LastWave 3.0b C-API

Emmanuel Bacry
CMAP, Ecole polytechnique, 91128 Palaiseau Cedex, France
email : lastwave@cmap.polytechnique.fr
web : http://www.cmap.polytechnique.fr/~ bacry/LastWave

This documentation includes description of

- **LastWave Kernel 3.0**, Author: E.Bacry
- **Signal package 3.0**, Authors: E.Bacry, N.Decoster and X.Surraud
- **Image package 3.0**, Authors: E.Bacry and J.Fraleu

GNU GENERAL PUBLIC LICENSE TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

Version 2, June 1991

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

- a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
- b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
- c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

- a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

- b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
- c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.
If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.
4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.
5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.
6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.
7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.
If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.
It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.
This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.
8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail

to address new problems or concerns.
Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.
10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Contents

Chapter 1

For LastWave 1.xxx users

As you already know, there are pretty big changes between the 1.xxx version and version 2.0 of LastWave. Here is a list of changes in the C-API. **This is not an exhaustive list but it should help you modifying quickly your 1.xxx C-files in order to include them in LastWave 2.0**

1.1 Some major changes

1.1.1 About VALUE

One of the main changes in 2.0 is that the variable content (formerly AVARCONTENT) corresponds now to the VALUE type. The structure is very different. AVARCONTENT worked using a message system whereas VALUE works using *virtual* functions.
If you wrote, in 1.xxx, your own AVARCONTENT, you will have to convert the whole structure to a VALUE structure. You should read Section 5 to learn how to do it. The conversion will take a little time. But it is the major change in 2.0. All the other changes will be very easy and very fast to make.

1.1.2 About commands

- The type CCOMMAND has been replaced by the type CPROC
- The type SCOMMAND has been replaced by the type SPROC
- The type CCommandTable has been replaced by the type CProcTable and the type CCOMMANDTABLE by the type CPROCTABLE. To add a table of commands you should use now the void AddCProcTable(CProcTable *table) function instead of the former AddCCommandTable function
- The help of a C-command has slightly changed syntax (it is now the same as for a script procedure). In 1.xxx, the syntax was

```
"{{<action1 arg names>} {<action1 help>}} ...
  {{<actionN arg names>} {<actionN help>}}"
```

now, it is

```
"{{{<action1 arg names>} {<action1 help>}} ...
  {{<actionN arg names>} {<actionN help>}}}"
```

i.e., the former syntax has been surrounded by {...}

1.1.3 About tSTR, tSTR and tWORD, tWORDLIST

- You have to be aware that the type `tSTR` now correponds to an *evaluated* string. If you want to parse a non evaluated string you must use `tWORD` instead (same thing for `tLIST` which must be replaced by `tWORDLIST`). This has to be done for instance if you wrote some commands with actions. The action name must be now parsed using the type `tWORD` and not `tSTR` as before.
- The former parsing type `tWORD` which corresponded to a `word` in the `mp` package now corresponds to a non evaluated string.

1.1.4 Returning values

Be careful: when a command returns it must return a value whichcan be of *any* type. So calling

```
SetResultStr("1");
```

is no longer equivalent to

```
SetResultInt(1);
```

To learn about all the `SetResult...` functions you should read Section 4.2.2

1.1.5 If you wrote your own graphic object

- If you wrote your own graphic object, you must change the `set` method (see Section 6.2.5). The only change is that now there is a `field` argument which corresponds to the field (without the `-` at the beginning). In `1.xxx`, the field corresponded to `argv[0]`.

1.2 Some minor changes

- Try to avoid manipulating directly the `argv` variables. Though it will still work, in a forthcomoing version of LastWave, it will be forbidden. You should use the various `ParseArgv...` or `Parse<Type>` functions and the `ParseOption` function (see Section 3.9) for parsing `-option`.

Chapter 2

For LastWave 2.xxx users

2.1 By default, numbers are coded using doubles

This is a major change. Numbers in LastWave were coded using floats before. The main drawback was that in the script language, integer could not be too big (so it was a problem for instance to use a signal which was too long). Now integers in the script language are also coded double numbers, so you should not have any problem.

From a C-Developer point of view, it means that all the float variables should use the type `LWFLOAT` which by default is set to double but can be set back to float using the compiler option `-D NUMFLOAT`.

Chapter 3

Adding some new commands using C-language

3.1 Hello World

Let us write a new C-command that just prints "Hello World !" on the terminal window. We will write it in a file named `hello.c` in the `user` directory. The directory `user` is meant to contain all the files you write (unless you create a new package, in which case you should read Section 3.13). The directory `user/src` should contain all the `.c` files and the directory `user/include` should contain all the `.h` files. Initially the directory `user/src` contains only one file which name is `user.c`. This file contains a single C-function called `UserInit` which is called by LastWave at startup. This is where you will need to put the declarations of the new commands or packages you write.
As we said, we will write the command in the file `hello.c` (you could write it directly in the file `user.c`). Thus, the first thing to do is to create this file: create a new (empty file) named `hello.c` in the directory `user/src`. Then you should declare it in the list of files of the `user`: edit the file `user/obj/FileList` and change the line

```
OBJS = $(OBJDIR)user.$(OBJEXT)
```

into

```
OBJS = $(OBJDIR)user.$(OBJEXT) $(OBJDIR)hello.$(OBJEXT)
```

You are now ready to write the code for the command in the file `hello.c`

The command we will write will be called `hello` and will correspond to the C-function `C_Hello` that will be defined in the following way :

```
#include "lastwave.h"

C_Hello(char **argv)
{
  Printf("Hello World!\n");
}
```

All LastWave files should start by the include statement `#include "lastwave.h"`. In order to inherit of LastWave stream management (see Section 3.12), you should **never use the input/output standard C-functions** `printf`, `scanf`, `fprintf`... but only the one defined in LastWave whose names are the same as the names of the C-functions except that their first letter is a capital letter (e.g., `Printf`, `Scanf`, `FPrintf`,...).

All C-functions that are associated to LastWave commands must be declared as functions of one variable of type `char **`. This variable corresponds to the list of arguments the command is called with.

We must now declare to the LastWave interpreter the new command `hello` associated to this function. For that purpose, you need to create a new procedure table somewhere after the definition of the function `C_Hello`

```
CProc demoProcs[] = {

  "hello",C_Hello,"{{{} {A simple help string !}}}",

  NULL,NULL,NULL
};

CProcTable demoTable = {demoProcs, "demo", "Just some demo commands"};
```

A procedure table (of type `CProcTable`) has 3 fields :

- a list of C-procedure definitions (each of them being of type `CProc`) ended by 3 NULLs
- the name of the package it belongs to
- and a help describing this table.

Each procedure definition has the following syntax

```
<procedure name>,<C-function>,<description string>,
```

and the description string should follow the syntax

```
{{{<argument names>} {<help text>}}}
```

if the command is a simple command (with no actions) such as the command `setv` or `printf`. If the command accepts several actions (such as `file` or `stats`) then the syntax of the description string is :

```
{{{<action1 arg names>} {<action1 help>}}...{{<actionN arg names>} {<actionN help>}}}
```

You can look at the file `kernel/scr/commands.c` for looking at all the standard command definitions.

Remark: let us note that there is an alternative way for declaring the help string of a command. For the sake of convenience, you might want to put the help string in the same file as the command definition itself so that when you change the command API you can change directly the help string. This can be done by replacing the line

```
"hello",C_Hello,"{{{} {A simple help string !}}}"
```

by the line

```
"hello",C_Hello,NULL
```

i.e., by setting the help string to `NULL` in the table of procedures. Whenever a help string is `NULL` in a table of procedure, LastWave expects the command itself to return the help string when passed a `NULL` argument. Consequently, we would have to redefine the `C_Hello` command by

```
C_Hello(char **argv)
{
  if (argv == NULL) return("{{{} {A simple help string !}}}"

  Printf("Hello World!\n");
}
```

Now we have to declare LastWave that there is a new command table. As we have already mentionned it, this will be done in the `UserInit` functionof the file `user/src/user.c`. You should add, at the end of this function, a simple line in order to add the table `demoTable`, i.e.,

```
void UserInit(void)
{
...
...

AddCProcTable(demoTable);
}
```

That's it. You have defined your first new C-command. You just need to recompile LastWave: just type `make` in the `Makefile` directory. Then just type `hello` in LastWave terminal window!

Remark Let us note that, by calling several times the `AddCProcTable` function, you can add as many procedure tables as you want and whenever you want (not only at startup during the execution of the `UserInit` function).

Remark If several procedure tables define twice the same command name, only the last one will be kept. However, at startup, LastWave will warn you that a C-procedure was overwritten. In this way, if you want, you can redefine some LastWave commands

Remark If you made a syntax error in the description string of a command, LastWave will warn you at startup, at the very beginning of the terminal window (you might need to scroll back your terminal window, since LastWave prints a lot of things at startup).

3.2 Arguments - The `ParseArgv` function - The *tTYPE*'s

Let us write a command that has arguments. As we said, these arguments are contained in the variable `argv` of type `char **` in the definition of the C function associated to the command. The variable `argv[n]` corresponds to the `n`th argument. Moreover, the function at `N` arguments then `argv[N]` is `NULL`.

In LastWave version 2.0, you should never access the content of the variable `argv[n]` directly. You should always use the standard C-functions to read arguments.
In order to interpret these arguments as signals, listv's, integers, floats,... you basically need to use a single function named `ParseArgv`. This function allows you to read sequentially the variable `argv` and interpret it according to the types you specify. Its syntax is the following :

```
argv = ParseArgv(argv,<type1> ,<pointer1> ,...,<typeN> ,<pointerN> ,0 or -1);
```

The last argument should be 0 if you do not expect any other arguments in `argv` and -1 if you want to make another call to `ParseArgv` later. Thus for instance, if you want to read sequentially one string and one integer you would write :

```
char *str;
int i;
argv = ParseArgv(argv,tSTR,&str,tINT,&i,0);
```

The name of any type corresponds to a (#-defined number) and always start with the letter `t` and then all the other letters are capital letters.
After these lines have been executed,

- either LastWave succeeded in reading a string (that is set in `str`) and then an integer (that is set in `i`) and no other arguments
- or it had a problem either in reading the arguments or because there are other arguments left and thus LastWave generated an error and returned automatically from your function.

If you expect the integer to be strictly positive you could add the following line :

```
if (i<=0) Errorf("Sorry the second argument ( %d) is not positive",i);
```

The `Errorf` function allows you to generate an error (and to return from your function right away) while printing an error message using the same syntax as `printf`.

Remark Let us note that LastWave did perform some allocation for the string `str`. If an error occurs, you do not have to worry, LastWave takes care of desallocation! This means that if you want to keep the string (after the functions has returned) you must copy it.

3.3 Optional arguments - The *tTYPE_*'s

LastWave interpreter allows you to use optional arguments. To all types (e.g., tINT, tSTR,...) correspond optional types by just adding a _ at the end of their names (e.g., tINT_, tSTR_,...). When you want to use an optional type in `ParseArgv`, you must use the following syntax

```
argv = ParseArgv(argv,<optionalType1> ,<defaultValue1> ,<pointer1> ,...);
```

Thus if you want to read an integer and an optional float with a default value of 1, you should write

```
int i;
LWFLOAT f;
argv = ParseArgv(argv,tINT,&i,tFLOAT_,1.0,&f,0);
```

WARNING : Let us note that it is VERY important that you write `1.0` and not `1`. It would lead to a complete misunderstanding of your request.The default value specified after a `tFLOAT_` MUST BE of type `LWFLOAT` and not of type `int`. Actually a numeric expression (such as 1.0) is always of type `LWFLOAT`. However, if you pass a variable after the `tFLOAT_`, this variable must be of type `LWLOAT` whereas the pointer following the default value must be a `LWFLOAT *`.

3.4 Basic *tTYPE*'s

We have already seen some "basic" *tTYPE*'s. "Basic" in the sense that they do not correspond to C-structure types (as a signal or an image will be) but to basic C-types. Those basic *tTYPE*'s are

- `tINT`, `tINT_` : integers corresponding to the C-type `int`
- `tFLOAT`, `tFLOAT_` : LWFLOAT's corresponding to the C-type `double` (by default unless compilation option `-D NUMFLOAT` is used)
- `tDOUBLE`, `tDOUBLE_` : doubles corresponding to the C-type `double` whatever the compilation flag
- `tSTR`, `tSTR_`: strings corresponding to the C-type `char *`
- `tCHAR`, `tCHAR_`: characters corresponding to the C-type `char`
- `tWORD`, `tWORD_` : same as `tSTR` except that the argument is not evaluated
- `tLIST`, `tLIST_` : this *tTYPE* corresponds to the LastWave type `&list`, i.e., list of strings. It is stored in a C-type `char **` variable. Thus, if `char **list` is such a variable, `list[n]` corresponds to the `n`th string. If it has `N` elements then `list[N]` is `NULL`. (Let us note that, for efficiency purposes, whatever the length of the list is, only

2 memory allocations are performed: i) list corresponds to an allocation of `N char *` and ii) `list[0]` corresponds to an allocation of `L char` where `L` is the total size of all the strings.).

- `tWORDLIST`, `tWORDLIST_`: same as `tLIST` except that the argument is not evaluated.

3.5 The *Parse<type>* and *Parse<type>* _ functions

Let us note that the function `ParseArgv` returns `argv+n` where `n` is the number of arguments that were read. Actually, `ParseArgv` calls for each type a C function that tries to read the corresponding type from a string (namely `argv[0]` then `argv[1]`,...). The name of these functions are `Parse<type>` where `<type>` is the corresponding type (without the `t` at the beginning and with just the first letter as capital). Thus for instance

```
int i;
ParseInt(argv[0],&i);
```

will try to read an integer from the argument `argv[0]` and if it succeeds it will be put in `i` and if not an error will be generated. In the same way

```
int i,answer;
answer = ParseInt_(argv[0],-1,&i);
```

will try to read an optional integer in `i` with default value -1. If it succeeded in reading it, it will return `YES` (which is nothing but 1), if not it will return `NO` (which is 0) and it will set an error message without generating the error. If later, for any reason you do want to generate the corresponding error, then you can call the `Errorf1` function. This function performs the same thing as `Errorf` except that it inherits the previous error message (e.g., the one that was set by `ParseInt_`).

Remark Let us note that for each type `tTYPE` there exists a C-function `ParseType` (for reading a single argument) and a C-function `ParseType_` (for reading an optional argument).

For instance, if the arguments of the command are an arbitrary number of integers. You could do

```
int array[1000];
int n,i;

for (n=0;n<1000;n++) {
  if (ParseInt_(*argv,0,array+n) == NO) break;
  argv++;
}
if (*argv != NULL) Errorf1("");
```

Let us note that, with a few more lines, you could do the same thing using the `ParseArgv` only

```
int array[1000];
int n,i;
char **argv1;

for (n=0;n<1000;n++) {
  argv1 = ParseArgv(argv,tINT_,0,array+n,-1);
  if (argv1 == argv;) break
  argv = argv1;
}
if (*argv != NULL) Errorf("");
```

When you expect `argv` to be empty (i.e., no more arguments), you could call

```
NoMoreArgs(argv);
```

that will either generate an error ("Too many arguments...") or will just return.

3.6 Levels

As explained in the main LastWave manual, *levels* refers in LastWave to the different calling environments. Let us recall that the level 0 corresponds to the global environment (the one which is accessible in your terminal window wne you type in a command). Then each time a script command (not a C-command!) is called, LastWave creates a *level*, i.e., a local environment, in which this command will be executed. Thus, level 1 corresponds to the environment of the script procedure called by the global environment, level 2 correponds to the environment of the script command called by the level 1 environment and so on. You can also access to levels using relative reference. Thus, level -1 refers to the calling environment, level -2 to the one that called the calling environment and so on. Level 0 always corresponds to the current environment.
Levels in LastWave correspond to the type `LEVEL` (which is a pointer to the C-structure `Level`). You do not need to know about the specific structure of a level. You should just know a few things about levels:

- The global variable `levelCur` of type `LEVEL` always corresponds to the current level
- The macro `GetLevel(levelNum)` returns the level corresponding to the level number `levelNum`. Thus, for instance `GetLevel(0)` returns `levelCur`
- By default the functions `Parse<type>` and `Parse<type>_` parse the string argument in the current level. To each function `Parse<type>` (resp. `Parse<type>_`) corresponds a function `Parse<type>Level` (resp. `Parse<type>Level_`) that parses the string argument in the level `level` which is of type `LEVEL` and which is passed as the first argument of these functions.
- For parsing a level number you can use the two parsing functions

- `void ParseLevel(char *arg, LEVEL *level)`
- `char ParseLevel_(char *arg, LEVEL default, LEVEL *level)`

3.7 Managing Errors

Whenever an error is generated (implicitely by LastWave or explicitely by your code), the function returns right away, proper desallocation is performed (see Sections 3.10 and 4.2.1) and an error message is printed. We have seen that `ParseArgv` and the `Parse<type>` functions could (implicitely) generate an error. For generating explicitely an error, we have seen the functions `Errorf` and `Errorf1`.
If you want to print a complex error message (e.g., several lines) you could use the functions `SetErrorf` and `AppendErrorf` :

```
SetErrorf(<format> ,<arg1> ,...,<argN>);
```

erase any former error message and sets it to the new one (using the same syntax as `printf`), and the function

```
AppendErrorf(<format> ,<arg1> ,...,<argN>);
```

that just appends to the existing error message another one. Then, when you call `Errorf1` everything will be printed.

3.8 Returning a basic type value

For a command to return a value which corresponds to a basic type, you should use one of the functions

- `void SetResultInt(int i)`: to return the integer `i`
- `void SetResultFloat(LWFLOAT f)`: to return the LWFLOAT `f`
- `void SetResultStr(char *str)`: to return the string `str` (`str` will be copied)
- `void SetResultf(char *format,...)`: to return a formatted string (using `printf` format)
- `void AppendResultStr(char *str)`: to append the string `str` to the string result (allows to define the result in an incremental way)
- `void AppendResultf(char *format,...)`: to append a formatted string to the string result
- `void SetResultList(char **list)`: to return the list `list`
- `void AppendListResultStr(char *str)`: to append a string to a list result
- `void AppendListResultf(char *format,...)`: to append a formatted string to a list result.

