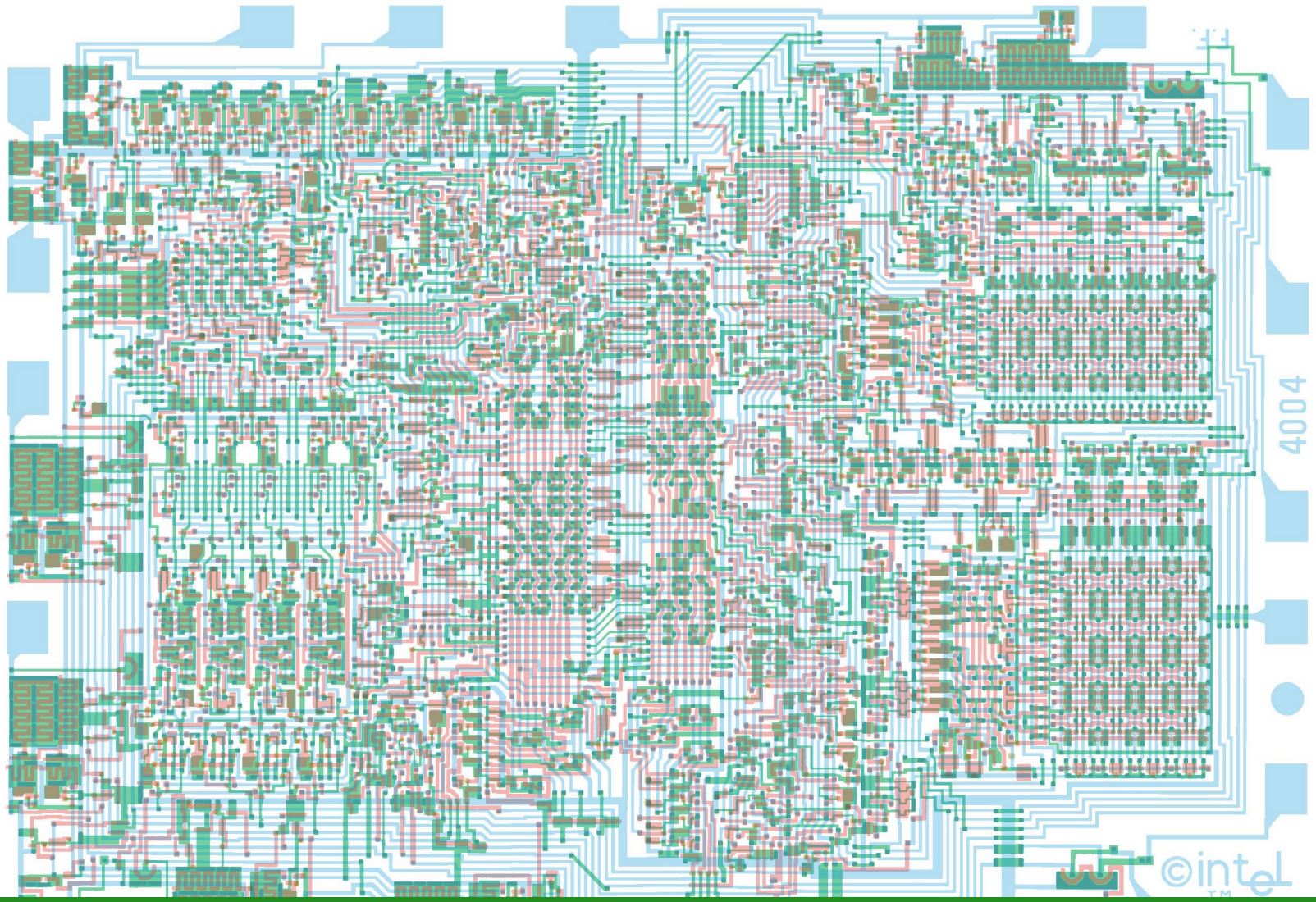
Technical and user documentation

LibOpenCIF Complete manual

Version 1.2.0



Contents

1 Introduction **1**
- 1.1 Who must read this document? . . . 2
- 1.2 What is a CIF file? . . . 2
- 1.3 What is LibOpenCIF? . . . 2
- 1.4 Why create LibOpenCIF? . . . 3
- 1.5 What can and can't do LibOpenCIF? . . . 3
- 1.6 Why use LibOpenCIF? . . . 4
- 1.7 The library can be used in other programing languages? . . . 4
- 1.8 About this document . . . 4
- 1.9 About the author . . . 6

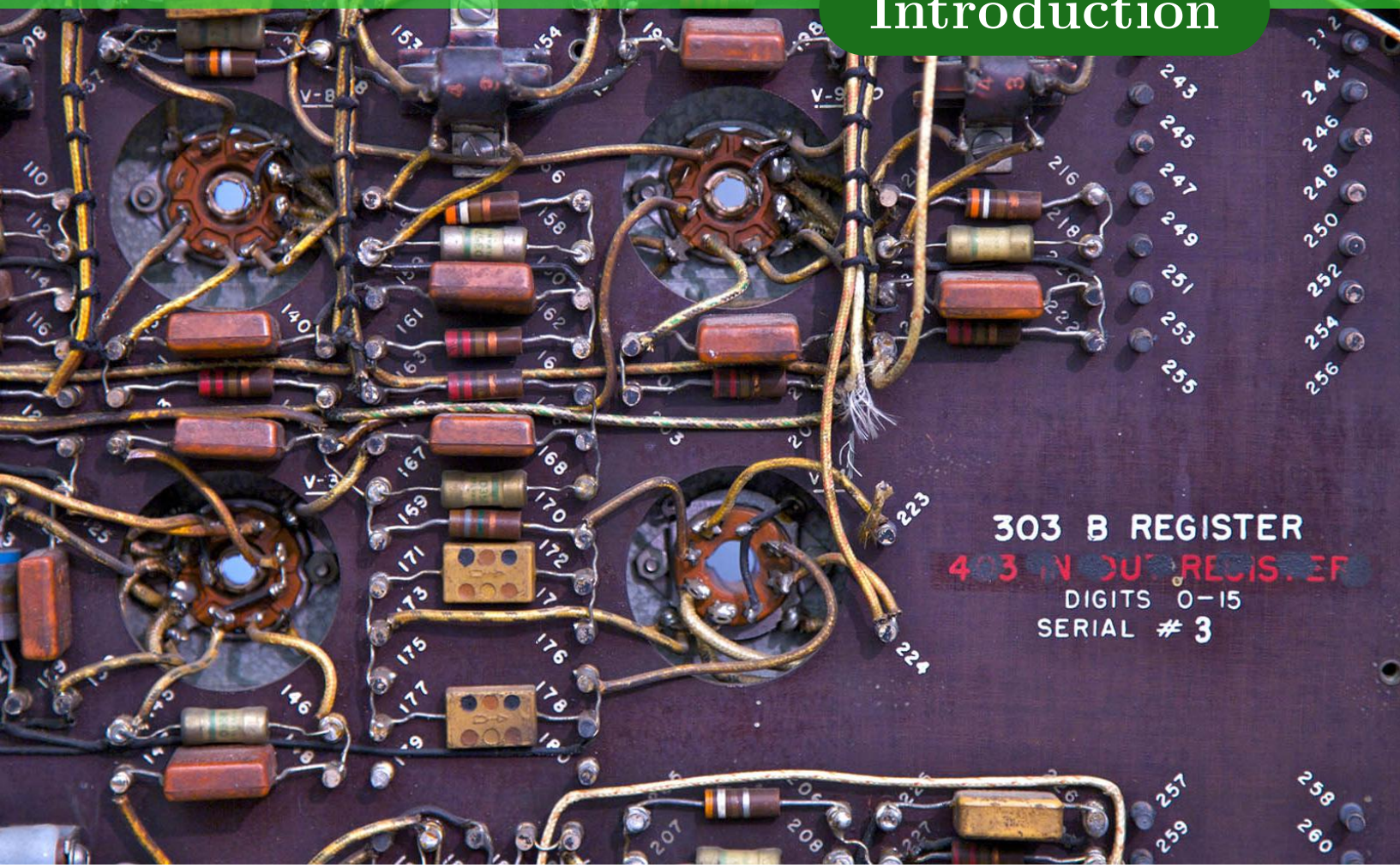
2 User documentation **7**
- 2.1 Installing the library . . . 8
- 2.2 Uninstalling the library . . . 10
- 2.3 Compiling a program with the library . . . 11
  - 2.3.1 Microsoft Windows . . . 12
  - 2.3.2 GNU/Linux . . . 13
    - Using the installed library . . . 13
    - Using the two-file version of the library . . . 15
  - 2.3.3 Mac OS X . . . 16
- 2.4 The loading of a file . . . 16
  - 2.4.1 Automatic loading . . . 16
  - 2.4.2 Manual loading . . . 21
- 2.5 Using the strings loaded . . . 25
- 2.6 Using the class instances created . . . 32
- 2.7 Extra components accesible by the user . . . 35
  - 2.7.1 Information about the library . . . 35
  - 2.7.2 How to validate and clean a single string . . . 36

3 Technical documentation **37**
- 3.1 The CIF format . . . 38
  - 3.1.1 Design coordinates . . . 38
  - 3.1.2 Theorical and real precision of values . . . 39
  - 3.1.3 Primitive Commands . . . 40
  - 3.1.4 Control Commands . . . 43
  - 3.1.5 RAW content commands . . . 45

3.1.6 Layer command . . . 45
3.1.7 BNF representation of the format . . . 45
3.2 Validation of a CIF file . . . 47
3.2.1 Finite state machine used . . . 47
3.2.2 Considerations . . . 48
3.3 Loading of a CIF file . . . 49
3.3.1 Cleaning process of commands . . . 49
Numeric commands . . . 49
Layer commands . . . 50
Call commands . . . 51
Definition control commands . . . 52
Expansion commands . . . 54
3.4 Converting strings into class instances . . . 54
3.4.1 Validation group of classes . . . 54
Class: State . . . 54
Class: FiniteStateMachine . . . 56
Class: CIFFSM . . . 57
3.4.2 Command group of classes . . . 59
Class: Point . . . 59
Class: Size . . . 60
Class: Fraction . . . 62
Class: Transformation . . . 63
Class: Command . . . 65
Class: PrimitiveCommand . . . 65
Class: PathBasedCommand . . . 65
Class: PolygonCommand . . . 65
Class: WireCommand . . . 65
Class: PositionBasedCommand . . . 65
Class: BoxCommand . . . 66
Class: RoundFlashCommand . . . 66
Class: ControlCommand . . . 66
Class: DefinitionStartCommand . . . 66
Class: DefinitionDeleteCommand . . . 66
Class: CallCommand . . . 66
Class: DefinitionEndCommand . . . 66
Class: EndCommand . . . 66
Class: RawContentCommand . . . 66
Class: UserExtentionCommand . . . 66
Class: CommentCommand . . . 66
Class: LayerCommand . . . 66
3.4.3 Loading group of classes . . . 66
Class: File . . . 67

Chapter 1

Introduction



In this chapter, the user will learn the background needed to understand crucial concepts related to the usage of the library LibOpenCIF. Some of these concepts are basic definitions of what is a CIF file and what is a library (in the context of the intended usage).

Also, there is recommended that the user takes the time to read this chapter, since here is explained how the contents are organized and how the information is provided (for example, is explained how to identify a command and how to read the example code).

1.1 Who must read this document?

This document is intended to be read by any user that needs to read a CIF file and wants or need to understand how to use the LibOpenCIF library. This document makes big assumptions about the user. One of the most important is that the user must be already aware of, and is familiarized with, the concepts related to integrated circuit designs, what are they and what is a layout.

Also, is important to note that this document is intended to be read by a user that already has knowledge about how to program in the C++ language, since here will not be explained, for example, what is a class, a variable, a class instance, and other core concepts of the Object Oriented Programming. This is the reason that this document might be easier to read for a programmer rather than a specialist in circuit design.

Finally, in this document we assume that the user is already familiarized with the use of the operative system on which is desired the use of the library. Among other operations, the user must be already capable of opening a terminal emulator, run commands on it, move itself between folders using commands and manipulate files using a file viewer (like Explorer on Microsoft Windows systems, Dolphin or Nautilus on GNU/Linux systems or Finder on Apple Mac OS X systems).

1.2 What is a CIF file?

The first concept that is needed to understand this document is the definition of what a CIF file is. This definition is critical to comprehend the whole idea behind the library and its functionality.

The **CIF** acronym comes from the name **Caltech Intermediate Form**, and it defines a very specific format used to represent and store the *layout* of an **integrated circuit design**. In other words, a CIF file describes *how a circuit design looks*. Such description is achieved using two-dimensional figures, like rectangles, circles and polygons.

All of the information related to the circuit is stored in a ***plain text file*** using ***commands***. The commands can be *instructions*, *figures* or *expansion commands*.

The instruction commands are used to control how the circuit is being defined. For example, they are used to instruct the program or device reading the file where a definition of a *cell* starts and ends, and when is being used. A figure command can define a visual component of the design, like a rectangle, its position, size and rotation. And finally, the expansion commands are special commands that the CIF format allows to be user-defined. This means that the reader of the file can interpret the command in a specific way, expanding the capabilities of the format, allowing the user to define things that where not originally intended to be supported, like cell names, node definitions, among others.

