

# The costella\_unblock library

John P. Costella

*9 Summerfield Drive, Mornington, Victoria 3931, Australia*  
*costella@bigpond.com; jpcostella@hotmail.com; assassinationscience.com/johnncostella*

(October 26, 2007)

## Abstract

This manual assists programmers in using the `costella_unblock` library in their applications.

## 1. API description

This section describes the `costella_unblock` API (Application Programming Interface). If you are an application programmer, then this section should, ideally, describe everything you need to know to use the library in your application. If something is missing, then please contact me directly.

If you actually want to know how the `costella_unblock` library code works, internally, then see Sec. 2, “Code documentation”.

### 1.1. Supported languages

The `costella_unblock` library is written completely in ANSI C. You may therefore call its functions and compile and link it in directly from any variant of C or C++.

To use the `costella_unblock` library from other languages (such as C#), you will need to link to dynamic link libraries created in C or C++. This falls beyond the scope of this document. (Note, however, that the liberal copyright permissions described in Sec. 3 facilitate the creation of redistributable libraries.)

### 1.2. Required files

To use the `costella_unblock` library, all you need are the following source code files, which are included in the standard distribution available from the author’s website, which is currently [assassinationscience.com/johnncostella](http://assassinationscience.com/johnncostella):

```
costella_base.c  
costella_base.h  
costella_body.h  
costella_debug.h  
costella_debug_default.h
```

```
costella_image.c
costella_image.h
costella_image_chrominance.c
costella_image_chrominance.h
costella_image_convert.c
costella_image_convert.h
costella_image_default.h
costella_image_<image_format>.h
costella_types.h
costella_types_<architecture>.h
costella_unblock.c
costella_unblock.h
costella_wrap.c
costella_wrap.h
```

The file `costella_image_<image_format>.h` listed above allows the `costella_image` libraries direct access to your image data in memory, without requiring you to copy the image into a special format. It is described in detail in Sec. 2.4. There are three versions of this file included in the standard distribution:

```
costella_image_separate_arrays.h
costella_image_rgb_triple.h
costella_image_win32_dib.h
```

These cover the most common storage formats, but of course can be modified for other custom formats, as described in detail in Sec. 2.4.

The file `costella_types_<architecture>.h` defines the types appropriate for your compiler, as described in Sec. 2.1. The file `costella_types_ansi.h` is included in the standard distribution, which is guaranteed to work on all compilers. You may choose to write a modified version of this file if it would lead to greater efficiency for your compiler.

### 1.3. Copyright and patent restrictions

Every file in the `costella_unblock` library contains an “MIT” type of license, as given at the end of this document. This means that you are free to use, modify, and incorporate any or all of the library into your own applications, without there being any obligation on you to do anything more than give appropriate credit to me in some appropriate place.

As far as I know, there are no patents covering any aspect of the `costella_unblock` library. Of course, ambit patent claims are now a fact of life, and it is up to you and your lawyers to determine whether you are juicy enough bait for the patent sharks.

### 1.4. Sample application: `unblock_jpeg`

Included in the standard distribution is a sample application, `unblock_jpeg`, contained in the files

```
unblock_jpeg.c
```

```
unblock_jpeg.h
costella_jpeg.c
costella_jpeg.h
```

This is an ANSI C console application, that reads in an existing JPEG file, performs the UnBlock algorithm on it, and then writes the results out to another JPEG file. To compile it, you need to have the Independent JPEG Group's `jpeglib` source code files. These are freely available on the internet from `ijg.org`. (The necessary files are also available from my website.) Note that it is *only* this sample application that requires the IJG library; the `costella_unblock` library itself *does not* need or use the IJG library in any way.

The remainder of this section uses the sample application `unblock_jpeg` to illustrate the use of the `costella_unblock` library.

## 1.5. Using the library

To use the `costella_unblock` library, you must (at the least) do the following:

1. Compile and link the following files as part of your build:

```
costella_base.c
costella_image.c
costella_image_chrominance.c
costella_image_convert.c
costella_unblock.c
costella_wrap.c
```

2. Include the header file `costella_unblock.h` in any of your source files that uses the functionality of the `costella_unblock` library.
3. Before calling any functions in the `costella_unblock` library, you must first “initialize” it by calling `costella_unblock_initialize()`. For the sample application `unblock_jpeg`, we have

```
if( !costella_unblock_initialize( stderr ) ||
    !costella_jpeg_initialize( stderr ) )
{
    fprintf( stderr, "\nERROR: Couldn't initialize libraries.\n\n" );
    exit( 1 );
}
```

(The sample application `unblock_jpeg` uses the `costella_jpeg` library in addition to the `costella_unblock` library, and hence needs to initialize both.) Note that every nontrivial function in any `costella` library returns a *success flag*; if it is not true, then something went wrong. (In this case, memory allocation in the initialization function must have failed.) The `costella` libraries will in general “clean up” after themselves if something goes wrong. (For instance, it is entirely valid for your application to retry a function call after it has failed, as many times as it likes.)

The use here of `fprintf()` and `exit()`, and the piping of error messages to `stderr`, is appropriate only because `unblock_jpeg` is a console application. In general, you need to decide how to handle errors, as described in greater detail in Sec. 2.1.

4. Your application will need to define at least one `COSTELLA_IMAGE` variable; for the sample application `unblock_jpeg`, there is only one image:

```
COSTELLA_IMAGE i;
```

5. You must “connect” this `COSTELLA_IMAGE` object with the image data that your program has allocated. For example, to set up a  $120 \times 100$  grayscale image, where the 12000 bytes of image data are in a single array `image_data` that you have previously allocated, you would write

```
i.bColor = 0;
i.bAlpha = 0;
i.udWidth = 120;
i.udHeight = 100;
i.sdRowStride = 120;
i.ig = image_data;
```

(The members of the `COSTELLA_IMAGE` structure are described in detail in Sec. 2.4; `ig` is a grayscale image structure, defined in the header file.) For the sample application, the `costella_jpeg_load()` function in the `costella_jpeg` library actually sets the `bColor`, `udWidth`, and `udHeight` members based on information in the JPEG file being read in, sets `bRgb` to false (because it stores color data in YCbCr format), and `bDownsampledChrominance` to true (because JPEG chrominance data is downsampled). The remaining members are then set in the callback function `image_new()` that is passed to `costella_jpeg_load()` to actually allocate the image memory:

```
unsigned char* abyY, * abyCb, * abyCr;
unsigned long ulNumPixels;

ulNumPixels = pi->udWidth * pi->udHeight;

if( !( abyY = malloc( ulNumPixels ) ) )
{
    /* Error. */
}

if( pi->bColor )
{
    if( !( abyCb = malloc( ulNumPixels ) ) )
    {
        /* Error. */
    }
}
```

```

    }

    if( !( abyCr = malloc( ulNumPixels ) ) )
    {
        /* Error. */
    }
}

if( bColor )
{
    pi->ic.aubRY = abyY;
    pi->ic.aubGcb = abyCb;
    pi->ic.aubBCr = abyCr;
}
else
{
    pi->ig = abyY;
}

pi->sdRowStride = pi->udWidth;

```

For this image header file, the `ic` (color image) member itself has three members, which are the array pointers for the three color channels, whereas the `ig` member is itself defined to be simply an array pointer. Note that, if you wish to change to a different image storage format, all you need to do is modify this “connection” code, and include the appropriate image header file. For example, if your color image data was stored in a single array, in `BGRBGR...` order (such as for `.bmp` files), then you would simply need to include `costella_image_rgb_triple.h`, and modify the above code to something like

```

unsigned char* abyImageData;
unsigned long ulNumBytes;

ulNumBytes = 3 * pi->udWidth * pi->udHeight;

if( !( abyImageData = malloc( ulNumBytes ) ) )
{
    /* Error. */
}

if( bColor )
{
    pi->ic = abyImageData;
}
else

```

```

{
    pi->ig = abyImageData;
}

```

```

pi->sdRowStride = 3 * pi->udWidth;

```

(If each row has padding to align it to a specific byte boundary, such as for BMP files, then you simply need to set the row stride appropriately.)

Once you have done all this, you can simply call the function `CostellaUnBlock()` to perform the `UnBlock` algorithm on your image:

```

if( !costella_unblock( &i, &i, 0, 0, 0, 0, stderr ) )
{
    fprintf( stderr, "\nERROR: Couldn't UnBlock image.\n\n" );
    exit( 1 );
}

```

The arguments to `costella_unBlock()` are as follows:

`piIn` (`COSTELLA_IMAGE*`): Pointer to your input image.

`piOut` (`COSTELLA_IMAGE*`): Pointer to your output image. Can be the same as the input image. Our sample application, `unblock_jpeg`, has no further need for the input image, and so it allows `costella_unblock()` to overwrite it by correcting the image “in place”.

`bPhotographic` (`int`, *i.e.*, `Boolean`): Set this flag to true if you know for certain that the image is “photographic” (*i.e.*, continuous tone). This tells `costella_unblock()` to be somewhat more aggressive in removing block artifacts; more accurately, it leans to the “aggressive” side of its statistical confidence interval. This generally only has a substantial effect for very small images. The sample application `unblock_jpeg` has no prior information about the JPEG image being loaded, and so does not set this flag.

`bCartoon` (`int`, *i.e.*, `Boolean`): Set this flag to true if you know for certain that the image is “cartoon-like” (*i.e.*, solid blocks of color with sharp boundaries between them). This tells `costella_unblock()` to be somewhat less aggressive in removing block artifacts; more accurately, it leans to the “conservative” side of its statistical confidence interval. This generally only has a substantial effect for very small images. Again, the sample application `unblock_jpeg` has no prior information about the JPEG image being loaded, and so does not set this flag.

`pfProgress` (pointer to function): This is an optional “progress callback” function pointer, that allows your application to be “called back” regularly during time-consuming processing tasks. This is described in detail in Sec. 2.1.

`pvPassback` (`void*`, *i.e.*, generic pointer): This is the generic object pointer that will be passed through to your callback function by `pfProgress()`. Again, see Sec. 2.1 for a full description.

Once you have finished using the `costella_unblock` library, you may, if you wish, call the function `costella_unblock_finalize()`, which cleans up the library and returns it to its uninitialized state. (You may subsequently reinitialize it and start using it again, if you wish.)

## 2. Code documentation

This section describes how the code in the `costella_unblock` library works. If you only want to know how to use the API to the library, then see Sec. 1, “API description”.

### 2.1. Introduction to the `costella` libraries

This section provides a general introduction to and overview of the `costella` libraries.

#### Requirements

The `costella` libraries have been written in ANSI C, and do not rely on any other code whatsoever. To use them, all you need is a compiler compatible with ANSI C, that has a memory model that uses at least 32-bit addressing, and that does not have restrictions on the length of unique identifiers.

Although they are written in ANSI C, the `costella` libraries are highly object-oriented. They have been written in such a way as to maximize both performance and portability. They also interface seamlessly to other languages of choice (*e.g.*, C++). You may, if you like, write a trivial “wrapper” (in your language of choice) for any classes of the `costella` libraries that you need to interface directly to.

#### Public interface

Each of the `costella` libraries, and each piece of functionality within each such library, has a “public interface”. This is a standard ANSI C interface, that has the same “look and feel” of any regular ANSI C API. If you are an application developer, you will generally use this “public interface”, and you will not generally need to read anything further in this document.

#### Internal coding conventions

If you want to look inside the `costella` libraries, to figure out how they work, then you have to contend with the fact that they do not look as “standard” as their respective public interfaces. They have been coded with a certain amount of the author’s “wrapping” which allows a more robust object-oriented approach, while remaining completely within the confines of ANSI C (with the above two exceptions). This means that you will have to spend a little time “getting up to speed” with the peculiarities of these conventions. Balancing this negative factor, however, is the fact that every line of every `costella` library is extensively documented.

Particular peculiarities of the conventions used internally within the `costella` libraries include the following:

- Hungarian notation is used for variables (see the next section).
- Spaces are used around binary operators, within parentheses (except type casts and “protection” parentheses around parameters in a macro), within square brackets, and after commas and semicolons. For example,

```
for( aub[ 1 ] = 0; aub[ 1 ] < 10; aub[ 1 ]++, aub[ 2 ]-- )
...

```

- Subsidiary blocks of code are indented by two spaces.
- Blocking braces are vertically aligned. For example,

```
{
  ub = 3;
}
```

- No line of source code should be more than 76 characters wide (excluding the end-of-line character).
- Extra spacing or line breaking is *not* used to align objects. Lines of code are only broken:
  - when dictated by the 76-character rule (subsequent lines are indented by two spaces, until the end of the code statement);
  - when the statement is completed with the concluding semicolon; or
  - when a subsidiary block is started (each opening and closing brace always has a line to itself).
- Comment blocks start with the ANSI standard `/*` on the first line, `*/` on the last line, and `**` in any lines in between, all aligned vertically with the code. An example of a comment that fits on a single line:

```
/* Initialize.
*/
```

An example of a multi-line comment:

```
/* Check that the minimum possible message is not greater than the
** maximum possible message.
*/
```

- Although C allows single statements for `if`, `else`, `for`, `while`, *etc.*, without blocking braces, this capability is not used; blocking braces are always used, as in Perl. For example,

```

if( ub > ubMaximum )
{
    ub = ubMaximum;
}

```

- Pointers are declared with the ‘\*’ attached to the type, rather than the variable. For example,

```
COSTELLA_UB* pub;
```

When more than one is declared, the extra ‘\*’ symbols are likewise separated from the variable by spaces; for example,

```
COSTELLA_UB* pubMaximum, * pubMinimum;
```

- Variables definitions at the start of a function (or the start of a file, for global variables) are collected together by type. They are ordered primarily on the “complexity” of the fundamental type (the COSTELLA\_XX types are defined below):

1. COSTELLA\_B;
2. COSTELLA\_C;
3. COSTELLA\_UB;
4. COSTELLA\_SB;
5. COSTELLA\_UW;
6. COSTELLA\_SW;
7. COSTELLA\_UD;
8. COSTELLA\_SD;
9. COSTELLA\_UQ;
10. COSTELLA\_SQ;
11. COSTELLA\_MC;
12. COSTELLA\_MI;
13. COSTELLA\_SF;
14. COSTELLA\_DF;
15. COSTELLA\_O;
16. COSTELLA\_FUNCTION\_POINTER;
17. simple structures, *i.e.*, those that are designed to be defined and used “as is”;
18. class structures, *i.e.*, those that have **new** and **delete** functions defined; and
19. functions.

Secondarily, they are ordered by the level of indirection, with variables of a given type appearing before pointers to variables of that type, before pointers to pointers of that type, and so on. Moreover, within each level of indirection, the following categories of pointers are defined in separate code statements:

1. pointers to variables that will be initialized and modified in the code;
  2. pointers initialized to zero that will hold a class object created and destroyed by appropriate `New` and `Delete` functions;
  3. array pointers that will be set to the value of an array pointer within a class;
  4. array pointers with fixed storage immediately defined using `[]` notation; and
  5. array pointers initialized to zero that will be allocated, reallocated, and freed using the functions `COSTELLA_MALLOC()`, `COSTELLA_REALLOC()`, `COSTELLA_FREE()`, and their variants.
- Pointers that are dynamically allocated and freed are initialized to zero when defined, and set back to zero when they are freed.
  - Each code statement (or set of closely-related statements) is preceded by a comment, which is itself preceded by two blank lines.
  - Each function is preceded by a comment block, describing the function, its parameters, and (for the public interface, and for some internal functions) its return value, which is itself preceded by three blank lines.
  - The name of every public interface function in the `costella_xxx` library, and every internal non-public-interface function that does not use the `COSTELLA_FUNCTION` functionality described below, begins with “`costella_xxx_`”.
  - The name of every non-public-interface function in the `costella_xxx` library that uses the `COSTELLA_FUNCTION` functionality described below begins with “`CostellaXxx`”.
  - The name of every constant defined in the `costella_xxx` library (whether as a `#define` or as an `enum`) begins with “`COSTELLA_XXX_`”.
  - The name of every object class (structure type definition) in the `costella_xxx` library begins with `COSTELLA_XXX_`.
  - The name of every public interface function in the `costella_xxx` library associated with a “`yyy`” object `COSTELLA_XXX_YYY` begins with “`costella_xxx_yyy_`”.
  - The name of every non-public-interface function in the `costella_xxx` library associated with a “`yyy`” object `COSTELLA_XXX_YYY` begins with “`CostellaXxxYyy`”.
  - The name of every macro in a `costella_xxx` library begins with “`COSTELLA_XXX_`”.
  - The name of every externally visible global variable in the `costella_xxx` library begins with “`hhhCostellaXxx`”, where “`hhh`” here stands for the Hungarian prefix (see the next section).

- Every library `costella_xxx` has an initialization function that must be called before using the library, and a finalization function that may be called when the library is no longer required. Either the public interface functions `costella_xxx_initialize()` and `costella_xxx_finalize()` may be used, or else the non-public-interface functions `CostellaXxxInitialize()` and `CostellaXxxFinalize()`; the effect is the same. Every `costella` library will itself initialize and finalize any other `costella` library that it requires; hence, an application developer need only initialize and finalize the “highest level” libraries.

## Variable types

The header file `costella_base.h` (included by every header file in the `costella` libraries) includes a header file, `costella_types.h`, that includes a header file that defines types for use by the libraries. The standard header file that is included is `costella_types_ansi.h`, which is guaranteed to work for all ANSI C compilers. For specific architectures, it might be possible to slightly optimize the `costella` libraries by copying this header file, modifying it as appropriate, and including the modified header file from `costella_types.h` instead of `costella_types_ansi.h`.

The following types must be defined by the file included by `costella_types.h`:

`COSTELLA_UB` (Hungarian prefix `ub`): Unsigned byte. Unsigned integer type representing an eight-bit byte. Generally `unsigned char`.

`COSTELLA_SB` (Hungarian prefix `sb`): Signed byte. Signed integer type representing an eight-bit integer. Generally `signed char`.

`COSTELLA_UW` (Hungarian prefix `uw`): Unsigned word. Unsigned integer type containing at least 16 bits. ANSI C guarantees that the type `unsigned short` satisfies this requirement.

`COSTELLA_SW` (Hungarian prefix `sw`): Signed word. Signed integer type containing at least 16 bits. ANSI C guarantees that the type `signed short` satisfies this requirement.

`COSTELLA_UD` (Hungarian prefix `ud`): Unsigned doubleword. Unsigned integer type containing at least 32 bits. ANSI C guarantees that the type `unsigned long` satisfies this requirement.

`COSTELLA_SD` (Hungarian prefix `sd`): Signed doubleword. Signed integer type containing at least 32 bits. ANSI C guarantees that the type `signed long` satisfies this requirement.

`COSTELLA_SF` (Hungarian prefix `sf`): Single-precision floating point quantity. At the current time, the precise width of this type is not defined; this may be specified in future releases. A default setting of `float` has been used for ANSI C.

`COSTELLA_DF` (Hungarian prefix `df`): Double-precision floating point quantity. At the current time, the precise width of this type is not defined; this may be specified in future releases. A default setting of `double` has been used for ANSI C.

The following types do not need to be defined in the file included by `costella_types.h`; they are defined in `costella_types.h` itself:

`COSTELLA_O` (Hungarian prefix `o`): Object. This type refers to a generic object. In ANSI C, pointers to generic objects are defined as `void*`; in the `costella` libraries, such pointers are defined as `COSTELLA_O*`. To bring about this equivalence, `COSTELLA_O` is defined as `void` using a `#define`.

`COSTELLA_B` (Hungarian prefix `b`): Boolean. Defined as `int`. Note that arrays of Boolean flags will usually be stored as single bits in some other integer type, so efficiency is not an issue with the definition of this type.

`COSTELLA_C` (Hungarian prefix `c`): Character. Defined as `char`. (Multibyte character sets are not supported by the `costella` libraries.)

`COSTELLA_MC` (Hungarian prefix `mc`): Memory count. Defined as `unsigned int`. Unsigned type that should be able to represent any valid count of bytes on the system. (Another option would have been `size_t`, but this is more specifically related to the functions in `stdlib.h`, which may or may not be relevant.)

`COSTELLA_MI` (Hungarian prefix `mi`): Memory index. Defined as `int`. Signed type that can represent a general array index (pointer offset).

The following further rules are used for Hungarian notation:

- A pointer to an array of objects is prefixed by `a`.
- A pointer to a specific object is prefixed by `p`.
- A pointer to a function has a prefix of `pf`.
- A global variable is prefixed by `g`.
- An argument of a macro, or a variable defined in a local block within a macro, is prefixed by `l` (ell) to make naming clashes impossible.
- Each class defined in a `costella` library has a suitable Hungarian prefix allocated to it (see the header file for the library in question to determine it what it is).