Thus, for instance, if you want to write a command that takes 2 float arguments and that returns their sum, you would write

```
C_Sum(char **argv)
{
  LWFLOAT f1,f2;
  argv = ParseArg(argv,tFLOAT,&f1,tFLOAT,&f2,0);
  SetResultFloat(f1+f2);
}
```

3.9 Command -options

In order to manage LastWave - options, you should use the `ParseOption` function. After reading all the (eventually optional) arguments using the `ParseArgv` function or the `Parse<type>` functions, the variable `argv` should have been incremented so that it "points" to the first eventual option (or to just `NULL` if no option). Then, you should include a loop which looks like

```
char opt;
...
...
while (opt = ParseOption(&argv)) {
switch(opt) {
  case < opt1 >:
    eventual parsing of arguments after the option
    break;
  case < opt2 >:
    eventual parsing of arguments after the option
    break;
  ...
  ...
  default:
    ErrorOption(opt);
}
NoMoreArgs(argv);
...
...
```

The `void ErrorOption(char opt)` function allows to generate an error if the option character `opt` is not valid.
Thus for instance, if you want to add an option `-o` to the `C_Sum` function defined in the previous section so that it specifies (using a single `char` argument) what operator should be applied. The operator should be one of `+,-` or `*` (default is `+`). We should write

```
C_Sum(char **argv)
{
```

```
LWFLOAT f1,f2;
char operator, opt;

argv = ParseArg(argv,tFLOAT,&f1,tFLOAT,&f2,-1);

operator = '+';
while (opt = ParseOption(&argv)) {
switch(opt) {
  case 'o':
  argv = ParseArgv(argv,tCHAR_,'+',&operator,-1);
  if (operator != '+' && operator != '-' && operator != '*') Errorf("bad operator '%s'
  break;
  default:
  ErrorOption(opt);
}
NoMoreArgs(argv);

switch(operator) {
case '+':
  SetResultFloat(f1+f2);
  break;
case '-':
  SetResultFloat(f1-f2);
  break;
case '*':
  SetResultFloat(f1*f2);
  break;
}
```

3.10 Basic memory allocation

When you perform memory allocation, you should always test whether it succeeded or not. If you did not, you should generate the corresponding error. LastWave can handle it for you if you use LastWave allocation functions. It is strongly advised to do so. These functions are

- `void *Malloc(size_t size)`
- `void * TMalloc(size_t size)`
- `void *Realloc(void *ptr, size_t size)`
- `void *Calloc(int n, size_t size)`
- `char *CharAlloc(int size)`
- `LWFLOAT *FloatAlloc(int size)`

- `double *DoubleAlloc(int size)`
- `int *IntAlloc(int size)`
- `void Free(void * ptr)`

they correspond to the standard C-functions for allocation

3.11 Allocation of strings and lists - Temporary pointers

As you might have noticed in one of the example above, when `ParseArgv` reads a string, you do not need to specify a size for it. ParseArgv will perform automatic memory allocations. You do not need to bother with it. Whatever the size of the string, it will be managed properly. However, these allocations are temporary allocations that will be automatically destroyed when your function returns (including when it returns with an error). (Let us note that it works exactly in the same way with lists.)
If you need to, you can also declare that a pointer points to a temporary structure (that you dinamically allocated previously), i.e., that the corresponding desallocation will be taken care by LastWave when the command returns (with or without an error). To do so you just need to call the function

- `void TempPtr(void *ptr)`

or, if `ptr` is a dynamically allocated string, you can use equivalently

- `void TempStr(char *str)`

if it is a list (remember that two allocations are performed for lists as explained in Section 3.4) you should not call any of these functions. Instead, you should call

- `void TempList(char **list)`

If you need to keep in a global variable or in a static variable any of the allocated arguments that you read using `ParseArgv`, it is very important that you do not use it as it is, since it will be deleted when the last executed command returns. Thus you must copy it. To copy a string you can use

- the `char *CopyStr(char *str)` function
- or the `char *TCopyStr(char *str)` function, that copies the string and make the copy a temporary string.

In the same way for lists there are 2 functions

- the `char *CopyList(char **list)` function
- or the `char *TCopyList(char **list)` function.

Remark: Temporary pointers can be very convenient. Indeed, in a C-function, you have to keep track of all the dynamic allocation you perform in order to be sure that when it returns, everything that should be desallocated is desallocated. Thus each time the function

generates an error or each time the command `return` is used, you must perform all the corresponding desallocation. If a lot of memory allocations are needed, the tracking can be be pretty heavy to handle. The use of temporary pointers makes the tracking unnecessary.

Remark: as you will see in Section 4.2.3, the temporary pointer manager is able to handle desallocation of more sophisticated structures such as signals, images,....

Remark: Sometimes, you might want to use temporary pointers and, at the same time, to control when the desallocation will be done. In order to do so you can use the functions

- `void SetTempAlloc(void)` : when called, this function sets an allocation "marker"
- `void ClearTempAlloc(void)` : when called, this function desallocates all the temporary pointers that where allocated since the last "marker" and it deletes the marker.

It is important that to each call of `SetTempAlloc` should correspond a call of `ClearTempAlloc`. You are responsible for that!

3.12 Managing inputs/outputs - Streams

3.12.1 Using C-functions dealing with the standard streams

You should avoid using the C-standard functions for i/o in the terminal window. Indeed, since the terminal window is managed by LastWave and since possible redirections might have been asked by the user, in order not to interfere, you should always use LastWave C-functions. To replace the C i/o functions that deal with the standard streams (`stdin`, `stderr`, `stdout`) you can use

- `void Printf(char *format,...)`: to use instead of `printf()`
- `void PrintfErr(char *format,...)`: to use instead of `fprintf()` using `stderr`
- `long GetChar(void)`: to use instead of `getchar()`
- `int GetLine(char *str)`: to get a whole line
- `char Eof(void)`: to use instead of `eof()`

Let us note that, these functions can be used using different streams (see Section 3.12.3).

3.12.2 Streams in LastWave

Streams in LastWave are represented using the type `STREAM` which corresponds to a pointer to the C-structure `Stream`. You can read a stream as an argument of a command using the `tTYPE`'s `tSTREAM` or `tSTREAM_`. You can also use the parsing functions

- `void ParseStream(char *arg, STREAM *stream)`
- `char ParseStream_(char *arg, STREAM default, STREAM *stream)`

The standard streams are accessed through the global variables

- `STREAM _StdinStream`: the input stream corresponding to the terminal window (corresponds to `stdin` on Unix computers)
- `STREAM _StdoutStream`: the output stream corresponding to the terminal window (corresponds to `stdout` on Unix computers)
- `STREAM _StderrStream`: the output stream corresponding to the terminal window (corresponds to `stderr` on Unix computers)
- `STREAM _StdnullStream`: a null stream that does not do anything

The corresponding current standard streams (that might be redirected by the user) are

- `STREAM StdinStream`
- `STREAM StdoutStream`
- `STREAM StderrStream`
- `STREAM StdnullStream`

3.12.3 Functions that deal with `STREAM`'s

These are the main functions that let you handle `STREAM`'s:

- `STREAM OpenFileStream(char *filename,char *mode)`: lets you open a `STREAM` associated to the file `filename` (using unix syntax for directories) using the mode `mode` (same `mode` argument as the C-function `fopen()`)
- `STREAM OpenStringStream(char *str)`: lets you open an input `STREAM` associated to the string `str`
- `void CloseStream(STREAM stream)`: closes the `stream`
- `void FPrintf(STREAM stream, char *format,...)`: same as the C-function `fprintf()` excepts that it operates on `STREAM`s
- `long FGetChar(STREAM stream)`: get a character from `stream`
- `int FGetLine(STREAM stream, char *str)`: get a line from `stream`

3.12.4 Using `FILE *`

If, for managing i/o on files, you do not want to use `STREAM`s but you want to use the C-standard `FILE` type, you must be aware that, depending on the computer, the syntax for filenames (and for directories) is not the same. Thus if you want to use machine indpendant code, you should either `convert` the filename before using `fopen()` or directly use the `FOpen()` LastWave function:

- `FILE * FOpen(char *file, char * mode)`: same as `fopen()` except that the filename is converted before calling `fopen()`

- `int FClose(FILE *s)`: should be used instead of `fclose()` whenever you used `FOpen()` instead of `fopen()`
- char *ConvertFilename(char *filename): returns the `converted` filename that you can use with `fopen()` so tath your code will be machine independent.

3.13 Defining your own package

These are the first step you should follow to write your own package

- create a directory named `package_xxx` where `xxx` is the name of the package. This directory must be located in the main `LastWave` directory.
- create 3 subdirectories named `src` (for `.src` files), `include` (for `.h` files) and `obj`.
- copy the file `user/obj/Makefile` into the subdirectory `obj`.
- copy the file `user/obj/FileList` into the subdirectory `obj`. Edit it and update the list of files (as described at the beginning of Section 3.1)

One of the file in the `package_xxx/src` directory should contain the main definitions of the package you created. Generally this file is called `xxx_package.c`. This file should have two C-functions

- The function that will be called to load the package. Generally this function is called `void LoadXXXPackage(void)`. This is where you add new command tables using the `AddCProcTable` (as explained in Section 3.1) and new C-structure definitions that LastWave will know about (see Section 5).
- The function that declares the package and which is generally called `void DeclareXXXPackage(void)`. This function should call the `DeclarePackage` function. This last function has the following syntax

```
DeclarePackage(char *packageName, void (*loadFunction)(void), int year, char *versi
```

  where

  - `packageName` is the name of the package that will be used in LastWave (e.g., to load the package you will have to type `package load packageName`
  - `loadFunction` is the function that will be called for laoding the package (this is the function we just described and that is generally called `LoadXXXPackage`
  - `year` is the year the first version of the package was written
  - `version` is a string that characterizes the current version (e.g. `"2.1b"`)
  - `authors` is a string that gives the list of authors (e.g., "John Smith and Albert Dupond")
  - `onelineDescription` is a string that describes what is the use of the package

Once you have done all that, you can compile your package typing `make` in the `Makefiles` directory. Then next time you run LastWave, your package should be declared (try the `package list` command). In order to load your package you should just type `package load xxx` where `xxx` is the package name (not quoted!)

You can add script definitions to a package. Indeed, each time a package named `xxx` is loaded using the function `package load`, LastWave looks for a file `xxx/xxx.pkg` in the `scripts` directory. If the directory `xxx` exists and contains a file named `xxx.pkg`, this file is automatically sourced when the package is loaded.

Chapter 4

VALUE's in LastWave

4.1 Playing around with complex objects

4.1.1 A simple example dealing with signals - Reference counters

Let us see how LastWave lets you handle complex objects such as signals. Let us write a command that takes a single argument that is an input signal (i.e., an argument of type `&signali`) and returnsa new signal which is the same signal added with a Gaussian noise of variance 1. This is done in the following way

```
C_AddNoise(char **argv)
{
  SIGNAL sigIn,sigOut;
  int i;

  argv = ParseArgv(argv,tSIGNALI,&sigIn,0);

  sigOut = NewSignal();
  SizeSignal(sigOut,sigIn->size,YSIG);

  for (i=0;i<sigIn->size,i++) {
    sigOut->Y[i] = sigIn->Y[i]+Grand(1.0);
  }

  SetResultValue(sigOut);

  DeleteSignal(sigOut);
}
```

Let us comment, this piece of code. The type of a signal is `SIGNAL`. It corresponds to a pointer on a C-structure named `Signal`, however you will never need to use directly the type `Signal`, you will always deal with `SIGNAL`s. (Let us note that, in LastWave, when a type is all capitalized it means that it is a pointer). Thus the first line

```
SIGNAL sigIn,sigOut;
```

is a simple variable definition, the variable `sigIn` will be used to store (a pointer to) the input signal and the variable `sigOut` will be used to store (a pointer to) the resulting signal. The line

```
argv = ParseArgv(argv,tSIGNALI,&sigIn,0);
```

allows to parse the argument line. It evals the first argument of the command and states that it should be an input signal (let us recall that an `input` signal is a signal that should ot be empty). Moreover the `0` indicates that no other arguments are expected. Let us note that a signal of type `&signal` (i.e., not necessary non-empty) would have been obtained using the `tTYPE tSIGNAL`. Moreover as for the basic `tTYPE`'s, you could use the optional `tTYPE`'s: `tSIGNALI_` or `tSIGNAL_`. If you used these optional values you would have had to specify right after it a default value for `sigIn`. The "usual" default value is `NULL`. Thus, for instance, if the command had a single optional input signal argument, you would have written `argv = ParseArgv(argv,tSIGNALI_,NULL,&sigIn,0);` and then test whether `sigIn` is `NULL` or not.
It is important to understand that, at this point, *no allocation has been made.* `sigIn` just points to a signal that already existed. Now, we need to create the output signal. This is done by

```
sigOut = NewSignal();
```

The `NewSignal` function allocates one instance of the structure `Signal` and returns a pointer to this instance, i.e., a value of type `SIGNAL`. It is important to understand that, at this point, the signal is empty. Before filling it up, you need to allocate the float arrays that will contain the y-values and (in the case of an xy-signal) the x-values. This is done using the `SizeSignal` function. Its first argument is the signal on which the allocation should be performed, the second argument is the requested size of the signal and finally the last argument is either `YSIG` if the signal is an y-signal (in which case only the y-values will be allocated) or a xy-signal (in which case both the y-values and the x-values will be allocated). For the sake of simplicity, we supposed that we just want to deal with the y-values, thus we wrote

```
SizeSignal(sigOut,sigIn->size,YSIG);
```

The next step consists in filling up the output signal

```
for (i=0;i<sigIn->size,i++) {
  sigOut->Y[i] = sigIn->Y[i]+Grand(1.0);
}
```

Then, we need to specify to LastWave that the result value is the signal `sigOut` which is done by

```
SetResultValue(sigOut);
```

Let us note that this function does not copy the signal. It just adds one *reference* to the output signal. Lastwave uses *reference counters* for all of its "complex" structures. A reference counter is a field of the structure (in our case a field of the structure pointed by `SIGNAL`) that is a positive integer that is used to keep track of the number of variables that point to this structure. Each time a new reference to the structure is made, the counter is incremented by 1. Each time this reference is lost (or deleted) the counter is decremented by 1. When it reaches 0, LastWave knows that the structure must be desallocated.

Reference counter At this point of our code, the output signal has 2 references: one which corresponds to `sigOut` and another one (that is hidden to your code) that was created by the `SetResultValue` call. The rule in LastWave is that **you are always responsible for the references you** *own*. In our case, you are responsible for the `sigOut` reference. You *must* delete this reference before the end of the command. If you don't, the reference `sigOut` will never be deleted, thus, the counter will never reach 0 and consequently the signal it points to will never be desallocated. In order to delete the reference you own, you must use the `DeleteSignal` function:

```
DeleteSignal(sigOut);
```

The `DeleteSignal` function basically decreases the counter by 1 and desallocate the signal structure if it reaches 0 otherwise it just returns.
Let us note that it is very important that `DeleteSignal(sigOut);` is called *after* `SetResultValue(sigOut);` and not before. Indeed, if it was called before, since there would be a single reference to `sigOut` the call to `DeleteSignal(sigOut);` would desallocate the signal, so you would not be able to return it. (Basically, you have to think this way: since we deleted the reference `sigOut`, you cannot use the variable `sigOut` anymore).

Other `tTYPE`'s and `Parse` functions related to signals In the same way as for basic `tTYPE`'s (see Section 3.5), you could have used the function `ParseSignalI` instead of the general `ParseArgv` function, i.e., the line

```
argv = ParseArgv(argv,tSIGNALI,&sigIn,0);
```

would then have been replaced by the lines

```
ParseSignalI(*argv,&sigIn);
argv++;
NoMoreArgs(argv);
```

As for basic `tTYPE`'s (see Section 3.5), the following functions exist: `ParseSignalI`, `ParseSignalI_`, `ParseSignal` (for `&signal`) and `ParseSignal`.

4.1.2 Temporary signals

In the last example we saw that we had to call `DeleteSignal(sigOut);` before leaving the function. That means that if, you write a more complex command, that could generate errors, i.e., before the error is generated you `must` call `DeleteSignal(sigOut);`. This is possible (though not very practical) if it is the C-function you write that generates the error, however if your function calls another C-function that generates an error, LastWave will return *directly* from this last function, there is no way you can call `DeleteSignal(sigOut);`.

There is a very convenient way for not having to deal with the `DeleteSignal(sigOut);` call. You just need to declare to LastWave that the reference you own is temporary. As for temporary pointers to basic types (see Section 3.11), it basically means that LastWave will delete it when the command is over. In order to declare the reference `sigOut` to be temporary, you can either write

```
sigOut = NewSignal();
TempValue(sigOut);
```

or more concisely

```
sigOut = TNewSignal();
```

(the `T` in `TNewSignal` stands for *Temporary*). Thus the whole code becomes

```
C_AddNoise(char **argv)
{
  SIGNAL sigIn,sigOut;
  int i;

  argv = ParseArgv(argv,tSIGNALI,&sigIn,0);

  sigOut = TNewSignal();
  SizeSignal(sigOut,sigIn->size,YSIG);

  for (i=0;i<sigIn->size,i++) {
    sigOut->Y[i] = sigIn->Y[i]+Grand(1.0);
  }

  SetResultValue(sigOut);
}
```

You should use temporary references as much as you can, it generally leads to much smaller code.

Warning: You cannot use the `TempPtr` function as seen in Section 3.11 to make the reference `sigOut` temporary. Indeed `TempPtr` **does not deal with reference counters**. Making a pointer temporary using the `TempPtr` function will systematically lead (at the

end of the command) to desallocation of the memory it points to (using a simple `free` call). Making it temporary using `TempValue` assumes the pointer points to a structure that includes a reference counting system.

4.1.3 A simple example dealing with listvs

As for signals, a `&listv` is represented by the type `LISTV` which corresponds to a pointer to the structure `Listv`. For parsing listv arguments you can use the $tTYPE$'s `tLISTV` and `tLISTV_` and the parsing functions `ParseListv` and `ParseListv_`. Let us write a command that returns a listv whose elements are the y-values of an input signal (eventually empty):

```
C_Sig2Listv(char **argv)
{
  SIGNAL sigIn;
  LISTV lv;
  int i;

  argv = ParseArgv(argv,tSIGNAL,&signal,0)

  lv = TNewListv();
  SetLengthListv(lv,signal->size);

  for (i=0;i<signal->size;i++) {
    AppendFloat2Listv(lv,signal->Y[i]);
  }

  SetResultValue(lv);
}
```

We are not give detailed description for this code. It should be pretty clear. Let us just note that

- `TNewListv` allocates a listv structure and returns a pointer to this listv, i.e., a pointer of type `LISTV`. As for signals, the listv structure uses a reference counter. The call `TNewListv` makes the returned reference temporary. Another way to write the same thing would have been to call `NewListv` and then `TempValue`.
- The listv is initially created with a length of 0. `SetLengthListv` is used to specify the length.
- `AppendFloat2Listv` appends a float at the end of the listv.

4.2 About VALUE's

4.2.1 What is a VALUE?

LastWave commands are able to deal with *values* that could be of very different types. They can be argument of a command or can be returned by a command. In Chapter 3 we have

learned how to deal with some "basic" types such as floats or strings, i.e., types that do not require a C-structure to be implemented. For the sake of simplicity as well as for efficiency purposes, LastWave lets you deal with these basic types as regular C-types.

In the last section, we have seen *values* which corresponded to "high-level" C-structures: the `SIGNAL` value and the `LISTV` value. You might have noticed that some C-functions were able to work on both `SIGNAL` and `LISTV`. For instance, this is the case of the function `SetResultValue` that is used in a command to specify what value should be returned, or the function `TempValue` that is used to declare to LastWave that a reference is temporary. This is possible because those "high-level" structures "inherit" from a common structure, the `Value` structure. By "inherits", we mean that they have a common header, i.e., the C-structures starts with the same fields. This set of common fields are grouped into the `Value` structure. The type `VALUE` is the type which corresponds to a pointer to a `Value` structure.