In the industry, various file formats exists that are intended to do the same thing: store integrated circuit designs. Along with the CIF format, there exists other popular formats to store designs. Of course, there are differences between formats, and every one has its advantages and disadvantages. Some files store the information in binary format (making the information very complex to read by a human, but easier to manage by a machine), others, like the CIF format, store the information in plain text (making the information easier to read by humans and making the format more flexible about its contents, but the information becomes more complex to read and manage by a machine).

1.3 What is LibOpenCIF?

LibOpenCIF is a C++ library intended to help the users with the task of reading and validating the contents of a CIF file into memory. The library is intended to be used in various operating systems, like GNU/Linux, Microsoft Windows and Apple Mac OS X.

For GNU/Linux and Mac OS X systems, the library can be compiled and installed into the system. After that, will be available as a static library. This means that any program compiled with it will be able to run even if the library is uninstalled from the system. This is also due to the fact that the library is very small.

For Microsoft Windows (and, in general, for all the systems if needed) the library has a two-file presentation. This means that, instead of compiling the library into a DLL file, it can be added directly to the file structure of programs as a regular piece of code (a source and header files are included). This means that all kinds of projects in C++ can use the library. This presentation is intended to help Microsoft Windows users, and also, if required, be of use for other systems.

1.4 Why create LibOpenCIF?

The original author of the library, Moises Chavez-Martinez, worked in various projects related to the process of design of integrated circuits. In all of those projects, the files used to store the designs were in CIF format. Working on applications that were made from the ground up, the lack of open utilities to read the CIF files became a reality, making really hard the process of reading the contents of such files, especially the process of validating the commands of the files (more of this in later sections).

The author decided to create a very specific library in C++ that can help him and other people writing and upgrading its own applications. The author decided that his work might be of help for others, and so, released his work as Open Source.

1.5 What can and can't do LibOpenCIF?

The LibOpenCIF library is intended to be able to:

- Open a desired CIF file.
- Try to load its contents, validating them using a specially designed Finite State Machine.
- Clean and format the contents of the file loaded into easy-to-process strings.
- Convert the commands loaded (in string form) into instances of special classes provided.
- Provide a manual or full-automatic loading process.
- Be partially relaxed or completely strict when validating the contents of the file.

Also, the library can provide to the user these next services, being them not directly intended to:

- If the input file has the expected format, the user can read its contents directly into memory using class instances.[1]
- The class instances can write their contents into a file, allowing the user to write CIF files using configured class instances.
- Is possible to validate commands (in string form).
- Is possible to clean commands (in string form).

[1] There is no automatic detection of which command is going to be loaded, that is a task left to the user. So, when using this capability, the user needs to know in advance which command is going to be the next, so he can use the correct class instance.

Finally, the library isn't intended to provide these services, since they are beyond the original scope of the design:

- The library will never convert values into floating point representations.[2] All of the values found in the input file will be stored and provided to the user in form of fractions.
- Validation of the design stored (looking for missing commands, logical errors, unused cells, cyclic calls, etc). The library is intended only to load and validate the commands, not the structure of the design. That is a task for which the user is responsible.

1.6 Why use LibOpenCIF?

There are various reasons why you can choose LibOpenCIF above other solutions to load a CIF file:

- The library is Open Source. You are not expected to pay any royalty for the usage of the library. Being released as GPL software, you just need to give credit to the original author.
- You can customize and extend the library. If the library can do more for you, and you know how to extend the library, then you can do it. Being released under the GPL license means that you can modify and extend the library as much as you want. You just need to give the proper credits to the original author.
- You will not need to implement this complex task by yourself. Most of the public projects that manage circuit designs in CIF format (Open Source or private) use their own routines to load the contents of the files. If you are starting a new project, you can have this library as a starting point.
- Easy to use. The library was designed to be as simple as possible.
- Portable. The library is programmed using pure standard C++. That means that you will not need external libraries to compile it, making the library platform independent.

1.7 The library can be used in other programing languages?

Is expected to be released a new library version for other major languages, since it has been ported to other languages, like Java, C and Python.

1.8 About this document

Is expected that the user reads all the contents of this document, and even in that case, the themes are ordered in incremental order. There are the contents expected to be covered in this document:

- Chapter 2 (User documentation): In this chapter, the user will learn basic aspects of the usage of the library, from how to install and uninstall it, to how to compile and use the capabilities provided in example programs. All of the themes covered are platform independent, except for the ones related to how to install, uninstall and compile (examples for every major operative system considered are provided).

[2] The values on an integrated circuit design, by definition, need floating point values for positions and sizes. Since those values are very important, and any error related to precision can affect negatively a design, the CIF format doesn't handle directly those values. In exchange, the format manages all of those values using ***fractions***, making the values error-less and leaving the conversion and precision in hands of the user.

- Chapter 3 (Technical documentation): In this chapter, the user will learn advanced topics related to the operation and design of the library. Among other themes, will be explained in detail how the library loads and validates the contents of the files (complete Finite State Machine included), how the commands are cleaned, the reasons behind the design decisions and a complete UML map of the classes and their relations.

In the document, when speaking about code, the user can expect to see sections like this one:

Listing 1.1: Example program

```
1  # include <iostream>
2
3  using std::cout;
4  using std::endl;
5
6  int main ()
7  {
8      // Single line comment
9      /*
10     Multi-line comment
11     */
12     cout << "This is a C++ example" << endl;
13     return ( 0 );
14 }
```

In the previous example, can be seen how the code will look once an example is provided. The code is in C++. The code examples are intended to be stored in plain text files. Those code files can have various filename extensions, depending if they are source or header files. Some of the supported file extensions for C++ source files are **.cc, .C,** and **.cpp**. All of them are valid (the second one has the problem of being easily confused with a C language source code file). In this document, we will be using the **.cc** file extension for source files, and the **.hh** file extension for the header files (instead of the popular **.hpp** used by some projects).

In this document, the user will also find system commands. Those system commands are intended to be entered and executed in terminal emulators. In GNU/Linux, Apple Mac OS X and similar systems, some examples of those terminal emulators are Konsole, Gnome Terminal, XTERM and Terminal. In Microsoft Windows systems, there are, at least, two options to use by the user: the Windows CMD or Windows PowerShell are the most common choices on terminals on these systems.

When needed, there will be shown to the user system commands. The next is an example of a command for GNU/Linux, Apple Mac OS X or similar systems:

```
$ cd /mnt/
$ ls
Disk1    Disk2    Disk3    Disk4
$
```

The previous example shows how a command can be executed and some of the output. These commands can be identified as intended as to be executed in GNU/Linux or similar systems because the commands start with a **$** character. Such character is commonly used in terminal emulators of these systems. The next command example is for Microsoft Windows systems:

```
> cd C:\Windows\System32\
>
```

The previous example shows how a command can be executed for Microsoft Windows systems. These commands can be identified as to be executed in such systems because the command starts with a > character. Such character is commonly used in terminals of these systems.

Finally, in this document, the user will find some special notes intended to help understanding some special concepts or critical information. The next boxes take care of various information types:

This is an information message
When reading this document, the user is recommended to pay attention to these messages, since these may contain non-critical information, like reminders about important paths or recommended usages about the library components.

This is a warning message
When reading this document, the user is expected to pay attention to these messages, since these may contain information about important details that might cause problems if ignored, like possible errors, common mistakes and more.

This is a critical or error message
When reading this document, the user is expected to pay special attention to these messages, since these may contain information about critical errors and mistakes that might affect in a very negative way your work.

1.9 About the author

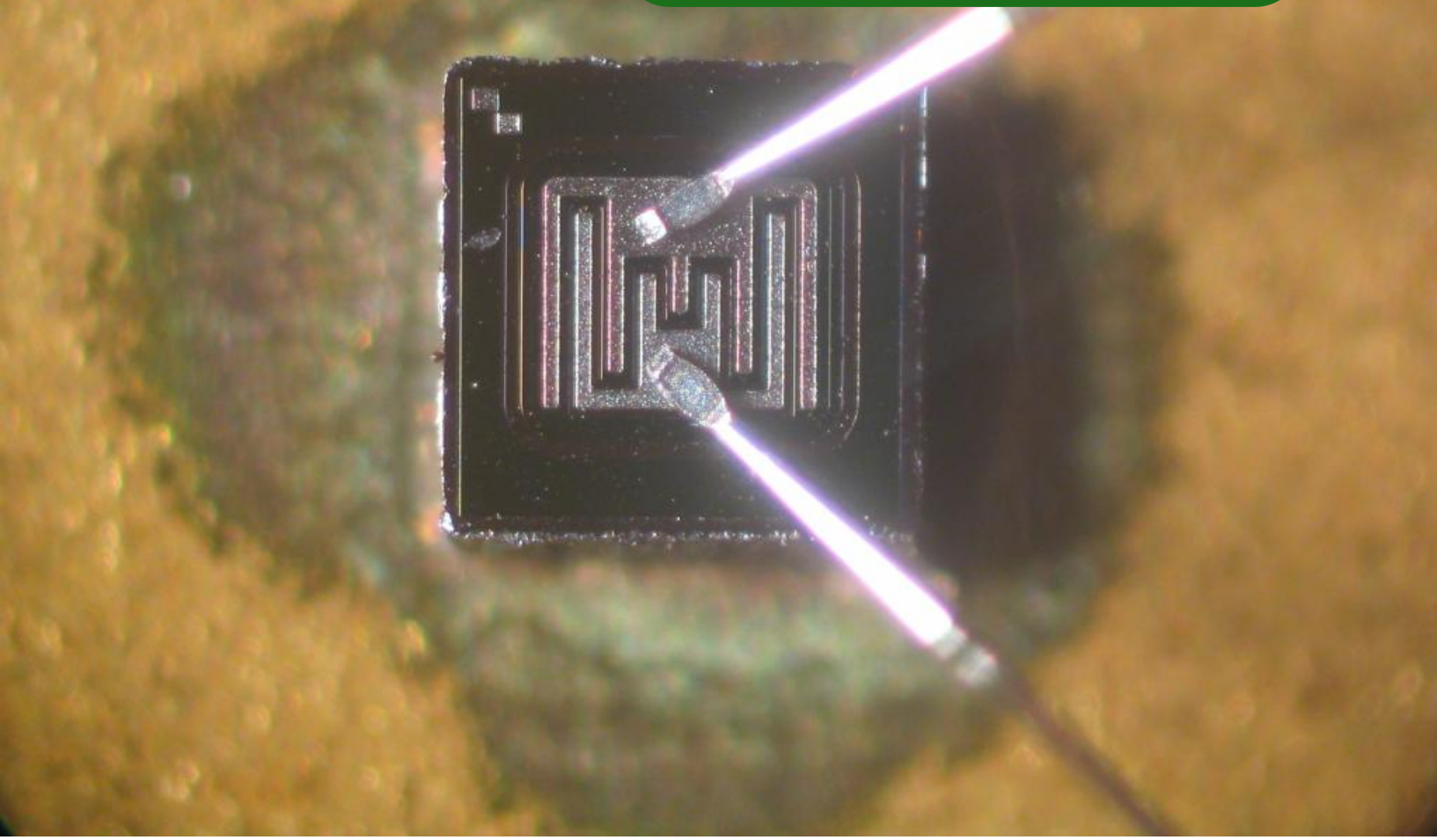
The author of this document is Moises Chavez-Martinez. He is a Computer Engineer with specialization on Systems Software from the Universidad de Guadalajara, in México. He is an enthusiast of the Open Source and likes to share his work.

During his career, he has worked in various professional applications aimed to help the task of designing Integrated Circuits. This granted him some knowledge of this work area, and shown him some deficiencies related to the work of applications for this industry area.

If you want to contact the autor to tell him something about the library, or to report a problem/bug, yo can use this e-mail: `moises.chavez.martinez@gmail.com`

Chapter 2

User documentation



In this chapter, the user will learn all the concepts and procedures needed to understand how to install, uninstall and use the library in various operating systems. Among other details, will be discussed:

- How to compile the source code in GNU/Linux and Mac OS X to create a static library.
- How to install the previously compiled static library.
- How to uninstall the installed library.
- How to compile a program using the library in all three major operative systems, in its two-file file mode and as a static library (mode available only on non-Windows systems).
- How to open a CIF file using the library.
- How to load the contents of a CIF file in a manually and fully automated way.
- How to use the information provided by the library once a file is loaded (a vector of strings or a vector of class instances).

2.1 Installing the library

The installation procedure is used only when a static library is needed, and such version is intended to be used only on the GNU/Linux and Apple Mac OS X systems. The library is not intended to be installed on Microsoft Windows systems, since in such systems the library must be used as a two-file part of the user projects (see section 2.3.1 to learn how to compile a project using the library).

These installation steps are intended to be used with the source version of the library. If the user already have a pre-compiled version (as static or dynamic library), follow its specific steps to install the library.

To being able to install the library, the user need to have some components before starting.

The first element is a package with the source code of the library. The needed file has a name like **`libopencif-X.Y.Z.tar.gz`**, where **`X`**, **`Y`** and **`Z`** are the major, minor and patch version numbers of the library. Is recommended to always look for the latest version of the library at SourceForge[1] or GitHub[2].

