Application Report
SPRA703 – December 2000

H.263 Decoder: 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 decoder does not, at the time of print, meet all of the baseline requirements to be eXpressDSP™ 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

Figures

Tables

1 Introduction

The TMS320C6000 implementation of H.263 decoder 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 partially compliant, at the time of print, with the *eXpressDSP Algorithm Standard* Refer to appropriate documentation for more information.
- The code has been tested extensively on TMS320C6201 EVM and TMS320C6211/C6711 DSK.
- Every VLD function is equipped with error detection capabilities. Once an error in a bitstream has been detected, the decoder will exit with an appropriate error code to indicate where the error has been detected. Throughout the development stage, this feature has been used to verify all the changes that have been made to the codes, thereby testing the error detection capabilities as well.
- The decoder has very little device specific code; executing the decoder on any of the TMS320C6000 family of 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 to either parse an RTP bitstream, or to process any information extracted from the bitstream.

- The decoder is structured to provide as much flexibility as possible, so that it can run under different system configurations. 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. Refer to the sales representatives for more information.

2 Decoder Implementation

2.1 Directory Structure

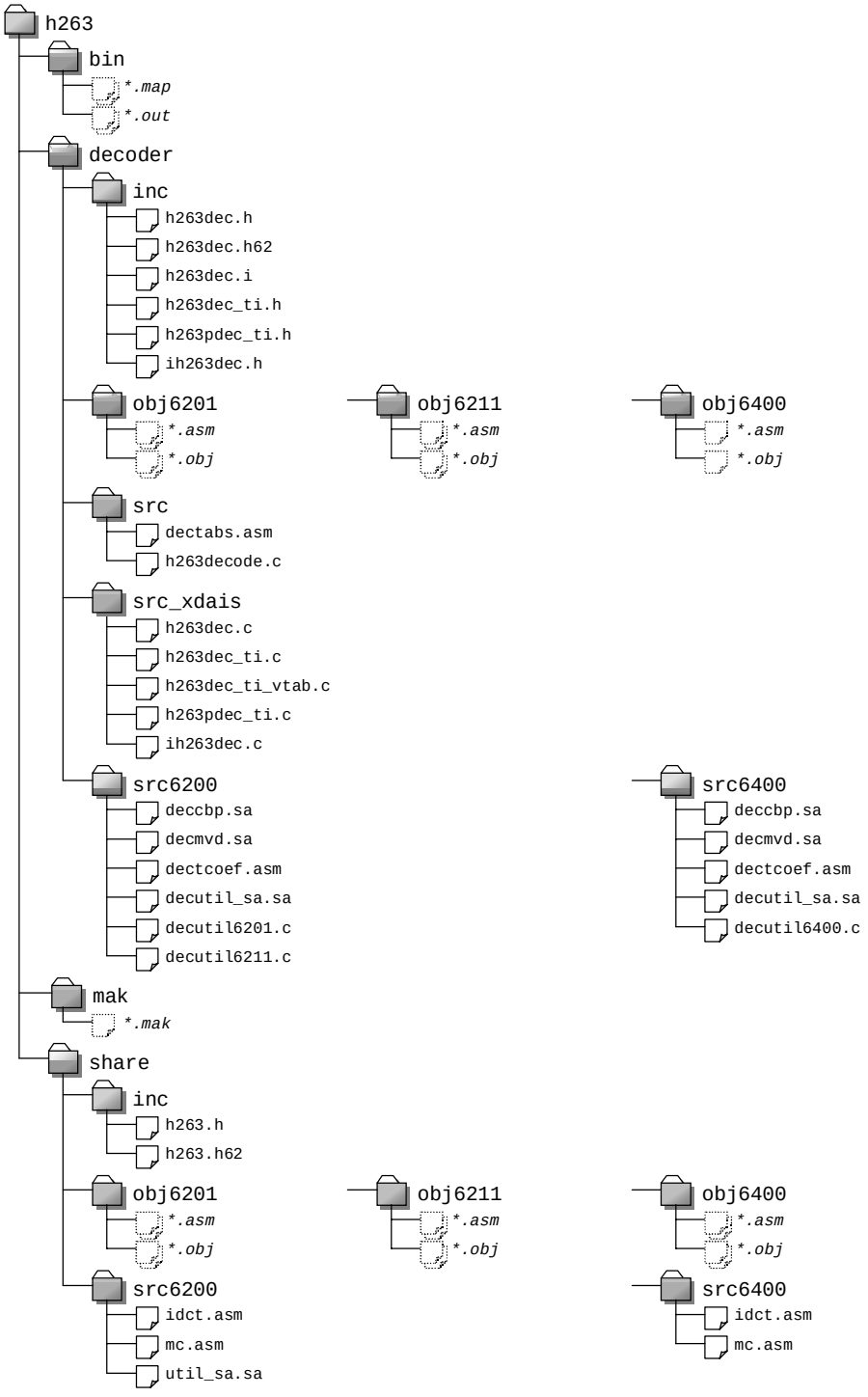


Figure 1. Decoder Directory Structure

h263: H.263 root directory

bin: Decoder COFF and map files

decoder: Decoder root directory

inc: Include files used by the 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: Decoder source files (device independent)

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

src6200: Source files specifically designed for TMS320C62x

src6400: Source files specifically designed for TMS320C64x

mak: Makefiles.

share: Root directory for shared files

inc: Include files

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 TMS320C62x

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.

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

2.2 H.263 Decoder Objects

The current implementation of the H.263 decoder defines a parent object `H263PDEC_TI_Obj` (defined in `h263pdec_ti.h`) that is used to hold a single copy of the decoder tables (look-up tables, VLD tables, etc.), since they are common to all H.263 decoder 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.

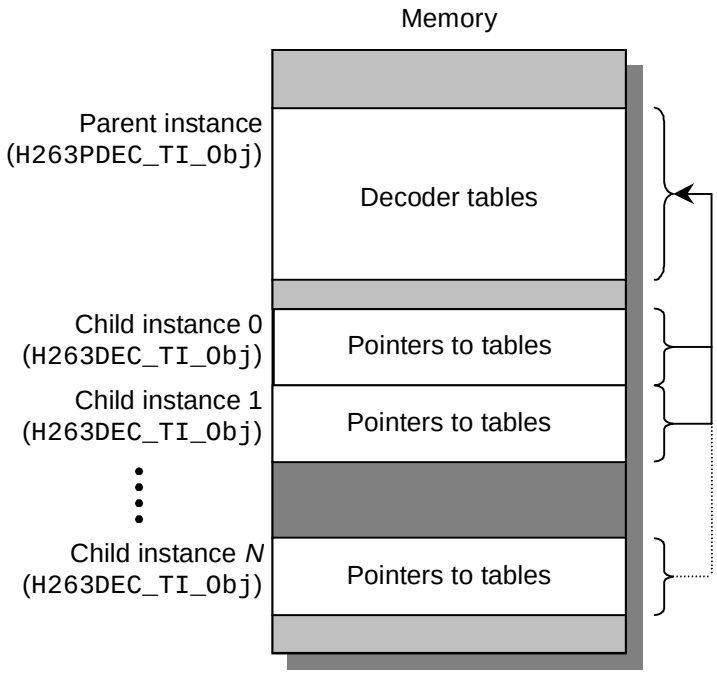


Figure 2. Using Parent and Child Instances

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 decoder is designed so as to allow the user to select which configuration is more suitable. This arrangement is illustrated in the figure below.

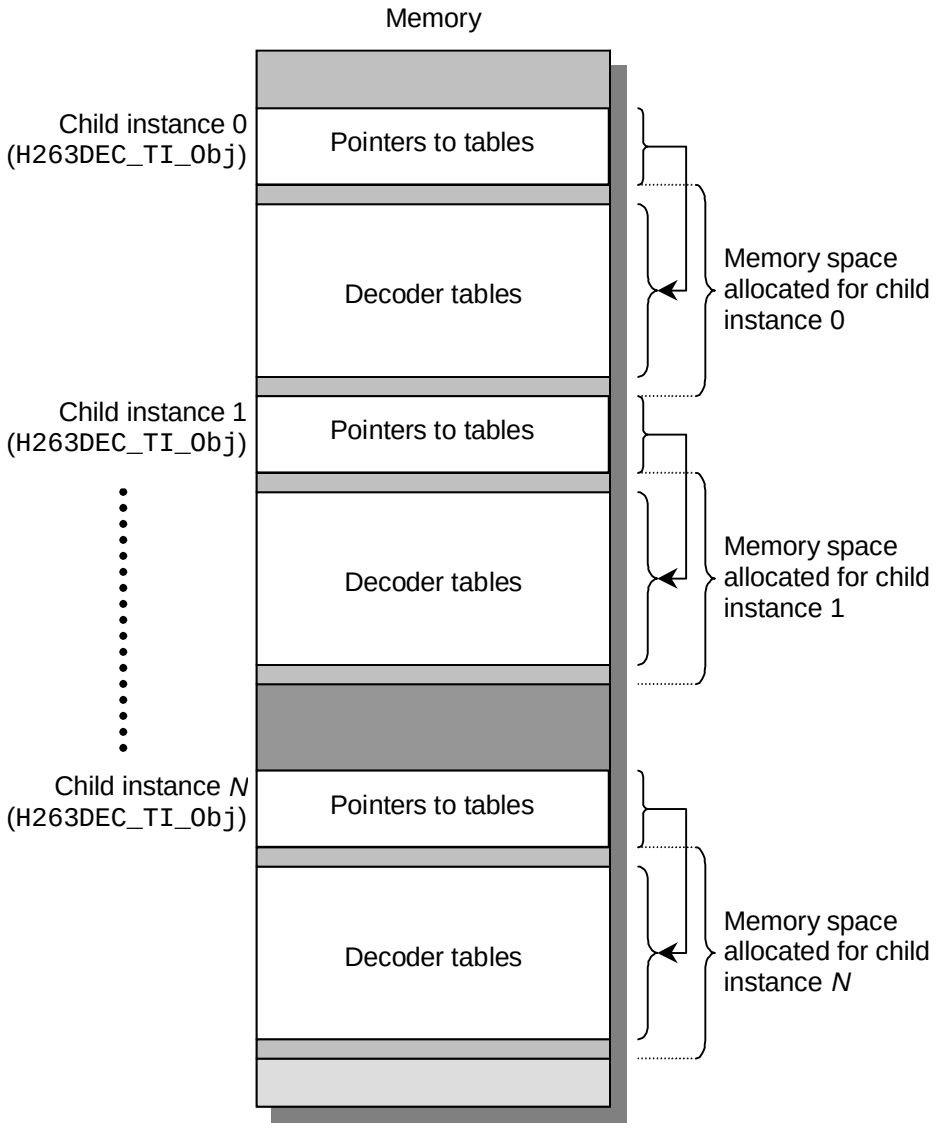


Figure 3. Using Only Child Instances

See Section 3, *Building H.263 Decoder* for more information on how to use this configuration.

2.3 APIs and Example Code

Shown below is how the IALG functions structure `IH263DEC_Fxns` (defined in `ih263dec.h`) looks like.

```
typedef struct IH263DEC_Fxns
{
    IALG_Fxns ialg;

    void (*control)(IH263DEC_Handle  handle,
                    IH263DEC_Cmd     cmd,
                    IH263DEC_Status *status);

    int  (*decode) (IH263DEC_Handle  handle,
                    uint            *in,
                    uchar           *out[3]);

} IH263DEC_Fxns;
```

`ialg`: This is the default IALG functions. Refer to appropriate TMS320 DSP Algorithm Standard documents for more information.

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

`decode`: Executes the H.263 decoder.

Shown below is an example code, in which one parent instance and one child instance are created. Note that since the decoder extracts whatever information it needs from the bitstream, parameters are not required at creation time.

Refer to *eXpressDSP Algorithm Standard Rules and Guidelines* for more information on specific function APIs.

```
void main()
{
    H263PDEC_TI_Obj *decParent; /* decoder parent handle    */
    IH263DEC_Handle  decHandle; /* decoder child handle     */
    IH263DEC_Status  decStatus; /* decoder status           */
    unsigned int    *in;        /* input bitstream          */
    unsigned char   *out[3];    /* output frame (Y, Cb, Cr) */

    /* Create parent instance of H.263 decoder */
    decParent = (H263PDEC_TI_Obj *)ALG_create((IALG_Fxns *)&H263PDEC_TI_IALG,
                                              NULL,
                                              (IALG_Params *)NULL);

    /* Create child instance of H.263 decoder */
    decHandle = (IH263DEC_Handle)ALG_create((IALG_Fxns *)&H263DEC_TI_IH263DEC,
                                            decParent,
                                            (IALG_Params *)NULL);

    while (1)
    {
        /* get pointer to input bitstream -> in      */
        /* get pointer to output frame buffer -> out */
        /* execute H.263 decoder                     */
        H263DEC_TI_IH263DEC.decode((IH263DEC_Handle)decHandle,
                                   in,
                                   out);

        /* Get updated status of the decoder                        */
        H263DEC_TI_IH263DEC.control((IH263DEC_Handle)decHandle,
                                    IH263DEC_GET_STATUS,
                                    &decStatus);
    }
}
```

2.4 H.263 Decoder Structures

2.4.1 Parent Object – `H263PDEC_TI_Obj`

The parent object `H263PDEC_TI_Obj` (defined in `h263pdec_ti.h`) is used to store tables (defined in `dectabs.asm`) that are common to all decoder child instances. Figure 4 shows how they are structured.