As we will see later on, not only listvs or signals but also ranges, images, scripts, procedures ... all of them correspond to pointers to structures that have the same first fields, i.e., the fields of the structure `Value`. Thus they can all be considered as (i.e., casted to) `VALUE`.

Remark: In LastWave, the defined types always start with a capital letter (e.g., `Value`). Moreover, to most of the defined types corresponds a type which is a pointer to this type. This pointer type has the same name as the type it points to except that all the letters are capitalized (e.g., `VALUE`). Since, in LastWave, basically you only deals with pointers, you will only deal with the "all capitalized" types (e.g., `SIGNAL, IMAGE, LISTV, RANGE,` ...).

Thus, for instance, the definition of the signal structure will define two new types, the type `Signal` which corresponds to the C-structure and the `SIGNAL` type which corresponds to pointer on `Signal`. Thus it will look like

```
typedef struct signal {
  /* The common fields of all Value's */
  ValueFields;
  ...
  The specific fields for signals
  ...
} Signal, *SIGNAL;
```

The keyword `ValueFields` corresponds to a macro that defines the fields common to all `Value`'s.

Important remark: Actually, even strings and floats are stored internally using C-structures inheriting from `VALUE` (to learn about `STRVALUE` and `NUMVALUE`, see Sections 4.10 and 4.9). However, as we already explained, both for efficiency reasons and for the sake of simplicity, LastWave lets you handle basic types such as strings and floats using regular C-types. Thus, for instance, if your command should return a float value, instead of creating a `NUMVALUE` and using `SetResultValue`, you can directly call the `SetResultFloat` function.

4.2.2 Returning a VALUE - The SetResultValue function

As we have already seen, whenever a command should return a value you must use the function

```
SetResultValue(VALUE value);
```

Actually this is a macro, so you do not need to cast the value to the VALUE type. Thus, for instance you can write

```
SIGNAL sig;
SetResultValue(sig);
```

and you do not need to write

```
SIGNAL sig;
SetResultValue((VALUE) sig);
```

Let us note that the call to SetResultValue increments the reference counter of the VALUE by 1. To learn more about the reference counters you should first read the previous Sections 4.1.1, 4.1.2, 4.1.3 and then the next section.

4.2.3 Reference counting of VALUEs - Temporary VALUEs

Before reading this section you should read Sections 4.1.1, 4.1.2 and 4.1.3.
Each VALUE includes a *reference counter* that is used to count the number of references to this specific VALUE. Whenever you create a reference, you are responsible for it, i.e., you are responsible for deleting it. To delete a reference, you can either use the generic macro

```
DeleteValue(VALUE value);
```

(since it is a macro no cast is necessary) or (this is completely equivalent) you can use the specific functions such as DeleteSignal or DeleteListv. Each time you delete a reference the counter is decremented by 1. The C-structure the VALUE points to is desallocated as soon as the corresponding counter reaches 0.
Sometimes, tracking the references you create is the pain in the neck. You can avoid doing that by declaring the reference as a temporary reference. This is done using the function

```
void TempValue(VALUE reference);
```

A reference which is declared as temporary will be destroyed as soon as the current LastWave command ends. In Sections 4.1.1 we have alraedy seen an example of how to use the TempValue function.
Let us note that the references created by a *parsing* function (e.g., ParseArgv, ParseSignal, ParseSignal_) is automatically declared as temporary. So you are not responsible for it. Thus, in the simple example

```
C_Mean(char **argv)
{
  SIGNAL sig;
  LWFLOAT mean;

  argv = ParseArgv(argv,tSIGNALI,&sig,0);

  mean = 0;
  for (i=0;i<sig->size;i++) {
    mean += sig->Y[i];
  }
  mean /= sig->size;

  SetResultFloat(mean);
}
```

the reference `sig` that is created by the `ParseArgv` call is declared temporary by the `ParseArgv` function itself, you do not *own* the reference, so **you must not call** `DeleteValue(sig)`. In the case you need to keep a reference to the signal `sig` (e.g., you want to keep it in a `static` variable) you can increment the reference counter by 1 using the macro

```
AddRefValue(VALUE value);
```

(no cast is necessary).
In any case, the parsing functions in LastWave take care of the desallocation of the arguments they return. If they correspond to basic C-types, they are temporary pointers and consequently they will be desallocated automatically when the current command ends. If they correspond to `VALUE`s, they are temporary references of these `VALUE`s.

4.2.4 The `tVAL` and `tVALOBJ` *tTYPE*'s

You can parse a `VALUE` using the `ParseArgv` function, along with the `tVAL` type. Thus if you want to write a command that has a single argument which is of type `&val` (i.e., *anything* that can be evaluated) and display it, you would write

```
C_MyPrint(char **argv)
{
  VALUE *val;
  argv = ParseArgv(argv,tVAL,&val,0);
  PrintValue(val);
}
```

Let us note that this is exactly how the `print` command works. As we will explain later on, the function `PrintValue` basically sends the message `print` to the value `val`. Of course, an optional value can be specified using `tVAL_`. In the same way the type `&valobj` corresponds to the *tTYPE* `tVALOBJ` (or `tVALOBJ_` if optional).
Several parse functions can be used for parsing values (of any type). Using the standard syntax, these functions are

- `char ParseVal_(char *arg, VALUE defVal, VALUE *val)`
- `void ParseVal(char *arg, VALUE *val)`
- `char ParseValLevel_(LEVEL level, char *arg, VALUE defVal, VALUE *val)`
- `void ParseValLevel(LEVEL level, char *arg, VALUE *val)`
- `char ParseValObj_(char *arg, VALUE defVal, VALUE *val)`
- `void ParseValObj(char *arg, VALUE *val)`
- `char ParseValObjLevel_(LEVEL level, char *arg, VALUE defVal, VALUE *val)`
- `void ParseValObjLevel(LEVEL level, char *arg, VALUE *val)`

Remark: As you we have seen in the previous section, for reading values of specific types, e.g., signals, listvs, ..., you can use specific *tTYPE*'s such as `tSIGNAL` or `tLISTV`.

4.2.5 The `GetTypeValue` macro - Casting `VALUE`'s - Static string type

In order to get the `&type` of a VALUE, you must use the `GetTypeValue` macro. It returns a static string corresponding to the type. It is a static string so that you can make direct comparisons using `==`. The LastWave type &<type> corresponds to a static string type in C called <`type`>`Type`. Thus for instance, the `&listv` type corresponds to the static string `listvType`. When you know what the type of the value is you can cast it. However, in order to cast it, you must first use the `ValueOf` macro. Lets see how it works on a simple example. Let us write a command that takes a single argument that could be either a listv or a signal and that performs actions depending on what it is.

```
C_MyCmd(char **argv)
{
  LISTV lv;
  SIGNAL signal;
  VALUE val;
  char *type;

  argv = ParseArgv(argv,tVAL,&val,0);

  type = GetTypeValue(val);
```

```
  if (type == listvType) {
    lv = (LISTV) ValueOf(val);
    ...
    ...
  }
  else if (type == signalType || type == signaliType) {
    signal = (SIGNAL) ValueOf(val);
    ...
    ...
  }
  else Errorf("Bad type '%s' for argument",type);
}
```

It is clear that, most of the time, the arguments of a command have a definite type. In that case, it is much easier to just use the `ParseArgv` function using the right *tTYPE*'s. The type `VALUE` is only useful when an argument could be of different types. Let us note that, even in this latter case, you can avoid using `VALUE`, e.g.,

```
C_MyCmd(char **argv)
{
  LISTV lv;
  SIGNAL signal;

  argv = ParseArgv(argv,tSIGNAL_,NULL,&signal,0);

  if (signal == NULL) {
    argv = ParseArgv(argv,tLISTV,&lv,0);
    ...
    ...
  }
  else {
    ...
    ...
  }
}
```

4.2.6 C-macros dealing with `VALUE`s

- `DeleteValue(VALUE ref)` : macro to delete a reference to a `VALUE`
- `NewValue(TYPESTRUCT ts)` : macro to create a new `VALUE` corresponding to the type `ts` (to learn about `TYPESTRUCT` you should read Section 5.2)
- `ToStrValue(VALUE value, char flagShort)` : macro that returns a string ("short" or "long" depending on the flag `flagShort`) that represents the value `value`. The

"long" representation is used by LastWave to display the returned value of a command and the "short" representation is used by LastWave when the `value` is within a listv which is displayed.

- `PrintValue(VALUE value)` : macro that prints a `value` the same way the `print` command does
- `PrintInfoValue(VALUE value)` : macro that prints information on a `value` the same way the `info` command does
- `TempValue(VALUE ref)` : macro that makes a reference to a `VALUE` temporary
- `AddRefValue(VALUE value)` : macro that increments the reference counter of a `value` by one
- `RemoveRefValue(VALUE value)` : macro that decrements the reference counter of a `value` by one
- `SetResultValue(VALUE value)` : macro that sets the returned `value` of a command
- `GetTypeValue(VALUE value)` : macro that returns the static string taht corresponds to the type of the `value`
- `ValueOf(VALUE value)` : macro that returns a valid `VALUE` for casting purposes (see previous section).

4.3 Managing `&listv`

As we have already explained (see Section 4.1.3), `&listv` are represented using the `LISTV` type which is a pointer to the structure `Listv`. They can store a list of `VALUE`s which can be of any type. Actually, for efficiency reasons, floats are not stored using `VALUE`s in a listv, they are stored directly as floats. Thus, as you will notice, in the C-functions that deal with the elements of a listv, floats always appear as particular cases.
The static string type of a listv (i.e., the string returned by the `GetTypeValue` function) is `listvType`.
The corresponding *tTYPE*'s are `tLISTV` and `tLISTV_`. For parsing argument you can use the parsing functions

- `void ParseListv(char *arg, LISTV *plv)`
- `char ParseListv_(char *arg, LISTV default, LISTV *plv)`
- `void ParseListvLevel(LEVEL level, char *arg, LISTV *plv)`
- `char ParseListvLevel_(LEVEL level, char *arg, LISTV default, LISTV *plv)`

You should not manipulate directly the structure of a listv. You should use the functions describe below:

- `LISTV NewListv(void)` : allocating an empty listv

- `LISTV TNewListv(void)` : creating a temporary listv (an already allocated `listv` can be made temporary using `TempValue(VALUE val)`)
- `void DeleteListv(LISTV lv)` : deleting a listv (instead, you could call the macro `DeleteValue(VALUE val)`)
- `void ClearListv(LISTV lv)` : empties the listv
- `LISTV CopyListv(LISTV lvIn, LISTV lvOut)` : if `lvOut` is `NULL` it returns a copy of the listv `lvIn` otherwise it copies `lvIn` into `lvOut` (in any case, the elements of the listv are not copied)
- `void SetLengthListv(LISTV lv, int length)`: sets the length of a listv (performs allocation)
- `int GetLengthListv(LISTV lv)`: returns the length of a listv
- `char *GetListvNth(LISTV lv, int n, VALUE *v, LWFLOAT f)` : gets the `n`th element of a listv. If it is a float then `v` is `NULL` and `f` is the corresponding float otherwise `v` contains the corresponding `VALUE`. In any case it returns the static string that corresponds to the type of the cooresponding element (e.g., `numType`, `signalType`, `listvType`,...)
- `LWFLOAT GetListvNthFloat(LISTV lv, int n)` : tries to read the `n`th element of a listv as a float. If it is not a float, it generates an error
- `char *GetListvNthStr(LISTV lv, int n)` : tries to read the `n`th element of a listv as a float. If it not a string, it generates an error
- `void SetListvNthValue(LISTV lv, int n, VALUE *v)` : sets the `n`th element of a listv with the `VALUE v`. No allocation is made, i.e., `n` must be strictly smaller than the `length` of the listv
- `void SetListvNthFloat(LISTV lv, int n, LWFLOAT f)` : sets the `n`th element of a listv with the float `f`. No allocation is made, i.e., `n` must be strictly smaller than the `length` of the listv
- `void AppendValue2Listv (LISTV lv, VALUE val)` : appends a value at the end of a listv (allocation might be performed)
- `void AppendFloat2Listv(LISTV lv, LWFLOAT f)` : appends a float at the end of a listv (allocation might be performed)
- `void AppendInt2Listv(LISTV lv, int i)` : appends an integer at the end of a listv (allocation might be performed)
  item `void AppendStr2Listv(LISTV lv, char *)` : appends a string at the end of a listv (allocation might be performed)
  item `void AppendListv2Listv(LISTV lv, LISTV lv1)` : appends a listv at the end of a listv (allocation might be performed)
- `void ConcatListv(LISTV lv1,LISTV lv2,LISTV lvOut)` : concatenates two listv's

- `void MultListv(LISTV lv1,int n,LISTV lvOut)` : fills `lvOut` with the repetition pf `lv1` `n` times.

4.4 Managing &proc

Commands correspond to the LastWave type `&proc`. In the C-language, they are represented using the type `PROC` which corresponds to a pointer to the C-structure `Proc`. This structure is used to store both commands defined in the C-language and commands defined using LastWave command language, i.e., a script command (which could eventually be anonymous). The flag `flagSP` of the structure is set to 1 if the command corresponds to script command otherwise it is 0.
The static string type of a `PROC` (i.e., the string returned by the `GetTypeValue` function) is `procType`.
The corresponding $tTYPE$'s are `tPROC` and `tPROC_`. For parsing argument you can use the parsing functions

- `void ParseProc(char *arg, PROC *pProc)`
- `char ParseProc_(char *arg, PROC default, PROC *pProc)`
- `void ParseProcLevel(LEVEL level, char *arg, PROC *pProc)`
- `char ParseProcLevel_(LEVEL level, char *arg, PROC default, PROC *pProc)`

You should not manipulate directly the structure of a `PROC`. The only functions you might need to use are

- `void ApplyProc2Listv(PROC proc, LISTV lv)`: This function allows to apply a command `proc` using as arguments the elements of the listv `lv`. The result is stored as the returned value of the current command. You can use the `GetResult...` functions (see Section 4.4.1) to get the result. To see an example of how to use this function, you should look into the file `kernel/src/int_listv.c` for the function `SortListv(...)`. This function is able to sort a listv using any `PROC` as a sorting function.
- `void ApplyProc2List(PROC proc, char **argv)`: This is the same function as `ApplyProc2Listv` except that the list of argument is a list of strings (of type `tLIST`, see Section 3.4) that will be evaluated in the current level before being passed to the command.

4.4.1 The `GetResult...` functions

To get the result of a command that was just executed, you can use the following functions

- `char *GetResultType(void)`: returns a string which corresponds to the static string type (e.g., `listvType`, `signalType`, `numType`, ...).
- `VALUE *GetResultValue(void)`: returns the result as a `VALUE`
- `char * GetResultStr(void)`: returns the result as a string if the result is of type `strType`

- `int GetResultInt(void)`: returns the result as an integer if it is of type `numType` and if it is an integer
- `LWFLOAT GetResultFloat(void)`: returns the result as a float if it is of type `numType`

4.5 Managing `&script`

Scripts correspond to the LastWave type `&script`. In the C-language, they are represented using the type `SCRIPT` which corresponds to a pointer to the C-structure `Script`. To get the script of a `PROC` (**which must of course correspond to a script command** and not a C-command, i.e., whose field `flagSP` is set to 1), you must use the following field

```
PROC proc;
SCRIPT script = proc->sp->script;
```

The static string type of a `SCRIPT` (i.e., the string returned by the `GetTypeValue` or the `GetResultType` function) is `scriptType`.
The corresponding $tTYPE$'s are `tSCRIPT` and `tSCRIPT_`. For parsing argument you can use the parsing functions

- `void ParseScript(char *arg, SCRIPT *pScript)`
- `char ParseScript_(char *arg, SCRIPT default, SCRIPT * pScript)`
- `void ParseScriptLevel(LEVEL level, char *arg, SCRIPT *pScript)`
- `char ParseScriptLevel_(LEVEL level, char *arg, SCRIPT default, SCRIPT * pScript)`

You should not manipulate directly the structure of a `SCRIPT`. The only function you might need to use are

- `void EvalScriptLevel(LEVEL level, SCRIPT script,char flagStoreResult)`; this function lets you evaluate the script `script` in the level `level` (let us recall that the current level is the global variable `levelCur`). If the flag `flagStoreResult` is set to 1 then the result of the last command of the script is saved by LastWave as if it was the result of a command that was just executed. In that case, you can get the result using the `GetResult...` functions (see Section 4.4.1). If you do not need to save the result, you should `flagStoreResult` to 0.

4.6 Managing `&signal`

4.6.1 The `SIGNAL` type and the corresponding $tTYPE$'s

Signals correspond to the LastWave types `&signal` or `&signali` (for non empty signals). In the C-language, they are represented using the type `SIGNAL` which corresponds to a pointer to the C-structure `Signal`. The static string types of a `SIGNAL` (i.e., the string returned by the `GetTypeValue` or the `GetResultType` function) is `signalType` or `signaliType`.