The next component needed is a compiler. The recommended compiler is **GCC v4.8.0** or newer (with its C++ module installed). The easiest way to know if the needed compiler is installed is opening a terminal emulator (like Konsole, Gnome Terminal or XTERM in GNU/Linux systems or Terminal in Apple Mac OS X systems) and executing this command:

```
$ g++
g++: fatal error: no input files
compilation complete.
$
```

As can be seen, the idea is to open a terminal emulator and run the command **`g++`** to see the output. If the compiler is installed, the user must see an output similar to the one in the previous example. If the user sees an output like this one:

```
$ g++
bash: g++: command not found
$
```

Then the GCC compiler is not installed. In such case, please, follow the specific instructions of the user system to install the program.

After the compiler is installed and ready, the next application needed is **CMake v2.6** or newer. In a similar way, to check if it is installed, the command **`cmake`** can be executed and its output checked. If the user see lots of text, then the application is installed (the output is a help message from the utility). But, if an output like this can be seen:

```
$ cmake
bash: cmake: command not found
$
```

Then the application is not installed. Again, please, follow the specific steps in the user system to install the utility.

After these steps, everything that is needed to compile and install the library is ready.

The first step is to extract the contents of the source package into a folder. To extract the contents of the package, run this command in the folder on which the package file is:

```
$ tar -zxvf libopencif-X.Y.Z.tar.gz
...
$
```

[1] https://sourceforge.net/projects/libopencif/
[2] https://github.com/Tuxman88/LibOpenCIF/

After running the command (replace the **X**, **Y** and **Z** characters with the numbers of the version downloaded), a new folder named **libopencif-X.Y.Z** will be created.

Enter to the new folder and create a new folder inside named **build**.

Now, enter such new folder and run this command inside:

```
$ cmake ..
```

After running the command, an output like this can be seen:

```
$ cmake ..
-- The C compiler identification is GNU 4.9.2
-- The CXX compiler identification is GNU 4.9.2
-- Check for working C compiler: /usr/lib64/ccache/cc
-- Check for working C compiler: /usr/lib64/ccache/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/lib64/ccache/c++
-- Check for working CXX compiler: /usr/lib64/ccache/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/user/libopencif-1.2.1/build
$
```

Compiling in 32 bit and 64 bit systems

As can be noticed, the command output in the example displays some references to **lib64**. Such references can be different if the user is compiling for a 32 bit system (these examples where captured in a 64 bit machine).

After running the command, run this new one in the same folder:

```
$ make
```

Now, the user will be able to see output like this one (the example is not complete):

```
$ make
Scanning dependencies of target opencif
[  3%] Building CXX object CMakeFiles/opencif.dir/src/command/command.cc.o
[  7%] Building CXX object CMakeFiles/opencif.dir/src/command/controlcommand/
    controlcommand.cc.o
...
[ 96%] Building CXX object CMakeFiles/opencif.dir/src/finitestatemachine/ciffsm.cc.o
[100%] Building CXX object CMakeFiles/opencif.dir/src/command/controlcommand/
    endcommand/endcommand.cc.o
Linking CXX static library libopencif.a
[100%] Built target opencif
$
```

Compilation errors

The library must be error-free for compilation (not even warning messages should appear). If any error is found, please contact with the author of the library (see section 1.9).

The user will now be able to install the library into its system. Run the following command and, when prompted, enter your user password to gain temporal root access:

```
$ sudo make install
```

After running the command, the user should see an output like this one:

```
$ sudo make install
Password:
[100%] Built target opencif
Install the project...
-- Install configuration: ""
-- Installing: /usr/local/lib/libopencif.a
-- Installing: /usr/local/include/opencif
$
```

Installation errors
If the user finds any installation error, read carefully the output of the command to check for trivial problems. If the errors can't be solved by the user, please, contact the author of the library (see section 1.9).

After the previous command, the library is expected to be correctly installed in the user system.

Files and folders installed
The installation is done by the CMake utility. It decides the paths where all the components will be installed. By default, it installs the static library in the path **`/usr/local/lib/`**, and the header file in the path **`/usr/local/include/`**.

Different paths
The installation paths might be different in the user system. When installing, is recommended to check and store the paths used by CMake to install the library for later use when uninstalling.

2.2 Uninstalling the library

The process of uninstalling the LibOpenCIF library is intended to be performed only by GNU/Linux and Apple Mac OS X users. This is due to the fact that the library is not installable on Microsoft Windows systems.

Using commands as root
Be careful when using commands as the root user, specially in this section since the procedures explained might require deleting files and folders recursively.

The uninstall process only requires removing the installed files of the system. Since this process requires root access, is recommended to do this operation using commands, or, if the user has file navigators with root access, can use them to ease the process.

The first element to remove from the system is the library file itself. By default, the file can be found in the folder **/usr/local/lib/**. The file name must be **libopencif.a**. Using a terminal emulator, the user can use these commands to go to the desired folder and remove the file:

```
$ cd /usr/local/lib
$ ls
libopencif.a
$ sudo rm libopencif.a
Password:
$
```

Using commands with sudo
Remember: When using commands with **sudo**, the password might be asked just the first time. After that, the current opened terminal is granted with a short period of time on which the password is no longer needed.

Deleting the library file
When removing the library file, remove it directly (using the full file name). Is not recommended to use wild-cards, since in the folder might be files with similar names.

As can be seen, the **ls** command was used to see the contents of the folder and see if the file is there.

After deleting the library file, the next and last component to remove is the header file. By default, the file can be found in the path **/usr/local/include/**. There can be found a file named **opencif**. To remove the file, use these commands:

```
$ cd /usr/local/include
$ ls
opencif
$ sudo rm opencif
Password:
$
```

After running the **rm** command as root, the system might ask you if you really want to remove the file. Answer yes (since it is no longer needed).

If the previous steps were performed correctly and none error was found, the library was completely and successfully uninstalled from your system.

2.3 Compiling a program with the library

In this section, the user will learn how to compile a program with LibOpenCIF support. Such procedure can vary depending on two factors:

- Operating system used.
- Library mode used (static pre-compiled or two-file version).

In the next sub-sections, the user will learn how to compile an example program in Microsoft Windows, GNU/Linux and Apple Mac OS X using the pre-compiled or two-file version of the library.

2.3.1 Microsoft Windows

For Microsoft Windows systems, the compilation can be done only in one way[3]: Using the two file version of the library.

To get access to the two-file version of the library, the user can download the regular source package of the library found in SourceForge or GitHub (see section 2.1 to find the URLs on which the package can be found). The downloaded file must be named **libopencif-X.Y.Z.tar.gz**, where **X**, **Y** and **Z** are the major, minor and patch version numbers of the package. Is recommended to always download the latest version of the library.

After getting the source package, uncompress it on your system (the user can use various applications to extract the contents of a **tar.gz** file on these systems, like 7-ZIP). Enter the new folder created after the uncompress process and look for the folder named **two-file**. Such folder must contain two files: **libopencif.cc** and **libopencif.hh**. Those two files are the only files needed for Microsoft Windows systems.

By default, those two files of the library requires to be together all the time, since the source file (the **.cc** file) has an **include** statement calling the header file (the **.hh** file). If you require to use a different structure on the files, like placing the source and header files in different folders, you can edit the source file and change the include path. Such instruction can be found right after the GNU license preamble in line 24, approximately.

From this point, we are assuming that the user has already a compilation mechanism in the intended system. In the case of the next examples, we use the MinGW[4] compiler for Microsoft Windows, being executed in a PowerShell window of Microsoft Windows 8.

For the next example, we created a temporal folder named **test1** in our **My Documents** path. Inside such folder, we copied the two files of the library and created a new file named **main.cc**. After the creation of the new file, the folder ended with these contents:

```
> dir


    Directory: C:\Users\User\Documents\Projects\test1


Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---      03/01/2015  08:57 p. m.      79037 libopencif.cc
-a---      03/01/2015  08:57 p. m.      31588 libopencif.hh
-a---      03/01/2015  09:57 p. m.        149 main.cc
>
```

Having all the needed files in place, the next step is to compile the library source file into object code. This is done using the next command:

```
> g++ -c .\libopencif.cc
>
```

The previous command will instruct GCC to compile the source file of the library into object code. After calling this command, a new file will appear, named **libopencif.o**. Then, open your **main.cc** file and add this example code:

[3]The library is not intended to be pre-compiled on Microsoft Windows systems. If the user finds a guide on how to pre-compile the library into, for example, a DLL file, then the user will be the only responsible of the success of such alternatives. The author will not provide support on non-official compilation methods.

[4]The installer can be found at **http://sourceforge.net/projects/mingw/files/Installer/**.

Listing 2.1: Include statement for Microsoft Windows the two-file version of the library.

```
1  # include <iostream>
2  # include "libopencif.hh"
3
4  using std::cout;
5  using std::endl;
6
7  int main ()
8  {
9     cout << "LibOpenCIF: Example about compilation." << endl;
10
11    return ( 0 );
12 }
```

As can be noticed, in line 2 can be found an include statement looking for the header file of the library. After adding the code and saving the changes, the user can compile the whole program using this command:

```
> g++ .\main.cc -o main.exe libopencif.o
>
```

The previous command will instruct GCC to compile the file **main.cc**, create the output binary file **main.exe** and add the object code found in the file **libopencif.o**.

After running the previous command, a new file can be found named **main.exe**. That file is the executable binary of our example program. To run the program in the terminal, use this command:

```
> .\main.exe
LibOpenCIF: Example about compilation.
>
```

If the output of the command is as expected, then, the compilation process was done correctly.

2.3.2 GNU/Linux

In GNU/Linux, the user has two options to include the library into his project. The first option is using the static library after installing it in the system. The second option is using the two-file version of the library.

Best usage method
The recommended usage method for the library is installing the library, and then, linking your program with the static library. However, the user is free to use the two-file version of the library. There is no major difference when using one option or another.

Using the installed library

To use the first option, the installed library, you first need to install the library. To do so, the user needs to follow the steps found in the section 2.1.

From this point, we are assuming that the user has already a compilation mechanism in the intended system. In the case of the next examples, we use the GCC compiler (with its C++ module) for GNU/Linux, available in most GNU/Linux distributions, being executed in a Konsole terminal emulator, running Bash on a Fedora 21 system.

For the next example, we created a temporal folder named **test1** in our **Home** folder. After that, we created a test file named **main.cc**, so that the folder, after the creation of the elements, ended with these contents:

```
$ pwd
/home/user/test1
$ ls
main.cc
$
```

Then , the user needs to add code to the example program. In this case, the example code is as follows:

Listing 2.2: Include statement for GNU/Linux with installed library

```
1  # include <iostream>
2  # include <opencif>
3
4  using std::cout;
5  using std::endl;
6
7  int main ()
8  {
9     cout << "LibOpenCIF: Example about compilation." << endl;
10
11    return ( 0 );
12 }
```

In the previous example, in line 2, the user can see the include statement for the library. Being installed, the library must be referenced as a regular header file.

Common incorrect include path
Pay special attention to the include path. The file name is **opencif**, not **libopencif**. This is a common error when adding the library to the source code.

The next step is to compile the code. To do so, in the terminal emulator, the user needs to run these commands:

```
$ gcc main.cc -o program.bin -lopencif
$ ls
main.cc
program.bin
$
```

As can be seen, the compilation process isn't complex. After the previous compilation command, the created file (a binary file) can be executed in terminal with the help of the next command:

```
$ ./program.bin
LibOpenCIF: Example about compilation.
$
```

As can be seen, the compilation process, using the installed version of the library, isn't complex nor difficult. The biggest problems on which the user can incur might be errors when typing names on the source code or the compilation commands.

Common incorrect library name
Pay special attention to the library name used in the compilation command. The name of the library is **opencif**, not **libopencif**. This is a common error when compiling with the library.

Right time to compile with the library
No matter if the user project has one or multiple files, is recommended to compile with the library only when linking the final binary file to prevent extra time compiling.

Using the two-file version of the library

In GNU/Linux, the second option that the user has to compile its program with the library, is using the two-file version of the library. First, the user needs to get the needed files of the library. To do so, follow the steps found in the section 2.3.1, right until the compilation process. Those steps will instruct the user about how to obtain the needed library files.

From this point, we are assuming that the user already has the needed two files of the library. Also, the user needs to already have a compilation mechanism ready on its system. In this case, for the next examples, we are using the GCC compiler, with its C++ module installed.

First, we created a test folder in our Home, named **test1**. In such folder, we copied the two library files (**libopencif.cc** and **libopencif.hh**) and created a new source file named **main.cc**. Using the next commands, the user can see the new contents found in the folder:

```
$ pwd
/home/user/test1/
$ ls
libopencif.cc
libopencif.hh
main.cc
$
```

When the files are ready, open the file **main.cc** and add the following code to it:

Listing 2.3: Include statement for GNU/Linux with two-file version of the library.

```
1  # include <iostream>
2  # include "libopencif.hh"
3
4  using std::cout;
5  using std::endl;
6
7  int main ()
8  {
9      cout << "LibOpenCIF: Example about compilation." << endl;
10
11     return ( 0 );
12 }
```

As can be seen, in the line 2 of the code, can be found an include statement pointing to the header file of the library. In this case, the header is a local file.

