigned 8-bit values. The `packmb` function takes signed 16-bit numbers, extracts the lower 8-bits, and applies `XOR` with 0x80, which adjusts the offset of 128, to produce the correct results.

The first figure shows how each block is processed. The second figure shows how an entire MB is processed. The outer-most loop goes around three times, processing two blocks each time.

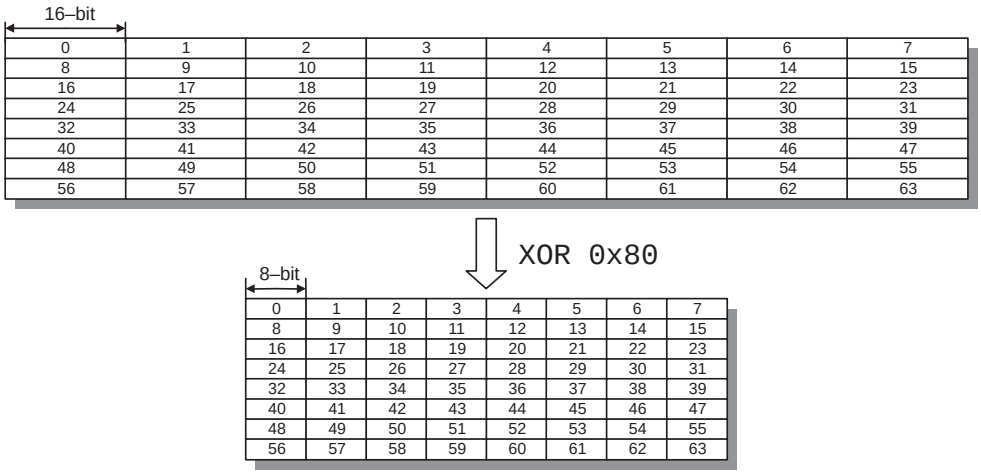


Figure 26. Processing One 8x8 block (TMS320C6200)

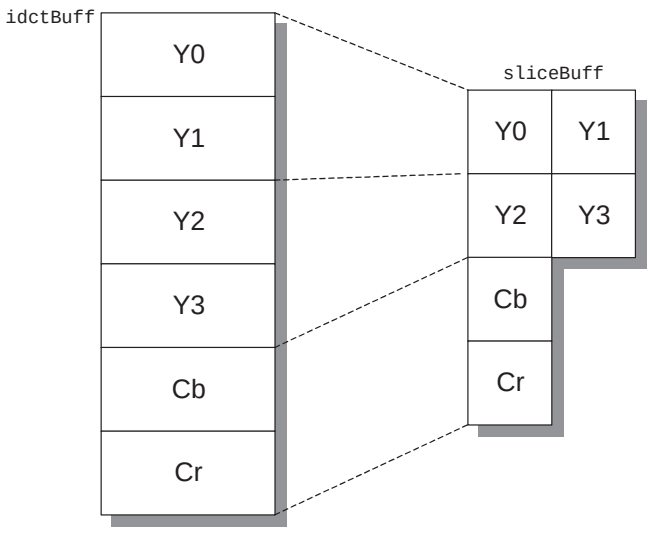


Figure 27. Processing One MB (TMS320C6200)

For optimal performance, different IDCT specifically designed for TMS320C64x is used. Unlike the TMS320C6200 version, this kernel outputs results for INTRA MB without a DC bias, therefore, simple saturation and packing is all that is required.

The figure below shows how each block is processed.

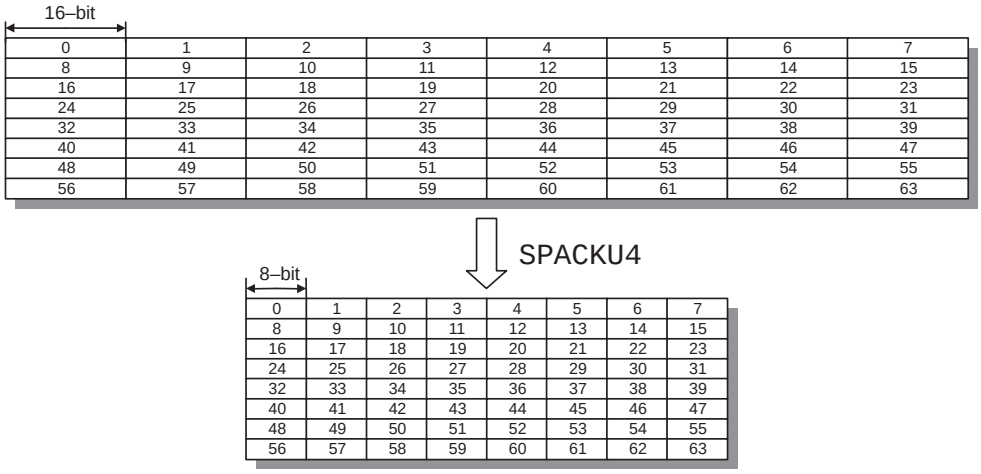


Figure 28. Processing One 8x8 block (TMS320C64x)