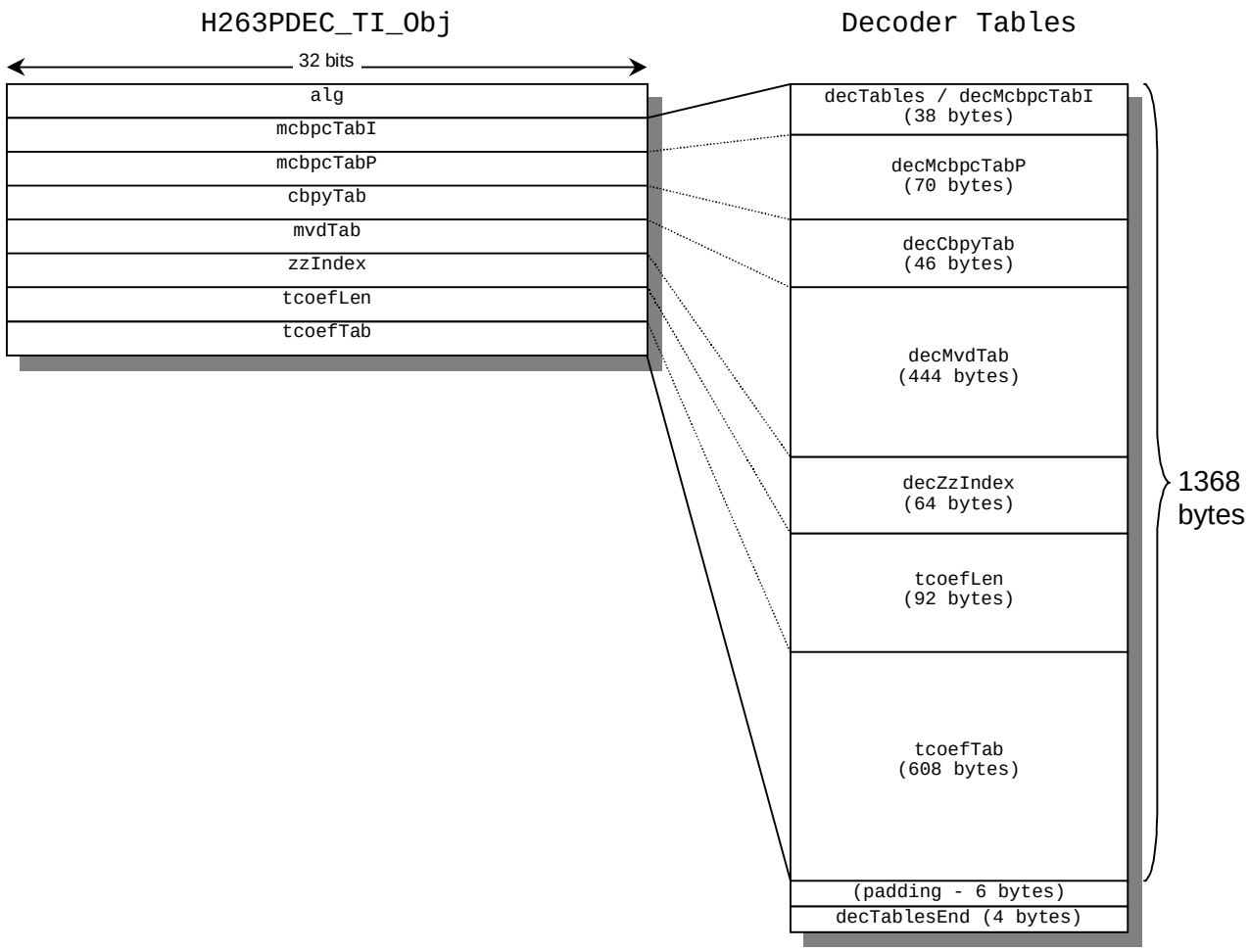


Figure 4. Parent Object – `H263PDEC_TI_Obj`

Table 1. Parent Object – `H263PDEC_TI_Obj`

Table name	Description
`alg`	Default IALG object
`mcbpcTabI`	VLD table for MCBPC (MB type and CBP for chroma for INTRA MB)
`mcbpcTabP`	VLD table for MCBPC (MB type and CBP for chroma for INTER MB)
`cbpyTab`	VLD table for CBPY (CBP for luma)
`mvdTab`	VLD table for MVD (Motion Vector Difference)
`zzIndex`	Zigzag index table
`tcoefLen`	TCOEF length table
`tcoefTab`	VLD table for TCOEF

2.4.2 Child Object – H263DEC_TI_Obj

The H263DEC_TI_Obj structure (defined in h263dec_ti.c) is used to store information specific to each instance of the decoder. Its organisation and descriptions are shown below.

The shaded field (rtpParam) is optional.

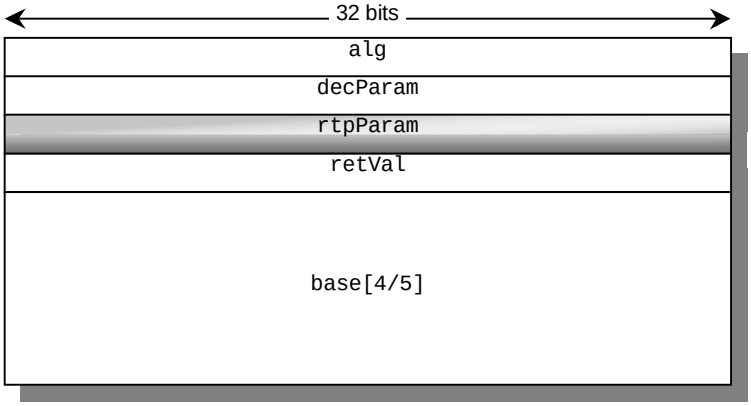


Figure 5. Child Object – H263DEC_TI_Obj

Table 2. Decoder object – H263DEC_TI_Obj

Name	Description
alg	Default IALG object
decParam	Pointer to decoder parameter structure H263DecParam (see below)
rtpParam	RTP parameter structure (H263RTPParam). This is valid only when RTP flag is used at build time. See Appendix D, *Real-time Transport Protocol (RTP)* for more information.
retVal	Return value from the decoder
base[4/5]	Base addresses for all memory spaces allocated for a particular child instance; 4 when used with a parent instance, and 5 without.

2.4.3 Decoder Parameters – H263DecParam

The H263DecParam structure (defined in h263decode.h) is the main structure that the decoder 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. Refer to the H.263 specification for more detail.

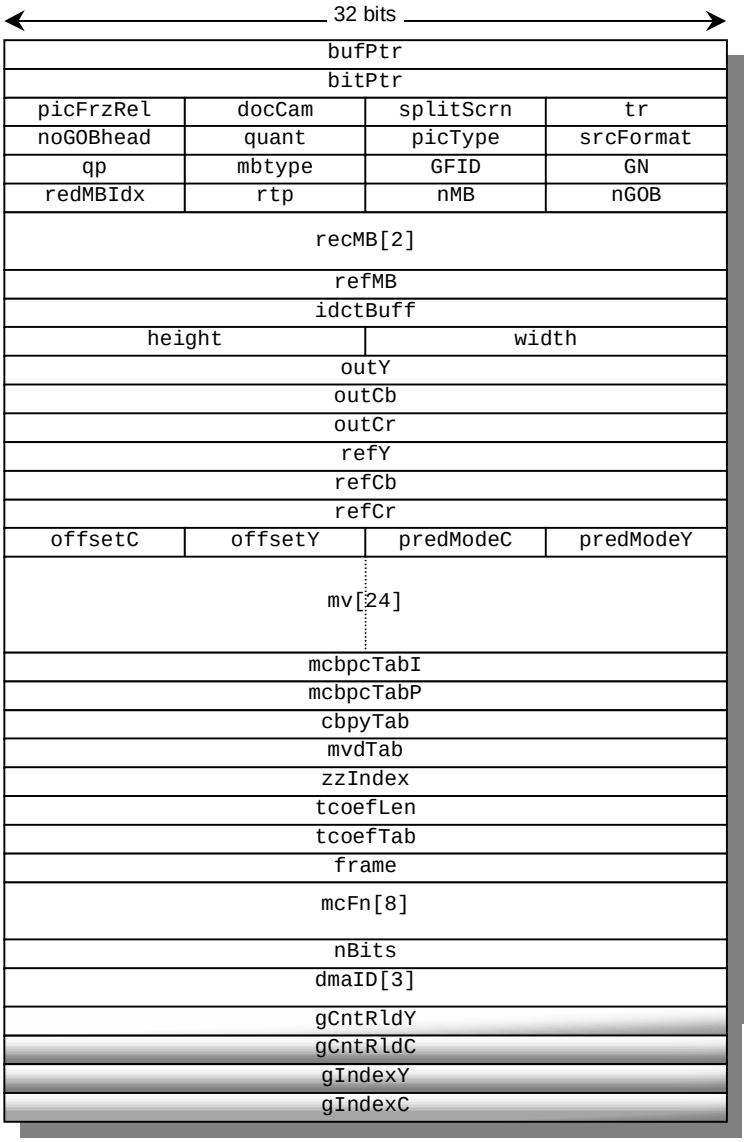


Figure 6. Decoder Parameters – `H263DecParam` (little endian)

Table 3. Decoder Parameters – `H263DecParam`

Name	Description
`bufPtr`	Points to the current 32-bit word of the input H.263 bitstream
`bitPtr`	Bit position, or number of remaining bits in the current 32-bit word; MSB=32; LSB=1.
`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

Name	Description
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
rtp	'1' for RTP mode; '0' for non-RTP
recMBIdx	Index into recMB array. It is used to point to the current reconstructed MB buffer. See Section 2.4.8, *Reconstructed MB Buffer (recMB)* for more information.
recMB[2]	Pointers to reconstructed MB buffers. See Section 2.4.8, *Reconstructed MB Buffer (recMB)* for more information.
refMB	Points to reference MB buffer
idctBuff	Points to IDCT buffer
width	Width of image
height	Height of image
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)
predModeY predModeC	Half-pel modes for luma and chroma, respectively
offsetY offsetC	Offsets inside reference MB for luma and chroma, respectively. See Section 2.4.6, *Reference Offsets (offsetY and offsetC)* for more information.
mv[24]	Motion vectors. See Section 2.4.7, *Motion Vectors (mv[24])* for more information.
mcbpcTabI mcbpcTabP cbpyTab mvdTab zzIndex tcoefLen tcoefTab	Pointers to VLD tables for MCBPC (I-frame), MCBPC (P-frame), CBPY, MVD, and tables used to decode TCOEF.
frame	Number of frames decoded
mcFn[8]	Array of function pointers for MC kernels. See Section 2.4.9, *Motion Compensation Kernels (mcFn_t mcFn[8])* for more information.
nBits	Number of bits used by decoded frame
dmaID[3]	DMA/EDMA IDs. See Section 2.4.10, *DMA/EDMA ID's (dmaID[3])* 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.

Name	Description
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.

2.4.4 Decoder Status – *IH263DEC_Status*

Shown below is how the H.263 decoder status structure `IH263DEC_Status` (defined in `ih263dec.h`) is organised.

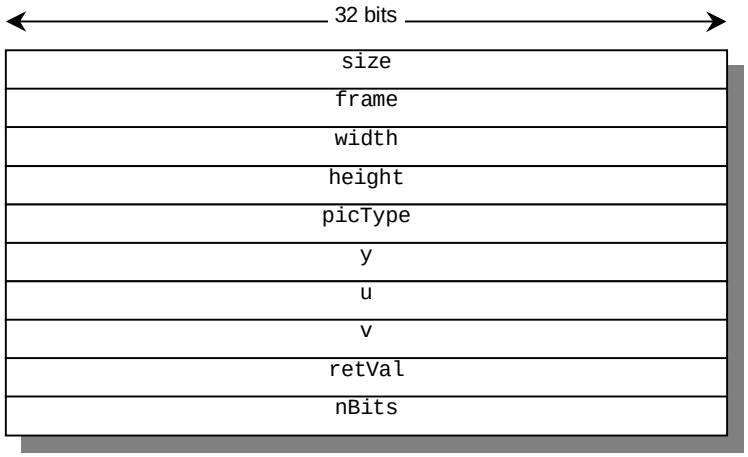


Figure 7. Decoder Status – `IH263DEC_Status`

Table 4. Decoder status – `IH263DEC_Status`

Name	Description
`size`	Size of the structure (required)
`frame`	Number of frames decoded
`width`	Width of decoded frame
`height`	Height of decoded frame
`picType`	Picture type of decoded frame
`y`	Address of decoded frame (luma)
`u`	Address of decoded frame (Cb)
`v`	Address of decoded frame (Cr)
`retVal`	Error value returned by the decoder
`nBits`	Number of bits used by the last decoded frame

2.4.5 Decoder Return Values

Table 5 below describes all the possible return values (defined in `h263decode.h`) used by the decoder.

Table 5. Decoder Return Values

retVal		Description
Label	Value	
H263D_ERR_NOERROR	0	No errors detected – normal completion
H263D_ERR_PSC	1	Invalid PSC
H263D_ERR_ZEROBIT	2	Bit 2 of PTYPE was not "0"
H263D_ERR_RSVDBITS	3	Reserved bits were not "0000"
H263D_ERR_SRCFORMAT	4	Invalid source format
H263D_ERR_PQUANT	5	Invalid PQUANT
H263D_ERR_CBP	6	Invalid MCBPC or CBPY
H263D_ERR_MVD	7	Invalid MVD
H263D_ERR_TCOEF	8	Invalid TCOEF

2.4.6 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.

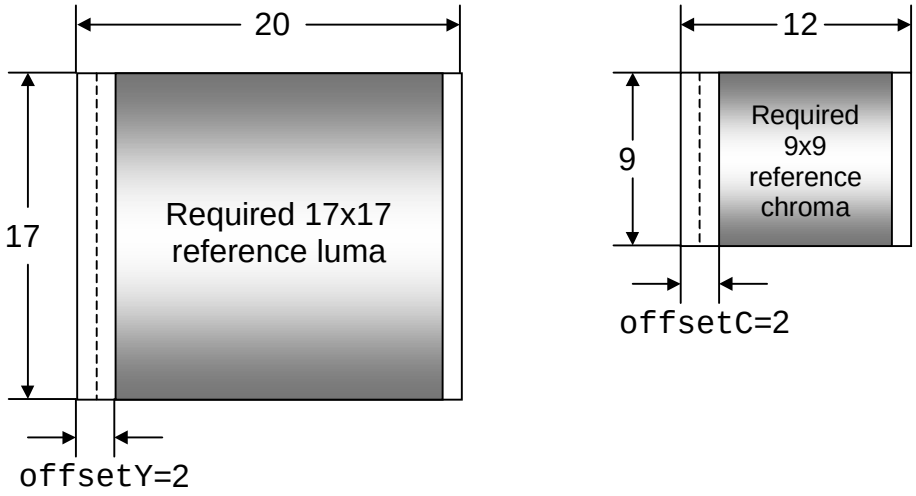


Figure 8. Example of Using `offsetY` and `offsetC`

2.4.7 Motion Vectors (mv[24])

Motion vector for the current MB depends on three previously decoded vectors: one to the left (mv1), above (mv2), and above right (mv3). As shown below, some or all of these are set to zero, depending on which MB is currently being decoded.