The compilation process is similar to the one described for Microsoft Windows. In a terminal emulator, the user needs to run the following commands:

```
$ gcc -c libopencif.cc
$ gcc main.cc -o program.bin libopencif.o
$
```

The first command compiles the source file of the library into object code (with `.o` file extension). The second command creates the final binary file, compiling the main source file and linking it with the object code file created previously.

To run the binary file, use the following command:

```
$ ./program.bin
LibOpenCIF: Example about compilation.
$
```

2.3.3 Mac OS X

Mac OS X: This is a test.

2.4 The loading of a file

After the user is able to compile a program using the library, the next needed step to learn is how to read the contents of a CIF file. The library provides to the user two ways to do the loading. The first one is the automatic procedure. The automatic operation tries to perform a series of operations, in order, to end with as much useful information as possible. The second one is a manual operation sequence. In this second mode, the user needs to perform the calls to the specific operations needed to open, load, validate and convert the information found within a CIF file.

The first option is useful if the user is confident about the operation of the library and if the expected result is really needed (more of this on later sections). The second option is useful if the user needs to keep control of the operation, step by step, and be able to perform some operation if, for example, an error occurs or if not all the steps are completed correctly.

For the next examples we are going to use a CIF file named **`inv_x1.cif`**. Such file is part of the examples found in Alliance VLSI 5.0, a set of CAD tools for Integrated Circuit designers, being such toolset Open Source.

The following examples will contain code that might only work for GNU/Linux, having the library installed in the system. The user must be aware of this. To make the examples work in other systems, the user needs to modify the include statement of the examples to the one needed for the target system. To know the difference between include statements in various systems, check the section 2.3.

2.4.1 Automatic loading

The library provides two loading methods: Automatic and manual. To learn how to use the manual method, please, refer to the next section. To learn how to use the automatic method, continue reading.

In the next example, the user can assume that we have a single **`main.cc`** file, located in our test folder **`test`**. Also, located in the same folder, the user can find our test CIF file **`inv_x1.cif`**.

The first step is to have a basic program that is ready to use the library. The following example already has the needed statement to include the library into the program.

Listing 2.4: Base program for the automatic loading.

```
1 # include <iostream>
2 # include <opencif>
3
4 using std::cout;
5 using std::endl;
6
7 int main ()
```

```
8  {
9      cout << "LibOpenCIF: Automatic loading example." << endl;
10
11     return ( 0 );
12 }
```

To read a CIF file, the library provides to the user a special class named **File**. Such class definition, and everything else provided by the library, can be found under a namespace named **OpenCIF**. The next step to load a CIF file is to create an instance of the class **File**. To create such class instance, we are using the following line of code:

```
1 OpenCIF::File cif_file;
```

Using dynamic memory
The user can use dynamic memory if needed. All the classes support the creation of instances as pointers. In our examples, we are using regular variables just for simplicity.

In this case, we are creating a variable named **cif_file**. Such class instance will help us to load the CIF file.

After creating the instance, the user needs to tell to the instance where is located the CIF file. To do so, the user must use a member function named **setPath**. Such member function expects a **string** class instance:

```
1 std::string file_path;
2 file_path = "inv_x1.cif";
3 cif_file.setPath ( file_path );
```

After doing this, the class instance has all the needed information to try to automatically load the file. To do so, the user can use the member function **loadFile**. Such member function will try to perform all the operations related to the load of the file, from opening the file to converting the loaded information into class instances. A very basic (but not recommended) way to call such member function is to just perform the call:

```
1 cif_file.loadFile ();
```

However, problems can be found when loading the file. The member function **loadFile** has a return value that can inform to the user of the result of the loading process. The returned value is a member of an enum structure, named **LoadStatus**. Such enum is declared inside the **File** class, and is accessible without an instance. By design, there are only four possible return values of the call to load the file. Those possible values are:

- **AllOk**: Such value is returned if there was found no error.
- **CantOpenInputFile**: Such value is returned if there was found an error while trying to open the CIF file. This problem might be related to an incorrect file path or a file without the needed permissions to be opened.
- **IncompleteInputFile**: Such value is returned if the contents of the CIF file are **correct** (no syntax errors, so all the commands are valid), but there was found no End command.[5]

[5] This is a common problem. Various CAD tools, when creating their output CIF files, they don't use an End command. Refer to the section 3.2 for more information related to this problem.

- **IncorrectInputFile**: Such value is returned if the loading process finds any command with an incorrect format.

The following code block shows the recommended way to catch and check the return value of the call to **loadFile**:

```
OpenCIF::File::LoadStatus status;

status = cif_file.loadFile ();

switch ( status )
{
   case OpenCIF::File::AllOk:
      std::cout << "LibOpenCIF: All ok when loading the file";
      break;

   case OpenCIF::File::CantOpenInputFile:
      std::cout << "LibOpenCIF: Can't open input file";
      break;

   case OpenCIF::File::IncompleteInputFile:
      std::cout << "LibOpenCIF: Incomplete input file (missing End command)";
      break;

   case OpenCIF::File::IncorrectInputFile:
      std::cout << "LibOpenCIF: Incorrect input file";
      break;
}
```

As can be seen, the idea is to first create a variable from the enum **LoadStatus**. Then, the user can call the member function **loadFile**, storing the returned valued in the previously created variable. Finally, the user can use a **switch** block to check all the possible return values from the call.

Using `if` conditions instead of a `switch`

When using the switch conditional structure, the user might be requested, by the compiler, to use all the possible values of the **LoadStatus** enum. If the user doesn't want to validate all the possible values, a series of nested **if** conditions can be used instead, or, if the **switch** is needed, an empty **default** condition can be used to skip the validation of the remaining values.

All the previously explained lines can be seen in the following working example.

Listing 2.5: Validation example of the return value when using the automatic loading.

```
# include <iostream>
# include <string>
# include <opencif>

using std::cout;
using std::endl;
using std::string;

int main ()
{
   cout << "LibOpenCIF: Automatic loading example." << endl;

```

```
13      OpenCIF::File cif_file;
14      string file_path;
15
16      file_path = "inv_x1.cif";
17      cif_file.setPath ( file_path );
18
19      OpenCIF::File::LoadStatus status;
20
21      status = cif_file.loadFile ();
22
23      switch ( status )
24      {
25          case OpenCIF::File::AllOk:
26              cout << "LibOpenCIF: All ok when loading the file";
27              break;
28
29          case OpenCIF::File::CantOpenInputFile:
30              cout << "LibOpenCIF: Can't open input file";
31              break;
32
33          case OpenCIF::File::IncompleteInputFile:
34              cout << "LibOpenCIF: Incomplete input file (missing End command)";
35              break;
36
37          case OpenCIF::File::IncorrectInputFile:
38              cout << "LibOpenCIF: Incorrect input file";
39              break;
40      }
41
42      return ( 0 );
43  }
```

As can be seen, in line 2, we added an include statement to add the **`string`** header (since the code needs to use the **`string`** class). In line 13 the path to the CIF file is set. In line 17 is called the **`loadFile`** member function (and its return value is stored), and finally, in lines 19 to 36, the switch structure takes care of all the possible values returned by the call to **`loadFile`**.

More steps needed

As the user can see, up to this point, the program is doing nothing useful but loading into memory the contents of the CIF file. In later sections will be discused how to use the elements loaded by the **`File`** class.

When loading the file, the member function **`loadFile`** is responsible for performing the needed calls to the four main steps of loading a CIF file. These steps are, in order:

- Opening the file. The first needed step to load a CIF file, is to open it in read-only mode. If this operation fails, the value **`CantOpenInputFile`** is returned.
- Syntax validation. The second operation is the loading of the file itself. Using a **Finite State Machine** designed to perform this task, the contents of the file are loaded, character by character, and validated by the aforementioned **Finite State Machine**. This step can face two problems. The first one is finding an incorrect command. In such case, the value **`IncorrectInputFile`** will be returned. The second problem can be detecting the lack of an End command, and in this case, the value **`IncompleteInputFile`** will be returned.

- Command cleaning. The third step on the loading chain is the command cleaning. If none error was found in the previous steps, all the commands found in the file are cleaned. This means that the commands are formatted to match a specific series of rules defined by the library author (more of this in the section 3.3.1). This step can't fail since all the loaded commands were validated by the previous step, so, this step can't generate an error.
- Command conversion. The fourth and last step on the loading chain is the command conversion. Here, all the commands loaded in the step two, that were already cleaned in the step three, are parsed and converted into class instances. Since the commands for this step are already validated and cleaned, this step can't generate an error.

If the loading process finds an error (like a syntax validation error), the whole loading chain will stop, so the remaining steps will not be performed. This is the default behaviour, but the user has the choice to omit some of the errors and instruct the member function to load as much as possible. To do so, the user can pass to the call to **loadFile** a special argument value that makes it skip as many errors as possible.

To instruct the call to **loadFile** to skip as many errors as possible, the user can use one of two possible values defined as enum constants within the **File** class:

- **StopOnError**: This value is, in fact, used by default when calling the **loadFile** member function. This constant instructs to the member function to stop if any error is detected.
- **ContinueOnError**: This value instructs to the member function **loadFile** to skip as many errors as possible.

To use one constant or the other, the user can use one of these lines:

```
1 cif_file.loadFile ( OpenCIF::File::StopOnError );
2 cif_file.loadFile ( OpenCIF::File::ContinueOnError );
```

The first line is effectively the same as calling the member function without passing any value. The second line, can be used to load as much as possible from the file. However, the user must be aware of the possible results of forcing the load of a file.

The first consideration is that the first operation, the opening of the file, can end with an error that is considered critical. If the file can't be opened, there is nothing that can be done to skip this problem.

After this, the operation of syntax validation can find invalid commands. To skip this errors, the library tries to omit characters from the file until a valid command is found. This, of course, can lead to problems related to the file being incomplete in memory (missing commands).

Forcing the load of a file

Even if possible, is recommended that the user doesn't use the error skipping capabilities of the library when using the automatic loading. Instead of that, is highly recommended to use the manual loading (see later sections).

Catching the result

Even if the library is loading as much information as possible (skipping errors), the call to **loadFile** will return a value describing the result of the operation. Is recommended to store such return value and do something according to it.

In later sections will be discussed how to use the information loaded by the **File** class.

2.4.2 Manual loading

The library provides two loading methods: Automatic and manual. To learn how to use the automatic method, please, refer to the previous section. To learn how to use the manual method, continue reading.

In the next example, the user can assume that we have a single **main.cc** file, located in our test folder **test**. Also, located in the same folder, the user can find our test CIF file **inv_x1.cif**.

The first step is to have a basic program that is ready to use the library. The following example already has the needed statement to include the library into the program.

Listing 2.6: Base program for the manual loading.

```
1  # include <iostream>
2  # include <opencif>
3
4  using std::cout;
5  using std::endl;
6
7  int main ()
8  {
9     cout << "LibOpenCIF: Manual loading example." << endl;
10
11    return ( 0 );
12 }
```

To read a CIF file, the library provides to the user a special class named **File**. Such class definition, and everything else provided by the library, can be found under a namespace named **OpenCIF**. The next step to load a CIF file is to create an instance of the class **File**. To create such class instance, we are using the following line of code:

```
1  OpenCIF::File cif_file;
```

Using dynamic memory
The user can use dynamic memory if needed. All the classes support the creation of instances as pointers. In our examples, we are using regular variables just for simplicity.

In this case, we are creating a variable named **cif_file**. Such class instance will help us to load the CIF file.

After creating the instance, the user needs to tell to the instance where is located the CIF file. To do so, the user must use a member function named **setPath**. Such member function expects a **string** class instance:

```
1  std::string file_path;
2  file_path = "inv_x1.cif";
3  cif_file.setPath ( file_path );
```

After doing this, the class instance has all the needed information to operate over the CIF file.

The first step to the manual loading of a CIF file using the library, is to open the file. To do so, the user must use the member function **openFile**:

```
1  cif_file.openFile ();
```

Such member function has a return value. The return value is a member of an enum structure, named **LoadStatus**. Such enum definition is defined within the **File** class. Is highly recommended, if not mandatory, to catch the return value and validate it. The possible return values of the **openFile** call are:

- **AllOk**: This value is returned if the file was successfully opened.
- **CantOpenInputFile**: This value is returned if the file can't be opened. Such error can be related to incorrect file paths or files without the needed permissions to be opened.

To catch the return value, is recommended to create a temporal variable of type **LoadStatus** as the following example:

```
1 OpenCIF::File::LoadStatus status;
2
3 status = cif_file.openFile ();
```

After the call, the user can use an **if** conditional structure to validate the result of the call:

```
1  status = cif_file.openFile ();
2
3  if ( status == OpenCIF::File::AllOk )
4  {
5     std::cout << "LibOpenCIF: All ok opening the CIF file." << std::endl;
6  }
7  else if ( status == OpenCIF::File::CantOpenInputFile )
8  {
9     std::cout << "LibOpenCIF: Can't open input file." << std::endl;
10 }
```

Opening the input file
This operation is critical. If the input file can't be opened, there is nothing that the library can do to continue. Is responsibility of the user to validate the result of the call and take the needed actions to try to solve the problem.