The method of constructing a Hungarian prefix is as follows:

1. Any `g` or `l` prefix appears first.
2. The prefix of the fundamental type appears last.
3. As many `a` and `p` prefixes appear as required in the correct order to describe the structure correctly, when read out from left to right. For example, a global array of pointers to `COSTELLA_UBs` would be `gapub`; a local pointer to an array of arrays of `COSTELLA_UWs` would be `lpaauw`.

## Function calling and error handling

Almost all of the non-public-interface functions in the `costella` libraries are defined with special macros that aid in the tracking of errors. The general paradigm for function definition, flow control, and return is as follows:

```
/* CostellaLibraryFunction:
**
** Does something useful.
*/

COSTELLA_FUNCTION( CostellaLibraryFunction, ( void ) )
{
    /* Check for general badness.
    */

    if( gbThingsAreBad )
    {
        /* This is an error.
        */

        COSTELLA_FUNDAMENTAL_ERROR( "Found an error" );

        /* Try to clean up. Free a string.
        */

        if( COSTELLA_FREE( gacStuff ) )
        {
            /* This failed too.
            */

            COSTELLA_CLEANUP_FUNDAMENTAL_ERROR( "Failed freeing gacStuff" );

            /* Return with error.
            */

            COSTELLA_RETURN;
        }

        /* Call some other cleanup function.
        */

        if( COSTELLA_CLEANUP_CALL( CostellaAnotherlibraryFunction() ) )
```

```

{
  /* Cleanup function also encountered an error.
  */

  COSTELLA_CLEANUP_ERROR( "Cleanup had an error too" );

  /* Return with error.
  */

  COSTELLA_RETURN;
}

/* Return with error.
*/

COSTELLA_RETURN;
}

/* Didn't find any general badness. Try to do something useful.
*/

if( COSTELLA_CALL( CostellaOtherlibraryFunction() ) )
{
  /* It "failed", but this represents a simple end of data. Ignore
  ** the error.
  */

  COSTELLA_IGNORE_ERRORS;

  /* Set appropriate flag.
  */

  gbDataEnded;
}
}
COSTELLA_END_FUNCTION

```

The macro `COSTELLA_FUNCTION()` starts the function definition, but also defines a number of hidden variables that allow for error tracking.

If the function wants to return with an error that it has fundamentally trapped itself (*i.e.*, not one that has come from calling some other `costella` library function using this paradigm), then it first calls `COSTELLA_FUNDAMENTAL_ERROR()`, enclosing an explanatory

string as its argument. (Note that, in terms of performance, this string *does not* get copied anywhere; a pointer to its static allocation is simply added to a tree of such error messages, as is a pointer to a statically allocated string for the function name itself; these are defined, hidden from view, in `COSTELLA_FUNCTION()`). It then invokes `COSTELLA_RETURN` to return; a pointer to the error string is automatically available to the calling function.

Before doing this, however, the function will generally try to “clean up” after itself, *i.e.*, to delete any objects it has created up to the point of error. If it detects any “fundamental” errors in the process, it logs the fact using `COSTELLA_CLEANUP_FUNDAMENTAL_ERROR()`. Note that both the original error string *and* the cleanup error string will be available to the calling function.

If an error is detected during cleanup, then it is assumed that the function will abort any attempt at further cleanup, and will immediately return.

If the function calls another `costella` library function (other than internal functions that are not defined using this paradigm) by using `COSTELLA_CALL()`, and the result of that call is nonzero, then the function call has itself trapped an error. The function then need simply call `COSTELLA_ERROR()` (rather than `COSTELLA_FUNDAMENTAL_ERROR()`), again enclosing its own explanatory string as its argument. The error reported by the `COSTELLA_CALL()` is automatically included in the tree of error messages that is returned to the calling function.

On the other hand, if the function does *not* wish to return this error to *its* calling function (for example, if the “error” is a valid signal of something, not a reason to return with an error), then it should issue the `COSTELLA_IGNORE_ERRORS` command. This ensures that any subsequent `COSTELLA_RETURN` that may be issued by the function will not be an “error” return, but rather an error-free return (unless another error is trapped in the interim). It also ensures that memory leakage does not occur, as would happen if another `COSTELLA_CALL()` were to be issued with a trapped error pending.

If the function wishes to call another `costella` library function during the cleanup of an error, then it uses `COSTELLA_CLEANUP_CALL()` rather than `COSTELLA_CALL()`. If the result of this call is nonzero, then it signifies that the cleanup function call has detected an error. The function then specifies its own error string using `COSTELLA_CLEANUP_ERROR()`, and returns using `COSTELLA_RETURN`. Likewise, if it wishes to ignore the cleanup error, then it can issue the command `COSTELLA_IGNORE_CLEANUP_ERROR`.

At the end of the function body, the macro `COSTELLA_END_FUNCTION` must be invoked. This returns to the calling function, advising it that no error occurs. (The same result can be achieved at any point of the code by issuing a `COSTELLA_RETURN` command, if no errors have been encountered or logged to that point.) It is necessary to encapsulate the function-ending functionality in a macro, because the definition of hidden variables in `COSTELLA_FUNCTION()` actually opens two extra sets of scoping braces that are hidden from the programmer, which are actually closed in `COSTELLA_END_FUNCTION`. It also frees any error-handling structures that the programmer has chosen not to process.

In summary, function definition, flow control, and return for each function in a `costella` library requires the use of the following macro constructions, in various combinations:

```
COSTELLA_FUNCTION( function_name, ( arguments ) )
```

```
    COSTELLA_FUNDAMENTAL_ERROR( error_string );
```

```

COSTELLA_CALL( function_call )
    COSTELLA_ERROR( error_string );
    COSTELLA_IGNORE_ERRORS;

COSTELLA_CLEANUP_FUNDAMENTAL_ERROR( error_string );

COSTELLA_CLEANUP_CALL( function_call )
    COSTELLA_CLEANUP_ERROR( error_string );
    COSTELLA_IGNORE_CLEANUP_ERROR;

COSTELLA_RETURN;

COSTELLA_END_FUNCTION

```

### Special requirements of public interface functions

The public interface functions in every `costella` library need to call `costella` library functions, but these public interface functions are (by definition) not themselves functions following the internal `costella` library conventions as described above. Such public interface functions must be defined using the macros

```

COSTELLA_ANSI_FUNCTION( return_type, function_name, ( arguments ) )
COSTELLA_END_ANSI_FUNCTION( return_value )
COSTELLA_END_ANSI_FUNCTION_NO_RETURN_VALUE

```

which set up the function in question to automatically trap and register any errors returned by a `costella` library function call, without requiring any function calling *it* to possess such functionality.

If you wish to hook in *directly* to any non-public-interface functions in any `costella` library, then you must similarly use these macros; otherwise, your application will leak memory. (Note, however, that in the vast majority of cases it will be preferable for you to only use the public interface functions.)

There is a catch with `COSTELLA_END_ANSI_FUNCTION()`: the return value expression cannot make use of any of the variables defined in the function in question, because such variables actually go out of scope before the macro is executed. If such a function needs to return a specific value, then it must use

```

COSTELLA_ANSI_RETURN( return_value )

```

just before the end of the function. (This macro actually be used anywhere in the function.) The function must then call `COSTELLA_END_ANSI_FUNCTION()` at the end of the function with a “dummy” value. (If the return type is not a standard type, then it may be necessary to declare a dummy global variable for this purpose: the variable needs to be global so that it is in scope.)

There is also a macro

## COSTELLA\_ANSI\_RETURN\_NO\_VALUE

for returning from a `COSTELLA_ANSI_FUNCTION` that has no return value from any point within the function. It is necessary to use these macros, rather than simply issuing `return` statements, if memory leakage in the error-handling functionality is to be avoided.

The macro `COSTELLA_END_ANSI_FUNCTION_NO_RETURN_VALUE` is used to end an ANSI function for which `return_type` has been set to `void`.

Now, if you create a function that directly accesses a non-public-interface `costella` library function, then you need to decide how to deal with errors returned by that `costella` library function. You know that an error has occurred if the `COSTELLA_CALL()` evaluates to a nonzero value. You then have four options:

1. You can ignore the error. (This is not recommended.)
2. You can simply use the fact that an error has occurred, without worrying about the details, and take appropriate action.
3. You can use the macro `COSTELLA_ERROR_FPRINT()` to output all of the details of the error tree to any file (or any stream, such as `stderr`); this is what all of the public interface functions do. This macro need simply be given the ANSI FILE pointer. For example, for a console program:

```
if( COSTELLA_CALL( CostellaLibraryFunction() ) )
{
    COSTELLA_ERROR_FPRINT( stderr );
    fprintf( stderr, "ERROR: Couldn't eat widget" );
    COSTELLA_ANSI_RETURN( 0 );
}
```

4. You can walk through the error tree itself, read all of the entries, and take appropriate action depending on what you find. The error information is contained in a tree of `COSTELLA_ERROR_NODE` objects. This object type is defined as follows:

```
typedef struct COSTELLA_ERROR_NODE_tag
{
    COSTELLA_B bAllocated;
    COSTELLA_C* acFunctionName, * acError, * acCleanupError;
    struct COSTELLA_ERROR_NODE_tag* penNext, * penCleanupNext;
}
COSTELLA_ERROR_MODE;
```

The symbol `COSTELLA_ERROR_ROOT_NODE` points to the root node of this tree. Once a `COSTELLA_CALL()` has been issued, `COSTELLA_ERROR_ROOT_NODE` will either be null (if no error occurred with the call), or else will point to the root node of the tree (if an error occurred).

For each node in the tree, the member `acFunctionName` is a pointer for the name of the function which created the error node; `acError` is a pointer to an error string (which

should not be null); `acCleanupError` is a pointer to a cleanup error string (which may be null, and indeed will usually be null, if there is no cleanup error); `penNext` points to the daughter node in the error tree (or is null if the node in question represents the fundamental error); and `penCleanupNext` points to the daughter node in the error tree for the cleanup error (or is null if there was no cleanup error, or if the node in question is the one at which the fundamental cleanup error was trapped).

The member `bAllocated` should normally be true: it indicates whether the node has been allocated. If this is false, then it indicates that the error-handling functionality was unable to allocate memory for the node. In this critical situation, it returns a pointer to the static automatic error node structure contained within the function in which the node memory allocation failed. Since this is not re-entrant, any daughter error trees are first freed, so that the tree cannot contain any endless self-referential loops. This means that all error information for the daughter trees is lost. However, in this situation, it is highly likely that the problem was that the process ran out of memory (since that is what happened when it tried to allocate the very modest amount of memory required for an error node), and so it is a fair assumption (although not absolutely certain) that this was the fundamental error. In any case, this is the best guess as to what the error was, in an environment in which dynamic memory cannot be allocated at all.

## Progress callback functions

Throughout the `costella` libraries, any function that can typically take a significant amount of processing time is passed two standardized pointers as the final two arguments:

```
... COSTELLA_CALLBACK_FUNCTION pfProgress, COSTELLA_0* poPassback, ...
```

These are provided to allow regular callback to the user's code during the progress of execution of time-consuming operations. These callback functions are, in turn, "wrapped" in the public interface functions, by using the functionality in the `costella_wrap` library.

The user's code may use these progress callbacks for any purpose (typically to return control to the operating system's message loop, or to update output).

If `pfProgress` is null, then the calling function is specifying that no progress callback functionality is required. This is generally the case if the user specified a null progress function pointer to the public interface function.

If `pfProgress` is not null, then the function will call `pfProgress()` sufficiently frequently to be useful in most situations. The function `pfProgress()` will be called with `poPassback` as its argument, which is therefore a generic pointer that the user's code can "pass through" the given `costella` library and back to itself, when the progress callback function is executed. (The `costella_wrap` library handles the "wrapping" of this passback pointer.) It can either be a direct pointer to something useful, or it can be a pointer to a structure containing as much identifying information as the user's code requires.

If the calling function specifies a progress callback function, then the passback function must be constructed using the function calling and error handling paradigm described above. If, however, the user specifies a progress callback function through a public interface function, then it will be of the form

```
int (*pfProgress)( void* pvPassback );
```

In such a case, a zero return value from the user's progress callback function indicates an error; a nonzero return value indicates no error. (Note that a zero return value has the opposite meaning than it does for a non-public-interface `costella` library function.)

If the passback function returns with an error, then the processing of the function will be halted; it will clean up and return with its own error string. Otherwise, processing continues uninterrupted.

Note that, from the point of view of the terminology, any reason for terminating the processing is considered an "error", even if the reason for termination is not truly an "error" but is rather a "cancel" signal from a user or another process.

## Debug mode

The `costella` libraries perform a small amount of extra consistency checking when the symbol `COSTELLA_DEBUG` is defined, such as:

- checking that the given library is initialized at the start of each call to a function of that library;
- checking that each pointer passed to a function that must be non-null is, in fact, non-null; and
- other checking of errors that are due to incorrect programming, rather than runtime or input errors.

The header file `costella_debug.h` must include a header file that defines `COSTELLA_DEBUG` as appropriate for your system. By default, the header file `costella_debug_default.h` is included, which defines `COSTELLA_DEBUG` if the symbol `_DEBUG` is defined. This file can be copied and modified to suit your particular system.

## 2.2. The `costella_base` library

This library is the fundamental library upon which all of the `costella` libraries are based. It handles all memory allocation and freeing (using functions that may be supplied by the calling application), and provides several macros of general usefulness.

### Dependencies

The `costella_base` library does not depend on any other library.

### Public interface functions

The public interface functions in the `costella_base` library are as follows:

```
costella_base_initialize()
```

This is the public interface function for initializing the `costella_base` library. Its function prototype is

```
int costella_base_initialize( void* (*pfMalloc)( unsigned int
    uiNumBytes ), void* (*pfRealloc)( void* pv, unsigned int
    uiNewNumBytes ), int (*pfFree)( void* pv ), FILE* pfileError );
```

`pfMalloc`: Function to be used to allocate memory. This function must behave in the same way as `costella_base_malloc_ansi()`, as described further below.

`pfRealloc`: Function to be used to reallocate memory. This function must behave in the same way as `costella_base_realloc_ansi()`, as described further below.

`pfFree`: Function to be used to free memory. This function must behave in the same way as `costella_base_free_ansi()`, as described further below.

`pfileError`: Pointer to a previously-opened ANSI FILE stream into which the error tree will be written if an error occurs. A value of 0 specifies that the error tree is not required.

*Return value*: Zero if there was an error, or nonzero if there was no error.

Usually, an application will simply initialize the highest-level `costella` libraries that are required, and will then allow the lower-level libraries, such as `costella_base`, to be automatically initialized by them. However, calling `costella_base_initialize()` explicitly, *before* any other `costella` library is initialized, allows the programmer to specify the functions to be used for allocating, reallocating, and freeing memory.

#### `costella_base_finalize()`

This is the public interface function for finalizing the `costella_base` library. Its function prototype is

```
int costella_base_finalize( FILE* pfileError );
```

`pfileError`: Pointer to a previously-opened ANSI FILE stream into which the error tree will be written if an error occurs. A value of 0 specifies that the error tree is not required.

*Return value*: Zero if there was an error, or nonzero if there was no error.

### **Public interface macros**

The public interface macros in the `costella_base` library are as follows:

#### `COSTELLA_INITIALIZE_ARRAY()`

This macro initializes all elements of an array to zero. It is typically used when one has an array of *pointers* to objects, and where it is intended that these pointers will be used to create new objects; they therefore need to be zeroed in order to flag to the `costella` libraries that they are unused, *i.e.*, available for use. However, the macro is also used throughout the `costella` libraries whenever it is necessary to zero the elements of an array of any built-in type (typically integers). Its macro prototype is

```
#define COSTELLA_INITIALIZE_ARRAY( lax, lcmLength, ltype ) \
```

`lax`: Array pointer.

`lcmLength`: Length of the array.

`ltype`: Type of object being initialized.

For example,

```
COSTELLA_INITIALIZE_ARRAY( apwidget, 100, COSTELLA_WIDGET* );
```

The macro computes the “end” address as the start address plus the number of elements in the array; the loop continues until the walking pointer hits this “end” address. The type is needed because ANSI C does not have a `typeof` command.

```
COSTELLA_INITIALIZE_OFFSET_ARRAY()
```

This macro initializes all elements of an offset array to zero. Its macro prototype is

```
#define COSTELLA_INITIALIZE_OFFSET_ARRAY( lax, lmiMin, lmiMax, ltype \
) \
```

`lax`: Array pointer.

`lmiMin`: Minimum offset of the array.

`lmiMax`: Maximum offset of the array.

`ltype`: Type of object being initialized.

For example,

```
COSTELLA_INITIALIZE_OFFSET_ARRAY( apwidget, -255, +255,
    COSTELLA_WIDGET* );
```

The macro computes the “start” and “last” addresses based on the minimum and maximum offsets; the loop continues while the walking pointer is less than or equal to this “last” address.

```
COSTELLA_SHIFT_RIGHT_TOWARDS_ZERO()
```

This macro shifts an integer one bit to the right, with the absolute value of the result never being greater than the absolute value of the original number. This operation is useful if one wants to divide a signed integer by a power of 2, with the result rounded “down” in absolute value terms; for example,  $-3/2 = -1.5 \rightarrow -1$ . Its macro prototype is

```
#define COSTELLA_SHIFT_RIGHT_TOWARDS_ZERO( lxx, lubNumBits ) \
```

`lxx`: Signed integer to shift right, towards zero.

`lubNumBits`: Number of bits to shift right.

The macro tests whether the integer in question is negative. If so, it performs the right-shift on its negative, and then negates the answer; if not, it simply performs the right-shift on the integer itself.

#### `COSTELLA_SHIFT_RIGHT_FLOOR()`

This macro shifts an integer one bit to the right, with the signed value of the result never being greater than the original number, in the sense that it would be to the right of it if they were both plotted on a number line. Its macro prototype is

```
#define COSTELLA_SHIFT_RIGHT_FLOOR( lxx, lubNumBits ) \
```

`lxx`: Signed integer to shift right.

`lubNumBits`: Number of bits to shift right.

This operation is useful if one wants to divide a signed integer by a power of 2, with the result rounded to the nearest integer. To do so, add half of denominator to the numerator in question, and then use this macro to shift by the required number of bits. For example, to divide `sdDog` by 8 and round to the nearest integer:

```
sdDogOn8 = COSTELLA_SHIFT_RIGHT_FLOOR( sdDog + 4, 3 );
```

This rounds to the nearest integer in a “democratic” fashion, namely, each possible output value maps to an equal number of input values; for this example, every possible output integer is mapped to 8 input integers. Note, however, that this is necessarily *asymmetrical* in absolute value terms: fractions of  $1/2$  are systematically rounded to the right on the number line, so that, for example,  $+4/8 \rightarrow 1$ , whereas  $-4/8 \rightarrow 0$ . (This is clear when one tries to list the eight input values that will yield a quotient of 0: seven are 0,  $\pm 1$ ,  $\pm 2$ , and  $\pm 3$ , with the eighth,  $-4$ , necessarily “breaking sign symmetry”, as a physicist would put it.)

If the symbol `COSTELLA_SHIFT_RIGHT_IS_SIGNED` is defined, then this macro uses the machine’s inbuilt right-shift functionality directly; otherwise, a minor amount of computation is required.

### Non-public-interface functions

The non-public-interface functions in the `costella_base` library are as follows:

#### `CostellaBaseInitialize()`

This function must be called before using any functionality of the library. It is called by the initialization function of all higher-level `costella` libraries. Its function prototype is

```
COSTELLA_FUNCTION( CostellaBaseInitialize, ( COSTELLA_0* (*pfMalloc)(
    COSTELLA_MC mc ), COSTELLA_0* (*pfRealloc)( COSTELLA_0* po,
    COSTELLA_MC mc ), COSTELLA_B (*pfFree)( COSTELLA_0* po ) ) )
```

`pfMalloc`: Pointer to the function that will be used to allocate memory. If null, use the function `costella_base_malloc_ansi()`.

`pfRealloc`: Pointer to the function that will be used to reallocate memory. If null, use the function `costella_base_realloc_ansi()`.

`pfFree`: Pointer to the function that will be used to free memory. If null, use the function `costella_base_free_ansi()`.

The function sets the initialization flag for the library, and then either stores the memory allocation, reallocation, and freeing function pointers supplied by the calling function, or else stores the ANSI function pointers if null pointers have been supplied.