Due to the larger register files, the TMS320C64x version of the `packmb` function is able to process all six MBs in parallel.

2.9.8 Writing Reconstructed Data (`wrRecBuff`)

The `wrRecMB` function writes the reconstructed MB to the output frame buffer. The diagram below illustrates an example of encoding a CIF frame, with `nMB2proc` equal to six, and the `h263Encode` function has just completed reconstructing the second set of MBs in the second GOB.

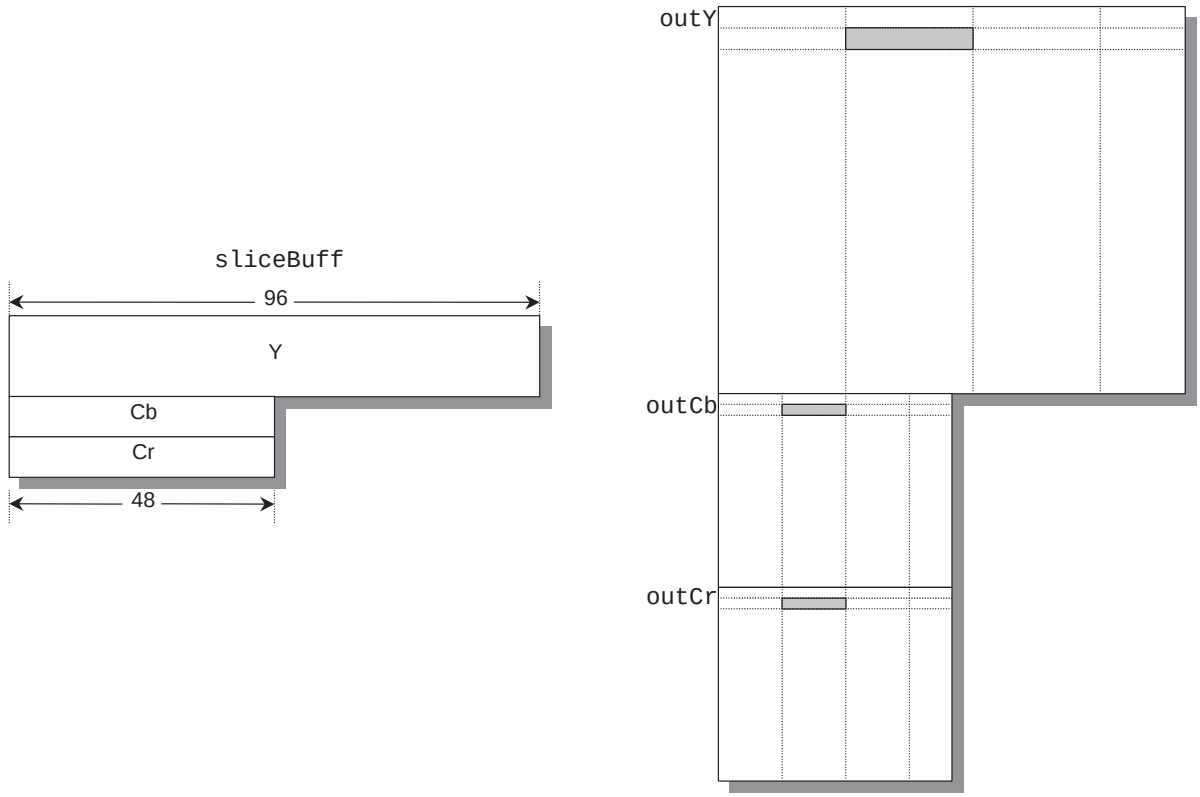


Figure 29. Writing Reconstructed Data

3 Building H.263 Encoder

Sample makefiles `h263e6201.mak` and `h263e6211.mak` (for TMS320C6201 and TMS320C6211/TMS320C6711, respectively) are provided in the `mak` directory for reference. Please note that this makefile is not complete and may not be used to build the complete encoder COFF file. The remainder of this section describes how one can configure the project makefile to one's own system.