Once the file is opened, the following step to do is to load and validate the contents of the file. To do so, the **File** class offers to the user a member function named **validateSyntax**. Such member function, when called, will try to load character by character the contents of the file and validate them using a **Finite State Machine** designed for this task.

The job of the aforementioned member function is to load into memory the commands present in the file and to validate them.

Commands loaded, not cleaned
Up to this point, the commands loaded into memory are **valid**, but they have the same format as in the CIF file itself. This means that its not recommended to use them yet.

The call to the member function **validateSyntax** can return three different values. All of them, as with the call to **openFile**, are constant values defined as an enum structure under the **File** class and are accessible without an instance. The possible return values of this call are:

- **AllOk**: This value is returned if the call ends without problems and none error was found when validating the contents of the file.

- **`IncompleteInputFile`**: This value is returned if the contents of the CIF file (the commands) are **valid**, but no End command was found. If this value is returned, it doesn't mean that the file is incorrect nor incomplete[6].

- **`IncorrectInputFile`**: This value is returned if the **Finite State Machine** used to validate the contents of the CIF file detected an error (an unexpected character) in a command. If this value is returned, this means that the contents of the file might be incorrect or incomplete.

```
1  status = cif_file.validateSyntax ();
2
3  if ( status == OpenCIF::File::AllOk )
4  {
5     std::cout << "LibOpenCIF: CIF file valid." << std::endl;
6  }
7  else if ( status == OpenCIF::File::IncompleteInputFile )
8  {
9     std::cout << "LibOpenCIF: No End command found." << std::endl;
10 }
11 else if ( status == OpenCIF::File::IncorrectInputFile )
12 {
13    std::cout << "LibOpenCIF: CIF file contents are invalid." << std::endl;
14 }
```

No End commands
Even when the lack of an End command isn't an error by itself, the CIF format defines the need of such command to know when a file ends. The user has the responsibility of taking the needed actions if such event is present (like continuing or stopping the loading).

Incomplete loading
By default, if the loading process finds an error (in this case, an incorrect command), the loading and validation process of the file stops, so, if the error is present, the information loaded, even if still on memory, might represent only a fraction of the file itself.

As part of the loading and validation process, the **`File`** class stores in memory all the commands found and validated. Even if an error is found, the commands loaded remains on memory and can be used by the user if needed.

The next step on the loading process is the command cleaning. This step refers to the process of formatting the loaded commands with special rules defined by the author. The cleaning process removes any unnecessary characters from the commands and adds/removes whitespaces as needed. Refer to the section 3.3.1 for more information about the cleaning process.

To perform the cleaning step, the user must use the member function **`cleanCommands`**. Such member function will clean any command stored in memory. Since the commands loaded into memory are already validated, this operation isn't expected to generate any kind of error, so, the user will not need to take care of any return value:

```
1 cif_file.cleanCommands ();
```

[6]This is a common problem. Various CAD tools, when creating their output CIF files, they don't use an End command. Refer to the section 3.2 for more information related to this problem.

The final step on the loading process is the conversion of the commands loaded into class instances. To perform such operation, the user needs to use the member function **convertCommands**. This will create a new class instance for every command loaded while storing them in a new list (in the same order).

```
cif_file.convertCommands ();
```

The whole process of loading the file manually can be visualized in the following working example:

Listing 2.7: Validating return values when using the manual loading process.

```
# include <iostream>
# include <string>
# include <opencif>

using std::cout;
using std::endl;
using std::string;

int main ()
{
   cout << "LibOpenCIF: Manual loading example." << endl;

   OpenCIF::File cif_file;

   string file_path;
   file_path = "inv_x1.cif";

   cif_file.setPath ( file_path );

   OpenCIF::File::LoadStatus status;

   status = cif_file.openFile ();

   if ( status == OpenCIF::File::AllOk )
   {
      cout << "LibOpenCIF: All ok opening the CIF file." << endl;
   }
   else if ( status == OpenCIF::File::CantOpenInputFile )
   {
      cout << "LibOpenCIF: Can't open input file." << endl;

      return ( 1 );
   }

   status = cif_file.validateSyntax ();

   if ( status == OpenCIF::File::AllOk )
   {
      cout << "LibOpenCIF: CIF file valid." << endl;
   }
   else if ( status == OpenCIF::File::IncompleteInputFile )
   {
      cout << "LibOpenCIF: CIF file contents are valid, but no End command found."
         << endl;
   }
   else if ( status == OpenCIF::File::IncorrectInputFile )
   {
      cout << "LibOpenCIF: CIF file contents are invalid." << endl;

```

```
49          return ( 1 );
50      }
51
52      cif_file.cleanCommands ();
53      cout << "LibOpenCIF: CIF file commands cleaned." << endl;
54
55      cif_file.convertCommands ();
56      cout << "LibOpenCIF: CIF file commands converted." << endl;
57
58      return ( 0 );
59  }
```

After the previous steps, the whole CIF file is loaded into memory. In the following sections the user will learn how to use the information loaded in memory.

2.5 Using the strings loaded

When loading the contents of a CIF file using the library, the user can have access to the commands found within the file itself. If the user is not familiarized with the CIF format, please, read the contents of section 3.1. If the user isn't familiarized with the loading steps yet, please, read the contents of section 2.4.

After the loading steps (automatic or manual, complete or partial), the **`File`** class will contain, loaded into memory, all the commands found in the CIF file. The commands are stored in a **`vector`** class instance as **`string`** class instances.

The user can access the string commands using the member function **`getRawCommands`**. Such member function will return a **`vector`** class instance with template type **`string`**. The following line is the prototype of the member function:

```
1  std::vector< std::string > getRawCommands ( void ) const;
```

As can be seen, the member funtion returns a **copy** of the commands loaded into memory. This is done by the library to preserve the contents of the file and to be able to access them again if the user needs to[7]. Also, the member function is **constant**, making it a member function that can be used even if the user is using a constant **`File`** class instance.

Order of commands

The order of the commands found in the **`vector`** class instance is the same as in the CIF file itself. None kind of sorting is applied to the commands, so, the first element in the **`vector`** (index 0) is the first command found in the file, and the last element in the **`vector`** (index N-1) is the last command found in the file.

Never sort the commands

Unless the user has an advance knowledge of the CIF format, and has a very good reason to do it, is mandatory to **never** sort the commands of a CIF file in any way, since the contents of the file (the Integrated Circuit structure) will be lost.

[7] Also, the library has this behaviour to prevent changes on the command list done by the user. If the user needs to modify something on the commands, then a copy of the contents are created.

The commands found in the **vector** class instance can be in two states. The first state is **exactly** as they can be found in the CIF file. This state will be referred in this document as **raw commands**. The raw commands are valid, since they only are added to the **vector** if the **Finite State Machine** that validates the contents of the CIf file says they are valid. This is a guarantee to the user that the commands found in such **vector** are **correct**.

However, this doesn't mean that such raw commands are in the best shape to be used. According to the CIF format, the components of the commands (like values and identifiers) must have separation characters. Such characters can be almost anything that can't be confused with command characters. So, the following command example is completely valid according to the format:

```
1 Babcharacters1000,,1000this.is.an.example2000ofvalid.commands2000that.are.ugly;
```

Obviously, a CAD application or any software that creates CIF files, isn't expected to create commands like this one (with "trash" as separation characters). However, this doesn't prevent that the CAD applications use a wide variety of characters to separate components of the commands, like commas, or to leave values togheter when possible.

The second possible state of the commands are the **clean commands**. Such commands are the same as the raw commands, but formatted following rules defined by the author (see section 3.3.1).

Raw VS Clean

The usage of one state or the other depends entirely on the application that the user is designing. The author recommends the usage of the clean commands, since they are expected to be more easy to work with.

To get access to the raw commands, the user must use the manual loading process (see section 2.4.2). The user is not expected to get access to the raw commands if the automatic loading process is used.

Using the manual loading process, the user can have access to the raw commands right after the second step of the loading (on which the contents of the file are validated). After the call to **validateSyntax**, the commands found are already stored in memory and are accessible through the call to **getRawCommands**.

Is important to mention that, if the user calls the member function **cleanCommands**, the raw commands will be **replaced** in memory with the clean ones. So, if the raw commands are needed by the user, a call to **getRawCommands** must be performed before calling **cleanCommands** (which is highly recommended to perform before calling **convertCommands**).

Calling cleanCommands before convertCommands

Don't forget to call **cleanCommands** before calling **convertCommands**. Please, remember that the later operation depends on the correct formatting of the commands to convert into class instances.

The following working example shows how to load a CIF file and have access to both versions of the commands, raw and clean. In the example, the program is just printing to the standard output the commands, side by side, to see the differences.

Listing 2.8: Accessing both command lists, raw and clean.

```
1 # include <iostream>
2 # include <string>
3 # include <vector>
4 # include <opencif>
5
```

```
6  using std::cout;
7  using std::endl;
8  using std::string;
9  using std::vector;
10
11 int main ()
12 {
13    cout << "LibOpenCIF: How to access the raw commands." << endl;
14
15    OpenCIF::File cif_file;
16    string file_path;
17
18    // Prepare the CIf file
19    file_path = "inv_x1.cif";
20    cif_file.setPath ( file_path );
21
22    OpenCIF::File::LoadStatus status;
23
24    // Try to open it
25    status = cif_file.openFile ();
26
27    if ( status != OpenCIF::File::AllOk )
28    {
29       cout << "LibOpenCIF: Error opening file.";
30       return ( 1 );
31    }
32
33    // Validate the contents (this will load the
34    // raw commands into memory)
35    status = cif_file.validateSyntax ();
36
37    // I'm not validating if the file is incomplete, since
38    // such case isn't an error in this example.
39    if ( status == OpenCIF::File::IncorrectInputFile )
40    {
41       cout << "LibOpenCIF: Error validating file.";
42       return ( 1 );
43    }
44
45    // Prepare two vectors, one to store the raw commands
46    // and the other to store the clean commands
47    vector< string > raw_commands;
48    vector< string > clean_commands;
49
50    raw_commands = cif_file.getRawCommands ();
51
52    // Clean the commands, this will replace the commands that
53    // I just saved in the previous instruction
54    cif_file.cleanCommands ();
55
56    // Get the new commands
57    clean_commands = cif_file.getRawCommands ();
58
59    // I'm going to print the raw commands side by side with
60    // the clean commands. I'm not validating the size of both
61    // vectors since both must have the same amount of commands.
62    for ( int i = 0; i < raw_commands.size (); i++ )
63    {
64       cout << "RAW:   " << raw_commands[ i ] << endl;
65       cout << "CLEAN: " << clean_commands[ i ] << endl;
```

```
66      }
67
68      // I'm skiping the remaining steps on the loading chain because
69      // they are not needed for this example.
70
71      return ( 0 );
72  }
```

After compiling and running the previous example, the user will be able to see command examples like the following ones:

```
1  RAW:   DS1 50 2;
2  CLEAN: D S 1 50 2 ;
3  ...
4  RAW:   DF;
5  CLEAN: D F ;
```

As can be seen, part of the formatting done by the cleaning process is assigning whitespaces to separate the components of the commands. This is explained in detail in the section 3.3.1.

Using the raw commands
The usage of the raw commands is responsibility of the user. The author will not provide direct support about how to manage such commands.

To use the clean commands, however, the author recommends the usage of **input streams**, one powerful mechanism of C++. This option can take full advantage of the formatting done to the commands. A second approach can be using tokenizer algorithms to split the strings using the whitespaces as tokenizer character.

The following example uses the first suggestion, printing to the standard output the **Box** commands only, while converting, on the fly, it's values to integers.

Listing 2.9: Using C++ string streams with the clean commands

```
1  # include <iostream>
2  # include <string>
3  # include <vector>
4  # include <sstream>
5  # include <opencif>
6
7  using std::cout;
8  using std::endl;
9  using std::string;
10 using std::vector;
11 using std::istringstream;
12
13 void PrintBoxCommand ( string command );
14
15 int main ()
16 {
17     cout << "LibOpenCIF: An approach of using the clean commands with C++ string
       streams." << endl;
18
19     OpenCIF::File cif_file;
20     string file_path;
21
22     // Prepare the CIf file
```

```
23    file_path = "inv_x1.cif";
24    cif_file.setPath ( file_path );
25
26    OpenCIF::File::LoadStatus status;
27
28    // Try to load the whole CIF file
29    status = cif_file.loadFile ();
30
31    if ( status == OpenCIF::File::IncorrectInputFile )
32    {
33       cout << "LibOpenCIF: Error loading file.";
34       return ( 1 );
35    }
36
37    // Get the clean commands
38    vector< string > clean_commands;
39    clean_commands = cif_file.getRawCommands ();
40
41    for ( int i = 0; i < clean_commands.size (); i++ )
42    {
43       PrintBoxCommand ( clean_commands[ i ] );
44    }
45
46    return ( 0 );
47 }
48
49 void PrintBoxCommand ( string command )
50 {
51    istringstream input ( command );
52    string piece;
53
54    // Get the command type. If the command isn't a Box command, return.
55    input >> piece;
56
57    if ( piece != "B" )
58    {
59       return;
60    }
61
62    // Get the size and position values
63    long int size_width , size_height;
64    input >> size_width >> size_height;
65
66    long int position_x , position_y;
67    input >> position_x >> position_y;
68
69    // Create default rotation (neutral rotation)
70    long int rotation_x = 1 , rotation_y = 0;
71
72    // Validate the next component. If is a semicolon (;),
73    // the Box doesn't have rotation. If is not a semicolon, the value
74    // is the X component of the rotation vector.
75    input >> piece;
76
77    if ( piece != ";" )
78    {
79       // Create other input stream to convert the X component
80       istringstream component ( piece );
81       component >> rotation_x;
82       input >> rotation_y;
```

```
83      }
84
85      cout << "--- Box command ---" << endl;
86
87      cout << "Size:     " << size_width << "x" << size_height << endl;
88      cout << "Position: (" << position_x << "," << position_y << ")" << endl;
89      cout << "Rotation: (" << rotation_x << "," << rotation_y << ")" << endl;
90
91      return;
92  }
```

As can be seen, the process of converting the commands isn't complex when using the clean commands. The user will be able to see output like the following[8]:

```
1  ...
2  --- Box command ---
3  Size:     16x16
4  Position: (40,120)
5  Rotation: (1,0)
6  --- Box command ---
7  Size:     16x16
8  Position: (16,180)
9  Rotation: (1,0)
10 ...
```

The following working example shows how to use the clean commands with whitespaces as a tokenizer character. For such task, the program is also using string streams to split the strings. The output of the program is exactly the same, but the idea is to show two different approaches of using the same data to generate the same output.