The function checks that the memory indexing types, `COSTELLA_MC` and `COSTELLA_MI`, are at least 32 bits in width; this is assumed throughout the `Costella` libraries (and, at this date, is unlikely to be violated by any compiler of practical interest). Since the sizes of these types are predetermined at compile time, some compilers will issue a warning that the condition will always be false; to avoid this warning, the sizes are read into automatic variables, which stops most compilers from complaining.

#### `CostellaBaseFinalize()`

This function may be called when no further use is required of the library. It is called by the finalization function of all higher-level `costella` libraries. Its function prototype is

```
COSTELLA_FUNCTION( CostellaBaseFinalize, ( void ) )
```

This function doesn't actually need to do anything, other than clear the initialization flag for this library. However, to avoid compiler warnings, the hidden variable `i_1_en` is manipulated in a harmless way.

#### `costella_base_malloc_ansi()`

This function allocates memory using the ANSI C function `malloc()`. Its function prototype is

```
static COSTELLA_0* costella_base_malloc_ansi( COSTELLA_MC mc );
```

`mc`: Number of bytes to allocate.

*Return value*: The pointer to the allocated memory, or a null pointer if memory allocation failed.

This function is used by `CostellaBaseInitialize()` if a null function pointer is passed for `pfMalloc`.

### `costella_base_realloc_ansi()`

This function reallocates memory using the ANSI C function `realloc()`. Its function prototype is

```
static COSTELLA_0* costella_base_realloc_ansi( COSTELLA_0* po,  
        COSTELLA_MC mc );
```

*po*: Pointer to memory previously allocated using `costella_base_malloc_ansi()`.

*mc*: New number of bytes to allocate.

*Return value*: The pointer to the allocated memory, or a null pointer if memory reallocation failed.

This function is used by `CostellaBaseInitialize()` if a null function pointer is passed for `pfRealloc`.

### `costella_base_free_ansi()`

This function frees memory using the ANSI C function `free()`. Its function prototype is

```
static COSTELLA_B costella_base_free_ansi( COSTELLA_0* po );
```

*po*: Pointer to memory previously allocated using `costella_base_malloc_ansi()` or `costella_base_realloc_ansi()`.

*Return value*: This function always returns a nonzero value, indicating no error. (The ANSI function `free()` has no provision for signaling an error.) Note that the logic of this return value follows that of all the public interface functions (namely, that a zero value indicates failure), which is opposite to that of the non-public-interface functions (namely, that a zero value indicates no error).

This function is used by `CostellaBaseInitialize()` if a null function pointer is passed for `pfFree`.

### `costella_base_error_fprint()`

This function prints, in plain English format, an error tree report to any ANSI file or stream (including `stdout` or `stderr` if desired). It is automatically called by the macro `COSTELLA_ERROR_FPRINT()` or `COSTELLA_APP_ERROR_FPRINT()`, which supplies the address of the root node of the error tree in the function from which it was issued, and passes through from the calling application the pointer to the ANSI file to write to. The latter macro also passes an application name string and an application error description string. Its function prototype is

```
void costella_base_error_fprint( FILE* pfile, COSTELLA_ERROR_NODE*  
        penRoot, COSTELLA_C* acApplicationName, COSTELLA_C*  
        acApplicationError );
```

`pfile`: Pointer to the ANSI file to write to. If null, then do nothing.

`penRoot`: Pointer to the root `COSTELLA_ERROR_NODE`.

`acApplicationName`: String pointer for an application name. If null, no application header is written, only the error tree.

`acApplicationError`: String pointer for an application error description. If null, no application header is written, only the error tree.

Since the whole purpose of the function is to be called in an error situation, if it finds itself unable to write to the output file it simply returns, to avoid an endless loop of error handling and reporting.

If the application name and error string pointers are not null, the function first prints a standardized error message header, including “`ERROR:`”, the application name string, and the application error description string.

The function then checks whether the root node pointer passed to it is null. If so, then there is no further error information to report; the function simply returns.

Otherwise, the function prints a header for the `costella` library error tree. If the application header was printed, the header used is:

```
Costella library error tree:  
-----
```

If no application header was printed, on the other hand, then the function prints a more explanatory header. This is because the error tree in this case will *precede* the application’s error message. The header used is:

```
AN ERROR OCCURRED: This is the costella library error tree:  
-----
```

The function then calls the internal function `costella_base_error_fprint_tree()` on the root error node, with the error depth and cleanup error depth both set to zero, to print the error tree recursively.

`costella_base_error_fprint_tree()`

This internal, recursive function prints an error tree to an ANSI file. Its function prototype is

```
static void costella_base_error_fprint_tree( FILE* pfile,  
      COSTELLA_ERROR_NODE* pen, COSTELLA_UD udDepth, COSTELLA_UD udCleanup  
      );
```

`pfile`: Pointer to the ANSI file to write to.

`pen`: Pointer to the current `COSTELLA_ERROR_NODE`.

`udDepth`: Current depth.

`udCleanup`: Current cleanup level.

The function first represents the depth of the error node in the tree by indenting the output by the commensurate number of spaces. It then represents the “cleanup level” by means of a commensurate number of square brackets. It then prints the function name and the error string. It closes the cleanup level brackets, and ends the line.

The function now checks whether the error node has a daughter error node which includes the fundamental error for the subtree represented by the given error node. If so, the function calls itself recursively for this daughter node, with an error depth one greater than the error node in question.

The function now checks whether the node has a non-null cleanup string pointer in it, or alternatively whether the `bAllocated` flag is false. If either of these conditions are true, then it means that there was a cleanup error of some sort: for the former, explicitly trapped by the program; for the latter, by the error-handling functionality being unable to allocate memory for the copy of the node to add to the tree. In either case, the cleanup level is incremented. The output is again indented, and the opening brackets for the new cleanup level are printed. The function now prints the function name and the cleanup error string; in the case of an unallocated node, the string is explicitly supplied by the function, to warn the programmer that memory was exhausted and, consequently, that the daughter error trees were deleted to prevent any endless loops due to non-reentrant behavior. The closing brackets for the new cleanup level are closed, and the line is ended.

Finally, if there was a cleanup error detected in the previous step, the function checks whether the daughter node for a cleanup error tree is non-null. If so, the function calls itself recursively for this daughter node, increasing the error depth by one, and using the new cleanup level.

`costella_base_sprint_error_tree()`

This function prints an error tree to a preallocated string. Its function prototype is

```
void costella_base_sprint_error_tree( COSTELLA_C* ac,  
    COSTELLA_ERROR_NODE* penRoot );
```

`ac`: Pointer to the preallocated string.

`penRoot`: Pointer to the root `COSTELLA_ERROR_NODE`.

The function starts off the process by setting the string to an empty string, by storing the terminating null character in the first character of the string. It then calls `costella_base_sprint_error_tree_recurse()` recursively on the root node, with a depth of zero and a cleanup level of zero.

`costella_base_sprint_error_tree_recurse()`

This internal recursive function prints an error tree to a preallocated string. Its function prototype is

```
static void costella_base_sprint_error_tree_recurse( COSTELLA_C* ac,
    COSTELLA_ERROR_NODE* pen, COSTELLA_UD udDepth, COSTELLA_UD udCleanup
);
```

`ac`: Pointer to the preallocated string.

`pen`: Pointer to the current `COSTELLA_ERROR_NODE`.

`udDepth`: The current error depth.

`udCleanup`: The current cleanup level.

This function carries out the same steps as `costella_base_error_fprint_tree()`, except that its output is directed to the given string `ac`, rather than the file `pfile`.

```
costella_base_error_node_delete()
```

This function deletes a `COSTELLA_ERROR_NODE`, including any daughter nodes. Its function prototype is

```
void costella_base_error_node_delete( COSTELLA_ERROR_NODE** ppen );
```

`ppen`: Pointer to pointer to the `COSTELLA_ERROR_NODE` to delete.

The function first checks whether the pointer to pointer to the error node is null. If so, then it is an error, but there is nothing that the function can but return. Since this cannot be flagged to the programmer, and hence will not be flagged for repair, this action is taken in both debug and release mode.

The function then extracts the pointer to the error node.

If this pointer is null, then nothing needs to be done, and the function returns.

Otherwise, the function deletes the daughter nodes of the error node, by calling itself recursively.

Finally, the function checks whether the error node was allocated, or whether it is a static automatic variable. If it is was allocated, it is freed.

## Non-public-interface macros

The non-public-interface macros in the `costella_base` library are as follows:

```
COSTELLA_MALLOC()
```

This macro allocates memory on a generic basis, using the memory allocation function `pfMalloc` specified to `CostellaBaseInitialize()`. The macro only needs to be told the pointer into which the memory should be allocated, and how many such objects are required. Its macro prototype is

```
#define COSTELLA_MALLOC( lpo, lmc ) \
```

`lpo`: Pointer to the allocated memory, once allocated.

`lmc`: Number of objects to allocate.

*Evaluates to*: Zero, if allocation succeeded; nonzero, if allocation failed.

The pointer `lpo` is set to zero if allocation failed. The general paradigm for allocating memory for a single object is therefore

```
if( COSTELLA_MALLOC( pwidget, 1 ) )
{
    COSTELLA_FUNDAMENTAL_ERROR( "Allocating widget" );
    COSTELLA_RETURN;
}
```

and the general paradigm for allocating an array of objects is

```
if( COSTELLA_MALLOC( awidget, mcNumWidgets ) )
{
    COSTELLA_FUNDAMENTAL_ERROR( "Allocating array of widgets" );
    COSTELLA_RETURN;
}
```

The non-public-interface functions in the `costella` libraries always signify that a pointer is available for use by setting it to zero, so that a non-null pointer signifies that it is already used. Hence, this macro first checks whether `lpo` is non-null. If so, the macro evaluates to `COSTELLA_TRUE`, *i.e.*, nonzero, which indicates failure.

Otherwise, the macro then calls the memory allocation function that was specified to `CostellaBaseInitialize()`, and assigns the result (*i.e.*, the pointer to the allocated memory) to `lpo`. If the allocation failed, the pointer will be null. Since `COSTELLA_MALLOC()` must itself evaluate to a *nonzero* value in the case of failure, the macro evaluates to the Boolean complement of `lpo`.

The argument passed to the memory allocation function is the number of bytes required, which is `lmc` times the size of each of the object(s) to be allocated. The macro first checks that `lmc` is nonzero; if not, it uses 1 instead of 0, as attempting to allocate zero bytes fails on some operating systems. The macro then multiplies this value by the size of the object type, obtained by applying `sizeof` to the result of dereferencing `lpo` itself.

Note that, being a macro, it is paradigmatically correct that `COSTELLA_MALLOC()` is able to assign the allocated memory to `lpo` directly, *i.e.*, it is *not* necessary to pass the *address* of `lpo` to the macro (and, indeed, doing so will result in erroneous behavior).

#### `COSTELLA_REALLOC()`

This macro reallocates memory previously allocated using `COSTELLA_MALLOC()`, or memory reallocated using the macro `COSTELLA_REALLOC()` itself. It uses the function `pfRealloc()` specified to `CostellaBaseInitialize()`. Its macro prototype is

```
#define COSTELLA_REALLOC( lpo, lmc ) \
```

`lpo`: Pointer to the previously allocated memory, which will be reallocated.

`lmc`: Number of new objects to allocate.

*Evaluates to*: Zero, if reallocation succeeded; nonzero, if reallocation failed.

The pointer `lpo` is set to zero if reallocation failed. The general paradigm for reallocating an array of objects is therefore

```
if( COSTELLA_REALLOC( awidget, mcNewNumWidgets ) )
{
    COSTELLA_FUNDAMENTAL_ERROR( "Reallocating array of widgets" );
    COSTELLA_RETURN;
}
```

If the pointer `lpo` passed to the macro is null, then the pointer is left null, and the macro evaluates to `COSTELLA_TRUE`, *i.e.*, nonzero, which indicates failure; the pointer `lpo` must be a pointer previously allocated with `COSTELLA_MALLOC()` (or `COSTELLA_REALLOC()` itself).

The macro computes the number of bytes required as per `COSTELLA_MALLOC()`, and calls the function `pfRealloc()` specified to `CostellaBaseInitialize()`.

#### `COSTELLA_FREE()`

This macro frees memory allocated by `COSTELLA_MALLOC()` or `COSTELLA_REALLOC()`, using the function `pfFree()` specified to `CostellaBaseInitialize()`. Its macro prototype is

```
#define COSTELLA_FREE( lpo ) \
```

`lpo`: Pointer to the previously allocated memory, which will be freed.

*Evaluates to*: Zero, if freeing succeeded; nonzero, if freeing failed.

If the pointer `lpo` passed to the macro is null, nothing happens. In other words, it is harmless to issue `COSTELLA_FREE()` on a pointer that is already free (or never used in the first place, and still initialized to zero).

If the macro succeeds, the pointer is set back to zero (indicating that it can now be re-used); if it fails, the pointer is left with its previous (nonzero) value, to flag the fact that an error occurred.

The macro calls the function `pfFree()` specified to `CostellaBaseInitialize()` for freeing memory.

The general paradigm for freeing memory is therefore

```

if( COSTELLA_FREE( pwidget ) )
{
    COSTELLA_FUNDAMENTAL_ERROR( "Freeing widget" );
    COSTELLA_RETURN;
}

```

#### COSTELLA\_MALLOC\_OFFSET()

This macro allocates memory for an array with a nonzero offset. The calling function specifies the minimum and maximum offset desired. (The idea for this functionality comes from *Numerical Recipes in C*.) Its macro prototype is

```
#define COSTELLA_MALLOC_OFFSET( lpo, lmiMin, lmiMax ) \
```

**lpo**: Offset pointer to the allocated memory, once allocated.

**lmiMin**: Minimum offset required.

**lmiMax**: Maximum offset required.

*Evaluates to*: Zero, if allocation succeeded; nonzero, if allocation failed.

The general paradigm for allocating an offset array is therefore

```

if( COSTELLA_MALLOC_OFFSET( awidget, miMinOffset, miMaxOffset ) )
{
    COSTELLA_FUNDAMENTAL_ERROR( "Allocating offset array of widgets" );
    COSTELLA_RETURN;
}

```

The macro first checks whether the pointer **lpo** passed to it is not null. If so, then the pointer is already in use, and the macro fails; it evaluates to `COSTELLA_TRUE`, *i.e.*, a nonzero value. The macro also checks at this time that the minimum offset **lmiMin** is not greater than the maximum offset **lmiMax**; if it is, then it likewise fails.

If neither of these error conditions are true, the macro calls the function `pfMalloc()` specified to `CostellaBaseInitialize()` to allocate the required amount of memory, and initially assigns the result of this allocation to **lpo** itself. It then checks whether this pointer is nonzero, *i.e.*, successful. If so, it then offsets **lpo** by the minimum offset, as described in *Numerical Recipes in C*, and then evaluates to `COSTELLA_FALSE`, *i.e.*, zero, indicating no error. If not, it leaves **lpo** null, and evaluates to `COSTELLA_TRUE`, *i.e.*, nonzero, indicating failure.

The argument to the memory allocation function is the number of bytes required. This is the number of objects required times the number of bytes of each object. The number of objects required is one more than the difference between the minimum and maximum offsets, which cannot be zero. The macro multiplies this by the size of each object in bytes.



The general paradigm for freeing an offset array is therefore

```
if( COSTELLA_FREE_OFFSET( awidget, miMin ) )
{
    COSTELLA_FUNDAMENTAL_ERROR( "Freeing offset array of widgets" );
    COSTELLA_RETURN;
}
```

The macro first checks whether the pointer `lpo` is null. If so, then nothing needs to be done; the macro evaluates to `COSTELLA_FALSE`, *i.e.*, zero, indicating success.

Otherwise, the macro un-offsets `lpo` by `lmiMin` and calls the memory freeing function `pfFree()` specified to `CostellaBaseInitialize()`. If this function returns a nonzero value, indicating success, then `lpo` is zeroed, signifying that it is now available for re-use, and the macro likewise evaluates to zero, indicating success. If the memory freeing function returns a zero value, on the other hand, signifying failure, then `lpo` is left at its nonzero value, indicating that it has not been successfully freed, and the macro evaluates to this nonzero value, again indicating failure.

#### COSTELLA\_FUNCTION()

This macro is used to declare or define a function that fully follows the `costella` library error-handling conventions. The syntax is the same for both uses. Its macro prototype is

```
#define COSTELLA_FUNCTION( lfFunctionName, larglist ) \
```

`lfFunctionName`: Name of the function.

`larglist`: Argument list for the function, contained within parentheses.

In `costella_base.h`, `COSTELLA_FUNCTION` is defined for the purpose of declaring function prototypes. The macro simply declares the function as returning a pointer to a `COSTELLA_ERROR_NODE` (a type that is actually hidden from the programmer, in most cases), with the specified name and argument list, and automatically terminates the declaration with a semicolon:

```
#define COSTELLA_FUNCTION( lfFunctionName, larglist ) \
    COSTELLA_ERROR_NODE* lfFunctionName larglist;
```

Once `costella_body.h` is included, `COSTELLA_FUNCTION` is redefined for the purpose of defining the function bodies themselves. The function is defined as per its prototype, but then several additional elements (hidden from the programmer) are also defined:

- `i_l_acCostellaFunctionName[]`, which is a static character string that contains the name of the function itself, which will be used in error-handling.

- `i_1_enCostella`, which is a static error node structure that is used to temporarily store error information until the function returns. Its address is only returned to the calling function in an “emergency” situation, namely, if a copy error node cannot be dynamically allocated at the time of an error being trapped.
- `i_1_bCostellaError`, an automatic variable initialized to false on each call to the function in question, which flags whether an error has occurred.

The variable name prefix (“i\_1\_”, standing for “internal local”) has been chosen so as to make naming clashes with these hidden variables essentially impossible.

This `costella_body.h` version of the macro (used for the actual function bodies) calls `COSTELLA_FUNCTION_COMMON()`, which initializes the members of `i_1_enCostella` to zero, except for the function name string pointer, which is set to point to the statically defined function name string. To allow the function itself to define its own variables, a further set of scoping braces is then opened.

#### `COSTELLA_FUNCTION_COMMON()`

This macro, only used in `costella_body.h`, is called by both `COSTELLA_FUNCTION()` and `COSTELLA_ANSI_FUNCTION()` to perform the actions that are common to both macros. Its macro prototype is

```
#define COSTELLA_FUNCTION_COMMON( lfFunctionName ) \
```

`lfFunctionName`: Name of the function.

The macro defines the “hidden internal” variables and initializes them, as described above.

#### `COSTELLA_END_FUNCTION`

This macro is used to end a `COSTELLA_FUNCTION`. Its macro prototype is simply

```
#define COSTELLA_END_FUNCTION \
```

It is necessary to use this macro to end a `COSTELLA_FUNCTION` for two reasons:

1. The macro `COSTELLA_FUNCTION` opens two sets of extra, hidden scoping braces, which need to be closed.
2. The macro needs to clean up the error-handling facilities and return the appropriate (null) `COSTELLA_ERROR_NODE` pointer if execution of the function runs through to its end.

It is assumed that, if the function executes through to the `COSTELLA_END_FUNCTION` macro, then the programmer does not wish to return any error signal, even if an error occurred in the function’s execution. For example, it may be that an “error” may be simply hitting the end of a file, which the programmer may have used to determine when to stop reading the file.

The macro `COSTELLA_END_FUNCTION` therefore deletes any daughter error nodes that may be present in the static error node structure, and returns a null pointer.



```
#define COSTELLA_END_ANSI_FUNCTION_NO_RETURN_VALUE \
```

After deleting any daughter error trees, the macro simply closes the two extra, hidden scoping braces.

#### COSTELLA\_FUNCTION\_POINTER()

This macro allows the declaration or definition of a pointer to a function defined with `COSTELLA_FUNCTION`. Its macro prototype is

```
#define COSTELLA_FUNCTION_POINTER( lpf, larglist ) \
```

`lpf`: Name of the function pointer.

`larglist`: Argument list for the function, contained within parentheses.

The macro automatically sets the return type to `COSTELLA_ERROR_NODE*`, and then declares or defines the pointer with the given argument list:

```
#define COSTELLA_FUNCTION_POINTER( lpf, larglist ) \
    COSTELLA_ERROR_NODE* (*lpf) larglist
```

#### COSTELLA\_RETURN

This macro allows a `COSTELLA_FUNCTION` to return at any point in its execution. Its macro prototype is simply

```
#define COSTELLA_RETURN \
```

The macro first checks whether the hidden variable `i_1_bCostellaError` has been set, indicating not only that an error has occurred during the function's execution, but moreover that the programmer has flagged this error by issuing a `COSTELLA_ERROR()` command. If this flag is not set, the macro deletes any daughter error trees that may have been returned from `COSTELLA_CALL()`s (but not acted on by the programmer in the form of a `COSTELLA_ERROR()` call) by calling `COSTELLA_DELETE_DAUGHTERS`, and then returns a null pointer, signifying that no error has occurred.

