The Preliminary Guides to the MegaWave2 Software, Versions 2.x

Volume Two

MegaWave2 System Library

by Jacques Froment

Ecole Normale Supérieure de Cachan
61 avenue du Président Wilson
94235 Cachan cedex, France

May 5, 2004

`http://www.cmla.ens-cachan.fr/Cmla/Megawave`

Contents

1 Introduction

1.1 What you will find in this guide

When you implement an algorithm in MegaWave2, you write a code in C language in what we call a module (See Volume one: "MegaWave2 User's Guide"). Your algorithm processes some objects which represent your data. So you need to know how to create an object of the type you want, how to access to it, how to remove it, etc.

This present guide will detail all the available MegaWave2 objects and most related functions which are part of the System Library (Sections 2 to 8). In addition, you will find the description of other functions which may be called by the user in the module - such as error handling functions - (Section 9). There is also a description of the Wdevice Library, a toolbox for the window interface (Section 10).

This guide is a reference manual : it would be boring to read it from the beginning to the end. If you are new with MegaWave2, you should entirely read this introduction were basic principles are explained, and all introductions of the next main sections, to get an idea about the various objects you may use. Afterward, when you will be searching for a particular structure or function, consult the contents page 2 or the index page 226.

1.2 The MegaWave2 memory (internal) types

MegaWave2 objects such as images, movies, signals, curves, ..., are represented in the module code as *pointers to a structure*. The type of the structure defines the object you want to process, as `struct fimage` for an image of Floating points values (the pointer of this structure is of type `Fimage`).

Each structure has particular fields, as `gray` for a `Fimage` which represents the gray levels plane. They are described in the section presenting the structure (Section 2.3.1 page 36 for `Fimage`).

Some fields are common to most structures, they are:

- `cmt` : string of maximum size `mw_cmtsize` where to put the comment associated to the object. For input objects and at the beginning of the module statement, this field contains the comment field of the corresponding file object (if the file type provides a comment field). For output objects and at the end of the module statement, this field contains the name of the module plus the comments of the input objects, if any. This default output value can be overwritten by setting a value to `cmt`.
- `name` : string of maximum size `mw_namesize` where to put the name of this object. For input objects, this field contains the file name of the corresponding file object. The default output value is "?". It can be overwritten.

You can of course access to any field in order to read its content. But be carefull when you want to overwrite the content of a field: some fields have to be updated by the system library only (e.g. the dimension fields `nrow` and `ncol` of image objects).

Some structures may contain undocumented fields: they are used internally by the system library and users should not access to them, especially for writing.

Some conversions between memory types are available as functions of the System Library, see Section 9.2 of this guide for a list of the most current conversion functions.

1.3 File (external) types or file formats

1.3.1 Generalities

When a module's command finishes, the output objects (of memory types) have to be saved on disk for future use. For example, they can be the input of another module's command. Data may be saved on disk also (or read from disk) when the module is run into an interpreter such as XMegaWave2, although in this case modules communicate with memory type structures.

This shows that external type objects are needed; they are files written in a predefined format. MegaWave2 can use some well-known formats available in the public domain, especially to carry the different image memory types. When no satisfying standard is available to match a given memory type, a specific format is used. Notice that, whereas there is only one memory type associated to an object, an object of a given memory type may be represented on disk with various file types.

Conversions between some formats are available: you may load an object written in a file type which is different from the regular one used for the memory type of your object. Depending on the case, you may however lose precision in your data (in that case, a warning message is send). For output ojects, MegaWave2 chooses a default file type to write the data. You can modify this choice using the system option `-ftype` (See Volume one: "MegaWave2 User's Guide").

A short description of the file types is given in the next sections about the different memory types.

1.3.2 Search path convention

When a module is called in the command line mode, MegaWave2 searches the file names of the input objects in different directories, following the order:

1. the current directory of the shell, i.e. "`.`",
2. `$MY_MEGAWAVE2/data` and its subdirectories,
3. `$MEGAWAVE2/data` and its subdirectories.

Notice that this search path convention has changed from MegaWave2 Versions 1.x to Versions 2.x.

The output objects are always written in the current directory of the shell. *Beware : if you give the same name as the one of an existing file, the content of the previous file will be overwritten (there is no confirmation message).*

2 Images

All image structures share the following important fields:

- `nrow,ncol` : define the size of the image, by the number of rows and the number of columns (not to be overwritten by user). Notice that the range over the x axis is $0 \ldots$ `ncol` $- 1$ and that the range over the y axis is $0 \ldots$ `nrow` $- 1$.
- `previous, next` : pointers to the previous and the next image. These fields are used only when the image is part of a movie.

Each image structure has also one or several fields to record the pixel values. When the image is monochrome, there is only one field called `gray`. Color images use three fields called `red`, `green` and `blue`. The C type of these array fields depends to the image object: they can be pointers to `unsigned char` values or pointers to `floating points` values.

You can put values in those arrays, at the expressed condition that you respect the C type of the field and that you do not exceed the maximum value of the index, given by ncol $\times$ nrow $- 1$. For example, `image->gray[y*image->ncol+x]` is the gray level of the pixel of coordinates (x, y) that is, the column $\#x$ and the row $\#y$. Ranges are $0 \ldots nrow - 1$ for y and $0 \ldots ncol - 1$ for x.

You can shorten this expression in your modules using C macro, for example:

```
#define _(a,i,j) ((a)->gray[(j)*(a)->ncol+(i)] )
```

allows you to access to the pixel (x, y) by writing `_(image,x,y)`.

Tip to speed your modules: images are built from left to right and up to down. If you can write your algorithm to access to the pixel following this natural order, you can speed it considerably using the following scheme. In this example, one copies each pixel of the cimage M into the fimage B only if the pixel of M is not equal to zero:

```
Cimage M;    /* Input of the module */
Fimage *B;   /* Output of the module */

register float *ptrB;
register unsigned char *ptrM;
register int i;

for (i=0, ptrB = (*B)->gray, ptrM = M->gray;
             i< M->ncol*M->nrow;
             i++, ptrB++, ptrM++)
          if (*ptrM) *ptrB = (float) *ptrM;
```

If you scan the pixels in a random order, you may rather define a bi-dimensional tab `A` so that `A[l][c]` points to the pixel's value `(c,l)`. See the functions `mw_newtab_cimage()`, `mw_newtab_fimage()`, ...

2.1 Char Images

Use preferably the *Char Images* memory type each time you can write an algorithm which directly computes the gray level as an integer between 0 (black) and 255 (white) : such discrete scheme will be

more accurate, faster and will require far less memory than a continuous scheme (i.e. with floating points computations).

2.1.1 The structure Cimage

Beginners should only focus on the first three fields of this structure. You should also consider the fields `previous` and `next` if your image is part of a movie. Some fields are not used this time, such `firstcol` ... `lastrow`, but future modules may access to them.

```
typedef struct cimage {
  int nrow;        /* Number of rows (dy) */
  int ncol;        /* Number of columns (dx) */
  unsigned char *gray;     /* The Gray level plane (may be NULL) */

  float scale;     /* Scale of the picture (should be 1 for original pict.) */
  char cmt[mw_cmtsize]; /* Comments */
  char name[mw_namesize]; /* Name of the image */

  /* Defines the signifiant part of the picture : */
  int firstcol;    /* index of the first col not affected by left side effect*/
  int lastcol;     /* index of the last col not affected by right side effect*/
  int firstrow;    /* index of the first row not aff. by upper side effect */
  int lastrow;     /* index of the last row not aff. by lower side effect */

  /* For use in Movies only */
  struct cimage *previous; /* Pointer to the previous image (may be NULL) */
  struct cimage *next; /* Pointer to the next image (may be NULL) */

} *Cimage;
```

Do not change by yourself the content of `nrow` and `ncol`: the size of the image has to be modified using functions of the library only (see section 2.1.3 page 10).

2.1.2 Related file (external) types

The list of the available formats is the following:

1. `"IMG"` Original format defined by the defunct software PCVision (from ImageAction), and widely used by MegaWave1. This format carries the comments field (`cmt`) of the memory object.
2. `"TIFF"` Tag Image Format with one 8-bits plane (unsigned char gray levels). This format carries the comments field (`cmt`) of the memory object. It has been developed by Sam Leffler and Silicon Graphics, Inc. To use this format, you need the TIFF library (libtiff). See the Volume One, section "Installation". Output objects are created without compression.
3. `"PGMA"` PGM (portable graymap file format) in Ascii version.
4. `"PGMR"` PGM (portable graymap file format) in Rawbits version.
5. `"PM_C"` PM format with one 8-bits plane (unsigned char gray levels). This format carries the comments field (`cmt`) of the memory object. It has been developed by the University of Pennsylvania, USA.

6. `"GIF"` GIF87 (Graphics Interchange Format) 8-bits per pixel, non interlaced. This format has been developed by CompuServe Incorporated.
7. `"BMP"` Microsoft BMP 8-bits per pixel. Output objects are created using Windows BMP format. Compression methods are not implemented.
8. `"JFIF"` JPEG/JFIF format with one 8-bits plane (unsigned char gray levels). This format carries the comments field (`cmt`) of the memory object. It has been developed by the Independent JPEG Group's software. To use this format, you need the JPEG library (libjpeg). See the Volume One, section "Installation". The compression ratio is defined by the quality factor, which is an integer between 1 (worse) and 100 (best). Default quality factor is 100. To change this value, add it as an option to the JFIF type. For example, JFIF:50 means JFIF file type with quality factor 50. Whatever the quality factor, output objects are created with loosely compression.
9. `"PS"` PostScript (level 1) format, *for output objects only.* This format has been developed by Adobe Systems Incorporated.
10. `"EPSF"` Encapsulated PostScript (level 1) format, *for output objects only.* Same as PS format but more suitable when used with some softwares that recognize encapsulated comments, such as LaTeX.
11. `"INR"` Original format defined by the software Inrimage (from INRIA). This is a very old version, implemented for backward-compatibility with MegaWave1, and it should not be used anymore.
12. `"MTI"` Original format defined by the software MultImage (from 2AI), and used by MegaWave1. Quite exotic now.
13. `"BIN"` This is the "universal" image format for 8-bits gray levels images. It records one byte per pixel, without header. Since it does not contain any header, the image file must be a square (i.e. the number of columns and the number of lines must be the same).

2.1.3 Functions Summary

The following is a description of all the functions related to the `Cimage` type. The list is in alphabetical order.

○Name

mw_alloc_cimage - Allocate the gray plane

○Summary

Cimage mw_alloc_cimage(image,nrow,ncol)

Cimage image;

int nrow, ncol;

○Description

This function allocates the gray plane of a `Cimage` structure previously created using `mw_new_cimage`. The size of the image is given by `nrow` (number of rows or maximum range of y plus one) and `ncol` (number of columns or maximum range of x plus one). Pixels can be addressed after this call, if the allocation successed. There is no default value for the pixels.

Do not use this function if `image` has already an allocated plane: use the function `mw_change_cimage` instead.

The function `mw_alloc_cimage` returns `NULL` if not enough memory is available to allocate the gray plane. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

○Example

```
Cimage image=NULL; /* Internal use: no Input neither Output of module */

if  ( ((image = mw_new_cimage()) == NULL) ||
      (mw_alloc_cimage(image,256,256) == NULL) )
    mwerror(FATAL,1,"Not enough memory.\n");

/* Set pixel (0,1) to white */
image->gray[256] = 255;
```

○Name

mw_change_cimage - Change the size of the gray plane

○Summary

Cimage mw_change_cimage(image, nrow, ncol)

Cimage image;

int nrow, ncol;

○Description

This function changes the memory allocation of the gray plane of a `Cimage` structure, even if no previously memory allocation was done. The new size of the image is given by `nrow` (number of rows or maximum range of y plus one) and `ncol` (number of columns or maximum range of x plus one).

It can also create the structure if the input `image = NULL`. Therefore, this function can replace both `mw_new_cimage` and `mw_alloc_cimage`. It is the recommended function to set image dimension of input/output modules. Since the function can set the address of `image`, the variable must be set to the return value of the function (See example below).

The function `mw_change_cimage` returns `NULL` if not enough memory is available to allocate the gray plane. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

○Example

```
Cimage Output; /* Output of module */

Output = mw_change_cimage(Output, 256, 256);
if (Output == NULL) mwerror(FATAL,1,"Not enough memory.\n");
```

○Name

mw_clear_cimage - Clear the gray plane

○Summary

void mw_clear_cimage(image, v)

Cimage image;

unsigned char v;

○Description

This function fills the cimage `image` with the gray value given by v: all pixels will have the gray level v.

The speed of this function depends to the C library implementation, but it is usually very fast (trying to do faster is a waste of time).

○Example

```
Cimage image; /* Output of module */

image = mw_change_cimage(image, 100, 100);
if (image == NULL) mwerror(FATAL,1,"Not enough memory.\n");
/* Set all pixels to white */
mw_clear_cimage(image,255);
```

○Name

mw_copy_cimage - Copy the pixel values of an image into another one

○Summary

void mw_copy_cimage(in, out)

Cimage in,out;

○Description

This function copies the content of the gray plane of the image `in` into the gray plane of the image `out`. The size of the two gray planes must be the same.

The speed of this function depends to the C library implementation, but it is usually very fast (trying to do faster is a waste of time).

○Example

```
Cimage G; /* Needed Input */
Cimage F; /* Optional Output */

  if (F) {
    printf("F option is active: copy G in F\n");
    if ((F = mw_change_cimage(F, G->nrow, G->ncol)) == NULL)
mwerror(FATAL,1,"Not enough memory.\n");
    else mw_copy_cimage(G, F);
   }
  else  printf("F option is not active\n");
```

○Name

mw_delete_cimage - Deallocate the gray plane

○Summary

void mw_delete_cimage(image)

Cimage image;

○Description

This function deallocates the gray plane of a `Cimage` structure previously allocated using `mw_alloc_cimage` or `mw_change_cimage`, and the structure itself.

You should set `image = NULL` after this call since the address pointed by `image` is no longer valid.

○Example

```
Cimage image=NULL; /* Internal use: no Input neither Output of module */

if  ( ((image = mw_new_cimage()) == NULL) ||
      (mw_alloc_cimage(image,256,256) == NULL) )
    mwerror(FATAL,1,"Not enough memory.\n");
.
.
.
mw_delete_cimage(image);
image = NULL;
```

○Name

mw_draw_cimage - Draw a line

○Summary

void mw_draw_cimage(image, a0, b0, a1, b1, c)

Cimage image;

int a0,b0,a1,b1; unsigned char c;

○Description

This function draws in `image` a connected line of gray level c between the pixel $(a0, b0)$ and the pixel $(a1, b1)$.

○Example

```
Cimage image; /* Output of module */

image = mw_change_cimage(image, 100, 100);
if (image == NULL) mwerror(FATAL,1,"Not enough memory.\n");
/* Set all pixels to white */
mw_clear_cimage(image,255);
/* Draw a black diagonal line */
mw_draw_cimage(image,0,0,99,99,0);
```

○Name

mw_getdot_cimage - Return the gray level value

○Summary

unsigned char mw_getdot_cimage(image, x, y)

Cimage image;

int x,y;

○Description

This function returns the gray level value (a number between 0 - black - and 255 - white -) of the given `image` for the pixel (x, y) (column $\#x$ and row $\#y$).

Notice that a call to this function is a slow (but easy and secure) way to read a pixel value. See section 2 page 8 for how to read pixels fast.

○Example

```
Cimage image; /* Needed Input of module */
int x,y;      /* Needed Inputs of module */

if ((x < image->ncol) && (y < image->nrow))
printf("image(%d,%d) = %d\n",x,y,mw_getdot_cimage(image,x,y));
else mwerror(ERROR,1,"Out of bounds !\n");
```

○Name

mw_isitbinary_cimage - Check if the image is binary

○Summary

unsigned char mw_isitbinary_cimage(image)

Cimage image;

○Description

This function returns 0 if `image` is not a binary image, a value > 0 if it is one. In that case, the returned value corresponds to the maxima value that is, to the only one value $\neq 0$. Image with two gray levels only but with the minimal value > 0 is not considered by this function as binary.

○Example

```
Cimage image; /* Needed Input of module */
unsigned char white;

if ((white=mw_isitbinary_cimage(image)) > 0)
 printf("Binary image with white set to %d\n",(int) white);
else
 printf("Not a binary image\n");
```

○Name

mw_new_cimage - Create a new Cimage

○Summary

Cimage mw_new_cimage();

○Description

This function creates a new `Cimage` structure with an empty gray plane. No pixels can be addressed at this time. The gray plane may be allocated using the function `mw_alloc_cimage` or `mw_change_cimage`.

Do not use this function for input/output of modules, since the MegaWave2 Compiler already created the structure for you if you need it (See Volume one: "MegaWave2 User's Guide"). Use instead the function `mw_change_cimage`. Do not forget to deallocate the internal structures before the end of the module, except if they are part of an input or output movie.

The function `mw_new_cimage` returns `NULL` if not enough memory is available to create the structure. Your code should check this value to send an error message in the `NULL` case, and do appropriate statement.

○Example

```
Cimage image=NULL; /* Internal use: no Input neither Output of module */

if  ( ((image = mw_new_cimage()) == NULL) ||
      (mw_alloc_cimage(image,256,256) == NULL) )
    mwerror(FATAL,1,"Not enough memory.\n");
```

○Name

mw_newtab_gray_cimage - Create a bi-dimensional tab for the pixels of a Cimage

○Summary

unsigned char ** mw_newtab_gray_cimage(image)

Cimage image;

○Description

This function creates a new bi-dimensional tab which allows an easy and fast access to the pixels' gray level. This tab is actually an one-dimensional tab of pointers, so that each pointer points to the beginning of a line in the gray plane of the given image.

This function must be called after the gray plane has been allocated, using for example one of the functions `mw_new_cimage`, `mw_alloc_cimage` or `mw_change_cimage`. After that, if the gray plane allocation is changed (by e.g. `mw_change_cimage` or `mw_delete_cimage`), the tab is no longer valid and must be deleted using `free(tab)`.

Ones the tab has been correctly created, is it possible to read or to write the value of the pixel (x, y) (x being an index for column and y for row) using `tab[y][x]`.

○Example

```
Cimage image; /* Needed Input of module (gray plane already allocated and filled) */
int x,y;      /* Needed Input of module */
unsigned char **tab;

tab =  mw_newtab_gray_cimage(image);
if (tab==NULL) mwerror(FATAL,1,"Not enough memory\n");

/* Put white color in the pixel (x,y) */
if ((x < image->ncol) && (y < image->nrow)) tab[y][x] = 255;
else mwerror(ERROR,1,"Out of bounds !\n");

free(tab);
```

○Name

mw_plot_cimage - Set the gray level value

○Summary

void mw_plot_cimage(image, x, y, v)

Cimage image;

int x,y;

unsigned char v;

○Description

This function set the gray level value of the given `image` for the pixel (x, y) (column $\#x$ and row $\#y$) to be `v` (a number between 0 - black - and 255 - white -).

Notice that a call to this function is a slow (but easy and secure) way to write a pixel value. See section 2 page 8 for how to write pixels fast.

○Example

```
Cimage image; /* Needed Input of module */
int x,y;      /* Needed Inputs of module */

/* Put white color in the pixel (x,y) */
if ((x < image->ncol) && (y < image->nrow))
        mw_plot_cimage(image,x,y,255));
else mwerror(ERROR,1,"Out of bounds !\n");
```

2.2 Color Char Images

Use the *Color Char Images* memory type each time you want to process color images. As in the Char Images case, the use of this format instead of the corresponding floating point format (`Cfimage`) is strongly recommended.

2.2.1 The structure Ccimage

Beginners should focus on the first five fields only of this structure. You should also consider the fields `previous` and `next` if your image is part of a movie. Some fields are not used at this time, such `firstcol` ... `lastrow`, but future modules may access to them.

```
typedef struct ccimage {
  int nrow;        /* Number of rows (dy) */
  int ncol;        /* Number of columns (dx) */

  unsigned char *red;     /* The red level plane (may be NULL) */
  unsigned char *green;   /* The green level plane (may be NULL) */
  unsigned char *blue;    /* The blue level plane (may be NULL) */

  float scale;     /* Scale of the picture (should be 1 for original pict.) */
  char cmt[mw_cmtsize]; /* Comments */
  char name[mw_namesize]; /* Name of the image */

  /* Defines the signifiant part of the picture : */
  int firstcol;    /* index of the first col not affected by left side effect*/
  int lastcol;     /* index of the last col not affected by right side effect*/
  int firstrow;    /* index of the first row not aff. by upper side effect */
  int lastrow;     /* index of the last row not aff. by lower side effect */

  /* For use in Movies only */
  struct ccimage *previous; /* Pointer to the previous image (may be NULL) */
  struct ccimage *next; /* Pointer to the next image (may be NULL) */

} *Ccimage;
```

Do not change by yourself the content of `nrow` and `ncol`: the size of the image has to be modified using functions of the library only (see section 2.2.3 page 23).

You can put `unsigned char` values in the arrays `red`, `green`, `blue` at the expressed condition that you do not exceed the maximum value of the index, given by ncol $\times$ nrow $-$ 1.

Actually, everything works as for the `Cimage` structure (see section 2.1.1 page 9) but you have to deal with three planes instead of only one. That is the proportion between each RGB component that will give you the color. Notice that you can get more than 16 millions of different colors ($2^{3\times 8}$ exactly), so you need appropriate device to see or print such image with fidelity.

2.2.2 Related file (external) types

The list of the available formats is the following:

1. `"TIFFC"` Tag Image Format with three 8-bits color planes (24 bits color). This format carries the comments field (`cmt`) of the memory object. It has been developed by Sam Leffler and Silicon Graphics, Inc. To use this format, you need the TIFF library (libtiff). See the Volume One, section "Installation". Output objects are created without compression.
2. `"PMC_C"` PM format with three 8-bits planes (24 bits color). This format carries the comments field (`cmt`) of the memory object. It has been developed by the University of Pennsylvania, USA.
3. `"BMPC"` Microsoft BMP 24-bits per pixel. Output objects are created using Windows BMP format. Compression methods are not implemented.
4. `"PPM"` Portable pixmap format (24 bits color). Only the "raw" PPM format is supported, the "plain" (ascii) one being definitely too wasteful of space to record color images.
5. `"JFIFC"` JPEG/JFIF format with three 8-bits planes (24 bits color). This format carries the comments field (`cmt`) of the memory object. It has been developed by the Independent JPEG Group's software. To use this format, you need the JPEG library (libjpeg). See the Volume One, section "Installation". The compression ratio is defined by the quality factor, which is an integer between 1 (worse) and 100 (best). Default quality factor is 100. To change this value, add it as an option to the JFIFC type. For example, JFIFC:50 means JFIFC file type with quality factor 50. Whatever the quality factor, output objects are created with loosely compression.

2.2.3 Functions Summary

The following is a description of all the functions related to the `Ccimage` type. The list is in alphabetical order.

○Name

mw_alloc_ccimage - Allocate the RGB planes

○Summary

Ccimage mw_alloc_ccimage(image,nrow,ncol)

Ccimage image;

int nrow, ncol;

○Description

This function allocates the RGB planes of a `Ccimage` structure previously created using `mw_new_ccimage`. The size of the image is given by `nrow` (number of rows or maximum range of y plus one) and `ncol` (number of columns or maximum range of x plus one). Pixels can be addressed after this call, if the allocation successed. There is no default value for the pixels.

Do not use this function if `image` has already an allocated plane: use the function `mw_change_ccimage` instead.

The function `mw_alloc_ccimage` returns `NULL` if not enough memory is available to allocate the RGB planes. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

○Example

```
Ccimage image=NULL; /* Internal use: no Input neither Output of module */

if  ( ((image = mw_new_ccimage()) == NULL) ||
      (mw_alloc_ccimage(image,256,256) == NULL) )
    mwerror(FATAL,1,"Not enough memory.\n");

/* Set pixel (0,1) to white */
image_>red[256] = image->green[256] = image->blue[256] = 255;
```

○Name

mw_change_ccimage - Change the size of the RGB planes

○Summary

Ccimage mw_change_ccimage(image, nrow, ncol)

Ccimage image;

int nrow, ncol;

○Description

This function changes the memory allocation of the RGB planes of a `Ccimage` structure, even if no previously memory allocation was done. The new size of the image is given by `nrow` (number of rows or maximum range of y plus one) and `ncol` (number of columns or maximum range of x plus one).

It can also create the structure if the input `image = NULL`. Therefore, this function can replace both `mw_new_ccimage` and `mw_alloc_ccimage`. It is the recommended function to set image dimension of input/output modules. Since the function can set the address of `image`, the variable must be set to the return value of the function (See example below).

The function `mw_change_ccimage` returns `NULL` if not enough memory is available to allocate the RGB planes. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

○Example

```
Cimage Output; /* Output of module */

Output = mw_change_ccimage(Output, 256, 256);
if (Output == NULL) mwerror(FATAL,1,"Not enough memory.\n");
```

○Name

mw_clear_ccimage - Clear the RGB planes

○Summary

void mw_clear_ccimage(image, r,g,b)

Ccimage image;

unsigned char r,g,b;

○Description

This function fills the ccimage `image` with the color given by the triplet `r,g,b`: all pixels will have this RGB value.

The speed of this function depends to the C library implementation, but it is usually very fast (trying to do faster is a waste of time).

○Example

```
Ccimage image; /* Output of module */

image = mw_change_ccimage(image, 100, 100);
if (image == NULL) mwerror(FATAL,1,"Not enough memory.\n");
/* Set all pixels to blue */
mw_clear_ccimage(image,0,0,255);
```

○Name

mw_copy_ccimage - Copy the pixel values of color image into another one

○Summary

void mw_copy_ccimage(in, out)

Ccimage in,out;

○Description

This function copies the content of the RGB planes of the image `in` into the RGB planes of the image `out`. The size of the two RGB planes must be the same.

The speed of this function depends to the C library implementation, but it is usually very fast (trying to do faster is a waste of time).

○Example

```
Ccimage G; /* Needed Input */
Ccimage F; /* Optional Output */

  if (F) {
    printf("F option is active: copy G in F\n");
    if ((F = mw_change_ccimage(F, G->nrow, G->ncol)) == NULL)
     mwerror(FATAL,1,"Not enough memory.\n");
    else mw_copy_ccimage(G, F);
   }
  else  printf("F option is not active\n");
```

○Name

mw_delete_ccimage - Deallocate the RGB planes

○Summary

void mw_delete_ccimage(image)

Ccimage image;

○Description

This function deallocates the RGB planes of a `Ccimage` structure previously allocated using `mw_alloc_ccimage` or `mw_change_ccimage`, and the structure itself.

You should set `image = NULL` after this call since the address pointed by `image` is no longer valid.

○Example

```
Ccimage image=NULL; /* Internal use: no Input neither Output of module */

if  ( ((image = mw_new_ccimage()) == NULL) ||
      (mw_alloc_ccimage(image,256,256) == NULL) )
    mwerror(FATAL,1,"Not enough memory.\n");
.
.
.
mw_delete_ccimage(image);
image = NULL;
```

○Name

mw_draw_ccimage - Draw a line

○Summary

void mw_draw_ccimage(image, a0, b0, a1, b1, r, g, b)

Ccimage image;

int a0,b0,a1,b1;

unsigned char r,g,b;

○Description

This function draws in `image` a connected line between the pixel $(a0, b0)$ and the pixel $(a1, b1)$. The color of the line is defined by the triplet `r,g,b`.

○Example

```
Ccimage image; /* Output of module */

image = mw_change_ccimage(image, 100, 100);
if (image == NULL) mwerror(FATAL,1,"Not enough memory.\n");
/* Set all pixels to white */
mw_clear_ccimage(image,255,255,255);
/* Draw a red diagonal line */
mw_draw_ccimage(image,0,0,99,99,255,0,0);
```

○Name

mw_getdot_ccimage - Return the RGB value

○Summary

void mw_getdot_ccimage(image, x, y, r, g, b)

Ccimage image;

int x,y;

unsigned char *r,*g,*b;

○Description

This function returns the RGB value of the given `image` for the pixel (x, y) (column $\#x$ and row $\#y$). The RGB value consists of the triplet `*r,*g,*b`: `*r` (a number between 0 and 255) gives you the proportion of red, `*g` the proportion of green and `*b` the proportion of blue.

Notice that a call to this function is a slow (but easy and secure) way to read a pixel value. See section 2 page 8 for how to read pixels fast.