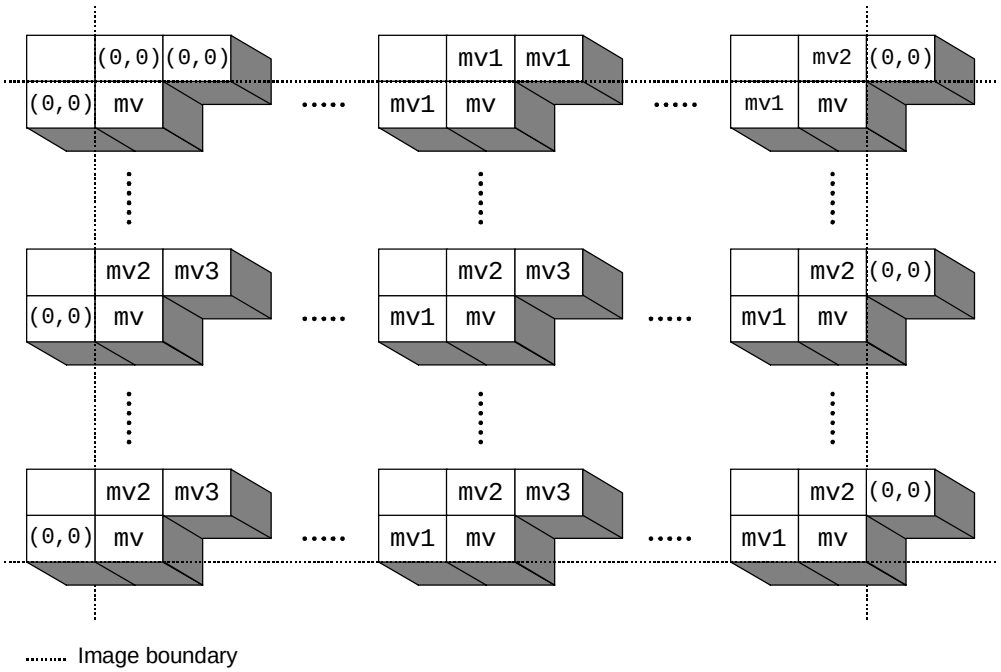


Figure 9. Motion Vector Candidates

To avoid having nested "if" statements, two extra bytes are allocated in the motion vector array. As soon as the decoder determines the source format, it sets the left-most and right-most array elements to zero for the edge conditions. The code now becomes a simple "if first GOB;else" Figure below illustrates examples for CIF and QCIF.

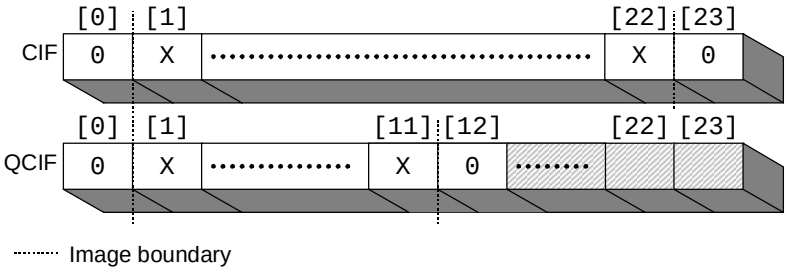


Figure 10. Example of how decoder sets mv[24]

Conceptually, mv[0] is outside the image boundary to the left; similarly mv[23] and mv[12] are outside the image boundary to the right for CIF and QCIF, respectively. "X" denotes the actual motion vectors that are set and read by the decoder; the shaded area for QCIF is ignored. Shown below is a section of the code (h263decode.c) that is responsible for selecting the three motion vector candidates.

```
mv1 = decParam->mv[j]; /* MV for previous MB */

if ( (i==0) || (!decParam->noGOBhead) ) /* if 1st GOB or no header, mv2=mv3=mv1 */
{
    mv2 = mv1;
    mv3 = mv1;
}
/* else mv2 is MV for corresponding MB in previous GOB & mv3 is the one after it */
else
{
    mv2 = decParam->mv[j+1];
    mv3 = decParam->mv[j+2];
}
    :
/* decode MVD */
    :
/* store new MV */
decParam->mv[j+1] = mv;
```

In the code above, variable `j` identifies the current MB, and ranges from 0 to 21 for CIF, or 0 to 10 for QCIF. However, since there is an extra byte at the beginning (and end) of the array, `mv[j]` actually contains motion vector for the previous MB; `mv[j+1]` and `mv[j+2]` contain motion vectors for the above and above right, respectively.

The figure below shows an example of how the array `mv[24]` is used to store motion vectors for the current GOB.

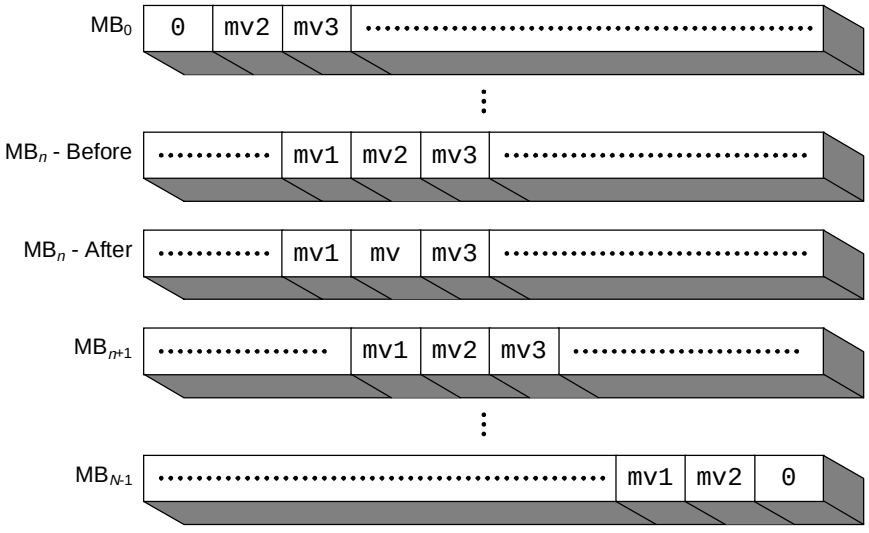


Figure 11. Example showing how `mv[24]` is used

Before decoding the MVD, three motion vector candidates `mv1`, `mv2`, and `mv3` are selected. `mv1` is the motion vector for the previous MB; `mv2` is the motion vector for the MB just above the current MB; `mv3` is the motion vector for the MB that is next to the one for `mv2`.

Since `mv2` is no longer required for processing remaining MBs in the current GOB, it is overwritten by the latest vector, which may be assigned to `mv1` for the next MB.

2.4.8 Reconstructed MB Buffer (`recMB`)

The CPU may be required to wait for the previously reconstructed MB to be written to the output frame buffer, before it can proceed with the reconstruction of the current MB. To minimise the time that the CPU may have to wait, two buffers are allocated for reconstructed MBs (`recMB[0]` and `recMB[1]`) to facilitate double-buffering. If the decoder has just issued a data transfer request out of `recMB[0]` for the first MB, then the CPU does not have to wait for that request to complete until it is ready to reconstruct the third MB. This is provided that the decoder has already reconstructed the second MB in `recMB[1]` and issued a request to write out its contents out to the output frame buffer.

At any given time, `recMBIdx` points to either one to indicate the current buffer that the CPU can use for reconstruction. The switch occurs at the end of decoding and reconstructing every MB.

2.4.9 Motion Compensation Kernels (`mcFn_t mcFn[8]`)

In motion compensation, there are four half-pel modes, with and without IDCT coefficients, requiring eight 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 eight 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 eight 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 initialisation stage (`H263DEC_TI_initObj` defined in `h263dec_ti.c`), the array is setup as shown below. Note that the structure `dp` is of the type `H263DecParam`.

```
dp->mcFn[0] = mcA;
dp->mcFn[1] = mcB;
dp->mcFn[2] = mcC;
dp->mcFn[3] = mcD;
dp->mcFn[4] = mcAi;
dp->mcFn[5] = mcBi;
dp->mcFn[6] = mcCi;
dp->mcFn[7] = mcDi;
```

See Section 2.7.3, *Motion Compensation (h263DecMC)* for more information on how the motion compensation is applied to a MB.

2.4.10 DMA/EDMA ID's (dmaID[3])

In the default configuration, all data transfers are initiated via the DAT module in the Chip Support Library (CSL), except when DIRDMA flag or CSLDMA flag is used to build the decoder. See Section 3, *Building H.263 Decoder* and Appendix B, *Data Transfer Methods* for more information.

For every MB, the decoder needs to keep track of three sets of data transfers into and out of `refMB`, `recMB[0]`, and `recMB[1]` (reference MB and reconstructed MB buffers); furthermore, each transfer requires three separate requests (one luma and two chroma blocks). However, since all transfer requests are serviced sequentially, the decoder only has to wait for the last of these three requests to complete before proceeding. In short, the decoder only keeps track of last transfer request for each buffer.

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 `recMB[0]`.
- `dmaID[1]` is the request ID associated with `recMB[1]`.
- `dmaID[2]` is the request ID associated with `refMB`.

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

2.5 Memory Requirements

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

The code sizes shown are based on compiling with "`-mtx –mh256 –o3`" with the appropriate "Target Device".

Table 6. H.263 Decoder Code Sizes (Bytes)

	TMS320C6201	TMS320C6211	TMS320C6400
Core decoder code	10,176	9,952	10,944
Shared code	8,384	8,384	2,188
Total	18,560	18,336	13,132

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

The code size for TMS320C6201 is slightly larger than that of TMS320C6211, due to the additional control code required for issuing DMA requests. See Section 2.8.7, *Copying Non-coded MB (cpMB)* and Appendix B, *Data Transfer Methods* for more information.

Note that the decoder 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.

Table 7. Internal Memory Requirements (TMS320C62x)

	Size (Bytes)		Alignment
	Parent	Child	
Reconstructed MB buffer (`recMB[0]`)	0	384	16 bytes (0x10)
Reconstructed MB buffer (`recMB[1]`)	0	384	16 bytes (0x10)
Reference MB Buffer (`refMB`)	0	556	16 bytes (0x10)
IDCT buffer	0	896	16 bytes (0x10)
Decoder parent object (`H263PDEC_TI_Obj`)	1,368	0	16 bytes (0x10)
Decoder child object (`H263DEC_TI_Obj`)	0	224	16 bytes (0x10)
Stack	0	336	N/A
Total	1,368	2,780	-

Table 8. Internal Memory Requirements (TMS320C64x)

	Size (Bytes)		Alignment
	Parent	Child	
Reconstructed MB buffer (`recMB[0]`)	0	384	16 bytes (0x10)
Reconstructed MB buffer (`recMB[1]`)	0	384	16 bytes (0x10)
Reference MB Buffer (`refMB`)	0	556	16 bytes (0x10)
IDCT buffer	0	768	16 bytes (0x10)
Decoder parent object (`H263PDEC_TI_Obj`)	1,368	0	16 bytes (0x10)
Decoder child object (`H263DEC_TI_Obj`)	0	224	16 bytes (0x10)
Stack	0	336	N/A
Total	1,368	2,652	-

Table 9. External Memory Requirements

	Size (Bytes)	Alignment
Decoder tables (	1,368	16 bytes (0x10)
Frame buffer 0	152,064	16 bytes (0x10)
Frame buffer 1	152,064	16 bytes (0x10)
Total	305,496	-

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 decoder on TMS320C6211/TMS320C6711/TMS320C64x.

The IDCT code for TMS320C62x 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 decoder 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 decoder 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.

2.5.1 Memory Maps

The figure below shows the memory map used for TMS320C6201 EVM and TMS320C6211 DSK. 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; Refer to the documentation provided with the board for more information.

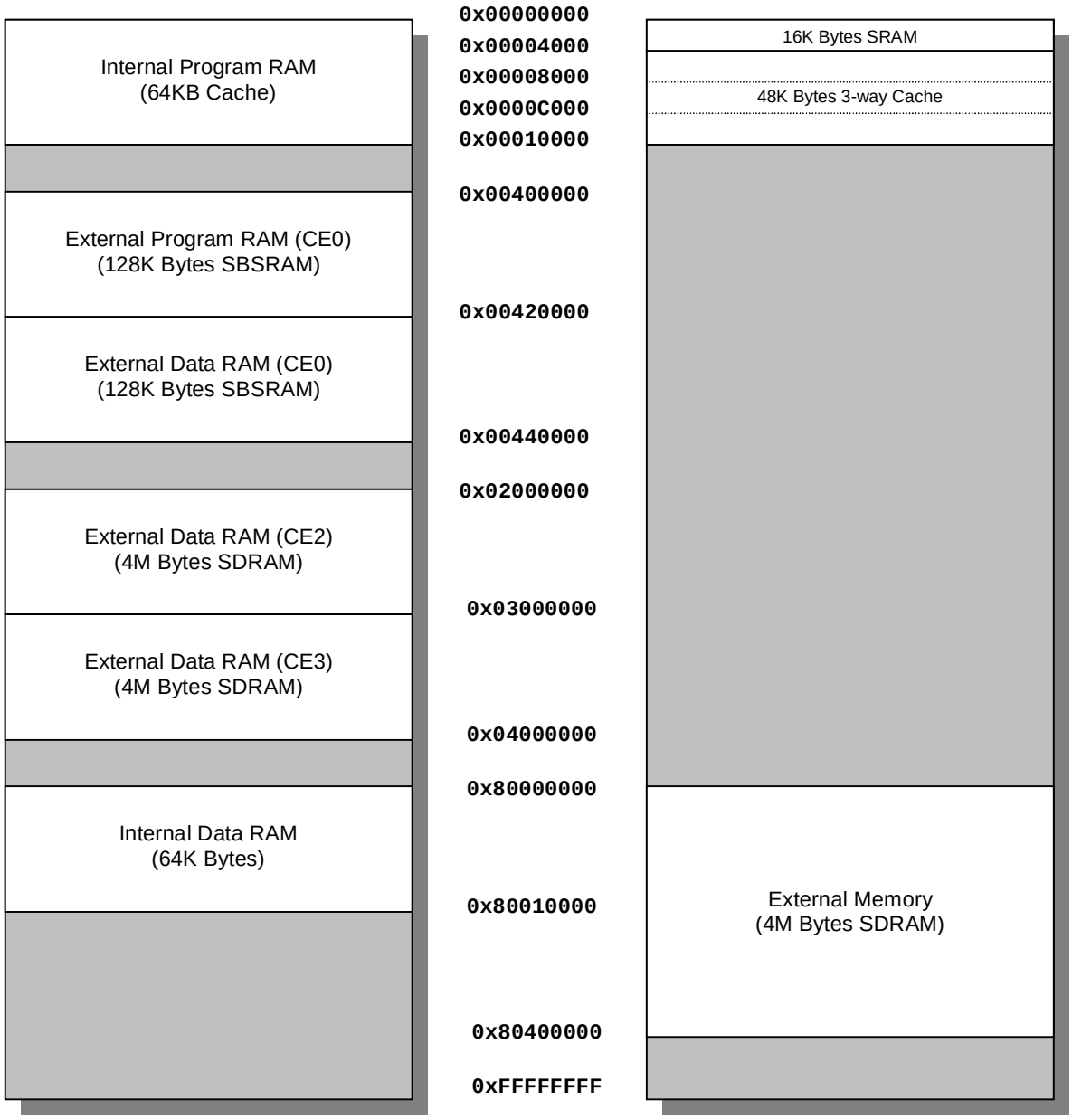


Figure 12. TMS320C6201 EVM & TMS320C6211 DSK Memory Maps

2.6 H.263 Decoder Functions

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

Table 10. H.263 Decoder Functions (Device Independent)