If the error flag has been set, on the other hand, the macro attempts to allocate an error node structure, which will be added to the top of the error tree and returned to the calling function. If this fails, the two daughter error nodes are deleted recursively, the flag `bAllocated` in the static error node structure is set to false, and the macro instead returns the address of the static error node. This "failsafe" provision allows the error-detection functionality to operate even when memory allocation operations fail, as the static error node has already been allocated at the start of program execution. However, because the result is no longer re-entrant, it is necessary to delete any daughter trees that have already been allocated. (This "failsafe" operation can be detected in the calling function because the flag `bAllocated` will be false, whereas it should normally be true.)

If the allocation succeeds, on the other hand, then the contents of the static error node structure are copied across to the newly allocated error node. The flag `bAllocated` in the allocated error node is set to true, and its address is returned to the calling function.

#### `COSTELLA_ANSI_RETURN()`

This macro is used to return from a `COSTELLA_ANSI_FUNCTION` that has a non-void return value. Its macro prototype is

```
#define COSTELLA_ANSI_RETURN( returnvalue ) \
```

`returnvalue`: Value to return.

The macro deletes any daughter error trees that may be present, by calling the macro `COSTELLA_DELETE_DAUGHTERS`, and then issues a `return` command with the specified return value.

#### `COSTELLA_ANSI_RETURN_NO_VALUE`

This macro is used to return from a `COSTELLA_ANSI_FUNCTION` which has no return value (*i.e.*, has been defined with a return type of `void`). Its macro prototype is simply

```
#define COSTELLA_RETURN \
```

The macro deletes any daughter error trees that may be present, by calling the macro `COSTELLA_DELETE_DAUGHTERS`, and then issues a `return` command.

#### `COSTELLA_FUNDAMENTAL_ERROR()`

This macro is called whenever an error is trapped in a `COSTELLA_FUNCTION` that has not been passed back from a `COSTELLA_CALL()` to another `COSTELLA_FUNCTION`, *i.e.*, “the buck stops here”. Its macro prototype is

```
#define COSTELLA_FUNDAMENTAL_ERROR( lacError ) \
```

`lacError`: Error description string. Note that this *must* be a statically defined string, *i.e.*, hard-coded into the application; it may *not* be constructed at run-time.

The macro deletes any daughter error trees that may be present, by calling the macro `COSTELLA_DELETE_DAUGHTERS`, and then calls `COSTELLA_ERROR()` to register the error.

#### `COSTELLA_CALL()`

This macro must be used to call any `COSTELLA_FUNCTION`. (If a `COSTELLA_FUNCTION` is called without using either `COSTELLA_CALL()` or `COSTELLA_CLEANUP_CALL()`, then the program will function correctly, but memory leakage will occur if an error occurs in the improperly-called function.) Its macro prototype is

```
#define COSTELLA_CALL( lcall ) \
```

`lcall`: The function call.

The macro first deletes any daughter error trees that may still be present in the static error node in the calling function. (Issuing a `COSTELLA_CALL()` signifies that any previously encountered errors should be disregarded and discarded.)

The macro then performs the specified function call, storing its return value in the next-error node pointer of the static error node structure. If no error occurred, then this will be null.

The macro evaluates to this stored pointer, *i.e.*, it evaluates to the failure flag. However, the function issuing the `COSTELLA_CALL()` need not register an error; if it does not, then the daughter error tree is deleted when the function hits the end of the function or when a `COSTELLA_RETURN` is issued.

#### `COSTELLA_ERROR()`

The programmer may call this macro to register an error following the return of an error from `COSTELLA_CALL()`. Its macro prototype is

```
#define COSTELLA_ERROR( lacError ) \
```

`lacError`: Error description string. Note that this *must* be a statically defined string, *i.e.*, hard-coded into the application; it may *not* be constructed at run-time.

It is erroneous for `COSTELLA_ERROR()` to be called if `COSTELLA_CALL()` has not returned an error.

The macro sets the error flag in the calling function to true, and sets the error string pointer in the static error node in the function to the address of the static string passed to the macro.

#### `COSTELLA_CLEANUP_FUNDAMENTAL_ERROR()`

This macro is called if an error is trapped in a `COSTELLA_FUNCTION`, that has not been passed back from a `COSTELLA_CLEANUP_CALL()` to another `COSTELLA_FUNCTION`, during the “cleanup” phase of processing another error. Its macro prototype is

```
#define COSTELLA_CLEANUP_FUNDAMENTAL_ERROR( lacError ) \
```

`lacError`: Error description string. Note that this *must* be a statically defined string, *i.e.*, hard-coded into the application; it may *not* be constructed at run-time.

The macro deletes any daughter cleanup error tree that may be present in the static error node, and then calls `COSTELLA_CLEANUP_ERROR()` to register the error.

## COSTELLA\_CLEANUP\_CALL()

This macro is used to call any `COSTELLA_FUNCTION` required during the “cleanup” of another detected error. (If a `COSTELLA_FUNCTION` is called without using either `COSTELLA_CALL()` or `COSTELLA_CLEANUP_CALL()`, then the program will function correctly, but memory leakage will occur if an error occurs in the improperly-called function.) Its macro prototype is

```
#define COSTELLA_CLEANUP_CALL( lcall ) \
```

`lcall`: The function call.

The macro first deletes any cleanup daughter error tree in the static error node of the calling function, if present. It then calls the function in question, storing its return value in the cleanup error node pointer of the static error node structure in the calling function. If no error occurred, then this will be null.

The macro evaluates to this stored pointer, *i.e.*, it evaluates to the failure flag.

## COSTELLA\_CLEANUP\_ERROR()

The programmer may call this macro to register an error following the return of an error from `COSTELLA_CLEANUP_CALL()`. Its macro prototype is

```
#define COSTELLA_CLEANUP_ERROR( lacError ) \
```

`lacError`: Error description string. Note that this *must* be a statically defined string, *i.e.*, hard-coded into the application; it may *not* be constructed at run-time.

It is erroneous to call `COSTELLA_CLEANUP_ERROR()` if `COSTELLA_CLEANUP_CALL()` has not returned an error.

The macro sets the error flag in the calling function to true, and sets the cleanup error string pointer in the static error node of the calling function to the address of the static string passed to the macro.

## COSTELLA\_DELETE\_DAUGHTERS

This macro is used internally, in several of the above macros, to delete any daughter error trees that may be present in the static error node in the function in question. Its macro prototype is simply

```
#define COSTELLA_DELETE_DAUGHTERS \
```

For performance reasons, the macro explicitly checks whether the daughter error node pointer is null before calling `costella_base_error_node_delete()`. This saves the overhead of a function call in the case that no errors have occurred.

The macro then calls `COSTELLA_DELETE_CLEANUP_DAUGHTER` to do likewise for the cleanup daughter error node.

## COSTELLA\_DELETE\_CLEANUP\_DAUGHTER

This macro is used internally, in several of the above macros, to delete the cleanup daughter error tree, if it is present in the static error node in the function in question. Its macro prototype is simply

```
#define COSTELLA_DELETE_CLEANUP_DAUGHTERS \
```

Again, for performance reasons, the macro explicitly checks whether the cleanup daughter node pointer is null before calling `costella_base_error_node_delete()`. This saves the overhead of a function call in the case that no errors have occurred.

## COSTELLA\_IGNORE\_ERRORS

This macro tells the `costella` libraries to ignore any error (and cleanup error, if applicable) that has been trapped. Its macro prototype is simply

```
#define COSTELLA_IGNORE_ERRORS \
```

The macro is equivalent to `COSTELLA_DELETE_DAUGHTERS`.

## COSTELLA\_IGNORE\_CLEANUP\_ERROR

This macro tells the `costella` libraries to ignore any cleanup error that has been trapped. Its macro prototype is simply

```
#define COSTELLA_IGNORE_CLEANUP_ERROR \
```

The macro is equivalent to `COSTELLA_DELETE_CLEANUP_DAUGHTER`.

## COSTELLA\_ERROR\_ROOT\_NODE

This definition provides a stable alias for the name of the root node of the error tree within any `COSTELLA_FUNCTION`. Its definition is simply

```
#define COSTELLA_ERROR_ROOT_NODE \
    (i_l_enCostella.penNext)
```

## COSTELLA\_ERROR\_FPRINT()

This macro allows a calling program to print the error tree to any file or stream (including `stdout` or `stderr`). Its macro prototype is

```
#define COSTELLA_ERROR_FPRINT( pfile ) \
```

`pfile`: Pointer to the previously-opened ANSI FILE stream to write the error tree to.

The macro calls `costella_base_error_fprint()` with the file pointer, the address of the root error node, and null string pointers for the application name and application error description.

## COSTELLA\_ERROR\_APP\_FPRINT()

This macro allows an application to print the error tree to any file or stream (including `stdout` or `stderr`), including a header that provides the name of the application together with an application-level description of the error. Its macro prototype is

```
#define COSTELLA_ERROR_APP_FPRINT( pfile, lacApplicationName, \  
    lacApplicationError ) \  
    \
```

`pfile`: Pointer to the previously-opened ANSI FILE stream to write the error tree to.

`lacApplicationName`: Name of the application. This *may* be dynamically constructed at run-time.

`lacApplicationError`: Application-level error description string. This *may* be dynamically constructed at run-time.

The macro calls `costella_base_error_fprint()` with the file pointer, the address of the root error node, and the application name and application error description string pointers.

## 2.3. The `costella_wrap` library

This library provides “wrappings” around public-interface callback functions so that they can be passed to functions following the paradigm of the `costella` libraries.

### Dependencies

The `costella_wrap` library depends on the following library:

`costella_base`

### Public interface functions

There is no public interface to the `costella_wrap` library; it is solely used internally by those `costella` libraries that make use of callback functions.

### Public interface macros

There is no public interface to the `costella_wrap` library.

### Non-public-interface functions

The non-public-interface functions in the `costella_wrap` library are as follows:

#### `CostellaWrapInitialize()`

This function must be called before using any functionality of the library. Its function prototype is

```
COSTELLA_FUNCTION( CostellaWrapInitialize, ( void ) )
```

The function sets the initialization flag for the library, initializes the `costella_base` library.

```
CostellaWrapFinalize()
```

This function may be called when no further use is required of the library. Its function prototype is

```
COSTELLA_FUNCTION( CostellaWrapFinalize, ( void ) )
```

The function clears the initialization flag, and finalizes the `costella_base` library.

```
CostellaWrapCallbackNew()
```

This function creates a new `COSTELLA_WRAP_CALLBACK` object, which is used to wrap a generic callback function (one that passed not only a passback pointer to the callback function, but also a generic object pointer). Its function prototype is

```
COSTELLA_FUNCTION( CostellaWrapCallbackNew, ( COSTELLA_WRAP_CALLBACK**  
    ppwc, int (*pfCallback)( void* pvPassback, void* pv ), void*  
    pvPassback ) )
```

`ppwc`: Pointer to receiving pointer for the new `COSTELLA_WRAP_CALLBACK` object.

`pfCallback`: Pointer to the public-interface callback function to be wrapped.

`pvPassback`: Passback pointer for the wrapped public-interface callback function.

The function creates the new `COSTELLA_WRAP_CALLBACK` object, stores its pointer in the specified location, and stores the public-interface callback function pointer and passback pointer.

```
CostellaWrapCallbackDelete()
```

This function deletes a `COSTELLA_WRAP_CALLBACK` object. Its function prototype is

```
COSTELLA_FUNCTION( CostellaWrapCallbackDelete, (  
    COSTELLA_WRAP_CALLBACK** ppwc ) )
```

`ppwc`: Pointer to pointer to the `COSTELLA_WRAP_CALLBACK` object to delete.

The function frees the object.

```
CostellaWrapCallback()
```

This function is used when calling a wrapped public-interface callback function. Its function prototype is

```
COSTELLA_FUNCTION( CostellaWrapCallback, ( COSTELLA_0* poPassback,
    COSTELLA_0* po ) )
```

`poPassback`: Pointer to the `costella` library passback pointer, which is actually a pointer to a `COSTELLA_WRAP_CALLBACK` object.

`po`: Generic object pointer to be passed to the callback function.

The function first casts `poPassback` to be the pointer to the `COSTELLA_WRAP_CALLBACK` object, and then extracts the public-interface callback function pointer and passback pointer.

If the public-interface callback function pointer is not null, the function then calls the function, passing it the passback pointer and generic object pointer. If the public-interface callback function signals an error (by returning a zero value), the function wraps this error condition by issuing a `COSTELLA_ERROR()` with a generic error string:

```
COSTELLA_ERROR( "Abort signaled by ANSI callback function; no further "  
    "information available through this channel" );  
COSTELLA_RETURN;
```

## Non-public-interface macros

There are no non-public-interface macros in the `costella_wrap` library.

### 2.4. The `costella_image` library

The `costella_image` library has been designed with the following aims:

- to provide a standardized interface between the `costella` libraries and images with a bit-depth of eight bits per channel per pixel, whether grayscale or color, with or without an alpha channel;
- to allow image pixel access that is identically as fast as direct pointer access, through the use of suitably defined types and macros; and
- to allow the variety of “standard” image structures commonly in use to be accessed “in place”, without the need for them to be translated to and from a specialized format.

This means that, depending on how non-standard the calling application’s image structure is, it may or may not be possible for an application programmer to “plug in” directly into this library. In the rare occurrence that it is not possible to “plug in” directly, the calling application can still use the image processing functionality of the `costella` libraries by simply translating the image data to a more standard format, and translating back at the end of the process.

Before using the `costella_image` library, the programmer of a calling application must first ensure that the appropriate “image structure header file” (henceforth “header file”) is included from `costella_image.h` for the pixel storage structure applicable to the calling application. At present, three “standard” header files have been constructed, selected by defining the corresponding preprocessor symbol on compilation:

`costella_image_separate_arrays.h` For images in which each of the channels is stored in its own array, with each row of a given channel stored as contiguous bytes. Selected if `COSTELLA_IMAGE_SEPARATE_ARRAYS` is defined.

`costella_image_rgb_triple.h` For images in which each row of color data is stored as RGB triples of bytes, in the order BGRBGR ..., and each row of grayscale data is stored as contiguous bytes; an alpha channel, if present, is also stored in rows of contiguous bytes. Selected if `COSTELLA_IMAGE_RGB_TRIPLE` is defined.

`costella_image_win32_dib.h` For 32-bit Windows Device Independent Bitmap (DIB) images, stored in `COLORREF` structures; an alpha channel, if present, is stored in the most significant byte of each `COLORREF` structure. Selected if `COSTELLA_IMAGE_WIN32_DIB` is defined.

Almost any image structure can be accommodated by constructing an appropriate header file (or modifying an existing one), using the description that follows. (If it is not clear how to do so, please contact the author for assistance.)

If none of the above preprocessor symbols are defined, then `costella_image.h` includes the file `costella_image_default.h`, which should in turn include the desired “default” header file. (As supplied in the standard distribution, `costella_image_default.h` includes `costella_image_separate_arrays.h`.)

The libraries that make use of the `costella_image` library act on six fundamental types of object:

`COSTELLA_IMAGE_GRAY` A grayscale image.

`COSTELLA_IMAGE_ALPHA` An alpha channel image.

`COSTELLA_IMAGE_COLOR` A color image.

`COSTELLA_IMAGE_GRAY_PIXEL` A pixel in a grayscale image.

`COSTELLA_IMAGE_ALPHA_PIXEL` A pixel in an alpha channel image.

`COSTELLA_IMAGE_COLOR_PIXEL` A pixel in a color image.

The header file specifies what these objects actually are. A `COSTELLA_IMAGE_XX` will typically be a pointer to the start of an array, and a `COSTELLA_IMAGE_XX_PIXEL` will typically be a pointer to a pixel, a pointer to a pixel structure, or a structure of pointers to pixels. (Here, and in the following, “XX” stands for “GRAY”, “ALPHA”, or “COLOR” as appropriate.)

The `costella_image` library essentially only assumes that your image data is stored in some regular array of rows. Specifically, it makes the following assumptions:

- Color images may store either RGB or YCbCr data, with R and Y stored in the same channel, G and Cb stored in the same channel, and B and Cr stored in the same channel. (If these equivalences are not valid for a given calling application, then it may still be possible to use the functionality of the `costella` libraries without modification, provided that certain restrictions are obeyed; or, alternatively, it may be possible to write a custom header file that “tricks” the `costella_image` library into behaving correctly. Please contact the author for more details.)

- The header file must be able to tell if two `COSTELLA_IMAGE_XXs` are actually one and the same `COSTELLA_IMAGE_XX` using nothing more than the `COSTELLA_IMAGE_XX` structures themselves. (This is typically achieved by testing whether `pointer == other pointer`.)
- The header file must be able to set the position of a `COSTELLA_IMAGE_XX_PIXEL` to be the same as the position of another given `COSTELLA_IMAGE_XX_PIXEL` without being given anything other than the two `COSTELLA_IMAGE_XX_PIXELs`. (This is typically achieved by `pointer(s) = other pointer(s)`.)
- The header file must be able to move a `COSTELLA_IMAGE_XX_PIXEL` right or left one column without being given anything other than the `COSTELLA_IMAGE_XX_PIXEL` itself. (This is typically achieved by `pointer(s)++` and `pointer(s)--`, or `pointer(s) += n` and `pointer(s) -= n`.)
- The header file must be able to move a `COSTELLA_IMAGE_COLOR_PIXEL` right or left two columns without being given anything other than the `COSTELLA_IMAGE_COLOR_PIXEL` itself. (This is typically achieved by `pointer(s) += 2` and `pointer(s) -= 2`, or `pointer(s) += 2 x n` and `pointer(s) -= 2 x n`.)
- There is a `COSTELLA_SD` (signed 32-bit integer) quantity, called the “row stride”, that the header file needs to be told to perform any action that involves more than one row of the image. (In other words, “intra-row” operations do not require the row stride; “inter-row” operations do.) When the calling application sets up a `COSTELLA_IMAGE`, it specifies the row stride. The functions in the `costella_image` library pass the row stride back to the header file when any macro in it wants to perform any of the following operations. (There is a separate row stride for the alpha channel.)
- The header file must be able to set a `COSTELLA_IMAGE_XX_PIXEL`’s position to the top-left position of a given `COSTELLA_IMAGE_XX` using only the `COSTELLA_IMAGE_XX`, the `COSTELLA_IMAGE_XX_PIXEL`, the image width, the image height, and the row stride. (For image origins in the top-left corner, this is typically a trivial assignment of `pointer(s)`. For other origin choices, pointer arithmetic typically achieves the result.)
- The header file must be able to move a `COSTELLA_IMAGE_XX_PIXEL` up or down one row without being given anything other than the `COSTELLA_IMAGE_XX_PIXEL` itself and the row stride. (This is typically achieved by `pointer(s) += ldRowStride` or `pointer(s) -= ldRowStride`.)
- The header file must be able move a `COSTELLA_IMAGE_COLOR_PIXEL` up or down two rows without being given anything other than the `COSTELLA_IMAGE_COLOR_PIXEL` itself and a quantity which is equal to the double of the row stride. (This is typically achieved by `pointer(s) += ldDoubleRowStride` or `pointer(s) -= ldDoubleRowStride`.)

In all cases, the calling application is responsible for creating and destroying the images themselves; the calling application then provides the `costella_image` library, through the `costella_image_xxx.h` file, an interface that allows the `costella_image` library to access the pixel data in the calling application’s images. To repeat, the `costella_image` library

*does not* create or destroy images (unlike the `costella_simage` library, for example, which does); it merely reads from and writes to images created by the calling application.

If you create your own header file, rather than using one of the supplied files, it should be named “`costella_image_xxx.h`”. You are required to perform the following tasks:

1. Specify the types described above, using `typedefs`:

`COSTELLA_IMAGE_GRAY` (Hungarian prefix `ig`): Pointer or structure that points to the data for a grayscale image. Width, height, and row stride do *not* need to be stored; they will be supplied to the header file each time that a function is called. A `COSTELLA_IMAGE_GRAY` is typically just a pointer to the start of your image data.

`COSTELLA_IMAGE_ALPHA` (Hungarian prefix `ia`): Pointer or structure that points to the data for an alpha channel image. If your alpha channel data is intermixed with grayscale or color data, then a `COSTELLA_IMAGE_ALPHA` may simply be a pointer to the byte containing the alpha value for the top-left pixel in the image; if not, then it may be the same as a `COSTELLA_IMAGE_GRAY`.