○Example

```
Ccimage image; /* Needed Input of module */
int x,y;       /* Needed Inputs of module */
unsigned char r,g,b;  /* Internal use */

if ((x < image->ncol) && (y < image->nrow))
  {
    mw_getdot_ccimage(image,x,y,&r,&g,&b);
    printf("image(%d,%d) = %d,%d,%d\n",x,y,(int)r,(int)g,(int)b);
  }
else mwerror(ERROR,1,"Out of bounds !\n");
```

○Name

mw_new_ccimage - Create a new Ccimage

○Summary

Ccimage mw_new_ccimage();

○Description

This function creates a new `Ccimage` structure with empty RGB planes. No pixels can be addressed at this time. The RGB planes may be allocated using the function `mw_alloc_ccimage` or `mw_change_ccimage`.

Do not use this function for input/output of modules, since the MegaWave2 Compiler already created the structure for you if you need it (See Volume one: "MegaWave2 User's Guide"). Use instead the function `mw_change_ccimage`. Do not forget to deallocate the internal structures before the end of the module.

The function `mw_new_ccimage` returns `NULL` if not enough memory is available to create the structure. Your code should check this value to send an error message in the `NULL` case, and do appropriate statement.

○Example

```
Ccimage image=NULL; /* Internal use: no Input neither Output of module */

if  ( ((image = mw_new_ccimage()) == NULL) ||
      (mw_alloc_ccimage(image,256,256) == NULL) )
    mwerror(FATAL,1,"Not enough memory.\n");
```

○Name

mw_newtab_blue_ccimage - Create a bi-dimensional tab for the blue pixels of a Ccimage

○Summary

unsigned char ** mw_newtab_blue_ccimage(image)

Ccimage image;

○Description

This function creates a new bi-dimensional tab which allows an easy and fast access to the pixels' blue level. This tab is actually an one-dimensional tab of pointers, so that each pointer points to the beginning of a line in the blue plane of the given image.

This function must be called after the blue plane has been allocated, using for example one of the functions `mw_new_ccimage`, `mw_alloc_ccimage` or `mw_change_ccimage`. After that, if the blue plane allocation is changed (by e.g. `mw_change_ccimage` or `mw_delete_ccimage`), the tab is no longer valid and must be deleted using `free(tab)`.

Ones the tab has been correctly created, is it possible to read or to write the blue value of the pixel (x, y) (x being an index for column and y for row) using `tab[y][x]`.

Red and green pixels' value can be accessed with such a tab using the corresponding functions `mw_newtab_red_ccimage` and `mw_newtab_green_ccimage`.

○Example

```
Ccimage image; /* Needed Input of module (RGB planes already allocated and filled) */
int x,y;      /* Needed Input of module */
unsigned char **red,**green,**blue;

red =  mw_newtab_red_ccimage(image);
if (red==NULL) mwerror(FATAL,1,"Not enough memory\n");
green =  mw_newtab_green_ccimage(image);
if (green==NULL) mwerror(FATAL,1,"Not enough memory\n");
blue =  mw_newtab_blue_ccimage(image);
if (blue==NULL) mwerror(FATAL,1,"Not enough memory\n");

/* Put gray color in the pixel (x,y) */
if ((x < image->ncol) && (y < image->nrow))
        red[y][x] = green[y][x] = blue[y][x] = 127;
else mwerror(ERROR,1,"Out of bounds !\n");

free(blue); free(green); free(red);
```

○Name

mw_newtab_green_ccimage - Create a bi-dimensional tab for the green pixels of a Ccimage

○Summary

unsigned char ** mw_newtab_green_ccimage(image)

Ccimage image;

○Description

This function creates a new bi-dimensional tab which allows an easy and fast access to the pixels' green level. This tab is actually an one-dimensional tab of pointers, so that each pointer points to the beginning of a line in the green plane of the given image.

This function must be called after the green plane has been allocated, using for example one of the functions `mw_new_ccimage`, `mw_alloc_ccimage` or `mw_change_ccimage`. After that, if the green plane allocation is changed (by e.g. `mw_change_ccimage` or `mw_delete_ccimage`), the tab is no longer valid and must be deleted using `free(tab)`.

Ones the tab has been correctly created, is it possible to read or to write the green value of the pixel (x, y) (x being an index for column and y for row) using `tab[y][x]`.

Red and blue pixels' value can be accessed with such a tab using the corresponding functions `mw_newtab_red_ccimage` and `mw_newtab_blue_ccimage`.

○Example

```
Ccimage image; /* Needed Input of module (RGB planes already allocated and filled) */
int x,y;      /* Needed Input of module */
unsigned char **red,**green,**blue;

red =  mw_newtab_red_ccimage(image);
if (red==NULL) mwerror(FATAL,1,"Not enough memory\n");
green =  mw_newtab_green_ccimage(image);
if (green==NULL) mwerror(FATAL,1,"Not enough memory\n");
blue =  mw_newtab_blue_ccimage(image);
if (blue==NULL) mwerror(FATAL,1,"Not enough memory\n");

/* Put gray color in the pixel (x,y) */
if ((x < image->ncol) && (y < image->nrow))
        red[y][x] = green[y][x] = blue[y][x] = 127;
else mwerror(ERROR,1,"Out of bounds !\n");

free(blue); free(green); free(red);
```

○Name

mw_newtab_red_ccimage - Create a bi-dimensional tab for the red pixels of a Ccimage

○Summary

unsigned char ** mw_newtab_red_ccimage(image)

Ccimage image;

○Description

This function creates a new bi-dimensional tab which allows an easy and fast access to the pixels' red level. This tab is actually an one-dimensional tab of pointers, so that each pointer points to the beginning of a line in the red plane of the given image.

This function must be called after the red plane has been allocated, using for example one of the functions `mw_new_ccimage`, `mw_alloc_ccimage` or `mw_change_ccimage`. After that, if the red plane allocation is changed (by e.g. `mw_change_ccimage` or `mw_delete_ccimage`), the tab is no longer valid and must be deleted using `free(tab)`.

Ones the tab has been correctly created, is it possible to read or to write the red value of the pixel (x, y) (x being an index for column and y for row) using `tab[y][x]`.

Green and blue pixels' value can be accessed with such a tab using the corresponding functions `mw_newtab_green_ccimage` and `mw_newtab_blue_ccimage`.

○Example

```
Ccimage image; /* Needed Input of module (RGB planes already allocated and filled) */
int x,y;      /* Needed Input of module */
unsigned char **red,**green,**blue;

red =  mw_newtab_red_ccimage(image);
if (red==NULL) mwerror(FATAL,1,"Not enough memory\n");
green =  mw_newtab_green_ccimage(image);
if (green==NULL) mwerror(FATAL,1,"Not enough memory\n");
blue =  mw_newtab_blue_ccimage(image);
if (blue==NULL) mwerror(FATAL,1,"Not enough memory\n");

/* Put gray color in the pixel (x,y) */
if ((x < image->ncol) && (y < image->nrow))
        red[y][x] = green[y][x] = blue[y][x] = 127;
else mwerror(ERROR,1,"Out of bounds !\n");

free(blue); free(green); free(red);
```

○Name

mw_plot_ccimage - Set the RGB value

○Summary

void mw_plot_ccimage(image, x, y, r, g, b)

Ccimage image;

int x,y;

unsigned char r,g,b;

○Description

This function set the RGB value of the given `image` for the pixel (x, y) (column $\#x$ and row $\#y$) to be the triplet `r,g,b`: `r` (a number between 0 and 255) gives you the proportion of red, `g` the proportion of green and `b` the proportion of blue.

Notice that a call to this function is a slow (but easy and secure) way to write a pixel value. See section 2 page 8 for how to write pixels fast.

○Example

```
Ccimage image; /* Needed Input of module */
int x,y;      /* Needed Inputs of module */

/* Put green color in the pixel (x,y) */
if ((x < image->ncol) && (y < image->nrow))
        mw_plot_ccimage(image,x,y,0,255,0));
else mwerror(ERROR,1,"Out of bounds !\n");
```

2.3 Float Images

You may want to use this format when your algorithm process image computations using floating point arithmetic (continuous scheme). You may also use this format to represent any kind of two-dimensional real data (such as matrix).

Notice that you may lose precision when you use such format as the input of another module which requires integer representation (`Cimage` type), e.g. printing or displaying devices. In the other side, a module which accepts `Fimage` type as the input will also work without degradation if you put a `Cimage` type instead. It is so better to use, if possible, `Cimage` type for output variables and `Fimage` type for input.

2.3.1 The structure Fimage

This memory type is exactly the same as `Cimage` (See section 2.1.1 page 9): the only difference is about the `gray` field which is a pointer to floating points values.

Consequently, there is no formal correspondance between a gray level value and a visual gray level (e.g. 255.0 may not represent "white").

```
typedef struct fimage {
  int nrow;        /* Number of rows (dy) */
  int ncol;        /* Number of columns (dx) */
  float *gray;     /* The Gray level plane (may be NULL) */

  float scale;     /* Scale of the picture (should be 1 for original pict.) */
  char cmt[mw_cmtsize]; /* Comments */
  char name[mw_namesize]; /* Name of the image */

  /* Defines the signifiant part of the picture : */
  int firstcol;    /* index of the first col not affected by left side effect*/
  int lastcol;     /* index of the last col not affected by right side effect*/
  int firstrow;    /* index of the first row not aff. by upper side effect */
  int lastrow;     /* index of the last row not aff. by lower side effect */

} *Fimage;
```

2.3.2 Related file (external) types

The list of the available formats is the following:

1. `"RIM"` Original format defined by MegaWave1. It is close to the IMG format, but it uses a 32-bits plane in order to record floating point values. This format carries the comments field (`cmt`) of the memory object.
2. `"PM_F"` PM format with one 32-bits plane (floating point gray levels). This format carries the comments field (`cmt`) of the memory object.

2.3.3 Functions Summary

The following is a description of all the functions related to the `Fimage` type. The list is in alphabetical order.

○Name

mw_alloc_fimage - Allocate the gray plane

○Summary

Fimage mw_alloc_fimage(image,nrow,ncol)

Fimage image;

int nrow, ncol;

○Description

This function allocates the gray plane of a `Fimage` structure previously created using `mw_new_fimage`. The size of the image is given by `nrow` (number of rows or maximum range of y plus one) and `ncol` (number of columns or maximum range of x plus one). Pixels can be addressed after this call, if the allocation successed. There is no default value for the pixels.

Do not use this function if `image` has already an allocated plane: use the function `mw_change_fimage` instead.

The function `mw_alloc_fimage` returns `NULL` if not enough memory is available to allocate the gray plane. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

○Example

```
Fimage image=NULL; /* Internal use: no Input neither Output of module */

if  ( ((image = mw_new_fimage()) == NULL) ||
      (mw_alloc_fimage(image,256,256) == NULL) )
    mwerror(FATAL,1,"Not enough memory.\n");

/* Set pixel (0,1) to the value -1.0 */
image->gray[256] = -1.0;
```

○Name

mw_change_fimage - Change the size of the gray plane

○Summary

Fimage mw_change_fimage(image, nrow, ncol)

Fimage image;

int nrow, ncol;

○Description

This function changes the memory allocation of the gray plane of a `Fimage` structure, even if no previously memory allocation was done. The new size of the image is given by `nrow` (number of rows or maximum range of y plus one) and `ncol` (number of columns or maximum range of x plus one).

It can also create the structure if the input `image = NULL`. Therefore, this function can replace both `mw_new_fimage` and `mw_alloc_fimage`. It is the recommended function to set image dimension of input/output modules. Since the function can set the address of `image`, the variable must be set to the return value of the function (See example below).

The function `mw_change_fimage` returns `NULL` if not enough memory is available to allocate the gray plane. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

○Example

```
Fimage Output; /* Output of module */

Output = mw_change_fimage(Output, 256, 256);
if (Output == NULL) mwerror(FATAL,1,"Not enough memory.\n");
```

○Name

mw_clear_fimage - Clear the gray plane

○Summary

void mw_clear_fimage(image, v)

Fimage image;

float v;

○Description

This function fills the fimage `image` with the gray value given by v: all pixels will have the gray level v.

○Example

```
Fimage image; /* Output of module */

image = mw_change_fimage(image, 100, 100);
if (image == NULL) mwerror(FATAL,1,"Not enough memory.\n");
/* Set all pixels to 0.0 */
mw_clear_fimage(image,0.0);
```

○Name

mw_copy_fimage - Copy the pixel values of an image into another one

○Summary

void mw_copy_fimage(in, out)

Fimage in,out;

○Description

This function copies the content of the gray plane of the image `in` into the gray plane of the image `out`. The size of the two gray planes must be the same.

The speed of this function depends to the C library implementation, but it is usually very fast (trying to do faster is a waste of time).

○Example

```
Fimage G; /* Needed Input */
Fimage F; /* Optional Output */

  if (F) {
    printf("F option is active: copy G in F\n");
    if ((F = mw_change_fimage(F, G->nrow, G->ncol)) == NULL)
mwerror(FATAL,1,"Not enough memory.\n");
    else mw_copy_fimage(G, F);
   }
  else  printf("F option is not active\n");
```

○Name

mw_delete_fimage - Deallocate the gray plane

○Summary

void mw_delete_fimage(image)

Fimage image;

○Description

This function deallocates the gray plane of a `Fimage` structure previously allocated using `mw_alloc_fimage` or `mw_change_fimage`, and the structure itself.

You should set `image = NULL` after this call since the address pointed by `image` is no longer valid.

○Example

```
Fimage image=NULL; /* Internal use: no Input neither Output of module */

if  ( ((image = mw_new_fimage()) == NULL) ||
      (mw_alloc_fimage(image,256,256) == NULL) )
    mwerror(FATAL,1,"Not enough memory.\n");
.
.
.
mw_delete_fimage(image);
image = NULL;
```

○Name

mw_draw_fimage - Draw a line

○Summary

void mw_draw_fimage(image, a0, b0, a1, b1, c)

Fimage image;

int a0,b0,a1,b1; float c;

○Description

This function draws in `image` a connected line of gray level c between the pixel $(a0, b0)$ and the pixel $(a1, b1)$.

○Example

```
Fimage image; /* Output of module */

image = mw_change_fimage(image, 100, 100);
if (image == NULL) mwerror(FATAL,1,"Not enough memory.\n");
/* Clear all pixels */
mw_clear_fimage(image,0.0);
/* Draw a diagonal line of gray level 1.0 */
mw_draw_fimage(image,0,0,99,99,1.0);
```

○Name

mw_getdot_fimage - Return the gray level value

○Summary

float mw_getdot_fimage(image, x, y)

Fimage image;

int x,y;

○Description

This function returns the gray level value (any floating point number) of the given `image` for the pixel (x, y) (column $\#x$ and row $\#y$).

Notice that a call to this function is a slow (but easy and secure) way to read a pixel value. See section 2 page 8 for how to read pixels fast.

○Example

```
Fimage image; /* Needed Input of module */
int x,y;      /* Needed Inputs of module */

if ((x < image->ncol) && (y < image->nrow))
printf("image(%d,%d) = %f\n",x,y,mw_getdot_fimage(image,x,y));
else mwerror(ERROR,1,"Out of bounds !\n");
```

○Name

mw_new_fimage - Create a new Fimage

○Summary

Fimage mw_new_fimage();

○Description

This function creates a new `Fimage` structure with an empty gray plane. No pixels can be addressed at this time. The gray plane may be allocated using the function `mw_alloc_fimage` or `mw_change_fimage`.

Do not use this function for input/output of modules, since the MegaWave2 Compiler already created the structure for you if you need it (See Volume one: "MegaWave2 User's Guide"). Use instead the function `mw_change_fimage`. Do not forget to deallocate the internal structures before the end of the module.

The function `mw_new_fimage` returns `NULL` if not enough memory is available to create the structure. Your code should check this value to send an error message in the `NULL` case, and do appropriate statement.

○Example

```
Fimage image=NULL; /* Internal use: no Input neither Output of module */

if  ( ((image = mw_new_fimage()) == NULL) ||
      (mw_alloc_fimage(image,256,256) == NULL) )
    mwerror(FATAL,1,"Not enough memory.\n");
```

○Name

mw_newtab_gray_fimage - Create a bi-dimensional tab for the pixels of a Fimage

○Summary

float ** mw_newtab_gray_fimage(image)

Fimage image;

○Description

This function creates a new bi-dimensional tab which allows an easy and fast access to the pixels' gray level. This tab is actually an one-dimensional tab of pointers, so that each pointer points to the beginning of a line in the gray plane of the given image.

This function must be called after the gray plane has been allocated, using for example one of the functions `mw_new_fimage`, `mw_alloc_fimage` or `mw_change_fimage`. After that, if the gray plane allocation is changed (by e.g. `mw_change_fimage` or `mw_delete_fimage`), the tab is no longer valid and must be deleted using `free(tab)`.

Ones the tab has been correctly created, is it possible to read or to write the value of the pixel (x, y) (x being an index for column and y for row) using`tab[y][x]`.

○Example

```
Fimage image; /* Needed Input of module (gray plane already allocated and filled) */
int x,y;      /* Needed Input of module */
float **tab;

tab =  mw_newtab_gray_fimage(image);
if (tab==NULL) mwerror(FATAL,1,"Not enough memory\n");

/* Put 0 in the pixel (x,y) */
if ((x < image->ncol) && (y < image->nrow)) tab[y][x] = 0.0;
else mwerror(ERROR,1,"Out of bounds !\n");

free(tab);
```

○Name

mw_plot_fimage - Set the gray level value

○Summary

void mw_plot_fimage(image, x, y, v)

Fimage image;

int x,y;

float v;

○Description

This function set the gray level value of the given `image` for the pixel (x, y) (column $\#x$ and row $\#y$) to be `v` (any floating point number).

Notice that a call to this function is a slow (but easy and secure) way to write a pixel value. See section 2 page 8 for how to write pixels fast.

○Example

```
Fimage image; /* Needed Input of module */
int x,y;      /* Needed Inputs of module */

/* Put 0.0 in the pixel (x,y) */
if ((x < image->ncol) && (y < image->nrow))
        mw_plot_fimage(image,x,y,0.0));
else mwerror(ERROR,1,"Out of bounds !\n");
```

2.4 Color Float Images

You may want to use this format when you need to process color images with floating point precision (continuous scheme). Please notice that this format wastes a lot of memory and computational time.

2.4.1 The structure Cfimage

This memory type is not exactly the same as `Ccimage` (See section 2.2.1 page 22): the difference is not only about the RGB fields which are pointers to floating points values and not to unsigned char, but also about the *color model*. A color model is a specification of a 3D-coordinate system and a subspace within that system where each color is represented by a single point. Whatever the color model, a cfimage is alway made by three planes called `red`, `green` and `blue`. The significance of those planes is given by the value of the `model` field. The first plane `red` matches the first letter of the model's name (e.g. R for RGB model, H for HSI model), the second plane `green` matches the second letter of the model's name (e.g. G for RGB model, S for HSI model), and the third plane `blue` matches the third letter (e.g. B for RGB model, I for HSI model).

The implemented color models are

- `MODEL_RGB` Cartesian coordinate system Red, Green, Blue.
- `MODEL_YUV` YUV coordinate system (CCIR 601-1).
- `MODEL_HSI` HSI coordinate system (H is Hue, S is Saturation and I is Intensity or luminance).
- `MODEL_HSV` HSV coordinate system (H is Hue, S is Saturation and V is Value).

Be aware that a MegaWave2 module which takes a cfimage in input performs a statement likely to work for one color model only. One should checks the value of the `model` field before any statement.

```
typedef struct cfimage {
  int nrow;        /* Number of rows (dy) */
  int ncol;        /* Number of columns (dx) */
  int model;       /* Model of the colorimetric system */

  float *red;      /* The Red plane if model=MODEL_RGB (may be NULL) or Y/H   */
  float *green;    /* The Green plane if model=MODEL_RGB (may be NULL) or U/S */
  float *blue;     /* The Blue plane if model=MODEL_RGB (may be NULL) or V/I  */

  float scale;     /* Scale of the picture (should be 1 for original pict.) */
  char cmt[mw_cmtsize]; /* Comments */
  char name[mw_namesize]; /* Name of the image */

  /* Defines the signifiant part of the picture : */
  int firstcol;    /* index of the first col not affected by left side effect*/
  int lastcol;     /* index of the last col not affected by right side effect*/
  int firstrow;    /* index of the first row not aff. by upper side effect */
  int lastrow;     /* index of the last row not aff. by lower side effect */

} *Cfimage;
```

2.4.2 Related file (external) types

The list of the available formats is the following:

PMC_F PM format with three 8-bits planes, each plane being of float values. This format carries the comments field (`cmt`) of the memory object. It has been developed by the University of Pennsylvania, USA. An extension has been performed to record the `model` value. In the case of RGB model, the format is exactly the same as the original.

2.4.3 Functions Summary

The following is a description of all the functions related to the `Cfimage` type. The list is in alphabetical order. Conversion between memory models are not implemented as functions of the system library, but as modules (See Volume three: "MegaWave2 User's Modules Library").

○Name

mw_alloc_cfimage - Allocate the RGB planes

○Summary

Cfimage mw_alloc_cfimage(image,nrow,ncol)

Cfimage image;

int nrow, ncol;

○Description

This function allocates the RGB planes of a `Cfimage` structure previously created using `mw_new_cfimage`. The size of the image is given by `nrow` (number of rows or maximum range of y plus one) and `ncol` (number of columns or maximum range of x plus one). Pixels can be addressed after this call, if the allocation successed. There is no default value for the pixels.

Do not use this function if `image` has already an allocated plane: use the function `mw_change_cfimage` instead.

The function `mw_alloc_cfimage` returns `NULL` if not enough memory is available to allocate the RGB planes. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

○Example

```
Cfimage image=NULL; /* Internal use: no Input neither Output of module */

if  ( ((image = mw_new_cfimage()) == NULL) ||
      (mw_alloc_cfimage(image,256,256) == NULL) )
    mwerror(FATAL,1,"Not enough memory.\n");

/* Set pixel (0,1) to (0.0,0.0,0.0) */
image_>red[256] = image->green[256] = image->blue[256] = 0.0;
```

○Name

mw_change_cfimage - Change the size of the RGB planes

○Summary

Cfimage mw_change_cfimage(image, nrow, ncol)

Cfimage image;

int nrow, ncol;

○Description

This function changes the memory allocation of the RGB planes of a `Cfimage` structure, even if no previously memory allocation was done. The new size of the image is given by `nrow` (number of rows or maximum range of y plus one) and `ncol` (number of columns or maximum range of x plus one).

It can also create the structure if the input `image = NULL`. Therefore, this function can replace both `mw_new_cfimage` and `mw_alloc_cfimage`. It is the recommended function to set image dimension of input/output modules. Since the function can set the address of `image`, the variable must be set to the return value of the function (See example below).

The function `mw_change_cfimage` returns `NULL` if not enough memory is available to allocate the RGB planes. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

○Example

```
Cimage Output; /* Output of module */

Output = mw_change_cfimage(Output, 256, 256);
if (Output == NULL) mwerror(FATAL,1,"Not enough memory.\n");
```

○Name

mw_clear_cfimage - Clear the RGB planes

○Summary

void mw_clear_cfimage(image, r,g,b)

Cfimage image;

float r,g,b;

○Description

This function fills the cfimage `image` with the color given by the triplet `r,g,b`: all pixels will have this RGB value.

○Example

```
Cfimage image; /* Output of module */

image = mw_change_cfimage(image, 100, 100);
if (image == NULL) mwerror(FATAL,1,"Not enough memory.\n");
/* Set all pixels to (0.0,0.0,1.0) */
mw_clear_cfimage(image,0.0,0.0,1.0);
```

○Name

mw_copy_cfimage - Copy the pixel values of color image into another one

○Summary

void mw_copy_cfimage(in, out)

Cfimage in,out;

○Description

This function copies the content of the RGB planes of the image `in` into the RGB planes of the image `out`. The size of the two RGB planes must be the same.

The speed of this function depends to the C library implementation, but it is usually very fast (trying to do faster is a waste of time).

○Example

```
Cfimage G; /* Needed Input */
Cfimage F; /* Optional Output */

  if (F) {
    printf("F option is active: copy G in F\n");
    if ((F = mw_change_cfimage(F, G->nrow, G->ncol)) == NULL)
mwerror(FATAL,1,"Not enough memory.\n");
    else mw_copy_cfimage(G, F);
   }
  else  printf("F option is not active\n");
```

○Name

mw_delete_cfimage - Deallocate the RGB planes

○Summary

void mw_delete_cfimage(image)

Cfimage image;

○Description

This function deallocates the RGB planes of a `Cfimage` structure previously allocated using `mw_alloc_cfimage` or `mw_change_cfimage`, and the structure itself.

You should set `image = NULL` after this call since the address pointed by `image` is no longer valid.

○Example

```
Cfimage image=NULL; /* Internal use: no Input neither Output of module */

if  ( ((image = mw_new_cfimage()) == NULL) ||
      (mw_alloc_cfimage(image,256,256) == NULL) )
    mwerror(FATAL,1,"Not enough memory.\n");
.
.
.
mw_delete_cfimage(image);
image = NULL;
```

○Name

mw_draw_cfimage - Draw a line

○Summary

void mw_draw_cfimage(image, a0, b0, a1, b1, r, g, b)

Cfimage image;

int a0,b0,a1,b1;

float r,g,b;

○Description

This function draws in `image` a connected line between the pixel $(a0, b0)$ and the pixel $(a1, b1)$. The color of the line is defined by the triplet `r,g,b`.

○Example

```
Cfimage image; /* Output of module */

image = mw_change_cfimage(image, 100, 100);
if (image == NULL) mwerror(FATAL,1,"Not enough memory.\n");
/* Set all pixels to (0.0,0.0,0.0) */
mw_clear_cfimage(image,0.0,0.0,0.0);
/* Draw a diagonal line with color (1.0,0.0,0.0) */
mw_draw_cfimage(image,0,0,99,99,1.0,0.0,0.0);
```

○Name

mw_getdot_cfimage - Return the RGB value

○Summary

void mw_getdot_cfimage(image, x, y, r, g, b)

Cfimage image;

int x,y;

float *r,*g,*b;

○Description

This function returns the RGB value of the given `image` for the pixel (x, y) (column $\#x$ and row $\#y$). The RGB value consists of the triplet `*r,*g,*b`: `*r` (any floating point number) gives you the proportion of red, `*g` the proportion of green and `*b` the proportion of blue.

Notice that a call to this function is a slow (but easy and secure) way to read a pixel value. See section 2 page 8 for how to read pixels fast.

○Example

```
Cfimage image; /* Needed Input of module */
int x,y;       /* Needed Inputs of module */
float r,g,b;   /* Internal use */

if ((x < image->ncol) && (y < image->nrow))
  {
    mw_getdot_cfimage(image,x,y,&r,&g,&b);
    printf("image(%d,%d) = %d,%d,%d\n",x,y,r,g,b);
  }
else mwerror(ERROR,1,"Out of bounds !\n");
```

○Name

mw_new_cfimage - Create a new Cfimage

○Summary

Cfimage mw_new_cfimage();

○Description

This function creates a new `Cfimage` structure with empty RGB planes. No pixels can be addressed at this time. The RGB planes may be allocated using the function `mw_alloc_cfimage` or `mw_change_cfimage`.

Do not use this function for input/output of modules, since the MegaWave2 Compiler already created the structure for you if you need it (See Volume one: "MegaWave2 User's Guide"). Use instead the function `mw_change_cfimage`. Do not forget to deallocate the internal structures before the end of the module.

The function `mw_new_cfimage` returns `NULL` if not enough memory is available to create the structure. Your code should check this value to send an error message in the `NULL` case, and do appropriate statement.

○Example

```
Cfimage image=NULL; /* Internal use: no Input neither Output of module */

if  ( ((image = mw_new_cfimage()) == NULL) ||
      (mw_alloc_cfimage(image,256,256) == NULL) )
    mwerror(FATAL,1,"Not enough memory.\n");
```

○Name

mw_newtab_blue_cfimage - Create a bi-dimensional tab for the blue pixels of a Cfimage

○Summary

float ** mw_newtab_blue_cfimage(image)

Cfimage image;

○Description

This function creates a new bi-dimensional tab which allows an easy and fast access to the pixels' blue level. This tab is actually an one-dimensional tab of pointers, so that each pointer points to the beginning of a line in the blue plane of the given image.

This function must be called after the blue plane has been allocated, using for example one of the functions `mw_new_cfimage`, `mw_alloc_cfimage` or `mw_change_cfimage`. After that, if the blue plane allocation is changed (by e.g. `mw_change_cfimage` or `mw_delete_cfimage`), the tab is no longer valid and must be deleted using `free(tab)`.

Ones the tab has been correctly created, is it possible to read or to write the blue value of the pixel (x, y) (x being an index for column and y for row) using `tab[y][x]`.

Red and green pixels' value can be accessed with such a tab using the corresponding functions `mw_newtab_red_cfimage` and `mw_newtab_green_cfimage`.