3.1 Target Device (REQUIRED)

The user must first select the target device.

CHIP_6201: Target device is TMS320C6201/C6202/C6203/C6204/C6205.

CHIP_6211: Target device is TMS320C6211/C6711.

CHIP_6400: Target device is TMS320C64x.

3.2 Other Flags (Optional)

NOPARENT: Do not create parent instance to store encoder tables – allow each child instance to keep its own copy of the tables. Only h263enc_ti.c has to be compiled with this flag.

RTP: Allocate structure for RTP parameters. Please see Appendix C, *Real-time Transport Protocol* for more information.

ESTATS: Profile the overall encoder performance. Please see Appendix B, *Profiling H.263 Encoder* for more information.

ESTATS_: Profile individual encoder functions. Please see Appendix B, *Profiling H.263 Encoder* for more information.

For files `h263enc_ti.c` and `h263penc_ti.c`, an additional "`-ml0`" flag is required, since it accesses labels that are defined to be of type `far`.

The encoder is also provided with a set of sample linker command files, which include the `MEMORY` and `SECTIONS` directives to help with setting up the user's own linker command file. The files are named so as to identify the target device. For example, a sample linker command file for TMS320C6211 is named `h263e6211.cmd`.

3.3 Building

To build the H.263 encoder, one requires a properly installed Code Composer Studio v1.20 (CCS) or above.

- Open CCS, and open the appropriate project file, or open an existing project and add the necessary files in the `src` and `src6200` or `src6400` directories inside both the `encoder` and `share` directories.
- Include the encoder's `inc` directory as part of the "Include Search Path", so that the application knows about the encoder's IALG APIs.
- Select a target device and add the symbol to the "Define Symbols".
- Select a preferred data transfer method and add the symbol to "Define Symbols", if necessary.
- Add the "`-ml0`" flag to `h263enc_ti.c` and `h263penc_ti.c`, if using a makefile other than the sample provided with the release.
- Add the linker command file for the target device, or edit an existing linker command file by adding the required parts from the sample linker command file.
- Make other changes to the options, including the final COFF file name, map file name, any external libraries such as an RTS library, etc., and build.
- The default location for the COFF file and the map file is the `bin` directory.

Please refer to *TMS320C6000 Optimizing C Compiler User's Guide* for more information regarding the different compiler and linker options.

4 Assumptions and Requirements

The following lists assumptions and requirements for the encoder.

- Baseline H.263 encoder implemented.
- No big endian support for assembly (ASM) and serial assembly (SA) codes. Please refer to individual source file for more information.

- The output buffer that is passed to the encoder to store the bitstream must be aligned on a 32-bit boundary.

Other assumptions and/or requirements may apply. Please refer to individual source file for more information.

References

1. *ITU-T H.263 Video Coding for Low Bit Rate Communication*, January 1998.
2. *eXpressDSP™ Algorithm Standard Rules and Guidelines* (SPRU352), September 1999
3. *Code Composer Studio User's Guide* (SPRU328), 1999
4. *TMS320C62x/C67x CPU and Instruction Set Reference Guide* (SPRU189), 1998
5. *TMS320C6000 Peripherals Reference Guide* (SPRU190), 1999
6. *TMS320C6000 Chip Support Library API Reference Guide*, 2000
7. *TMS320C6000 Optimizing C Compiler User's Guide* (SPRU187), 1999
8. Zhou, M., *Implementation of Hybrid Matching Gradient Motion Estimation for H.263 Real-Time Video Encoding on TI TMS320C6X*, 1999

Appendix A Performance

Table A–1 shows the performances of all the kernels used by the encoder. Please note that the performance numbers for the SA codes may change depending on which release of the compiler (and which options) is used to build them. The numbers were obtained by compiling the codes with release 4.00 of the code generation tools, using the following options: "`–mtx –mh256 –o3`".

