CellB Codec 15

The Cell codec discussed in the preceding chapter is useful primarily in authoring applications. Its advantages are its fast decoding and the high quality of the images it produces, particularly on indexed-color frame buffers. The CellB codec, which derives from its Cell counterpart, is intended for use primarily in videoconferencing applications. It features a greater balance between the time spent compressing and decompressing images than the Cell codec and employs a fixed colormap. The CellB codec's strengths include

- Software compression at interactive rates
- Very fast decoding and display, especially on indexed-color frame buffers
- Low rates of CPU use
- Good quality output

The remainder of this chapter is divided into four sections. First, the chapter explains how the CellB compressor/decompressor works. Second, it explains briefly how to create a CellB CIS. Third, it discusses the CIS attributes that apply specifically to the CellB codec (as opposed to the general CIS attributes covered in the section "General CIS Attributes" on page 257.) Fourth, the chapter introduces the subject of accelerating the playback of CellB bytestreams. For further information on this subject, see Chapter 21, "Acceleration in XIL Programs."

15

How the Codec Works

The CellB compressor works on YC_bC_r images that conform to the guidelines set forth in CCIR Recommendation 601. The compressor performs *intraframe* compression by representing 4-by-4 blocks of pixels using cell codes. It performs *interframe* encoding using skip codes.

Cell Codes

The images you compress using the CellB compressor must have a width and height that are multiples of four because the compressor works with 4-by-4 cells of pixels. For each frame, the compressor begins with the cell in the upper-left corner and then proceeds from left to right. The compressor processes rows of cells in this way, moving from the top of the frame to the bottom.

When the compressor encodes a cell without reference to a cell in a preceding frame, it uses a four-byte cell code to represent the content of that cell. This cell code specifies two colors and includes a 16-bit bit mask that indicates which of the two colors should be used to represent each pixel in the cell. See Figure 15-1.

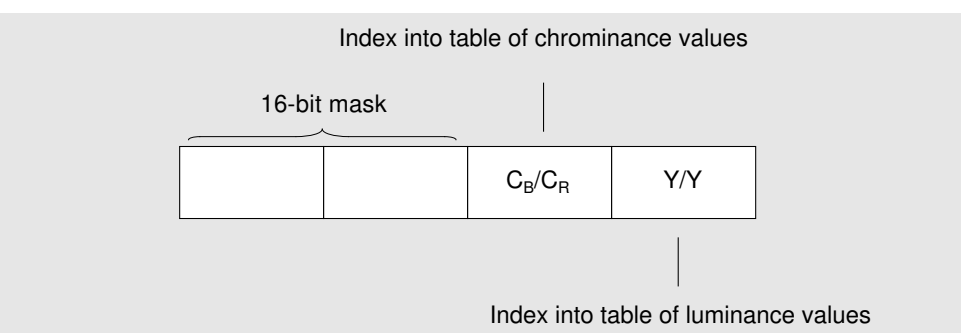

Figure 15-1 Cell Code

The two colors are encoded as follows. The compressor calculates the average C_b and C_r values for the 4-by-4 block. Then, it calculates an index into a table of 256 vectors in which each vector looks like the one shown in Figure 15-2.

Byte 0	Byte 1
C_B value	C_R value

Figure 15-2 Vectors in Chrominance Table

The values in the vector pointed to by the index are the pair of values in the table nearest to the mean C_b and C_r values for the cell. The compressor writes the index to this vector to the third byte of the cell code.

The compressor also analyzes the luminance values in the cell. First, it calculates the mean luminance for the cell. Second, it partitions the 16 luminance values in the cell into those values that fall below the mean and those that fall above the mean. Then, it calculates the average luminance in the two partitions.

After arriving at the average luminance values for the two partitions, the compressor calculates an index into a table of vectors of the form shown in Figure 15-3.

Byte 0	Byte 1
Y_0 value	Y_1 value

Figure 15-3 Vectors in Luminance Table

The values in the vector pointed to by the index are the pair of values in the table closest to the average luminance values for the two partitions. The compressor then writes the index to this vector to the fourth byte of the cell code.

The first color for a cell consists of the first byte of the cell's luminance vector and the chrominance values in the cell's chrominance vector. The second color consists of the second byte of the cell's luminance vector and the same chrominance values.

The bit mask shown in Figure 15-1 is filled out in this way. Each bit in the mask is associated with a pixel in the cell. If the luminance value for a pixel is below the mean luminance value for the cell, its bit is set to 0. This pixel will be represented by the first color when the cell is decompressed. If a pixel's luminance value is above the mean, its bit is set to 1.

Because each cell code represents the values of 16 pixels using 32 bits, compression using cell codes alone leads to a compression rate of 2 bits per pixel. To better this compression rate, the CellB compressor uses skip codes to achieve interframe compression.