○Example

```
Cfimage image; /* Needed Input of module (RGB planes already allocated and filled) */
int x,y;      /* Needed Input of module */
float **red,**green,**blue;

red =  mw_newtab_red_cfimage(image);
if (red==NULL) mwerror(FATAL,1,"Not enough memory\n");
green =  mw_newtab_green_cfimage(image);
if (green==NULL) mwerror(FATAL,1,"Not enough memory\n");
blue =  mw_newtab_blue_cfimage(image);
if (blue==NULL) mwerror(FATAL,1,"Not enough memory\n");

/* Put black color in the pixel (x,y) */
if ((x < image->ncol) && (y < image->nrow))
        red[y][x] = green[y][x] = blue[y][x] = 0.0;
else mwerror(ERROR,1,"Out of bounds !\n");

free(blue); free(green); free(red);
```

○Name

mw_newtab_green_cfimage - Create a bi-dimensional tab for the green pixels of a Cfimage

○Summary

float ** mw_newtab_green_cfimage(image)

Cfimage image;

○Description

This function creates a new bi-dimensional tab which allows an easy and fast access to the pixels' green level. This tab is actually an one-dimensional tab of pointers, so that each pointer points to the beginning of a line in the green plane of the given image.

This function must be called after the green plane has been allocated, using for example one of the functions `mw_new_cfimage`, `mw_alloc_cfimage` or `mw_change_cfimage`. After that, if the green plane allocation is changed (by e.g. `mw_change_cfimage` or `mw_delete_cfimage`), the tab is no longer valid and must be deleted using `free(tab)`.

Ones the tab has been correctly created, is it possible to read or to write the green value of the pixel (x, y) (x being an index for column and y for row) using `tab[y][x]`.

Red and blue pixels' value can be accessed with such a tab using the corresponding functions `mw_newtab_red_cfimage` and `mw_newtab_blue_cfimage`.

○Example

```
Cfimage image; /* Needed Input of module (RGB planes already allocated and filled) */
int x,y;      /* Needed Input of module */
float **red,**green,**blue;

red =  mw_newtab_red_cfimage(image);
if (red==NULL) mwerror(FATAL,1,"Not enough memory\n");
green =  mw_newtab_green_cfimage(image);
if (green==NULL) mwerror(FATAL,1,"Not enough memory\n");
blue =  mw_newtab_blue_cfimage(image);
if (blue==NULL) mwerror(FATAL,1,"Not enough memory\n");

/* Put black color in the pixel (x,y) */
if ((x < image->ncol) && (y < image->nrow))
        red[y][x] = green[y][x] = blue[y][x] = 0.0;
else mwerror(ERROR,1,"Out of bounds !\n");

free(blue); free(green); free(red);
```

○Name

mw_newtab_red_cfimage - Create a bi-dimensional tab for the red pixels of a Cfimage

○Summary

float ** mw_newtab_red_cfimage(image)

Cfimage image;

○Description

This function creates a new bi-dimensional tab which allows an easy and fast access to the pixels' red level. This tab is actually an one-dimensional tab of pointers, so that each pointer points to the beginning of a line in the red plane of the given image.

This function must be called after the red plane has been allocated, using for example one of the functions `mw_new_cfimage`, `mw_alloc_cfimage` or `mw_change_cfimage`. After that, if the red plane allocation is changed (by e.g. `mw_change_cfimage` or `mw_delete_cfimage`), the tab is no longer valid and must be deleted using `free(tab)`.

Ones the tab has been correctly created, is it possible to read or to write the red value of the pixel (x, y) (x being an index for column and y for row) using `tab[y][x]`.

Green and blue pixels' value can be accessed with such a tab using the corresponding functions `mw_newtab_green_cfimage` and `mw_newtab_blue_cfimage`.

○Example

```
Cfimage image; /* Needed Input of module (RGB planes already allocated and filled) */
int x,y;      /* Needed Input of module */
float **red,**green,**blue;

red =  mw_newtab_red_cfimage(image);
if (red==NULL) mwerror(FATAL,1,"Not enough memory\n");
green =  mw_newtab_green_cfimage(image);
if (green==NULL) mwerror(FATAL,1,"Not enough memory\n");
blue =  mw_newtab_blue_cfimage(image);
if (blue==NULL) mwerror(FATAL,1,"Not enough memory\n");

/* Put black color in the pixel (x,y) */
if ((x < image->ncol) && (y < image->nrow))
        red[y][x] = green[y][x] = blue[y][x] = 0.0;
else mwerror(ERROR,1,"Out of bounds !\n");

free(blue); free(green); free(red);
```

○Name

mw_plot_cfimage - Set the RGB value

○Summary

void mw_plot_cfimage(image, x, y, r, g, b)

Cfimage image;

int x,y;

float r,g,b;

○Description

This function set the RGB value of the given `image` for the pixel (x, y) (column $\#x$ and row $\#y$) to be the triplet `r,g,b`: `r` (a floating point number) gives you the proportion of red, `g` the proportion of green and `b` the proportion of blue.

Notice that a call to this function is a slow (but easy and secure) way to write a pixel value. See section 2 page 8 for how to write pixels fast.

○Example

```
Cfimage image; /* Needed Input of module */
int x,y;      /* Needed Inputs of module */

/* Put color (0.0,0.0,0.0) in the pixel (x,y) */
if ((x < image->ncol) && (y < image->nrow))
        mw_plot_cfimage(image,x,y,0.0,0.0,0.0));
else mwerror(ERROR,1,"Out of bounds !\n");
```

3 Movies

A movie is a succession of images. In MegaWave2, it is implemented as a chain of images: you may notice that all types of images have the fields `previous` and `next`(see section 2). Normally set to `NULL`, these fields point to the previous image and to the next image respectively, when the image is part of a movie. The first image of the movie has a `NULLprevious` field and the last image of the movie has a `NULLnext` field.

A movie structure is basically a pointer to the first image. For each image structure corresponds a movie structure.

3.1 Char movies

The *Char Movie* memory type corresponds to movies where each images are of the `Cimage` memory type. The use of this memory type is strongly recommended, since other movies types waste a lot of memory and computational time.

3.1.1 The structure Cmovie

Beginners should only focus on the field `first` of this structure: if `movie` is of `Cmovie` type, then `movie->first` is of `Cimage` type and it is the first image of the movie; `movie->first->next` is the second image, etc.

```
typedef struct cmovie {
  float scale;     /* Scale (time-domain) of the movie (should be 1) */
  char cmt[mw_cmtsize]; /* Comments */
  char name[mw_namesize]; /* Name of the image */
  Cimage first;    /* Pointer to the first image */
} *Cmovie;
```

3.1.2 Related file (external) types

The way MegaWave2 records movies is the following: each image is recorded in a separate file, using one of the external types available for the corresponding type Cimage (see section 2.1.2 for more information). Let suppose that the external name of the movie object is `movie`. Then, MegaWave2 creates a MegaWave2 Data Ascii file named `movie` with a `def Cmovie` area. In this area is listed the name of the image files, following the order given by the sequence of images. By changing the order of two file names, you change the order of the images in the sequence (i.e. the name of the file is not meaningful by itself, for example the image file name `movie_002` may not be the second image of the sequence). There is no limitation for the number of images, up to the available memory.

Note: there is an old format, which is still recognized for backward compatibility. Is this old format, the file name `movie` is empty and the name of each image file is meaningful (e.g. `movie_002` is always the second image of the sequence). In that case, no more than 999 images per movie can be recorded.

3.1.3 Functions Summary

The following is a description of all the functions related to the `Cmovie` type. The list is in alphabetical order.

○Name

mw_change_cmovie - Define the movie structure, if not defined

○Summary

Cmovie mw_change_cmovie(movie)

Cmovie movie;

○Description

This function returns a movie structure if the input `movie = NULL`. It is provided despite the `mw_new_cmovie` function for global coherence with other memory types.

The function `mw_change_cmovie` returns `NULL` if not enough memory is available to define the structure. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

Since the MegaWave2 compiler allocates structures for input and output objects (See Volume one: "MegaWave2 User's Guide"), this function is normally used only for internal objects. Do not forget to deallocate the internal structures before the end of the module.

Images have to be allocated using the appropriate functions (See the example below).

○Example

```
Cmovie movie=NULL; /* Internal use: no Input neither Output of module */

movie = mw_change_cmovie(movie);
if (movie == NULL) mwerror(FATAL,1,"Not enough memory.\n");
...
```

(End of this example as for the `mw_new_cmovie` function).

○Name

mw_delete_cmovie - Deallocate all the movie

○Summary

void mw_delete_cmovie(movie)

Cmovie movie;

○Description

This function deallocates all the memory used by a `Cmovie` structure: it deallocates the gray plane of all images, the image structures and the movie structure itself.

You should set `movie = NULL` after this call since the address pointed by `movie` is no longer valid.

○Example

See the example of the `mw_new_cmovie` function: when a memory allocation fails for `mw_change_cimage`, all the previously memory allocations are freed by the call to `mw_delete_cmovie(movie)`.

○Name

mw_new_cmovie - Create a new Cmovie

○Summary

Cmovie mw_new_cmovie();

○Description

This function creates a new `Cmovie` structure. It returns `NULL` if not enough memory is available to create the structure. Your code should check this value to send an error message in the `NULL` case, and do appropriate statement.

Since the MegaWave2 compiler allocates structures for input and output objects (See Volume one: "MegaWave2 User's Guide"), this function is normally used only for internal objects. Do not forget to deallocate the internal structures before the end of the module.

Images have to be allocated using the appropriate functions (See the example below).

○Example

```
/* Create a movie with 10 images of size (100,100) */

Cmovie movie;  /* Internal use: no Input neither Output of module */
Cimage image,image_prev; /* Internal use */

movie = mw_new_cmovie();
if (movie == NULL) mwerror(FATAL,1,"Not enough memory.\n");
for (l=1;l<=10;l++)
 {
   if ((image = mw_change_cimage(NULL,100,100)) == NULL)
    {
      mw_delete_cmovie(movie);
      mwerror(FATAL,1,"Not enough memory.");
    }
    if (l == 1) movie->first = image;
    else
      {
        image_prev->next = image;
        image->previous = image_prev;
       }
     image_prev = image;
 }
```

3.2 Color Char movies

The *Color Char Movie* memory type corresponds to movies where each images are of the `Ccimage` memory type. Use this memory type each time you have to process color movies. As in the Char Movies case, the use of this format instead of the corresponding floating point format (`Cfmovie`) is strongly recommended.

3.2.1 The structure Ccmovie

Beginners should focus on the field `first` only of this structure: if `movie` is of `Ccmovie` type, then `movie->first` is of `Ccimage` type and it is the first image of the movie; `movie->first->next` is the second image, etc.

```
typedef struct ccmovie {
  float scale;     /* Scale (time-domain) of the movie (should be 1) */
  char cmt[mw_cmtsize]; /* Comments */
  char name[mw_namesize]; /* Name of the image */
  Ccimage first;    /* Pointer to the first image */
} *Ccmovie;
```

3.2.2 Related file (external) types

The way MegaWave2 records movies is the following: each image is recorded in a separate file, using one of the external types available for the corresponding type Ccimage (see section 2.2.2 for more information). Let suppose that the external name of the movie object is `movie`. Then, MegaWave2 creates a MegaWave2 Data Ascii file named `movie` with a `def CCmovie` area. In this area is listed the name of the image files, following the order given by the sequence of images. By changing the order of two file names, you change the order of the images in the sequence (i.e. the name of the file is not meaningful by itself, for example the image file name `movie_002` may not be the second image of the sequence). There is no limitation for the number of images, up to the available memory.

Note: there is an old format, which is still recognized for backward compatibility. Is this old format, the file name `movie` is empty and the name of each image file is meaningful (e.g. `movie_002` is always the second image of the sequence). In that case, no more than 999 images per movie can be recorded.

3.2.3 Functions Summary

The following is a description of all the functions related to the `Ccmovie` type. The list is in alphabetical order.

○Name

mw_change_ccmovie - Define the movie structure, if not defined

○Summary

Ccmovie mw_change_ccmovie(movie)

Ccmovie movie;

○Description

This function returns a movie structure if the input `movie = NULL`. It is provided despite the `mw_new_ccmovie` function for global coherence with other memory types.

The function `mw_change_ccmovie` returns `NULL` if not enough memory is available to define the structure. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

Since the MegaWave2 compiler allocates structures for input and output objects (See Volume one: "MegaWave2 User's Guide"), this function is normally used only for internal objects. Do not forget to deallocate the internal structures before the end of the module.

Images have to be allocated using the appropriate functions (See the example below).

○Example

```
Ccmovie movie=NULL; /* Internal use: no Input neither Output of module */

movie = mw_change_ccmovie(movie);
if (movie == NULL) mwerror(FATAL,1,"Not enough memory.\n");
...
```

(End of this example as for the `mw_new_ccmovie` function).

○Name

mw_delete_ccmovie - Deallocate all the movie

○Summary

void mw_delete_ccmovie(movie)

Ccmovie movie;

○Description

This function deallocates all the memory used by a `Ccmovie` structure: it deallocates the color planes of all images, the image structures and the movie structure itself.

You should set `movie = NULL` after this call since the address pointed by `movie` is no longer valid.

○Example

See the example of the `mw_new_ccmovie` function: when a memory allocation fails for `mw_change_ccimage`, all the previously memory allocations are freed by the call to `mw_delete_ccmovie(movie)`.

○Name

mw_new_ccmovie - Create a new Ccmovie

○Summary

Ccmovie mw_new_ccmovie();

○Description

This function creates a new `Ccmovie` structure. It returns `NULL` if not enough memory is available to create the structure. Your code should check this value to send an error message in the `NULL` case, and do appropriate statement.

Since the MegaWave2 compiler allocates structures for input and output objects (See Volume one: "MegaWave2 User's Guide"), this function is normally used only for internal objects. Do not forget to deallocate the internal structures before the end of the module.

Images have to be allocated using the appropriate functions (See the example below).

○Example

```
/* Create a movie with 10 images of size (100,100) */

Ccmovie movie;  /* Internal use: no Input neither Output of module */
Ccimage image,image_prev; /* Internal use */

movie = mw_new_ccmovie();
if (movie == NULL) mwerror(FATAL,1,"Not enough memory.\n");
for (l=1;l<=10;l++)
 {
   if ((image = mw_change_ccimage(NULL,100,100)) == NULL)
    {
      mw_delete_ccmovie(movie);
      mwerror(FATAL,1,"Not enough memory.");
    }
    if (l == 1) movie->first = image;
    else
      {
        image_prev->next = image;
        image->previous = image_prev;
       }
     image_prev = image;
 }
```

3.3 Float movies

The *Float Movie* memory type corresponds to movies where each images are of the `Fimage` memory type. The use of this memory type is discouraged, since it wastes a lot of memory and computational time. Use it when you must process data using floating point arithmetic, and when you cannot lose precision by converting the output to integer values.

3.3.1 The structure Fmovie

Beginners should focus on the field `first` only of this structure: if `movie` is of `Fmovie` type, then `movie->first` is of `Fimage` type and it is the first image of the movie; `movie->first->next` is the second image, etc.

```
typedef struct fmovie {
  float scale;     /* Scale (time-domain) of the movie (should be 1) */
  char cmt[mw_cmtsize]; /* Comments */
  char name[mw_namesize]; /* Name of the image */
  Fimage first;    /* Pointer to the first image */
} *Fmovie;
```

3.3.2 Related file (external) types

The way MegaWave2 records movies is the following: each image is recorded in a separate file, using one of the external types available for the corresponding type Fimage (see section 2.3.2 for more information). Let suppose that the external name of the movie object is `movie`. Then, MegaWave2 creates a MegaWave2 Data Ascii file named `movie` with a `def Fmovie` area. In this area is listed the name of the image files, following the order given by the sequence of images. By changing the order of two file names, you change the order of the images in the sequence (i.e. the name of the file is not meaningful by itself, for example the image file name `movie_002` may not be the second image of the sequence). There is no limitation for the number of images, up to the available memory.

Note: there is an old format, which is still recognized for backward compatibility. Is this old format, the file name `movie` is empty and the name of each image file is meaningful (e.g. `movie_002` is always the second image of the sequence). In that case, no more than 999 images per movie can be recorded.

3.3.3 Functions Summary

The following is a description of all the functions related to the `Fmovie` type. The list is in alphabetical order.

○Name

mw_change_fmovie - Define the movie structure, if not defined

○Summary

Fmovie mw_change_fmovie(movie)

Fmovie movie;

○Description

This function returns a movie structure if the input `movie = NULL`. It is provided despite the `mw_new_fmovie` function for global coherence with other memory types.

The function `mw_change_fmovie` returns `NULL` if not enough memory is available to define the structure. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

Since the MegaWave2 compiler allocates structures for input and output objects (See Volume one: "MegaWave2 User's Guide"), this function is normally used only for internal objects. Do not forget to deallocate the internal structures before the end of the module.

Images have to be allocated using the appropriate functions (See the example below).

○Example

```
Fmovie movie=NULL; /* Internal use: no Input neither Output of module */

movie = mw_change_fmovie(movie);
if (movie == NULL) mwerror(FATAL,1,"Not enough memory.\n");
...
```

(End of this example as for the `mw_new_fmovie` function).

○Name

mw_delete_fmovie - Deallocate all the movie

○Summary

void mw_delete_fmovie(movie)

Fmovie movie;

○Description

This function deallocates all the memory used by a `Fmovie` structure: it deallocates the gray plane of all images, the image structures and the movie structure itself.

You should set `movie = NULL` after this call since the address pointed by `movie` is no longer valid.

○Example

See the example of the `mw_new_fmovie` function: when a memory allocation fails for `mw_change_fimage`, all the previously memory allocations are freed by the call to `mw_delete_fmovie(movie)`.

○Name

mw_new_fmovie - Create a new Fmovie

○Summary

Fmovie mw_new_fmovie();

○Description

This function creates a new `Fmovie` structure. It returns `NULL` if not enough memory is available to create the structure. Your code should check this value to send an error message in the `NULL` case, and do appropriate statement.

Since the MegaWave2 compiler allocates structures for input and output objects (See Volume one: "MegaWave2 User's Guide"), this function is normally used only for internal objects. Do not forget to deallocate the internal structures before the end of the module.

Images have to be allocated using the appropriate functions (See the example below).

○Example

```
/* Create a movie with 10 images of size (100,100) */

Fmovie movie;  /* Internal use: no Input neither Output of module */
Fimage image,image_prev; /* Internal use */

movie = mw_new_fmovie();
if (movie == NULL) mwerror(FATAL,1,"Not enough memory.\n");
for (l=1;l<=10;l++)
 {
   if ((image = mw_change_fimage(NULL,100,100)) == NULL)
    {
      mw_delete_fmovie(movie);
      mwerror(FATAL,1,"Not enough memory.");
    }
    if (l == 1) movie->first = image;
    else
      {
        image_prev->next = image;
        image->previous = image_prev;
       }
     image_prev = image;
 }
```

3.4 Color Float movies

The *Color Float Movie* memory type corresponds to movies where each images are of the `Cfimage` memory type. The use of this memory type is discouraged, since it wastes a lot of memory and computational time. Use it when you must process data using floating point arithmetic, and when you cannot lose precision by converting the output to integer values.

3.4.1 The structure Cfmovie

Beginners should focus on the field `first` only of this structure: if `movie` is of `Cfmovie` type, then `movie->first` is of `Cfimage` type and it is the first image of the movie; `movie->first->next` is the second image, etc.

```
typedef struct cfmovie {
  float scale;     /* Scale (time-domain) of the movie (should be 1) */
  char cmt[mw_cmtsize]; /* Comments */
  char name[mw_namesize]; /* Name of the image */
  Cfimage first;    /* Pointer to the first image */
} *Cfmovie;
```

3.4.2 Related file (external) types

The way MegaWave2 records movies is the following: each image is recorded in a separate file, using one of the external types available for the corresponding type Cfimage (see section 2.4.2 for more information). Let suppose that the external name of the movie object is `movie`. Then, MegaWave2 creates a MegaWave2 Data Ascii file named `movie` with a `def Cfmovie` area. In this area is listed the name of the image files, following the order given by the sequence of images. By changing the order of two file names, you change the order of the images in the sequence (i.e. the name of the file is not meaningful by itself, for example the image file name `movie_002` may not be the second image of the sequence). There is no limitation for the number of images, up to the available memory.

Note: there is an old format, which is still recognized for backward compatibility. Is this old format, the file name `movie` is empty and the name of each image file is meaningful (e.g. `movie_002` is always the second image of the sequence). In that case, no more than 999 images per movie can be recorded.

3.4.3 Functions Summary

The following is a description of all the functions related to the `Cfmovie` type. The list is in alphabetical order.

○Name

mw_change_cfmovie - Define the movie structure, if not defined

○Summary

Cfmovie mw_change_cfmovie(movie)

Cfmovie movie;

○Description

This function returns a movie structure if the input `movie = NULL`. It is provided despite the `mw_new_cfmovie` function for global coherence with other memory types.

The function `mw_change_cfmovie` returns `NULL` if not enough memory is available to define the structure. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

Since the MegaWave2 compiler allocates structures for input and output objects (See Volume one: "MegaWave2 User's Guide"), this function is normally used only for internal objects. Do not forget to deallocate the internal structures before the end of the module.

Images have to be allocated using the appropriate functions (See the example below).

○Example

```
Cfmovie movie=NULL; /* Internal use: no Input neither Output of module */

movie = mw_change_cfmovie(movie);
if (movie == NULL) mwerror(FATAL,1,"Not enough memory.\n");
...
```

(End of this example as for the `mw_new_cfmovie` function).

○Name

mw_delete_cfmovie - Deallocate all the movie

○Summary

void mw_delete_cfmovie(movie)

Cfmovie movie;

○Description

This function deallocates all the memory used by a `Cfmovie` structure: it deallocates the color planes of all images, the image structures and the movie structure itself.

You should set `movie = NULL` after this call since the address pointed by `movie` is no longer valid.

○Example

See the example of the `mw_new_cfmovie` function: when a memory allocation fails for `mw_change_cfimage`, all the previously memory allocations are freed by the call to `mw_delete_cfmovie(movie)`.

○Name

mw_new_cfmovie - Create a new Cfmovie

○Summary

Cfmovie mw_new_cfmovie();

○Description

This function creates a new `Cfmovie` structure. It returns `NULL` if not enough memory is available to create the structure. Your code should check this value to send an error message in the `NULL` case, and do appropriate statement.

Since the MegaWave2 compiler allocates structures for input and output objects (See Volume one: "MegaWave2 User's Guide"), this function is normally used only for internal objects. Do not forget to deallocate the internal structures before the end of the module.

Images have to be allocated using the appropriate functions (See the example below).

○Example

```
/* Create a movie with 10 images of size (100,100) */

Cfmovie movie;  /* Internal use: no Input neither Output of module */
Cfimage image,image_prev; /* Internal use */

movie = mw_new_cfmovie();
if (movie == NULL) mwerror(FATAL,1,"Not enough memory.\n");
for (l=1;l<=10;l++)
 {
   if ((image = mw_change_cfimage(NULL,100,100)) == NULL)
    {
      mw_delete_cfmovie(movie);
      mwerror(FATAL,1,"Not enough memory.");
    }
    if (l == 1) movie->first = image;
    else
      {
        image_prev->next = image;
        image->previous = image_prev;
       }
     image_prev = image;
 }
```

4 Signals

We call signal a one-dimensional sequence of scalars. Signals may be used to represent various kind of physical data (such as sound) , as well as mathematical data (e.g. impulse response of filters, vectors, ...).

Notice that at this time, only signals of floating points values are implemented.

4.1 Float signals

The *Float Signals* memory type is used to represent one-dimensional sequences of floating points values.

4.1.1 The structure Fsignal

Beginners should only focus on the first two fields of this structure:

```
typedef struct fsignal {
  int size;        /* Number of samples */
  float *values;   /* The samples */

  float scale;     /* Scale of the signal */
  float shift;     /* shifting of the signal with respect to zero */
  float gain;      /* Gain of the signal given by the digitalization process */
  float sgrate;    /* Sampling rate given by the digitalization process */
  int bpsample;    /* Number of bits per sample for audio drivers */

  char cmt[mw_cmtsize]; /* Comments */
  char name[mw_namesize]; /* Name of the image */

  /* Defines the signifiant part of the signal : */
  int firstp;     /* index of the first point not aff. by left side effect */
  int lastp;      /* index of the last point not aff. by right side effect */
  float param;    /* distance between two succesive uncorrelated points */

} *Fsignal;
```

The field `size` gives the number of samples loaded in the signal. Do not change by yourself the content of this field: the size of the signal has to be modified using functions of the library only (see section 4.1.3 page 78).

The field `values` is an array which gives the value of each sample: if `signal` is a variable of `Fsignal` type, `signal->values[0]` is the first sample of the signal, `signal->values[1]` the second, and so one up to the last sample `signal->values[signal->size-1]`.

4.1.2 Related file (external) types

The list of the available formats is the following:

1. `"A_FSIGNAL"` MegaWave2 Data Ascii format with a `def fsignal` area. This area includes the value of the different fields of the object, as `comments, scale, shift, ...` and at the end the

samples of the signal. Since this format uses Ascii encoding, you may read or modify the file just by editing it using a text editor. It can also be plotted using the standard tool `gnuplot`.

2. `"WAVE_PCM"` Microsoft's RIFF WAVE sound file format with PCM encoding. Use this format to perform sound and speech processing with MegaWave2. Stereo inputs are converted to mono when loaded into a `Fsignal`. Since this format performs bit-encoding, on any output `Fsignal` variables you should set the field `bpsample` to the number of bits you want the data to be saved. Default value is $8\times$`sizeof(float)` (on most architectures 32), because this matches the size of the samples in the `Fsignal` structure. However, this value leads to strange results on some audio drivers. If you plan to send the signal on a audio driver, recommended numbers of bits are 16 (signed word) or 8 (signed char). Take care to format your data to fit the corresponding range before playing the file ($[-32768, +32767]$ for signed word and $[-128, +127]$ for signed char) or you will not get the expected sound. Another important field to get the right result is `sgrate`, where you have to set the sample rate in Hz that is, the number of samples per second.

4.1.3 Functions Summary

The following is a description of all the functions related to the `Fsignal` type. The list is in alphabetical order.

○Name

mw_alloc_fsignal - Allocate the array of values

○Summary

Fsignal mw_alloc_fsignal(signal,n)

Fsignal signal;

int n;

○Description

This function allocates the array `values` of a `Fsignal` structure previously created using `mw_new_fsignal`. The size of the signal is given by `n`, it corresponds to the number of samples.

Values can be addressed after this call, if the allocation successed. There is no default values.

Do not use this function if `signal` has already an allocated array: use the function `mw_change_fsignal` instead.

The function `mw_alloc_fsignal` returns `NULL` if not enough memory is available to allocate the array. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

○Example

```
Fsignal signal=NULL; /* Internal use: no Input neither Output of module */
int i;

/* Create a signal with 1000 samples */
if  ( ((signal = mw_new_fsignal()) == NULL) ||
      (mw_alloc_fsignal(signal,1000) == NULL) )
    mwerror(FATAL,1,"Not enough memory.\n");

/* Set the sample #i to the value i */
for (i=0;i<signal->size;i++) signal->values[i] = i;
```

○Name

mw_change_fsignal - Change the size of the array of values

○Summary

Fsignal mw_change_fsignal(signal, n)

Fsignal signal;

int n;

○Description

This function changes the memory allocation of the array `values` of a `Fsignal` structure, even if no previously memory allocation was done. The new size of the signal is given by `n`, it corresponds to the number of samples.

The function `mw_change_fsignal` can also create the structure if the input `signal = NULL`. Therefore, this function can replace both `mw_new_fsignal` and `mw_alloc_fsignal`. It is the recommended function to set signal size of input/output modules. Since the function can set the address of `signal`, the variable must be set to the return value of the function (See example below).

The function `mw_change_fsignal` returns `NULL` if not enough memory is available to allocate the array. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

○Example

```
Fsignal Output; /* Output of module */

/* Set the size of the signal to be 1000 */
Output = mw_change_fsignal(Output, 1000);
if (Output == NULL) mwerror(FATAL,1,"Not enough memory.\n");
```

○Name

mw_clear_fsignal - Clear all values

○Summary

void mw_clear_fsignal(signal, v)

Fsignal signal;

float v;

○Description

This function fills the fsignal `signal` with the value given by v: all samples will have the value v.