`COSTELLA_IMAGE_COLOR` (Hungarian prefix `ic`): Pointer or structure that points to the data for a color image. This will typically either be a pointer to the first color structure in an array, or else a structure of three pointers to arrays of channel data.

`COSTELLA_IMAGE_GRAY_PIXEL` (Hungarian prefix `igp`): A pixel in a grayscale image, which will generally mirror the `COSTELLA_IMAGE_GRAY` structure.

`COSTELLA_IMAGE_ALPHA_PIXEL` (Hungarian prefix `iap`): A pixel in an alpha channel image, which will generally mirror the `COSTELLA_IMAGE_ALPHA` structure.

`COSTELLA_IMAGE_COLOR_PIXEL` (Hungarian prefix `icp`): A pixel in a color image, which will generally mirror the `COSTELLA_IMAGE_COLOR` structure.

2. Define the following macros:

`COSTELLA_IMAGE_GRAY_IS_SAME()`: Determines whether `lig1` and `lig2` point to the same image. Typical code:

```
( (lig1) == (lig2) )
```

`COSTELLA_IMAGE_ALPHA_IS_SAME()`: Determines whether `lia1` and `lia2` point to the same image. Typical code:

```
( (lia1) == (lia2) )
```

`COSTELLA_IMAGE_COLOR_IS_SAME()`: Determines whether `lic1` and `lic2` point to the same image. Typical code:

```
( (lic1) == (lic2) )
```

or equivalent for each of the R/Y, G/Cb, and B/Cr channels.

**COSTELLA\_IMAGE\_GRAY\_PIXEL\_ASSIGN():** Sets the grayscale pixel `ligpLeft` to point to the image and position specified by `ligpRight`. Typical code:

```
(ligpLeft) = (ligpRight)
```

**COSTELLA\_IMAGE\_ALPHA\_PIXEL\_ASSIGN():** Sets the alpha channel pixel `liapLeft` to point to the image and position specified by `liapRight`. Typical code:

```
(liapLeft) = (liapRight)
```

**COSTELLA\_IMAGE\_COLOR\_PIXEL\_ASSIGN():** Sets the color pixel `licpLeft` to point to the image and position specified by `licpRight`. Typical code:

```
(licpLeft) = (licpRight)
```

or equivalent for the R/Y, G/Cb, and B/Cr channels separately.

**COSTELLA\_IMAGE\_GRAY\_PIXEL\_SET\_TOP\_LEFT():** Sets the grayscale image pixel `ligp` to point to the top-left pixel position in the image `lig` with specified width `ludWidth`, height `ludHeight`, and row stride `lsdRowStride`. Typical code:

```
(ligp) = (lig)
```

for top-left origin; or

```
(ligp) = (lig) + (lsdRowStride) * ( (ludHeight) - 1 )
```

for bottom-left origin; or

```
(ligp) = (lig) + (ludWidth) - 1
```

for top-right origin; or

```
(ligp) = (lig) + (lsdRowStride) * ( (ludHeight) - 1 )  
+ (ludWidth) - 1
```

for bottom-right origin.

**COSTELLA\_IMAGE\_ALPHA\_PIXEL\_SET\_TOP\_LEFT():** Set the alpha channel pixel `liap` to point to the top-left pixel position in the image `lia` with specified width `ludWidth`, height `ludHeight`, and row stride `lsdRowStride`. Typical code: as for a grayscale pixel.

**COSTELLA\_IMAGE\_COLOR\_PIXEL\_SET\_TOP\_LEFT():** Set the color pixel `licp` to point to the top-left pixel position in the image `lic` with specified width `ludWidth`, height `ludHeight`, and row stride `lsdRowStride`. Typical code: as for a grayscale pixel; or appropriate assignments for R/Y, G/Cb, and B/Cr channels separately.

**COSTELLA\_IMAGE\_GRAY\_PIXEL\_MOVE\_LEFT():** Move the grayscale pixel `ligp` left by one column. Range checking does *not* need to be performed. Typical code:

```
(ligp)--
```

**COSTELLA\_IMAGE\_ALPHA\_PIXEL\_MOVE\_LEFT():** Move the grayscale pixel `liap` left by one column. Range checking does *not* need to be performed. Typical code:

(liap)--

COSTELLA\_IMAGE\_COLOR\_PIXEL\_MOVE\_LEFT(): Move the color pixel licp left by one column. Range checking does *not* need to be performed. Typical code:

(licp)--

or R/Y, G/Cb, and B/Cb pointer each -= 3.

COSTELLA\_IMAGE\_GRAY\_PIXEL\_MOVE\_RIGHT(): Move the grayscale pixel ligp right by one column. Range checking does *not* need to be performed. Typical code:

(ligp)++

COSTELLA\_IMAGE\_ALPHA\_PIXEL\_MOVE\_RIGHT(): Move alpha channel pixel liap right by one column. Range checking does *not* need to be performed. Typical code:

(liap)++

COSTELLA\_IMAGE\_COLOR\_PIXEL\_MOVE\_RIGHT(): Move the color pixel licp right by one column. Range checking does *not* need to be performed. Typical code:

(licp)++

or R/Y, G/Cb, and B/Cr pointer each += 3.

COSTELLA\_IMAGE\_GRAY\_PIXEL\_MOVE\_UP(): Move the grayscale pixel ligp up by one row. Range checking does *not* need to be performed. Typical code:

(ligp) -= (lsdRowStride)

COSTELLA\_IMAGE\_ALPHA\_PIXEL\_MOVE\_UP(): Move the alpha channel pixel liap up by one row. Range checking does *not* need to be performed. Typical code:

(liap) -= (lsdAlphaRowStride)

COSTELLA\_IMAGE\_COLOR\_PIXEL\_MOVE\_UP(): Move the color pixel licp up by one row. Range checking does *not* need to be performed. Typical code:

(licp) -= (lsdRowStride)

or equivalent for R/Y, G/Cb, and B/Cr pointers.

COSTELLA\_IMAGE\_GRAY\_PIXEL\_MOVE\_DOWN(): Move the grayscale pixel ligp down by one row. Range checking does *not* need to be performed. Typical code:

(ligp) += (lsdRowStride)

COSTELLA\_IMAGE\_ALPHA\_PIXEL\_MOVE\_DOWN(): Move the alpha pixel liap down by one row. Range checking does *not* need to be performed. Typical code:

(liap) += (lsdAlphaRowStride)

COSTELLA\_IMAGE\_COLOR\_PIXEL\_MOVE\_DOWN(): Move the color pixel licp down by one row. Range checking does *not* need to be performed. Typical code:

(licp) += (lsdRowStride)

or equivalent for R/Y, G/Cb, and B/Cr pointers.

**COSTELLA\_IMAGE\_COLOR\_PIXEL\_MOVE\_LEFT\_TWO():** Move the color pixel `licp` left by two columns. Range checking does *not* need to be performed. Typical code:

```
(licp) -= 2
```

or equivalent for R/Y, G/Cb, and B/Cr pointers.

**COSTELLA\_IMAGE\_COLOR\_PIXEL\_MOVE\_RIGHT\_TWO():** Move the color pixel `licp` right by two columns. Range checking does *not* need to be performed. Typical code:

```
(licp) += 2
```

or equivalent for R/Y, G/Cb, and B/Cr pointers.

**COSTELLA\_IMAGE\_COLOR\_PIXEL\_MOVE\_UP\_TWO():** Move the color pixel `licp` up by two columns. Range checking does *not* need to be performed. Typical code:

```
(licp) -= lsdDoubleRowStride
```

or equivalent for R/Y, G/Cb, and B/Cr pointers.

**COSTELLA\_IMAGE\_COLOR\_PIXEL\_MOVE\_DOWN\_TWO():** Move the color pixel `licp` down by two columns. Range checking does *not* need to be performed. Typical code:

```
(licp) += lsdDoubleRowStride
```

or equivalent for R/Y, G/Cb, and B/Cr pointers.

**COSTELLA\_IMAGE\_GRAY\_PIXEL\_GET\_Y():** Extract the luminance value of the grayscale pixel `ligp`. Typical code:

```
*(ligp)
```

**COSTELLA\_IMAGE\_COLOR\_PIXEL\_GET\_R\_Y():** Extract the R/Y channel value of a color pixel `licp`. Typical code: varies.

**COSTELLA\_IMAGE\_COLOR\_PIXEL\_GET\_G\_CB():** Extract the G/Cb channel value of a color pixel `licp`. Typical code: varies.

**COSTELLA\_IMAGE\_COLOR\_PIXEL\_GET\_B\_CR():** Extract the B/Cr value of a color pixel `licp`. Typical code: varies.

**COSTELLA\_IMAGE\_GRAY\_PIXEL\_SET\_Y():** Set the luminance value of the grayscale pixel `ligp` to `ubY`. Typical code:

```
*(ligp) = (ubY)
```

**COSTELLA\_IMAGE\_COLOR\_PIXEL\_SET\_R\_Y():** Set the R/Y channel value of the color pixel `licp` to `ubRY`. Typical code: varies.

**COSTELLA\_IMAGE\_COLOR\_PIXEL\_SET\_G\_CB():** Set the G/Cb channel value of the color pixel `licp` to `ubGcb`. Typical code: varies.

**COSTELLA\_IMAGE\_COLOR\_PIXEL\_SET\_B\_CR():** Set the B/Cr channel value of a color pixel `licp` to `ubBCr`. Typical code: varies.

`COSTELLA_IMAGE_COLOR_PIXEL_SET_RGB_YCBCR()`: Set the R, G, and B (or Y, Cb, and Cr) channel intensity values of the color pixel `licp` to `ubRY`, `ubGcb`, and `ubBCr` respectively. Typical code: call each of the three functions in turn, or it may be more efficient to do in one step.

The header file `costella_image.h` defines a type `COSTELLA_IMAGE` (Hungarian prefix `i`) which houses all of the information relevant to a given image. Its fields are as follows:

`bColor` Flag indicating whether the image is color. The alternative is that it is grayscale.

`bAlpha` Flag indicating whether the image has an alpha channel attached to it.

`bRgb` Flag indicating whether a color image is in the RGB colorspace. The alternative is that it is in the YCbCr colorspace.

`bDownsampledChrominance` Flag indicating whether a color image has downsampled sampled chrominance channels, meaning that there is only a single Cb value and a single Cr value for each  $2 \times 2$  block of pixels. (The next flag specifies where this chrominance data is actually stored.) This flag should be set to true for color images obtained from a JPEG file. (Note that *no* “magic” or “fancy” upsampling should be performed on the chrominance channels of a JPEG image, before passing it to a `costella` library.) This flag can be true in either the RGB colorspace or the YCbCr colorspace.

`bNonreplicatedDownsampledChrominance` If a color image has downsampled chrominance, this flag specifies whether that chrominance data is only contained in the top-left pixel of each  $2 \times 2$  block of pixels. The alternative is that the chrominance data is replicated into each pixel of the given  $2 \times 2$  block of pixels.

`udWidth` The width of the image.

`udHeight` The height of the image.

`sdRowStride` The row stride of the image.

`sdAlphaRowStride` The row stride of the alpha channel image.

`ia` Alpha channel image object, if an alpha channel is attached.

`ig` Grayscale image object, if the image is grayscale.

`ic` Color image object, if the image is color.

A `COSTELLA_IMAGE` is a “non-allocated” object: the calling application need simply define a variable of this type, fill in the required information, and pass a pointer to it to the `costella` libraries. For example, if using the header file `costella_image_separate_arrays.h`, the calling application might set up a  $120 \times 100$  grayscale image as follows:

```

COSTELLA_IMAGE i;

i.bColor = 0;
i.bAlpha = 0;
i.udWidth = 120;
i.udHeight = 100;
i.sdRowStride = 120;
i.ig = abyImageData;

if( !costella_do_something_to_my_image( &i, stderr ) )
{
    /* Handle the error in some way.
    */

    error_exit( "Couldn't do something to image" );
}

```

where `abyImageData` is an array of 12 000 bytes that the calling application has previously allocated and filled with grayscale image data.

## Dependencies

The `costella_image` library depends on the following library:

`costella_base`

## Public interface functions

The public interface functions in the `costella_image` library are as follows:

`costella_image_initialize()`

This is the public interface function for initializing the `costella_image` library. Its function prototype is

```
int costella_image_initialize( FILE* pfileError );
```

`costella_image_finalize()`

This is the public interface function for finalizing the `costella_image` library. Its function prototype is

```
int costella_image_finalize( FILE* pfileError );
```

## Public interface macros

The public interface macros in the `costella_image` library are as follows:

### `COSTELLA_IMAGE_LIMIT_RANGE()`

This macro limits the range of an integer to `[0, 255]`. Its macro prototype is

```
#define COSTELLA_IMAGE_LIMIT_RANGE( sx ) \
```

`sx`: The value to be range-limited.

*Evaluates to:* The range-limited value.

For example,

```
byY = (unsigned char) COSTELLA_IMAGE_LIMIT_RANGE( lY );
```

## Non-public-interface functions

The non-public-interface functions in the `costella_image` library are as follows:

### `CostellaImageInitialize()`

This function must be called before using any functionality of the library. Its function prototype is

```
COSTELLA_FUNCTION( CostellaImageInitialize, ( void ) )
```

The initialization flag for this library is set, and the `costella_base` library is initialized.

### `CostellaImageFinalize()`

This function may be called when the calling application no longer requires the functionality of the library. Its function prototype is

```
COSTELLA_FUNCTION( CostellaImageFinalize, ( void ) )
```

The initialization flag for this library is cleared, and the `costella_base` library is finalized.

## Non-public-interface macros

There are no non-public-interface macros in the `costella_image` library.

## 2.5. The `costella_image_chrominance` library

The `costella_image_chrominance` library provides several functions that allow the chrominance channels of a color `COSTELLA_IMAGE` in the YCbCr colorspace to be downsampled, upsampled, and replicated, in the way that is done for images compressed using the JPEG standard.

## Dependencies

The `costella_image_chrominance` library depends on the following libraries:

```
costella_image
costella_wrap
costella_base
```

## Public interface functions

The public interface functions in the `costella_image_convert_chrominance` library are as follows:

```
costella_image_chrominance_initialize()
```

This is the public interface function for initializing the `costella_image_chrominance` library. Its function prototype is

```
int costella_image_chrominance_initialize( FILE* pfileError );
```

```
costella_image_chrominance_finalize()
```

This is the public interface function for finalizing the `costella_image_chrominance` library. Its function prototype is

```
int costella_image_chrominance_finalize( FILE* pfileError );
```

```
costella_image_chrominance_average_downsample_replicate()
```

This public interface function downsamples the chrominance channels of a color YCbCr `COSTELLA_IMAGE`, following the methods generally employed for JPEG images (namely, simple averaging of each  $2 \times 2$  block), and replicates that chrominance data into each pixel of the  $2 \times 2$  block. Its function prototype is

```
int costella_image_chrominance_average_downsample_replicate(
    COSTELLA_IMAGE* piIn, COSTELLA_IMAGE* piOut, int (*pfProgress)(
    void* pvPassback ), void* pvPassback, FILE* pfileError );
```

`piIn`: Pointer to the input `COSTELLA_IMAGE`. Must be in the YCbCr colorspace, and must not have downsampled chrominance.

`piOut`: Pointer to the output `COSTELLA_IMAGE`. This may be the same as `piIn`. It will be set to the YCbCr colorspace and will be flagged as having replicated downsampled chrominance.

```
costella_image_chrominance_magic_upsample()
```

This public interface function upsamples the chrominance channels of a color YCbCr `COSTELLA_IMAGE` using the “magic” kernel. Its function prototype is

```
int costella_image_chrominance_magic_upsample( COSTELLA_IMAGE* piIn,
        COSTELLA_IMAGE* piOut, int (*pfProgress)( void* pvPassback ), void*
        pvPassback, FILE* pfileError );
```

piIn: Pointer to the input COSTELLA\_IMAGE. Must be in the YCbCr colorspace, and must have downsampled chrominance (either replicated or nonreplicated).

piOut: Pointer to the output COSTELLA\_IMAGE. This may be the same as piIn. It will be set to the YCbCr colorspace, and will be flagged as not having downsampled chrominance.

```
costella_image_chrominance_replicate_upsample_eq()
```

This public interface function upsamples the chrominance channels of a color YCbCr COSTELLA\_IMAGE by replicating the top-left pixel of each 2×2 block into the other positions of each such block, in place. This function is of use if the chrominance data has only been stored in these top-left positions, and if one wishes to use other JPEG libraries (such as the IJG JPEG library) which averages the chrominance channel over each block. Its function prototype is

```
int costella_image_chrominance_replicate_upsample_eq( COSTELLA_IMAGE*
        pi, int (*pfProgress)( void* pvPassback ), void* pvPassback, FILE*
        pfileError );
```

pi: Pointer to the COSTELLA\_IMAGE. This must be in the YCbCr colorspace, and must have nonreplicated downsampled chrominance. On exit from this function, it will still be in the YCbCr colorspace, and will have replicated downsampled chrominance.

## Public interface macros

There are no public interface macros in the `costella_image_chrominance` library.

## Non-public-interface functions

The non-public-interface functions in the `costella_image_chrominance` library are as follows:

```
CostellaImageChrominanceInitialize()
```

This function must be called before using any functionality of the library. Its function prototype is

```
COSTELLA_FUNCTION( CostellaImageChrominanceInitialize, ( void ) )
```

The initialization flag is set, and the `costella_image` and `costella_wrap` libraries are initialized.

### CostellaImageChrominanceFinalize()

This function may be called when the functionality of the library is no longer required. Its function prototype is

```
COSTELLA_FUNCTION( CostellaImageChrominanceFinalize, ( void ) )
```

The initialization flag is cleared, and the `costella_wrap` and `costella_image` libraries are finalized.

### CostellaImageChrominanceAverageDownsampleReplicate()

This function downsamples the chrominance channels of a YCbCr `COSTELLA_IMAGE` by simple averaging, and replicates this downsampled data into each pixel of each  $2 \times 2$  block. If an alpha channel is attached to the image, and the output image is not the same as the input image, then the alpha channel values are copied over to the output image. Its function prototype is

```
COSTELLA_FUNCTION( CostellaImageChrominanceAverageDownsampleReplicate,  
    ( COSTELLA_IMAGE* piIn, COSTELLA_IMAGE* piOut,  
    COSTELLA_CALLBACK_FUNCTION pfProgress, COSTELLA_0* poPassback ) )
```

`piIn`: Pointer to the input `COSTELLA_IMAGE`. Must be in the YCbCr colorspace, and must not have downsampled chrominance.

`piOut`: Pointer to the output `COSTELLA_IMAGE`. This may be the same as `piIn`. Will be set to the YCbCr colorspace, and will be flagged as having replicated downsampled chrominance.

The function first checks that the input and output `COSTELLA_IMAGES` are color, and that the input `COSTELLA_IMAGE` is in the YCbCr colorspace and does not already have downsampled chrominance. The output `COSTELLA_IMAGE` colorspace flag is set to YCbCr, and it is flagged as having replicated downsampled chrominance.

The function then extracts the information it requires from the input and output `COSTELLA_IMAGES`. It computes the double row strides, and one less than the width and the height, as these constants will be used in the following.

The function then walks through the downsampled rows and columns of the input and output `COSTELLA_IMAGES`, checking whether it is in the last row or the last column each time; this will only occur if the height or width is odd, respectively.

The function first extracts the top-left (A) values of Y, Cb, and Cr. It then switches on whether it is in the last row, the last column, neither, or both.

If the function is in both the last row and the last column, then there is but one pixel to “average”; this is stored in the sole pixel in the bottom-right corner of the output `COSTELLA_IMAGE`. No pixel movements are necessary, because this is the last pixel to be done in the `COSTELLA_IMAGE`.

If the function is in the last row but not the last column, there are two pixels to average. The input pixel is moved to the right, and the D values extracted. The input pixel is then moved right again, ready for the next block. The chrominance values are averaged. The average chrominance values and A luminance value are stored in the left output pixel. The output pixel is then moved to the right, and the average chrominance values and D luminance value are stored in the right output pixel. The output pixel is then moved right again, ready for the next block.

If the function is in the last column but not the last row, there again are two pixels to average. The input pixel is moved down, and the B values extracted. No further movement of the input pixel is needed, because this is the last pixel in the row to be done. The chrominance values are averaged. The average chrominance values and A luminance value are stored in the top output pixel. The output pixel is then moved down, and the average chrominance values and B luminance value are stored in the bottom output pixel. Again, no further movement of the output pixel is needed.