Table A–1. Kernels Performance (TMS320C6200)

Function	SA/ASM	No. of Cycles	Per
bytealign	SA	15	Call
diffmb	SA	336	MB
enccbp	SA	25	MB
encmvd	SA	39	MB
encvlcI	SA	10	Symbol
encvlcI	SA	9	Symbol
fdct	ASM	160	Block
idctI	ASM	168	Block
idctP	ASM	168	Block
mcA	ASM	28	Block
mcAi	ASM	109	Block
mcB	ASM	88	Block
mcC	ASM	90	Block
mcD	ASM	133	Block
mesad_a	SA	256	MB
mesad_b	SA	352	MB
mesad_c	SA	352	MB
mesad_d	SA	384	MB
mesadavg	SA	272	MB
packmb	SA	192	MB
putbits	SA	15	Call
unpackmb	SA	192	MB

No specific numbers are available for TMS320C6400 at the time of print.

Appendix B Profiling H.263 Encoder

The H.263 encoder is equipped with a simple profiling capability that uses the device's timer via the CSL TIMER module. To enable this function, specify ESTATS or ESTATS_ flag at build time. Figure B–1 is the structure that the encoder uses to store profiling information (defined in `h263encode.h`).

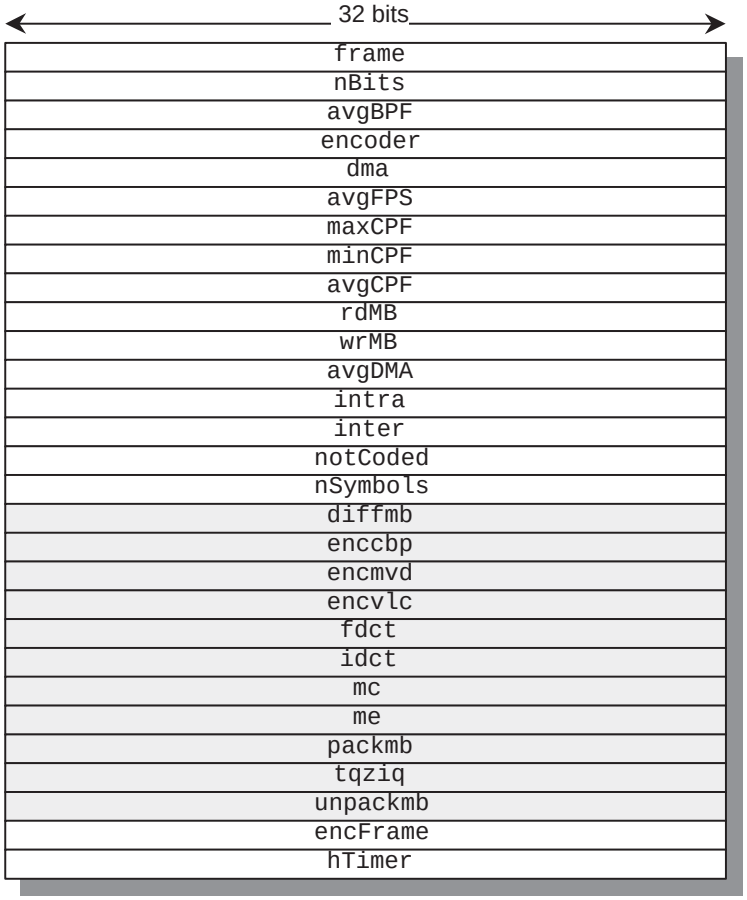


Figure B–1. Encoder Statistics – `H263EncStats`

Table B–1. Encoder Statistics – H263EncStats

Name	Description
frame	Number of frames encoded. (Valid with ESTATS and ESTATS_)
nBits	Total number of bits processed
avgBPF	Average number of bits per frame
encoder	Total number of TIMER cycles used by the encoder
dma	Total number of TIMER cycles used to setup and issue data transfer requests
avgFPS	Average frames per second (equals to CPU clock frequency divided by avgCPF)
maxCPF	Maximum number of cycles per frame
minCPF	Minimum number of cycles per frame
avgCPF	Average number of cycles per frame
rdMB	Total number of TIMER cycles used by rdRefY and rdRefC
wrMB	Total number of TIMER cycles used by wrRecBuff
avgDMA	Average number of cycles per frame used by DMA/EDMA
intra	Total number of INTRA MB
inter	Total number of INTER MB
notCoded	Total number of MB that was not coded
nSymbols	Total number of symbols coded
diffmb	Total number of TIMER cycles used by diffmb
enccbp	Total number of TIMER cycles used by enccbp
encmvd	Total number of TIMER cycles used by encmvd
encvlc	Total number of TIMER cycles used by encvlcI and encvlcP
fdct	Total number of TIMER cycles used by fdct
idct	Total number of TIMER cycles used by idctI and idctP, or idctIP
mc	Total number of TIMER cycles used by h263DecMC, including all the kernels in mc_asm.asm
me	Total number of TIMER cycles used by h263DecME
packmb	Total number of TIMER cycles used by packmb
tqziq	Total number of TIMER cycles used by tqziq
unpackmb	Total number of TIMER cycles used by unpackmb
encFrame	Number of cycles used by encoder per frame; this is set to 0 before every call
hTimer	Handle to the allocated timer through the CSL's TIMER module