○Example

```
Fsignal signal; /* Output of module */

signal = mw_change_fsignal(signal, 1000);
if (signal == NULL) mwerror(FATAL,1,"Not enough memory.\n");
/* Set all samples to 0.0 */
mw_clear_fsignal(signal,0.0);
```

○Name

mw_copy_fsignal - Copy a signal into another one

○Summary

void mw_copy_fsignal(in, out)

Fsignal in,out;

○Description

This function copies the header and the content of the array `values` of the signal `in` into the corresponding fields of the signal `out`. The size of the two signals must be the same (this implies the `out` signal to be allocated).

The speed of this function depends to the C library implementation, but it is usually very fast (trying to do faster is a waste of time).

○Example

```
Fsignal G; /* Needed Input */
Fsignal F; /* Optional Output */

  if (F) {
    printf("F option is active: copy G in F\n");
    if ((F = mw_change_fsignal(F, G->size)) == NULL)
        mwerror(FATAL,1,"Not enough memory.\n");
    else mw_copy_fsignal(G, F);
   }
  else  printf("F option is not active\n");
```

○Name

mw_delete_fsignal - Deallocate the signal

○Summary

void mw_delete_fsignal(signal)

Fsignal signal;

○Description

This function deallocates the array `values` of a `Fsignal` structure previously allocated using `mw_alloc_fsignal` or `mw_change_fsignal`, and the structure itself.

You should set `signal = NULL` after this call since the address pointed by `signal` is no longer valid.

○Example

```
Fsignal signal=NULL; /* Internal use: no Input neither Output of module */

if  ( ((signal = mw_new_fsignal()) == NULL) ||
      (mw_alloc_fsignal(signal,1000) == NULL) )
    mwerror(FATAL,1,"Not enough memory.\n");
.
.
.
mw_delete_fsignal(signal);
signal = NULL;
```

○Name

mw_new_fsignal - Create a new Fsignal

○Summary

Fsignal mw_new_fsignal();

○Description

This function creates a new `Fsignal` structure with an empty array `values`. No samples can be addressed at this time. The array `values` should be allocated using the function `mw_alloc_fsignal` or `mw_change_fsignal`.

Do not use this function for input/output of modules, since the MegaWave2 Compiler already created the structure for you if you need it (See Volume one: "MegaWave2 User's Guide"). Use instead the function `mw_change_fsignal`. Do not forget to deallocate the internal structures before the end of the module.

The function `mw_new_fsignal` returns `NULL` if not enough memory is available to create the structure. Your code should check this value to send an error message in the `NULL` case, and do appropriate statement.

○Example

```
Fsignal signal=NULL; /* Internal use: no Input neither Output of module */

if  ( ((signal = mw_new_fsignal()) == NULL) ||
      (mw_alloc_fsignal(signal,1000) == NULL) )
    mwerror(FATAL,1,"Not enough memory.\n");
```

5 Wavelets

The wavelet memory types are used to represent the result of a wavelet transform applied to some data. The data can be a signal, in this case the operation is called a one-dimensional wavelet tranform, or it can be an image. In that case, the operation is called a two-dimensional wavelet transform. Operations on data of higher dimension are not supported at this time.

A wavelet transform is a time-scale operator: it adds therefore one dimension to the data (the scale). The meaning of the wavelet coefficients recorded into a wavelet-type variable depends to the choice of the discretization. The finest one is known as the *continuous wavelet transform*: several voices per octave can be computed for the scale. The *orthogonal (or biorthogonal) wavelet transform* allows to decompose the data into an orthogonal (or biorthogonal) basis: a wavelet coefficient corresponds to a scalar product. In this case, only one voice per octave is computed and a decimation is achieved on the time (or space) domain. The *dyadic wavelet transform* computes also only one voice per octave, but without decimation along the time axis. It corresponds to a decomposition into wavelets which generate a *frame*. It is often used to obtain a translation-invariant representation, from which the *wavelet maxima representation* can be deduced.

5.1 One-dimensional wavelet

The *One-dimensional wavelet* memory type is used to represent the result of a wavelet transform applied to a signal.

5.1.1 The structure Wtrans1d

The C structure is the following:

```
  char cmt[mw_cmtsize]; /* Comments */
  char name[mw_namesize]; /* Name of the wtrans1d */

  int type;  /* Type of the wtrans1d performed */
  int edges; /* Type of the edges statments */
  char filter_name[mw_namesize][mw_max_nfilter_1d];   /* Filters used */
  int size; /* Size of the original signal */

  int nlevel;   /* Number of levels (octave) for this decomposition */
  int nvoice;   /* Number of voices per octave for this decomposition */
  int complex;  /* 1 if the wavelet is complex that is, if P[][] is used */
  int nfilter;  /* Number of filters used to compute the decomposition */

  Fsignal A[mw_max_nlevel+1][mw_max_nvoice]; /* Average or low-pass signals */
  Fsignal AP[mw_max_nlevel+1][mw_max_nvoice];/* Phase of the average */
  Fsignal D[mw_max_nlevel+1][mw_max_nvoice]; /* Detail or wavelet coefficients*/
  Fsignal DP[mw_max_nlevel+1][mw_max_nvoice];/* Phase of the Detail */

} *Wtrans1d;
```

The first two fields of this structure is well known by the reader. The field `type` records the type of the wavelet transform used. Its value can be:

- `mw_orthogonal` : orthogonal wavelet transform;

- `mw_biorthogonal` : biorthogonal wavelet transform;
- `mw_dyadic` : dyadic wavelet transform;
- `mw_continuous` : continuous wavelet transform.

The field `edges` gives the type of the edges statment used to compute the transformation. Indeed, since it is implemented as a bank of convolution products, errors occur near the borders if no special statment is performed. This field can have the following values:

- `mw_edges_zeropad` : the signal is zero-padded (no special statment);
- `mw_edges_periodic` : the signal is made periodic;
- `mw_edges_mirror` : the signal is padded by mirror effect (avoid first-order discontinuities);
- `mw_edges_wadapted` : special border functions are added to the wavelets (wavelets on the interval).

The field `filter_name` is an array of strings, where the names of the filters used for the decomposition are put. The number of filters is put into the field `nfilter`. This number and the meaning of each filter depend to the wavelet type.

The field `size` contains the size of the original signal, which is put into `average[0][0]` (see below). The field `nlevel` is the number of levels used in this decomposition; it corresponds to the number of octaves; `nvoice` is the number of voices per octave. The field `complex` is set to 1 when the wavelet used has complex values, 0 elsewhere.

The result of the wavelet decomposition is put into two two-dimensional arrays of signals called `A` and `D`: `A[l][v]` for $l = 0 \ldots$ `nlevel` and for $v = 0 \ldots \text{nvoice} - 1$ is the low-pass signal at the octave l and at the voice v, that is the signal at the scale $2^{(l+v/\texttt{nvoice})}$. The signal `A[0][0]` is the original signal, `A[0][1]` is the smoothed signal at the scale $2^{\frac{1}{\texttt{nvoice}}}$, etc. `D[l][v]` for $l = 0 \ldots$ `nlevel` and for $v = 0 \ldots$ `nvoice` $- 1$ is the band-pass (or detail) signal at the octave l and at the voice v, that is the wavelet coefficients signal at the scale $2^{(l+v/\texttt{nvoice})}$. The signal `D[0][0]` is unused.

When the wavelet is complex, the fields `A` and `D` represent the modulus values only; the phase values is put in the fields `AP` and `DP`.

5.1.2 Related file (external) types

The list of the available formats is the following:

1. `"A_WTRANS1D"` MegaWave2 Data Ascii format with a `def Wtrans1d` area. This area includes the value of the different fields of the object, as `comments, type, edges, ....` The values of the wavelet coefficients are not recorded in this file, but in a set of `Fimage` objects. Let be `wavelet` the name of the object. The names of these image files are, for `<j>` the level number (octave) and `<v>` the voice number,
   - `wavelet_<j>_A.wtrans1d` Average field of the object (voice 0);
   - `wavelet_<j>.<v>_A.wtrans1d` Average field of the object (voice > 0);
   - `wavelet_<j>_AP.wtrans1d` Phase of the Average field of the object (voice 0);
   - `wavelet_<j>.<v>_AP.wtrans1d` Phase of the Average field of the object (voice > 0);
   - `wavelet_<j>_D.wtrans1d` Detail field of the object (voice 0);
   - `wavelet_<j>.<v>_D.wtrans1d` Detail field of the object (voice > 0);
   - `wavelet_<j>_DP.wtrans1d` Phase of the Detail field of the object (voice 0);

- `wavelet_<j>.<v>_DP.wtrans1d` Phase of the Detail field of the object (voice > 0).

Notice that, regarding to the type of the wavelet transform, only a subset of those files may be generated.

5.1.3 Functions Summary

The following is a description of all the functions related to the `Wtrans1d` type. The list is in alphabetical order.

○Name

mw_alloc_biortho_wtrans1d - Allocate the arrays of the decomposition

○Summary

void *mw_alloc_biortho_wtrans1d(wtrans,level,size)

Wtrans1d wtrans;

int level;

int size;

○Description

This function allocates the arrays `A` and `D` of a `Wtrans1d` structure previously created using `mw_new_wtrans1d`, in order to receive an biorthonormal wavelet representation (one voice per octave, decimation along the time axis). Each signal `A[l][v]` and `D[l][v]` for $l = 0 \ldots$ `nlevel`, $v = 0 \ldots$ `nvoice` $- 1$ ($(l, v) \neq (0, 0)$) is created and allocated to the right size. Previously allocations are deleted, if any.

The number of levels for the decomposition is given by `level` and the size of the original signal is given by `size`.

The arrays `A` and `D` can be addressed after this call, if the allocation successed. There is no default values for the signals. The `type` field of the `Wtrans1d` structure is set to `mw_biorthogonal`.

The function `mw_alloc_biortho_wtrans1d` returns `NULL` if not enough memory is available to allocate one of the signals. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

Notice that, if the wavelet transform is an output of a MegaWave2 module, the structure has been already created by the compiler if needed (See Volume one: "MegaWave2 User's Guide"): do not perform additional call to `mw_new_wtrans1d` (see example below).

○Example

```
Wtrans1d Output; /* optional Output of the module */
Fsignal Signal;  /* needed Input of the module: original signal */
int J;           /* internal use */

if (Output)
{
  /* Output requested : allocate Output for 8 levels of decomposition */
  if(mw_alloc_biortho_wtrans1d(Output, 8, Signal->size)==NULL)
    mwerror(FATAL,1,"Not enough memory.\n");

  Output->A[0][0] = Signal;
```

```
  for (J = 1; J <= 8; J++)
  {
     .
     . (Computation of the voice #J)
     .
  }
}
```

○Name

mw_alloc_continuous_wtrans1d - Allocate the arrays of the decomposition

○Summary

void *mw_alloc_continuous_wtrans1d(wtrans,level,voice,size,complex)

Wtrans1d wtrans;

int level;

int voice;

int size;

int complex;

○Description

This function allocates the arrays `D` of a `Wtrans1d` structure previously created using `mw_new_wtrans1d`, in order to receive an continuous wavelet representation (several voices per octave, no decimation along the time axis, wavelet with complex or real values). The arrays `DP` are allocated if `complex` is set to 1. Each signal `D[l][v]` (and `DP[l][v]` in the complex case) for $l = 0 \ldots$`nlevel`, $v = 0 \ldots$`nvoice` $- 1$ ($(l, v) \neq (0, 0)$) is created and allocated to the right size. Previously allocations are deleted, if any. Notice that, at this time, there is no function to allocate a continuous wavelet transform recording the low-pass signals (`A` and `AP`).

The number of levels for the decomposition is given by `level`, the number of voice per octave is given by `voice` and the size of the original signal is given by `size`.

The arrays `D` and `DP` can be addressed after this call, if the allocation successed. There is no default values for the signals. The `type` field of the `Wtrans1d` structure is set to `mw_continuous`.

The function `mw_alloc_continuous_wtrans1d` returns `NULL` if not enough memory is available to allocate one of the signals. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

Notice that, if the wavelet transform is an output of a MegaWave2 module, the structure has been already created by the compiler if needed (See Volume one: "MegaWave2 User's Guide"): do not perform additional call to `mw_new_wtrans1d` (see example below).

○Example

```
Wtrans1d Output; /* optional Output of the module */
Fsignal Signal;  /* needed Input of the module: original signal */
int J;           /* internal use */

if (Output)
```

```
{
  /* Output requested : allocate Output for 8 levels of decomposition
                        and 10 voices per octave, complex wavelet.
   */
  if(mw_alloc_continuous_wtrans1d(Output, 8, 10, Signal->size,1)==NULL)
    mwerror(FATAL,1,"Not enough memory.\n");

  for (J = 1; J <= 8; J++)
  {
     .
     . (Computation of the voice #J)
     .
  }
}
```

○Name

mw_alloc_dyadic_wtrans1d - Allocate the arrays of the decomposition

○Summary

void *mw_alloc_dyadic_wtrans1d(wtrans,level,size)

Wtrans1d wtrans;

int level;

int size;

○Description

This function allocates the arrays `A` and `D` of a `Wtrans1d` structure previously created using `mw_new_wtrans1d`, in order to receive an dyadic wavelet representation (one voice per octave, no decimation along the time axis). Each signal `A[l][v]` and `D[l][v]` for $l = 0 \ldots$`nlevel`, $v = 0 \ldots$`nvoice` $- 1$ ($(l, v) \neq (0, 0)$) is created and allocated to the right size. Previously allocations are deleted, if any.

The number of levels for the decomposition is given by `level` and the size of the original signal is given by `size`.

The arrays `A` and `D` can be addressed after this call, if the allocation successed. There is no default values for the signals. The `type` field of the `Wtrans1d` structure is set to `mw_dyadic`.

The function `mw_alloc_dyadic_wtrans1d` returns `NULL` if not enough memory is available to allocate one of the signals. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

Notice that, if the wavelet transform is an output of a MegaWave2 module, the structure has been already created by the compiler if needed (See Volume one: "MegaWave2 User's Guide"): do not perform additional call to `mw_new_wtrans1d` (see example below).

○Example

```
Wtrans1d Output; /* optional Output of the module */
Fsignal Signal;  /* needed Input of the module: original signal */
int J;           /* internal use */

if (Output)
{
  /* Output requested : allocate Output for 8 levels of decomposition */
  if(mw_alloc_dyadic_wtrans1d(Output, 8, Signal->size)==NULL)
    mwerror(FATAL,1,"Not enough memory.\n");

  Output->A[0][0] = Signal;
```

```
  for (J = 1; J <= 8; J++)
  {
     .
     . (Computation of the voice #J)
     .
  }
}
```

○Name

mw_alloc_ortho_wtrans1d - Allocate the arrays of the decomposition

○Summary

void *mw_alloc_ortho_wtrans1d(wtrans,level,size)

Wtrans1d wtrans;

int level;

int size;

○Description

This function allocates the arrays `A` and `D` of a `Wtrans1d` structure previously created using `mw_new_wtrans1d`, in order to receive an orthonormal wavelet representation (one voice per octave, decimation along the time axis). Each signal `A[l][v]` and `D[l][v]` for $l = 0 \ldots$ `nlevel`, $v = 0 \ldots$ `nvoice` $- 1$ ($(l, v) \neq (0, 0)$) is created and allocated to the right size. Previously allocations are deleted, if any.

The number of levels for the decomposition is given by `level` and the size of the original signal is given by `size`.

The arrays `A` and `D` can be addressed after this call, if the allocation successed. There is no default values for the signals. The `type` field of the `Wtrans1d` structure is set to `mw_orthogonal`.

The function `mw_alloc_ortho_wtrans1d` returns `NULL` if not enough memory is available to allocate one of the signals. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

Notice that, if the wavelet transform is an output of a MegaWave2 module, the structure has been already created by the compiler if needed (See Volume one: "MegaWave2 User's Guide"): do not perform additional call to `mw_new_wtrans1d` (see example below).

○Example

```
Wtrans1d Output; /* optional Output of the module */
Fsignal Signal;  /* needed Input of the module: original signal */
int J;           /* internal use */

if (Output)
{
  /* Output requested : allocate Output for 8 levels of decomposition */
  if(mw_alloc_ortho_wtrans1d(Output, 8, Signal->size)==NULL)
    mwerror(FATAL,1,"Not enough memory.\n");

  Output->A[0][0] = Signal;
```

```
  for (J = 1; J <= 8; J++)
  {
     .
     . (Computation of the voice #J)
     .
  }
}
```

○Name

mw_delete_wtrans1d - Deallocate the wavelet transform space

○Summary

void mw_delete_wtrans1d(wtrans)

Wtrans1d wtrans;

○Description

This function deallocates the memory used by the wavelet transform space `wtrans` that is, all the memory used by the arrays of signals `A`, `AP`, `D`, `DP` (if any), and the structure itself.

You should set `wtrans = NULL` after this call since the address pointed by `wtrans` is no longer valid.

○Example

```
Wtrans1d wtrans=NULL; /* Internal use: no Input neither Output of module */

if  ( ((wtrans = mw_new_wtrans1d()) == NULL) ||
       (mw_alloc_ortho_wtrans1d(wtrans, 8, 1024)==NULL) )
    mwerror(FATAL,1,"Not enough memory.\n");
.
.
.
mw_delete_wtrans1d(wtrans);
wtrans = NULL;
```

○Name

mw_new_wtrans1d - Create a new Wtrans1d

○Summary

Wtrans1d mw_new_wtrans1d();

○Description

This function creates a new `Wtrans1d` structure with empty arrays of signals `A`, `AP`, `D`, `DP`. No signal can be addressed at this time. The arrays of signals should be allocated using one of the functions `mw_alloc_X_wtrans1d` where `X` depends of the type of the transformation.

You don't need this function for input/output of modules, since the MegaWave2 Compiler already created the structure for you if you need it (See Volume one: "MegaWave2 User's Guide"). This function is used to create internal variables. Do not forget to deallocate the internal structures before the end of the module.

The function `mw_new_wtrans1d` returns `NULL` if not enough memory is available to create the structure. Your code should check this value to send an error message in the `NULL` case, and do appropriate statement.

○Example

```
Wtrans1d wtrans=NULL; /* Internal use: no Input neither Output of module */

if  ( ((wtrans = mw_new_wtrans1d()) == NULL) ||
       (mw_alloc_continuous_wtrans1d(wtrans, 8, 10, 1024)) )
    mwerror(FATAL,1,"Not enough memory.\n");
```

5.2 Two-dimensional wavelet

The *Two-dimensional wavelet* memory type is used to represent the result of a wavelet transform applied to an image. Notice that, at this time, the structure does not allow to record more than one voice per octave for the decomposition. Consequently, the continuous wavelet transform is not available in the 2D case. The wavelet is also assumed to be of real values (complex case not supported).

5.2.1 The structure Wtrans2d

The C structure is the following:

```
  char cmt[mw_cmtsize]; /* Comments */
  char name[mw_namesize]; /* Name of the wtrans2d */

  int type;  /* Type of the wtrans2d performed */
  int edges; /* Type of the edges statments */
  char filter_name[mw_namesize][mw_max_nfilter_2d];   /* Filters used */

  int nrow;
  int ncol;     /* Size of the original image */
  int nlevel;   /* Number of levels (octave) for this decomposition */
  int norient;  /* Number of orientations for this decomposition */
  int nfilter;  /* Number of filters used to compute the decomposition */

  Fimage images[mw_max_nlevel+1][mw_max_norient+1]; /* Wavelet decomposition space */

} *Wtrans2d;
```

The first two fields of this structure is well known by the reader. The field `type` records the type of the wavelet transform used. Its value can be:

- `mw_orthogonal` : orthogonal wavelet transform;
- `mw_biorthogonal` : biorthogonal wavelet transform;
- `mw_dyadic` : dyadic wavelet transform.

The field `edges` gives the type of the edges statment used to compute the transformation. Indeed, since it is implemented as a bank of convolution products, errors occur near the borders if no special statment is performed. This field can have the following values:

- `mw_edges_zeropad` : the image is zero-padded (no special statment);
- `mw_edges_periodic` : the image is made periodic;
- `mw_edges_mirror` : the image is padded by mirror effect (avoid first-order discontinuities);
- `mw_edges_wadapted` : special border functions are added to the wavelets (wavelets on the rectangle).

The field `filter_name` is an array of strings, where the names of the filters used for the decomposition are put. The number of filters is put into the field `nfilter`. This number and the meaning of each filter depend to the wavelet type.

The fields `nrow` (number of rows) and `ncol` (number of columns) contain the size of the original image, which is put into `images[0][0]` (see below). The field `nlevel` is the number of levels used in this decomposition; it corresponds to the number of octaves.

The field `norient` gives the number of orientations used for the decomposition; usually (but the user may modify that) the first orientation (index $r = 0$ in the array `images[][r]`) corresponds to the coarse image at the given resolution (low-pass image or smooth image); the second orientation (index $r = 1$) corresponds to the detail image (wavelet coefficients) along the y direction (horizontal details); the third orientation (index $r = 2$) corresponds to the detail image (wavelet coefficients) along the x direction (vertical details); in the orthonormal and biorthonormal cases, there is another direction (index $r = 3$) which corresponds to the detail image (wavelet coefficients) along the diagonal direction (cross details).

The result of the wavelet decomposition is put into one two-dimensional arrays of images called `images`: `images[l][r]` for $l = 1 \ldots$`nlevel` and for $r = 0 \ldots$`norient` is the coarse or the detail image at the octave l and at the orientation r.

Notice that the images `images[0][r]` are unused except for $r = 0$.

5.2.2 Related file (external) types

The list of the available formats is the following:

1. `"A_WTRANS2D"` MegaWave2 Data Ascii format with a `def Wtrans2d` area. This area includes the value of the different fields of the object, as `comments, type, edges, ....` The values of the wavelet coefficients are not recorded in this file, but in a set of `Fimage` objects. Let be `wavelet` the name of the object. The names of these image files are, for `<j>` the level number (octave) and `<r>` the orientation number,
   - `wavelet_<j>_S.wtrans2d` Average image of the object (`<r>` $= 0$);
   - `wavelet_<j>_D<r>.wtrans2d` Detail image of the object (`<r>` > 0).

5.2.3 Functions Summary

The following is a description of all the functions related to the `Wtrans2d` type. The list is in alphabetical order.

○Name

mw_alloc_biortho_wtrans2d - Allocate the arrays of the decomposition

○Summary

void *mw_alloc_biortho_wtrans2d(wtrans,level,nrow,ncol)

Wtrans2d wtrans;

int level;

int nrow,ncol;

○Description

This function allocates the array `images` of a `Wtrans2d` structure previously created using `mw_new_wtrans2d`, in order to receive an biorthonormal wavelet representation (spatial decimation, `norient` = 3). Each image `images[l][r]` for $l = 1 \ldots$ `nlevel`, $r = 0 \ldots$ `norient` is created and allocated to the right size. Previously allocations are deleted, if any.

The number of levels for the decomposition is given by `level` and the size of the original image is given by `nrow` (number of rows), `ncol` (number of columns).

The array `images` can be addressed after this call, if the allocation successed. There is no default values for the images. The `type` field of the `Wtrans2d` structure is set to `mw_biorthogonal`.

The function `mw_alloc_biortho_wtrans2d` returns `NULL` if not enough memory is available to allocate one of the images. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

Notice that, if the wavelet transform is an output of a MegaWave2 module, the structure has been already created by the compiler if needed (See Volume one: "MegaWave2 User's Guide"): do not perform additional call to `mw_new_wtrans2d` (see example below).

○Example

```
Wtrans2d Output; /* optional Output of the module */
Fimage Image;    /* needed Input of the module: original image */
int J;           /* internal use */

if (Output)
{
  /* Output requested : allocate Output for 8 levels of decomposition */
  if(mw_alloc_biortho_wtrans2d(Output, 8, Image->nrow, Image->ncol)==NULL)
    mwerror(FATAL,1,"Not enough memory.\n");

  Output->images[0][0] = Image;
```

```
  for (J = 1; J <= 8; J++)
  {
     .
     . (Computation of the voice #J)
     .
  }
}
```

○Name

mw_alloc_dyadic_wtrans2d - Allocate the arrays of the decomposition

○Summary

void *mw_alloc_dyadic_wtrans2d(wtrans,level,nrow,ncol)

Wtrans2d wtrans;

int level;

int nrow,ncol;

○Description

This function allocates the array `images` of a `Wtrans2d` structure previously created using `mw_new_wtrans2d`, in order to receive an dyadic wavelet representation (no spatial decimation, `norient` = 2). Each image `images[l][r]` for $l = 1 \ldots$ `nlevel`, $r = 0 \ldots$ `norient` is created and allocated to the right size. Previously allocations are deleted, if any.

The number of levels for the decomposition is given by `level` and the size of the original image is given by `nrow` (number of rows), `ncol` (number of columns).

The array `images` can be addressed after this call, if the allocation successed. There is no default values for the images. The `type` field of the `Wtrans2d` structure is set to `mw_dyadic`.

The function `mw_alloc_dyadic_wtrans2d` returns `NULL` if not enough memory is available to allocate one of the images. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

Notice that, if the wavelet transform is an output of a MegaWave2 module, the structure has been already created by the compiler if needed (See Volume one: "MegaWave2 User's Guide"): do not perform additional call to `mw_new_wtrans2d` (see example below).

○Example

```
Wtrans2d Output; /* optional Output of the module */
Fimage Image;    /* needed Input of the module: original image */
int J;           /* internal use */

if (Output)
{
  /* Output requested : allocate Output for 8 levels of decomposition */
  if(mw_alloc_dyadic_wtrans2d(Output, 8, Image->nrow, Image->ncol)==NULL)
    mwerror(FATAL,1,"Not enough memory.\n");

  Output->images[0][0] = Image;
```

```
  for (J = 1; J <= 8; J++)
  {
     .
     . (Computation of the voice #J)
     .
  }
}
```

○Name

mw_alloc_ortho_wtrans2d - Allocate the arrays of the decomposition

○Summary

void *mw_alloc_ortho_wtrans2d(wtrans,level,nrow,ncol)

Wtrans2d wtrans;

int level;

int nrow,ncol;

○Description

This function allocates the array `images` of a `Wtrans2d` structure previously created using `mw_new_wtrans2d`, in order to receive an orthonormal wavelet representation (spatial decimation, `norient` = 3). Each image `images[l][r]` for $l = 1 \ldots$ `nlevel`, $r = 0 \ldots$ `norient` is created and allocated to the right size. Previously allocations are deleted, if any.

The number of levels for the decomposition is given by `level` and the size of the original image is given by `nrow` (number of rows), `ncol` (number of columns).

The array `images` can be addressed after this call, if the allocation successed. There is no default values for the images. The `type` field of the `Wtrans2d` structure is set to `mw_orthogonal`.

The function `mw_alloc_ortho_wtrans2d` returns `NULL` if not enough memory is available to allocate one of the images. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

Notice that, if the wavelet transform is an output of a MegaWave2 module, the structure has been already created by the compiler if needed (See Volume one: "MegaWave2 User's Guide"): do not perform additional call to `mw_new_wtrans2d` (see example below).

○Example

```
Wtrans2d Output; /* optional Output of the module */
Fimage Image;    /* needed Input of the module: original image */
int J;           /* internal use */

if (Output)
{
  /* Output requested : allocate Output for 8 levels of decomposition */
  if(mw_alloc_ortho_wtrans2d(Output, 8, Image->nrow, Image->ncol)==NULL)
    mwerror(FATAL,1,"Not enough memory.\n");

  Output->images[0][0] = Image;
```

```
  for (J = 1; J <= 8; J++)
  {
     .
     . (Computation of the voice #J)
     .
  }
}
```

○Name

mw_delete_wtrans2d - Deallocate the wavelet transform space

○Summary

void mw_delete_wtrans2d(wtrans)

Wtrans2d wtrans;

○Description

This function deallocates the memory used by the wavelet transform space `wtrans` that is, all the memory used by the array of images `images` (if any), and the structure itself.

You should set `wtrans = NULL` after this call since the address pointed by `wtrans` is no longer valid.

○Example

```
Wtrans2d wtrans=NULL; /* Internal use: no Input neither Output of module */

if  ( ((wtrans = mw_new_wtrans2d()) == NULL) ||
       (mw_alloc_ortho_wtrans2d(wtrans, 6, 512, 512)==NULL) )
    mwerror(FATAL,1,"Not enough memory.\n");
.
.
.
mw_delete_wtrans2d(wtrans);
wtrans = NULL;
```

○Name

mw_new_wtrans2d - Create a new Wtrans2d

○Summary

Wtrans2d mw_new_wtrans2d();

○Description

This function creates a new `Wtrans2d` structure with empty array of images `images`. No image can be addressed at this time. The array of images should be allocated using one of the functions `mw_alloc_X_wtrans2d` where `X` depends of the type of the transformation.