Skip Codes

The CellB compressor encodes the first image in a sequence using cell codes exclusively. But after the first image, the compressor begins looking for cells in the current image that match—within a certain tolerance—the corresponding cells in the preceding image. Anywhere from 1 to 32 consecutive cells that match their counterparts in the previous image can be represented by a single 1-byte skip code.

The only restriction on the use of skip codes is that the cell at a particular set of coordinates can not be skipped indefinitely. There are a couple of reasons for this restriction. First, a user might join a videoconference that is already in progress. Cells represented by skip codes in the first image he receives will not be displayed correctly, and this problem must be corrected within a certain period of time. Similarly, if the CellB bytestream is being sent over an unreliable transport and a packet of data is lost, the period of the resulting error should be limited.

The policy concerning skip codes is that a particular cell must be updated at least every *n* frames, where *n* is an implementation-specific maximum. The exact value used is selected randomly for each cell each time that cell is encoded using a cell code. A random number is used to prevent the periodic bit-rate increase that might result were each cell to be updated at a fixed interval.

In a typical videoconference, about 80 percent of the cells in the average frame are represented by skip codes. This ratio leads to an average compression rate of about .8 bits per pixel.

Note – Details about the makeup of a CellB bytestream can be found in Appendix D, "Cell and CellB Bytestream Definitions."

Creating a CellB CIS

Before you can use the CellB codec to compress images or decompress a CellB bytestream, you must create a CellB CIS. You do this by passing the compressor name `CellB` to the function `xil_cis_create()`. See the code fragment below.

```
XilCis cis;
XilSystemState state;

cis = xil_cis_create(state, "CellB");
```

CellB Decompression Attributes

As discussed in the section "General CIS Attributes" on page 219, there is a class of attributes that can be set for any CIS. There is also a set of attributes that are valid only for a CIS attached to a CellB codec. You set these codec-specific attributes using the function `xil_cis_set_attribute()` and read them using `xil_cis_get_attribute()`.

WIDTH and HEIGHT

If you have put compressed data into your CIS using `xil_cis_put_bits()` or `xil_cis_put_bits_ptr()`, you *must* set the values of these attributes to the width and height in pixels of the images to be decompressed. If you do not set these attributes, their values will be 0, and an error will occur if you:

- Create an XIL image into which to decompress the compressed images in the CIS by calling the functions `xil_cis_get_output_type()` and `xil_create_from_type()`
- Decompress an image stored in the CIS by calling `xil_decompress()`

The legal values for both attributes are integers in the range 4 to 32764 (`short int)`. The code fragment below shows the `WIDTH` attribute being set to 320 and the `HEIGHT` attribute being set to 240.

```
XilCis cis;
short width = 320;
short height = 240;

xil_cis_set_attribute(cis, "WIDTH", (void *)width);
xil_cis_set_attribute(cis, "HEIGHT", (void *)height);
```

IGNORE_HISTORY

The `IGNORE_HISTORY` attribute affects your ability to seek forward and backward in a CellB bytestream. These seeks are somewhat problematic in CellB because the codec relies so heavily on interframe encoding and does not require periodic key frames.

By default, `IGNORE_HISTORY` is set to `FALSE`. In this case, backward seeks are illegal because no past frame contains all the information necessary to reproduce an entire image. Forward seeks are possible, but to ensure that it can decode the frame you seek to properly, the decoder must actually decode all the frames you "skip."

If you set `IGNORE_HISTORY` to `TRUE`, you're telling the decoder that it's acceptable if, after a seek, it does not produce a correct image. You're willing to let it decode only the cells described with cell codes in the frame you seek to, and to fill in bad cells as it decodes subsequent frames. In this case, backward seeks are legal, and forward seeks are faster than they would be otherwise. The drawback to setting `IGNORE_HISTORY` to `TRUE` is that you may have to decode a number of frames after a seek before you get a properly reconstructed picture. (The exact number of frames is implementation dependent.)

The code below shows `IGNORE_HISTORY` being set to `TRUE`.

```
XilCis cis;

xil_cis_set_attribute(cis, "IGNORE_HISTORY", (void *)TRUE);
```

You can also read the value of this attribute using code similar to that shown below.

```
XilCis cis;
Xil_boolean history_status;

xil_cis_get_attribute(cis, "IGNORE_HISTORY",
    (void **)&history_status);
```

CellB Molecules

The XIL library includes a series of *molecules* that accelerate the playback of CellB bytestreams. These molecules are optimized routines that perform the jobs of two or more functions from the XIL API. You do not call such an optimized routine directly; rather, the library calls a molecule when your program calls a predefined sequence of XIL functions, sometimes with specific arguments.

For example, if your program calls `xil_decompress()` to decode an image stored in a CellB CIS and then calls `xil_ordered_dither()` to convert the decoded image from a 24- to an 8-bit image, the library may not call these two functions. Instead, it may call a molecule that performs the decompression and the dithering in an optimized way.

For information about the CellB molecules that are available and information about how to call those molecules, see the section "XIL Molecules" on page 398.

15