The term "TIMER cycles" refers to the value stored in the device's timer register, as opposed to the actual CPU cycles. When an internal timer source is selected, every timer tick is equal to four CPU cycles. This enables the use of the timer for four times as many cycles, before an overflow can occur. Please refer to *TMS320C6000 Peripherals Reference Guide* and *TMS320C6000 CSL Support Library API Reference Guide* for more information on how to configure and use the timer.

Please note that although it is possible to specify both ESTATS and ESTATS_, and while the numbers for individual kernels will be accurate, the numbers specific to ESTATS will be much larger than they would be if ESTATS was specified alone. This is due to the fact that by specifying ESTATS_, the overall encoder performance number includes the time taken to record the kernel performance figures as well. When used separately, however, the overhead associated with the recording is minimal.

Shown below is an example pseudo code on how to use this feature.

```
#include <timer.h>
#include <h263encode.h>
far int cpuClock; /* system clock frequency (MHz) */
/* function prototypes for encoder statistics */
#if ( (ESTATS) || (ESTATS_) )
extern H263EncStats estats; /* defined in h263encode.c */
void encStatsInit(TIMER_HANDLE hTimer);
void encStatsUpdate(H263EncParam *ep);
#endif
void main()
{
    TIMER_HANDLE  hTimer1;  /* timer handle                */
    uint          timerCtl; /* timer control word          */
    H263EncParam *ep;       /* encoder parameters          */
    uint         *stream;   /* pointer to bitstream buffer */
    hTimer1 = TIMER_Open(TIMER_DEV1,NULL);
    timerCtl = TIMER_MK_CTL(TIMER_CTL_FUNC_GPIO,
                            TIMER_CTL_INVOUT_NO,
                            TIMER_CTL_DATOUT_0,
                            TIMER_CTL_PWID_ONE,
                            TIMER_CTL_GO_NO,
                            TIMER_CTL_HLD_NO,
                            TIMER_CTL_CP_PULSE,
                            TIMER_CTL_CLKSRC_CPUOVR4,
                            TIMER_CTL_INVINP_NO);
    /* configure timer */
    TIMER_ConfigB(hTimer1, timerCtl, 0xFFFFFFFF, 0);
    /* start timer */
    TIMER_Start(hTimer1);
    /* create encoder and decoder instances */
    ep = (H263EncParam *)((int)encHandle0+sizeof(IALG_Obj));
    while (1)
    {
        /* execute encoder and decoder */
#if ( (ESTATS) || (ESTATS_) )
        /* update encoder statistics */
        encStatsUpdate(ep);
#endif
    }
}
```

Appendix C Real–time Transport Protocol (RTP)

Figure C–1. RTP Parameters – `H263RTPParam` (Little Endian)

The current implementation of the encoder does not support RTP, although the `H263RTPParam` structure (defined in `h263.h`) exists for future expansion.

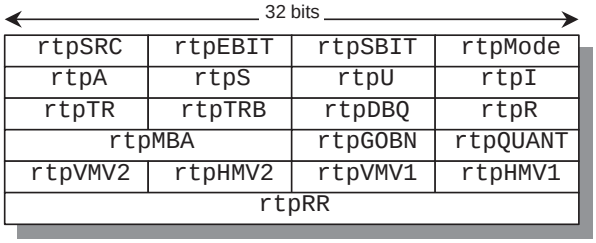


Table C–1. RTP Parameters – `H263RTPParam`

Name	Description
`rtpMode`	RTP mode (A, B, or C)
`rtpSBIT`	Starting bit position
`rtpEBIT`	Ending bit position
`rtpSRC`	Source format
`rtpI`	Picture coding type
`rtpU`	Unrestricted motion vector option
`rtpS`	Syntax–based arithmetic coding opiton
`rtpA`	Advanced prediction option
`rtpR`	Reserved (must be set to zero)
`rtpDBQ`	Differential quantisation parameter
`rtpTRB`	Temporal reference for the B–frame
`rtpTR`	Temporal reference for the P–frame
`rtpQUANT`	Quantisation value for the first MB in the packet
`rtpGOBN`	GOB number in effect at the start of the packet
`rtpMBA`	MB address within the GOB of the first MB in the packet
`rtpHMV1` `rtpVMV1`	Horizontal and vertical motion vector predictors for the first MB in the packet
`rtpHMV2` `rtpVMV2`	Horizontal and vertical motion vector predictors for block number 3 in the first MB in the packet
`rtpRR`	Reserved (must be set to zero)

Please refer to the RTP specification for more information.