Function	Description	Returns	Source File
h263DecGOB	Decodes one GOB	int	h263decode.c
h263DecMB	Decodes one MB	int	h263decode.c
h263DecMC	Applies half-pel motion compensation to a MB	void	h263decode.c
h263Decode	Decodes one frame	int	h263decode.c

Table 11. H.263 Decoder Functions (TMS320C62x)

Function	Description	Returns	Source File
cpMB	Copies MB from reference to output	void	decutil6201.c decutil6211.c
deccbp	Decodes MCBPC and CBPY	int	deccbp.sa
decmvd	Decodes MVD	ushort	decmvd.sa
dectcoef	Decodes TCOEF	int	dectcoef.asm
getbits	Extracts required number of bits from the bitstream	uint	decutil_sa.sa
rdRefMB	Reads reference MB	void	decutil6201.c decutil6211.c
prezero	Pre-zeros required number of blocks	void	util_sa.sa
wrRecMB	Writes reconstructed MB to output	void	decutil6201.c decutil6211.c

† "uint" and "ushort" are "unsigned int" and "unsigned short", respectively.

Table 12. H.263 Shared Functions (TMS320C62x)

Function	Description	Returns	Source File
idctI	Applies IDCT for INTRA MB	void	idct.asm
idctP	Applies IDCT for INTER MB	void	idct.asm
mcA mcAi mcB mcBi mcC mcCi mcD mcDi	Applies motion compensation mode A, B, C, and D, with and without IDCT output	void	mc.asm
packmb	Packs output from IDCT	void	util_sa.sa

Table 13. H.263 Decoder Functions (TMS320C64x)

Function	Description	Returns	Source File
cpMB	Copies MB from reference to output	void	decutil6400.c
deccbp	Decodes MCBPC and CBPY	int	deccbp.sa
decmvd	Decodes MVD	ushort	decmvd.sa
dectcoef	Decodes TCOEF	int	dectcoef.asm
getbits	Extracts required number of bits from the bitstream	uint	decutil_sa.sa
rdRefMB	Reads reference MB	void	decutil6400.c
prezero	Pre-zeros required number of blocks	void	h263dec_ti.h
wrRecMB	Writes reconstructed MB to output	void	decutil6400.c

Table 14. H.263 Shared Functions (TMS320C64x)

Function	Description	Returns	Source File
idctIP	Applies IDCT for INTRA & INTER MB	void	idct.asm
mcA mcAi mcB mcBi mcC mcCi mcD mcDi	Applies motion compensation mode A, B, C, and D, with and without IDCT output	void	mc.asm
packmb	Packs output from IDCT	void	util_sa.sa

Note that although functions cpMB, rdRefMB, and wrRecMB are defined in files decutil6201.c, decutil6211.c, and decutil6400.c, appropriate versions for the device will be selected, depending on the device designation flag used at build time.

2.7 Code Flow

The following sections describe the order in which the decoder processes a bitstream.

2.7.1 Main Decoder Function (h263Decode)

The figure below shows the high-level code flow.

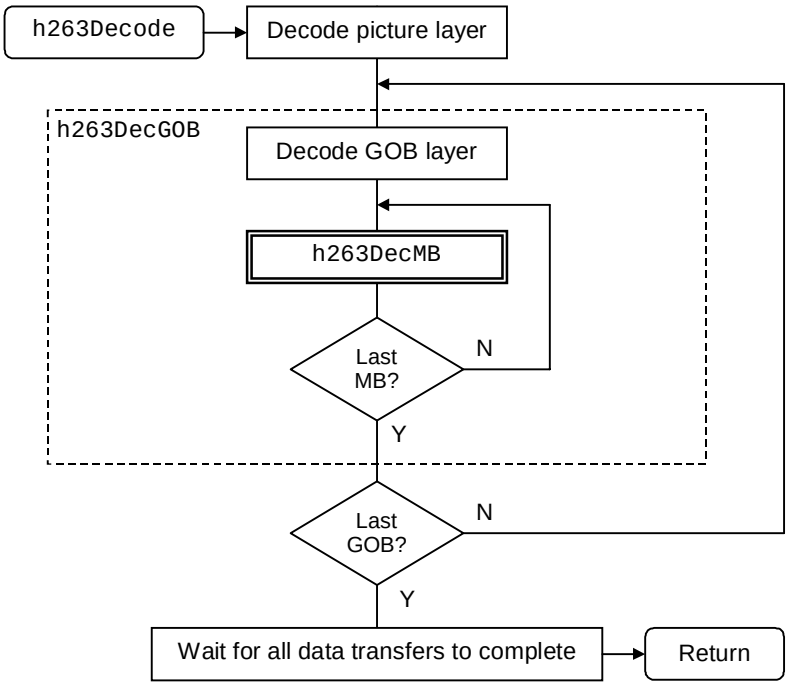


Figure 13. Code Flow – h263Decode

For each frame, the decoder is provided with input H.263 bitstream. The h263Decode function starts parsing the bitstream and extracts information pertaining to the entire frame (the picture layer). Based on the information, it then sets up several variables in the main parameter structure (H263DecParam), including frame buffer pointers, dimension of the image, etc. The function calls h263DecGOB the appropriate number of times.

The h263DecGOB function extracts GOB layer specific information from the bitstream and calls h263DecMB the appropriate number of times.

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

2.7.2 Decoding MB (h263DecMB)

The first thing the function needs to know is if the current MB has been coded or not. If it has not been coded, then the corresponding MB in the reference frame buffer must be copied to the output frame buffer, so that the decoder can properly reconstruct the next frame. This is done by calling the cpMB function.

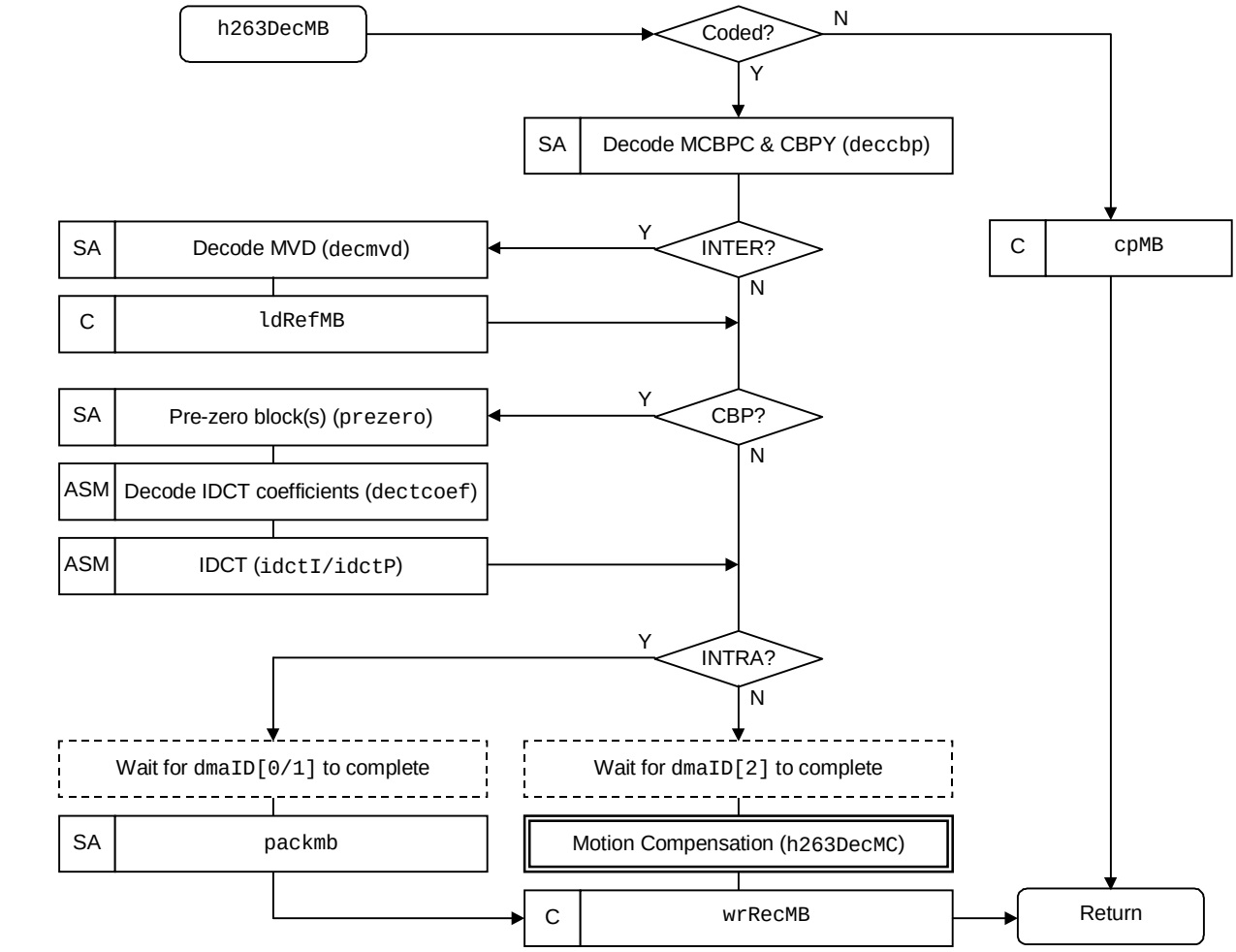


Figure 14. Code Flow – h263DecMB

If, however, the MB has been coded, then further processing is required, starting with MCBPC and CBPY (MB type, CBP for chroma, and CBP for luma). The VLD tables for these are set up so that the return value of the deccbp function is as shown in the figure below.

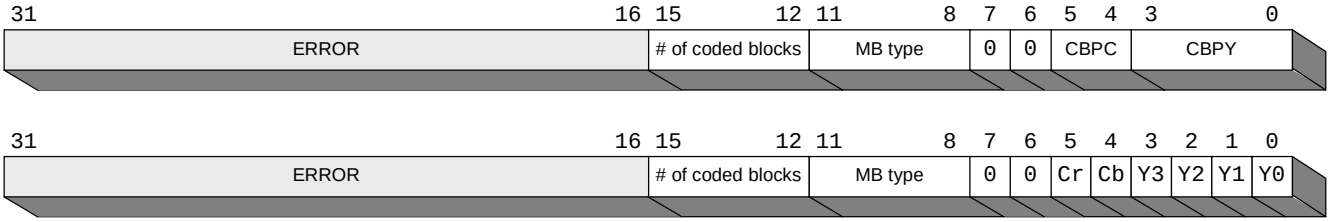


Figure 15. Bit Fields of Value Returned by deccbp

A value of 0xFFFF in the upper 16 bits (bits [31:16]) indicates that the function has detected an error in the bitstream. If the MB type is INTER, then the motion vectors for luma and chroma are decoded by calling the decmvd function. Its return value is shown below.

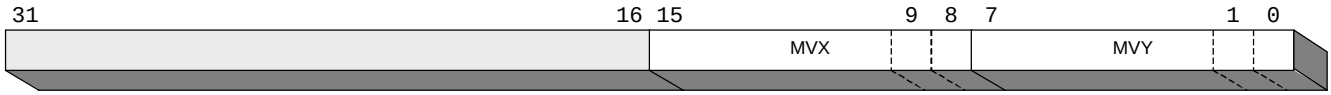


Figure 16. Bit Fields of Value Returned by `decmvd`

Bits [8] and [0] determine the half-pel modes for horizontal and vertical components of luma block; for chroma, the half-pel modes depend on bits [9:8] and [1:0]. A value of `0x80` in bits [15:8] or bits [7:0] indicates that the function has detected an error in the bitstream. The decoded motion vectors are then used to bring the predicted MB from the reference frame buffer by calling the `rdRefMB` function.

If at least one of the six CBP bits is set, then the decoder must decode the IDCT coefficients by calling the `dectcoef` function and apply IDCT. The `dectcoef` function returns the following value.

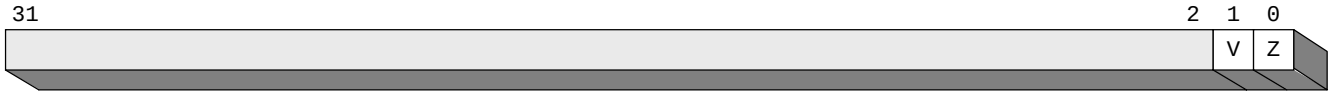


Figure 17. Bit Fields of Value Returned by `dectcoef`

A return value of zero denotes normal completion; if the `Z` bit is set, it means that the zigzag array has over-run; if the `V` bit is set, it means that it has encountered an invalid VLC symbol.