You don't need this function for input/output of modules, since the MegaWave2 Compiler already created the structure for you if you need it (See Volume one: "MegaWave2 User's Guide"). This function is used to create internal variables. Do not forget to deallocate the internal structures before the end of the module.

The function `mw_new_wtrans2d` returns `NULL` if not enough memory is available to create the structure. Your code should check this value to send an error message in the `NULL` case, and do appropriate statement.

○Example

```
Wtrans2d wtrans=NULL; /* Internal use: no Input neither Output of module */

if  ( ((wtrans = mw_new_wtrans2d()) == NULL) ||
       (mw_alloc_dyadic_wtrans2d(wtrans, 6, 512, 512)) )
    mwerror(FATAL,1,"Not enough memory.\n");
```

6 Geometrical structures : Point, Curves, Polygons and Lists

The family of curves, polygons and lists objects are mainly used to handle geometrical processes, as mathematical morphology algorithms, shape analysis, snakes, ...

In MegaWave2, a *curve* (section 6.2) is a set of points in the plane that is, a set of (x, y) coordinates. Although there is no such explicit condition in the system library, most modules assume that this set is really a curve, meaning that points are adjacent for the 4 or 8-connectivity, and that the dimension of the set is less than 2. For a two-dimensional set of points, to avoid memory blowup, consider the segment structure (Section 7.4). A curve is implemented as a chain of points: the curve begins with a first point, from which we can go to the next point, and so one up to the last point. There is no condition set about the geometry of the curve (e.g. the curve can cut itself) but your algorithm may want to put some. There is no an a priori rule to interpolate the curve between two adjacent points in the chain, in the case where they are not adjacent in the plane. Your algorithm may have to process such interpolation.

You may want to handle a *set of curves* (it can be for example the result of an edge detector applied to an image). Such object is also provided in MegaWave2 (section 6.3) and it is implemented as a chain of curves.

What we call *polygon* (section 6.4) is basically a closed curve that is, a chain of (x, y) coordinates where the point next the last point is assumed to be the first point. But one can associate to a polygon a list of real parameters. It can be, for example, only one value which gives the gray level of the constant region delimited by the closed curve. The meaning of the parameters is not pre-defined, so you can used it freely in your algorithms.

You may also want to handle a *set of polygons* (it can be for example the result of a region-segmentation algorithm applied to an image). This object, explained in section 6.5, is of course implemented as a chain of polygons.

All of the objects we have enumerated can record integer or real coordinates (for some applications, you may need real coordinates - e.g. when you compute a P.D.E. to evolve a curve -). In the following, we give a full description of the objects for which coordinates are integers. By putting a `F` (floating-point precision) or `D` (double) at the beginning of the *curve* and *polygon* object's name, you get the corresponding object with real coordinates fields (see section 6.6 for a short description).

We shall begin our description by the basic object used by curves and polygons: the point.

6.1 Point of a planar curve

A `Point_curve` is nothing more than two coordinates (x, y) which can be linked to a previous and to a next `Point_curve`, in order to constitute a curve.

6.1.1 The structure Point_curve

This is the C definition of the structure:

```
typedef struct point_curve {
  int x,y; /* Coordinates of the point */

  /* For use in Curve only */
  struct point_curve *previous; /*Pointer to the previous point (may be NULL)*/
  struct point_curve *next; /* Pointer to the next point (may be NULL) */
} *Point_curve;
```

The first two fields `x` and `y` are the coordinates (x, y) of the point in the plane. Since the `Curve` and the `Polygon` structures are defined as a chain of `Point_curve`, there are two pointers `previous` and `next` associated to each point.

6.1.2 Related file (external) types

Not available: at this time, the `Point_curve` object cannot be used as input/output variables of modules.

6.1.3 Functions Summary

The following is a description of all the functions related to the `Point_curve` type. The list is in alphabetical order.

○Name

mw_change_point_curve - Define the point_curve structure, if not defined

○Summary

Point_curve mw_change_point_curve(point)

Point_curve point;

○Description

This function returns a Point_curve structure if the input `point = NULL`. It is provided despite the `mw_new_point_curve` function for global coherence with other memory types.

The function `mw_change_point_curve` returns `NULL` if not enough memory is available to allocate the structure. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

Since the MegaWave2 compiler allocates structures for input and output objects (See Volume one: "MegaWave2 User's Guide"), this function is normally used only for internal objects. Do not forget to deallocate the internal structures before the end of the module, except if they are part of an input or output chain.

○Example

```
Point_curve point=NULL; /* Internal use: no Input neither Output of module */

/* Define the point (5,1) of the plane */

point = mw_change_point_curve(point);
if (point == NULL) mwerror(FATAL,1,"Not enough memory.\n");
point->x = 5;
point->y = 1;
```

○Name

mw_copy_point_curve - Copy all points starting from the given one

○Summary

Point_curve mw_copy_point_curve(in,out)

Point_curve in, out;

○Description

This function copies the current point and the next points contained in the chain defined at the starting point `in`. The result is put in `out`, which may not be a predefined structure : in case of `out=NULL`, the `out` structure is allocated.

The function `mw_copy_point_curve` returns `NULL` if not enough memory is available to perform the copy, or `out` elsewhere. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

○Example

```
Point_curve in; /* Predefined point */
Point_curve out=NULL;

out=mw_copy_point_curve(in,out);
if (!out) mwerror(FATAL,1,"Not enough memory.\n");
```

○Name

mw_delete_point_curve - Deallocate the point_curve structure

○Summary

void mw_delete_point_curve(point)

Point_curve point;

○Description

This function deallocates the `Point_curve` structures starting from the given `point`, including this point itself. You should set `point = NULL` after this call since the address pointed by `point` is no longer valid. Warning : to deallocate only a point and not all the next points of a chain, just use `free(point)`.

○Example

```
/* Remove the first point of an existing curve */

Curve curve;  /* Existing curve (e.g. Input of module) */
Point_curve point;  /* Internal use */

point = curve->first;
curve->first=point->next;
point->next->previous = NULL;
free(point);
point = NULL;

/* Remove all points of an existing curve */

mw_delete_point_curve(curve->first);
```

○Name

mw_new_point_curve - Create a new point_curve structure

○Summary

Point_curve mw_new_point_curve();

○Description

This function creates a new `Point_curve` structure. It returns `NULL` if not enough memory is available to create the structure. Your code should check this value to send an error message in the `NULL` case, and do appropriate statement.

Since the MegaWave2 compiler allocates structures for input and output objects (See Volume one: "MegaWave2 User's Guide"), this function is normally used only for internal objects. Do not forget to deallocate the internal point structures before the end of the module, except if they are part of an input or output curve.

○Example

```
/* Insert the point (0,0) at the end of an existing curve */

Curve curve;  /* Existing curve (e.g. Input of module) */
Point_curve point,p;  /* Internal use: no Input neither Output of module */

/* Define the point (0,0) */
point = mw_new_point_curve();
if (point == NULL) mwerror(FATAL,1,"Not enough memory.\n");
point->x = point->y = 0;
point->next = NULL;

/* Find the last point of the curve */
p = curve->first; while (p->next) p=p->next;

/* Insert the point */
p->next = point;
point->previous = p;

/* Do not deallocate point or curve will become inconsistent */
```

6.2 Planar curve

You may want to use the `Curve` memory type each type you need to constitute a chain of (x, y) coordinates.

6.2.1 The structure Curve

If `curve` is of `Curve` memory type, then `curve->first` is of `Point_curve`memory type and it is the first point of the curve; `curve->first->next` is the second point, etc. The end of the curve occurs when the `next` field of a point is `NULL`.

```
typedef struct curve {
  Point_curve first; /* Pointer to the first point of the curve */

  /* For use in Curves only */
  struct curve *previous; /* Pointer to the previous curve (may be NULL) */
  struct curve *next; /* Pointer to the next curve (may be NULL) */
} *Curve;
```

You may notice that the `Curve`type includes also the fields `previous` and `next`, as the `Point_curve`type. This is because curves can be linked together to define a set of curves (See 6.2.3 page 114). If the curve is not part of a set, those pointers must be `NULL`.

6.2.2 Related file (external) types

The list of the available formats is the following:

1. `"MW2_CURVE"` MegaWave2 binary format.

6.2.3 Functions Summary

The following is a description of all the functions related to the `Curve` type. The list is in alphabetical order.

○Name

mw_change_curve - Define the curve structure, if not defined

○Summary

Curve mw_change_curve(curve)

Curve curve;

○Description

This function returns a Curve structure if the input `curve = NULL`. It is provided despite the `mw_new_curve` function for global coherence with other memory types.

The function `mw_change_curve` returns `NULL` if not enough memory is available to allocate the structure. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

Since the MegaWave2 compiler allocates structures for input and output objects (See Volume one: "MegaWave2 User's Guide"), this function is normally used only for internal objects. Do not forget to deallocate the internal curve structures before the end of the module, except if they are part of an input or output chain.

○Example

```
/* Define a curve with 10 points which is the straight line (0,0)-(9,9) */

Curve curve=NULL; /* Internal use: no Input neither Output of module */
Point_curve newp,oldp=NULL;
int i;

curve = mw_change_curve(curve);
if (curve == NULL) mwerror(FATAL,1,"Not enough memory.\n");
...
```

(End of this example as for the `mw_new_curve` function).

○Name

mw_copy_curve - Copy a curve into another one

○Summary

Curve mw_copy_curve(in,out)

Curve in, out;

○Description

This function duplicates the points contained in `in`. The result is put in `out`, which may not be a predefined structure : in case of `out=NULL`, the `out` structure is allocated.

The function `mw_copy_curve` returns `NULL` if not enough memory is available to perform the copy, or `out` elsewhere. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

○Example

```
Curve in; /* Predefined curve */
Curve out=NULL;

out=mw_copy_curve(in,out);
if (!out) mwerror(FATAL,1,"Not enough memory.\n");
```

○Name

mw_delete_curve - Deallocate a curve

○Summary

void mw_delete_curve(curve)

Curve curve;

○Description

This function deallocates all the memory allocated by the curve variable that is, all the points belonging to this chain and the `Curve`structure itself. You should set `curve = NULL` after this call since the address pointed by `curve` is no longer valid.

○Example

```
/* Remove the first curve of an existing curve set (curves) */

Curves curves;/* Existing curve set (e.g. Input of module) */
Curve curve;  /* Internal use */

curve = curves->first;
curves->first=curves->next;
curves->next->previous = NULL;
mw_delete_curve(curve);
curve = NULL;
```

○Name

mw_length_curve - Return the number of points of a curve

○Summary

unsigned int mw_length_curve(cv);

Curve cv;

○Description

This function return the number of points contained in the given curve `cv`. It returns 0 if the structure is empty.

○Example

```
Curve curve; /* Internal use: no Input neither Output of module */
Point_curve newp,oldp=NULL;
int i;

curve = mw_new_curve();
if (curve == NULL) mwerror(FATAL,1,"Not enough memory.\n");

/* Define a curve with 5 points */
for (i=1;i<=5;i++)
{
 newp = mw_new_point_curve();
 if (newp == NULL) mwerror(FATAL,1,"Not enough memory.\n");
 if (i=0) curve->first = newp;
 newp->x = newp->y = i;
 newp->previous = oldp;
 if (oldp) oldp->next = newp;
 oldp=newp;
}

/* The length is 5 */
printf("Length=%d\n",mw_length_curve(curve));
```

○Name

mw_new_curve - Create a new curve

○Summary

Curve mw_new_curve();

○Description

This function creates a new `Curve`structure. It returns `NULL` if not enough memory is available to create the structure. Your code should check this value to send an error message in the `NULL` case, and do appropriate statement.

Since the MegaWave2 compiler allocates structures for input and output objects (See Volume one: "MegaWave2 User's Guide"), this function is normally used only for internal objects. Do not forget to deallocate the internal structures before the end of the module, except if they are part of an input or output chain.

○Example

```
/* Define a curve with 10 points which is the straight line (0,0)-(9,9) */

Curve curve; /* Internal use: no Input neither Output of module */
Point_curve newp,oldp=NULL;
int i;

curve = mw_new_curve();
if (curve == NULL) mwerror(FATAL,1,"Not enough memory.\n");

for (i=0;i<10;i++)
{
 newp = mw_new_point_curve();
 if (newp == NULL) mwerror(FATAL,1,"Not enough memory.\n");
 if (i=0) curve->first = newp;
 newp->x = newp->y = i;
 newp->previous = oldp;
 if (oldp) oldp->next = newp;
 oldp=newp;
}
```

6.3 Set of planar curves

The `Curves` memory type is used when you want to handle several curves into only one variable.

6.3.1 The structure Curves

This is the C definition of the structure:

```
typedef struct curves {
  char cmt[mw_cmtsize]; /* Comments */
  char name[mw_namesize]; /* Name of the set */
  Curve first;    /* Pointer to the first curve */
} *Curves;
```

If `curves` is of `Curves` memory type, then `curves->first` is of `Curve` memory type and it is the first curve of the set; therefore, `curves->first->first` is the first point of the first curve. `curves->first->next` is the second curve, etc. The end of the set occurs when the `next` field of a curve is `NULL`.

6.3.2 Related file (external) types

The list of the available formats is the following:

1. `"MW2_CURVES"` MegaWave2 binary format.

6.3.3 Functions Summary

The following is a description of all the functions related to the `Curves` type. The list is in alphabetical order.

○Name

mw_change_curves - Define the curves structure, if not defined

○Summary

Curves mw_change_curves(curves)

Curves curves;

○Description

This function returns a Curves structure if the input `curves = NULL`. It is provided despite the `mw_new_curves` function for global coherence with other memory types.

The function `mw_change_curves` returns `NULL` if not enough memory is available to allocate the structure. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

Since the MegaWave2 compiler allocates structures for input and output objects (See Volume one: "MegaWave2 User's Guide"), this function is normally used only for internal objects. Do not forget to deallocate the internal curves structures before the end of the module.

○Example

```
/* Define a curves set to be two pre-defined curves  */

Curves curves=NULL;   /* Internal use: no Input neither Output of module */
Curve curve1,curve2;  /* Pre-defined curves (e.g. inputs of module) */

curves = mw_change_curves(curves);
if (curves == NULL) mwerror(FATAL,1,"Not enough memory.\n");
...
```

(End of this example as for the `mw_new_curves` function).

○Name

mw_delete_curves - Deallocate a curves set

○Summary

void mw_delete_curves(curves)

Curves curves;

○Description

This function deallocates all the memory allocated by the curves variable that is, all the points belonging to all curves into this set, all `Curve`structures and the `Curves`structure itself. You should set `curves = NULL` after this call since the address pointed by `curves` is no longer valid.

○Example

```
Curves curves=NULL;   /* Internal use: no Input neither Output of module */

curves = mw_new_curves();
if (curves == NULL) mwerror(FATAL,1,"Not enough memory.\n");
.
.
.
mw_delete_curves(curves);
```

○Name

mw_length_curves - Return the number of curves into a curves structure

○Summary

unsigned int mw_length_curves(cvs);

Curves cvs;

○Description

This function returns the number of curves contained in the given `cvs`. It returns 0 if the structure is empty.

○Example

```
/* Define a curves set to be two pre-defined curves  */

Curves curves=NULL;   /* Internal use: no Input neither Output of module */
Curve curve1,curve2;  /* Pre-defined curves (e.g. inputs of module) */

curves = mw_new_curves();
if (curves == NULL) mwerror(FATAL,1,"Not enough memory.\n");

curves->first=curve1;
curve1->previous = curve2->next = NULL;
curve1->next = curve2;
curve2->previous = curve1;

/* The length would be 2 */
printf("Length=%d\n",mw_length_curves(curves));
```

○Name

mw_new_curves - Create a new curves

○Summary

Curves mw_new_curves();

○Description

This function creates a new `Curves`structure. It returns `NULL` if not enough memory is available to create the structure. Your code should check this value to send an error message in the `NULL` case, and do appropriate statement.

Since the MegaWave2 compiler allocates structures for input and output objects (See Volume one: "MegaWave2 User's Guide"), this function is normally used only for internal objects. Do not forget to deallocate the internal structures before the end of the module.

○Example

```
/* Define a curves set to be two pre-defined curves  */

Curves curves=NULL;   /* Internal use: no Input neither Output of module */
Curve curve1,curve2;  /* Pre-defined curves (e.g. inputs of module) */

curves = mw_new_curves();
if (curves == NULL) mwerror(FATAL,1,"Not enough memory.\n");

curves->first=curve1;
curve1->previous = curve2->next = NULL;
curve1->next = curve2;
curve2->previous = curve1;
```

○Name

mw_npoints_curves - Return the total number of points a curves structure contains

○Summary

unsigned int mw_npoints_curves(cvs);

Curves cvs;

○Description

This function returns the total number of points contained in the given `cvs`, that is the sum of `mw_length_curve(cv)` for all curves `cv` contained in `cvs`. The function returns 0 if the structure is empty.

6.4 Polygon, a variant of curve

You should use the `Polygon` memory type when you need to constitute a chain of (x, y) coordinates with some global properties.

6.4.1 The structure Polygon

The first two fields of the structure register the global properties, assumed to be represented as an array of channels; each channel is a real number. The meaning of each channel has to be defined by the user; the number of channels can be selected using the function `mw_alloc_polygon` or `mw_change_polygon` (see below).

The next fields of the structure are similar to those in the `Curve`memory type.

```
typedef struct polygon {
  int nb_channels;  /* Number of channels */
  float *channel;  /* Tab to the channel */
                   /* The number of elements is given by nb_channels */
  Point_curve first; /* Pointer to the first point of the curve */

  /* For use in Polygons only */
  struct polygon *previous; /* Pointer to the previous poly. (may be NULL) */
  struct polygon *next; /* Pointer to the next poly. (may be NULL) */
} *Polygon;
```

6.4.2 Related file (external) types

The list of the available formats is the following:

1. `"A_POLY"` MegaWave2 Data Ascii format with a `def Polygon` area. If a file of this format has several `def Polygon` areas, only the first one is meaningful for the `Polygon` object. Since this format uses Ascii coding, you may read or modify the file just by editing it using a text editor.

6.4.3 Functions Summary

The following is a description of all the functions related to the `Polygon` type. The list is in alphabetical order.

○Name

mw_alloc_polygon - Allocate the channels array

○Summary

Polygon mw_alloc_polygon(polygon,nc)

Polygon polygon;

int nc;

○Description

This function allocates the channels array of a `Polygon`structure previously created using `mw_new_polygon`. The size of the array is given by `nc`, it is the number of different channels. A channel corresponds to a real parameter associated to the polygon. The meaning of such channel has to be defined by the user. For example, `polygon->channel[0]` may be the gray level of the polygon.

Do not use this function if `polygon` has already an allocated channels array: use the function `mw_change_polygon` instead.

The function `mw_alloc_polygon` returns `NULL` if not enough memory is available to allocate the structure or the channels array. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

○Example

See the example of the function `mw_new_polygon`.

○Name

mw_change_polygon - Change the number of channels

○Summary

Polygon mw_change_polygon(polygon,nc)

Polygon polygon;

int nc;

○Description

This function changes the memory allocation for the channels array of a `Polygon`structure, even if no previously memory allocation was done.

The number of channels is given by `nc`; a channel corresponds to a real parameter associated to the polygon. The meaning of such channel has to be defined by the user. For example, `polygon->channel[0]` may be the gray level of the polygon.

This function can also create the structure if the input `polygon = NULL`. Therefore, this function can replace both `mw_new_polygon` and `mw_alloc_polygon`. It is the recommended function to set the number of channels for polygons which are input/output of a module. Since the function can set the address of `polygon`, the variable must be set to the return value of the function (See example below).

The function `mw_change_polygon` returns `NULL` if not enough memory is available to allocate the structure or the channels array. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

○Example

```
Polygon polygon; /* Input of module */

polygon = mw_change_polygon(polygon,1);
if (polygon == NULL) mwerror(FATAL,1,"Not enough memory.\n");
polygon->channel[0] = 255.0;
...
```

(End of this example as for the `mw_new_polygon` function).

○Name

mw_delete_polygon - Deallocate a polygon

○Summary

void mw_delete_polygon(polygon)

Polygon polygon;

○Description

This function deallocates all the memory allocated by the polygon variable that is, all the points belonging to this chain, the channels array (if needed) and the `Polygon`structure itself. You should set `polygon = NULL` after this call since the address pointed by `polygon` is no longer valid.

○Example

```
/* Remove the first polygon of an existing polygon set (polygons) */

Polygons polygons;/* Existing polygons set (e.g. Input of module) */
Polygon polygon;  /* Internal use */

polygon = polygons->first;
polygons->first=polygons->next;
polygons->next->previous = NULL;
mw_delete_polygon(polygon);
polygon = NULL;
```

○Name

mw_length_polygon - Return the number of points of a polygon

○Summary

unsigned int mw_length_polygon(poly);

Polygon poly;

○Description

This function return the number of points contained in the given polygon `poly`. It returns 0 if the structure is empty.

○Example

```
Polygon polygon; /* Internal use: no Input neither Output of module */
point_curve newp,oldp=NULL;
int i;

polygon = mw_new_polygon();
if (polygon == NULL) mwerror(FATAL,1,"Not enough memory.\n");

/* Define a polygon with 5 points */
for (i=1;i<=5;i++)
{
 newp = mw_new_point_curve();
 if (newp == NULL) mwerror(FATAL,1,"Not enough memory.\n");
 if (i=0) polygon->first = newp;
 newp->x = newp->y = i;
 newp->previous = oldp;
 if (oldp) oldp->next = newp;
 oldp=newp;
}

/* The length is 5 */
printf("Length=%d\n",mw_length_polygon(polygon));
```

○Name

mw_new_polygon - Create a new polygon

○Summary

Polygon mw_new_polygon();

○Description

This function creates a new `Polygon`structure with an empty channels array. It returns `NULL` if not enough memory is available to create the structure. Your code should check this value to send an error message in the `NULL` case, and do appropriate statement.

Since the MegaWave2 compiler allocates structures for input and output objects (See Volume one: "MegaWave2 User's Guide"), this function is normally used only for internal objects. Do not forget to deallocate the internal structures before the end of the module, except if they are part of an input or output polygons set.

○Example

```
/* Define a polygon with 10 points which is the straight line (0,0)-(9,9) */

Polygon polygon; /* Internal use: no Input neither Output of module */
Point_curve newp,oldp=NULL;
int i;

polygon = mw_new_polygon();
if ((polygon == NULL) || (mw_alloc_polygon(polygon,1) == NULL))
    mwerror(FATAL,1,"Not enough memory.\n");
polygon->channel[0] = 255.0;

for (i=0;i<10;i++)
{
 newp = mw_new_point_curve();
 if (newp == NULL) mwerror(FATAL,1,"Not enough memory.\n");
 if (i=0) polygon->first = newp;
 newp->x = newp->y = i;
 newp->previous = oldp;
 if (oldp) oldp->next = newp;
 oldp=newp;
}
```

6.5 Set of polygons

The `Polygons` memory type is used when you want to handle several polygons into only one variable.

6.5.1 The structure Polygons

This is the C definition of the structure:

```
typedef struct polygons {
  char cmt[mw_cmtsize]; /* Comments */
  char name[mw_namesize]; /* Name of the set */
  Polygon first;    /* Pointer to the first polygon */
} *Polygons;
```

If `polygons` is of `Polygons` memory type, then `polygons->first` is of `Polygon` memory type and it is the first polygon of the set; therefore, `polygons->first->first` is the first point of the first polygon. `polygons->first->next` is the second polygon, etc. The end of the set occurs when the `next` field of a polygon is `NULL`.

6.5.2 Related file (external) types

The list of the available formats is the following:

1. `"A_POLY"` MegaWave2 Data Ascii format with as many `def Polygon` areas as the number of polygons recorded. Since this format uses Ascii coding, you may read or modify the file just by editing it using a text editor.

6.5.3 Functions Summary

The following is a description of all the functions related to the `Polygons` type. The list is in alphabetical order.

○Name

mw_change_polygons - Define the polygons structure, if not defined

○Summary

Polygons mw_change_polygons(polygons)

Polygons polygons;

○Description

This function returns a Polygons structure if the input `polygons = NULL`. It is provided despite the `mw_new_polygons` function for global coherence with other memory types.

The function `mw_change_polygons` returns `NULL` if not enough memory is available to allocate the structure. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

Since the MegaWave2 compiler allocates structures for input and output objects (See Volume one: "MegaWave2 User's Guide"), this function is normally used only for internal objects. Do not forget to deallocate the internal polygons structures before the end of the module.

○Example

```
/* Define a polygons set to be two pre-defined polygons  */

Polygons polygons=NULL;   /* Internal use: no Input neither Output of module */
Polygon polygon1,polygon2;  /* Pre-defined polygons (e.g. inputs of module) */

polygons = mw_change_polygons(polygons);
if (polygons == NULL) mwerror(FATAL,1,"Not enough memory.\n");
...
```

(End of this example as for the `mw_new_polygons` function).

○Name

mw_delete_polygons - Deallocate a polygons set

○Summary

void mw_delete_polygons(polygons)

Polygons polygons;

○Description

This function deallocates all the memory allocated by the polygons variable that is, all the points belonging to all polygons into this set, all channels arrays (if any), all `Polygon`structures and the `Polygons`structure itself. You should set `polygons = NULL` after this call since the address pointed by `polygons` is no longer valid.

○Example

```
Polygons polygons=NULL;   /* Internal use: no Input neither Output of module */

polygons = mw_new_polygons();
if (polygons == NULL) mwerror(FATAL,1,"Not enough memory.\n");
.
.
.
mw_delete_polygons(polygons);
```

○Name

mw_length_polygons - Return the number of polygons into a polygons structure

○Summary

unsigned int mw_length_polygons(polys);

Polygons polys;

○Description

This function returns the number of polygons contained in the given `polys`. It returns 0 if the structure is empty.

○Example

```
/* Define a polygons set to be two pre-defined polygons  */

Polygons polygons=NULL;   /* Internal use: no Input neither Output of module */
Polygon polygon1,polygon2;  /* Pre-defined polygons (e.g. inputs of module) */

polygons = mw_new_polygons();
if (polygons == NULL) mwerror(FATAL,1,"Not enough memory.\n");

polygons->first=polygon1;
polygon1->previous = polygon2->next = NULL;
polygon1->next = polygon2;
polygon2->previous = polygon1;

/* The length would be 2 */
printf("Length=%d\n",mw_length_polygons(polygons));
```

○Name

mw_new_polygons - Create a new polygons

○Summary

Polygons mw_new_polygons();

○Description

This function creates a new `Polygons`structure. It returns `NULL` if not enough memory is available to create the structure. Your code should check this value to send an error message in the `NULL` case, and do appropriate statement.

Since the MegaWave2 compiler allocates structures for input and output objects (See Volume one: "MegaWave2 User's Guide"), this function is normally used only for internal objects. Do not forget to deallocate the internal structures before the end of the module.

○Example

```
/* Define a polygons set to be two pre-defined polygons  */

Polygons polygons=NULL;   /* Internal use: no Input neither Output of module */
Polygon polygon1,polygon2;  /* Pre-defined polygons (e.g. inputs of module) */

polygons = mw_new_polygons();
if (polygons == NULL) mwerror(FATAL,1,"Not enough memory.\n");

polygons->first=polygon1;
polygon1->previous = polygon2->next = NULL;
polygon1->next = polygon2;
polygon2->previous = polygon1;
```

6.6 Points, Curves and Polygons with real coordinates

Until now, all of the objects we have described in the section 6 record the coordinates as integers. Use the following objects if you need coordinates of floating point values: `Point_fcurve`, `Fcurve`, `Fcurves`, `Fpolygon`, `Fpolygons`. If you need higher precision, use the following objects (coordinates are recorded as double): `Point_dcurve`, `Dcurve`, `Dcurves`.

We will not give the full description of these objects and of their related functions since it is equivalent to the former description, just keep in mind to translate the words *curve* to *fcurve* or *dcurve* and *polygon* to *fpolygon*, both in the type names (the first letter being upper-case) and in the function names.

And, of course, do not forget that the coordinates are now real. The C definition of the structure `Point_fcurve` is the following:

```
typedef struct point_fcurve {
  float x,y; /* Coordinates of the point */

  /* For use in Fcurve only */
  struct point_fcurve *previous; /*Pointer to the previous point (may be NULL)*/
  struct point_fcurve *next; /* Pointer to the next point (may be NULL) */
} *Point_fcurve;
```

The C definition of the structure `Point_dcurve` is the following:

```
typedef struct point_dcurve {
  double x,y; /* Coordinates of the point */

  /* For use in Dcurve only */
  struct point_dcurve *previous; /*Pointer to the previous point (may be NULL)*/
  struct point_dcurve *next; /* Pointer to the next point (may be NULL) */
} *Point_dcurve;
```

6.7 Lists of n-tuple reals

Some algorithms dealing with curves can be made more efficient if image coordinates are not recorded as a chain of points (x, y), but as part of an array. In such case, use one of the `Flist`, `Flists`, `Dlist`, `Dlists` objects above. These types can more generally be used to handle any list of n-tuple reals, the case of points in the plane corresponding to $n = 2$. As for curves, `Dlist` and `Dlists` are the counterpart of `Flist` and `Flists` : the only difference between them is that values are of type double instead of float.