The corresponding $tTYPE$'s are tSIGNAL and tSIGNAL_ or tSIGNALI and tSIGNALI_. The last two $tTYPE$'s will parse only signals that are not empty whereas the first two will parse any signal (empty or non empty). For parsing argument you can use the parsing functions

- void ParseSignal(char *arg, SIGNAL *pSignal)
- char ParseSignal_(char *arg, SIGNAL default, SIGNAL * pSignal)
- void ParseSignalI(char *arg, SIGNAL *pSignal)
- char ParseSignalI_(char *arg, SIGNAL default, SIGNAL * pSignal)
- void ParseSignalLevel(LEVEL level, char *arg, SIGNAL *pSignal)
- char ParseSignalLevel_(LEVEL level, char *arg, SIGNAL default, SIGNAL * pSignal)
- void ParseSignalILevel(LEVEL level, char *arg, SIGNAL *pSignal)
- char ParseSignalILevel_(LEVEL level, char *arg, SIGNAL default, SIGNAL * pSignal)

The SIGNAL type is defined in the file package/_signals/include/signals.h. It corresponds to a pointer to the structure struct signal (or Signal) which defines all the fields of a signal. There should not be any use for you to handle directly a variable of type struct signal, you should only use variables of type SIGNAL. The only fields of the signal structure you should know about are

```
int size;           /* the size of the signal (i.e., number of samples) */
int type;           /* the type of the signal.  If it is YSIG then it means that just t
LWFLOAT *Y;         /* the ordinate array */
LWFLOAT *X;         /* the abscissa array (for XYSIG only) */
LWFLOAT x0;         /* first abscissa value (for YSIG only) */
LWFLOAT dx;         /* distance between two successive samples (for YSIG only) */

char *name;         /* The name of the signal.  Should be set using the SetNameSignal(S

int firstp;         /* The index of the first sample not affected by border effects */
int lastp;          /* The index of the last sample not affected by border effects */
```

In order to allocate the arrays X and Y of a signal you should use the SizeSignal function. Its syntax is

```
void SizeSignal(SIGNAL signal, int size, int type)
```

It performs all the desallocations and allocations in signal so that it will be allow to store a signal of type type (which must be one of XYSIG or YSIG) and of size size. Let us note that it initializes the fields x0, dx, firstp and lastp.
Section 4.1.1 gives you an example of how to handle SIGNAL's.
If you want to write a command which syntax is

```
mysin <signal> <size>
```

and who sets the signal <signal> so that it corresponds to onne arc of a sinusoid on <size> points uniformly sampled between 0 and 1, you would write

```
C_MySin(char **argv)
{
  char **argv;
  int size;

  argv = ParseArgv(argv,tSIGNAL,&signal,tINT,&size,0);

  SizeSignal(signal,size,YSIG);

  for (i=0;i<size;i++) signal->Y[i] = sin((2*PI*i)/size);

  signal->x0 = 0;
  signal->dx = 1.0/size;
}
```

4.6.2 Basic functions for dealing with SIGNAL's

- `SIGNAL NewSignal(void)`: allocates a `SIGNAL`
- `SIGNAL TNewSignal(void)`: allocates a temporary `SIGNAL`
- `void DeleteSignal(SIGNAL ref)`: deletes a `SIGNAL` reference (same as `DeletValue(ref)`)
- `void ClearSignal(SIGNAL signal)`: inits all the fields of a `SIGNAL` (including desallocation of the `X` and `Y` arrays)
- `void SizeSignal(SIGNAL signal, int size, int type)`: ensures that both the `X` and `Y` arrays have sufficient allocation to store a signal of size `size` and of type `type` (which must be either `XYSIG` or `YSIG`). Sets the `size` field to `size`, `firstp` to 0, `lastp` to `size-1`.
- `void CopyFieldsSig(SIGNAL in,SIGNAL out)`: copies the fields (except `X` and `Y`) of the signal `in` to the fields of the signal `out`
- `void CopySig(SIGNAL in, SIGNAL out)`: copies (all the fields including the values in `X` and `Y`) the signal `in` to the signal `out`
- `LWFLOAT XSig(SIGNAL sig,int index)`: returns the `x` value corresponding index `index` (handles both `XYSIG` or `YSIG`)
- `int ISig(SIGNAL signal,LWFLOAT xValue)`: returns the closest `index` corresponding to the `x` alue `xValue` (handles both `XYSIG` or `YSIG`)

- `void ZeroSig(SIGNAL sig)`: sets the `y` value of a signal to 0.

Signals of type `XYSIG` must always have an `X` array which sorted in an increasing order. If you fill up the `X` array with non sorted values, you can use the `SortSig()` function to sort it (see next Section)

4.6.3 Mathematical functions on `SIGNAL`'s

Each time a function has a parameter named `flagCausal`, it means that if this flag is set, only the indices between `firstp` (included) and `lastp` (included) are taken into account. Each time a function has a parameter named `borderType`, it specifies the border effects and the value of `borderType` must be one of `BorderPad` (constant border effect), `BorderPad0` (zero border effect), `BorderPeriodic` (periodic border effect), `BorderMirror` (mirror border effect).

Basic functions

- `void ExtractSig(SIGNAL sig,SIGNAL sigOut, int borderType, int firstPoint,int newSize)`: extracts the signal `sig` from the first index `firstPoint` (which can be out of range) on a number of points of `newSize`.

- `void MinMaxSig(SIGNAL signal,LWFLOAT *pxMin,LWFLOAT *pxMax,LWFLOAT *pyMin,LWFLOAT *pyMax, int *piyMin,int *piyMax,int flagCausal)`: computes the the `min` and `max` `y` and `x` values of the `signal`. When it returns, `*pxMin` (resp. `*pxMax`) corresponds to the minimum (resp. maximum) `x` value. `*pyMin` (resp. `*pyMax`) corresponds to the minimum (resp. maximum) `y` value and `*piyMin` (resp. `piyMax`) to the corresponding indices.

- `void PaddSig(SIGNAL sig,SIGNAL sigOut,int borderType,int newSize)`: padds a signal `sig` (the resulting signal is set into `sigOut`) so that its new size is `newSize`.

- `void ThreshSig(SIGNAL in, SIGNAL out, int flagX, int flagY,int flagMin, LWFLOAT min, int flagMax,LWFLOAT max)`: thresholds the signal `in` and puts the result into the signal `out` between values `min` and `max`. If `flagX` is set, the threshold is on `x` values. If `flagY` is set, the threshold is on `y` values. If `flagMin` (resp. `flagMax`) is not set then `min` (resp. `max`) is not taken into account.

- `void SortSig(SIGNAL signal)`: is `signal` is of tpe `XYSIG`, it sorts the `x` values (and move the corresponding `y` values). If it is of type `YSIG`, it sorts the `y` values. In any case the sorting is made in increasing order.

Advanced functions

- `LWFLOAT GetCorrelation(SIGNAL signal1,SIGNAL signal2,int flagCausal)`: returns the correlation between `signal1` and `signal2`.

- `LWFLOAT GetLpNormSig(SIGNAL signal, LWFLOAT p,int flagCausal)`: returns the `Lp` norm of `signal`

- `LWFLOAT GetNthMoment(SIGNAL signal, int n, LWFLOAT *pNthMoment,int flagCausal, int flagCentered)`: returns in `*pNthMoment` the `n`th moment of the `y` values of the `signal`. If `flagCentered` is set the centered `n`th moment is computed. In any case the mean of the `signal` is returned.

- `LWFLOAT GetAbsMoment(SIGNAL signal, LWFLOAT f1, LWFLOAT *pMoment,int flagCausal, int flagCentered)`: same as `GetNthMoment()` except the moment is computed using the absolute value of the `y` values

- `void HistoSig(SIGNAL input, SIGNAL output, int n,LWFLOAT xmin, LWFLOAT xmax,LWFLOAT ymin, LWFLOAT ymax, SIGNAL weight, int flagCausal)`: computes in `output` an histogram with `n` branches of the `y` values of the signal `input`. If `xMin<xMax`, only the `y` values corresponding to abscissa between `xMin` and `xMax` are taken into account. If `yMin<yMax`, only the `y` values between `yMin` and `yMax` are taken into account. If `weight` is not `NULL`, it must be of the same size as the `input` signal and it corresponds to weights on each `y` value.

- `void LineFitSig(SIGNAL signal,LWFLOAT *pA,LWFLOAT *pSigA,LWFLOAT *pB,LWFLOAT *pSigB,int iMin,int iMax)`: performs a linear fit of signal `signal` (using `L2` minimization) restricted to the index range `[iMin,iMax]`. The linear fit is `y=ax+b`, `a` is stored in `*pA` and `b` in `*pB`. The satndard deviation of `a` (resp. `b`) is stored in `*pSigA` (resp. `*pSigB`).

- `void FFTConvolution (SIGNAL signal, SIGNAL filter, SIGNAL out, int borderType, LWFLOAT xMin, LWFLOAT xMax)`: computes the convolution of the signal `in` (which is padded using `borderType`) with the (compact support) `filter` and puts the result in the signal `out`. The convolution is computed in between the abscissa `xMin` and `xMax`. The fast convolution algorithm with FFT is used. This function deals also wit the border type `BorderMir1`.

- `void DirectConvolution (SIGNAL signal,SIGNAL filter, SIGNAL out, int borderType, LWFLOAT xMin, LWFLOAT xMax)`: computes the convolution of the signal `in` (which is padded using `borderType`) with the (compact support) `filter` and puts the result in the signal `out`. The convolution is computed in between the abscissa `xMin` and `xMax`. The direct convolution algorithm is used. This function deals also wit the border type `BorderMir1`.

- `void Fft(SIGNAL inReal,SIGNAL inImag,SIGNAL outReal,SIGNAL outImag,int fftSign,char flagShift)`: if `fftSign` is 1, it computes the Fourier transform of the complex signal whose real part is `inReal` and the imaginary part is `inImag`. The input signal must have a size which is a power of 2. The (complex) result is put in the signals `outReal` and `outImag`. If the input signal is real then you should set `inImag` to `NULL`. In this case, if its size is `N` then the output signals have the size `N/2+1` corresponding to the frequencies `[0,`π`]`. In the case the input signal is complex, the output signals have the same size as the input signals and correspond to the frequencies `[0,`2π`[` unless `flagShift` is set to 1 in which case the frequencies are `[-`π, π`[`. If `fftSign` is set to -1, the exact inverse action is performed (in that case if the input signals have a size of `N/2+1` the ouput signal will be real).

4.6.4 i/o functions for SIGNAL's

For output

- `void WriteSigStream(SIGNAL signal,STREAM stream, char flagBinary, char *mode,int flagHeader)`: writes the `signal` in the `stream`. If `flagBinary` is 1, it is written using binary coding, if it is 0 it is written using ascii coding. If `flagHeader` is 0 then no header (i.e, field values) is written. In both ascii and binary coding `mode` must be one of `"xy"` (in ascii coding, two columns are written, one for `x` values and one for `y` values, in binary coding `x` values are followed by `y` values), `"yx"` (same as `"xy"` except that `x` and `y` are exchanged), `"y"` (only `y` values are written), `"x"` (only `x` values are written) or `""` (LastWave uses the more "efficient" mode, e.g., for `YSIG` signals the `x` values are not written extensively).

- `void WriteSigFile(SIGNAL signal,char *filename, char flagBinary, char *mode,int flagHeader)`: same as `WriteSigStream()` except that it writes into the file named `filename`

- `void WriteSigRawStream(SIGNAL signal,STREAM stream, char binaryCoding)`: writes the `y` values of the `signal` in the `stream` using a `raw` format, i.e., binary coding is used and there is no header. `binaryCoding` is one of `BinaryBigEndian` or `BinaryLittleEndian` depending on the coding you want to use (you can test the variable `IsCPULittleEndian` to know the coding type of the computer you are running LastWave on).

- `void WriteSigRawFile(SIGNAL signal,char *filename, char binaryCoding)`: same as `WriteSigRawStream` except that it writes into the file named `filename`

For input

- `char ReadInfoSigStream(STREAM stream, SIGNAL siginfo, char *header, char *binaryMode, char *binaryCoding, int *nColumns)` reads information about a signal storedin a `stream` without reading the signal values themselves. When this function returns, the fields of `siginfo` contains the corresponding fields of the signal in the `stream` (except for the `Y` and `X` arrays), `*header` is `YES` if there is a header and `NO` if not, `*binaryMode` is `YES` if the signal is binary coded and `NO` if ascii coded, `*binaryCoding` is one of `BinaryBigEndian` or `BinaryLittleEndian` and `*nColumns` is the number of columns (only if `*binaryMode` is `NO`). The function returns `YES` if the format is valid LastWave format and `NO` otherwise. `In any case, the position of the stream is not change`.

- `char ReadInfoSigFile(char *filename, SIGNAL siginfo, char *header, char *binaryMode, char *binaryCoding, int *nColumns)`: same as `ReadInfoSigStream` except that it reads from the file named `filename`

- `void ReadSigStream(SIGNAL signal,STREAM stream, int firstIndex,int sizeToRead, int xcol,int ycol)`: function to read a `signal` from a `stream`. It knows how to read any `stream` that was generated by the `WriteSigStream()` function. Moreover, it can also be used for multicolumn (any number) ascii coding too. `firstIndex` is the first index to be read in the `stream` (starting from 0), `sizeToRead` is the total number

of values to be read from this first index. In the case the coding is an ascii coding, `xcol` (resp. `ycol`) is the number of the column (starting from 1) corresponding to the `x` (resp. `y`) values. If `xcol` or `ycol` is 0, LastWave tries to make some inductions on what to do. If `xcol` is -1, then no `x` values are read.

- `void ReadSigFile(SIGNAL signal,char *filename, int firstIndex,int sizeToRead, int xcol,int ycol)`: same as `ReadSigStream` except that it reads from the file named `filename`
- `void ReadSigRawStream(SIGNAL signal,STREAM stream, int firstIndex,int sizeToRead, char binaryCoding)`: function to read from a `stream` a `signal` which is coded in a `raw` format, i.e., no header and binary coding. `firstIndex` is the first index to be read in the `stream` (starting from 0), `sizeToRead` is the total number of values to be read from this first index. `*binaryCoding` is one of `BinaryBigEndian` or `BinaryLittleEndian`.
- `void ReadSigRawFile(SIGNAL signal,char *filename, int firstIndex,int sizeToRead, char flagMode)`: same as `ReadSigStream` except that it reads from the file named `filename`.

4.7 Managing &range

Let us recall that whenever a signal is expected, a range will be automatically converted to a signal. Thus it is very rare that you will want to manipulate explicitely ranges. Generally, you want to use ranges for efficiency since a signal uses more memory (all the values are stored) than a range. This is, for instance, the case of the `foreach` command.

Signals correspond to the LastWave types `&range`. In the C-language, they are represented using the type `RANGE` which corresponds to a pointer to the C-structure `Range`. The static string types of a `RANGE` (i.e., the string returned by the `GetTypeValue` or the `GetResultType` function) is `rangeType`.

The corresponding $tTYPE$'s are `tRANGE` and `tRANGE_`. For parsing argument you can use the parsing functions

- `void ParseRange(char *arg, RANGE *pRange)`
- `char ParseRange_(char *arg, RANGE default, RANGE * pRange)`
- `void ParseRangeLevel(LEVEL level, char *arg, RANGE * pRange)`
- `char ParseRangeLevel_(LEVEL level, char *arg, RANGE default, RANGE * pRange)`

The `RANGE` type is defined in the file `package/_signals/include/value.h`. It corresponds to a pointer to the structure `struct range` (or `Range`) which defines all the fields of a range. The only fields of the range structure you should know about are

```
LWFLOAT first;          /* the first value of the range*/
LWFLOAT step;           /* the step of the range */
int size;               /* the number of points of the range */
```

You should know about the macros

- `RangeVal(RANGE range, int n)`: returns the `n`th value of the `range`
- `RangeLast(RANGE range)`: returns the last value of the `range`
- `RangeFirst(RANGE range)`: returns the first value of the `range`
- `RangeMin(RANGE range)`: returns the minimum value of the `range`
- `RangeMax(RANGE range)`: returns the maximum value of the `range`

You should know also about the allocation functions

- `RANGE NewRange(void)`: to create a range
- `RANGE TNewRange(void)`: to create a temporary range (you could as well use `TempValue(range)` to make a range temporary)
- `void DeleteRange(RANGE ref)`: to delete a reference to a range (you could as well use `DeleteValue(range)`.

4.8 Managing &image

4.8.1 The IMAGE type and the corresponding $tTYPE$'s

Images correspond to the LastWave types `&image` or `&imagei` (for non empty images). In the C-language, they are represented using the type `IMAGE` which corresponds to a pointer to the C-structure `Image`. The static string types of an `IMAGE` (i.e., the string returned by the `GetTypeValue` or the `GetResultType` function) is `imageType` or `imageiType`.
The corresponding $tTYPE$'s are `tIMAGE` and `tIMAGE_` or `tIMAGEI` and `tIMAGEI_`. The last two $tTYPE$'s will parse only images that are not empty whereas the first two will parse any image (empty or non empty). For parsing argument you can use the parsing functions

- `void ParseImage(char *arg, IMAGE *pImage)`
- `char ParseImage_(char *arg, IMAGE default, IMAGE * pImage)`
- `void ParseImageI(char *arg, IMAGE * pImage)`
- `char ParseImageII_(char *arg, IMAGE default, IMAGE * pImage)`
- `void ParseImageLevel(LEVEL level, char *arg, IMAGE * pImage)`
- `char ParseImageILevel_(LEVEL level, char *arg, IMAGE default, IMAGE * pImage)`
- `void ParseImageILevel(LEVEL level, char *arg, IMAGE * pImage)`
- `char ParseImageILevel_(LEVEL level, char *arg, IMAGE default, IMAGE * pImage)`

The `IMAGE` type is defined in the file `package/_signals/include/images.h`. It corresponds to a pointer to the structure `struct image` (or `Image`) which defines all the fields of a signal. There should not be any use for you to handle directly a variable of type `struct image`, you should only use variables of type `IMAGE`. The only fields of the image structure you should know about are

```
int nrow,ncol;            /* the number of rows and columns of the image */
LWFLOAT *pixels;          /* the 1d array with the pixel values.  The pixel value corres
char *name;               /* The name of the image.  Should be set using the SetNameImag
```

In order to allocate the `pixels` array you must use the `SizeImage` function. Its syntax is

```
void SizeImage(IMAGE image, int ncols, int nrows)
```

It ensures that the `pixels` array has sufficient allocation to store an image whose number of columns is `ncols` and number of rows `nrows`.

4.8.2 Functions for dealing with `IMAGE`'s

Basic functions are

- `IMAGE NewImage(void)`: allocates an `IMAGE`
- `IMAGE TNewImage(void)`: allocates a temporary `IMAGE`
- `void DeleteImage(IMAGE ref)`: deletes an `IMAGE` reference (same as `DeletValue(ref)`)
- `void ClearImage(IMAGE image)`: inits all the fields of an `IMAGE` (including desallocation of the `pixels`)
- `void SizeImage(IMAGE image, int ncols, int nrows)`: ensures that the `pixels` array has sufficient allocation to store an image whose number of columns is `ncols` and number of rows `nrows`.
- `void CopyImage(IMAGE in, IMAGE out)`: copies (all the fields including the values in `pixels`) the image `in` to the image `out`

More advanced functions are

- `LWFLOAT GetLpNormImage(IMAGE im, LWFLOAT p)`: returns the `Lp` norm of `image`
- `LWFLOAT GetNthMomentImage(IMAGE image, int n, LWFLOAT *pNthMoment, int flagCentered)`: returns in `*pNthMoment` the `n`th moment of the `pixels` values of the `image`. If `flagCentered` is set the centered `n`th moment is computed. In any case the mean of the `image` is returned.
- `double ImageScalarProduct(IMAGE image1,IMAGE image2)`: returns the scalar product of two images.

4.8.3 i/o functions for IMAGE's

For output

- `void WriteImageStream(IMAGE image,STREAM s,char flagChar,LWFLOAT min, LWFLOAT max, char flagHeader)`: writes the image `image` into the stream `s`. If `flagHeader` is `YES` then a header is written. If `flagChar` is `NO` the pixel values are written using `LWFLOAT` (binary) coding. If it is `YES`, each pixel is coded using a `char`, i.e., an integer ranging in `[0,255]`. In that case if `min<max` then rescaling will be performed so that the pixel value `min` is matched to 0 and the pixel value `max` is matched to 255. If `max>min`, pixel values below 0 are set to 0 and pixel values above 255 are set to 255.

For input

- `void ReadImageStream(IMAGE image ,STREAM s, char flagHeader,int nrow, int ncol,char flagChar)`: reads an `image` from the stream `s`. If `flagHeader` is `YES` then it means that the file has a header, otherwise `nrow` and `ncol` must be specified. `flagChar` indicates wether the image is coded using binary `LWFLOAT` coding or `char` coding.