IDCT kernels for TMS320C62x and TMS320C64x 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 TMS320C62x 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 decoder 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 TMS320C62x version, a final saturation stage is required for INTRA MB, before packing the results.

For the INTER MB, the motion compensation function (`h263DecMC`) is called to add IDCT output and the reference MB to reconstruct the current MB.

Note that although the CPU has to wait for the transfer out of recMB to complete (i.e., wait on `dmaID[0]` or `dmaID[1]` to complete) before entering the `packmb` function, it only has to wait for the transfer into `refMB` to complete (i.e., wait for `dmaID[2]` to complete) prior to applying motion compensation. This is because the requests are serviced in the order in which they are issued, so if `dmaID[2]` has completed, it means that any previous requests associated with `recMB` have also completed, and hence it is safe to start reconstructing the current MB.

The final stage of h263DecMB involves simply writing out the reconstructed MB to the output frame buffer by calling the wrRecMB function. Note that because of the double buffering (recMB[0] and recMB[1]), the completion of a particular request is not checked until two (coded) MBs after the current one.

2.7.3 Motion Compensation (h263DecMC)

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

```
void h263DecMC(H263DecParam *dp, int rc, int c)
{
    int      predModeY, predModeC;
    uchar    offsetY,   offsetC;
    mcFn_t   mcfY0, mcfY1, mcfY2, mcfY3, mcfCb, mcfCr;
    uchar   *s0,   *s1,   *s2,   *s3,   *s4,   *s5;
    uchar   *d0,   *d1,   *d2,   *d3,   *d4,   *d5;
    short   *idct0,*idct1,*idct2,*idct3,*idct4,*idct5,*idct;

    idct      = dp->idctBuff;
    predModeY = dp->predModeY;
    predModeC = dp->predModeC;
    offsetY   = dp->offsetY;
    offsetC   = dp->offsetC;

    /* Set source addresses for Y0, Y1, Y2, Y3, Cb & Cr                    */
    s0 = dp->refMB;
    s1 = s0 + 8;
    s2 = s0 + (20* 8);
    s3 = s2 + 8;
    s4 = s0 + (20*17);
    s5 = s4 + (12* 9);

    /* Set destination addresses for Y0, Y1, Y2, Y3, Cb & Cr               */
    d0 = dp->recMB;
    d1 = d0 + 8;
    d2 = d0 + (16* 8);
    d3 = d2 + 8;
    d4 = d0 + (16*16);
    d5 = d4 + ( 8* 8);

    /* for each block MB:-                                                 */
```

```
    /*   - select appropriate MC kernel, depending on pred mode and CBP.    */
    /*   - set address of IDCT coefficients.                                */
    /*   - if CBP bit is set, update idct to point to next set              */
    mcfY0=dp->mcFn[predModeY+((c&1)<<2)]; idct0=idct; idct+=((c&1)<<6); c>>=1;
    mcfY1=dp->mcFn[predModeY+((c&1)<<2)]; idct1=idct; idct+=((c&1)<<6); c>>=1;
    mcfY2=dp->mcFn[predModeY+((c&1)<<2)]; idct2=idct; idct+=((c&1)<<6); c>>=1;
    mcfY3=dp->mcFn[predModeY+((c&1)<<2)]; idct3=idct; idct+=((c&1)<<6); c>>=1;
    mcfCb=dp->mcFn[predModeC+((c&1)<<2)]; idct4=idct; idct+=((c&1)<<6); c>>=1;
    mcfCr=dp->mcFn[predModeC+((c&1)<<2)]; idct5=idct; idct+=((c&1)<<6); c>>=1;

    /* apply MC                                                            */
    mcfY0(s0, d0, offsetY, 20, 16, rc, idct0);
    mcfY1(s1, d1, offsetY, 20, 16, rc, idct1);
    mcfY2(s2, d2, offsetY, 20, 16, rc, idct2);
    mcfY3(s3, d3, offsetY, 20, 16, rc, idct3);
    mcfCb(s4, d4, offsetC, 12,  8, rc, idct4);
    mcfCr(s5, d5, offsetC, 12,  8, rc, idct5);
}
```

For each block, the required motion compensation kernel is selected based on its prediction mode and the CBP bit. The function array `dp->mcFn` is set up so that the first four entries point to kernels that do not require IDCT coefficients, and the last four point to the ones that do.

If CBP for the current block is not set, then the prediction mode determines which one of the first four kernels is required for this block; otherwise, the prediction mode determines which one of the last four kernels is required for this block.

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.

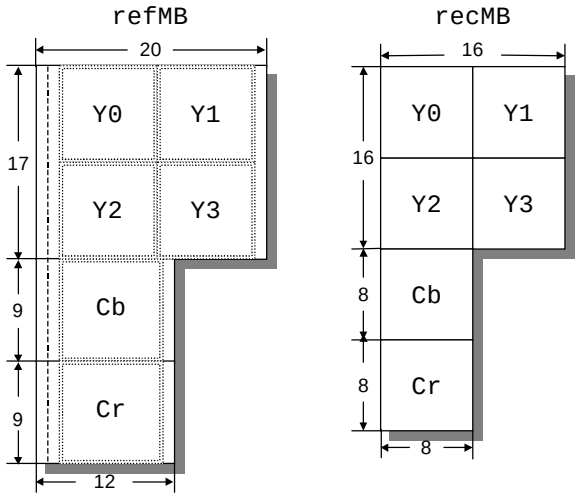


Figure 18. Example of Motion Compensation

The source address of `Y0` is equal to `refMB`; `Y1` is eight pixels to the right; `Y2` is eight lines of 20 pixels after `Y0`; `Y3` is eight pixels after `Y2`; `Cb` is 17 lines of 20 pixels after `Y0`; `Cr` is nine lines of 12 pixels after `Cb`. The offset (equal to two in this example) is passed to and dealt with by each kernel.

Similarly, the destination address of `Y0` is equal to `recMB` (`recMB[0]` or `recMB[1]`); `Y1` is eight pixels to the right; `Y2` is eight lines of 16 pixels after `Y0`; `Y3` is eight pixels after `Y2`; `Cb` is 16 lines of 16 pixels after `Y0`; `Cr` is eight lines of eight pixels after `Cb`.

2.8 Data Flow

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

2.8.1 Frame Buffer

The following figure shows how a frame buffer looks for CIF and QCIF, respectively. The pixels are stored contiguously, so for QCIF, they occupy the first 38,016 bytes of the buffer.

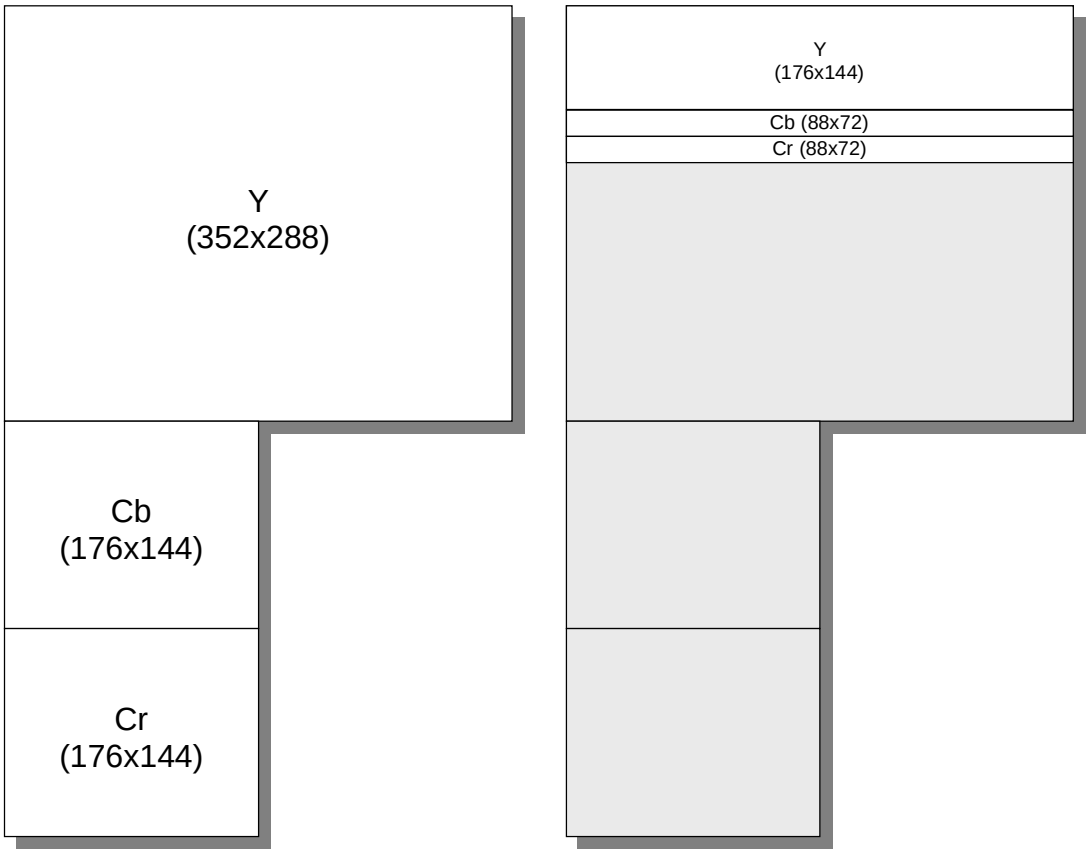


Figure 19. Frame Buffer for CIF and QCIF

2.8.2 Reading Reference MB (`rdRefMB`)

The `rdRefMB` function transfers the required data from the reference frame buffer to `refMB`.

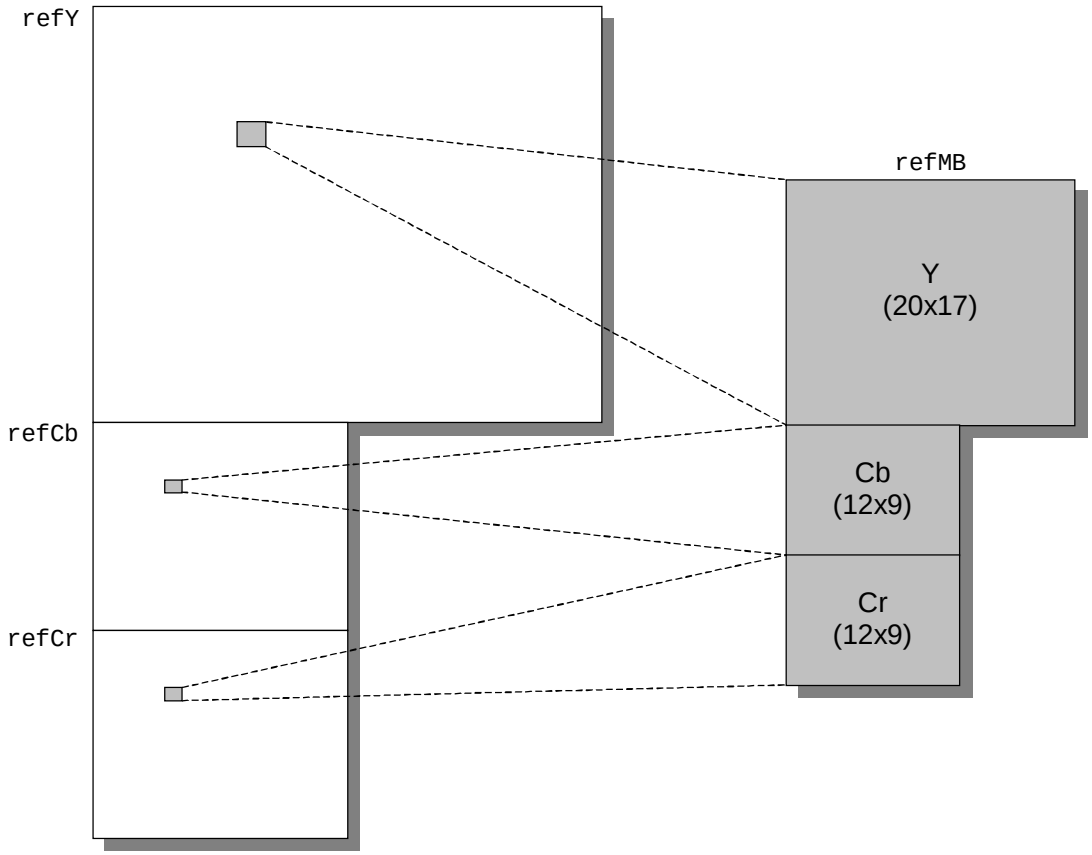


Figure 20. Reading Reference MB

2.8.3 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 TMS320C62x and TMS320C64x.

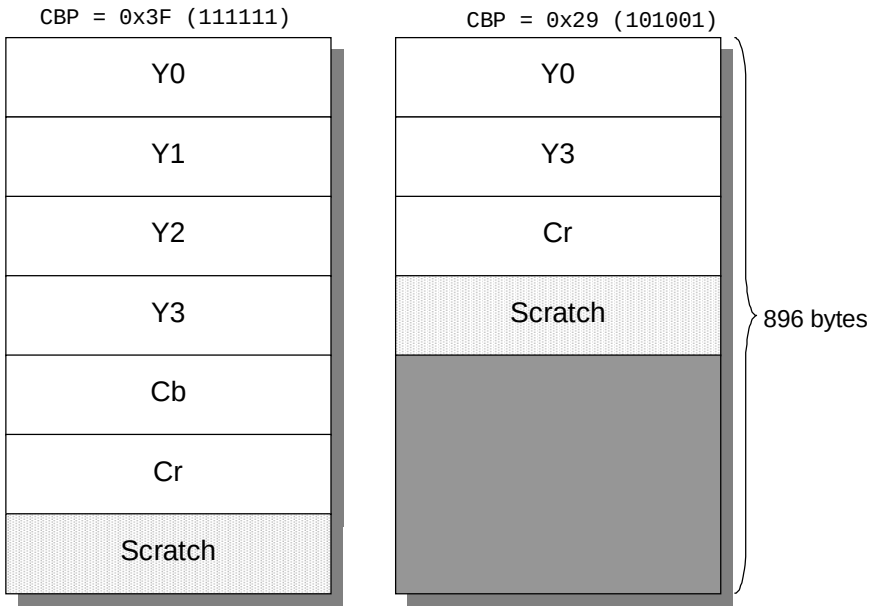


Figure 21. Examples of Using `idctBuff` (TMS320C62x)

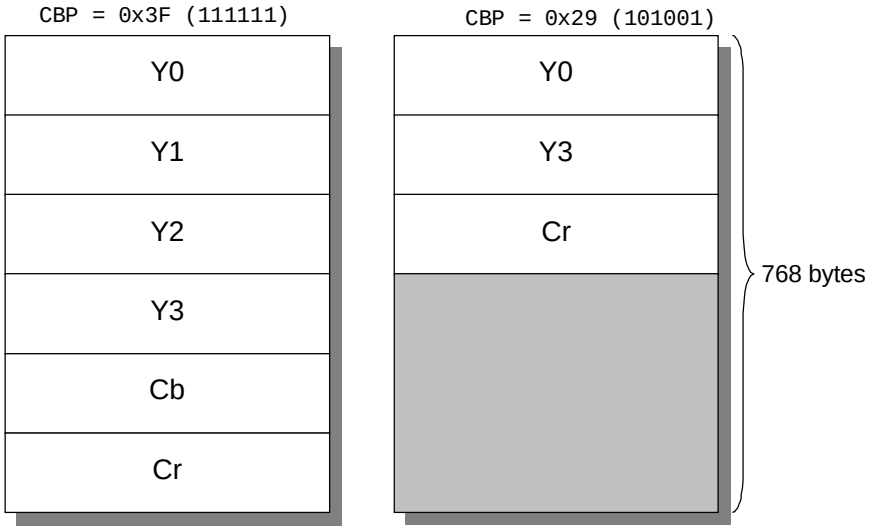


Figure 22. Examples of Using `idctBuff` (TMS320C64x)

2.8.4 Packing INTRA MB (packmb)

The decoder uses a highly optimised IDCT kernel, which 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, which 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 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.