6.7.1 The structure Flist

In a variable of `Flist` memory type, data such as coordinates are recorded in the array named `values`. We call *dimension* (field named `dim`) the number of components per elements the array is composed, while the field named `size` gives the number of elements. When a `Flist` is used as a `Fcurve`, the dimension is 2 (number of coordinates in the plane) and the size is the number of points.

The field `data` can be used to record any additional information (when no information is available, it is set to `NULL`). The size of the space pointed by `data` is set in `data_size`.

```
typedef struct flist {
```

```
  int size;          /* size (number of elements) */
  int max_size;      /* currently allocated size (maximum number of elements) */
  int dim;           /* dimension (number of components per element) */
  float *values;     /* values = size * dim array
                        nth element = values[n*dim+i], i=0..dim-1 */
  int data_size;     /* size of data[] in bytes */
  void* data;        /* User defined field (saved). A pointer to something */
} *Flist;
```

6.7.2 Related file (external) types

The list of the available native formats is the following:

1. `"MW2_FLIST"` MegaWave2 binary format.

6.7.3 Functions Summary

The following is a description of all the functions related to the `Flist` type. The list is in alphabetical order. Notice that these functions do not manage the `data` field.

○Name

mw_change_flist - Define and allocate a `Flist` structure

○Summary

Flist mw_change_flist(l,max_size,size,dim)

Flist l; int max_size,size,dim;

○Description

This function changes the memory allocation of the `values` array of a `Flist` structure, even if no previously memory allocation was done. The new size (number of elements) of the structure is given by `size`, the size to allocate (maximal number of elements) by `max_size`, and the dimension by `dim`.

It can also create the structure if the input `l = NULL`. Therefore, this function can replace both `mw_new_flist` and `mw_realloc_flist`. Since the function can set the address of `l`, the variable must be set to the return value of the function (See example below).

The function `mw_change_flist` returns `NULL` if not enough memory is available to allocate the structure or the `values` array, and an error message is issued. Your code should check this return value to eventually send a fatal error message in the `NULL` case, and do appropriate statement.

○Example

```
Flist l;

/*
   Allocate l to handle at most 10 samples of couples (2) of
   floating point values, the default number of samples being 0.
*/
l = mw_change_flist(NULL,10,0,2);
if (!l) mwerror(FATAL,1,"Not enough memory to continue !\n");
```

○Name

mw_clear_flist - Clear the array of a `Flist` structure

○Summary

void mw_clear_flist(l,v)

Flist l; float v;

○Description

This function clears the `values` array by filling it with the value `v` (up to the current number of samples).

○Example

```
Flist l;

/*
   Allocate l to handle at most 10 samples of couples (2) of
   floating point values, the default number of samples being 5.
*/
l = mw_change_flist(NULL,10,5,2);
if (!l) mwerror(FATAL,1,"Not enough memory to continue !\n");

/*
  Clear the 5 current samples with 0.
*/
mw_clear_flist(l,0.0);
```

○Name

mw_copy_flist - Copy a the array `Flist` structure

○Summary

Flist mw_copy_flist(in,out)

Flist in,out;

○Description

This function copies the `values` array and `data` field of the `Flist` structure `in` into `out`. The duplicated `Flist out` is allocated to at least the current size of `in`.

Since the function can set the address of `out`, the variable must be set to the return value of the function (See example below).

The function `mw_copy_flist` returns `NULL` if not enough memory is available to allocate the structure or the `values` array, and an error message is issued. Your code should check this return value to eventually send a fatal error message in the `NULL` case, and do appropriate statement.

○Example

```
Flist in,out=NULL;

/*
   Allocate in to handle at most 10 samples of couples (2) of
   floating point values, the current number of samples being 5.
*/
in = mw_change_flist(NULL,10,5,2);
if (!in) mwerror(FATAL,1,"Not enough memory to continue !\n");

/*
  Clear the 5 current samples with 1.
*/
mw_clear_flist(in,1.0);

/*
 Copy in into out. Allocated size for out is 5 samples.
*/
out=mw_copy_flist(in,out);
if (!out) mwerror(FATAL,1,"Not enough memory to copy flist !\n");
```

○Name

mw_delete_flist - Delete the array and the `Flist` structure

○Summary

void mw_delete_flist(l)

Flist l;

○Description

This function deletes the `values` array and the structure itself. Warning : the memory of the user-defined field `data` is not freed. If this field has been allocated, you should free it before calling `mw_delete_flist`.

○Example

```
Flist l;

/*
   Allocate l to handle at most 10 samples of couples (2) of
   floating point values, the default number of samples being 5.
*/
l = mw_change_flist(NULL,10,5,2);
if (!l) mwerror(FATAL,1,"Not enough memory to continue !\n");

/*
  Allocate the data field for 20 integers.
*/
l->data_size=20*sizeof(int);
l->data= (int *)malloc(l->data_size);
if (!l->data) mwerror(FATAL,1,"Not enough memory to continue !\n");


/*
    ... (statement)...
*/

/*
  Free the list, including data field.
*/
free(l->data);
mw_delete_flist(l);
```

○Name

mw_enlarge_flist - Enlarge the array of a `Flist`

○Summary

Flist mw_enlarge_flist(l)

Flist l;

○Description

This function performs a memory reallocation on the array `l->values` to increase the number of elements that can be recorded. The enlargement factor is fixed by the constant `MW_LIST_ENLARGE_FACTOR` defined in the include file `list.h`. This function is useful when one does not know by advance the size of the list, and when one wish to avoid multiple reallocations.

If not enough memory is available to perform the reallocation, an error message is issued and the function returns `NULL`. Otherwise, the function returns `l`.

○Example

```
/* Fill a flist with diagonal points using mw_enlarge_flist
   up to a random size, unknown by advance.
*/

Flist l;

l = mw_change_flist(NULL,2,0,2);
if (l==NULL) mwerror(FATAL,1,"Not enough memory to continue !\n");
i=0;
do
  {
   if ((2*i == l->max_size) && (!mw_enlarge_flist(l)))
       mwerror(FATAL,1,"Not enough memory to continue !\n");
   l->values[i++] = l->values[i++] = i;
  }  while (rand() != 0);
l->size=(i+1)/2;
```

○Name

mw_new_flist - Create a `Flist` structure

○Summary

Flist mw_new_flist()

○Description

This function creates a new `Flist` structure. The fields are initialized to 0 or `NULL`value. The function returns the address of the new structure, or `NULL` if not enough memory is available.

○Example

```
Flist l;

/*
  Define the structure
*/
l = mw_new_flist();
if (!l) mwerror(FATAL,1,"Not enough memory to define the list !\n");

/*
  At that time, the FList is empty.
*/
```

○Name

mw_realloc_flist - Realloc the array of a `Flist`

○Summary

Flist mw_realloc_flist(l,n)

Flist l;

int n;

○Description

This function performs a memory reallocation on the array `l->values` so that at most n elements can be recorded.

If not enough memory is available to perform the reallocation, an error message is issued and the function returns `NULL`. Otherwise, the function returns `l`.

○Example

```
Flist l;

/*
   Allocate l to handle at most 1000 samples of 500-tuple of
   floating point values, the default number of samples being 1000.
*/
l = mw_change_flist(NULL,1000,1000,500);
if (!l) mwerror(FATAL,1,"Not enough memory to continue !\n");

/*
    ... (statement)...
*/

/*
   Now we need space for 20 samples only : by doing reallocation,
   we allow to free some memory.
*/
l = mw_realloc_flist(l,20);
if (!l) mwerror(FATAL,1,"Couldn't realloc flist !\n");
```

6.7.4 The structure Flists

A `Flists` structure is an array of `Flist` not necessary of the same size. As for the `Flist` structure, the `Flists` structure contains a `data` field that can be used to record any additional information (when no information is available, it is set to `NULL`). The size of the space pointed by `data` is set in `data_size`.

```
typedef struct flists {
  char cmt[mw_cmtsize];     /* Comments */
  char name[mw_namesize];   /* Name */
  int size;          /* size (number of lists) */
  int max_size;      /* currently allocated size (maximum number of lists) */
  Flist *list;       /* array of Flist */
  int data_size;     /* size of data[] in bytes */
  void* data;        /* User defined field (saved). A pointer to something */
} *Flists;
```

6.7.5 Related file (external) types

The list of the available native formats is the following:

1. `"MW2_FLISTS"` MegaWave2 binary format.

6.7.6 Functions Summary

The following is a description of all the functions related to the `Flists` type. The list is in alphabetical order. Notice that these functions do not manage the `data` field.

○Name

mw_change_flists - Define and allocate a `Flists` structure

○Summary

Flists mw_change_flists(ls,max_size,size)

Flist ls;

int max_size,size;

○Description

This function changes the memory allocation of the `list` array of a `Flists` structure, even if no previously memory allocation was done. The new size (number of lists) of the structure is given by `size`, and the size to allocate (maximal number of lists) by `max_size`.

It can also create the structure if the input `ls = NULL`. Therefore, this function can replace both `mw_new_flists` and `mw_realloc_flists`. Since the function can set the address of `ls`, the variable must be set to the return value of the function (See example below).

The function `mw_change_flists` returns `NULL` if not enough memory is available to allocate the structure or the `list` array, and an error message is issued. Your code should check this return value to eventually send a fatal error message in the `NULL` case, and do appropriate statement.

○Example

```
Flists ls;

/*
   Allocate ls to handle at most 10 lists, the current number of
   lists being 0 (no list).
*/
ls = mw_change_flists(NULL,10,0);
if (!ls) mwerror(FATAL,1,"Not enough memory to continue !\n");
```

○Name

mw_copy_flists - Copy the lists contained in a `Flists` structure

○Summary

Flists mw_copy_flists(in,out)

Flists in,out;

○Description

This function copies the `list` array and `data` field of the `Flists` structure `in` into `out` : each list contained in `in` are duplicated. The duplicated `Flists` `out` is allocated to at least the current size of `in`.

Since the function can set the address of `out`, the variable must be set to the return value of the function (See example below).

The function `mw_copy_flists` returns `NULL` if not enough memory is available to allocate the structure or the `list` array, and an error message is issued. Your code should check this return value to eventually send a fatal error message in the `NULL` case, and do appropriate statement.

○Example

```
Flists in,out=NULL;

/*
   Allocate ls to handle at most 10 lists, the current number of
   lists being 3.
*/
ls = mw_change_flists(NULL,10,3);
if (!ls) mwerror(FATAL,1,"Not enough memory to continue !\n");


/* ... (Here fill the lists) ... */

/*
 Copy in into out. Allocated size for out is 3 lists.
*/
out=mw_copy_flists(in,out);
if (!out) mwerror(FATAL,1,"Not enough memory to copy the lists !\n");
```

○Name

mw_delete_flists - Delete the lists and the `Flists` structure

○Summary

void mw_delete_flists(ls)

Flist ls;

○Description

This function deletes the lists contained in the `list` array, and the structure `Flists` itself. Warning : the memory of the user-defined field `data` is not freed. If this field has been allocated, you should free it before calling `mw_delete_flists`.

○Example

```
Flist ls;
int i;

/*
    ... (Assume ls has been previoulsy allocated)...
*/

/*
  Free the lists, including data field.
*/
for (i=ls->size;i--;) if (ls->list[i]->data) free(ls->list[i]->data);
if (ls->data) free(ls->data);
mw_delete_flists(ls);
```

○Name

mw_enlarge_flists - Enlarge the number of lists a `Flists` may contain

○Summary

Flists mw_enlarge_flist(ls)

Flist ls;

○Description

This function performs a memory reallocation on the array `ls->list` to increase the number of lists that can be recorded. The enlargement factor is fixed by the constant `MW_LIST_ENLARGE_FACTOR` defined in the include file `list.h`. This function is useful when one does not know by advance the number of lists, and when one wish to avoid multiple reallocations.

If not enough memory is available to perform the reallocation, an error message is issued and the function returns `NULL`. Otherwise, the function returns `ls`.

○Example

```
/* Fill a flists with lists until the user enters 'Q'.
*/

Flist ls;
Flist l;
char c;

ls = mw_change_flists(NULL,10,0);
if (ls==NULL) mwerror(FATAL,1,"Not enough memory to continue !\n");
do {
  if (ls->size == ls->max_size)
    if (mw_enlarge_flists(ls)==NULL)
      mwerror(FATAL,1,"Not enough memory to continue !\n");
  l = mw_change_flist(NULL,10,10,2);
  if (l==NULL) mwerror(FATAL,1,"Not enough memory to continue !\n");
  mw_clear_flist(l,1.0)
  ls->list[ls->size++] = l;
  scanf("%c",&c);
    } while (c!='Q');
```

○Name

mw_new_flists - Create a `Flists` structure

○Summary

Flists mw_new_flists()

○Description

This function creates a new `Flists` structure. The fields are initialized to 0 or `NULL`value. The function returns the address of the new structure, or `NULL` if not enough memory is available.

○Example

```
Flists ls;

/*
  Define the structure
*/
ls = mw_new_flists();
if (!ls) mwerror(FATAL,1,"Not enough memory to define the lists !\n");

/*
  At that time, the FLists is empty (no lists).
*/
```

○Name

mw_realloc_flists - Realloc the list array of the `Flists`

○Summary

Flists mw_realloc_flists(ls,n)

Flists ls;

int n;

○Description

This function performs a memory reallocation on the array `ls->list` so that at most n lists can be recorded.

If not enough memory is available to perform the reallocation, an error message is issued and the function returns `NULL`. Otherwise, the function returns `ls`.

○Example

```
Flists ls;

/*
   Allocate ls to handle 10 lists.
*/

ls = mw_new_flists();
if (!ls) mwerror(FATAL,1,"Not enough memory to continue !\n");
ls = mw_realloc_flists(ls,10);
if (!ls) mwerror(FATAL,1,"Not enough memory to continue !\n");
```

6.7.7 The structures Dlist and Dlists

As for curves, `Dlist` and `Dlists` are the counterpart of `Flist` and `Flists` : the only difference between them is that values are of type double instead of float. Since you can easily imagine how it works, we will not document the functions associated to `Dlist` and `Dlists`. Just change the letter `f` to `d`.

```
typedef struct dlist {

  int size;          /* size (number of elements) */
  int max_size;      /* currently allocated size (number of ELEMENTS) */
  int dim;           /* dimension (number of components per element) */

  double *values;    /* values = size * dim array
                        nth element = values[n*dim+i], i=0..dim-1 */

  int data_size;     /* size of data[] in bytes */
  void* data;        /* User defined field (saved). A pointer to something */

} *Dlist;


typedef struct dlists {

  char cmt[mw_cmtsize];     /* Comments */
  char name[mw_namesize];   /* Name */

  int size;          /* size (number of elements) */
  int max_size;      /* currently allocated size (number of ELEMENTS) */

  Dlist *list;       /* array of Dlist */

  int data_size;     /* size of data[] in bytes */
  void* data;        /* User defined field (saved). A pointer to something */

} *Dlists;
```

6.7.8 Related file (external) types

Here is the list of available native formats associated to `Dlist` internal type :

1. `"MW2_DLIST"` MegaWave2 binary format.

The list of available native formats associated to `Dlists` internal type is

1. `"MW2_DLISTS"` MegaWave2 binary format.

7 Level sets and morphological structures

This section describes the various morphological structures used to represent images. We call morphological representation any complete decomposition which is invariant by (local or global) contrast changes. More precisely, if $\mathcal{R}$ is the representation operator and c a contrast change function (that is, any non-decreasing real function), the contrast change invariance corresponds to the property $\mathcal{R}(c(u)) = c(\mathcal{R}(u))$ for every image u. Exemples of such representations are based by level sets, level lines and connected components of level sets.

We begin our description with the `Shape` and `Shapes` structures. These are not the first developed in MegaWave2, but they are going to play an increasing role : they allow to handle level sets and connected components of level sets in a tree structure very useful to develop morphological shape-based applications. In addition, computation of these structures can be performed in a way faster than the traditional level set decomposition, using the Fast Level Set Transform (*FLST* in short). The FLST has been created by Pascal Monasse during its PhD thesis. The following description of the `Shape` and `Shapes` structures has been written with his help.

7.1 Shape

A `Shape` is a set of pixels based on a level set of an image. It can be a level set itself, one of its connected component, or a shape as defined by the FLST (see module `flst`) that is, in short, a connected component of a level set with filled holes. Notice that a `Shape` has no reference to the image in which it is extracted, so a `Shape` can be constructed from scratch, without an initial image.

The basic fields are:

- `inferior_type`: a nonzero value indicates that the `Shape` corresponds to a lower level set of level λ ($\{x : u(x) \leq \lambda\}$[1] or $\{x : u(x) < \lambda\}$, those sets being noted in short by $[u \leq \lambda]$ and $[u < \lambda]$), while a zero value indicates an upper level set ($[u \geq \lambda]$ or $[u > \lambda]$).
- `value`: the gray level λ of the level set.
- `area`: the area, i.e., the number of pixels of the shape.
- `pixels`: an array of pixel coordinates containing `area` elements.
- `boundary`: a `Flist` of dimension two containing the vertices of a polygonal representation of the boundary.
- `open`: a nonzero value indicates that the `Shape` meets the border of the image. The name of this field comes from the fact that if the boundary is a curve, it is an open curve.

Moreover, there is an additional field `removed` indicating if the shape is to be taken into account. This field is interesting only in the case where the shape is part of a structure.

A shape is supposed to be included in a tree structure driven by inclusion. This is the case for example when the shapes are all lower (or all upper) level sets: in this case the tree has no ramification, since the level sets are monotone for inclusion. There is a true tree structure when they are *connected components* of lower (or upper) level sets. This is also true for the shapes in the sense of the FLST.

In the vocabulary of graphs, the edges of the tree adjacent to the shape are stored in the fields `parent`, `child` and `next_sibling`. The `child` field corresponds actually to the first child of the shape. The other ones can be recovered by following the pointers `next_sibling`. For example, to call the function `foo` successively with the children of shape `s` as argument, we would write the following code snippet:

[1] we denote by x a point in the image u.

```
for(c = s->child; c != NULL; c=c->next_sibling) foo(c);
```

The parent contains the shape while the shape contains its children. Functions for accessing these three fields are given: they take into account that some shapes may be ignored, as indicated by the field `removed`.

It is dangerous to remove the root of the tree by setting its `removed` field: many functions rely on the fact that we have a root.

7.1.1 The structure Shape

The meaning of the different fields is explained above. There are two additional fields, `data` and `data_size`, whose content is left to the choice of the user. `data` is supposed to point to a memory extent of (at least) `data_size` bytes, if this value is positive. Failure in this assumption may lead to a memory corruption in I/O operations.

```
typedef struct shape
{
  char inferior_type; /* Indicates if it is extracted from a superior
                         or inferior level set */
  float value; /* Limiting gray-level of the level set */
  char open; /* Indicates if the shape meets the border of the image */
  int area; /* Area of the shape = area of the cc of level set
                                  + areas of the holes */
  char removed; /* Indicates whether the shape exists or not */

  Point_plane pixels; /* The array of pixels contained in the shape */

  Flist boundary; /* The boundary curve defining the shape */

  /* Data to include it in a tree. It has a parent (the smallest containing
     shape), children (the largest contained shapes, whose first is pChild
     and the others are its siblings), and siblings (the other children of
     its parent) */
  struct shape *parent, *next_sibling, *child;

  int data_size;     /* size of data[] in bytes */
  void* data;        /* User defined field (saved). A pointer to something */

} *Shape;
```

7.1.2 Related file (external) types

1. `"MW2_SHAPE"` MegaWave2 binary format.

7.1.3 Functions Summary

The following is a description of all the functions related to the `Shape` type. The list is in alphabetical order.

○Name

mw_change_shape - Create a `Shape` structure if necessary

○Summary

Shape mw_change_shape(sh)

Shape sh;

○Description

This function creates a `Shape` structure if `sh` is not already defined. The fields are initialized to 0 or `NULL`value. The function returns the address of the structure, or `NULL` if not enough memory is available.

○Example

```
Shape sh=NULL;

/*
  Define the structure
*/
sh = mw_change_shape(sh);
if (!sh) mwerror(FATAL,1,"Not enough memory to define the shape !\n");

/*
  At that time, the shape is empty.
*/
```

○Name

mw_delete_shape - Free the memory allocated for a `Shape` structure

○Summary

void mw_delete_shape(sh)

Shape sh;

○Description

This function deletes the `pixels` array, the `boundary` `Flist`, the `data` array (if needed), and the structure itself.

○Example

```
Shape sh;

/*
  Define the structure
*/
sh = mw_new_shape();
if (!sh) mwerror(FATAL,1,"Not enough memory to define the shape !\n");

/*
    ...(computation of the shape)...
*/

/*
  Free the shape, including data field.
*/
mw_delete_shape(sh);
```

○Name

mw_get_first_child_shape - Return the first child of a shape in the tree

○Summary

Shape mw_get_first_child_shape(sh)

Shape sh;

○Description

This function returns the first child of the shape `sh`, skipping removed shapes (field `removed`). This is equivalent to `sh->child` if this shape is not removed.

○Name

mw_get_next_sibling_shape - Return the next sibling of a shape in the tree

○Summary

Shape mw_get_next_sibling_shape(sh)

Shape sh;

○Description

This function returns the next sibling (shape sharing the same parent) of the shape `sh`, skipping removed shapes (field `removed`). This is equivalent to `sh->next_sibling` if this shape is not removed.

○Name

mw_get_not_removed_shape - Return the first shape not removed in subtree

○Summary

Shape mw_get_not_removed_shape(sh)

Shape sh;

○Description

This function returns `sh` if this shape is not removed (field `removed`), else it is equivalent to `mw_get_first_child(sh)` that is, it returns the first child, skipping removed shapes.

○Name

mw_get_parent_shape - Return the parent of the shape in the tree

○Summary

Shape mw_get_parent_shape(sh)

Shape sh;

○Description

This function returns the parent of the shape `sh`, skipping removed shapes (field `removed`). This is equivalent to `sh->parent` if this shape is not removed.

○Name

mw_get_smallest_shape - Return the smallest shape containing a given pixel

○Summary

Shape mw_get_smallest_shape(shs,x,y)

Shapes shs; int x,y;

○Description

This function returns the smallest shape containing the pixel at position (x, y), ignoring removed shapes (field `removed`). This is equivalent to `shs->smallest_shape[y*shs->ncol+x]` provided this shape is not removed.

○Name

mw_new_shape - Create a `Shape` structure

○Summary

Shape mw_new_shape()

○Description

This function creates a new `Shape` structure. The fields are initialized to 0 or `NULL`value. The function returns the address of the new structure, or `NULL` if not enough memory is available.

○Example

```
Shape sh;

/*
  Define the structure
*/
sh = mw_new_shape();
if (!sh) mwerror(FATAL,1,"Not enough memory to define the shape !\n");

/*
  At that time, the shape is empty.
*/
```

7.2 Shapes

A `Shapes` structure is a collection of shapes extracted from an image. The fields `nrow` and `ncol` are the dimensions of the image. The field `interpolation` is the convention used to extract level lines. Currently, the valid values are 0 (module `flst`) and 1 (module `flst_bilinear`).

The elements are stored consecutively in the array `the_shapes` of size `nb_shapes`. By convention, the shape at index 0 is the root of the tree.

The field `smallest_shape` is an array of size `nrow`×`ncol` giving for each pixel the smallest shape in the tree that contains it. By going upward in the tree, it is possible to know all the shapes containing a given pixel.

7.2.1 The structure Shapes

The meaning of the fields is explained above. The fields `data_size` and `data` are left to the user.

```
typedef struct shapes
{
  char cmt[mw_cmtsize];   /* Comments */
  char name[mw_namesize]; /* Name of the set */
  int nrow;               /* Number of rows (dy) of the image */
  int ncol;               /* Number of columns (dx) of the image */
  int interpolation;      /* Interpolation used for the level lines:
                             0=nearest neighbor, 1=bilinear */
  Shape the_shapes; /* Array of the shapes.
                       The root of the tree is at index 0 */

  int nb_shapes; /* The number of shapes (the size of the array the_shapes) */

  /* Link between pixels and shapes */
  Shape *smallest_shape; /* An image giving for each pixel
                            the smallest shape containing it */

  int data_size;     /* size of data[] in bytes */
  void* data;        /* User defined field (saved). A pointer to something */

} *Shapes;
```

7.2.2 Related file (external) types

1. `"MW2_SHAPES"` MegaWave2 binary format.

7.2.3 Functions Summary

The following is a description of all the functions related to the `Shapes` type. The list is in alphabetical order.

○Name

mw_alloc_shapes - Allocate the fields of a `Shapes` structure

○Summary

Shapes mw_alloc_shapes(shs, nrow, ncol, value)

Shapes shs;

int nrow, ncol;

float value; /* gray level value of the root */

○Description

This function takes as argument a `Shapes` structure and returns it after having allocated all necessary fields. The input `nrow` and `ncol` are the dimensions of the image. The field `the_shapes` is allocated to contain `nrow`×`ncol`+1 shapes, which is the maximal number of shapes extracted by the FLST (see module `flst`). In fact, only one shape is put, the root of the tree, supposed to be extracted at gray level `value`. The field `smallest_shape` is also allocated and initialized, each pixel having as smallest shape the root.

The function returns `shs`, or `NULL` if not enough memory is available to do the allocation.

○Example

```
Shapes shs;
Fimage image; /* Assume image is allocated */

/*
  Define the structure
*/
shs = mw_new_shapes();
if (!shs) mwerror(FATAL,1,"Not enough memory to define the shapes !\n");

/*
  At that time, the structure exists but fields are empty : alloc them
  to handle the Fimage image.
*/
if (!mw_alloc_shapes(shs, image->nrow, image->ncol, image->gray[0]))
  mwerror(FATAL,1,"Not enough memory to alloc the shapes !\n");
```

○Name

mw_change_shapes - (Re)alloc the fields of a `Shapes` structure

○Summary

Shapes mw_change_shapes(shs, nrow, ncol, value)

Shapes shs;

int nrow, ncol;

float value; /* gray level value of the root */

○Description

If the input pointer `shs` is `NULL`, create a new structure, otherwise delete the currently allocated fields (if any) and call `mw_alloc_shapes()`.

The function returns the new structure or `shs`, or `NULL` if not enough memory is available to do the allocation.

○Example

```
Shapes shs=NULL;
Fimage image; /* Assume image is allocated */

/*
  Define the structure and alloc the field to handle the Fimage image.
*/
shs = mw_change_shapes(shs, image->nrow, image->ncol, image->gray[0]);
if (!shs) mwerror(FATAL,1,"Not enough memory to alloc the shapes !\n");
```

○Name

mw_delete_shapes - Delete a `Shapes` structure

○Summary

void mw_delete_shapes(shs)

Shapes shs;

○Description

This function frees the allocated fields and the structure itself. After this call, the memory pointed to by `shs` must not be accessed any longer. Warning : in the contrary to `mw_delete_shape()`, the memory of the user-defined field `data` is not freed. If this field has been allocated, you should free it before calling `mw_delete_shapes()`.

○Example

```
Shapes shs=NULL;
Fimage image; /* Assume image is allocated */

/*
  Define the structure and alloc the field to handle the Fimage image.
*/
shs = mw_change_shapes(shs, image->nrow, image->ncol, image->gray[0]);
if (!shs) mwerror(FATAL,1,"Not enough memory to alloc the shapes !\n");

/*
  ... (do the computation) ...
*/

/*
  Delete the shapes
*/
if (!shs->data) free(shs->data);
mw_delete_shapes(shs);
```

○Name

mw_new_shapes - Create a `Shapes` structure

○Summary

Shapes mw_new_shapes()

○Description

This function creates a new `Shapes` structure. The fields are initialized to 0 or `NULL`value. The function returns the address of the new structure, or `NULL` if not enough memory is available.

○Example

```
Shapes shs;

/*
  Define the structure
*/
shs = mw_new_shapes();
if (!shs) mwerror(FATAL,1,"Not enough memory to define the shapes !\n");

/*
  At that time, the structure exists but is empty.
*/
```

7.3 Point with a type field

The `Point_type` structure is complementary to the `Point_curve` structure (See Section 6.1): it is used to record the type of the point, a valuable information in morphological shape-based algorithms. While the `Point_curve` structure was mainly defined to be used as part of a `Curve` structure, the `Point_type` structure is related to the `Morpho_line` structure (See Section 7.7).