Listing 2.10: Using a tokenizer approach with the clean commands

```
1  # include <iostream>
2  # include <string>
3  # include <vector>
4  # include <sstream>
5  # include <opencif>
6
7  using std::cout;
8  using std::endl;
9  using std::string;
10 using std::vector;
11 using std::istringstream;
12
13 void PrintBoxCommand ( string command );
14 vector< string> TokenizeCommand ( string command );
15 long int StringToValue ( string to_convert );
16
17 int main ()
18 {
19     cout << "LibOpenCIF: Another approach of using the clean commands with a basic
       tokenizer algorithm." << std::endl;
20
21     OpenCIF::File cif_file;
22     string file_path;
23
```

[8] The user can see that the example program is using the C++ integer type **long int** while the values shown can perfectly fit in a **int** or **short** types. This is due to the fact that the example CIF file is a really small design, while the big designs can require integer values even bigger than the ones used in this example.

```
24    // Prepare the CIf file
25    file_path = "inv_x1.cif";
26    cif_file.setPath ( file_path );
27
28    OpenCIF::File::LoadStatus status;
29
30    // Try to load the whole CIF file
31    status = cif_file.loadFile ();
32
33    if ( status == OpenCIF::File::IncorrectInputFile )
34    {
35       cout << "LibOpenCIF: Error loading file.";
36       return ( 1 );
37    }
38
39    // Get the clean commands
40    vector< string > clean_commands;
41    clean_commands = cif_file.getRawCommands ();
42
43    for ( int i = 0; i < clean_commands.size (); i++ )
44    {
45       PrintBoxCommand ( clean_commands[ i ] );
46    }
47
48    return ( 0 );
49 }
50
51 void PrintBoxCommand ( string command )
52 {
53    vector< string > command_tokens;
54    command_tokens = TokenizeCommand ( command );
55
56    // Get the command type. If the command isn't a Box command, return.
57    if ( command_tokens[ 0 ] != "B" )
58    {
59       return;
60    }
61
62    // Get the size and position values
63    long int size_width = StringToValue ( command_tokens[ 1 ] );
64    long int size_height = StringToValue ( command_tokens[ 2 ] );
65
66    long int position_x = StringToValue ( command_tokens[ 3 ] );
67    long int position_y = StringToValue ( command_tokens[ 4 ] );
68
69    // Create default rotation (neutral rotation)
70    long int rotation_x = 1 , rotation_y = 0;
71
72    // Validate the amount of tokens. If not enough, there is no
73    // rotation.
74    if ( command_tokens.size () > 6 ) // B W H X Y ;   <-- There are 6 elements
75    {
76       rotation_x = StringToValue ( command_tokens[ 5 ] );
77       rotation_y = StringToValue ( command_tokens[ 6 ] );
78    }
79
80    cout << "--- Box command ---" << endl;
81
82    cout << "Size:     " << size_width << "x" << size_height << endl;
83    cout << "Position: (" << position_x << "," << position_y << ")" << endl;
```

```
84      cout << "Rotation: (" << rotation_x << "," << rotation_y << ")" << endl;
85
86      return;
87  }
88
89  vector< string> TokenizeCommand ( string command )
90  {
91      istringstream tokenizer_stream ( command );
92      string token;
93      vector< string > tokens;
94
95      do
96      {
97          tokenizer_stream >> token;
98          tokens.push_back ( token );
99      }
100     while ( token != ";" );
101
102     return ( tokens );
103 }
104
105 long int StringToValue ( string to_convert )
106 {
107     istringstream converter ( to_convert );
108     long int value;
109     converter >> value;
110
111     return ( value );
112 }
```

As the user can see, the usage of the commands is possible and made easy thanks to the format applied to the strings. In the following section, the user will learn how to use the class instances created by the **File** class when loading the file.

2.6 Using the class instances created

The final step when loading a CIF file is the conversion of the string commands found and cleaned into class instances. These class instances represent the commands themselves, and they are intended to ease even more the process of accessing the values of the commands.

To access the class instances, the user must use the **getCommands** member function. The prototype of such member function is as follows:

```
1  std::vector< OpenCIF::Command* > getCommands ( void ) const;
```

As can be seen, the call will return an instance of a **vector** class instance. Such **vector** stores pointers to a class named **Command**, which is the base class to all the classes that represent commands.

Using such pointers and the polymorphism mechanisms that C++ allows, the user can validate the type of each command and cast it into a derivative class pointer and use it as needed.

So, the first step is to get a copy of the vector of pointers. Such task is done as in the next example:

```
1  std::vector< OpenCIF::Command* > commands;
2  commands = cif_file.getCommands ();
```

After getting the commands, the user can iterate over the pointers and validate the type of the commands stored. In this case, to do such task, the class **Command** has a polymorphic member function named **type**. Such member function will return a value of type **CommandType**, which is an enum structure defined whithin the **Command** class and is public.

The enum structure **CommandType** has various constants defined. The following is a list of the constants intended to be used by the user (the list is not complete, since there are other constants defined used only by the library). Every constant represents a unique command type that can be found within a CIF file:

- **Call**
- **DefinitionEnd**
- **Comment**
- **UserExtension**
- **Polygon**
- **Wire**
- **Box**
- **RoundFlash**
- **Layer**
- **End**

Each constant can be accessed in the same way. Please, refer to the following example:

```
1  OpenCIF::Command* command;
2  command = commands[ 0 ];
3
4  switch ( command->type () )
5  {
6     case OpenCIF::Command::Call:
7        std::cout << "Call command" << std::endl;
8        break;
9     case OpenCIF::Command::DefinitionEnd:
10       std::cout << "Definition End command" << std::endl;
11       break;
12    case OpenCIF::Command::Comment:
13       std::cout << "Comment command" << std::endl;
14       break;
15    case OpenCIF::Command::UserExtention:
16       std::cout << "User Extention command" << std::endl;
17       break;
18    case OpenCIF::Command::Polygon:
19       std::cout << "Polygon command" << std::endl;
20       break;
21    case OpenCIF::Command::Wire:
22       std::cout << "Wire command" << std::endl;
23       break;
24    case OpenCIF::Command::Box:
25       std::cout << "Box command" << std::endl;
26       break;
27    case OpenCIF::Command::RoundFlash:
28       std::cout << "Round Flash command" << std::endl;
29       break;
30    case OpenCIF::Command::Layer:
31       std::cout << "Layer command" << std::endl;
32       break;
```

```
33      case OpenCIF::Command::End:
34         std::cout << "End command" << std::endl;
35         break;
36      default:
37         std::cout << "Internal class (used only by the library)" << std::endl;
38         break;
39   }
```

As can be seen, a way to validate the type of the command is to use a switch conditional structure. In this case, we are validating only those command types that represent final commands (those found in the CIF file). We are using the default option to skip the remaining command types that are not intended to be used by the user.

The next step is being able to cast the **Command** class pointer into the correct class pointer. In this case, the casting can be done without the use of complex mechanisms. A simple explicit cast when copying the pointer is enough. The following code example shows to how to convert a **Command** base pointer into a **BoxCommand** derivative class pointer:

```
1  OpenCIF::Command* command_pointer;
2  command_pointer = commands[ 0 ];
3
4  OpenCIF::BoxCommand* box_pointer;
5  box_pointer = (OpenCIF::BoxCommand*)command_pointer;
```

The next working example shows how to perform the same task as the previous two working examples. In this case, the idea is to show how to use the last mechanism available to the user to be able to do the same task, but even easier.

Listing 2.11: Using the class instances to print all the Box commands

```
1  # include <iostream>
2  # include <string>
3  # include <vector>
4  # include <opencif>
5
6  using std::cout;
7  using std::endl;
8  using std::string;
9  using std::vector;
10
11 void PrintBoxCommand ( OpenCIF::Command* command );
12
13 int main ()
14 {
15    cout << "LibOpenCIF: Another approach of using the clean commands with a basic
          tokenizer algorithm." << std::endl;
16
17    OpenCIF::File cif_file;
18    string file_path;
19
20    // Prepare the CIf file
21    file_path = "inv_x1.cif";
22    cif_file.setPath ( file_path );
23
24    OpenCIF::File::LoadStatus status;
25
26    // Try to load the whole CIF file
27    status = cif_file.loadFile ();
28
29    if ( status == OpenCIF::File::IncorrectInputFile )
```

```
30      {
31          cout << "LibOpenCIF: Error loading file.";
32          return ( 1 );
33      }
34
35      // Get the class instances
36      vector< OpenCIF::Command* > command_instances;
37      command_instances = cif_file.getCommands ();
38
39      for ( int i = 0; i < command_instances.size (); i++ )
40      {
41          PrintBoxCommand ( command_instances[ i ] );
42      }
43
44      return ( 0 );
45  }
46
47  void PrintBoxCommand ( OpenCIF::Command* command )
48  {
49      // Get the command type. If the command isn't a Box command, return.
50      if ( command->type () != OpenCIF::Command::Box )
51      {
52          return;
53      }
54
55      // Cast command
56      OpenCIF::BoxCommand* box;
57      box = (OpenCIF::BoxCommand*)command;
58
59      cout << "--- Box command ---" << endl;
60
61      cout << "Size:     " << box->getSize ().getWidth () << "x"
62                          << box->getSize ().getHeight () << endl;
63
64      cout << "Position: (" << box->getPosition ().getX () << ","
65                           << box->getPosition ().getY () << ")" << endl;
66
67      cout << "Rotation: (" << box->getRotation ().getX () << ","
68                           << box->getRotation ().getY () << ")" << endl;
69
70      return;
71  }
```

Using the class instances, the usage of the information found within a CIF file becomes easier. To learn more about the classes provided in the library, please, refer to the section 3.4.

2.7 Extra components accesible by the user

2.7.1 Information about the library

The LibOpenCIF library provides to the user some special fields defined under the **`OpenCIF`** namespace. These fields are string constants intended to help the user to identify important information about the library itself. These fields are:

- **`LibraryVersion`**: This field of constant type **`string`**, holds the complete version of the library. As an example, the string can store the version as ”1.2.0”.

- `LibraryVersionMajor`: This field of constant type `string`, holds the **major** number of the version of the library. If the library version is "1.2.0", this field will store the value "1".
- `LibraryVersionMinor`: This field of constant type `string`, holds the **minor** number of the version of the library. If the library version is "1.2.0", this field will store the value "2".
- `LibraryVersionPatch`: This field of constant type `string`, holds the **patch** number of the version of the library. If the library version is "1.2.0", this field will store the value "0".
- `LibraryName`: This field of constant type `string`, holds the **official name** of the library. In this case, the string will be "LibOpenCIF".
- `LibraryAuthor`: This field of constant type `string`, holds the **author name** of the library. The expected value of such string is "Moises Chavez-Martinez".
- `LibrarySupport`: This field of constant type `string`, holds the **support e-mail** intended to the used to contact the author about the library.
- `LibraryCIFVersion`: This field of constant type `string`, holds the **CIF format version** used to design the library. The expected value is "2.0".

All these fields are intended to provide extra information to the user when using the library, and to be able to identify the library if needed.

2.7.2 How to validate and clean a single string

The `File` class provides a public and static member function intended to be used by the user to validate, on demand, a single string. The validation performed can tell to the user if such string represents a valid CIF command.

To do such operation, the user can use the static member functions `isCommandValid` and, after validating the command, `cleanCommand` to validate and clean a string command, respectively.

The first member function, `isCommandValid`, takes as argument a single `string` class instance. This member function will apply the Finite State Machine found within the library and will return a bool value. A `true` value means that the string represent a valid CIF command. A `false` value means that the string doesn't represent a valid CIF command.