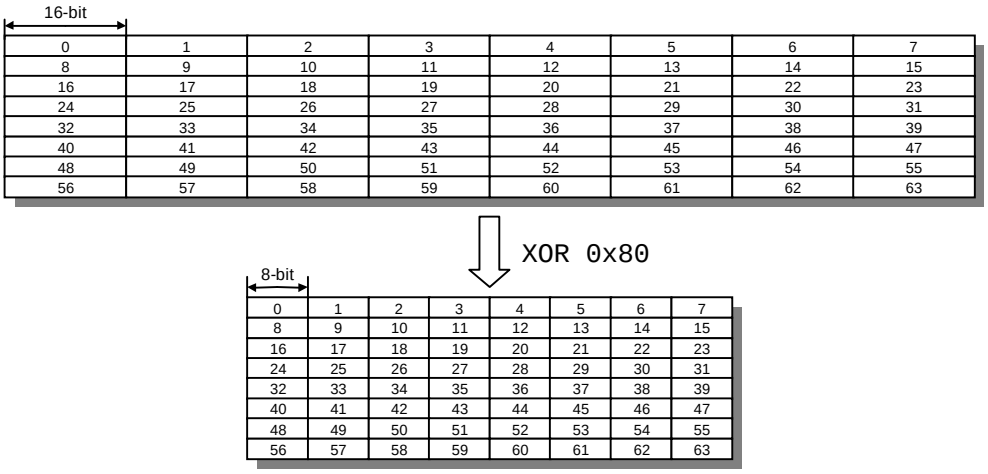


Figure 23. Processing One 8x8 Block (TMS320C62x)

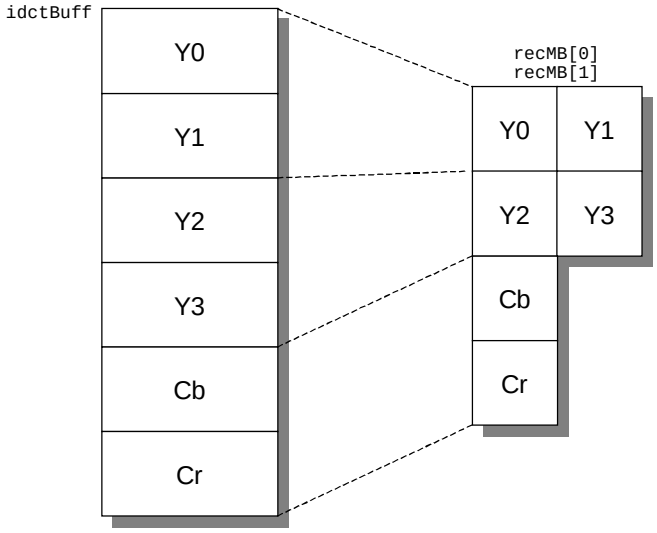


Figure 24. Processing One MB (TMS320C62x)

For optimal performance, different IDCT specifically designed for TMS320C64x is used. Unlike the TMS320C62x version, this kernel outputs results for INTRA MB in the correct manner, that is, simple saturation and packing is all that is required.

The figure below shows how each block is processed.

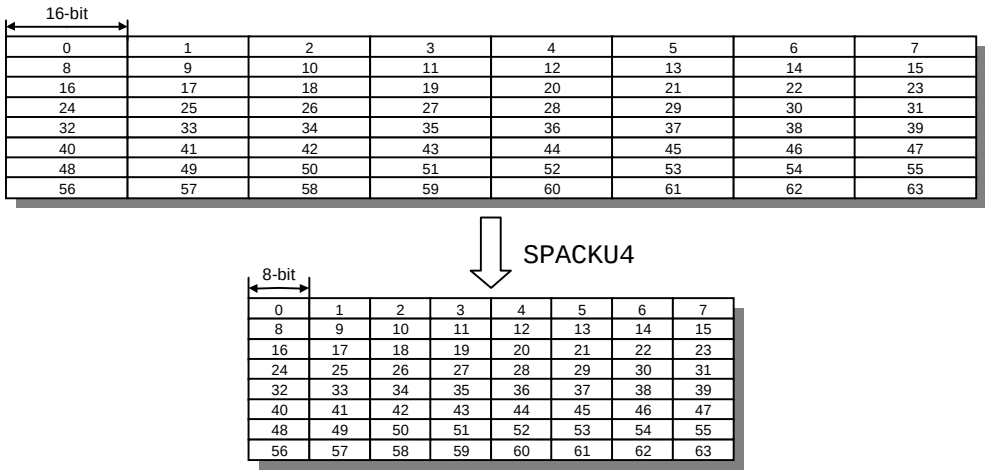


Figure 25. Processing One 8x8 Block (TMS320C64x)

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

2.8.5 Motion Compensation (h263DecMC)

The figures below show two examples of how motion compensation is applied to a MB.

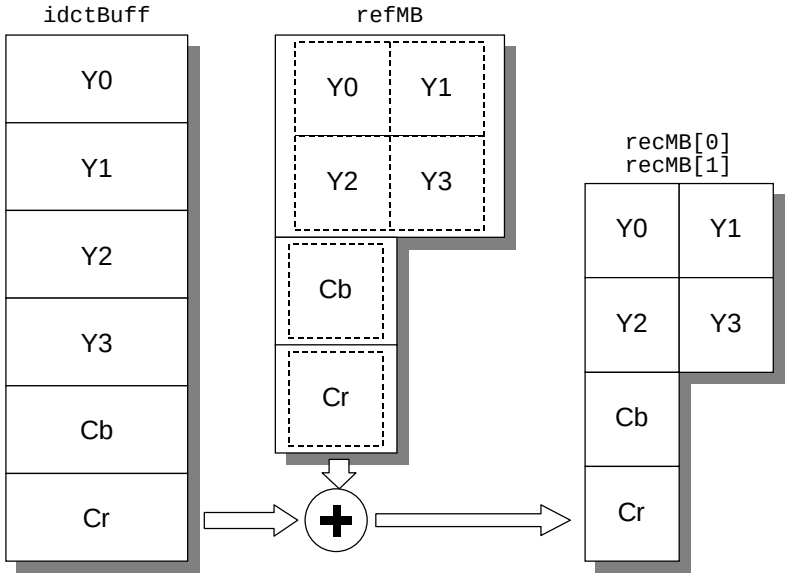


Figure 26. Motion Compensation (CBP=0x3F)

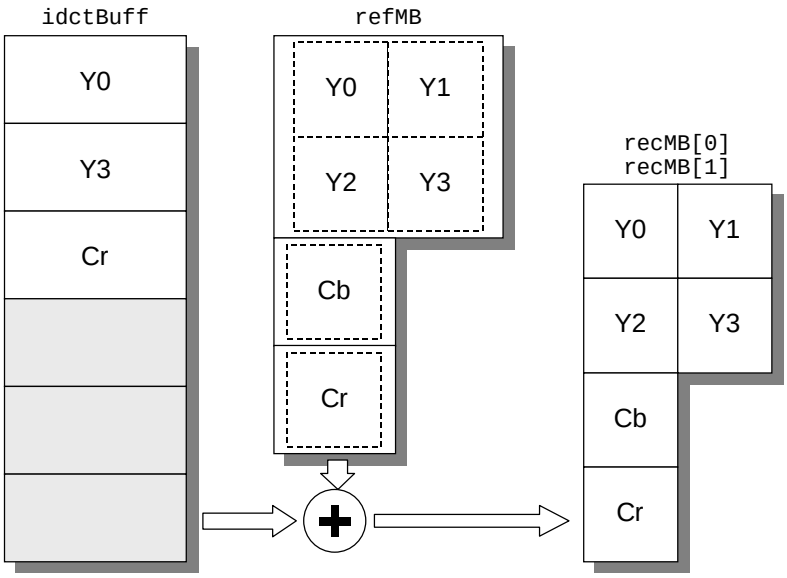


Figure 27. Motion Compensation (CBP=0x29)

2.8.6 Writing Reconstructed MB (wrRecMB)

The `wrRecMB` function writes the reconstructed MB to the output frame buffer.

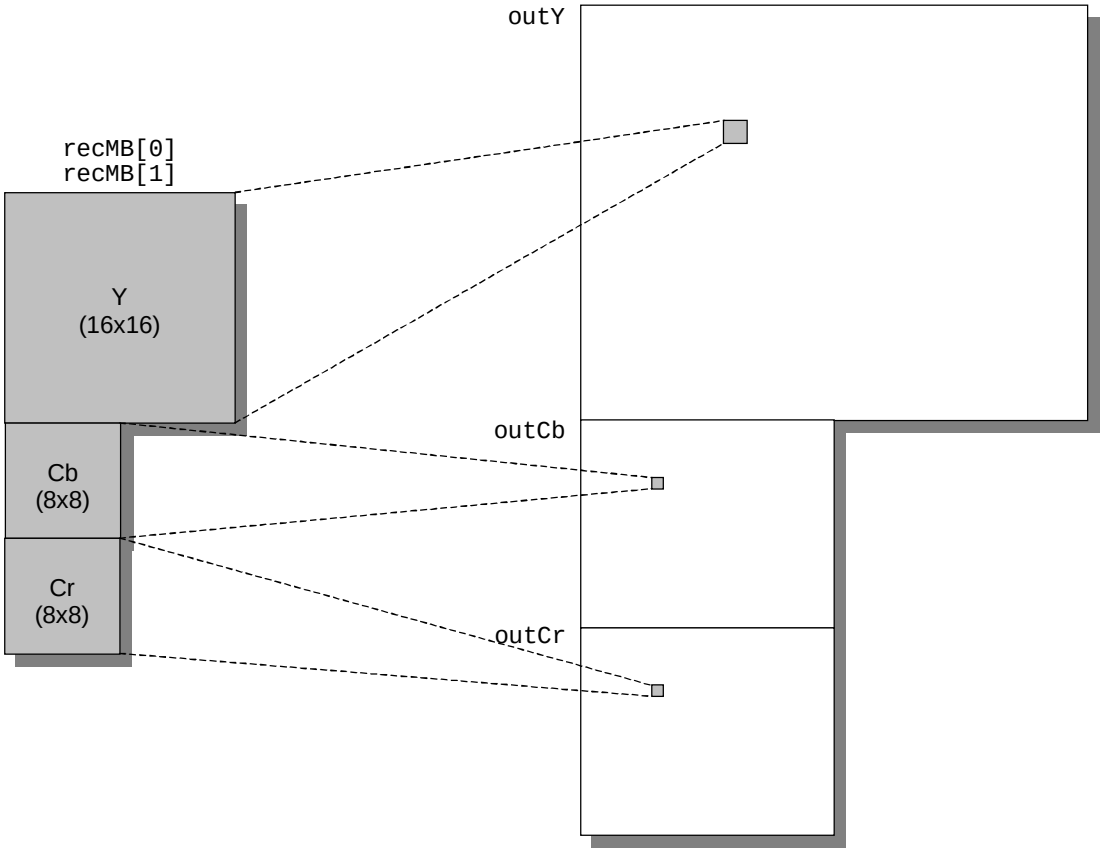


Figure 28. Writing Reconstructed MB

2.8.7 Copying Non-coded MB (cpMB)

When a MB is not coded, it is necessary to copy it from the reference frame buffer to the output frame buffer, so that the decoder may use it to reconstruct the next frame.

Due to the nature of the TMS320C6201 DMA, the MB is moved from external reference frame buffer to internal memory (`recMB`), and then to external output frame buffer.