4.9 About NUMVALUE

As we have already explained, numbers are represented internally as `VALUE`'s of type `NUMVALUE`. The associated `&type` is `&num` (corresponding to the C static string `numType`). There is a priori no reason for you to handle the corresponding structure since LastWave's API lets you address number values as simple `LWFLOAT`.
The `NUMVALUE` type is a pointer to a structure which basically has (apart from the fields that are shared by all `VALUE`'s) a single field named `f` to store the corresponding value.

4.10 About STRVALUE

As we have already explained, strings are represented internally as `VALUE`'s of type `STRVALUE`. The associated `&type` is `&str` (corresponding to the C static string `strType`). There is a priori no reason for you to handle the corresponding structure since LastWave's API lets you address string values as simple `char *`.
The `STRVALUE` type is a pointer to a structure which basically has (apart from the fields that are shared by all `VALUE`'s) a single field named `str` to store the corresponding value.

Warning: the field `str` must be allocated dynamically and must have an allocation size greater than the constant `MinStringSize`

To set the string of a `STRVALUE` you should use the function

```
void SetStrValue(STRVALUE sc, char *str)
```

You can get the string using

```
char *GetStrFromStrValue(STRVALUE sc)
```

and you can get the corresponding `&list` using

```
char **GetListFromStrValue(STRVALUE sc)
```

let us not that this latter function can generate an error if the string does not have a valid list representation.

4.11 The `null` VALUE

The value `null` in LastWave corresponds to the C `VALUE nullValue`. The `&type` of this value is `&null` and corresponds to the static string `nullType`. There is a single instance of this value.

Chapter 5

Defining new *tTYPE*'s, new &*type*'s and new VALUE's

In this chapter we are going to explain how you can create a new type of VALUE with its corresponding &*type* and tTYPE. For the sake of clarity, we will explain how to do it on a simple example. To adapt this example to your own needs should be easy.

The CIRCLES example
Let us build a new type which corresponds to a list of circles. Each circle will be defined using its position in the plane (abscissa and ordinate) and its radius. Moreover a positive number (a weight) will be associated to each circle. All the codes that you will find below can be found in the file user/src/circles.c. We are going to create a package named circle that will contain all the corresponding definitions. In order this package to be available at startup, you must declare it in the UserInit() function in the file user/src/user.c (the corresponding lines are commented). Let us start!

5.1 The main structures - the &*type*

First of all you must define the structure that will correspond to the VALUE. In our example it must be a list of circles. We first need to define what a circle is:

```
/* A single circle */
typedef struct circle {
  LWFLOAT x;
  LWFLOAT y;
  LWFLOAT r;
} Circle;
```

We choose to define the list of circles as a simple (dynamically allocated) array. Since the corresponding type must be a VALUE, we must include the VALUE fields, i.e.,

```
/* The CIRCLES VALUE */
typedef struct circles {

  ValueFields;      /* The fields of the VALUE structure */
```

```
  Circle *array;     /* The array of circles */
   int n;                    /* The size of this array */

  char *name;       /* The name of the list */

} Circles, *CIRCLES;
```

We chose to associate a name to each of this `VALUE` (as for `SIGNAL`'s). This name will be stored in the field `name`.
The so-defined `CIRCLES` structure corresponds to a `VALUE` (cast is thus possible). We must define the corresponding *&type* and the associated static string

```
/* The corresponding &type */
static char *circlesType = "&circles";
```

5.2 The TypeStruct definition

To each `VALUE` type corresponds a `TypeStruct`. It stores all the information about this `VALUE`, e.g., the methods `new`, `delete`, `copy`, ..., the corresponding *&type*, the fields, ... For our type `CIRCLES`, the corresponding `TypeStruct` looks like

```
TypeStruct tsCircles = {

  "{{{&circles} {A description of the type '&circles'.}}}",  /* Documentation */

  &circlesType,       /* The basic (unique) &type name */
  NULL,  /* The GetType function if several &types are associated (e.g., &signal, &signali)

  DeleteCircles,      /* The Delete function */
  NewCircles,          /* The New function */

  CopyCircles,         /* The copy function */
  ClearCircles,         /* The clear function */

  ToStrCircles,  /* String conversion */
  PrintCircles,   /* The Print function : print the object when 'print' is called */
  PrintInfoCircles,   /* The PrintInfo function : called by 'info' */

  NULL,       /* The NumExtract function : used to deal with syntax like 10a */

  fieldsCircles,        /* The list of fields */
};
```

Before describing precisely the role of each of these fields let us see how the final declaration of the new `VALUE` looks like

5.3 Defining a package with a new VALUE and *tTYPE*s

As already explained (see Section 3.13), a package declaration is made in the following

```
void DeclareCirclesPackage(void)
{
  DeclarePackage("circles",LoadCirclesPackage,2003,"1.0","E.Bacry",
  "Demo package");
}
```

where the function DeclareCirclesPackage must be called in the UserInit function at startup. this function is in the file user/src/user.c, it should look like

```
void UserInit(void)
{
  extern void DeclareSignalPackage(void);
   ...
  extern void DeclareCirclesPackage(void);

  DeclareSignalPackage();
  ...
  DeclareCirclesPackage();
}
```

When the package is loaded the function LoadCirclesPackage will be called. Thus this function must define the new VALUE type CIRCLES and associate it to the TypeStruct tsCircles and the *&type* circlesType. Moreover it must define the new tTYPES tCIRCLES and tCIRCLES_. This is simply done in the following way

```
int tCIRCLES, tCIRCLES_;

static void LoadCirclesPackage(void)
{
  tCIRCLES = AddVariableTypeValue(circlesType, &tsCircles, NULL);
  tCIRCLES_ = tCIRCLES+1;
}
```

(Let us note that no command will be defined in this package).

5.3.1 The AddVariableType... functions

For defining new VALUE's you should always use the AddVariableTypeValue function as shown above. Its full syntax is

```
int AddVariableTypeValue(char *type, TypeStruct *ts, char (*parse) (LEVEL level, char *,
```

where type is the *&type* static string, ts is the corresponding TypeStruct and the last (optional) argument) is a parsing function that is called whenever the corresponding tTYPE is parsed. By default it evaluates the argument and expect a value of type type. However,

if you want to be able to use some special syntax you can always bypass the default parsing function.

If you want to define a new *&type* and new `tTYPES` associated to simple types such as `LWFLOAT` you must use one of the functions

- `int AddVariableTypeInt(char *type, char (*parse) (LEVEL level, char *, int, int *))`
- `int AddVariableTypeFloat(char *type, char (*parse) (LEVEL level, char *, LWFLOAT, LWFLOAT *))`
- `int AddVariableTypeStr(char *type, char (*parse) (LEVEL level, char *, char *, char **))`

In any case you must always specify the parsing function (it is no longer an optional argument). Thus for instance, if we want to define a new *&type* and the corresponding *tTYPE*s for dealing with positive integers we would write

```
char ParsePosInt(LEVEL level, char *arg, int default, int *p)
{
  if (ParseInt_(level,arg,default,p)==NO) return(NO);

  if (*p < 0) {
    SetErrorf("Expect a positive integer");
    return(NO);
  }

  return(YES);
}
```

and at startup we would write

```
  intpType = "&intp";
  tINTP = AddVariableTypeInt(intpType, ParsePosInt);
  tINTP_ = tINTP +1;
```

Let us note that this type will not correspond to a value type, i.e., it will treated and stored as a regular number. However this definition allows you to use this new type for type definition of C or script procedure arguments.

5.4 The *&type* and the documentation in the `TypeStruct` definition

In Section 5.2 we have define the `TypeStruct` associated to `CIRCLES`. It started with the lines

```
TypeStruct tsCircles = {
```

```
  "{{{&circles} {A description of the type '&circles'.}}}",  /* Documentation */

  &circlesType,       /* The basic (unique) &type name */
  NULL,  /* The GetType function if several &types are associated (e.g., &signal, &signali)
   ...
```

We shall start to explain the last line If `NULL` is put then itmeans that this `VALUE` corresponds to a single basic *&type* that is specified just above (in our case `&circlesType`). If it is not `NULL` it mus be a function that takes a single argument of type `CIRCLES` and that returns a static string which must corresponds to its *&type*. This is, for instance, how the types `&signal` and `&signali` are implemented for signals. Both *&types* correspond to `SIGNAL` but the type depends whether the signal is empty or not. In any case you must always specify a basic *&type* on the second line. For `SIGNAL`s the basic *&type* is `&signal`.
The first line is a string that gives description of each of the *&types*. If the third line is `NULL` there is a single *&type*. The general syntax for this help is

```
"{{{&type1} {description of &type1}} ... {{&typeN} {description of &typeN}}}"
```

5.5 The main functions of the TypeStruct

5.5.1 The New function

This function is used for allocation. In our case it is

```
CIRCLES NewCircles(void)
{
  CIRCLES c;
  extern TypeStruct tsCircles;

  c = Malloc(sizeof(Circles));

  InitValue(c,&tsCircles);

  c->array = NULL;
  c->n = 0;
  c->name = CopyStr("");

  return(c);
}
```

It must return a pointer to a dynamically allocated structure of the corresponding `VALUE` type. Moreover this structure must be initialized using the `InitValue(VALUE,TypeStruct *)` function. Of course, this function must also performs your own initialization of the structure. In our case in includes allocating a string for the `name` field. When it is created the name is set to the empty string.

5.5.2 The Delete function

This function is called for delting a reference to th `VALUE`. In our case, it is

```
void DeleteCircles(CIRCLES c)
{
  RemoveRefValue(c);
  if (c->nRef > 0) return;

  if (c->n != 0) Free(c->array);
  if (c->name) Free(c->name);
  Free(c);
}
```

This function must start by decreasing the reference counter by 1 and testing if it reached 0 or not. If it did not then it just returns. Otherwise it means that the corresponding structure must be desallocated.

5.5.3 The Clear function

This function is called when th `clear` command is called. It must "clear" the structure corresponding to the `VALUE`. In our case, we will just erase all the circles, i.e.,

```
void ClearCircles(CIRCLES c)
{
  if (c->n != 0) {
    Free(c->array);
    c->array = NULL;
    c->n = 0;
  }
}
```

5.5.4 The Copy function

This function is called whenever the `copy` command is called. It takes two arguments. The first one if the `VALUE` to be copied. If the second one is `NULL` the function must allocate and return a copy of the `VALUE`. If not the secong one is a `VALUE` that you must use to put the copy in and finally return it. In our case it is

```
CIRCLES  CopyCircles(CIRCLES in, CIRCLES out)
{
  if (out == NULL) out = NewCircles();

  ClearCircles(out);

  out->array = Malloc(sizeof(Circles)*in->n);
  out->n = in->n;

  memcpy(out->array,in->array,sizeof(Circles)*in->n);

  return(out);
}
```

5.5.5 The ToStr function

This function takes two arguments, the first one is the VALUE to be converted to a (static) string and the second one is a flag. If it set to YES it means that the string should be short (it will be used to display the VALUE when in a listv) if not the so obtained string will be used to display the VALUE when returned by a command as a result value. In our case:

```
char *ToStrCircles(CIRCLES c, char flagShort)
{
  static char str[30];

  if (!strcmp(c->name,"")) {
    sprintf(str,"<&circles[%d];%p>",c->n,c);
  }
  else if (strlen(c->name) < 15) {
    sprintf(str,"<&circles[%d];%s>",c->n,c->name);
  }
  else {
    sprintf(str,"<&circles[%d];...>",c->n);
  }
  return(str);
}
```

5.5.6 The Print function

This function is called whenever the print command is called. it should print extensively the VALUE. In our case:

```
void PrintCircles(CIRCLES c)
{
  int i;

  if (c->n == 0) Printf("<empty>\n");
  else {
    for (i=0;i<c->n;i++)
      Printf("%d : x=%g, y=%g, r= %g\n",i,c->array[i].x,c->array[i].y,c->array[i].r);
  }
}
```

5.5.7 The PrintInfo function

This function is called whenever the info command is called. it should print information on the VALUE. In our case:

```
void PrintInfoCircles(CIRCLES c)
{
  Printf("  name  :  %s\n",c->name);
  Printf("  number of circles  :  %d\n",c->n);
}
```

5.6 The NumExtract function

This function is used to interpret 10a like syntax where a is a VALUE of our type. In the case of CIRCLES we did not allow such a syntax. This is why we set this function to NULL. However we could have decided not to. In the case pf &wtrans (i.e., 1d wavelet transform structure defined in the package wtrans1d) this function is

```
static char *numdoc = "The syntax <ij> corresponds to A[i,j] and the syntax <.ij> correspon
static void *NumExtractWtrans(WTRANS val,void **arg)
{
  VALUE *pValue;
  int n;
  char flagDot;
  int v;
  int o;

     /* doc */
  if (val == NULL) return(numdoc);

  n = ARG_NE_GetN(arg);
  flagDot = ARG_NE_GetFlagDot(arg);
  v = n%10;
  o = n/10;

  if (o < 0 || o >= NOCT) {
    SetErrorf("Octave index '%d' out of range : should be in [0,%d]",o,NOCT-1);
    return(NULL);
  }
  if (v < 0 || v >= NVOICE) {
    SetErrorf("Voice index '%d' out of range : should be in [0,%d]",v,NVOICE-1);
    return(NULL);
  }

  if (flagDot) ARG_NE_SetResValue(arg,val->D[o][v]);
  else ARG_NE_SetResValue(arg,val->A[o][v]);
  return(signalType);
}
```

If the first argument (i.e., the VALUE) is NULL it must return a one-line help on the syntax. Otherwise, you can obtain the number (in front of the a) using the ARG_NE_GetN macro and the flag flagDot using the macro ARG_NE_GetFlagDot (it is YES in the case a dot is used atthe begining, e.g., .10a). The result must be a VALUE that is stored using the macro ARG_NE_SetResValue, moreover it must return its corresponding *&type*.

If you want to generate an error you must not use the Errorf function. You must return NULL and sets the error message before (using the SetErrorf function).

5.7 Managing fields : an introduction

The last line of the `TypeStruct` (see Section 5.2) must be an array of `field`'s decalration. In our case this array is in the variable `fieldsCircles`. A field declaration looks like

```
"name", Get, Set, GetExtractOption, GetExtractInfo
```

where `name` is the name of the field, `Get` the function that allows to get the value of the field, `Set` the function that allows to set the value of the field, `GetExtractOption` the function that allows to manage `*options` in extractions and `GetExtractInfo` the function that manages getting some information about extractions on this field (e.g., `signal.X[1,2]` is an extraction the `VALUE signal` and with the field `X`). If no extraction is possible with this field then both `GetExtractOption` and `GetExtractInfo` must be `NULL`. Moreover if the `name` of the field is the empty string it corresponds to extraction on the `VALUE` itself (i.e., `signal[2,3]`).

Let us note that if a `Set` function is set to `NULL` then the field is `read-only`. Moreover, when no extraction function (`GetExtractOption` and `GetExtractInfo`) is specified for a field, **it does not mean** that no extraction is available for this field. It just means that the extraction on this field will not be managed by this `VALUE`. In case extraction of the type `c.f[1,2]` is performed, LastWave first starts by getting the `VALUE` corresponding to `c` then it asks if this `VALUE` knows about extraction on a field named `f`, if it does then it asks this `VALUE` to process it. If it does not, then it asks the `VALUE c` to return the value for the field `f`. The returned `VALUE` is then asked to perform the extraction. This behavior allows you to choose whether you want the fields of the `VALUE` you define to handle extraction `themselves` or to bypass their handling.

In our case the array of fields' declaration is

```
struct field fieldsCircles[] = {

  "", GetExtractCirclesV, SetExtractCirclesV, GetExtractOptionsCirclesV, GetExtractInfoCirc
  "r", GetRCirclesV, SetRCirclesV, GetExtractOptionsRXYCirclesV, GetExtractInfoRXYCirclesV,
  "x", GetXCirclesV, SetXCirclesV, GetExtractOptionsRXYCirclesV, GetExtractInfoRXYCirclesV,
  "y", GetYCirclesV, SetYCirclesV, GetExtractOptionsRXYCirclesV, GetExtractInfoRXYCirclesV,
  "name", GetNameCirclesV, SetNameCirclesV, NULL, NULL,
  "n", GetNCirclesV, SetNCirclesV, NULL, NULL,

  NULL, NULL, NULL, NULL, NULL
};
```

We shall first describe fields which do not deal with extractions (the fields `n` and `names`)

5.8 Managing fields (no extraction)

Both the `Get` and the `Set` functions must return a documentation string if the `VALUE` that is the first argument is `NULL`. In, LastWave, several functions let you manage standard type fields very easily. For instance, the function that gets the field `n` (an integer) is just

```
static char *nDoc = "{[= <n>]} {Sets/Gets the number of circles of a circles object}";
static void * GetNCirclesV(CIRCLES c, void **arg)
{
  /* Documentation */
  if (c == NULL) return(nDoc);

  return(GetIntField(c->n,arg));
}
```

The function `GetIntField(int n, void **arg)` allows to return the integer `n` as the value of the field.
The function that allows to set this field is

```
static void * SetNCirclesV(CIRCLES c, void **arg)
{
  int n,i;

  /* doc */
  if (c == NULL) return(nDoc);

  n = c->n;

  if (SetIntField(&n,arg,FieldSPositive)==NULL) return(NULL);

  if (n<c->n) c->n = n;
  else {
    ClearCircles(c);
    c->array = Malloc(sizeof(Circles)*n);
    c->n = n;
    for (i= 0;i<n;i++) {
      c->array[i].x = 0;
      c->array[i].y = 0;
      c->array[i].r = 0;
    }
  }

  return(numType);
}
```

It is a little more complicated. However you must be aware that this function lets you deal with pretty complex syntaxes such as `c.n+=3`. In order to do so it uses the `SetIntField(int *p, void **arg, char flag)` function. This function must be called with a pointer to the field value as the first argument (if it is an integer of course), the `arg` argument as the second argument and a `flag` argument that is one of `FieldPositive`, `FieldSPositive`, `FieldNegative` or `FieldSNegative`. It manages setting the field using the constraint given by the `flag` (`FieldSPositive` indicates that the integer must be a strictly positive integer). Let us note that in our case we need to remember the old value of the field since it tells you

what the allocation size of the array is. This is why we used another variable `n` and called the `SetIntField` function on `&n` instead of `&c->n`. The `SetIntField` function returns `NULL` if an error occurred.

In the case the field is a string (this is the case of the `name` field), the functions basically follow the same logic except that the `GetIntField` and `SetIntField` functions are replaced by the `GetStrField` and `SetStrField` functions:

```
static char *nameDoc = "{[= <name>]} {Sets/Gets the name of a circles}";
static void * GetNameCirclesV(CIRCLES c, void **arg)
{
  /* Documentation */
  if (c == NULL) return(nameDoc);

  return(GetStrField(c->name,arg));
}
```

and

```
static void * SetNameCirclesV(CIRCLES c, void **arg)
{
  /* doc */
  if (c == NULL) return(nameDoc);

  return(SetStrField(&(c->name),arg));
}
```

Let us note that this shows you a case of a field which extraction is not handled by the `VALUE` but directly by the field itself. Indeed, though you did not specify any extraction behavior, you can use syntax such as `s = c.name[*no,2:39]` or `c.name[1:2] := "e"`. They are handled by the `TypeStruct` associated to the string `VALUE` (i.e., `STRVALUE`).