7.3.1 The structure Point_type

This is the C definition of the structure:

```
typedef struct point_type {
  unsigned char type; /* Type of the point, e.g. (exact meaning can vary; See modules)
                         0 : regular point;
                         1 : point in the image's border;
                         2 : T-junction;
                         3 : Tau-junction;
                         4 : X-junction;
                         5 : Y-junction.
                         */
  struct point_type *previous; /* Pointer to the previous point (may be NULL) */
  struct point_type *next; /* Pointer to the next point (may be NULL) */
} *Point_type;
```

7.3.2 Related file (external) types

Not available: at this time, the `Point_type` object cannot be used as input/output variables of modules. It can be saved as part of a `Morpho_line` or `Fmorpho_line` structure.

7.3.3 Functions Summary

The following is a description of all the functions related to the `Point_type` type. The list is in alphabetical order.

○Name

mw_change_point_type - Define the point_type structure, if not defined

○Summary

Point_type mw_change_point_type(pt)

Point_type pt;

○Description

This function returns a `Point_type` structure if the input `pt = NULL`. It is provided despite the `mw_new_point_type()` function for global coherence with other memory types.

The function `mw_change_point_type` returns `NULL` if not enough memory is available to allocate the structure. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

Since the MegaWave2 compiler allocates structures for input and output objects (See Volume one: "MegaWave2 User's Guide"), this function is normally used only for internal objects. Do not forget to deallocate the internal structures before the end of the module, except if they are part of an input or output chain.

○Example

```
Point_type pt=NULL; /* Internal use: no Input neither Output of module */

/* Define a point type as image border */

pt = mw_change_point_type(pt);
if (pt == NULL) mwerror(FATAL,1,"Not enough memory.\n");
pt->type = 1; /* image border */
```

○Name

mw_copy_point_type - Copy all point types starting from the given one

○Summary

Point_type mw_copy_point_type(in,out)

Point_type in, out;

○Description

This function copies the current point type and the next point types contained in the chain defined at the starting point type `in`. The result is put in `out`, which may not be a predefined structure : in case of `out=NULL`, the `out` structure is allocated.

The function `mw_copy_point_type` returns `NULL` if not enough memory is available to perform the copy, or `out` elsewhere. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

○Example

```
Point_type in; /* Predefined point */
Point_type out=NULL;

out=mw_copy_point_type(in,out);
if (!out) mwerror(FATAL,1,"Not enough memory.\n");
```

○Name

mw_delete_point_type - Deallocate the point_type structure

○Summary

void mw_delete_point_type(pt)

Point_type pt;

○Description

This function deallocates the `Point_type` structures starting from the given `pt`, including this point itself. You should set `pt = NULL` after this call since the address pointed by `pt` is no longer valid. To deallocate a point only and not all the next points of the chain, just use `free(pt)`.

○Example

```
/* Remove the first point_type of an existing morpho_line */

Morpho_line ll; /* Existing morpho_line (e.g. Input of module) */
Point_type pt;  /* Internal use */

pt = ll->first_type;
ll->first_type=pt->next;
pt->next->previous = NULL;
free(pt);
pt = NULL;

/* Remove all point_type of an existing morpho_line */

mw_delete_point_type(ll->first_type);
```

○Name

mw_new_point_type - Create a new point_type structure

○Summary

Point_type mw_new_point_type();

○Description

This function creates a new `Point_type` structure. It returns `NULL` if not enough memory is available to create the structure. Your code should check this value to send an error message in the `NULL` case, and do appropriate statement.

Since the MegaWave2 compiler allocates structures for input and output objects (See Volume one: "MegaWave2 User's Guide"), this function is normally used only for internal objects. Do not forget to deallocate the internal point structures before the end of the module, except if they are part of an input or output curve.

○Example

```
/* Insert the point (0,0) with type 1 at the end of an existing morpho_line */

Morpho_line ll;  /* Existing morpho_line (e.g. Input of module) */
Point_curve point,p;  /* Internal use: no Input neither Output of module */
Point_type pt,t;

/* Define the point (0,0) with type 1 */
point = mw_new_point_curve();
if (point == NULL) mwerror(FATAL,1,"Not enough memory.\n");
pt = mw_new_point_type();
if (pt == NULL) mwerror(FATAL,1,"Not enough memory.\n");
point->x = point->y = 0;
pt->type=1;

/* Find the last point of the morpho_line */
p = ll->first_point; t = ll->first_type;
while (p->next) {p=p->next; t=t->next;}

/* Insert the point */
p->next = point;
t->next = pt;
point->previous = p;
pt->previous = t;

/* Do not deallocate point_curve and point_type or morpho_line will become inconsistent */
```

7.4 Horizontal segment

The `Hsegment` structure is useful for describing all pixels belonging to a (connected or non-connected) set, without taking the border into consideration. An horizontal segment is given by a left and a right point. If the shape of the set is more height than width, you should rather use vertical segments (not yet defined). The morpho set defined in Section 7.5 makes the use of the `Hsegment` structure, which defines an horizontal segment.

7.4.1 The structure Hsegment

This is the C definition of the structure `Hsegment`:

```
typedef struct hsegment {
  int xstart; /* Left x-coordinate of the segment */
  int xend;   /* Right x-coordinate of the segment */
  int y;      /* y-coordinate of the segment */
  struct hsegment *previous; /* Pointer to the previous segment (may be NULL) */
  struct hsegment *next;     /* Pointer to the next segment (may be NULL) */
} *Hsegment;
```

7.4.2 Related file (external) types

Not available: at this time, the `Hsegment` object cannot be used as input/output variables of modules. It can be saved as part of a `Morpho_set` structure.

7.4.3 Functions Summary

The following is a description of all the functions related to the `Hsegment` type. The list is in alphabetical order.

○Name

mw_change_hsegment - Define the hsegment structure, if not defined

○Summary

Hsegment mw_change_hsegment(seg)

Hsegment seg;

○Description

This function returns a `Hsegment` structure if the input `seg = NULL`. It is provided despite the `mw_new_hsegment()` function for global coherence with other memory types.

The function `mw_change_hsegment` returns `NULL` if not enough memory is available to allocate the structure. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

Since the MegaWave2 compiler allocates structures for input and output objects (See Volume one: "MegaWave2 User's Guide"), this function is normally used only for internal objects. Do not forget to deallocate the internal structures before the end of the module, except if they are part of an input or output chain.

○Example

```
Hsegment seg=NULL; /* Internal use: no Input neither Output of module */

/* Define the horizontal segment (0,10)-(200,10) */

seg = mw_change_hsegment(seg);
if (seg == NULL) mwerror(FATAL,1,"Not enough memory.\n");
seg->xstart=0;
seg->xend=200;
seg->y=10;
```

○Name

mw_delete_hsegment - Deallocate a chain of horizontal segments

○Summary

void mw_delete_hsegment(seg)

Hsegment seg;

○Description

This function deallocates the chain of horizontal segments starting from `seg`. Previous segments are not deallocated. You should set `seg = NULL` after this call since the address pointed by `seg` is no longer valid.

○Example

```
Hsegment seg0,newseg,oldseg;
int i;

/* Create a chain of 10 horizontal segments, starting from seg0 */

if (!(seg0=mw_new_hsegment())) mwerror(FATAL,1,"Not enough memory.\n");
seg0->xstart=0; seg0->xend=200; seg0->y=1;
oldseg=seg0;
for (i=2; i<=10; i++)
{
  if (!(newseg=mw_new_hsegment())) mwerror(FATAL,1,"Not enough memory.\n");
  newseg->xstart=0; newseg->xend=200; newseg->y=i;
  newseg->previous=oldseg;
  oldseg->next=newseg;
  oldseg=newseg;
}

/* .
   .
   (statement)
   .
   .
*/

/* Deallocate the chain of segments */
mw_delete_hsegment(seg0);
```

○Name

mw_new_hsegment - Create a new hsegment structure

○Summary

Hsegment mw_new_hsegment()

○Description

This function returns a new `Hsegment` structure, or `NULL` if not enough memory is available to allocate the structure. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

The new structure is created with fields set to 0 or `NULL`.

○Example

```
Hsegment seg; /* Internal use: no Input neither Output of module */

/* Define the horizontal segment (0,10)-(200,10) */

if (!(seg=mw_new_hsegment())) mwerror(FATAL,1,"Not enough memory.\n");
seg->xstart=0;
seg->xend=200;
seg->y=10;
```

7.5 Morpho set

We call morpho set any connected component of set of the form $\{x : \lambda_1 \leq u(x) \leq \lambda_2\}$, this set being noted in short by $[\lambda_1 \leq u \leq \lambda_2]$. Notice that for $\lambda_1 = -\infty$ we get a lower level set and for $\lambda_2 = +\infty$ an upper level set. In the case $\lambda_1 = \lambda_2$ the morpho set will be called iso set. The structure `Morpho_set` can be used to handle such morpho set. A `Morpho_set` is given by a list of horizontal segments (See Section 7.4), where levels λ_1 and λ_2 are recorded. Some additional information can be recorded, such as the neighbor morpho sets. Please notice that some fields are likely to change in the future.

7.5.1 The structure Morpho_set

This is the C definition of the structure `Morpho_set`:

```
typedef struct morpho_set {
  unsigned int num;      /* Morpho set number (range in the Morpho_sets struct.) */
  Hsegment first_segment; /* Pointer to the first segment of the morpho set */
  Hsegment last_segment;  /* Pointer to the last segment of the morpho set */
  float minvalue;        /* Minimum gray level value of this set */
  float maxvalue;        /* Maximum gray level value of this set */
  unsigned char stated;  /* 1 if this m.s. has already been stated, 0 otherwise */
  int area;              /* Area of the set (number of pixels belonging to this set) */
  struct morpho_sets *neighbor; /* Pointer to a chain of neighbor morpho sets (may be NULL)*/
} *Morpho_set;
```

7.5.2 Related file (external) types

1. `"MW2_MORPHO_SET"` MegaWave2 binary format.

7.5.3 Functions Summary

The following is a description of all the functions related to the `Morpho_set` type. The list is in alphabetical order.

○Name

mw_change_morpho_set - Define a morpho set, if not already defined

○Summary

Morpho_set mw_change_morpho_set(ms)

Morpho_set ms;

○Description

This function returns a `Morpho_set` structure if the input `ms = NULL`. It is provided despite the `mw_new_morpho_set()` function for global coherence with other memory types.

The function `mw_change_morpho_set` returns `NULL` if not enough memory is available to allocate the structure. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

Since the MegaWave2 compiler allocates structures for input and output objects (See Volume one: "MegaWave2 User's Guide"), this function is normally used only for internal objects. Do not forget to deallocate the internal structures before the end of the module, except if they are part of an input or output chain.

○Example

```
Morpho_set ms=NULL; /* Internal use: no Input neither Output of module */
Hsegment seg=NULL;

/* Define a morpho set containing one segment only */

if (!(seg=mw_change_hsegment(seg)) ||
    !(ms=mw_change_morpho_set(ms))) mwerror(FATAL,1,"Not enough memory.\n");
seg->xstart=0;
seg->xend=200;
seg->y=10;

ms->first_segment=seg;
ms->minvalue=0.0;
ms->maxvalue = 1.0;
ms->area=201;
```

○Name

mw_copy_morpho_set - Copy a morpho set into another one

○Summary

Morpho_set mw_copy_morpho_set(in,out)

Morpho_set in, out;

○Description

This function copies the `Morpho_set in` into `out`. The chain of segments are also duplicated. The result is put in `out`, which may not be a predefined structure : in case of `out=NULL`, the `out` structure is allocated.

The function `mw_copy_morpho_set` returns `NULL` if not enough memory is available to perform the copy, or `out` elsewhere. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

○Example

```
Morpho_set in; /* Predefined morpho_set */
Morpho_set out=NULL;

out=mw_copy_morpho_set(in,out);
if (!out) mwerror(FATAL,1,"Not enough memory.\n");
```

○Name

mw_delete_morpho_set - Deallocate a morpho set

○Summary

void mw_delete_morpho_set(ms)

Morpho_set ms;

○Description

This function deallocates the `Morpho_set ms`, including the chain of horizontal segments. You should set `ms = NULL` after this call since the address pointed by `ms` is no longer valid.

○Example

```
Morpho_set ms; /* Internal use: no Input neither Output of module */
Hsegment seg;

/* Define a morpho set containing one segment only */

if (!(seg=mw_new_hsegment()) ||
    !(ms=mw_new_morpho_set())) mwerror(FATAL,1,"Not enough memory.\n");
seg->xstart=0;
seg->xend=200;
seg->y=10;

ms->first_segment=seg;
ms->minvalue=0.0;
ms->maxvalue = 1.0;
ms->area=201;

/* .
   .
   (statement)
   .
   .
*/

/* Deallocate the morpho_set */
mw_delete_morpho_set(ms);
```

○Name

mw_length_morpho_set - Return the number of segments a morpho set contains

○Summary

unsigned int mw_length_morpho_set(ms)

Morpho_set ms;

○Description

This function returns the number of segments contained in the input `ms`. It returns 0 if the structure is empty or undefined.

○Example

```
Morpho_set ms=NULL; /* Internal use: no Input neither Output of module */
Hsegment seg=NULL;

/* Define a morpho set containing one segment only */

if (!(seg=mw_change_hsegment(seg)) ||
    !(ms=mw_change_morpho_set(ms))) mwerror(FATAL,1,"Not enough memory.\n");
seg->xstart=0;
seg->xend=200;
seg->y=10;

ms->first_segment=seg;
ms->minvalue=0.0;
ms->maxvalue = 1.0;
ms->area=201;

/* This will print 1 */
printf("%d",mw_length_morpho_set(ms));
```

○Name

mw_new_morpho_set - Create a new morpho set

○Summary

Morpho_set mw_new_morpho_set()

○Description

This function returns a new `Morpho_set` structure, or `NULL` if not enough memory is available to allocate the structure. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

The new structure is created with fields set to 0 or `NULL`.

○Example

```
Morpho_set ms; /* Internal use: no Input neither Output of module */
Hsegment seg;

/* Define a morpho set containing one segment only */

if (!(seg=mw_new_hsegment()) ||
    !(ms=mw_new_morpho_set())) mwerror(FATAL,1,"Not enough memory.\n");
seg->xstart=0;
seg->xend=200;
seg->y=10;

ms->first_segment=seg;
ms->minvalue=0.0;
ms->maxvalue = 1.0;
ms->area=201;
```

7.6 Chain of morpho sets

The `Morpho_sets` structure is useful to record a set (or chain) of morpho sets. This structure is used by the `Mimage` structure (See Section 7.9) to handle all the morpho sets an image contains.

7.6.1 The structure Morpho_sets

This is the C definition of the structure `Morpho_sets`:

```
typedef struct morpho_sets {
  Morpho_set morphoset;         /* Pointer to the current morpho set */
  struct morpho_sets *previous; /* Pointer to the previous morpho sets of the chain */
  struct morpho_sets *next;     /* Pointer to the next morpho sets of the chain */
  /* For use in Mimage only */
  struct morpho_line *morpholine;  /* Pointer to the associated morpho line */
} *Morpho_sets;
```

7.6.2 Related file (external) types

1. `"MW2_MORPHO_SETS"` MegaWave2 binary format.

7.6.3 Functions Summary

The following is a description of all the functions related to the `Morpho_sets` type. The list is in alphabetical order.

○Name

mw_change_morpho_sets - Define a morpho sets, if not already defined

○Summary

Morpho_sets mw_change_morpho_sets(mss)

Morpho_sets mss;

○Description

This function returns a `Morpho_sets` structure if the input `mss = NULL`. It is provided despite the `mw_new_morpho_sets()` function for global coherence with other memory types.

The function `mw_change_morpho_sets` returns `NULL` if not enough memory is available to allocate the structure. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

Since the MegaWave2 compiler allocates structures for input and output objects (See Volume one: "MegaWave2 User's Guide"), this function is normally used only for internal objects. Do not forget to deallocate the internal structures before the end of the module, except if they are part of an input or output chain.

○Example

```
Morpho_sets mss=NULL; /* Internal use: no Input neither Output of module */
Hsegment seg;
Morpho_set ms;

/* Define a morpho sets containing one morpho set */

if (!(seg=mw_new_hsegment()) ||
    !(ms=mw_new_morpho_set()) ||
    !(mss=mw_change_morpho_sets(mss))) mwerror(FATAL,1,"Not enough memory.\n");
seg->xstart=0;
seg->xend=200;
seg->y=10;

ms->first_segment=seg;
ms->minvalue=0.0;
ms->maxvalue = 1.0;
ms->area=201;

mss->morphoset=ms;
```

○Name

mw_copy_morpho_sets - Copy a morpho sets into another one

○Summary

Morpho_sets mw_copy_morpho_sets(in,out)

Morpho_sets in, out;

○Description

This function copies the `Morpho_sets in` into `out`. The `Morpho_set` pointed by the `in->morphoset` field is not only copied, but also all the chain starting from `in`. The neighbor `Morpho_sets` pointed by each `Morpho_set` are also copied. The result is put in `out`, which may not be a predefined structure : in case of `out=NULL`, the `out` structure is allocated.

The function `mw_copy_morpho_sets` returns `NULL` if not enough memory is available to perform the copy, or `out` elsewhere. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

○Example

```
Morpho_sets in; /* Predefined morpho_sets */
Morpho_sets out=NULL;

out=mw_copy_morpho_sets(in,out);
if (!out) mwerror(FATAL,1,"Not enough memory.\n");
```

○Name

mw_delete_morpho_sets - Deallocate a morpho sets

○Summary

void mw_delete_morpho_sets(mss)

Morpho_sets mss;

○Description

This function frees the `Morpho_set mss->morphoset`, all the chain starting from `mss` and it deallocates the `Morpho_sets mss` structure. You should sets `mss = NULL` after this call since the address pointed by `mss` is no longer valid.

○Example

```
Morpho_sets mss; /* Internal use: no Input neither Output of module */
Hsegment seg;
Morpho_set ms;

/* Define a morpho sets containing one morpho set */

if (!(seg=mw_new_hsegment()) ||
    !(ms=mw_new_morpho_set()) ||
    !(mss=mw_new_morpho_sets())) mwerror(FATAL,1,"Not enough memory.\n");
seg->xstart=0;
seg->xend=200;
seg->y=10;

ms->first_segment=seg; ms->minvalue=0.0; ms->maxvalue = 1.0; ms->area=201;
mss->morphoset=ms;

/* .
   .
   (statement)
   .
   .
*/

/* Deallocate the morpho_sets mss ; ms and seg will be also deallocated. */
mw_delete_morpho_set(mss);
```

○Name

mw_length_morpho_sets - Return the number of morpho sets a Morpho_sets structure contains

○Summary

unsigned int mw_length_morpho_sets(mss)

Morpho_sets mss;

○Description

This function returns the number of morpho sets the `Morpho_sets` structure `mss` contains, starting the chain from the current position given by `mss`. It returns 0 if the structure is empty or undefined.

○Example

```
Morpho_sets mss; /* Internal use: no Input neither Output of module */
Hsegment seg;
Morpho_set ms;

/* Define a morpho sets containing one morpho set */

if (!(seg=mw_new_hsegment()) ||
    !(ms=mw_new_morpho_set()) ||
    !(mss=mw_new_morpho_sets())) mwerror(FATAL,1,"Not enough memory.\n");
seg->xstart=0;
seg->xend=200;
seg->y=10;

ms->first_segment=seg; ms->minvalue=0.0; ms->maxvalue = 1.0; ms->area=201;
mss->morphoset=ms;

/* This will print 1 */
printf("%d",mw_length_morpho_sets(mss));
```

○Name

mw_new_morpho_sets - Create a new morpho sets

○Summary

Morpho_sets mw_new_morpho_sets()

○Description

This function returns a new `Morpho_sets` structure, or `NULL` if not enough memory is available to allocate the structure. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

The new structure is created with fields set to 0 or `NULL`.

○Example

```
Morpho_sets mss; /* Internal use: no Input neither Output of module */
Hsegment seg;
Morpho_set ms;

/* Define a morpho sets containing one morpho set */

if (!(seg=mw_new_hsegment()) ||
    !(ms=mw_new_morpho_set()) ||
    !(mss=mw_new_morpho_sets())) mwerror(FATAL,1,"Not enough memory.\n");
seg->xstart=0;
seg->xend=200;
seg->y=10;

ms->first_segment=seg;
ms->minvalue=0.0;
ms->maxvalue = 1.0;
ms->area=201;

mss->morphoset=ms;
```

7.7 Morpho line

A morpho line is the border of a morpho set. Assuming a right choice of grid and point connectivity so that a Jordan's theorem follows, a morpho line is a closed curve dividing the grid in two connected components : the interior of the morpho set and the exterior one. Actually, because an image has a finite support, a morpho line may also intersects the image border : in such case, the curve remains open. There is another restriction to the Jordan's theorem : most of modules using morpho lines (such as `ml_extract`) consider the 4-connectivity only in the square grid, so the border may cut the connected component to several pieces and the corresponding morpho lines may be self-intersecting. Notice that if the morpho set is a level set, the corresponding border is a level line. And if the morpho set is an iso set, its border is an iso line.

The structure `Morpho_line` can be used to handle such morpho line. First a `Morpho_line` is a curve, so the `Point_curve` structure is used to record it (field `first_point`). There are additional fields, to give information on the line (type of the points, closed or open curve) and to allow the reconstruction of the morpho set (`minvalue`, `maxvalue`).

7.7.1 The structure Morpho_line

This is the C definition of the structure `Morpho_line`:

```
typedef struct morpho_line {
  Point_curve first_point;/* Pointer to the first point of the morpho_line curve */
  Point_type first_type;  /* Pointer to the first Point_type */
  float minvalue;         /* Minimum gray level value of this morpho line */
  float maxvalue;         /* Maximum gray level value of this morpho line */
  unsigned char open;     /* 0 if the morpho line is closed, opened otherwise */
  float data;             /* User-defined data field (saved) */
  void *pdata;            /* User-defined data field : pointer to something (not saved) */

  /* For use in Mimage only */
  struct morpho_sets *morphosets;/* Pointer to the associated morpho sets */
  unsigned int num;              /* Morpho line number (range in the chain) */
  struct morpho_line *previous;  /* Pointer to the previous m.l. (may be NULL) */
  struct morpho_line *next;      /* Pointer to the next m.l. (may be NULL) */
} *Morpho_line;
```

7.7.2 Related file (external) types

1. `"MW2_MORPHO_LINE"` MegaWave2 binary format.

7.7.3 Functions Summary

The following is a description of all the functions related to the `Morpho_line` type. The list is in alphabetical order.

○Name

mw_change_morpho_line - Define a morpho line, if not already defined

○Summary

Morpho_line mw_change_morpho_line(ml)

Morpho_line ml;

○Description

This function returns a `Morpho_line` structure if the input `ml = NULL`. It is provided despite the `mw_new_morpho_line()` function for global coherence with other memory types.

The function `mw_change_morpho_line` returns `NULL` if not enough memory is available to allocate the structure. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

Since the MegaWave2 compiler allocates structures for input and output objects (See Volume one: "MegaWave2 User's Guide"), this function is normally used only for internal objects. Do not forget to deallocate the internal structures before the end of the module, except if they are part of an input or output chain.

○Example

```
/* Copy the curve of a morpho line into another morpho line */
Morpho_line in,out=NULL;

out=mw_change_morpho_line(out);
if (!out) mwerror(FATAL,1,"Not enough memory !\n");
out->open = in->open;
if ( ((out->first_point = mw_new_point_curve()) == NULL) ||
     ((out->first_type = mw_new_point_type()) == NULL) )
   mwerror(FATAL, 1,"Not enough memory !\n");
mw_copy_point_curve(in->first_point,out->first_point);
mw_copy_point_type(in->first_type,out->first_type);
```

○Name

mw_copy_morpho_line - Copy a morpho line into another one

○Summary

Morpho_line mw_copy_morpho_line(in,out)

Morpho_line in, out;

○Description

This function copies the `Morpho_line in` into `out`. All fields are copied but the following : `pdata`, `morphosets`, `num`, `previous` and `next`. The result is put in `out`, which may not be a predefined structure : in case of `out=NULL`, the `out` structure is allocated.

The function `mw_copy_morpho_line` returns `NULL` if not enough memory is available to perform the copy, or `out` elsewhere. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

○Example

```
Morpho_line in; /* Predefined morpho_line */
Morpho_line out=NULL;

out=mw_copy_morpho_line(in,out);
if (!out) mwerror(FATAL,1,"Not enough memory.\n");
```

○Name

mw_delete_morpho_line - Deallocate a morpho line

○Summary

void mw_delete_morpho_line(ml)

Morpho_line ml;

○Description

This function deallocates the `Morpho_line` `ml` structure, including the curve (`Point_curve`) and the chain of types (`Point_type`). Other pointers are not deallicated. You should line `ml = NULL` after this call since the address pointed by `ml` is no longer valid.

○Example

```
Morpho_line ml; /* Internal use: no Input neither Output of module */
Point_curve pt;

/* Define a morpho line containing the point (0,0) only */

if (!(pt=mw_new_point_curve()) ||
    !(ml=mw_new_morpho_line())) mwerror(FATAL,1,"Not enough memory.\n");
pt->x=pt->y=0;
ml->first_point=pt;

/* .
   .
   (statement)
   .
   .
*/

/* Deallocate the morpho_line */
mw_delete_morpho_line(ml);
```

○Name

mw_length_morpho_line - Return the number of points a morpho line contains

○Summary

unsigned int mw_length_morpho_line(ml)

Morpho_line ml;

○Description

This function returns the number of points contained in the input `ml`. It returns 0 if the structure is empty or undefined. If the field `first_type` is not `NULL`, the number of points defined by this field must equal the number of points in the curve.

○Example

```
Morpho_line ml; /* Internal use: no Input neither Output of module */
Point_curve pt;

/* Define a morpho line containing the point (0,0) only */

if (!(pt=mw_new_point_curve()) ||
    !(ml=mw_new_morpho_line())) mwerror(FATAL,1,"Not enough memory.\n");
pt->x=pt->y=0;
ml->first_point=pt;

/* This will print 1 */
printf("%d",mw_length_morpho_line(ml));
```

○Name

mw_new_morpho_line - Create a new morpho line

○Summary

Morpho_line mw_new_morpho_line()

○Description

This function returns a new `Morpho_line` structure, or `NULL` if not enough memory is available to allocate the structure. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

The new structure is created with fields set to 0 or `NULL`.

○Example

```
/* Copy the curve of a morpho line into another morpho line */
Morpho_line in,out;

out=mw_new_morpho_line();
if (!out) mwerror(FATAL,1,"Not enough memory !\n");
out->open = in->open;
if ( ((out->first_point = mw_new_point_curve()) == NULL) ||
     ((out->first_type = mw_new_point_type()) == NULL) )
   mwerror(FATAL, 1,"Not enough memory !\n");
mw_copy_point_curve(in->first_point,out->first_point);
mw_copy_point_type(in->first_type,out->first_type);
```

7.8 Morpho line in the continuous plane

The structure `Fmorpho_line` is used to handle morpho lines in the continuous plane. Indeed, if the morpho lines obtained from digital images contain discrete (integer) coordinates, one may want to process the morpho lines using continuous operators, such as geometric smoothing. The resulting morpho lines are no more made by discrete coordinates. In a `Fmorpho_line`, the points are recorded using the `Point_fcurve` structure (See Section 6.6).

7.8.1 The structure Fmorpho_line

This is the C definition of the structure `Fmorpho_line`:

```
typedef struct fmorpho_line {
  Point_fcurve first_point;/* Pointer to the first point of the fmorpho_line curve */
  Point_type first_type;   /* Pointer to the first Point_type */
  float minvalue;          /* Minimum gray level value of this morpho line */
  float maxvalue;          /* Maximum gray level value of this morpho line */
  unsigned char open;      /* 0 if the morpho line is closed, opened otherwise */
  float data;             /* User-defined data field (saved) */
  void *pdata;            /* User-defined data field : pointer to something (not saved) */

  /* For use in Mimage only */
  struct fmorpho_line *previous; /* Pointer to the previous m.l. (may be NULL) */
  struct fmorpho_line *next;     /* Pointer to the next m.l. (may be NULL) */
} *Fmorpho_line;
```

7.8.2 Related file (external) types

1. `"MW2_FMORPHO_LINE"` MegaWave2 binary format.

7.8.3 Functions Summary

We won't waste space to describe functions related to the `Fmorpho_line` structure : they are the same than those related to `Morpho_line`, except that the name "morpho_line" has to be changed to "fmorpho_line".