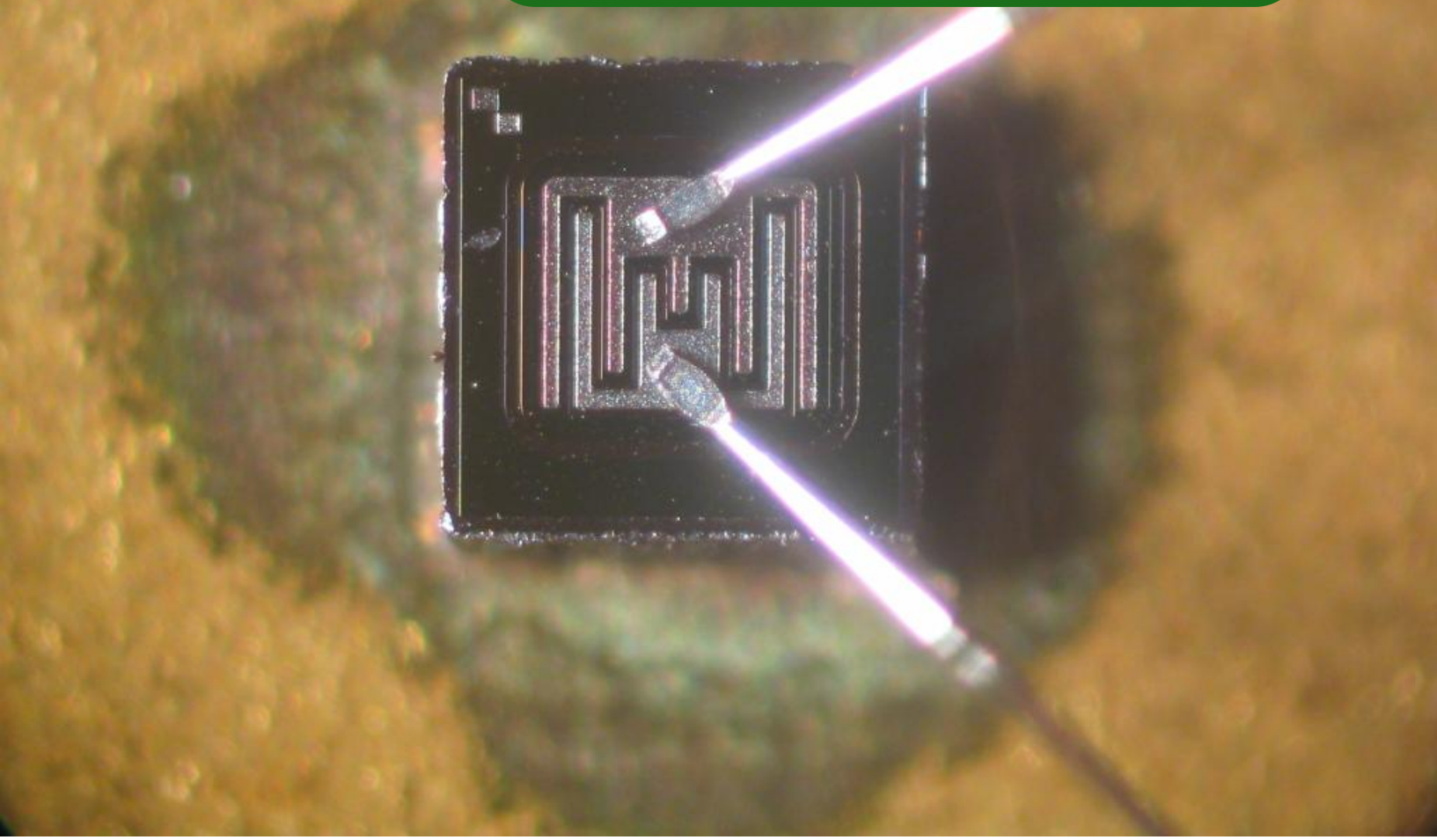
Once a command is validated, the user can use the second member function to clean the string. The member function `cleanCommand` takes as argument a previously validated string and performs all the cleaning operations needed.

Not validating the commands
Is never recommended to try to clean a command that hasn't been validated previously. Even if the user has manually validated the command, the rules applied to the strings can differ.

Cleaning non-validated commands
If the user calls the cleaning member function with a string that has not been validated, such string can contain errors. The cleaning member function doesn't perform any validation process, so, if an error is found (due to an error in the string), the application running can crash.

Chapter 3

Technical documentation



In this chapter, the user will be able to learn various topics about how the library works, the design decisions made and the reasons behind them.

This chapter is expected to be extensive, so, the recommendation for the user is to read carefully the contents of every section to be able to understand the concepts and themes that follow each one.

All the contents are explained as precisely and with as much detail as possible, so all the contents can be understood.

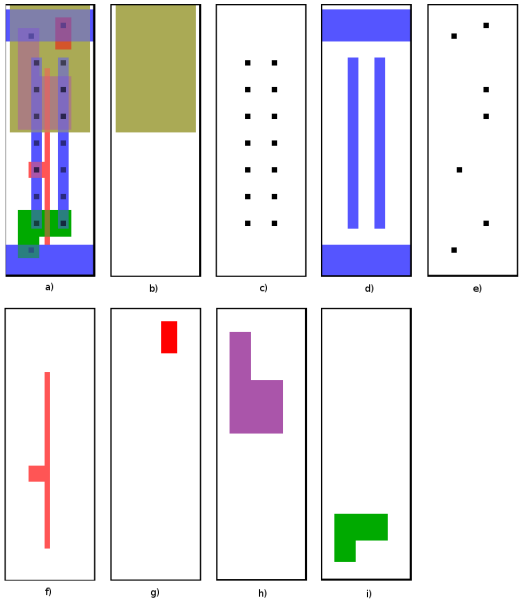


Figure 3.1: Basic inverter where all its layers has been separated: a) Full cell, b) NWELL, c) NDIF, d) PDIF, e) NTIE, f) POLY, g) CONT, h) ALU1, i) REF.

3.1 The CIF format

In this section, the user will learn information related to the CIF format itself. Here, will be explained various concepts related to how the format works. However, this section is not an official technical description of the format. The user can find on-line various sites where the format is explained in greater detail.

The CIF format describes the physical layout (appearance) of an Integrated Circuit design. An Integrated Circuit design can be seen as a collection of cells. Each cell can be seen, in turn, as a smaller collection of other cells, up until we reach a point where a cell is a very basic and specific mathematical operation, like an AND or an OR. These basic components, in turn, are made from various layers of materials, each being composed by some basic figures, like squares, rectangles, circles and polygons. A CIF file is just one file format created to store such physical distribution (layout) of figures that form an Integrated Circuit design. In the figure 3.1 can be seen an example of this concept, where a basic cell (an inverter) is separated in all the different layers (materials) that form it.

3.1.1 Design coordinates

An Integrated Circuit design can be seen as a 2D area where all the components that form it are placed in a Cartesian Plane. In the figure 3.2 can be seen such plane, in which can be found an origin (coordinates `[0,0]`), X increases to the right (and goes negative to the left) and Y increases to the top (and goes negative to the bottom). Most of the Integrated Circuit designs are defined and positioned in the first quadrant (positive values for the coordinates)[1], so the lower left corner of the design, usually, falls in the `[0,0]` coordinate of the plane.

[1]Is important to note that, even if most designs are defined in the first Cartesian quadrant, nothing prevents them to be defined in any of the other three quadrants.

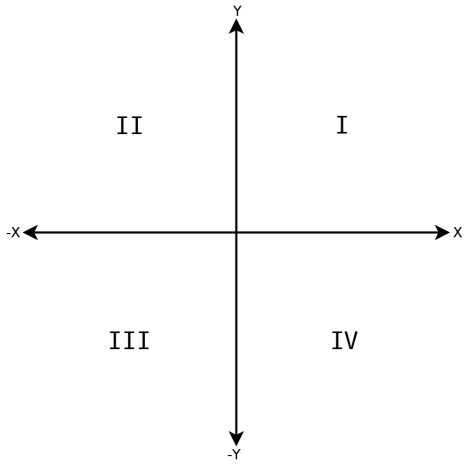


Figure 3.2: Representation of the Cartesian Plane used in the Integrated Designs.

Having the previous detail explained, is important to have in mind that the designs are intended to represent real components that are really small. Such components use as measuring unit the **micrometer** (μ). This unit, even when it is really small, sometimes it not enough precision to represent coordinates nor sizes of pieces of the design, so, various components are measured in hundreds or even thousands of micrometer.

As can be seen, the precision of values is really important when working with Integrated Circuit designs. That brings the importance of floating point values and how to represent them in a design (and, more important, how to store them).

3.1.2 Theorical and real precision of values

As previously mentioned, an Integrated Circuit design is composed by various geometrical figures. Those figures need to be placed with a high level of precision. The previous detail brings a great challenge to any format intended to store an Integrated Circuit design. Even beyond details of how to store the design structure, how to place components, how to correctly organize all the elements, the biggest problem to solve is precision. Various formats available (like the GDSII or OASIS) store the positions and sizes of components using floating point values in binary form, but such method has its limitations[2] .

A floating point representation of a number in a design requires a nearly perfect representation of the original value. Such precision can be achieved using 32 bit floating point values, or even 64 bit floating point values, but even then, they have limitations about the length and real precision stored. Due to this complex problem of how to store a floating point value, the CIF format doesn't allow to use any value of such type. In a CIF file, all the values (positions and sizes) are stored as integer values, and if a floating point is required, an auxiliary **fraction** is used to scale the values (more of this later).

The idea behind the previously defined concept, is that the CIF format doesn't restrict the precision allowed to be used, so, all the tasks related to the process of loading and conversion of the values, are left to the applications that are intended to use the CIF files. Since the format itself doesn't restrict in any way the length (and therefore, the precision) of a value, all the applications that interact with CIF files must apply their own rules to be able to read such values and convert them, as precisely as possible, to floating point representations of the original values.

[2]However, in this document we will not touch other formats nor discuss if any is better than other, that is not the point of this document.

In a CIF file, even when using fractions to help represent high precision values, there is important to note that the values used in positions and sizes are, by themselves, **hundredths** of a **micrometer**. So, if a square is defined in a file as having a length of 100 units per side, we can say that such figure has a length of 1 μ per side. Also, as can be seen, this can be used to represent up to two decimal positions of a floating point value, this means that a value of 1234, in CIF format, represents a real value of 12.34 μ.

With the previous details defined, we can start to define the basic components found in a design: The Primitive Commands.

3.1.3 Primitive Commands

The first group of commands, named in this document as **Primitive Commands**, are those that allow us to define, place and control geometrical figures. The figures that can be used are:

- Box
- Polygon
- Wire
- Round Flash

A **Box** command defines a rectangle with some properties:

- Size
- Position
- Rotation

The **size** of a **Box** is defined as the **width** and **height** of such figure, and both values must be positive. The **position** of a **Box** is defined as the coordinates on which its **center** must be placed. The rotation of a Box is totally optional. The rotation is represented by the coordinates of a point. Such point, along with the center of the Box, forms an angle with respect of the X axis. Such angle is the rotation to be applied to the figure[3] . There is important to note that all the rotations (no only for Boxes) are **counter-clockwise**, never **clockwise**. In the figure 3.3 can be seen an example, where a Box has a rotation described by the points [1,0], [1,1], [-1,-1] and [10,-1].

A **Polygon** defines a list of points that form a figure. Such figure can be as complex or simple as needed. The CIF format doesn't set any kind of restriction about which kind of figures can be defined. The polygons formed must be closed, so the first and last point are implicitly closed (there is no need that the first and last point be the same), can be concave and have self-interceptions. The figure 3.4 shows a basic example of a closed **Polygon**.

A **Wire** is similar to a **Polygon**. The **Wire** has the following properties:

- Width
- List of points

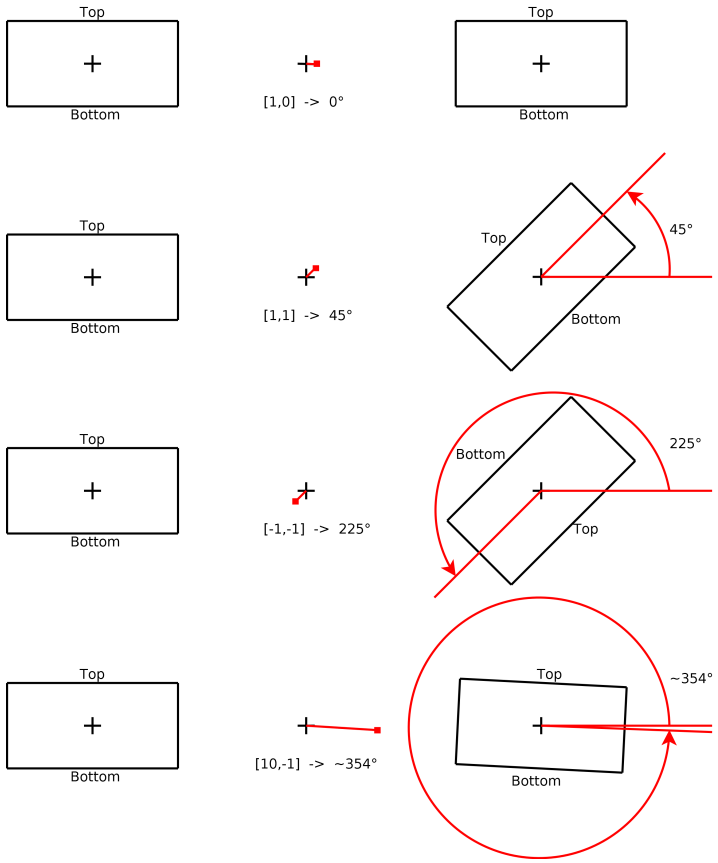


Figure 3.3: Visualization of the effect of the rotation over a Box object.