Appendix D Testing H.263 Encoder

The encoder has been tested on both TMS320C6201 EVM and TMS320C6211/TMS320C6711 DSK. The following sections describe how the testing was carried out on both platforms.

Figure D–1 shows the setup used to test the encoder on TMS320C6201 EVM.

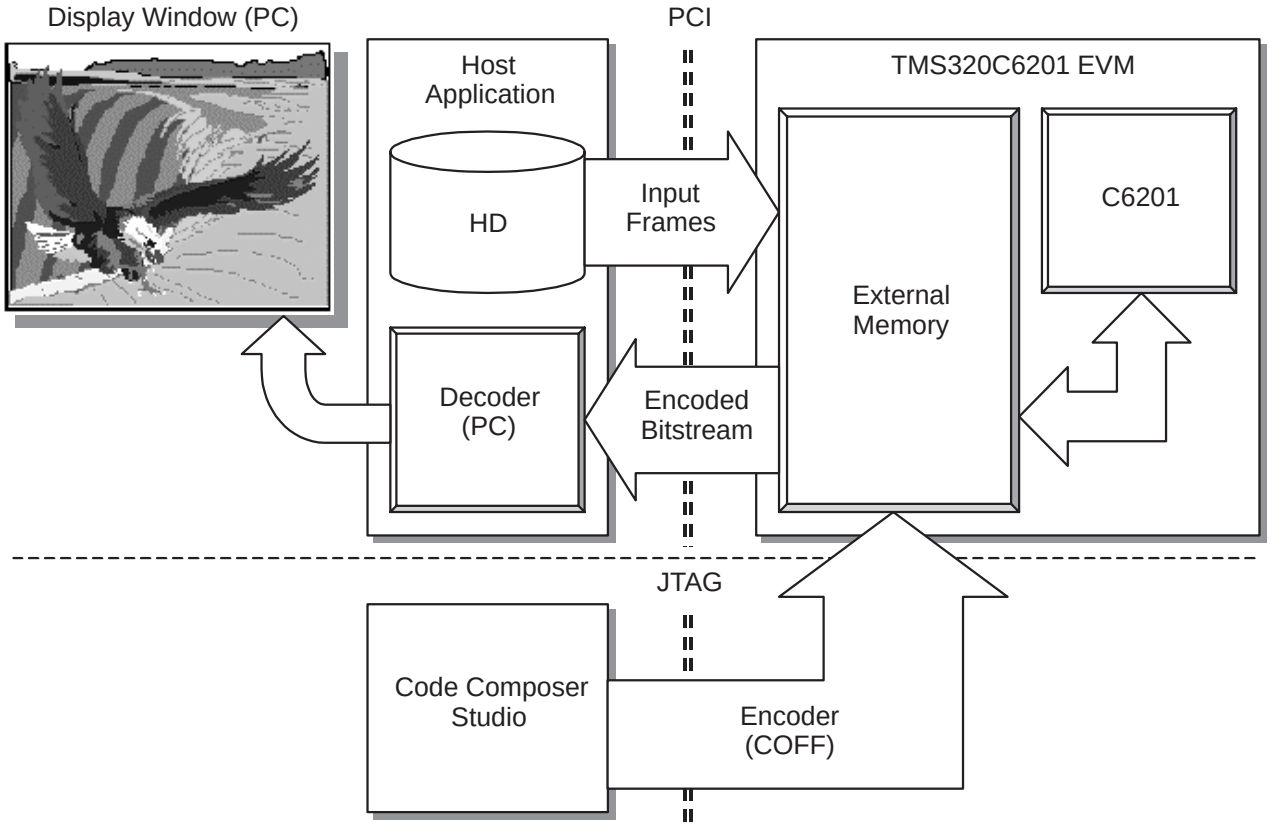


Figure D–1. Test Setup on TMS320C6201 EVM

Figure D–2 shows the setup used to test the encoder on TMS320C6211 DSK.