7.9 Morphological image

A morphological image may record in a structure called `Mimage` all morpho sets and morpho lines the image contains. It is therefore potentially a very redundant (and very huge) structure, but this plenty of information may be useful to perform morphological operations. Of course, not all fields need to be set as the same time, for example a `Mimage` may contain the level lines only. But from this (complete) information, all other fields may be computed.

The `Mimage` structure has been created before the `Shapes` structure was developed (See Section 7.2). It does not use the tree structure associated to FLST-based algorithms. For this reason, the `Shapes` object should be preferred to the `Mimage` one for future developments.

7.9.1 The structure Mimage

This is the C definition of the structure `Mimage`:

```
typedef struct mimage {
  char cmt[mw_cmtsize];   /* Comments */
  char name[mw_namesize]; /* Name of the set */
  int nrow;               /* Number of rows (dy) */
  int ncol;               /* Number of columns (dx) */
  float minvalue;         /* Minimal Gray level value in the image */
  float maxvalue;         /* Maximal Gray level value in the image */
  Morpho_line first_ml;   /* Pointer to the first morpho line in the discrete grid */
  Fmorpho_line first_fml; /* Pointer to the first morpho line in the continuous plane */
  Morpho_sets first_ms;   /* Pointer to the first morpho sets in the discrete grid */
} *Mimage;
```

7.9.2 Related file (external) types

1. `"MW2_MIMAGE"` MegaWave2 binary format.

7.9.3 Functions Summary

The following is a description of all the functions related to the `Mimage` type. The list is in alphabetical order.

○Name

mw_change_mimage - Define a morphological image, if not already defined

○Summary

Mimage mw_change_mimage(mi)

Mimage mi;

○Description

This function returns a `Mimage` structure if the input `mi = NULL`. It is provided despite the `mw_new_mimage()` function for global coherence with other memory types.

The function `mw_change_mimage` returns `NULL` if not enough memory is available to allocate the structure. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

Since the MegaWave2 compiler allocates structures for input and output objects (See Volume one: "MegaWave2 User's Guide"), this function is normally used only for internal objects. Do not forget to deallocate the internal structures before the end of the module, except if they are part of an input or output chain.

○Example

```
/* Copy the morpho lines only of a morphological image into another morphological image */

Mimage in,out=NULL;

out=mw_change_mimage(out);
if (!out) mwerror(FATAL,1,"Not enough memory !\n");
out->nrow = in->nrow;
out->ncol = in->ncol;
out->minvalue=in->minvalue;
out->maxvalue=in->maxvalue;
if (in->firstml)
  {
    out->firstml=mw_copy_morpho_line(in->firstml, out->firstml);
    if (!out->firstml) mwerror(FATAL, 1,"Not enough memory !\n");
  }
```

○Name

mw_copy_mimage - Copy a morphological image into another one

○Summary

Mimage mw_copy_mimage(in,out)

Mimage in, out;

○Description

This function copies the `Mimage in` into `out`. All fields are copied, including the chains of `Morpho_sets`, `Morpho_line` and `Fmorpho_line`. The result is put in `out`, which may not be a predefined structure : in case of `out=NULL`, the `out` structure is allocated.

The function `mw_copy_mimage` returns `NULL` if not enough memory is available to perform the copy, or `out` elsewhere. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

○Example

```
mimage in; /* Predefined mimage */
mimage out=NULL;

out=mw_copy_mimage(in,out);
if (!out) mwerror(FATAL,1,"Not enough memory.\n");
```

○Name

mw_delete_mimage - Deallocate a morphological image

○Summary

void mw_delete_mimage(mi)

mimage mi;

○Description

This function deallocates the `Mimage mi` structure, including the chains of `Morpho_sets`, `Morpho_line` and `Fmorpho_line`. You should line `mi = NULL` after this call since the address pointed by `mi` is no longer valid.

○Example

```
Mimage mi; /* Internal use: no Input neither Output of module */

Morpho_line ml;
Point_curve pt;
Fmorpho_line fml;
Point_fcurve fpt;
Morpho_sets mss;
Hsegment seg;
Morpho_set ms;


/* Define a morpho line containing the point (0,0) only */

if (!(pt=mw_new_point_curve()) ||
    !(ml=mw_new_morpho_line())) mwerror(FATAL,1,"Not enough memory.\n");
pt->x=pt->y=0;
ml->first_point=pt;

/* Define a fmorpho line containing the point (0.5,0.5) only */

if (!(fpt=mw_new_point_fcurve()) ||
    !(fml=mw_new_fmorpho_line())) mwerror(FATAL,1,"Not enough memory.\n");
fpt->x=fpt->y=0.5;
fml->first_point=fpt;

/* Define a morpho sets containing one morpho set */
```

```
if (!(seg=mw_new_hsegment()) ||
    !(ms=mw_new_morpho_set()) ||
    !(mss=mw_new_morpho_sets())) mwerror(FATAL,1,"Not enough memory.\n");
seg->xstart=0;
seg->xend=200;
seg->y=10;
ms->first_segment=seg; ms->minvalue=0.0; ms->maxvalue = 1.0; ms->area=201;
mss->morphoset=ms;

/* Define a morphological image made by one morpho line, one fmorpho line and
   one morpho sets.
*/

if (!(mi=mw_new_morpho_line())) mwerror(FATAL,1,"Not enough memory.\n");
mi->first_ml=ml;
mi->first_fml=fml;
mi->first_ms=ms;

/* .
   .
   (statement)
   .
   .
*/


/* Deallocate the mimage, including ml, fml and ms */
mw_delete_mimage(mi);
```

○Name

mw_length_fml_mimage - Return the number of morpho lines a morphological image contains

○Summary

unsigned int mw_length_fml_mimage(mi)

Mimage mi;

○Description

This function returns the number of fmorpho lines contained in the input `mi`. It returns 0 if the structure is empty or undefined.

○Example

See example page 203.

○Name

mw_length_ml_mimage - Return the number of morpho lines a morphological image contains

○Summary

unsigned int mw_length_ml_mimage(mi)

Mimage mi;

○Description

This function returns the number of morpho lines contained in the input `mi`. It returns 0 if the structure is empty or undefined.

○Example

```
Mimage mi; /* Internal use: no Input neither Output of module */

Morpho_line ml;
Point_curve pt;
Fmorpho_line fml;
Point_fcurve fpt;
Morpho_sets mss;
Hsegment seg;
Morpho_set ms;


/* Define a morpho line containing the point (0,0) only */

if (!(pt=mw_new_point_curve()) ||
    !(ml=mw_new_morpho_line())) mwerror(FATAL,1,"Not enough memory.\n");
pt->x=pt->y=0;
ml->first_point=pt;

/* Define a fmorpho line containing the point (0.5,0.5) only */

if (!(fpt=mw_new_point_fcurve()) ||
    !(fml=mw_new_fmorpho_line())) mwerror(FATAL,1,"Not enough memory.\n");
fpt->x=fpt->y=0.5;
fml->first_point=fpt;

/* Define a morpho sets containing one morpho set */
```

```
if (!(seg=mw_new_hsegment()) ||
    !(ms=mw_new_morpho_set()) ||
    !(mss=mw_new_morpho_sets())) mwerror(FATAL,1,"Not enough memory.\n");
seg->xstart=0;
seg->xend=200;
seg->y=10;
ms->first_segment=seg; ms->minvalue=0.0; ms->maxvalue = 1.0; ms->area=201;
mss->morphoset=ms;

/* Define a morphological image made by one morpho line, one fmorpho line and
   one morpho sets.
*/

if (!(mi=mw_new_morpho_line())) mwerror(FATAL,1,"Not enough memory.\n");
mi->first_ml=ml;
mi->first_fml=fml;
mi->first_ms=ms;


/* This will print 1 */
printf("%d",mw_length_ml_mimage(mi));
/* This will print 1 */
printf("%d",mw_length_fml_mimage(mi));
/* This will print 1 */
printf("%d",mw_length_ms_mimage(mi));
```

○Name

mw_length_ms_mimage - Return the number of morpho sets a morphological image contains

○Summary

unsigned int mw_length_ms_mimage(mi)

Mimage mi;

○Description

This function returns the number of morpho sets contained in the input `mi`. It returns 0 if the structure is empty or undefined.

○Example

See example page 203.

○Name

mw_new_mimage - Create a new morphological image

○Summary

Mimage mw_new_mimage()

○Description

This function returns a new `Mimage` structure, or `NULL` if not enough memory is available to allocate the structure. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

The new structure is created with fields set to 0 or `NULL`.

○Example

```
/* Copy the morpho lines only of a morphological image into another morphological image */

Mimage in,out;

out=mw_new_mimage();
if (!out) mwerror(FATAL,1,"Not enough memory !\n");
out->nrow = in->nrow;
out->ncol = in->ncol;
out->minvalue=in->minvalue;
out->maxvalue=in->maxvalue;
if (in->firstml)
  {
    out->firstml=mw_copy_morpho_line(in->firstml, out->firstml);
    if (!out->firstml) mwerror(FATAL, 1,"Not enough memory !\n");
  }
```

8 Unstructured material or raw data

When none of the previous structures matches your need, or when you want to write or to read files in a format which is not recognized by MegaWave2, use the raw data type : this internal type allows you to load/save any kind of data from/to disk.

8.1 The structure Rawdata

The `Rawdata` structure is nothing else than an array of bytes (`data` field). The size of the array is set in the `size` field.

```
typedef struct rawdata {
  int size;               /* Number of samples */
  unsigned char *data;    /* data field */
} *Rawdata;
```

8.2 Related file (external) types

There is no file types associated to the `Rawdata` structure : when the content of a `Rawdata` variable is written into a file, the content of the file is exactly the content of the `data` field. There is no header added. Consequently, file of any format can be loaded into a `Rawdata` variable. If this file contains a header (as most of MegaWave2 file formats), the header will be loaded into the `data` field together with the data themselves. Of course, there cannot be any conversion format associated to `Rawdata`.

8.3 Functions Summary

The following is a description of all the functions related to the `Rawdata` type. The list is in alphabetical order.

○Name

mw_alloc_rawdata - Allocate the data array of a `Rawdata` structure

○Summary

Rawdata mw_alloc_rawdata(rd,size)

Rawdata rd;

int size;

○Description

This function allocates the `data` array of a `Rawdata`structure previously created using `mw_new_rawdata`. The size of the data is given by `size`, it corresponds to the number of bytes.

Values can be addressed after this call, if the allocation successed. There is no default values.

Do not use this function if `rd` has already an allocated array: use the function `mw_change_rawdata` instead.

The function `mw_alloc_rawdata` returns `NULL` if not enough memory is available to allocate the array. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

○Example

```
Rawdata rd=NULL; /* Internal use: no Input neither Output of module */
int i;

/* Create a rawdata of 1000 bytes */
if  ( ((rd = mw_new_rawdata()) == NULL) ||
      (mw_alloc_rawdata(rd,1000) == NULL) )
    mwerror(FATAL,1,"Not enough memory.\n");

/* Set the byte #i to the value i mod 256 */
for (i=0;i<rd->size;i++) rd->data[i] = i % 256;
```

○Name

mw_change_rawdata - Change the size of the data array of a `Rawdata` structure

○Summary

Rawdata mw_change_rawdata(rd, newsize)

Rawdata rd;

int newsize;

○Description

This function changes the memory allocation of the `data` array of a `Rawdata` structure, even if no previously memory allocation was done. The new size of the array is given by `newsize`, it corresponds to the number of allocated bytes.

The function `mw_change_rawdata` can also create the structure if the input `rd = NULL`. Therefore, this function can replace both `mw_new_rawdata` and `mw_alloc_rawdata`. It is the recommended function to allocate `Rawdata` variables used as input/output of modules. Since the function can set the address of `rd`, the variable must be set to the return value of the function (See example below).

The function `mw_change_rawdata` returns `NULL` if not enough memory is available to allocate the array. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

○Example

```
Rawdata Output; /* Output of module */

/* Set the size of the array to be 1000 bytes */
Output = mw_change_rawdata(Output, 1000);
if (Output == NULL) mwerror(FATAL,1,"Not enough memory.\n");
```

○Name

mw_copy_rawdata - Copy the data of a `Rawdata+` structure into another one

○Summary

void mw_copy_rawdata(in, out)

Rawdata in,out;

○Description

This function copies the content of the array `data` of the `Rawdata+` structure `in` into the array `data` of `out`. The variable `out` must be an allocated `Rawdata` structure of same size than `in`.

The speed of this function depends to the C library implementation, but it is usually very fast (trying to do faster is a waste of time).

○Example

```
Rawdata G; /* Needed Input */
Rawdata F; /* Optional Output */

  if (F) {
    printf("F option is active: copy G in F\n");
    if ((F = mw_change_rawdata(F, G->size)) == NULL)
        mwerror(FATAL,1,"Not enough memory.\n");
    else mw_copy_rawdata(G, F);
   }
  else  printf("F option is not active\n");
```

○Name

mw_delete_rawdata - Deallocate the data array of a `Rawdata` structure

○Summary

void mw_delete_rawdata(rd)

Rawdata rd;

○Description

This function deallocates the array `values` of a `Rawdata` structure previously allocated using `mw_alloc_rawdata` or `mw_change_rawdata`, and the structure itself.

You should set `rd = NULL` after this call since the address pointed by `rd` is no longer valid.

○Example

```
Rawdata rd=NULL; /* Internal use: no Input neither Output of module */

if  ( ((rd = mw_new_rawdata()) == NULL) ||
      (mw_alloc_rawdata(rd,1000) == NULL) )
    mwerror(FATAL,1,"Not enough memory.\n");
/* .
   .
   (statement)
   .
   .
*/
mw_delete_rawdata(rd);
rd = NULL;
```

○Name

mw_new_rawdata - Create a new `Rawdata` structure

○Summary

Rawdata mw_new_rawdata();

○Description

This function creates a new `Rawdata` structure with an empty `data` array and `size` field set to 0. No data can be addressed at this time. The `data` should be allocated using the function `mw_alloc_rawdata` or `mw_change_rawdata`.

Do not use this function for input/output of modules, since the MegaWave2 Compiler already created the structure for you if you need it (See Volume one: "MegaWave2 User's Guide"). Use instead the function `mw_change_rawdata`. Do not forget to deallocate the internal structures before the end of the module.

The function `mw_new_rawdata` returns `NULL` if not enough memory is available to create the structure. Your code should check this value to send an error message in the `NULL` case, and do appropriate statement.

○Example

```
Rawdata rd=NULL; /* Internal use: no Input neither Output of module */

if  ( ((rd = mw_new_rawdata()) == NULL) ||
      (mw_alloc_rawdata(rd,1000) == NULL) )
    mwerror(FATAL,1,"Not enough memory.\n");
```

9 Miscellaneous Features

You will find in this section some utilities which may help you to write your modules. Contrary to the former sections, some functions described here are not about a memory format.

9.1 Global System Variables

At any time in a module, you can access to the following external variables. Those variables are for reading only, do not change their values ! Notice that you don't have to define those variables in your module, the definitions are done into the include file `mw.h`.

- `char *mwname` : This variable contains the name of the current module.
- `char *mwgroup` : This variable contains the group name of the current module, as for example `"common/signal"` which means that the current module belongs to the subgroup `signal` which is part of the main group `common`.
- `int mwerrcnt` : Give the number of time the function `mwerror` has be called with the argument `ERROR` (see section 9.3 page 215). Since `ERROR` is not a fatal event, the user has the possibility to terminate the algorithm by checking `mwerrcnt`, if too many errors have been encountered.
- `int mwrunmode` : The value of this variable indicates in which context the module is executed.
  - If set to 1, the module is called in the run-time mode;
  - if set to 2, the module is called by the window-oriented interpreter (XMegaWave2).

9.2 Conversion between memory types

The System Library contains functions to convert memory types. However do not expect to find a function to convert structures which are very dissimilar, as `Curves` and `Cimage`. If the meaning one can give of a conversion is not evident or not unique, a conversion procedure has to be implemented as a module rather than as a system function.

Conversion function summaries follow the following rule : `out = (Y) mw_x_to_y(in,old)` where `x` is the internal C type of the input `in`, `y` the internal C type of the requested output `out` (letters in lowercase) and `Y` the cast to the output (internal C type of `out` with first letter in uppercase). In the last argument `old` you may put the name of a variable of type `Y` : in such a case, the memory allocation will be reused for `out` (the pointer `old` will have the same address than `out`). This is especially useful when converting lot of images with same size, to avoid memory blowup. If you do not want to use this possibility, just put `NULL` as the last argument : memory for `out` will be allocated.

In addition to the various `mw_x_to_y()` undocumented conversion functions, there exists an "all-purpose" conversion function called `mw_conv_internal_type()` and documented next page.

○Name

mw_conv_internal_type - Convert any possible internal type to another one

○Summary

void *mw_conv_internal_type(mwstruct,typein,typeout)

void *mwstruct; /* Any type of MegaWave2 structure */

char *typein; /* Type of the input ¡mwstruct¿ */

char *typeout; /* Type of the output structure */

○Description

This function may be used instead of the `mw_x_to_y()` various functions to convert any possible internal type `a` to `b`, even if the `mw_a_to_b()` function does not exist : the system creates `mw_conv_internal_type()` by analyzing existing `mw_x_to_y()` functions, by finding the shortest path between two internal types, say `a` and `b`, and by calling appropriate `mw_x_to_y()` functions (for example, `mw_a_to_c()` and `mw_c_to_b()` if those functions exist).

The input `mwstruct` is a variable of internal C type given by the string `typein` (use lower letters only). The output of the function is a variable of internal C type given by the string `typeout`, or `NULL` if the conversion is impossible.

Do not forget to cast the output to the right type.

○Example

```
Ccimage in;
Cimage out1;
Fimage out2;

/* The line */
out1 = (Cimage) mw_conv_internal_type(in,"ccimage","cimage");
/* is equivalent to */
out1 = (Cimage) mw_ccimage_to_cimage(in);

if (out1==NULL) mwerror(FATAL,1,"Cannot convert Ccimage to Cimage !\n");

/* But to convert a Ccimage to a Fimage you shall use */
out2 = (Fimage) mw_conv_internal_type(in,"ccimage","fimage");
/* Since the following function does not exist at this time */
out2 = (Fimage) mw_ccimage_to_fimage(in);

if (out2==NULL) mwerror(FATAL,1,"Cannot convert Ccimage to Fimage !\n");
```

9.3 Miscellaneous System Functions

The following is a description of some miscellaneous system functions which may be of interest for the user. The list is in alphabetical order.

The most usefull are `mwerror` and `mwdebug`. Please notice that you need to process any error (especially memory allocation failure) by displaying an error message using `mwerror`, and by doing appropriate statement.

Some other functions are about dynamic memory allocation. They are important, since you are discouraged to use static memory allocation (as `double data[10000]`), but you may skip their description if you are familiar with the standard C dynamic memory functions.

○Name

mwcalloc - Dynamic memory allocation

○Summary

void *mwcalloc (nelem, elsize)

unsigned nelem, elsize;

○Description

This function allocates space for an array of `nelem` elements, each of size `elsize` bytes, and initializes the space to zeros.

This function returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object. Do not forget to cast the return value to the right type of your variable (see example below). If not enough memory is available to allocate the array, the function returns `NULL`. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

Each space allocated by `mwcalloc` must be deallocated using `mwcfree` before exiting the module.

Notice that in the MegaWave2 modules, the standard C function `calloc` is redefined to be `mwcalloc`. Therefore, if you use `calloc` in your code you actually call `mwcalloc`.

○Example

```
float *data=NULL; /* Internal use: no Input neither Output of module */

/* Allocates space for 1000 samples of float values */
if  ( (data = (float *) mwcalloc (1000, sizeof(float))) == NULL )
    mwerror(FATAL,1,"Not enough memory.\n");

/* Set the sample #i to the value i */
for (i=0;i<1000;i++) data[i] = i;
```

○Name

mwcfree - Dynamic memory deallocation

○Summary

void mwcfree (ptr)

char *ptr;

○Description

This function deallocates the space pointed to by `ptr` and which has previously been allocated by `mwcalloc`. It does nothing if `ptr` = `NULL`.

You should set `ptr` to `NULL`after this call since the address pointed to by `ptr` is no longer valid.

Notice that in the MegaWave2 modules, the standard C function `cfree` is redefined to be `mwcfree`. Therefore, if you use `cfree` in your code you actually call `mwcfree`.

○Example

```
float *data=NULL; /* Internal use: no Input neither Output of module */

/* Allocates space for 1000 samples of float values */
if  ( (data = (float *) mwcalloc (1000, sizeof(float))) == NULL )
    mwerror(FATAL,1,"Not enough memory.\n");
.
. (statement)
.
/* End of statement: deallocation of the array */
mwcfree((char *) data);
```

○Name

mwdebug - print if debug

○Summary

void mwdebug(format, ...);

char *format;

○Description

This function prints its arguments in ... under control of the format in `format`, exactly in the same manner that the standard C function `printf` does. The string `<dbg>` is added to the beginning of the line.

The print is active only when the module has been called with the debugging option on.

○Example

```
Fimage image;
int x,y;

for (x=0;x<image->ncol;x++) for (y=0;y<image->nrow;y++)
{
  mwdebug("processing pixel (%d,%d)...\n",x,y);
  .
  . (statement)
  .
}
```

○Name

mwerror - print an error message

○Summary

void mwerror(type, exit_code, format, ...);

int type;

int exit_code;

char *format;

○Description

This function prints its arguments in ... under control of the format in `format` on the standard error output, in the same manner that the standard C function `fprintf(stdout,format,...)` does.

A message is added to the print, and an action may be performed, according to the value in `type` :

- `WARNING` : the additional message is `MegaWave warning (`*mwname*`) :` (following is the requested print);
- `ERROR` : the additional message is `MegaWave error (`*mwname*`) :` (following is the requested print), and the variable `mwerrcnt` is incremented.
- `FATAL` : the additional message is `MegaWave fatal (`*mwname*`) :` (following is the requested print), and a call to `mwexit(exit_code)` is performed.
- `INTERNAL` : the additional message is `MegaWave internal (`*mwname*`) :` (following is the requested print), and a call to `mwexit(exit_code)` is performed. Use it when such error normally never may occur. Then, such event points out a fault of the algorithm and the code should be fixed. One uses to add in the beginning of the print the text `[X]` where `X` is the name of the function where the error has been found, in order to make easier the debugging process (see example below).
- `USAGE` : after the requested print, is printing the usage of the module. Use it when the input values you get in your module function does not correspond to what the usage requests.

○Example

```
/* Compute some norm of any fimage */
static float fnorm(image)
Fimage image;

{ float norm;  /* result of the computation */
  .
  .  (statement)
```

```
  .
  if (norm < 0.0)
   mwerror(INTERNAL,1,"[fnorm] Negative norm value computed ! (norm=%f)",norm);
  else return(norm);
}
```

○Name

mwexit - Module termination

○Summary

void mwexit (status)

int status;

○Description

This function causes normal program termination of a MegaWave2 module. The variable `status` indicates the status of the module when the termination occurred; value 0 means successful termination, other values are user-dependent.

Notice that in the MegaWave2 modules, the standard C function `exit` is redefined to be `mwexit`. Therefore, if you use `exit` in your code you actually call `mwexit`.

○Example

```
Fimage image; /* Output of module */

/* Try several times an allocation of a fimage of size 256x256 */
while (mw_alloc_fimage(image,256,256) == NULL)
  {
     mwerror(ERROR,1,"Not enough memory !\n");
     if (mwerrcnt > 10) mwexit(-1);
     sleep(2);  /* Wait 2 seconds */
  }
```

○Name

mwfree - Dynamic memory deallocation

○Summary

void mwfree (ptr)

char *ptr;

○Description

This function deallocates the space pointed to by `ptr` and which has previously been allocated by `mwmalloc`. It does nothing if `ptr` = `NULL`.

You should set `ptr` to `NULL`after this call since the address pointed to by `ptr` is no longer valid.

Notice that in the MegaWave2 modules, the standard C function `free` is redefined to be `mwfree`. Therefore, if you use `free` in your code you actually call `mwfree`.

○Example

```
float *data=NULL; /* Internal use: no Input neither Output of module */

/* Allocates space for 1000 samples of float values */
if  ( (data = (float *) mwmalloc (1000*sizeof(float))) == NULL )
    mwerror(FATAL,1,"Not enough memory.\n");
.
. (statement)
.
/* End of statement: deallocation of the array */
mwfree((char *) data);
```

○Name

mwmalloc - Dynamic memory allocation

○Summary

void *mwmalloc (size)

size_t size;

○Description

This function allocates space for a block of at least size bytes, but does not initialize the space.

This function returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object. Do not forget to cast the return value to the right type of your variable (see example below). If not enough memory is available to allocate the array, the function returns `NULL`. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

Each space allocated by `mwmalloc` must be deallocated using `mwfree` before exiting the module.

Notice that in the MegaWave2 modules, the standard C function `malloc` is redefined to be `mwmalloc`. Therefore, if you use `malloc` in your code you actually call `mwmalloc`.

○Example

```
long *data=NULL; /* Internal use: no Input neither Output of module */

/* Allocates space for 5000 samples of long values */
if  ( (data = (long *) mwmalloc (5000*sizeof(long))) == NULL )
    mwerror(FATAL,1,"Not enough memory.\n");

/* Set the sample #i to the value i */
for (i=0;i<5000;i++) data[i] = i;
```

○Name

mwrealloc - Dynamic memory re-allocation

○Summary

void *mwrealloc(ptr, size)

char *ptr; unsigned size;

○Description

This function changes the size of the block pointed to by `ptr` to `size` bytes and returns a pointer to the (possibly moved) block. Existing contents are unchanged up to the lesser of the new and old sizes. If `ptr` is a `NULL`pointer, `mwrealloc` behaves like `mwmalloc` for the specified size. If `size` is zero and `ptr` is not a `NULL`pointer, the object it points to is freed and `NULL`is returned.

If not enough memory is available to allocate the array, the function returns `NULL`. Your code should check this return value to send an error message in the `NULL` case, and do appropriate statement.

Do not forget to cast the return value to the right type of your variable, and to cast the type of the input pointer `ptr` to be `char *` (see example below).

Each space allocated by `mwrealloc` must be deallocated using `mwfree` before exiting the module.

Notice that in the MegaWave2 modules, the standard C function `realloc` is redefined to be `mwrealloc`. Therefore, if you use `realloc` in your code you actually call `mwrealloc`.

○Example

```
long *ldata=NULL;   /* Internal use: no Input neither Output of module */
double *ddata=NULL;

/* Allocates space for 5000 samples of long values */
if  ( (ldata = (long *) mwmalloc (5000*sizeof(long))) == NULL )
    mwerror(FATAL,1,"Not enough memory.\n");
.
.
.
/* Re-allocates space for 2000 samples of double values, using space allocated
   for ldata
*/
if  ( (ddata = (double *) mwrealloc ((char *) ldata, 2000*sizeof(double))) == NULL )
    mwerror(FATAL,1,"Not enough memory.\n");

/* Warning : do not use anymore the array ldata ! */
```

10 Wdevice Library and window facilities

The Wdevice library provides an interface to the window manager: it helps the user to write modules which have to access to the window manager resources, as the screen, the mouse, etc. It not only replaces some painful operations which require a lot of code (such opening a window, mapping the content of an image into a window, etc.) to a simple call to one function, but it provides also an interface which is independant to the type of the window system: the calls to the Wdevice functions remain the same even if the window system changes (and the result should be the same).

This library is independant to the MegaWave2 System Library although some modules cannot be linked without it : it is added when needed during the link process of a MegaWave2 command. Of course, one Wdevice library per window system is needed. At this time, there exists a Wdevice library for the X Window System Version 11 (X11) only. In the past, one could find a Wdevice library for the Suntools System but, because of the renunciation of Suntools from Sun MicroSystem, this library is no longer maintained (and no longer distributed).

On can found in the system library some packages that use functions defined by Wdevice to perform more high-levels tasks, such as Wpanel : The Wpanel (Panel display facilities) is a small package that allows to handle buttons and bars. It is not documented yet, and will probably change quite much in the future. For the time being, it is used only in the module `llview.c`.

10.1 Functions Summary

The following is a description of the Wdevice library functions which may be called by the user. The list is in alphabetical order.

You may notice that each function name begins with the letter `W`.

Warning: the functions summary is not documented yet. If you need to access to the screen into a MegaWave2 module (e.g. to draw some figure, etc.) please read the code of the following public MegaWave2 modules, and take inspiration from those:

- `view_demo.c`;
- `cview.c`;
- `ccview.c`;
- `cmview.c`;
- `splot.c`;
- `readpoly.c`.

Nevertheless, and because those MegaWave2 modules already exist, you are not likely to really need to learn about the Wdevice library.

Index