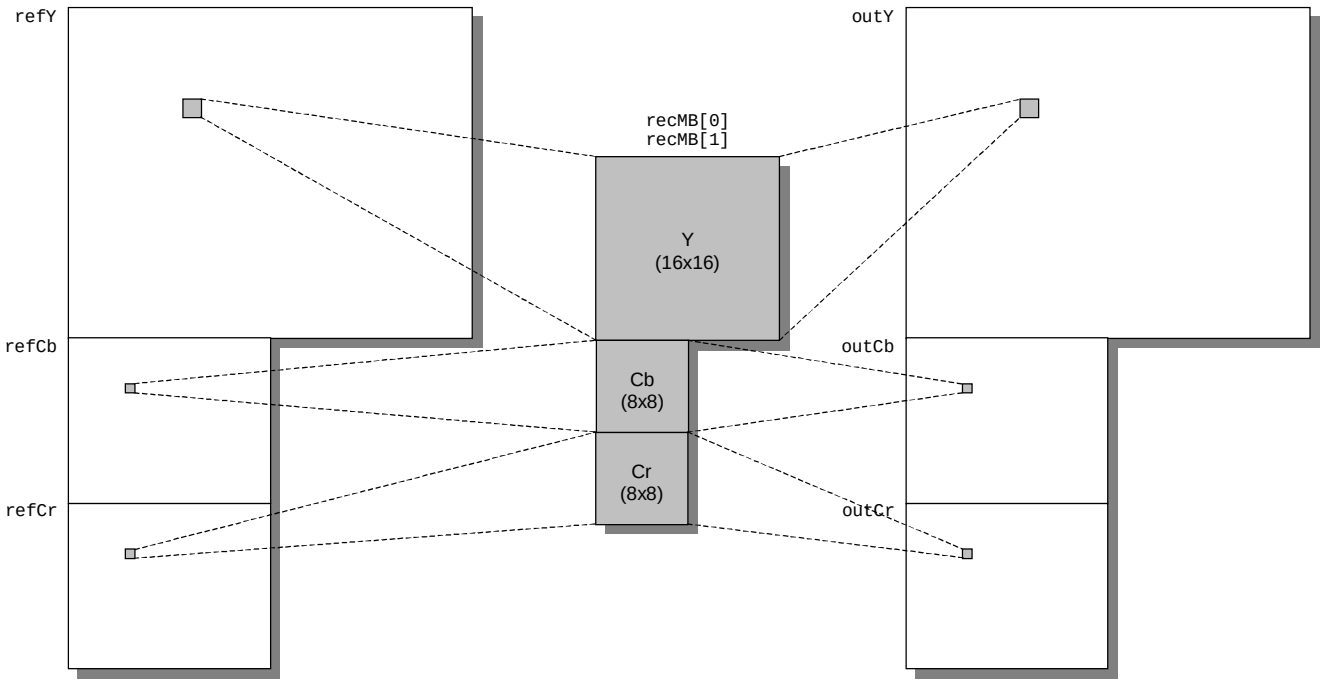


Figure 29. Copying MB from Reference to Output (DMA)

The EDMA (TMS320C6211 and TMS320C64x) processes external-to-external data transfers efficiently, so the code simply issues three external-to-external transfer requests, as shown in the figure below.

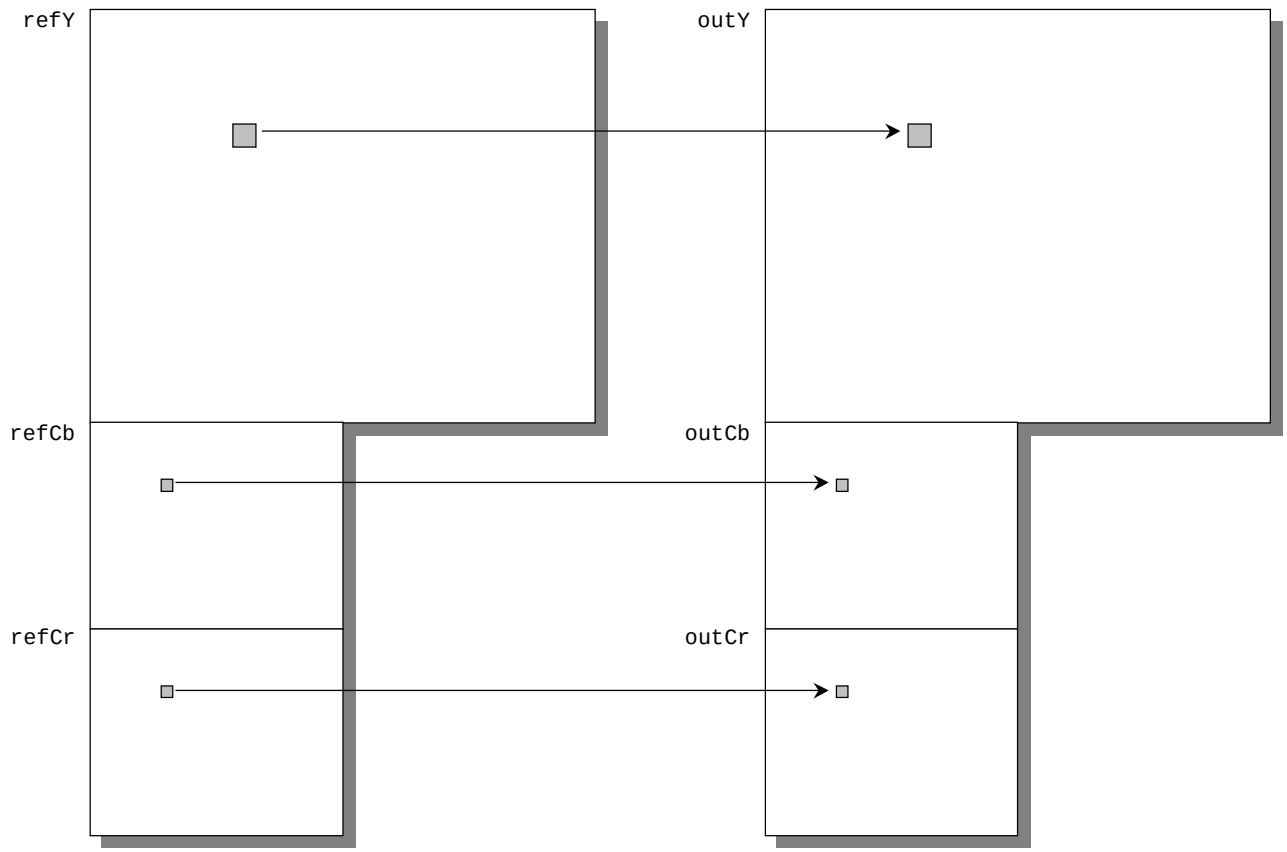


Figure 30. Copying MB from Reference to Output (EDMA)

3 Building H.263 Decoder

Sample makefiles `h263d6201.mak` and `h263d6211.mak` (for TMS320C6201 and TMS320C6211/TMS320C6711, respectively) are provided in the `mak` directory for reference. Note that this makefile is not complete and may not be used to build the complete decoder 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 Data Transfer Methods (Optional)

When no flag is specified, CSL's DAT module is used for data transfers; this is the default setting. The user may select alternative data transfer methods, depending the way the rest of the system is configured. The available options are described below.

CSLDMA: Use CSL modules DMA or EDMA (QDMA) for data transfers; the former is selected if CHIP_6201 is specified, or the latter if CHIP_6211 or CHIP_6400 is specified.

DIRDMA: Bypass CSL. With CHIP_6201 flag, the decoder programmes the DMA registers directly to issue data transfer requests. The decoder currently uses DMA channel 1 for luma, and 0 for chroma (both Cb and Cr). The user may change these by setting the defined variables `DMA_CHANNEL_Y` and `DMA_CHANNEL_C` (for luma and chroma, respectively) in `h263decode.h` as needed. With CHIP_6211 and CHIP_6400 flags, the decoder writes to the appropriate memory mapped registers to issue QDMA requests. Note that using this option means that the decoder is no longer eXpressDSP compliant.

The table below summarises which method is used to issue data transfer requests, depending on the flags specified.

Table 15. Data Transfer Methods

	CHIP_6201	CHIP_6211	CHIP_6400
None	CSL's DAT module	CSL's DAT module	CSL's DAT module
CSLDMA	CSL's DMA module	CSL's EDMA module	CSL's EDMA module
DIRDMA	Directly program DMA	Directly program QDMA	Directly program QDMA

See Appendix B, *Data Transfer Methods* for more information.

3.3 Other Flags (Optional)

NOPARENT: Do not create parent instance to store decoder tables – allow each child instance to keep its own copy of the tables. Note that only `h263dec_ti.c` has to be compiled with this flag.

RTP: Allocate structure for RTP parameters. See Appendix D, *Real-time Transport Protocol* for more information.

DSTATS: Profile the overall decoder performance. See Appendix C, *Profiling H.263 Decoder* for more information.

DSTATS_: Profile individual decoder functions. See Appendix C, *Profiling H.263 Decoder* for more information.

DEBUG: When this flag is used, the decoder spins upon detecting an error in the bitstream. Without this flag, the decoder simply exits all the way to the calling application with an appropriate error code, indicating the type of the error detected.

For files `h263dec_ti.c` and `h263pdec_ti.c`, an additional "`-ml0`" flag is required, since it accesses labels that are defined to be of type `far`.

The decoder 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 `h263d6211.cmd`.

3.4 Building

To build the H.263 decoder, 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 `decoder` and `share` directories.
- Include the decoder's `inc` directory as part of the "Include Search Path", so that the application knows about the decoder'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 `h263dec_ti.c` and `h263pdec_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.

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

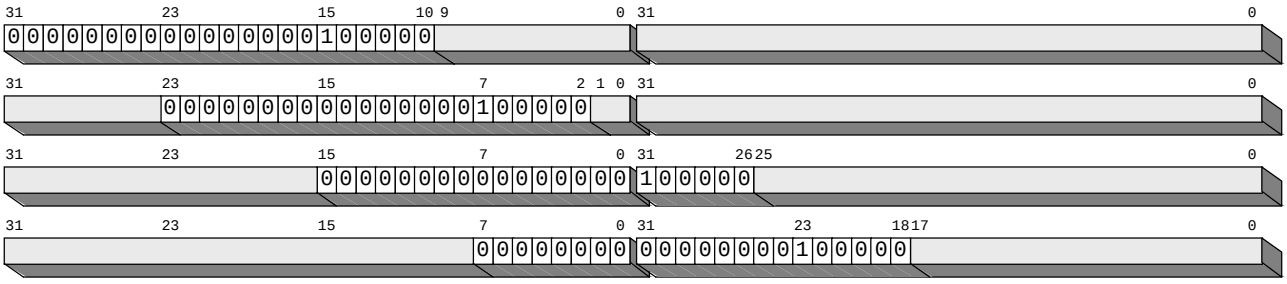
4 Assumptions and Requirements

The following lists assumptions and requirements for the decoder.

- Baseline H.263 decoder implemented.
- No big endian support for assembly (ASM) and serial assembly (SA) codes. Refer to the individual source file for more information.
- The decoder requires the input bitstream to contain a full frame per call.

- The input buffer that contains the bitstream must be aligned on a 32-bit boundary, but the bitstream itself does not have to satisfy the same criteria.

- The first word in the bitstream buffer must contain at least the first byte of PSC, as shown below. The decoder attempts to find a PSC inside the first word, before processing the current bitstream; if it does not find one, it will return with `H263D_ERR_PSC`.



- The input bitstream must match the device's endian mode within a word, as shown below. If the byte ordering is incorrect, the data must be swapped prior to calling the decoder.

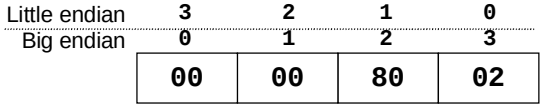


Figure 31. Loading `PSC` in both endian modes

Other assumptions and/or requirements may apply. 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

Appendix A. Performance

The tables below show the performances of all the kernels, followed by the overall decoder performance on TMS320C6201 and TMS320C6211. 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 16. Kernels Performance (TMS320C62x)

Function	SA/ASM	No. of Cycles	Per
deccbp	SA	69	MB
decmvd	SA	68	MB
dectcoef	ASM	12	Symbol
getbits	SA	14	Call
idctI	ASM	168	Block
idctP	ASM	168	Block
mcA	ASM	28	Block
mcAi	ASM	109	Block
mcB	ASM	88	Block
mcBi	ASM	165	Block
mcC	ASM	90	Block
mcCi	ASM	171	Block
mcD	ASM	133	Block
mcDi	ASM	183	Block
packmb	SA	192	MB
prezero	SA	8	Block

Table 17. H.263 Decoder Performance

					TMS320C6201		TMS320C6211	
				Frequency (MHz)	200		150	
Bitstream	Format	% INTRA	% INTER	% Coded	Cycles/ Frame	Frame Rate	Cycles/ Frame	Frame Rate
News	QCIF	1.41	39.04	59.55	246,388	812	177,532	845
News	QCIF	0.92	36.75	62.33	236,648	845	168,648	889
Foreman	QCIF	4.02	88.07	7.91	346,264	578	290,084	517
Coastguard	QCIF	1.09	92.47	6.43	341,892	585	286,860	523
Coastguard	QCIF	0.36	82.81	16.83	305,952	654	252,536	594
Foreman	CIF	6.74	82.08	11.18	1,324,240	151	1,089,296	138
Silent	CIF	0.58	31.27	68.15	890,972	224	616,624	243
Silent	CIF	1.56	35.24	63.20	943,480	212	668,564	224

For every test bitstream, the TMS320C6211 showed superior performance over the TMS320C6201. This is due largely to the EDMA and its ability to execute external-to-external transfers without having to break it up into two separate requests (which forces the CPU to wait for the first request to complete). Note that the average number of cycles used by the CPU to decode one frame (“Cycles/Frame”) includes the core decoder codes, control codes, as well as any overhead associated with calling and exiting the entire decoder instance. For the TMS320C6211, the numbers also include stalls incurred by any cache misses (L1-I, L1-D, and L2). Note also that for bitstreams with high percentage of MBs not coded (News and Silent), the TMS320C6211 is able to provide a higher overall frame rate despite a lower clock frequency.

No specific numbers are available for TMS320C64x at the time of publication.

Appendix B. Data Transfer Methods

The decoder can be configured to use one of three data transfer methods, depending on which flag is used at build time. Within each type, the code required to setup and issue transfer requests is almost identical for the two devices. In `cpMB`, however, the TMS320C6201 version first moves the whole MB to `recMB`, waits for all three requests (one for luma and two for chroma) to complete, and then writes it out to the output frame buffer (total of six requests). The TMS320C6211 and TMS320C64x versions simply issue three requests per MB.

Using CSL's DAT module

This option requires virtually no user intervention, and is the default method. Channel selection, priority, queue management, etc. are handled by CSL.

Using CSL's DMA and EDMA modules

Using the DMA (for TMS320C6201) and EDMA (for TMS320C6211/TMS320C6711 and TMS320C64x) modules gives the user more control over how each transfer request is issued, while remaining eXpressDSP compliant.

For TMS320C6201, the decoder uses DMA channel 0 for chroma and DMA channel 1 for luma. `DMA_CHANNEL_Y` and `DMA_CHANNEL_C` (defined in `h263.h`) can be modified to use different DMA channels, if necessary.