You can learn about all the available `Get...Field` and `Set..Field` functions by reading Section 5.11

5.9 Managing extraction (no field)

This is the case when you type something like `c[1:3]` where `c` is a `&circles`. No field is specified. Extraction is directly performed on the `VALUE` itself. In the array of field declaration `fieldsCircles`, it corresponds to the firs line, i.e., where the `field` name is the empty string.

5.9.1 The `GetExtractOption` function

The `GetExtractOption` allows you to specify what the `*options` are in case of a get. **Even if there are no `*options` this function must not be set to `NULL`**. This function must return a (`NULL` terminated) list of strings corresponding to the `*options`. In the case the `VALUE` is `NULL` it must return a documentation on these `*options`. In our case, we only

want to deal with the `*nolimit` option which allows to usie out of range indices. The `GetExtractOptionsCirclesV` looks like

```
static char *optionDoc = "{{*nolimit} {*nolimit : indexes can be out of range}}";
static char *extractOptionsCircles[] = {"*nolimit",NULL};
enum {
  FSIOptCirclesNoLimit = FSIOption1
};

static void *GetExtractOptionsCirclesV(CIRCLES c, void **arg)
{
   /* doc */
  if (c == NULL) return(optionDoc);

  return(extractOptionsCircles);
}
```

Let us note that in the case of several `*option`'s the documentation syntax would be

```
"{{*option1} {doc of option1}} ...  {{*optionN} {doc of optionN}}"
```

Moreover, we define `enum` values for each of the options. As we will see this will be more practical. After dealing with eventual `*option`'s, LastWave will call the `ExtractInfo` function.

5.9.2 The `ExtractInfo` function - the `ExtractInfo` structure

This function is called before the list of indices is read. It asks for a information about how the extraction indices should look like. If the `VALUE` happens to be such that no extraction is possible (e.g., an empty signal) you must return `NULL` after setting the error message using the `SetErrorf` function (no error interruption must be made in this function). In any other case the function must return a pointer to a structure of type `ExtractInfo` that contains the required information. Some of the information does not depend on the `VALUE` (this is the case generally of the minimum allowed index) so for efficiency purpose, we use a static `ExtractInfo` variable and when the function `ExtractInfo` is first called we initialize it. In the next calls, only the part of this structure that depends on the `VALUE` will be changed. In our case it gives

```
static void *GetExtractInfoCirclesV(CIRCLES c, void **arg)
{
  static ExtractInfo extractInfo;
  static char flagInit = YES;

  unsigned long *options = ARG_EI_GetPOptions(arg);

  if (flagInit == YES) {
    extractInfo.nSignals = 1;
```

```
    extractInfo.xmin = 0;
    extractInfo.dx = 1;
  }

  if (c->n == 0) {
    SetErrorf("No extraction possible on empty list of circles");
    return(NULL);
  }

  extractInfo.xmax = c->n-1;

  extractInfo.flags = EIIntIndex;
  if (!(*options & FSIOptCirclesNoLimit)) extractInfo.flags |= EIErrorBound;

  return(&extractInfo);
}
```

The macro `ARG_EI_GetPOptions` allows to get obtain the flag that corresponds to the `*options`'s that are on (using the `enum` of the last section). The fields of the `ExtractInfo` structure that must be filled in are

- `unsigned char nSignals`: 1 or 2 depending on whether one or two (`;` separated) list of extraction indices are expected.
- `char flags`: a combination of `EIIntIndex` and/or `EIErrorBound`. If you set `EIIntIndex` it means that the indices are expected to be integers (error will be automatically generated if not the case) and if you set `EIErrorBound` it means that an error is generated if the indices are out of range (cf below)
- `LWFLOAT xmin,xmax,dx`: the range of the indices. If `xmax<xmin` then this range is not active. Otherwise the indices must be in the range `[xmin,xmax]` and a multiple of `dx`. Actually this range is the range for the first list of `indices` only (in case `nSignals` is equal to 2)
- `LWFLOAT ymin,ymax,dy`: same as `LWFLOAT xmin,xmax,dx` except that it applies on the second list of ranges.

In our case the indices must be integers (flag `EIIntIndex` is set) they must be in the range `[0,n-1]` where `n` is the number of circles. An out of range error must be generated (i.e., the flag `EIErrorBound` must be set) only if the `*option *nolimit` is not set.

5.9.3 The `GetExtract` function - The `FSIList` structure

Now we are ready for extraction. The list of indices is stored in a structure called an `FSIList` structure. Let us note that a list of indices can be composed of numbers, signals, ranges, images... The `FSIList` structure is used to code all this information without duplicating anything (for efficiency purposes a range, for instance, *is not* transformed into a signal). Though it means that it is very efficient, it makes the access to the structure a little trickier. However, most of the times, you will not have to access this structure directly. There are

several macros that let you deal wit this structure. The fields of this structure you must know about are

- `char flagImage`: `YES` if two (`;` separated) list of indices are specified
- `int nx`: the total number of indices of the first list
- `int ny`: the total number of indices of the second list
- `int nx1`: the total number of indices of the first list which are not out of range
- `int ny1`: the total number of indices of the second list which are not out of range
- `unsigend long options`: the `*option`'s which are on (using the `enum` coding)

The main macros you should know about are

- `FSI_FIRST(list)`: returns the first index (you must first check there is at least 1 index)
- `FSI_SECOND(list)`: returns the second index (you must first check there are at least 2 indices)
- `FSI_FOR_START(list)` and `FSI_FOR_END`: these macros allow you to loop on the indices. In order to use them you must make the declaration `FSI_DECL;` along the variable declaration. It works as a `for` loop. The syntax is

```
FSI_DECL;
...
FSI_FOR_START(list)
...
FSI_FOR_END
```

  In between the two calls some variables can be used

  - `_f`: the current (float) index
  - `_i`: corresponds to the integer part of `_f`
  - `_k`: the current index is the `_k`th index

  You can go out of the loop at any moment using the macro `FSI_BREAK`

The `GetExtract` function must return the extracted result in a pointer variable and it must return its corresponding *&type*. If `NULL` is returned then it means than an error occurred and that the error message was set (no error interruption are allowed). If the input `VALUE` is `NULL` it must return a string documentation. In our case of extraction on `CIRCLES` we want this function to return a new `CIRCLES` value with the corresponding circles only. In the case the `*nolimit` option is set only indices which are not out of range are taken into account. Thus we write

```
static char *doc = "{[*nolimit,...] [:]= list of <x,y,r>} {Get/Set the circle values}";
static void *GetExtractCirclesV(CIRCLES c, void **arg)
{
  FSIList *fsiList;
  VALUE *pValue;
  CIRCLES c1;
  FSI_DECL;

  /* doc */
  if (c == NULL) return(doc);

  fsiList = ARG_G_GetFsiList(arg);

  c1 = NewCircles();
  TempValue(c1);
  ARG_G_SetResValue(arg,c1);

  if (fsiList->nx1 == 0) return(circlesType);

  c1->array = Malloc(sizeof(Circle)*fsiList->nx1);
  c1->n = fsiList->nx1;

  FSI_FOR_START(fsiList);
  if (fsiList->options & FSIOptCirclesNoLimit && (_i<0 || _i >= c->n)) continue;
  memcpy(&(c1->array[_k]),&(c->array[_i]),sizeof(Circle));
  FSI_FOR_END;

  return(circlesType);
}
```

The macro `ARG_G_GetFsiList` allows to get the `FSIList` structure and the macro `ARG_G_SetResValue` allows to set the variable that must contain the result `VALUE` at return time (you could use the `ARG_G_GetResPValue(arg)` to get a pointer to this variable). Let us note that if you want to return a float or a string you can use directly the macros `ARG_G_SetResFloat(arg,flt)` (or `ARG_G_GetResPFloat(arg)` to get the pointer) or `ARG_G_SetResStr(arg,str)` (or `ARG_G_GetResPStr(arg)` to get the pointer). They avoid creating `NUMVALUE` or `STRVALUE` structure and consequently are more efficient.

5.9.4 The `SetExtract` function

We are now ready to use extraction for setting values. As for the other functions, if a `NULL VALUE` is passed as an argument the function should return a string documentation. Moreover, as for the `GetExtract` function (see previous Section), the list of indices will be coded using the `FSIList` structure. For our example, for the sake of simplicity, we will allow only two extraction syntaxes for `CIRCLES`, the first one

```
c[list of indices] := {}
```

should delete each circle which corresponds to one of the index (in that case, the list of indices must be made of strictly increasing indidces) and the second one

```
c[list of indices] = listv of <x y r>
```

in which the list of indices must have the same length as the listv on the right handside. The corresponding function is the following

```
static void *SetExtractCirclesV(CIRCLES c,void **arg)
{
  FSIList *fsiList = (FSIList *)  ARG_S_GetFsiList(arg);
  char *equal = ARG_S_GetEqual(arg);
  VALUE val = ARG_S_GetRightValue(arg);
  char *type = ARG_S_GetRightType(arg);

  LISTV lv;
  FSI_DECL;
  LWFLOAT f;
  SIGNAL sig;
  int _iold, i;

  /* doc */
  if (c == NULL) return(doc);

  /* The right handside value must be a listv */
  if (type != listvType) {
    SetErrorf("Right hand side of assignation should be a &listv");
    return(NULL);
  }

  /* Cast the right handside value to a listv */
  lv = CastValue(val,LISTV);

  /*********************************
   *
   * Case the assignation is of the form c[fsilist] := {}
   *
   *********************************/
  if (*equal == ':') {

   /* If (right handside) listv is not empty --> error */
    if (lv->length != 0) {
      SetErrorf("With := syntax, right handside should be an empty listv");
      return(NULL);
    }

    /* Testing the indices of the FSIlist are strictly increasing */
```

```
  _iold = -1;
  FSI_FOR_START(fsiList);
  if (_i <= _iold) {
    SetErrorf("Indices should be strictly increasing");
    return(NULL);
  }
  _iold = _i;
  FSI_FOR_END;

  /* Let's perform assignation : let's remove the corresponding circles ! */
  _iold = -1;
  i = 0;
  FSI_FOR_START(fsiList);
  if (_i == 0) {
    _iold = 0;
    continue;
  }
  else if (_iold == -1) {
    i = _i;
    _iold = _i;
    continue;
  }
  if (_i == _iold+1) {
    _iold = _i;
    continue;
  }
  memmove(&(c->array[i]),&(c->array[_iold+1]),(_i-_iold-1)*sizeof(Circle));
  i+=_i-_iold-1;
  _iold = _i;
  FSI_FOR_END;
  if (_i != c->n-1) {
    memmove(&(c->array[i]),&(c->array[_iold+1]),(c->n-1-_iold)*sizeof(Circle));
  }

  c->n -= fsiList->nx;

  /* we return the same structure CIRCLES (which has been modified) */
  ARG_S_SetResValue(arg,c);
  return(circlesType);
}


/*********************************
 *
 * Case the equal sign is one of +=, -=, *=, /=, ^=  --> ERROR
 *
```

```
 *********************************/
if (*equal != '=') {
  SetErrorf("%s syntaxnot valid",equal);
  return(NULL);
}


/*********************************
 *
 * Case the assignation is of the form c[fsilist] = listv of signals of size 3 (<x,y,r>)
 *
 *********************************/

/* The number of indices must be the same as the number of signals */
if (lv->length != fsiList->nx) {
  SetErrorf("Right and left handside should have the same size");
  return(NULL);
}

/* Let's loop on the indices */
FSI_FOR_START(fsiList);
GetListvNth(lv,_k,&val,&f);

/* We only expect non empty signals in the listv .... */
if (val == NULL || GetTypeValue(val) != signaliType) {
  SetErrorf("Expect a listv of non empty signals on right handside");
  return(NULL);
}
sig = CastValue(val,SIGNAL);

/* .... signals that must be of size 3 */
if (sig->size != 3) {
  SetErrorf("Expect a listv of signals of length 3 on right handside");
  return(NULL);
}

/* .... and with positive radius */
if (sig->Y[2] < 0) {
  SetErrorf("Radius must be positive");
  return(NULL);
}

/* Let's perform the assignation */
c->array[_i].x = sig->Y[0];
c->array[_i].y = sig->Y[1];
c->array[_i].r = sig->Y[2];
```

```
  /* end of loop */
  FSI_FOR_END;

  /* we return the same structure CIRCLES (after modification)
  ARG_S_SetResValue(arg,c);
  return(circlesType);
}
```

Let us note that the equal sign that is used for assignation (=, +=, :=, ...) is obtained using the macro `ARG_S_GetEqual`. The right handside type value is obtained using `ARG_S_GetRightType`. If it is a float, it is obtained using `ARG_S_GetRightFloat` otherwise it corresponds to a value that is obtained using `ARG_S_GetRightValue`.

5.10 Managing extraction with field

In this section we explain how to make LastWave understand a syntax like

```
c.x[1,2] = <0,0>
```

where `c` is a `CIRCLES`. Since the field `x` is implemented using an array of float (which is not a `VALUE`) there is no way the extraction of the field can be handled by the field itself. It must be directly handled by `c`.

The extraction system with a field works exactly the same way as the extraction system with no field (read Section 5.9) except that, if you need it you can get the name of the field using the macros `ARG_G_GetField` (for the `get` functions), `ARG_S_GetField` (for the `set` functions), `ARG_EI_GetField` (for the `GetExtractInfo` functions) and `ARG_EO_GetField` (for the `GetExtractOptions` functions).

In our case, we want to manage extraction of fields `x`, `y` and `r`. Since the functions will look alike, all the functions will call the same basic functions.

5.10.1 The `GetExtractOption` function

We use the same functions for the three fields. Since there is no `*option` the function must return a single element array with `NULL`, i.e.,

```
static void *GetExtractOptionsRXYCirclesV(CIRCLES c, void **arg)
{
  static char *gextractOptionsCircles[] = {NULL};
  return(gextractOptionsCircles);
}
```

5.10.2 The `GetExtractInfo` function

A single function is used again

```
static void *GetExtractInfoRXYCirclesV(CIRCLES c, void **arg)
{
  static ExtractInfo extractInfo;
  static char flagInit = YES;

  unsigned long *options = ARG_EI_GetPOptions(arg);

  if (flagInit == YES) {
    extractInfo.nSignals = 1;
    extractInfo.xmin = 0;
    extractInfo.dx = 1;
    extractInfo.flags = EIIntIndex | EIErrorBound;
  }
  if (c->n == 0) {
    SetErrorf("No extraction possible on empty list of circles");
    return(NULL);
  }

  extractInfo.xmax = c->n-1;

  return(&extractInfo);
}
```

5.10.3 The GetExtract functions

We start with the generic function (`flag` is either `'x'`, `'y'` or `'r'` depending on the field which is treated). For the sake of simplicity, in order to handle the `get` we build the corresponding signal (made only of the abscissa, ordinates or radii) and we call the `get` of the signal type. Since this `get` function knows how to handle `signal` fields, we first must set the field (which is a `CIRCLES` field not a `SIGNAL` field) to NULL. This is done using the `ARG_G_SetField` macro. Since the resulting extracted signal could have an `x0` field different from 0 or a `dx` field different from 1, we set these two fields.

```
static void * GetRXYCirclesV(CIRCLES c, void **arg, char flag)
{
  SIGNAL sig;
  int i;
  void *res;

  sig = TNewSignal();
  if (c->n != 0) SizeSignal(sig,c->n,YSIG);

  switch(flag) {
  case 'r' : for (i=0;i<c->n;i++) sig->Y[i] = c->array[i].r; break;
  case 'x' : for (i=0;i<c->n;i++) sig->Y[i] = c->array[i].x; break;
  case 'y' : for (i=0;i<c->n;i++) sig->Y[i] = c->array[i].y; break;
  }
```

```
  ARG_G_SetField(arg,NULL);
  res = GetSignalExtractField(sig,arg);
  if (res == NULL) return;
  if (res == signaliType) {
    sig = *((SIGNAL *) ARG_G_GetResPValue(arg));
    sig->dx = 1;
    sig->x0 = 0;
  }
}
```

The specific function for the `r` field is really simple

```
static char *rDoc = "{[[+-*/:]= (<signal> | <range>])} {Sets/Gets the radii}";
static void * GetRCirclesV(CIRCLES c, void **arg)
{
  if (c == NULL) return(rDoc);
  return(GetRXYCirclesV(c,arg,'r'));
}
```

and so are the other ones

```
static char *xDoc = "{[[+-*/:]= (<signal> | <range>])} {Sets/Gets the abscissa}";
static void * GetXCirclesV(CIRCLES c, void **arg)
{
  if (c == NULL) return(xDoc);
  return(GetRXYCirclesV(c,arg,'x'));
}

static char *yDoc = "{[[+-*/:]= (<signal> | <range>])} {Sets/Gets the ordinates}";
static void * GetYCirclesV(CIRCLES c, void **arg)
{
  if (c == NULL) return(yDoc);
  return(GetRXYCirclesV(c,arg,'y'));
}
```

5.10.4 The `SetExtract` functions

In the same way as for the `get` function, we start writing a generic function. Again, for the sake of simplicity, we will build `SIGNAL`s and call the corresponding `set` function (i.e., the function `SetSignalField` (you can learn about all the available `Get...Field` and `Set..Field` functions by reading Section 5.11). However, we have to be careful that the syntax for signals let you change the size of the signal (e.g. `s[1:2]:=0`), so we shall not allow this type of syntax. This will simply be done by checking that after call the `set` function for `SIGNAL`s the resulting signal has the same size as the original signal.

```
static void * SetRXYCirclesV(CIRCLES c, void **arg, char flag)
{
  SIGNAL sig;
```

```
  int i;
  void *res;

  sig = TNewSignal();
  if (c->n != 0) SizeSignal(sig,c->n,YSIG);

  switch(flag) {
  case 'r' : for (i=0;i<c->n;i++) sig->Y[i] = c->array[i].r; break;
  case 'x' : for (i=0;i<c->n;i++) sig->Y[i] = c->array[i].x; break;
  case 'y' : for (i=0;i<c->n;i++) sig->Y[i] = c->array[i].y; break;
  }

  if ((res = SetSignalField(sig,arg)) == NULL) return(NULL);

  if (sig->size != c->n) {
    SetErrorf("Sorry, right handside should have the same size as left handside");
    return(NULL);
  }

  if (flag == 'r') {
    for (i=0;i<c->n;i++) {
      if (sig->Y[i]<0) {
        SetErrorf("Sorry, radii should be positive");
        return(NULL);
      }
    }
  }

  switch(flag) {
  case 'r' : for (i=0;i<c->n;i++) c->array[i].r = sig->Y[i]; break;
  case 'x' : for (i=0;i<c->n;i++) c->array[i].x = sig->Y[i]; break;
  case 'y' : for (i=0;i<c->n;i++) c->array[i].y = sig->Y[i]; break;
  }

  return(res);
}
```

The specific functions are then very simple

```
static void * SetRCirclesV(CIRCLES c, void **arg)
{
  if (c == NULL) return(rDoc);
  return(SetRXYCirclesV(c,arg,'r'));
}

static void * SetXCirclesV(CIRCLES c, void **arg)
{
```

```
  if (c == NULL) return(xDoc);
  return(SetRXYCirclesV(c,arg,'x'));
}
```

and

```
static void * SetYCirclesV(CIRCLES c, void **arg)
{
  if (c == NULL) return(yDoc);
  return(SetRXYCirclesV(c,arg,'y'));
}
```

5.11 The `Get...Field` and the `Set...Field` functions

5.11.1 The `Get...Field` functions

As we have seen the `Get...Field` functions can be used to avoid handling directly with the `get` system. The extensive list of these functions are

- `void *GetIntField(int i, void **arg)`: gets the field which corresponds to the integer `i`
- `void *GetFloatField(LWFLOAT f, void **arg)`: gets the field which corresponds to the float `f`
- `void *GetStrField(char *str, void **arg)`: gets the field which corresponds to the string `str`. This function does not manage extraction (both the `FSIList` and the `field` are set to `NULL`).
- `void *GetValueField(VALUE val, void **arg)`: gets the field which corresponds to the `VALUE val`. This function does not manage extraction (both the `FSIList` and the `field` are set to `NULL`).
- `void * GetStrFieldExtract(char *str, void **arg)`: gets the field which corresponds to the string `str`. This function manages extraction (however the `field` is set to `NULL`).
- `void *GetSignalExtractField(SIGNAL sig, void **arg)`: gets the field which corresponds to the `SIGNAL sig`. This function manages extraction (however the `field` is set to `NULL`).
- `void *GetImageExtractField(IMAGE im, void **arg)`: gets the field which corresponds to the `IMAGE im`. This function manages extraction (however the `field` is set to `NULL`).