Finally, if the function is in neither the last row nor the last column, then there is a full quota of four pixels to average. The input pixel is moved down, and the B values extracted; it is moved right, and the C values extracted; and finally it is moved up, and the D values extracted. It is then moved right again, ready for the next block. The four sets of chrominance values are averaged. The average chrominance values and A luminance value are stored in the top-left output pixel. The output pixel is then moved down, and the average chrominance values and B luminance value are stored in the bottom-left output pixel. The output pixel is moved right, and the average chrominance values and C luminance value are stored in the bottom-right output pixel. Finally, the output pixel is moved up, and the average chrominance values and D luminance value are stored in the top-right output pixel. The output pixel is then moved right, ready for the next block.

In each case, if the alpha channel needs to be copied, it is copied across.

#### `CostellaImageChrominanceMagicUpsample()`

This function upsamples the chrominance channels of a YCbCr `COSTELLA_IMAGE` using the “magic” kernel. This function has been optimized for “in-place” upsampling. If the output `COSTELLA_IMAGE` is not the same as the input `COSTELLA_IMAGE`, then the luminance channel is copied across; if there is also an alpha channel attached, then this is also copied across. Its function prototype is

```
COSTELLA_FUNCTION( CostellaImageChrominanceMagicUpsample, (
    COSTELLA_IMAGE* piIn, COSTELLA_IMAGE* piOut,
    COSTELLA_CALLBACK_FUNCTION pfProgress, COSTELLA_0* poPassback ) )
```

`piIn`: Pointer to the input `COSTELLA_IMAGE`. Must be in the YCbCr colorspace, and must have downsampled chrominance data (either replicated or nonreplicated).

`piOut`: Pointer to the output `COSTELLA_IMAGE`. This may be the same as `piIn`. Will be set to the YCbCr colorspace with nondownsampled chrominance data.

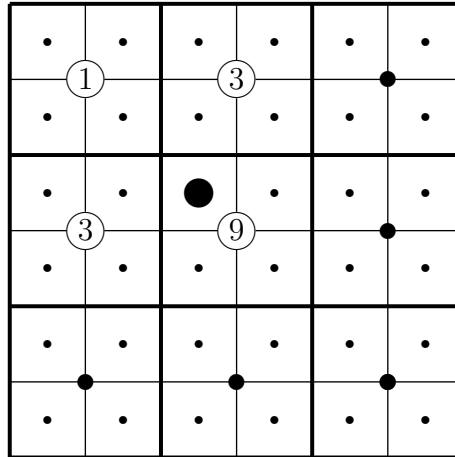


Figure 1: The destination pixel marked with the large spot receives contributions from the four source pixels that have been circled, with the weights shown. (The result is divided by 16.) This destination pixel is referred to as a “top-left” pixel in the text. Each circled “source” value is actually stored in the “destination” pixel to its upper-left.

After checking that the image formats are correct, and extracting the standard information from the `COSTELLA_IMAGES`, the function computes the values of the last row and column, namely, one less than the height and one less than the width respectively, as these constants will be used in the following.

The function next allocates two buffers, one for each of the `Cb` and `Cr` channels, each of which is able to store a complete downsampled row of the corresponding chrominance channel. The width of this buffer is therefore half of the width of the `COSTELLA_IMAGE`, rounded up.

The function now walks through the rows and columns of the `COSTELLA_IMAGE`, keeping track of whether it is in a “top” or “bottom” row, and whether it is in a “left” or a “right” column. The pixel location being computed is referred to as the “target” pixel. For each row, the function maintains three buffer pointers for each chrominance channel, pointing to the current buffer position, one position left of this, and one position right of this respectively.

The function now extracts values of `Cb` and `Cr` from the four positions of the given  $2 \times 2$  block. These four values (for each channel) are labeled `A`, `B`, `C`, and `D`. Value `A` is the nearest input value (weighted by 9 in the magic kernel); values `B` and `C` are the next-nearest input values (weighted by 3); and value `D` is the farthest input value (weighted by 1). To do so, it first switches on the position of the target pixel within each  $2 \times 2$  block.

For top-left target pixels, refer to Fig. 1. The `A` values are extracted from the target pixel itself; the `B` values, corresponding to the pixel two rows above the target pixel, are extracted from the current position of the buffer (unless we are in the top row of the image, in which case they are copied from the `A` values); the `C` values, corresponding to the pixel two columns to the left of the target pixel, are taken from “saved” buffer values from the left of our current target position (unless we are in the leftmost column,

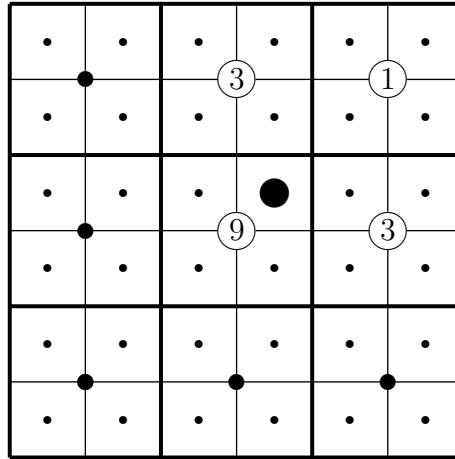


Figure 2: Source pixel contributions to a top-right destination pixel, analogous to Fig. 1.

in which case they are copied from the A values); and the D values, corresponding to the pixel two rows above and two rows to the left of the target pixel, are taken from the left buffer position. The “saved” values to the left are then stored in the left position of the buffer (unless we are in the leftmost column of the image), and the A values are stored as the new “saved” values.

For top-right target pixels, refer to Fig. 2. The A values, corresponding to the pixel to the left of the target pixel, are extracted from the “saved left” values; the B values, corresponding to the pixel two rows above and one column to the left of the target pixel, are taken from the current buffer position (unless we are in the top row, in which case they are copied from the A values); the C values, corresponding to the pixel one column to the right of the target pixel, are read from the position to the right of the target pixel (unless we are in the rightmost column, in which case they are copied from the A values); and the D values, corresponding to the pixel two rows above and one column to the right of the target pixel, are taken from the right buffer position (unless we are in the top row or the rightmost column, in which case they are copied from the A values).

For bottom-left target pixels, refer to Fig. 3. The A values, corresponding to the pixel above the target pixel, are taken from the current buffer position; the B values, corresponding to the pixel below the target pixel, are read from the pixel below the target pixel (unless we are in the last row, in which case they are copied from the A values); the C values, corresponding to the pixel one row above and two columns to the left of the target pixel, are read from the left buffer position (unless we are in the leftmost column, in which case they are copied from the A values); and the D values, corresponding to the pixel one row below and two columns to the left of the target pixel, are read from that pixel (unless we are in the bottom row or the leftmost column, in which case they are copied from the A values).

Finally, for bottom-right target pixels, refer to Fig. 4. The A values, corresponding to the pixel one row above and one column to the left of the target pixel, are taken from the current buffer position; the B values, corresponding to the pixel one row below

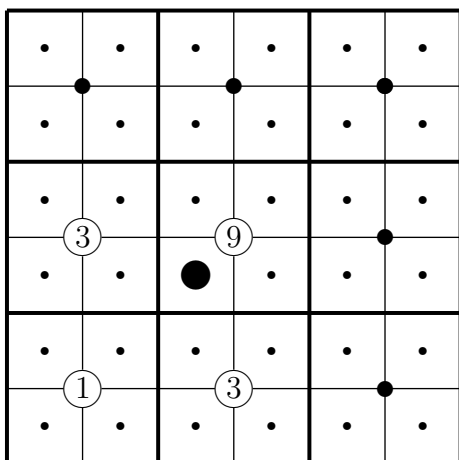


Figure 3: Source pixel contributions to a bottom-left destination pixel, analogous to Fig. 1.

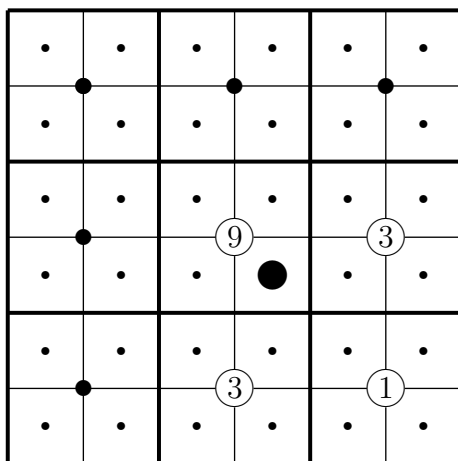


Figure 4: Source pixel contributions to a bottom-right destination pixel, analogous to Fig. 1.

and one column to the left of the target pixel, are read from the image (unless we are in the bottom row, in which case they are copied from the **A** values); the **C** values, corresponding to the pixel one row above and one column to the right of the target pixel, are taken from the right buffer position (unless we are in the rightmost column of the image, in which case they are copied from the **A** values); and the **D** values, corresponding to the pixel one row below and one column to the right of the target pixel, are read from the image (unless we are in the rightmost column or the bottom row of the image, in which case they are copied from the **A** values).

The four values are then combined according to the magic kernel, and stored in the image. If the output image is not the same as the input image, the luminance value is copied across; and if there is also an alpha channel, its value is copied across.

If we are in a “right” column, the buffer positions are all shifted one position to the right, as we are about to enter a new  $2 \times 2$  block. Regardless of the position, the output and target input pixels are then moved to the right.

At the end of the row, if we are moving from a “top” row to “bottom” row, the last buffer values are stored; if the last column of the image is missing, we first need to increment the left-buffer pointers (as these are incremented in the top-right pixel’s code block, which will not have been executed). For every row, the starting pixels are moved down.

Finally, the buffers are freed.

#### `CostellaImageChrominanceReplicateEq()`

This function replicates the chrominance values of the top-left pixel of each  $2 \times 2$  block into the other pixels in the block, in place. Its function prototype is

```
COSTELLA_FUNCTION( CostellaImageChrominanceReplicateEq, (
    COSTELLA_IMAGE* pi, COSTELLA_CALLBACK_FUNCTION pfProgress,
    COSTELLA_0* poPassback ) )
```

**pi**: Pointer to the `COSTELLA_IMAGE`. Must be in the YCbCr colorspace, and must have nonreplicated downsampled chrominance.

After checking the image format and extracting the standard information from it, the function computes the double row stride, and one less than the width and the height, as these constants will be used in the following.

The function then walks through the downsampled rows and columns of the input and output images, checking whether it is in the last row or the last column each time; this will only occur if the height or width is odd, respectively.

The function first extracts the chrominance values for the block from the top-left pixel. It then switches on whether it is in the last row, the last column, neither, or both.

If the function is in both the last row and the last column, then there is only one pixel in the block, which already has its chrominance values set; nothing more needs to be done.

If the function is in the last row but not the last column, the chrominance needs to be replicated into the pixel to the right. The pixel is then moved one more position to the right, ready for the next block.

If the function is in the last column but not the last row, the chrominance needs to be replicated into the pixel below. No further movement of the pixel is needed, as it will be reinitialized at the start of the next row.

Finally, if the function is in neither the last row nor the last column, then the chrominance values need to be set in the bottom-left, bottom-right, and top-right pixels. The pixel is then moved again to the right, ready for the next block.

## Non-public-interface macros

There are no non-public-interface macros in the `costella_image_chrominance` library.

### 2.6. The `costella_image_convert` library

The `costella_image_convert` library provides functionality for `COSTELLA_IMAGES` to be converted between the RGB and YCbCr colorspaces.

The transformation from the RGB colorspace (where  $R \in [0, 1]$ ,  $G \in [0, 1]$ , and  $B \in [0, 1]$ ) to the YCbCr colorspace (where  $Y \in [0, 1]$ ,  $C_b \in [-0.5, +0.5]$ , and  $C_r \in [-0.5, +0.5]$ ) is generally defined to be

$$\begin{aligned} Y &\equiv K_r R + (1 - K_r - K_b) G + K_b B, \\ C_b &\equiv \frac{B - Y}{2(1 - K_b)}, \\ C_r &\equiv \frac{R - Y}{2(1 - K_r)}. \end{aligned} \tag{1}$$

The ITU-R BT.601 (CCIR 601) standard, which forms the reference for JPEG and MPEG (and hence the `costella` libraries), specifies

$$\begin{aligned} K_b &= 0.114, \\ K_r &= 0.299. \end{aligned} \tag{2}$$

Inserting (2) into (1) yields, to six decimal places, the familiar standard conversion formulas

$$\begin{aligned} Y &= +0.299\,000\,R + 0.587\,000\,G + 0.114\,000\,B, \\ C_b &= -0.168\,736\,R - 0.331\,264\,G + 0.500\,000\,B, \\ C_r &= +0.500\,000\,R - 0.418\,688\,G - 0.081\,312\,B. \end{aligned} \tag{3}$$

The subtlety arises when one wishes to use quantized integer values for these six quantities. Most bitmap formats use the full range of 8-bit bytes,  $[0, 255]$  for each of  $R$ ,  $G$ , and  $B$ ; it is therefore desirable to do likewise for  $Y$ ,  $C_b$ , and  $C_r$ . The  $Y$  value is straightforward enough, because its conversion coefficients are all positive, and hence the range of  $Y$  is the same as each of the  $R$ ,  $G$ , and  $B$  channels, namely,  $[0, 255]$ : any reasonable quantization of

the conversion coefficients yields a satisfactory conversion. The `costella_image_convert` library represents them as a binary fractions with a width of 16 bits (about five decimal digits of accuracy; far more than is needed for data that is only accurate to 8 bits in the first place):

$$Y = (19\,595 R + 38\,470 G + 7471 B + 32\,768) \gg 16, \quad (4)$$

where it can be confirmed that the three coefficients sum to 65 536 (*i.e.*, unity); the addition of 32 768 ensures that the shift-right by 16 bits is correctly rounded. The use of these specific integers (and those to follow) ensures that this conversion is performed identically on all machines; this is of importance for some uses of the `costella_image_convert` library, for which the use of this particular set of integers is mandatory.

The use of the quantized integers listed in (4) means that the key constants in the luminance formula are slightly different than the standard 0.299 and 0.114 listed in (2). Specifically, we now have

$$Y = +0.298\,996 R + 0.587\,006 G + 0.113\,998 B, \quad (5)$$

Eqs.(1) can be used to adjust the remaining coefficients for the chrominance channels, to ensure the set of equations remains self-consistent:

$$\begin{aligned} C_b &= -0.168\,733 R - 0.331\,267 G + 0.500\,000 B, \\ C_r &= +0.500\,000 R - 0.418\,689 G - 0.081\,311 B, \end{aligned} \quad (6)$$

which shall be used in the following.

Now, the chrominance channels  $C_b$ , and  $C_r$  provide somewhat greater problems than the luminance channel  $Y$  in terms of general use. Any translation of the luminance channel range of  $[-0.5, +0.5]$  to a  $[0, 255]$  byte value hinges on the question of what value zero should map to. It is generally assumed that 0 should map to 128. Of course, it is impossible to maintain symmetry between positive and negative numbers if one of 256 values is mapped to zero; the remaining 255 values must be broken up into parts of 127 and 128. Making 0 map to 128 means that there are more negative values than positive; each chrominance channel then takes on the effective range  $[-128, +127]$ . This asymmetry is unobjectionable, as it is simply a by-product of the fact that we have quantized the chrominance channels to 8-bit values in the first place. However, the author has seen a bizarre alternative definition in which zero is alternately mapped to either 127 or 128, depending on the combined parity of the (downsampled) row and column in question, ostensibly to avoid this asymmetry. The author cannot see the sense of this convoluted definition.

The `costella_image_convert` library computes the  $C_b$ , and  $C_r$  values according to

$$\begin{aligned} C_b &= (-11\,058 R - 21\,709 G + 32\,767 B + 8\,421\,376) \gg 16, \\ C_r &= (32\,767 R - 27\,438 G - 5329 B + 8\,421\,376) \gg 16, \end{aligned} \quad (7)$$

It will be noted that two of the coefficients in each formula are one integer smaller in absolute magnitude (out of 65 536) than is obtained by rounding (6) to the nearest integer. They have actually been calculated by multiplying the coefficients in (6) by  $2 \times 32\,767$ , rather than by 65 536. This has been done to ensure that the positive coefficients and the negative

coefficients in each formula each separately sum to 32 767, rather than 32 768, for a reason that will become clear shortly. The additive constant 8 421 376 is equal to  $128.5 \times 65\,536$ , where the 128.5 is composed of the 128 (defined to represent zero), plus 0.5 to ensure correct rounding for values away from zero. Thus, when the  $R$ ,  $G$ , and  $B$  values are all equal (corresponding to the case of zero chrominance), each chrominance channel rounds 128.5 down, to 128. This ensures that zero chrominance maps to exactly 128, and that negative and positive values of chrominance map symmetrically to integers below and above 128 respectively.

Let us now see why the positive and negative coefficients in each formula have been adjusted to sum to 32 767 rather than 32 768. If one considers the case of  $R = G = 0$  and  $B = 255$ , one finds that  $C_b$  is given by

$$\begin{aligned} & 32\,767 \times 255 + 128.5 \times 65\,536 \\ &= (0.5 \times 65\,536 - 1) \times 255 + 128.5 \times 65\,536 \\ &= (127.5 + 128.5) \times 65\,536 - 255 \\ &= 256 \times 65\,536 - 255. \end{aligned}$$

Thus, when we shift this right by 16 bits (divide by 65 536), the final term ensures that the result will be 255, not 256. Note that, if 32 767 had instead been 32 768, we would have had

$$\begin{aligned} & 32\,768 \times 255 + 128.5 \times 65\,536 \\ &= (0.5 \times 65\,536) \times 255 + 128.5 \times 65\,536 \\ &= (127.5 + 128.5) \times 65\,536 \\ &= 256 \times 65\,536, \end{aligned}$$

which would have (just) overflowed an eight-bit byte value. Reducing 32 768 to 32 767 (an insignificant change for quantities that are only accurate to eight bits anyway) has avoided the need for an explicit range-limiting check. (The same holds true for  $C_r$ , for the case of  $G = B = 0$  and  $R = 255$ .)

The other (negative chrominance) end of the scale is less critical than the positive end, because it has an extra value to play with ( $-128$  compared to  $+127$ ). Explicitly, if we look at  $C_b$  for  $R = G = 255$  and  $B = 0$ , we have

$$\begin{aligned} & -32\,767 \times 255 + 128.5 \times 65\,536 \\ &= -(0.5 \times 65\,536 - 1) \times 255 + 128.5 \times 65\,536 \\ &= (-127.5 + 128.5) \times 65\,536 + 255 \\ &= 1 \times 65\,536 + 255. \end{aligned}$$

Thus, after shifting right by 16 bits, we will obtain the value 1. This is completely symmetrical with the above case: we are, in both cases, only  $255/65\,536$  of a unit away from rounding to the “next” value (256 or 0 respectively). Thus, the above definitions actually give us a chrominance range of  $[1, 255]$ ; the “asymmetry” has actually been done away with, as the value 0 is never used for  $C_b$  or  $C_r$ .

Again, the use of the specific quantized integers, plus the adjustment that we have made to the multiplier (from 65 536 to  $2 \times 32\,767$ ), means that the transformations (7) to  $C_b$  and

$C_r$  are not exactly those specified by (6). Explicitly, we now actually have

$$\begin{aligned} C_b &= 128 - 0.168\,732\,R - 0.331\,253\,G + 0.499\,985\,B, \\ C_r &= 128 + 0.499\,985\,R - 0.418\,671\,G - 0.081\,314\,B, \end{aligned} \quad (8)$$

which shall be used in the following.

The initialization function in the `costella_image_convert` library therefore sets up nine lookup tables for the conversion of RGB to YCbCr. The additive constants are included in the  $R \rightarrow Y$ ,  $B \rightarrow C_b$ , and  $R \rightarrow C_r$  tables, the latter two of which are actually identical (and hence only actually allocated one array of memory, with two pointers pointing to the same array).