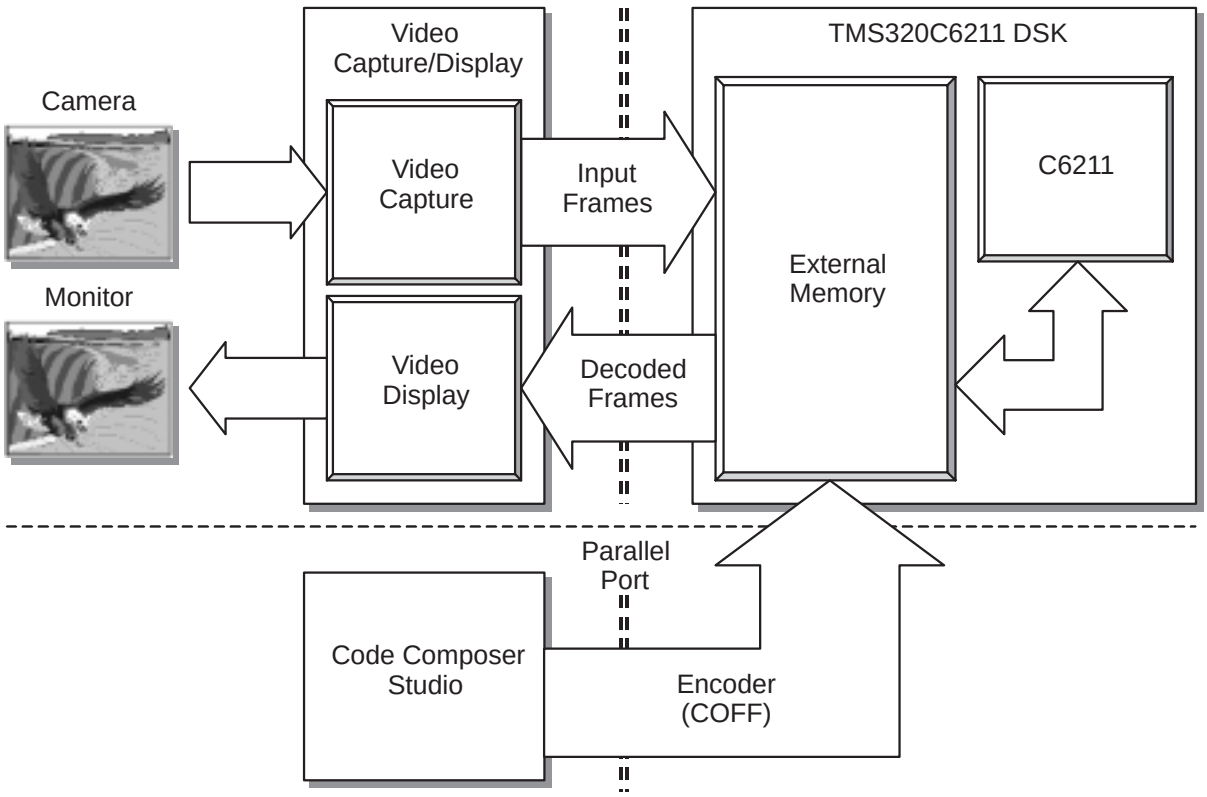


Figure D–2. Test Setup on TMS320C6211 DSK

Appendix E Encoding Custom Resolutions

In the default case, the encoder processes the input frames in the format specified at creation time. One may initialise a particular encoder to process CIF, or one may even choose to tell the encoder to switch to one's own format. This can be achieved simply by modifying the `H263ENC_TI_control` function (defined in h263enc_ti.c).

For example, if one decides to encode 320x240 frames, one simply needs to add the following lines of code to the appropriate portion of the function. The encoder will take care of the rest. Please note, however, that the decoder must be aware of this change, and modified accordingly to ensure proper decoding of the bitstream.

```
case H263_SRCFMT_320x240: nGOB   = 15;
                          nMB    = 20;
                          width  = 320;
                          height = 240;
                          break;
```

Please note that the width and height of a frame must be multiples of 16 pixels (the size of one MB) in order for the encoder to work properly, and it is the user's responsibility to ensure that they are.

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgment, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its semiconductor products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Customers are responsible for their applications using TI components.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty or endorsement thereof.