5.11.2 The `Set...Field` functions

As we have seen the `Set...Field` functions can be used to avoid handling directly with the `set` system. The extensive list of these functions are

- `void *SetIntField(int *pint, void **arg,char flag)`: sets the field which corresponds to the integer pointed by `pi`. The `flag` is one of `FieldPositive` (result must be positive), `FieldSPositive` (result must be strictly positive), `FieldNegative` (result must be negative), `FieldSNegative` (result must be strictly negative).
- `void *SetFloatField(LWFLOAT *pflt, void **arg,char flag)`: sets the field which corresponds to the float pointed by `pflt`. The `flag` is one of `FieldPositive` (result must be positive), `FieldSPositive` (result must be strictly positive), `FieldNegative` (result must be negative), `FieldSNegative` (result must be strictly negative).
- `void *SetStrField(char **pstr, void **arg)`: sets the field which corresponds to the string pointed by `pstr`. This function manages extraction (however the `field` is set to `NULL`)
- `void *SetListvField(LISTV *plv,void **arg)`: sets the field which corresponds to the listv pointed by `plv`. This function manages extraction (however the `field` is set to `NULL`)
- `void *SetSignalField(SIGNAL sig,void **arg)`: sets the field which corresponds to the `SIGNAL sig`. This function manages extraction (however the `field` is set to `NULL`)
- `void *SetImageField(IMAGE im,void **arg)`: sets the field which corresponds to the `IMAGE im`. This function manages extraction (however the `field` is set to `NULL`)

5.12 Playing around with CIRCLES

This is it! Your package is ready (do not forget to include the file `circles.c` in the `FileList` file in the directory `user/obj` (as explained in Section 3.1). Just type `make` in the `Makefiles` directory to recompile LastWave and start it up!
This is an example of what you can do with the `circles` package

```
> package load 'circles'
> c = [new &circles]
= <&circles[0];0x090fe300>
> c.n=10
= 10
> c
= <&circles[10];0x090fe300>
> c.r = abs(Grand(10))
= <size=10;0.287257,0.0415209,1.28227,0.0587326,0.189727,1.18633,...>
> c.x = 0:9
= <size=10;0,1,2,3,4,5,...>
> c.y = 2*c.x
= <size=10;0,2,4,6,8,10,...>
> c.r[3:5]
= <0.0587326,0.189727,1.18633>
> c.r[3:5] := 8
```

```
= <size=10;0.287257,0.0415209,1.28227,8,8,8,...>
> print c
c =
0 : x=0, y=0, r= 0.287257
1 : x=1, y=2, r= 0.0415209
2 : x=2, y=4, r= 1.28227
3 : x=3, y=6, r= 8
4 : x=4, y=8, r= 8
5 : x=5, y=10, r= 8
6 : x=6, y=12, r= 0.0458967
7 : x=7, y=14, r= 1.49387
8 : x=8, y=16, r= 1.37574
9 : x=9, y=18, r= 0.20473
> find(c.r>1)
= <size=6;2,3,4,5,7,8>
> c[find(c.r>1)] := {}
= <&circles[4];0x091b3ab0>
> print c
c =
0 : x=0, y=0, r= 0.287257
1 : x=1, y=2, r= 0.0415209
2 : x=6, y=12, r= 0.0458967
3 : x=9, y=18, r= 0.20473
```

Chapter 6

Managing graphics

6.1 Graphic objects

6.1.1 The `GOBJECT` structure

We have already seen that values in LastWave "inherit" from the type `VALUE`. By "inherit" we meant that all the C-structures that are used to represent values have a common header, i.e., they start with the same fields. This set of common fields are grouped into the `Value` structure. The type `VALUE` is the type which corresponds to a pointer to a `Value` structure. Exactly in the same way, all the graphic objects inherit from the type `GOBJECT` which corresponds to a pointer to the structure `Gobject`. The fields of `Gobject` you should know about are the following

- `GCLASS gclass`: a pointer to the graphic class it belongs to (read next section)
- `GLIST father`: the graphic list it belongs to (if any)
- char flagHide: `YES` if object is not visible
- `int x, y, w, h`: the absolute position/size in the window coordinates
- `LWFLOAT rx, ry, rw, rh`: the relative position/size in the graphic list coordinates
- `RectType rectType`: where `RectType` is a structure with the 4 (*short*) fields: `left`, `top`, `right` and `bottom`. Each time the absolute position is computed from the relative position, the so-obtained rectangle is enlarged/reduced on each side using these 4 numbers. There are 3 predefined `RectTypes`:
  - `NormalRect = {0,0,0,0}` (default value): the point `(x,y)` belongs to the object whereas the point `(x+w,y+h)` does not belong to the object
  - `LargeRect = {0,0,1,1}`: both the points `(x,y)` and `(x+w,y+h)` belong to the object
  - `SmallRect = {-1,-1,0,0}`: neither the point `(x,y)` nor the point `(x+w,y+h)` belong to the object
- `unsigned char flagClip`: the possible values are

  - `0`: the object is not clipped
  - `1`: the object is clipped using the rectangle `x,y,w,h`
  - `2`: the object is clipped using the rectangle `x,y,w,h` only when displayed on the screen. It is not clipped when a postscript file is generated.

- `char flagGrid`: `YES` if the object is positionned/sized on a grid
- `unsigned char i,j,m,n`: position/size (relative to grid coordinates) if `flagGrid` is set.
- `FONT font`: font of the object
- `unsigned long fgColor, bgColor`: foreground and background colors
- `unsigned char penSize`: size of the pen
- `unsigned char penMode`: mode of the pen (either `PenPlain` or `PenInverse`)
- `unsigned char lineStyle`: style of line (either `LinePlain` or `LineDash`)
- `unsigned char flagFrame`: `YES` if object is framed

6.1.2 Parsing graphic objects: `tGOBJECT`, `tGOBJECT_`, `tGOBJECTLIST` and `tGOBJECTLIST_`

In order to write a command that uses a graphic object, you need to be able to parse such a graphic object. This can be done using the `ParseArgv` function along with tth `tTYPES`, `tGOBJECT` or `tGOBJECT_`. You can also use directly one of the parsing functions

- `void ParseGObject(char *arg, GOBJECT *obj)`
- `char ParseGObject_(char *arg, GOBJECT def, GOBJECT *obj)`

Let us note that this does not allow you to parse a list of graphic objects (as most of the commands dealing on graphic objects do, indeed, for instance you canuse wild cards). A list of graphic objects is simply represented in `C` using a `NULL` terminated array of `GOBJECT`. You can parse such a list using the `tTYPES tGOBJECTLIST` or `tGOBJECTLIST_` or directly using the parsing functions:

- `void ParseGObjectList(char *arg, GOBJECT **objlist)`
- `char ParseGObjectList_(char *arg, GOBJECT *def, GOBJECT **objlist)`

6.1.3 Functions that deal with local/global coordinates

- `void Local2Global(GOBJECT o,LWFLOAT x, LWFLOAT y, int *mx, int *my)`: converts local coordinates (i.e., using `o` coordinates) of a point to global (window) coordinates.
- `void Global2Local(GOBJECT o,int mx, int my, LWFLOAT *x, LWFLOAT *y)`: reverse of `Local2Global`

- `void Local2GlobalRect(GOBJECT o,LWFLOAT x, LWFLOAT y, LWFLOAT w, LWFLOAT h, RectType rectType, int *mx, int *my, int*mw, int*mh)`: converts local coordinates (i.e., using `o` coordinates) of a rectangle to global (window) coordinates. The type of the rectangle (see Section 6.1.1) is specified by `rectType`).
- `Global2LocalRect(GOBJECT o,int mx, int my, int mw, int mh, LWFLOAT *x, LWFLOAT *y, LWFLOAT *w, LWFLOAT *h, RectType rectType)`: reverse of `Global2LocalRect`

6.1.4 Useful functions that deal with windows

- `char IsWin(GOBJECT o)`: returns `YES` if the object is a `WINDOW` (in which case you can cast `o` to the type `WINDOW`)
- `WINDOW GetWin(GOBJECT o)`: gets the window the object `o` is in.

6.2 Graphic classes

6.2.1 The GCLASS structure

Again, as for values (`VALUE`) and graphic objects (`GOBJECT`), graphic classes in LastWave "inherit" from the type `GCLASS`. The type `GCLASS` is the type which corresponds to a pointer to a `GClass` structure. The fields of `Gclass` you should know about are the following

- `GCLASS fatherClass`: the graphic class it inherits from (the class att he very top is `theGObjectClass`)
- `size_t nbBytes`: the number of bytes for allocation of a graphic object
- `unsigned long flags`: combination of
  - `GClassProtected`: if set, the class cannot be overdefined (is set by default)
  - `GClassMoveResize`: if set, the graphic objects of that class can be moved and resized (is set by default)
  - `GClassLocalCoor`: if set, each graphic object has its own local coordinates (is set by default)
- char *varType: if not `NULL` this field must correspond to an *&type*. It means that this graphic class will be used to display variables of type *&type*.
- char *info: a description of the graphic class.

Then, there are a set of C-functions that will be fully described in the following sections.

- `void (*init) (GOBJECT gobj)`: C-function for initialization of a graphic object (see Section 6.2.2)
- `void (*deleteContent) (GOBJECT gobj)`: C-function for desallocation of the content of a graphic object (the graphic object itself is not freed) (see Section 6.2.3)
- `void (*draw) (WINDOW win, GOBJECT gobj, int x, int y, int w, int h)`: C-function that specifies how the graphic objects should be drawn (see Section 6.2.4).

- `int (*set)(GOBJECT gobj, char *field, char**argv)`: C-function for managing fields (see Section 6.2.5)
- `char (*msge) (GOBJECT gobj, char *msge, char**argv)`: C-function for managing messages (see Section 6.2.6)
- `LWFLOAT (*isIn)(GOBJECT gobj,GOBJECT *pobj, int x, int y)`: C-function that test wether a point is within a graphic object or not (see Section 6.2.7)

6.2.2 The `init` function

The `init` field of `GCLASS` corresponds to the type

```
void (*init) (GOBJECT gobj)
```

This C-function is called right after a graphic object `gobj` of the corresponding graphic class is created. It is used to initialize this graphic object.

6.2.3 The `deleteContent` function

The `deleteContent` field of `GCLASS` corresponds to the type

```
void (*deleteContent) (GOBJECT gobj)
```

This C-function is called right before the graphic object `gobj` is deleted. It must delete any allocation that was made within the graphic object (it must not delete the graphic object itself!).

6.2.4 The `draw` function

The `draw` field of `GCLASS` corresponds to the type

```
void (*draw) (WINDOW win, GOBJECT gobj, int x, int y, int w, int h)
```

This C-function is called whenever a part of the graphic object `gobj` that belongs to the window `win` must be redrawn. The part is specified in absolute (window) coordinates by `x`, `y`, `w` and `h`.

A few important things happen before `draw` is called: Before the draw function of a graphic object `gobj` is called

- Depending on the field `flagClip` of `gobj` (see Section 6.1.1), the current clipping rectangle (see Section 6.3.4) might be set to the (absolute coordinate) rectangle of `gobj`.
- The rectangle corresponding to `gobj` is cleared using the background color (i.e., the field `bgColor`) of `gobj` (see Section 6.3.3). If `bgColor` is `invisibleColor` nothing happens

- The current pen mode (see Section 6.3.1) is set to the `penMode` field of `gobj`, the current pen size (see Section 6.3.1) is set to the `penSize` field of `gobj`
- The current line style (see Section 6.3.2) is set to the `lineStyle` field of `gobj`
- The current color (see Section 6.3.3) is set to the `fg` field of `gobj`
- If the field `flagFrame` of `gobj` is set then a rectangle frame is drawn (using the color `fg`) corresponding to the rectangle of `gobj`.

6.2.5 The `set` function

The `set` field of `GCLASS` corresponds to the type

```
int (*set)(GOBJECT gobj, char *field, char**argv)
```

This C-function handles setting/getting fields of the graphic object `gobj`. The result of that function must be set using the `SetResult...()` functions (as you do for a command). If `gobj` is `NULL`, then the result must be a string which corresponds to the help on all the fields. The help string must have the following syntax (it is similar to the syntax of the help of a command which has actions)

```
"{{{fieldUsage1} {fieldHelp1}} ...{{fieldUsageN} {fieldHelpN}}}
```

If `gobj` is not `NULL` then `field` corresponds to the field name (without the `-` character) to be set or get and `argv` is the (`NULL` terminated) argument list that can be read in the same way as the argument list of a command (using the `Parse...` functions). Generally if there is no argument (i.e., `argv[0]` is `NULL`) it means that the result should correspond to the field value (i.e., the field is `get`). Otherwise it means it must be set.
If the field is an unknown one then `NO` should be returned (you can generate an error using `Errorf` if the field is known but the syntax is incorrect). If `YES` is returned it means that everything went well. In case of a `set`, if it returns `-1` it means that the value of the field was already set to the same value.

6.2.6 The `msge` function

The `msge` field of `GCLASS` corresponds to the type

```
char (*msge) (GOBJECT, char *msge, char**argv)
```

This C-function handles messages sent to the graphic object `gobj`.
If `gobj` is `NULL`, then the result must be a string which corresponds to the help on all the messages. The help string must have the following syntax (it is similar to the syntax of the help of a command which has actions)

```
"{{{msgeUsage1} {msgeHelp1}} ...{{msgeUsageN} {msgeHelpN}}}
```

If gobj is not NULL then msge corresponds to the msge sent to gobj. argv corresponds to the (NULL terminated) list of arguments that can be read in the same way as the argument list of a command (using the Parse... functions). The result of that function must be set using the SetResult...() functions (as you do for a command).
It should return YES if the message is a valid message and NO if it is an unknown one (you can generate an error using Errorf if the message is known but the syntax is incorrect).

6.2.7 The isIn function

The isin field of GCLASS corresponds to the type

```
LWFLOAT (*isIn)(GOBJECT gobj, GOBJECT *pobj, int x, int y)
```

This C-function is used to test wether the point x,y (in absolute coordinate) is within the graphic object gobj or not. If it is not it should return a negative number. In any other cases, pobj should be set to the graphic object the point is closest to. Generally it is gobj, however, in the case of a GLIST it must be set to the closest graphic object of the GLIST. If 0 is returned then it means that the point is in the graphic object gobj and LastWave should not look for any other graphic objects the point could be in. If you return a strictly positive number, it must correspond to a distance of the point to the graphic object gobj. LastWave will then search for the closest graphic object.
This routine is generally used to decide which graphic object should receive a mouse event (in any case, one one object receives the event). LastWave looks for the one which is closest to the mouse. If a distance of 0 is found, then LastWave sends the event to that object.

6.2.8 Parsing graphic classes: tGCLASS and tGCLASS_

In order to write a command that uses a graphic class, you need to be able to parse such a graphic class. This can be done using the ParseArgv function along with tth tTYPES, tGCLASS or tGCLASS_. You can also use directly one of the parsing functions

- void ParseGClass(char *arg, GCLASS *class)
- char ParseGClass_(char *arg, GCLASS def, GCLASS *class)

6.3 Drawing!

6.3.1 The pen

The pen is used for all the drawing procedures. There are 2 pen modes

- PenPlain: regular
- PenInverse: inverse color is used

The current pen mode is set using

```
void WSetPenMode(WINDOW win, int mode)
```

and the pen size using

```
void WSetPenSize(WINDOW win,int size)
```

6.3.2 The line style

The line style is used when drawing lines (ellipses, rectangles,...). The line style can be

- `LinePlain`: solid line
- `LineDash`: dash line

The current line style is set using

```
void WSetLineStyle(WINDOW win,int style)
```

6.3.3 Managing colors and colormaps: `tCOLOR`, `tCOLOR_`, `tCOLORMAP` and `tCOLORMAP_`

In LastWave, both colors and colormaps are represented using the type `unsigned long`. In order to parse colormaps and colors you can use the parsing functions

- `void ParseColorMap(char *arg, unsigned long *colormap)`
- `char ParseColorMap_(char *arg, unsigned long defVal, unsigned long *colormap)`
- `void ParseColor(char *arg, unsigned long *color)`
- `char ParseColor_(char *arg, unsigned long defVal, unsigned long *color)`

You might need to use the global variables

- `unsigned long bgColor`: the current background color
- `unsigned long fgColor`: the current foreground color
- `const unsigned long invisibleColor`: the transparent color

To test wether a color `c` is transparent you should use the test `(c &invisibleColor)` and not the test `(c == invisibleColor)`.
Some useful functions are

- `unsigned long GetColorMapCur(void)`: returns the current colormap
- `char *GetColorMapName (unsigned long cm)`: get the name of the colormap `cm`
- `int ColorMapSize(unsigned long colorMap)` returns the number of colors in a colormap
- `char *GetColorName(unsigned long color)` returns the name of the `color`

In order to set the current color (not the foreground color!) that will be used for all drawing procedure, you must use

- `extern void WSetColor(WINDOW win,unsigned long color)`: sets the color that will be used when drawing functions are called (this is not the foreground color!)