We now need to consider the conversion of YCbCr back to RGB space. If one inverts the matrix equations (5) and (8) (*without* rounding off the decimals first, as done here), one obtains

$$\begin{aligned} R &= +1.000\,000\,Y + 0.000\,017\,(C_b - 128) + 1.402\,057\,(C_r - 128), \\ G &= +1.000\,000\,Y - 0.344\,148\,(C_b - 128) - 0.714\,151\,(C_r - 128), \\ B &= +1.000\,000\,Y + 1.772\,059\,(C_b - 128) + 0.000\,013\,(C_r - 128). \end{aligned} \quad (9)$$

Multiplying by 65 536, and rounding to the nearest integer, we would obtain

$$\begin{aligned} R &= Y + \left\{ \left[ (C_b - 128) + 91\,885\,(C_r - 128) + 32\,768 \right] \gg 16 \right\}, \\ G &= Y + \left\{ \left[ -22\,554\,(C_b - 128) - 46\,803\,(C_r - 128) + 32\,768 \right] \gg 16 \right\}, \\ B &= Y + \left\{ \left[ 116\,134\,(C_b - 128) + (C_r - 128) + 32\,768 \right] \gg 16 \right\}. \end{aligned} \quad (10)$$

The presence of  $C_b$  in  $R$ , and  $C_r$  in  $B$ , is due to the fact that we quantized the coefficients above, as well as modifying the multiplier to  $2 \times 32\,767$ ; these contributions are identically zero in an exact floating-point representation. It will be demonstrated below that these ‘‘polluting’’ terms can be omitted without detriment, yielding

$$\begin{aligned} R &= Y + \left\{ \left[ 91\,885\,(C_r - 128) + 32\,768 \right] \gg 16 \right\}, \\ G &= Y + \left\{ \left[ -22\,554\,(C_b - 128) - 46\,803\,(C_r - 128) + 32\,768 \right] \gg 16 \right\}, \\ B &= Y + \left\{ \left[ 116\,134\,(C_b - 128) + 32\,768 \right] \gg 16 \right\}. \end{aligned} \quad (11)$$

Expanding out the parentheses in (11), we obtain

$$\begin{aligned} R &= Y + \left\{ (91\,885\,C_r - 11\,728\,512) \gg 16 \right\}, \\ G &= Y + \left\{ (-22\,554\,C_b - 46\,803\,C_r + 8\,910\,464) \gg 16 \right\}, \\ B &= Y + \left\{ (116\,134\,C_b - 14\,832\,384) \gg 16 \right\}. \end{aligned} \quad (12)$$

Clearly, in this case we need to limit the resulting  $R$ ,  $G$ , and  $B$  values to the permissible range  $[0, 255]$ .

It was asserted above that the omission of the “polluting” cross-contributions of  $C_b$  to  $R$  and of  $C_r$  to  $B$  in Eqs. (11) does not cause any detriment to the transformation inversion process. The proof of this assertion is an exhaustive conversion of all 16 777 216 possible RGB combinations to YCbCr according to Eqs. (4) and (7), and then back to RGB according to either Eqs. (10) or Eqs. (11). Using Eqs. (10), one finds that 33 847 744 of the 50 331 648 individual  $R$ ,  $G$ , and  $B$  values are reproduced exactly; 8 241 372 of the values come out one unit too small; and 8 242 532 of the values come out one unit too large; in other words, around two-thirds of values come back correct, one-sixth come back too small by one unit, and one-sixth come back too large by one unit. Using the simplified and “unpolluted” equations (11) yields exactly the same number of values reproduced exactly, namely, 33 847 744; in this case, 8 241 575 values come out one unit too small, and 8 242 329 values come out too large. In both cases, the error is never larger in magnitude than one unit. Thus, the extra “polluting” terms can be safely omitted, and indeed the formulas (12) used in the `costella_image_convert` library mandate their omission. These formulas, again, will be computed identically on all architectures and on all compilers.

The initialization function in the `costella_image_convert()` library therefore sets up four lookup tables for the conversion of YCbCr back to RGB, namely, for  $C_r \rightarrow R$ ,  $C_b \rightarrow B$ ,  $C_b \rightarrow G$ , and  $C_r \rightarrow G$ . The additive constants are included in the first three of these tables.

## Dependencies

The `costella_image_convert` library depends on the following libraries:

```
costella_image
costella_wrap
costella_base
```

## Public interface functions

The public interface functions in the `costella_image_convert` library are as follows:

```
costella_image_convert_initialize()
```

This is the public interface function for initializing the `costella_image_convert` library. Its function prototype is

```
int costella_image_convert_initialize( FILE* pfileError );
```

```
costella_image_convert_finalize()
```

This is the public interface function for finalizing the `costella_image_convert` library. Its function prototype is

```
int costella_image_convert_finalize( FILE* pfileError );
```

```
costella_image_convert_rgb_to_ycbcr()
```

This public interface function converts a color `COSTELLA_IMAGE` from the RGB colorspace to the YCbCr colorspace. Its function prototype is

```
int costella_image_convert_rgb_to_ycbcr( COSTELLA_IMAGE* piIn,
    COSTELLA_IMAGE* piOut, int (*pfProgress)( void* pvPassback ), void*
    pvPassback, FILE* pfileError );
```

piIn: Pointer to the input COSTELLA\_IMAGE.

piOut: Pointer to the output COSTELLA\_IMAGE. This may be the same as piIn.

costella\_image\_convert\_ycbcr\_to\_rgb()

This public interface function converts a color COSTELLA\_IMAGE from the YCbCr colorspace to the RGB colorspace. Its function prototype is

```
int costella_image_convert_ycbcr_to_rgb( COSTELLA_IMAGE* piIn,
    COSTELLA_IMAGE* piOut, int (*pfProgress)( void* pvPassback ), void*
    pvPassback, FILE* pfileError );
```

piIn: Pointer to the input COSTELLA\_IMAGE.

piOut: Pointer to the output COSTELLA\_IMAGE. This may be the same as piIn.

## Public interface macros

The public interface macros in the costella\_image\_convert library are as follows:

COSTELLA\_IMAGE\_CONVERT\_RGB\_TO\_Y()

This macro computes the Y value corresponding to a set of RGB values. Its macro prototype is

```
#define COSTELLA_IMAGE_CONVERT_RGB_TO_Y( lubR, lubG, lubB, lpubY ) \
```

lubR: R channel value.

lubG: G channel value.

lubB: B channel value.

lpubY: Pointer to where the Y value will be stored.

The macro computes the Y value using the conversion lookup tables set up during initialization.

COSTELLA\_IMAGE\_CONVERT\_RGB\_TO\_YCBCR()

This macro computes the YCbCr values corresponding to a set of RGB values. Its macro prototype is

```
#define COSTELLA_IMAGE_CONVERT_RGB_TO_YCBCR( lubR, lubG, lubB, lpubY, \
    lpubCb, lpubCr ) \
```



### CostellaImageConvertFinalize()

This function may be called when the user no longer requires the functionality of the library. Its function prototype is

```
COSTELLA_FUNCTION( CostellaImageConvertFinalize, ( void ) )
```

The initialization flag for this library is cleared.

The lookup tables for RGB↔YCbCr conversions are then freed.

Finally, the `costella_wrap` and `costella_image` libraries are finalized.

### CostellaImageConvertRgbToYcbcr()

This function converts a color `COSTELLA_IMAGE` from the RGB colorspace to the YCbCr colorspace. If an alpha channel is attached, and the output image is not the same image as the input image, then the alpha channel is copied across to the new image. Its function prototype is

```
COSTELLA_FUNCTION( CostellaImageConvertRgbToYcbcr, ( COSTELLA_IMAGE*  
    piIn, COSTELLA_IMAGE* piOut, COSTELLA_CALLBACK_FUNCTION pfProgress,  
    COSTELLA_0* poPassback ) )
```

`piIn`: Pointer to the input `COSTELLA_IMAGE`.

`piOut`: Pointer to the output `COSTELLA_IMAGE`. May be the same as `piIn`.

The function first checks that the input and output images are both color, that the input image is in the RGB colorspace. It sets the colorspace of the output image to YCbCr, and copies the downsampled chrominance and nonreplicated downsampled chrominance flags from the input image.

The function then extracts information from the input image: the alpha channel flag, the width, and height; and information from both the input and output images: the image and alpha channel row strides; and pointers to alpha channel and color image structures. The function then checks that the alpha channel flag and the image dimensions agree in the output image.

The function then walks through the rows and columns of the input and output images, converting the input RGB values to YCbCr values, and storing them in the output image. If an alpha channel is attached, and the output image is different from the input image, the alpha values are copied across.

### CostellaImageConvertYcbcrToRgb()

This function converts a color `COSTELLA_IMAGE` from the YCbCr colorspace to the RGB colorspace. Its function prototype is

```
COSTELLA_FUNCTION( CostellaImageConvertYcbcrToRgb, ( COSTELLA_IMAGE*  
    piIn, COSTELLA_IMAGE* piOut, COSTELLA_CALLBACK_FUNCTION pfProgress,  
    COSTELLA_0* poPassback ) )
```

`piIn`: Pointer to the input `COSTELLA_IMAGE`.

`piOut`: Pointer to the output `COSTELLA_IMAGE`. May be the same as `piIn`.

If the input image has nonreplicated downsampled chrominance, then only the top-left pixel in each  $2 \times 2$  block is converted with chrominance data; the other pixels in each block are converted as luminance only. If an alpha channel is attached, and the output image is not the same image as the input image, then the alpha channel is copied across to the new image.

The function first checks that the input and output images are both color, and that the input image is in the YCbCr colorspace. It sets the colorspace of the output image to RGB.

The function then extracts information from the input image: the alpha channel flag; the downsampled chrominance flag; the nonreplicated downsampled chrominance flag; and the width and height; and information from both the input and output images: the image and alpha channel row strides; and pointers to alpha channel and color image structures. The function then checks that the alpha channel flag and the image dimensions agree in the output image. The downsampled chrominance flag in the output image is set equal to that of the input image.

The function then walks through the rows and columns of the input and output images, keeping track of the parities of the row and column that it is in. It converts the input YCbCr values to RGB values, and stores them in the output image. As noted, if the input image has nonreplicated downsampled chrominance, the chrominance values are set to zero for all pixels except those in the top-left corner of each  $2 \times 2$  block. If an alpha channel is attached, and the output image is different from the input image, the alpha values are copied across.

## Non-public-interface macros

There are no non-public-interface macros in the `costella_image_convert` library.

## 2.7. The `costella_unblock` library

The `costella_unblock` library automatically removes, as far as possible, the “blocking” artifacts visible in images that have been compressed by JPEG at low quality settings.

The UnBlock algorithm itself is described in a separate scientific paper, available from the author’s website, currently [assassinationscience.com/johncostella](http://assassinationscience.com/johncostella). A number of terms defined in the scientific paper are used in the following, without further explanation; it is assumed that the reader has read the scientific paper before reading this document.

It should be noted that the width and height of a JPEG image are constrained to be 16-bit quantities, even though they are described by `COSTELLA_UDs` in this library in order to be compatible with the `costella_image` library. Therefore, the total number of pixels in a JPEG image can always be described by a 32-bit quantity, and so quadword support is not required in this library.

## Dependencies

The `costella_unblock` library depends on the following libraries:

```
costella_image_chrominance
costella_image_convert
costella_image
costella_wrap
costella_base
```

## Public interface functions

The public interface functions in the `costella_unblock` library are as follows:

`costella_unblock_initialize()`

This is the public interface function for initializing the `costella_unblock` library. Its function prototype is

```
int costella_unblock_initialize( FILE* pfileError );
```

`costella_unblock_finalize()`

This is the public interface function for finalizing the `costella_unblock` library. Its function prototype is

```
int costella_unblock_finalize( FILE* pfileError );
```

`costella_unblock()`

This public interface function performs the UnBlock algorithm on a `COSTELLA_IMAGE`. Its function prototype is

```
int costella_unblock( COSTELLA_IMAGE* piIn, COSTELLA_IMAGE* piOut, int
    bPhotographic, int bCartoon, int (*pfProgress)( void* pvPassback ),
    void* pvPassback, FILE* pfileError );
```

`piIn`: Pointer to the input `COSTELLA_IMAGE`.

`piOut`: Pointer to the output `COSTELLA_IMAGE`. Can be the same as `piIn`.

`bPhotographic`: If nonzero, the image is known to be photographic (continuous-tone).

`bCartoon`: If nonzero, the image is known to be “cartoon”-like (solid blocks of color with sharp edges between them).

## Public interface macros

There are no public interface macros in the `costella_unblock` library.

## Non-public-interface functions

The non-public-interface functions in the `costella_unblock` library are as follows:

### `CostellaUnblockInitialize()`

This function must be called before using any functionality of the library. Its function prototype is

```
COSTELLA_FUNCTION( CostellaUnblockInitialize, ( void ) )
```

The initialization flag for the library is set, and the `costella_image_chrominance` and `costella_image_convert` libraries are initialized.

The function then allocates memory for several global lookup tables that are then populated: a square-root table; several multiplication and rounding tables for values in the domain  $[0, 255]$ ; and several multiplication tables for values in the domain  $[-255, +255]$ .

### `CostellaUnblockFinalize()`

This function may be called when the user no longer requires the functionality of the library. Its function prototype is

```
COSTELLA_FUNCTION( CostellaUnblockFinalize, ( void ) )
```

The initialization flag for the library is cleared, the global lookup tables are freed, and the `costella_image_chrominance` and `costella_image_convert` libraries are finalized.

### `CostellaUnblock()`

This function performs the UnBlock algorithm on an image that has been subjected to JPEG compression. Its function prototype is

```
COSTELLA_FUNCTION( CostellaUnblock, ( COSTELLA_IMAGE* piIn,  
    COSTELLA_IMAGE* piOut, COSTELLA_B bPhotographic, COSTELLA_B bCartoon,  
    COSTELLA_CALLBACK_FUNCTION pfProgress, COSTELLA_0* poPassback ) )
```

`piIn`: Pointer to the input `COSTELLA_IMAGE`.

`piOut`: Pointer to the output `COSTELLA_IMAGE`. Can be the same as `piIn`.

`bPhotographic`: If nonzero, the image is known to be photographic (continuous-tone).

`bCartoon`: If nonzero, the image is known to be “cartoon”-like (solid blocks of color with sharp edges between them).

The UnBlock algorithm can either be performed “in place” on a `COSTELLA_IMAGE`, or else an output `COSTELLA_IMAGE` can be specified that is different from the input `COSTELLA_IMAGE`.

The two flags `bPhotographic` and `bCartoon` provide functionality not described in the scientific paper: the UnBlock algorithm can be told to act “conservatively”, for two specific types of image:

1. “photographic” images (continuous tone images), for which it is to be assumed that sharp edges are the exception, rather than the rule; and
2. “cartoon” image (images containing only a finite number of colors), for which it is to be assumed that sharp edges are the rule, rather than the exception.

This “conservatism” is implemented by performing a rough statistical estimate of the uncertainty in the frequency tables, and then adjusting to either edge of the confidence interval, depending on which option is chosen. If neither of these options are chosen, the function proceeds to apply the UnBlock algorithm as explained in the scientific paper, which uses the “best guess” of the statistical analysis.

The function first extracts information about the input and output `COSTELLA_IMAGES` and, in debug mode, performs consistency checks on that information. It then allocates memory for frequency tables for the discrepancies, adjustment tables for the discrepancies, and buffers that will be passed to the correction function.

The function then checks whether the input `COSTELLA_IMAGE` is already in the YCbCr colorspace. If not, the function converts it from the RGB colorspace to the YCbCr colorspace using `CostellaImageConvertRgbToYcbcr()`. The result is stored in the output `COSTELLA_IMAGE`.

If the image is color, the function next ensures that the chrominance values have been downsampled. It does this by simply averaging the chrominance values in each  $2 \times 2$  block; if they are already downsampled, then this step will achieve nothing. (It is actually more efficient to simply ensure that this *has* been done, rather than to try to detect *whether* it has been done.)

The function then computes the vertical discrepancies, computes the adjustments, and corrects them; it then does the same for the horizontal discrepancies.

The function then upsamples the output `COSTELLA_IMAGE`’s chrominance values, either by simple replication (if the output `COSTELLA_IMAGE` specifies downsampled chrominance), or else by using the magic kernel.

Finally, the function then checks whether the output `COSTELLA_IMAGE` is specified to be in the RGB colorspace rather than the YCbCr colorspace; if so, it is converted in-place.

#### `CostellaUnblockComputeVerticalDiscrepancies()`

This internal function computes the vertical discrepancy statistics for an image. Its function prototype is

```
static COSTELLA_FUNCTION( CostellaUnblockComputeVerticalDiscrepancies,
    ( COSTELLA_SW* aswBufferY, COSTELLA_SW* aswBufferCb, COSTELLA_SW*
    aswBufferCr, COSTELLA_UD* audYBoundaryU, COSTELLA_UD* audYBoundaryV,
    COSTELLA_UD* audCbBoundaryU, COSTELLA_UD* audCbBoundaryV,
    COSTELLA_UD* audCrBoundaryU, COSTELLA_UD* audCrBoundaryV,
    COSTELLA_UD* audYInternalU, COSTELLA_UD* audYInternalV, COSTELLA_UD*
    audCbInternalU, COSTELLA_UD* audCbInternalV, COSTELLA_UD*
```

```

    audCrInternalU, COSTELLA_UD* audCrInternalV, COSTELLA_UD*
    pudTotalLuminance, COSTELLA_UD* pudTotalChrominance, COSTELLA_IMAGE*
    pi, COSTELLA_CALLBACK_FUNCTION pfProgress, COSTELLA_0* poPassback ) )

```

**aswBufferY**: Buffer for the Y channel.

**aswBufferCb**: Buffer for the Cb channel.

**aswBufferCr**: Buffer for the Cr channel.

**audYBoundaryU**: Array to hold the frequencies for the  $u$  discrepancies at block boundaries for the Y channel.

**audYBoundaryV**: Array to hold the frequencies for the  $v$  discrepancies at block boundaries for the Y channel.

**audCbBoundaryU**: Array to hold the frequencies for the  $u$  discrepancies at block boundaries for the Cb channel.

**audCbBoundaryV**: Array to hold the frequencies for the  $v$  discrepancies at block boundaries for the Cb channel.

**audCrBoundaryU**: Array to hold the frequencies for the  $u$  discrepancies at block boundaries for the Cr channel.

**audCrBoundaryV**: Array to hold the frequencies for the  $v$  discrepancies at block boundaries for the Cr channel.

**audYInternalU**: Array to hold the frequencies for the  $u$  discrepancies internal to each block for the Y channel.

**audYInternalV**: Array to hold the frequencies for the  $v$  discrepancies internal to each block for the Y channel.

**audCbInternalU**: Array to hold the frequencies for the  $u$  discrepancies internal to each block for the Cb channel.

**audCbInternalV**: Array to hold the frequencies for the  $v$  discrepancies internal to each block for the Cb channel.

**audCrInternalU**: Array to hold the frequencies for the  $u$  discrepancies internal to each block for the Cr channel.

**audCrInternalV**: Array to hold the frequencies for the  $v$  discrepancies internal to each block for the Cr channel.

**pudTotalLuminance**: Location where the total luminance in the `COSTELLA_IMAGE` will be stored.

**pudTotalChrominance**: Location where the total chrominance in the `COSTELLA_IMAGE` will be stored.

**pi**: Pointer to the `COSTELLA_IMAGE` to analyze.

The function starts by computing the double row stride, and initializing the luminance and chrominance totals. It then initializes the frequency tables for boundary and internal  $u$  and  $v$  values.

The function then proceeds to analyze the luminance channel. It walks through the rows of the image. For each row, it starts to the right of the leftmost pixel, *i.e.*, at  $x = 1$  (where the leftmost column is  $x = 0$ ). It then fills positions 8 and 9 of the values array from the  $x = 1$  and  $x = 2$  pixels; these will be shifted back to the start of the values array in the main loop.

The function then walks through the vertical boundaries of the image. It only measures discrepancies for a block if it has a right boundary, *i.e.*, at least one pixel to its right; this ensures that the total counts for internal and boundary discrepancies will be identically equal. The function ensures this by checking if the column that is eight pixels to the right of the block in question is within the image.

The function then extracts the next eight pixels from the image, making sure that it doesn't go past the right edge of the image. Any unused entry in the values array is filled with the value  $-1$ , which the discrepancy function recognizes as a missing entry.

The function then calls `CostellaUnblockComputeDiscrepancies()` to compute the internal and boundary discrepancies. The signs of the discrepancies are discarded, and the absolute values used to increment the appropriate entries in the frequency tables. The total count of luminance values is incremented.

For a color image, the function then proceeds to analyze the discrepancies for the chrominance channels in the same way. The main difference here is that the chrominance data is only contained on every second row and column of the image.

Finally, the total counts of luminance and chrominance values are stored in the specified locations.