For TMS320C6211/TMS320C6711 and TMS320C64x, QDMA is used to request transfers by passing a specific handle (`EDMA_HQDMA`) to the `EDMA_ConfigB` function. All the requests made by `rdRefMB` and `wrRecMB` are issued as high priority requests, whereas those made by `cpMB` are issued as low priority requests, since the completion of these requests do not affect the decoding process until it is ready to exit.

Directly Programming DMA/EDMA

This option provides maximum overall performance, but the decoder ceases to be eXpressDSP compliant, and is best suited to a standalone system.

For TMS320C6201, the behaviour is similar to using CSL's DMA module.

For TMS320C6211/TMS320C6711 and TMS320C64x, QDMA is used by directly writing to its memory mapped registers. Refer to *TMS320C6000 Peripherals Reference Guide* for more information.

Refer to *TMS320C6000 Chip Support Library API Reference Guide* for more information regarding CSL's DAT, DMA, and EDMA modules.

Appendix C. Profiling H.263 Decoder

The H.263 decoder is equipped with a simple profiling capability that uses the device's timer via the CSL TIMER module. To enable this function, specify DSTATS or DSTATS_ flag at build time. Shown below is the structure that the decoder uses to store profiling information (defined in h263decode.h).

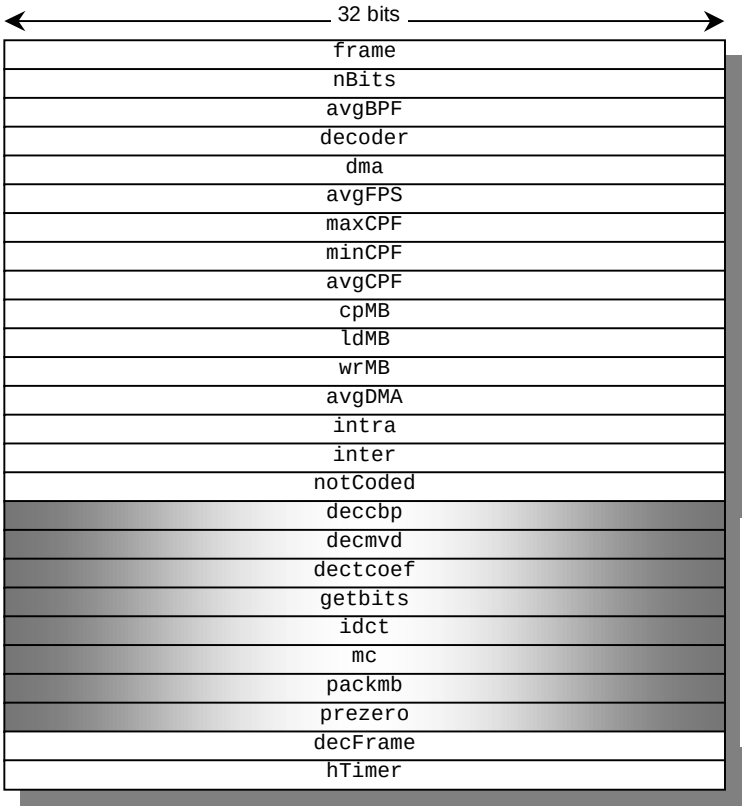


Figure 32. Decoder Statistics – H263DecStats

The shaded fields are used only when DSTATS_ flag is specified at build time. When this flag is used, other information (decoder, maxCPF, minCPF, avgCPF, and avgFPS) will not be collected.

Table 18. Decoder Statistics – H263DecStats

Name	Description
frame	Number of frames decoded. (Valid with DSTATS and DSTATS_)
nBits	Total number of bits processed
avgBPF	Average number of bits per frame
decoder	Total number of TIMER cycles used by the decoder
dma	Total number of TIMER cycles used to setup and issue data transfer requests; this is equal to the sum of cpMB, rdMB, and wrMB
avgFPS	Average frames per second (equals to CPU clock frequency divided by avgCPF)
maxCPF	Maximum number of cycles per frame

Name	Description
minCPF	Minimum number of cycles per frame
avgCPF	Average number of cycles per frame
rdMB	Total number of TIMER cycles used by rdRefMB
cpMB	Total number of TIMERcycles used by cpMB
wrMB	Total number of TIMER cycles used by wrRecMB
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
deccbp	Total number of TIMER cycles used by deccbp
decmvd	Total number of TIMER cycles used by decmvd
dectcoef	Total number of TIMER cycles used by dectcoef
getbits	Total number of TIMER cycles used by getbits
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
packmb	Total number of TIMER cycles used by packmb
prezero	Total number of TIMER cycles used by prezero
decFrame	Number of cycles used by decoder 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. 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.

Note that although it is possible to specify both DSTATS and DSTATS_, and while the numbers for individual kernels will be accurate, the numbers specific to DSTATS will be much larger than they would be if DSTATS was specified alone. This is due to the fact that by specifying DSTATS_, the overall decoder 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 <h263decode.h>

far int cpuClock; /* system clock frequency (MHz) */

/* function prototypes for decoder statistics */
#if ( (DSTATS) || (DSTATS_) )
extern H263DecStats dstats; /* defined in h263decode.c */
void decStatsInit(TIMER_HANDLE hTimer);
void decStatsUpdate(H263DecParam *dp, uint *in);
#endif

void main()
{
    TIMER_HANDLE  hTimer1;  /* timer handle                */
    uint          timerCtl; /* timer control word          */
    H263DecParam *dp;       /* decoder 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 decoder instance */

    dp = (H263DecParam *)((int)decHandle0+sizeof(IALG_Obj));

    while (1)
    {
        /* execute decoder */

#if ( (DSTATS) || (DSTATS_) )
        /* update decoder statistics */
        decStatsUpdate(dp, stream);
#endif
    }
}
```

Appendix D. Real-time Transport Protocol (RTP)

The decoder has a set of hooks for RTP support, which are partially in place, including a structure `H263RTPParam` (defined in `h263.h`) that is used to store RTP specific information.

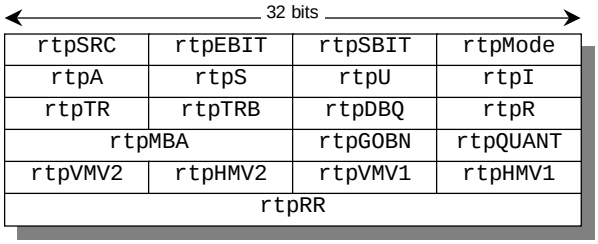


Figure 33. RTP Parameters – `H263RTPParam` (little endian)

Table 19. 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)

Refer to the RTP specification for more information on the different fields.

Appendix E. Testing H.263 Decoder

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

The figure below shows the setup used to test the decoder on TMS320C6201 EVM.

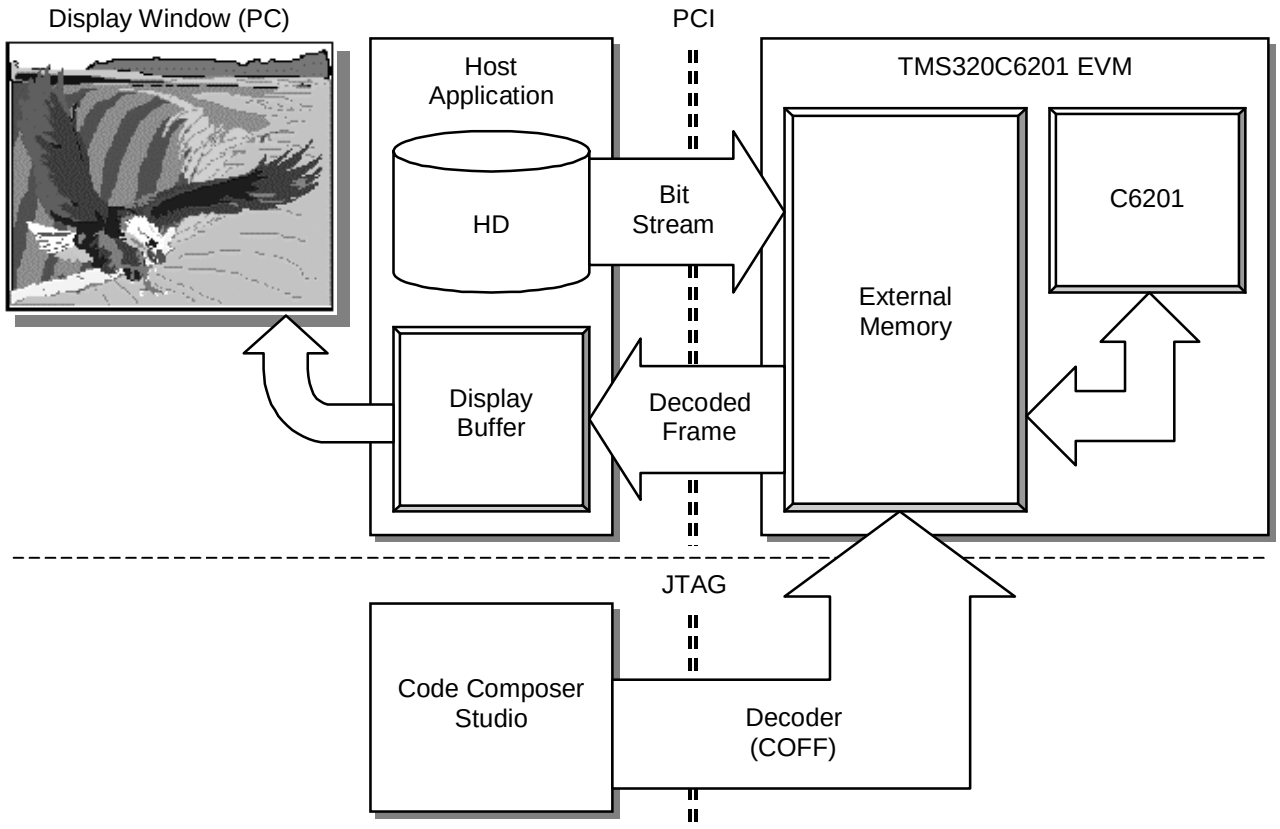


Figure 34. Test Setup on TMS320C6201 EVM

- A Windows host application running on PC sends an entire H.263 bitstream to the EVM through a PCI bus.
- The DSP parses the bitstream, decodes each frame and sends the decoded frames back to the host application through the PCI bus.
- The host application then converts the YUV data to RGB and displays the decoded video in a window on the monitor.

The figure below shows the setup used to test the decoder on TMS320C6211 DSK.

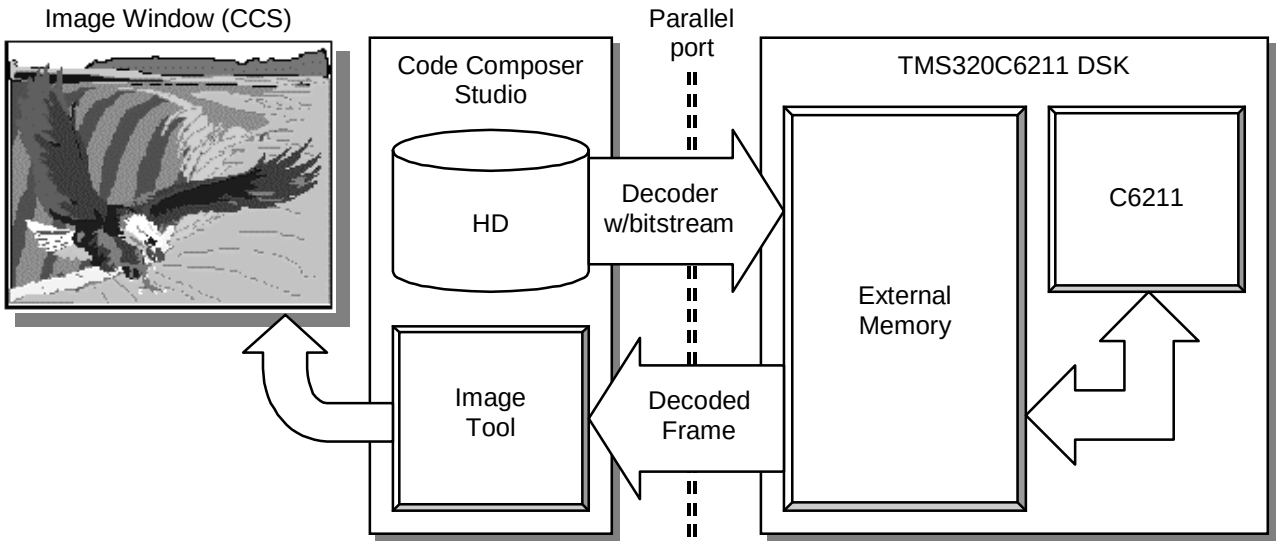


Figure 35. Test Setup on TMS320C6211 DSK

Since TMS320C6211 DSK is not equipped with a PCI interface, the testing is done in a different way. A bitstream is first converted to an ASM file, assembled and linked with the rest of the code. The whole COFF file is then loaded onto the DSK through a parallel port. The rest of the processing done on the DSP is identical, except that the decoded sequence cannot be viewed real-time. The decoded frames are checked using the image tool inside the Code Composer Studio. Refer to *Code Composer Studio User's Guide* for more information.

Appendix F. Decoding Custom Resolutions

In the default case, the decoder extracts the source format from the bitstream and sets the width and height of the image, as well as the number of GOBs and MBs (`nGOB` and `nMB`, respectively) that are present. For each frame, `h263Decode` calls `h263DecGOB` `nGOB` times; `h263DecGOB`, in turn, calls `h263DecMB` `nMB` times. This enables customising the code to decode non-standard resolutions with great ease.

For example, if the decoder is only going to decode 320x240 frames, one simply needs to make the following changes to the portion of the code in `h263Decode` (defined in `h263decode.c`). The decoder will take care of the rest.

Before

```
if (decParam->srcFormat == H263_SRCFMT_SQCIF)
{
    nGOB = 6;
    nMB  = 8;
}
else
{
    nGOB = 9  << (decParam->srcFormat-2);
    nMB  = 11 << (decParam->srcFormat-2);
}
```

After

```
nGOB = 15;  /* 320/16 = 15 GOBs/frame */
nMB  = 20;  /* 240/16 = 20 MBs/GOB    */
```

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.