6.3.4 The clipping rectangle

The current clipping rectangle corresponds to the (rectangle) zone all the drawings will be happening. You can set/get it using

- `void WSetClipRect(WINDOW win, int x, int y, int w, int h);`
- `void WGetClipRect(WINDOW *win, int *x, int *y, int *w, int *h);`

6.3.5 The main drawing functions

These functions use the current pen mode, the current pen size, the current line style and the current color.

- `void WDrawLine(GOBJECT o,LWFLOAT x,LWFLOAT y,LWFLOAT x1,LWFLOAT y1)`: draw a line in graphic object `o` between the points (local coordinates) `x,y` and `x1,y1`
- `void WDrawPoint(GOBJECT o,LWFLOAT x,LWFLOAT y)`: draw a point in graphic object `o` at (local coordinates) `x,y`
- `void WDrawRect(GOBJECT o,LWFLOAT x,LWFLOAT y,LWFLOAT dx,LWFLOAT dy,char flagSizeIsInPixel,RectType rectType)`: draw a rectangle in graphic object `o` between the points (local coordinates) `x,y` and `x+dx,y+dy`. If `flagSizeIsInPixel` is set then `dx` and `dy` are specified in pixels (not local coordinates). `rectType` corresponds to the type of rectangle that will be drawn (as explained in Section 6.1.1).
- `void WFillRect(GOBJECT o,LWFLOAT x,LWFLOAT y,LWFLOAT dx,LWFLOAT dy,char flagSizeIsInPixel,RectType rectType)` same as `WDrawRect` except that the rectangle is filled.
- `void WDrawCenteredRect(GOBJECT o,LWFLOAT x,LWFLOAT y,LWFLOAT r1,LWFLOAT r2,char flagSizeIsInPixel)`: same as `WDrawRect` except that the rectangle is centered at point `x,y` and has width `r1` and height `r2`.
- `void WFillCenteredRect(GOBJECT o,LWFLOAT x,LWFLOAT y,LWFLOAT r1,LWFLOAT r2,char flagSizeIsInPixel)` same as `WFillRect` except that the rectangle is centered at point `x,y` and has width `r1` and height `r2`.
- `void WClearRect(GOBJECT o,unsigned long color, LWFLOAT x,LWFLOAT y,LWFLOAT dx,LWFLOAT dy,char flagSizeIsInPixel,RectType rectType)`: same as `WDrawRect` except that the color `color` is used.
- `void WDrawEllipse(GOBJECT o,LWFLOAT x,LWFLOAT y,LWFLOAT dx,LWFLOAT dy,char flagSizeIsInPixel,RectType rectType)`: same as `WDrawRect` except that an ellipse is drawn instead of a rectangle.
- `void WFillEllipse(GOBJECT o,LWFLOAT x,LWFLOAT y,LWFLOAT dx, LWFLOAT dy, char flagSizeIsInPixel,RectType rectType)`: same as `WDrawEllipse` except that the ellipse is filled.

- `void WDrawCenteredEllipse(GOBJECT o,LWFLOAT x,LWFLOAT y,LWFLOAT r1,LWFLOAT r2,char flagSizeIsInPixel)`: same as `WDrawCenteredRect` except that an ellipse is drawn instead of a rectangle.
- `void WFillCenteredEllipse(GOBJECT o,LWFLOAT x,LWFLOAT y,LWFLOAT r1, LWFLOAT r2,char flagSizeIsInPixel)`: same as `WFillCenteredRect` except that an ellipse is drawn instead of a rectangle.
- `void WDrawCenteredCross(GOBJECT o,LWFLOAT x,LWFLOAT y,LWFLOAT r,char flagSizeIsInPixel` same as `WDrawCenteredRect` except that a cross is drawn (`r1` and `r2` are both equal to `r`)

6.3.6 Drawing images

To draw an image, you must follow the following points

- Allocate the current `pixmap` using `void WInitPixMap(int nRows, int nCols)`.
- Decide which colormap will be used (we will call it `cm`)
- Get the `size` of `cm` using `size = ColorMapSize(cm);`
- Loop on each point of the image and set the corresponding color using *void WSetPixelPixMap(int colNumber, int rowNumber, unsigned long color)* where `color` is equal to `n+cm` where `n` is an unsigned integer in the range `[0,size[` that specifies which color of the colormap `cm` must be used (the colors are indexed from the first one to the last one).
- At the end of the loop you must call `WDisplayPixMap(WINDOW win,int x,int y)` where `win` is the window the image will be displayed in and `x,y` the (window) coordinate it will be displayed at.

6.4 Adding a new graphic class using the C-Language

Ok, it's time to build a new graphic class! We will show how to do it on a specific example. We build a graphic class for display the `CIRCLES` value that was previously defined (see Section 5).

6.4.1 The creation of the graphic class

We must add a line in the load function of the package in order to define the new graphic class. We will call a function `DefineGraphCircles()` that we will have to write. Thus the load function becomes

```
static void LoadCirclesPackage(void)
{
  tCIRCLES = AddVariableTypeValue(circlesType, &tsCircles, NULL);
  tCIRCLES_ = tCIRCLES+1;
```

```
  DefineGraphCircles();

  AddCProcTable(&circlesTable);
}
```

The `DefineGraphicClass()` will create a new graphic class corresponding to the graphic objects called `GraphCircles`. The definition of this structure is

```
typedef struct graphCircles {

  GObjectFields;

  CIRCLES circles;  /* The CIRCLES to be displayed */

  unsigned long fillColor;  /* the color used to fill the circles */

} GraphCircles, *GRAPHCIRCLES;
```

(Let us note that you cannot use the `bgColor` field of the graphic object to store the color used for filling the circles since LastWave clears the object rectangle using the `bgColor` color).
To create the corresponding class, we need to use the `NewGClass()` function. This function returns a `GCLASS` which main fields (see Section 6.1.1) must be filled up. The final `DefineGraphicClass()` looks like

```
void DefineGraphCircles(void)
{
  theGraphCirclesClass = NewGClass("GraphCircles",theGObjectClass,"circles");
  theGraphCirclesClass->nbBytes = sizeof(GraphCircles);
  theGraphCirclesClass->init = _InitGraphCircles;
  theGraphCirclesClass->deleteContent = _DeleteContentGraphCircles;
  theGraphCirclesClass->set = _SetGraphCircles;
  theGraphCirclesClass->draw = _DrawGraphCircles;
  theGraphCirclesClass->isIn = _IsInGraphCircles;
  theGraphCirclesClass->varType = circlesType;
  theGraphCirclesClass->flags &= ~(GClassMoveResize+GClassLocalCoor);
  theGraphCirclesClass->info = "Graphic Class that allows to display circles";
}
```

The line

```
 theGraphCirclesClass->flags &= ~(GClassMoveResize+GClassLocalCoor);
```

ensures that the corresponding gobjects will not be allowed to be moved or resized (they will be displayed in a `VIEW`, just setting the bounding rectangle of the view will allow zooming) and that no local coordinates are associated to this object (the `VIEW` coordinates it is in will be used). We need to define the different methods.

6.4.2 The `init` function

```
/* Initialization of the GraphCircles structure */
static void _InitGraphCircles(GOBJECT o)
{
  GRAPHCIRCLES graph;

  graph = (GRAPHCIRCLES) o;

  graph->circles = NULL;

  graph->fillColor = graph->bgColor = invisibleColor;

  /* We increase the rectangle by 1 on each side to avoid clipping problems on the rectangle
graph->rectType.left = graph->rectType.right = graph->rectType.bottom = graph->rectType.top
}
```

6.4.3 The `deleteContent` function

```
static void _DeleteContentGraphCircles(GOBJECT o)
{
  GRAPHCIRCLES graph;

  graph = (GRAPHCIRCLES) o;

  if (graph->circles != NULL) DeleteCircles(graph->circles);
}
```

6.4.4 The `draw` function

```
static void _DrawGraphCircles (WINDOW win, GOBJECT obj, int x, int y,int w,int h)
{
  GRAPHCIRCLES graph;
  GOBJECT o1;
  CIRCLES c;
  LWFLOAT x0,y0,x1,y1;
  unsigned long fg,fill;
  int i;

  /* Some inits */
  graph = (GRAPHCIRCLES) obj;
  c = graph->circles;
  if (c == NULL) return;
  if (c->n == 0) return;

  fg = graph->fgColor;
  fill = graph->fillColor;
```

```
  o1 = (GOBJECT) obj->father;

  for (i=0;i<c->n;i++) {
    if (fill != invisibleColor) {
      WSetColor(win,fill);
      WFillEllipse(obj,c->array[i].x-c->array[i].r,c->array[i].y-c->array[i].r,2*c->array[i]
    }
    WSetColor(win,fg);
    WDrawEllipse(obj,c->array[i].x-c->array[i].r,c->array[i].y-c->array[i].r,2*c->array[i].
  }
}
```

Let us note that we could optimze this procedure by only drawing what intersects the rectangle `x,y,w,h`.

6.4.5 The `set` function

In order the `disp` command to be able to deal with circles, the graphic class must implement 2 fields:

- the `graph` field (read and write field), which must corresponds to the correspnding `CIRCLES` (should return a `VALUE`)
- the `rect` field (read only) must return a listv of the (local coordinates) rectangle which bounds the `CIRCLES` object. By default the `rect` field is a field of any graphic object. It returns the fields `rx,ry,rw,rh` (see Section 6.1.1). If we wanted to keep this default behavior, we would need to update the fields `rx,ry,rw,rh` as soon as the `CIRCLES` is changed. This is very heavy to write. A very simple way to avoid that (though it results in longer computations) is to recompute the bounding rectangle each time it is asked for. Thus we are going to overwrite the default `rect` field so that it computes the bounding rectangle, stores it in `rx,ry,rw,rh` and return it.

The functio to compute the bounding rect is

```
void _GetCirclesBound(CIRCLES c,LWFLOAT *x,LWFLOAT *y,LWFLOAT *w,LWFLOAT *h)
{
  int i;
  LWFLOAT f,x1,x2,y1,y2;

  if (c == NULL || c->n == 0) {
    *x = *y = *w = *h = 0;
    return;
  }

  x1 = FLT_MAX;
  x2 = FLT_MIN;
  y1 = FLT_MAX;
  y2 = FLT_MIN;
```

```
  for (i=0;i<c->n;i++) {
    f = c->array[i].x-c->array[i].r;
    x1 = MIN(x1,f);
    f = c->array[i].x+c->array[i].r;
    x2 = MAX(x2,f);

    f = c->array[i].y-c->array[i].r;
    y1 = MIN(y1,f);
    f = c->array[i].y+c->array[i].r;
    y2 = MAX(y2,f);
  }

  *x = x1;
  *y = y1;
  *w = x2-x1;
  *h = y2-y1;
}
```

and the `set` function is

```
static int _SetGraphCircles (GOBJECT o, char *field, char**argv)
{
  GRAPHCIRCLES graph;
  CIRCLES c;
  char *str;
  int i;
  LISTV lv;

   /* The help command */
  if (o == NULL) {
    SetResultStr("{{{graph [<circles>]} {Gets/Sets the circles object to be displayed by the
is equivalent to that field).}} \
{{fill [<color>]} {Sets/Gets the color that will be used to fill up the circles.}}}");
    return(YES);
  }

  graph = (GRAPHCIRCLES) o;
  c = graph->circles;

  /* the 'graph' and 'cgraph' fields */
  if (!strcmp(field,"graph") || !strcmp(field,"cgraph")) {
    if (*argv == NULL) {
      SetResultValue(c);
      return(YES);
    }
    argv = ParseArgv(argv,tCIRCLES,&c,0);
    if (c->n == 0) Errorf("_SetGraphCircles() : You cannot display an empty 'circles'");
```

```
    if (graph->circles != NULL) DeleteCircles(graph->circles);
    graph->circles = c;
    AddRefValue(c);
    _GetCirclesBound(c,&(o->rx),&(o->ry),&(o->rw),&(o->rh));
    UpdateGlobalRectGObject(o);
    return(YES);
  }

  /* The 'fill' field */
  if (!strcmp(field,"fill")) {
    if (*argv == NULL) {
      SetResultStr(GetColorName(graph->fillColor));
      return(YES);
    }
    argv = ParseArgv(argv,tCOLOR,&(graph->fillColor),0);
    return(YES);
  }

  /* The 'rect' field */
  if (!strcmp(field,"rect")) {
    NoMoreArgs(argv);
    _GetCirclesBound(c,&(o->rx),&(o->ry),&(o->rw),&(o->rh));
    UpdateGlobalRectGObject(o);

    lv = TNewListv();
    SetResultValue(lv);

    AppendFloat2Listv(lv,o->rx);
    AppendFloat2Listv(lv,o->ry);
    AppendFloat2Listv(lv,o->rw);
    AppendFloat2Listv(lv,o->rh);

    return(YES);
  }

  return(NO);
}
```

6.4.6 The `isIn` function

```
static LWFLOAT _IsInGraphCircles(GOBJECT o, GOBJECT *o1, int x, int y)
{
  LWFLOAT rx, ry;
  GRAPHCIRCLES g;
  int i;
  CIRCLES c;
```

```
  g = (GRAPHCIRCLES) o;
  c = g->circles;
  *o1 = NULL;

  /* Get the local coordinate */
  Global2Local(o,x,y,&rx,&ry);

  /* is this point in a circle ? */
  for (i=0;i<c->n;i++) {
    if ((c->array[i].x-rx)*(c->array[i].x-rx)+(c->array[i].y-ry)*(c->array[i].y-ry) < c->ar
      *o1 = o;
      return(0);
    }
  }
  return(-1);
}
```

6.5 Managing disp related scripts for the new graphic class

In order to load script files when the package `circles` is loaded you just need to create a directory `circles` in the script directory and create in it a file called `circles.pkg`. This file will be automatically executed when the package is loaded. In this file we suggest you should do the following

6.5.1 disp windows

When a window will be used to display `&circles`, the `disp` function looks for the variable `disp.circles.rect` which tells what the default position and size of the window should be. So in the file `circles.pkg` you can type

```
disp.circles.rect={20 55 330 330}
```

6.5.2 Managing the zoom

If you want to inherit from the zoom script system of LastWave when circles are displayed, it is extremely simple. You need to use the script command

```
SetZoomBindings gclass listv_of_modes
```

where `gclass` is the (non evaluated) graphic class the zoom should be performed on and the modes are to be chosen among the strings `"rect"` (a rectangle is specified with the mouse), `"xrect"` (same as "rect" but the width is contrained to the whole width), `"yrect"` (same as "rect" but the height is contrained to the whole height) and `"normal"` (same behavior as the default behavior for signals, i.e., using left/right/middle button). The `z` key will allow to change mode. Thus we can write in the file `circles.pkg`

```
SetZoomBindings GraphCircles \{'rect' 'xrect' 'yrect'\}
```

6.5.3 Managing the cursor

If you want to inherit from the cursor script system of LastWave when circles are displayed, it is extremely simple. You need to use the script command

```
SetCursorBindings gclass listv_of_commands
```

where `gclass` is the (non evaluated) graphic class the cursor should operate on and the commands are the commands that are called each time the mouse is moved and that should return the string that is displayed at the bottom of the window. A simple command that would return some information to be displayed at the bottom of the window could be.

```
setproc _CursorTextGraphCircles {}  {
  c = [setg @object -graph]
  sprintf id "%v" c
  return "$@objname ($id): $@x $@y"
}
```

The procedures that are passed in the listv of procedures of the `SetCursorBindings` command must have one (`&array`) argument named `cursor`. After drawing the cursor you need to specify in the variable `cursor.erase` the script that should be called to erase the cursor. To help you drawing cross-hair style cursors, you can use the `_ViewDrawCrossHair` script procedure. Thus a simple cross hair procedure will look like

```
setproc _DrawCursorGraphCircles {cursor}  {
  _ViewDrawCrossHair cursor.view @x @y
  cursor.erase = %%‘_ViewDrawCrossHair ’$cursor.view’ $@x $@y‘
  return [_CursorTextGraphCircles]
}
```

A procedure that would draw nothing would be

```
setproc _DrawCursorNoneGraphCircles {cursor} {
  cursor.erase=null
  return [_CursorTextGraphCircles]
}
```

and a procedure that would draw a cross-hair cursor centered at the center of the closest circle would be

```
setproc _DrawCursor1GraphCircles {cursor}  {
  c = [setg @object -graph]
  i = [circles closest c @x @y]
  x = c.x[i]
  y = c.y[i]
  _ViewDrawCrossHair cursor.view x y
  cursor.erase = %%‘_ViewDrawCrossHair ’$cursor.view’ $x $y‘
  return "$@objname : $x $y [index = $i] "
}
```

where the `circles closest` command is defined by

```
void C_Circles(char **argv)
{
  CIRCLES c;
  char *action;
  LWFLOAT x,y,dist,d;
  int i,i0;

  argv = ParseArgv(argv,tWORD,&action,tCIRCLES,&c,-1);

  if (!strcmp(action,"closest")) {
    argv = ParseArgv(argv,tFLOAT,&x, tFLOAT,&y,0);

    dist = FLT_MAX;
    i0 = -1;
    for (i=0;i<c->n;i++) {
      d = (c->array[i].x-x)*(c->array[i].x-x)+(c->array[i].y-y)*(c->array[i].y-y);
      if (d<dist) {
        i0 = i;
        dist = d;
      }
    }
    SetResultInt(i0);
    return;
  }

  Errorf("Unknown action '%s'",action);
}

static CProc circlesCommands[] = {

  "circles",C_Circles,"{{{closest <circles> <x> <y>} \
{Gets the index of the closes circle}}}",
  NULL,NULL,NULL
};

static CProcTable  circlesTable = {circlesCommands, "circles", "Commands related to circles
```

Thus finally, you should put in the `circles.pkg` file

```
SetCursorBindings GraphCircles {%_DrawCursorNoneGraphCircles %_DrawCursorGraphCircles %_Dra
```

6.5.4 Let's play around with circles!

This is an example of what you can do with the `circles` package

```
> package load 'circles'
```

```
> c = [new &circles]
= <&circles[0];0x090fe300>
> c.n=10
= 10
> c
= <&circles[10];0x090fe300>
> c.r = abs(Grand(10))
= <size=10;0.287257,0.0415209,1.28227,0.0587326,0.189727,1.18633,...>
> c.x = 0:9
= <size=10;0,1,2,3,4,5,...>
> c.y = 2*c.x
= <size=10;0,2,4,6,8,10,...>
> disp c -..1 -fill 'blue'
```

Play around with the mouse, zooming... Try to hit the `c` key for changing cursor mode and `z` for changinng zoom mode. You can display signals on top of circles using the `disp` command... and many more!

Chapter 7

Appendices

7.1 Appendix A: The random generator in LastWave

You should know about three C-functions that deal with the random generator:

- `LWFLOAT Grand(LWFLOAT sigma)`: returns a random number using a gaussian law with standard deviation `sigma`
- `LWFLOAT Urand(void)`: returns a random number using a uniform law on [0,1]
- `void RandInit(long int idum1)`: initializes the random generator using the seed `idum1`.