#### `CostellaUnblockComputeHorizontalDiscrepancies()`

This internal function computes the horizontal discrepancy statistics for an image. Its function prototype is

```
static COSTELLA_FUNCTION(
    CostellaUnblockComputeHorizontalDiscrepancies, ( COSTELLA_SW*
    aswBufferY, COSTELLA_SW* aswBufferCb, COSTELLA_SW* aswBufferCr,
    COSTELLA_UD* audYBoundaryU, COSTELLA_UD* audYBoundaryV, COSTELLA_UD*
    audCbBoundaryU, COSTELLA_UD* audCbBoundaryV, COSTELLA_UD*
    audCrBoundaryU, COSTELLA_UD* audCrBoundaryV, COSTELLA_UD*
    audYInternalU, COSTELLA_UD* audYInternalV, COSTELLA_UD*
    audCbInternalU, COSTELLA_UD* audCbInternalV, COSTELLA_UD*
    audCrInternalU, COSTELLA_UD* audCrInternalV, COSTELLA_UD*
    pudTotalLuminance, COSTELLA_UD* pudTotalChrominance, COSTELLA_IMAGE*
    pi, COSTELLA_CALLBACK_FUNCTION pfProgress, COSTELLA_0* poPassback ) )
```

`aswBufferY`: Buffer for the Y channel.

`aswBufferCb`: Buffer for the Cb channel.

`aswBufferCr`: Buffer for the Cr channel.

**audYBoundaryU**: Array to hold the frequencies for the  $u$  discrepancies at block boundaries for the Y channel.  
**audYBoundaryV**: Array to hold the frequencies for the  $v$  discrepancies at block boundaries for the Y channel.  
**audCbBoundaryU**: Array to hold the frequencies for the  $u$  discrepancies at block boundaries for the Cb channel.  
**audCbBoundaryV**: Array to hold the frequencies for the  $v$  discrepancies at block boundaries for the Cb channel.  
**audCrBoundaryU**: Array to hold the frequencies for the  $u$  discrepancies at block boundaries for the Cr channel.  
**audCrBoundaryV**: Array to hold the frequencies for the  $v$  discrepancies at block boundaries for the Cr channel.  
**audYInternalU**: Array to hold the frequencies for the  $u$  discrepancies internal to each block for the Y channel.  
**audYInternalV**: Array to hold the frequencies for the  $v$  discrepancies internal to each block for the Y channel.  
**audCbInternalU**: Array to hold the frequencies for the  $u$  discrepancies internal to each block for the Cb channel.  
**audCbInternalV**: Array to hold the frequencies for the  $v$  discrepancies internal to each block for the Cb channel.  
**audCrInternalU**: Array to hold the frequencies for the  $u$  discrepancies internal to each block for the Cr channel.  
**audCrInternalV**: Array to hold the frequencies for the  $v$  discrepancies internal to each block for the Cr channel.  
**pubTotalLuminance**: Location where the total luminance in the `COSTELLA_IMAGE` will be stored.  
**pubTotalChrominance**: Location where the total chrominance in the `COSTELLA_IMAGE` will be stored.  
**pi**: Pointer to the `COSTELLA_IMAGE` to analyze.

This function proceeds completely analogously (with rows and columns interchanged) to `CostellaUnblockComputeVerticalDiscrepancies()`.

#### `CostellaUnblockCorrectVerticalDiscrepancies()`

This internal function corrects the vertical discrepancies in an image. If there is an alpha channel, and if the output alpha image is not the same as the input alpha image, the alpha channel values are copied over. Its function prototype is

```
static COSTELLA_FUNCTION( CostellaUnblockCorrectVerticalDiscrepancies,
    ( COSTELLA_SW* aswBufferY, COSTELLA_SW* aswBufferCb, COSTELLA_SW*
```

```

    aswBufferCr, COSTELLA_UB* aubYAdjustedU, COSTELLA_UB* aubYAdjustedV,
    COSTELLA_UB* aubCbAdjustedU, COSTELLA_UB* aubCbAdjustedV,
    COSTELLA_UB* aubCrAdjustedU, COSTELLA_UB* aubCrAdjustedV,
    COSTELLA_IMAGE* piIn, COSTELLA_IMAGE* piOut,
    COSTELLA_CALLBACK_FUNCTION pfProgress, COSTELLA_0* poPassback ) )

```

aswBufferY: Buffer for the Y channel.

aswBufferCb: Buffer for the Cb channel.

aswBufferCr: Buffer for the Cr channel.

aubYAdjustedU: Adjustment table for  $u$  for the Y channel.

aubYAdjustedV: Adjustment table for  $v$  for the Y channel.

aubCbAdjustedU: Adjustment table for  $u$  for the Cb channel.

aubCbAdjustedV: Adjustment table for  $v$  for the Cb channel.

aubCrAdjustedU: Adjustment table for  $u$  for the Cr channel.

aubCrAdjustedV: Adjustment table for  $v$  for the Cr channel.

piIn: Pointer to the input COSTELLA\_IMAGE.

piOut: Pointer to the output COSTELLA\_IMAGE.

After defining the usual convenient constants, the function begins with the luminance channel. It walks through the rows of the image. If the width of the image is less than 9 pixels, then there are no vertical boundaries in the image at all, and the function merely copies the row across from the input to the output image. Otherwise, the function loads the first eight pixels of the row into the righthand side of the values array, which get shifted to the lefthand side in the main loop.

The function then walks through the vertical boundaries of the row. It shifts the righthand half of the values array to the left, and loads up the next eight values, filling any unused positions with  $-1$ . It then computes the discrepancies at the boundary, adjusts them, corrects for them, and then writes the 16 corrected values to the output image.

At the end of the row, the function writes out any remaining pixels that have not already been written out.

The function then proceeds to correct the chrominance channels in the same fashion, working on the downsampled rows and columns.

#### CostellaUnblockCorrectHorizontalDiscrepancies()

This internal function corrects the horizontal discrepancies in an image. Its function prototype is

```

static COSTELLA_FUNCTION(
    CostellaUnblockCorrectHorizontalDiscrepancies, ( COSTELLA_SW*
    aswBufferY, COSTELLA_SW* aswBufferCb, COSTELLA_SW* aswBufferCr,

```

```

COSTELLA_UB* aubYAdjustedU, COSTELLA_UB* aubYAdjustedV, COSTELLA_UB*
aubCbAdjustedU, COSTELLA_UB* aubCbAdjustedV, COSTELLA_UB*
aubCrAdjustedU, COSTELLA_UB* aubCrAdjustedV, COSTELLA_IMAGE* piIn,
COSTELLA_IMAGE* piOut, COSTELLA_CALLBACK_FUNCTION pfProgress,
COSTELLA_0* poPassback ) )

```

aswBufferY: Buffer for the Y channel.

aswBufferCb: Buffer for the Cb channel.

aswBufferCr: Buffer for the Cr channel.

aubYAdjustedU: Adjustment table for  $u$  for the Y channel.

aubYAdjustedV: Adjustment table for  $v$  for the Y channel.

aubCbAdjustedU: Adjustment table for  $u$  for the Cb channel.

aubCbAdjustedV: Adjustment table for  $v$  for the Cb channel.

aubCrAdjustedU: Adjustment table for  $u$  for the Cr channel.

aubCrAdjustedV: Adjustment table for  $v$  for the Cr channel.

piIn: Pointer to the input COSTELLA\_IMAGE.

piOut: Pointer to the output COSTELLA\_IMAGE.

This function proceeds completely analogously (with rows and columns interchanged) to `CostellaUnblockCorrectVerticalDiscrepancies()`.

#### `CostellaUnblockComputeDiscrepancies()`

This internal function computes the discrepancy values  $u$  and  $v$  from an array of six values, according to the formulas provided in the scientific paper. If any of the last two values are  $-1$ , then the pixel in question is missing, and the function applies the alternative formulas described in the scientific paper. Its function prototype is

```

static COSTELLA_FUNCTION( CostellaUnblockComputeDiscrepancies, (
    COSTELLA_SW* aswValues, COSTELLA_SW* pswU, COSTELLA_SW* pswV ) )

```

aswValues: Array containing the six intensity values. If any of the last two values are  $-1$ , it indicates missing pixels; alternative formulas are used.

pswU: Location where  $u$  will be stored. Must be a valid pointer.

pswV: Location where  $v$  will be stored, or null if  $v$  is not wanted.

#### `CostellaUnblockCorrectDiscrepancies()`

This internal function corrects an array of 16 values for the (adjusted) discrepancies  $u$  and  $v$ , according to the formulas provided in the scientific paper. Its function prototype is

```

static COSTELLA_FUNCTION( CostellaUnblockCorrectDiscrepancies, (
    COSTELLA_SW* asw, COSTELLA_SW swU, COSTELLA_SW swV ) )

```

**asw**: Array of sixteen intensity values, covering two complete blocks.

**swU**: The adjusted  $u$  value to correct for.

**swV**: The adjusted  $v$  value to correct for.

#### CostellaUnblockComputeAdjustments()

This internal function computes the adjustment lookup tables. Its function prototype is

```
static COSTELLA_FUNCTION( CostellaUnblockComputeAdjustments, (
    COSTELLA_UD* audReference, COSTELLA_UD* audMeasured, COSTELLA_UD
    udTotal, COSTELLA_B bConservativePhotographic, COSTELLA_B
    bConservativeCartoon, COSTELLA_UB* aubAdjusted ) )
```

**audReference**: Frequency table for the reference (internal) discrepancy values.

**audMeasured**: Frequency table for the measured (block boundary) discrepancy values.

**udTotal**: The total frequency.

**bConservativePhotographic**: If nonzero, compute the adjustments conservatively, assuming a continuous-tone image.

**bConservativeCartoon**: If nonzero, compute the adjustments conservatively, assuming a solid-tone cartoon image with sharp boundaries.

The function first determines whether either of the conservative modes have been specified.

It then computes half of the total frequency, rounded up.

It then walks simultaneously (but at different rates) through the measured and discrepancy frequency tables; the measured discrepancy is incremented on each iteration. For each measured discrepancy, the cumulative measured frequency is incremented by the frequency at that measured discrepancy.

If a conservative estimate has been requested, the function then computes a conservative bound that takes into account the limited statistics of both the measured and the reference values. A rough estimate of the standard deviation is computed as the approximate square-root of the cumulative frequency (if less than half the total) or the difference between the cumulative frequency and the total cumulative frequency (if greater than half the total). A lower bound of two levels is imposed on the result. The function then uses a rough estimate of eight times this approximate standard deviation as an adjustment frequency (four for the measured, and four for the reference). If the image is photographic, this amount is subtracted from the measured value, with the lower limit being zero; if cartoon, it is added, with the upper limit being the total cumulative frequency. (If not in either conservative mode, the measured cumulative frequency is used unmodified.)

The function then builds the reference cumulative frequency until it is greater than or equal to the conservative bound of the cumulative measured frequency. It continues

adding the frequency of the reference discrepancy value, and incrementing this reference discrepancy value, until the cumulative reference frequency is equal to or greater than the conservative bound of the cumulative measured frequency. If the reference cumulative frequency has consequently exceeded the conservative bound of the cumulative measured frequency (rather than simply being equal to it), then the addition of the final reference value is undone.

The function now computes the adjusted discrepancy. If the reference discrepancy is zero, then the adjusted discrepancy is simply the measured discrepancy. Otherwise, the function checks whether the measured discrepancy is greater than or equal to the reference discrepancy. If so, then the adjusted discrepancy is one more than the difference between them (one more, because the reference value is one less than the value the function is sitting on). Otherwise, the measured discrepancy is less than the reference discrepancy, then the adjusted discrepancy is zero.

`costella_unblock_approx_square_root()`

This internal function returns an approximate square-root (accurate to its leading three bits) of a `COSTELLA_UD`, using the table computed in initialization. Its function prototype is

```
static COSTELLA_UD costella_unblock_approx_square_root( COSTELLA_UD ud
)
```

`ud`: Value to approximately square-root.

*Return value*: The approximate square-root.

The function first determines the bit-width of `ud`. If this is less than 9, then the square-root table can be used directly. Otherwise, the function square-roots the top 7 or 8 bits, depending on whether the bit-width is odd or even, and uses the fact that  $\sqrt{100} = 10$  in any number system (including binary) to simply halve the number of remaining bits in the number.

## Non-public-interface macros

The non-public-interface macros in the `costella_unblock` library are as follows:

`COSTELLA_UNBLOCK_ADJUST_DISCREPANCY()`

This internal macro function adjusts a discrepancy value using a lookup table. Its macro prototype is

```
#define COSTELLA_UNBLOCK_ADJUST_DISCREPANCY( lsw, laubAdjusted ) \
```

`lsw`: Discrepancy value to be adjusted.

`laubAdjusted`: Adjustment table to be used.

*Evaluates to*: The adjusted discrepancy value.

The macro checks whether the given value is positive or negative, and negates where appropriate for negative values, as the lookup tables contain positive values only.

#### `COSTELLA_UNBLOCK_CORRECT()`

This internal macro function corrects a discrepancy value. Its macro prototype is

```
#define COSTELLA_UNBLOCK_CORRECT( lpsw, lswD ) \
```

`lpsw`: Pointer to the discrepancy value to be corrected.

`lswD`: Difference to be added to the discrepancy value.

The macro adds the specified difference to the discrepancy value, and then range-limits the result.

## 2.8. The `costella_jpeg` library

The `costella_jpeg` library is not a part of the core set of `costella` libraries, but rather is supplied only to support the various sample applications that have been provided.

This library provides rudimentary support for reading and writing JPEG image files using the Independent JPEG Group (IJG) JPEG library version 6b.

The IJG `.c` and `.h` library files are not included in the standard distribution. They can be obtained from [ijg.org](http://ijg.org). (They are also available from the author's website.) The API and workings of the IJG library are not described in the following; please refer to the IJG documentation if you want further information in this regard.

### Dependencies

The `costella_jpeg` library depends on the following libraries:

```
costella_image  
costella_wrap  
costella_base
```

### Public interface functions

The public interface functions in the `costella_jpeg` library are as follows:

#### `costella_jpeg_initialize()`

This is the public interface function for initializing the `costella_jpeg` library. Its function prototype is

```
int costella_jpeg_initialize( FILE* pfileError );
```

### `costella_jpeg_finalize()`

This is the public interface function for finalizing the `costella_jpeg` library. Its function prototype is

```
int costella_jpeg_finalize( FILE* pfileError );
```

### `costella_jpeg_load()`

This public interface function loads a `COSTELLA_IMAGE` from a JPEG file. Its function prototype is

```
int costella_jpeg_load( COSTELLA_IMAGE* pi, char* acFilename, int
    (*pfImageNew)( COSTELLA_IMAGE* pi, void* pvPassback ), void*
    pvPassback, FILE* pfileError );
```

`pi`: Pointer to the `COSTELLA_IMAGE` structure into which the JPEG image will be loaded.

`acFilename`: String containing the filename of the JPEG image to load, including any extension (such as “.jpg”).

`pfImageNew`: Function that will allocate memory in the `COSTELLA_IMAGE` structure on the basis of the dimensions and color flag stored in that structure by this function.

`pvPassback`: Pointer that will be passed back to `pfImageNew()`.

### `costella_jpeg_save()`

This public interface function saves a `COSTELLA_IMAGE` as a JPEG file. Its function prototype is

```
int costella_jpeg_save( COSTELLA_IMAGE* pi, char* acFilename, unsigned
    char byQuality, FILE* pfileError );
```

`pi`: Pointer to the `COSTELLA_IMAGE` to save.

`acFilename`: String containing the filename of the desired JPEG image file, including any extension (such as “.jpg”).

`byQuality`: IJG JPEG quality factor, in the range 0–100.

## **Public interface macros**

There are no public interface macros in the `costella_jpeg` library.

## Non-public-interface functions

The non-public-interface functions in the `costella_jpeg` library are as follows:

### `CostellaJpegInitialize()`

This function must be called before using any functionality of the library. Its function prototype is

```
COSTELLA_FUNCTION( CostellaJpegInitialize, ( void ) )
```

The initialization flag for the library is set.

The `costella_image` library is initialized.

### `CostellaJpegFinalize()`

This function may be called when the functionality of the library is no longer requires. Its function prototype is

```
COSTELLA_FUNCTION( CostellaJpegFinalize, ( void ) )
```

The initialization flag for the library is cleared.

The `costella_image` library is finalized.

### `CostellaJpegLoad()`

This function loads a `COSTELLA_IMAGE` from a JPEG file. Its function prototype is

```
COSTELLA_FUNCTION( CostellaJpegLoad, ( COSTELLA_IMAGE* pi, COSTELLA_C*  
    acFilename, COSTELLA_FUNCTION_POINTER( pfImageNew, ( COSTELLA_IMAGE*  
    pi, COSTELLA_0* poPassback ) ), COSTELLA_0* poPassback ) )
```

`pi`: Pointer to the `COSTELLA_IMAGE` structure into which the JPEG image will be loaded.

`acFilename`: String containing the filename of the JPEG image to load, including any extension (such as “.jpg”).

`pfImageNew`: Function that will allocate memory in the `COSTELLA_IMAGE` structure on the basis of the dimensions and color flag stored in that structure by this function.

`poPassback`: Pointer that will be passed back to `pfImageNew()`.

The function first opens the specified file. It then sets up the IJG error handling structures. The JPEG decompression object is created. The input source is specified to be the file that was opened. The header of the JPEG file is read, and the output dimensions calculated. “Fancy upsampling” is turned off; this ensures that chrominance information is replicated throughout each  $2 \times 2$  block. The function then determines whether the input image is grayscale or color, based on the number of components it

has. The image dimensions are extracted. The grayscale flag and image dimensions are stored in the `COSTELLA_IMAGE` structure.

The calling application's callback function is then called to allocate memory for the image of the specified dimensions and grayscale flag.

The JPEG decompressor is then started. The width of a row of JPEG samples is computed, and the required memory allocated. A pointer to this row is stored in the dummy array required by the IJG library.

The function then walks through the image, reading each row of the JPEG image, and storing the image data in the `COSTELLA_IMAGE`.

Finally, the JPEG decompressor is finished and destroyed, the input file is closed, and the row of JPEG samples is freed.

### `CostellaJpegSave()`

This function writes an image out to a JPEG file. Its function prototype is

```
COSTELLA_FUNCTION( CostellaJpegSave, ( COSTELLA_IMAGE* pi, COSTELLA_C*
    acFilename, COSTELLA_UB ubQuality ) )
```

`pi`: Pointer to the `COSTELLA_IMAGE` to save.

`acFilename`: String containing the filename of the desired JPEG image file, including any extension (such as “.jpg”).

`ubQuality`: IJG JPEG quality factor, in the range 0–100.

The function opens the output file. It then sets up the IJG error handling structures. The JPEG compression object is created. The output file is specified as the destination of the JPEG compression object. The width, height, and colorspace of the output JPEG file are specified. Default compression parameters are then computed by the IJG library. The width of a row of JPEG samples is computed, and the required memory allocated. A pointer to this row is stored in the dummy array required by the IJG library. The JPEG compressor is then started.

The function then walks through the image, filling the JPEG sample row from the image, and writing them out to the JPEG file.

Finally, the JPEG compressor is finished and destroyed, the output file is closed, and the row of samples is freed.

### `costella_jpeg_error_exit_load()`

This internal function is used to allow the IJG library to jump directly back to an error jump-point in `CostellaJpegLoad()`. Its function prototype is

```
static void costella_jpeg_error_exit_load( j_common_ptr cinfo );
```

`cinfo`: Pointer to the JPEG decompression object.

The function extracts the JPEG error manager, and performs a `longjmp()` to the error handling location previously set up in `CostellaJpegLoad()`.

`costella_jpeg_error_exit_save()`

This internal function is used to allow the IJG library to jump directly back to an error jump-point in `CostellaJpegSave()`. Its function prototype is

```
static void costella_jpeg_error_exit_save( j_common_ptr cinfo );
```

`cinfo`: Pointer to the JPEG compression object.

The function extracts the JPEG error manager, and performs a `longjmp()` to the error handling location previously set up in `CostellaJpegSave()`.

`CostellaJpegWrapImageNew()`

This internal function is used to “wrap” a public-interface `COSTELLA_IMAGE` creator function specified to `costella_jpeg_load()`, along the same lines as the functionality in the `costella_wrap` library. Its function prototype is

```
static COSTELLA_FUNCTION( CostellaJpegWrapImageNew, ( COSTELLA_IMAGE*  
    pi, COSTELLA_O* poPassback ) )
```

`pi`: Pointer to the `COSTELLA_IMAGE` structure.

`poPassback`: Passback pointer from `CostellaJpegLoad()`.

The function casts `poPassback` into a `COSTELLA_JPEG_WRAP_IMAGE_NEW*`, and then extracts from this structure the public interface creator and passback pointer. It then calls this public interface creator, handling the case of an error return.

### Non-public-interface macros

There are no non-public-interface macros in the `costella_jpeg` library.

## 3. Copyright notice

Copyright © 1989–2007 John P. Costella.

Permission is hereby granted, free of charge, to any person obtaining a copy of this document and associated software (the “Software”) to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES

OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY CLAIM, DAMAGES, OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT, OR OTHERWISE, ARISING FROM, OUT OF, OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.