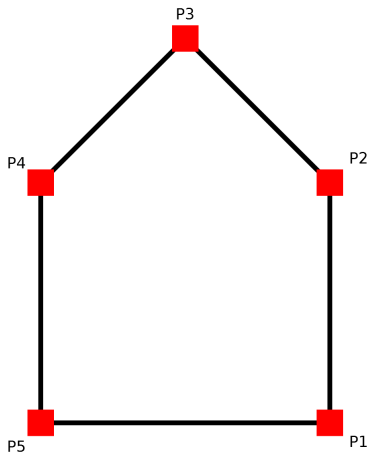


Figure 3.4: Visualization of an example of **Polygon**.

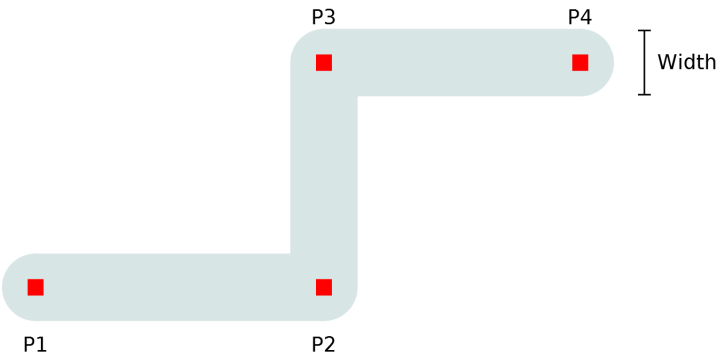


Figure 3.5: Visualization of an example of **Wire**.

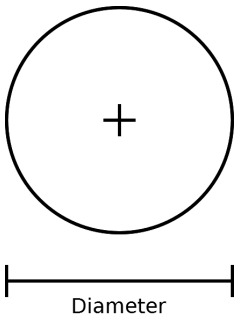


Figure 3.6: Visualization of an example of **Round Flash**.

The **Wire** defines a list of points that aren't a polygon, they're just a path of points over which a stripe of material will be placed, and the width defines the width of such stripe. The figure 3.4 shows a basic example of a **Wire** that has four points and a specific width.

The **Round Flash** defines a circle. Such figure has two properties:

- Position
- Diameter

The **position** of a **Round Flash** corresponds to its center, while the **diameter** is its size. The figure 3.6 shows a basic example of a **Round Flash**.

The **Box** and **Polygon** figures are usually used to represent the smaller components that are used in an Integrated Circuit design and its power connections, while the **Wire** and **Round Flash** are usually used to form the data connection between the small components.

[3] If the Box has a rotation applied, it must rotate around its own center.

3.1.4 Control Commands

The second group of commands, named in this document as **Control Commands**, are those that allow us to define, call, place and rotate those basic components of a design named **cells**.

The first element to understand is a low-level cell . Such component represent a small group of figures (Box, Polygon, Wire or Round Flash or the **Layer** and **User Extension** commands, discussed later). This kind of cell represents small mathematical operations, like an inverter, an OR or an AND, and can be used multiple times in a single Integrated Circuit design. To define such component we need two CIF commands. The first one defines a start and a second defines an end. All the commands found within this two commands are considered part of this cell. These two commands are known in this document as **Definition Start** and **Definition End**.

Cell or Symbol?
In the CIF formal technical documentation, the cells are named **Symbols**, but most of the applications, books and almost any other documentation refers to the cells as **cells**. This detail must be handled with care. In this document, we refer to the cells as **cells**.

The **Definition Start** command have two properties. The first is mandatory, and is an ID that must be assigned to the cell defined, so, that cell can be referred by its unique ID. The second is optional and is formed by two integers (named A and B) that defines a constant **fraction** for which will be multiplied all the size and position values inside the cell definition (this multiplication is used to convert big integers into small and precise floating point values).

The **Definition End** command doesn't need any properties, it just end the definition of a cell and must be preceded by a Definition Start command.

Cells within cells
Is considered an error if a Definition Start command is found after another Definition Start command (before its closing Definition End is found).

Non-consecutive IDs
The CIF format doesn't restrict in any way the order of the IDs used to define cells. They can have any integer as long as such value has not yet been used by another cell and it must be positive.

The following command found in this group is a **Call** command. Such command is used to instantiate a previously defined cell. This behaves just like a function. The user can define small pieces of design (just like small pieces of code) and can use them whenever such component (of behaviour) is needed. As can be inferred, a cell can have calls to other cells that can, in turn, have calls to another cells, and so on.

Cycles are not good
Even if in programming languages a function can be directly call himself, or indirectly call itself calling another function, in a CIF file, such call cycles are not possible and are considered an error since there is no code to stop such commands to call themselves infinitely.

The **Call** command has two main group of properties. The first is mandatory and is the ID of the cell that must be called. Is important to note that such cell ID must be already defined when the Call command is found, in other case is considered an error. The second group of properties is a list of **Transformations** that can be applied to the cell called.

If no transformations are applied to the cell, the cell is just used as defined, so, for example, if there are three calls to the same cell, without any transformation, can be considered that there are three cells overlapped in the same position. Even if such situation is not an error *per se*, is obvious that positioning the same components in the same position over and over is of no help for a design, so, most calls have at least one Transformation applied to the cell.

The transformations can be of three types: **Rotation**, **Translation** or **Mirror**.

A **Translation** transformation adds a very specific value to all the X and Y coordinates of all the elements found within a cell, so this transformation is used to call cells to a specific place in a design. The values used for this transformation are integers that can be positive or negative.

Cell definitions

Most of the times, the low-level cells are defined so their lower left corners are placed in the origin coordinates ([0,0]). This is done to simplify the process of calling cells in a design.

A **Rotation** transformation, as can be inferred by its name, is used to rotate a cell with respect of the origin coordinates of the whole design. In a similar way as how a Box is rotated, a cell rotation is defined using two integers (that can be positive or negative) and represent the angle that the cell must be rotated. Also, as with the Box command, the rotation point can be arbitrary, so, any angle is allowed and it must be counter-clockwise.

A **Mirror** transformation is used to flip a design horizontally or vertically by multiplying the coordinates of the cell components by -1 in X or Y, depending on the transformation used[4].

Another consideration with the transformations is that they no only apply to the figures defined within the cell called, they must be also applied to the Call commands found within this cell, so, they are applied recursively in the cell until no more Call commands are found. This is done to apply transformations no only to the top level cell, but to all the lower level cells.

The next command found in this group is the **Definition Delete** command. As far as the author experience can prove, this command is not extensively used in the Integrated Circuit designs. It is used to remove a previously defined cell, so its ID is available to use again and its contents are not available. This command is useful if the CIF file is being processed command by command (in a linear way) so that cells that are not going to be used again in the design can be safety deleted (freeing memory from the reading device or program). Even if this command is not widely used in today designs, the support must be provided.

The Definition Delete command has one mandatory property, and it is the ID of the cell (that must be already defined) to delete. When deleting, is expected to delete only the direct contents of the cell referred, not the cells mentioned in call commands found within the deleted cell.

The last command found in this group is the **End** command. Such command is intended to mark the **end of the file**. Even if the file has contents later, if this command is found, the reading application or device must consider that the design is over.

[4]Is important to note that, for example, if a figure is located in the center of the first quadrant and an Mirror transformation is applied to it horizontally, the result must be it being placed, horizontally flipped and located in the **second quadrant** (such behaviour is not an error, is expected).

3.1.5 RAW content commands

This group of commands include those which contents are user-defined and that doesn't require any validation from the reading application or device. Any interpretation performed over the contents of these commands are directly dependant of the reading application or device, so, are not covered by the format itself and might not be compatible between applications.

The first command found in this group is the **User Extension** command. This command is intended to provide a way to extend the functionality and capabilities of the format, allowing to define new commands that are application dependant. These commands allow, in some common situations, to define the name of cells and to define connection points, port names and other details that are equally important in a design, but that the CIF format, *per se*, doesn't allow to define.

The second and last command of this group is the **Comment** command. These commands are intended to introduce comment lines on a design to store information not directly used in the design, like author name, date on which the design was stored, or a description of the design.

3.1.6 Layer command

An Integrated Circuit design is composed, as discussed, by geometrical figures that represent components of a design. Those figures must be made of some material and such material must be defined. In a CIF file, exist the **Layer** command, that allow the reader application to know of which material are made the figures it is reading.

When reading a CIF file, once the user finds a Layer command, the following Primitive Commands that it finds can be considered to be made of the material specified by the previously found Layer command. To change the material of which a figure is made, a new Layer command must be placed. As can be seen, all the Primitive commands found after a Layer command are made of the material specified unless a new Layer command is found.

The Layer command has one mandatory property. Such property is the name of the material to use. The initial definition of this command states that the name must be formed with 4 characters, but, since various applications using the CIF format doesn't respect this restriction (using names shorter or longer than 4 characters), this library, when validating a CIF file, will not have in consideration the length of the material names[5].

3.1.7 BNF representation of the format

All the components that form an Integrated Circuit design can be represented by the CIF format using the supported **commands** that where previously described. The CIF commands need to be represented and identified using as less characters as possible. So, the solution used by the format is to represent every command with a specific character. The following text defines the complete **BNF** (Backus Naur Form) description of the format[6]. The BNF is described using the following rules:

- The curly brackets represent a repetition of one or more of its contents.
- The brackets represents that its contents are optional (can or can't be present).
- A character within apostrophes means that it is a literal character.
- The parentheses represent selection between its contents (separated by pipes).

[5] Is also important to note that most applications use their own set of material names even if there exist a standard that respects the 4-char rule.

[6] The format described in this document is not exactly the same as that described in the official documentation. The **BNF** description is adapted to the purposes of this document.

```
FILE ::= {COMMAND}
BLANK ::= Any ASCII except DIGIT, UPPER, '-', '(', ')' or ';'
DIGIT ::= Any digit from '0' to '9'
UPPER ::= Any character from 'A' to 'Z'
SEP ::= An UPPER or BLANK
POINT ::= ['-'] {DIGIT} {SEP} ['-'] {DIGIT}
INTEGER ::= {DIGIT}
LAYERCHAR ::= A DIGIT, UPPER or '_'
EXTCHAR ::= Any ASCII character except ';'
COMMENTCHAR ::= Any ASCII character except '(' or ')'
COMMAND ::= [{BLANK}] ( PRIMITIVE | CONTROL | EXTENSION | COMMENT )
PRIMITIVE ::= ( POLYGON | BOX | ROUNDFLASH | WIRE )
CONTROL ::= ( LAYER | SYMSTART | SYMEND | SYMDEL | CALL | END )
POLYGON ::= 'P' [{BLANK}] POINT [{SEP POINT}] [{SEP}] ';'
BOX ::= 'B' [{BLANK}] INTEGER {SEP} INTEGER {SEP} POINT [ {SEP} POINT [{SEP}] ] ';'
ROUNDFLASH ::= 'R' [{BLANK}] INTEGER {SEP} POINT [{SEP}] ';'
WIRE ::= 'W' [{BLANK}] INTEGER {SEP} POINT [{SEP POINT}] [{SEP}] ';'
LAYER ::= 'L' [{BLANK}] {LAYERCHAR} [{BLANK}] ';'
SYMSTART ::= 'D' [{BLANK}]'S' [{SEP}] INTEGER [{SEP}INTEGER{SEP} INTEGER] [{SEP}] ';'
SYMEND ::= 'D' [{BLANK}] 'F' [{SEP}] ';'
SYMDEL ::= 'D' [{BLANK}] 'D' [{BLANK}] INTEGER [{SEP}] ';'
CALL ::= 'C' [{BLANK}] INTEGER [{[{BLANK}] (TRANS | MIRROR | ROTATION)}] [{BLANK}] ';'
TRANS ::= 'T' [{BLANK}] POINT
MIRROR ::= 'M' [{BLANK}] ('X' | 'Y')
ROTATION ::= 'R' [{BLANK}] POINT
EXTENTION ::= DIGIT [{EXTCHAR}] ';'
COMMENT ::= '(' [{COMMENTCHAR}] ')' [{BLANK}] ';'
END ::= 'E' [ [{SEP}] ';' ]
```

The previous definition explains, in BNF format, how a CIF file is structured. Some of the differences between the definition presented and the official definition are these:

- The order on which the format defines how components (specially the whitespaces) are used is different in order to try to prevent a non-deterministic definition.
- Since in the real world exists examples of production designs that doesn't respect the 4-character rule for the layer names, in our format we are not limiting the names to those 4 characters.
- The format doesn't have in consideration the possible contents of the file beyond the END command since such contents are not of interest for the design (but are not considered an error *per se*).
- There must not be considered a non-deterministic problem the definition of the SEP character. The uppercase characters (A to Z) must override the definition of the BLANK character.

Validating two points of view
Is recommended for the user to take a look at the original documentation to compare our definition and the original one.

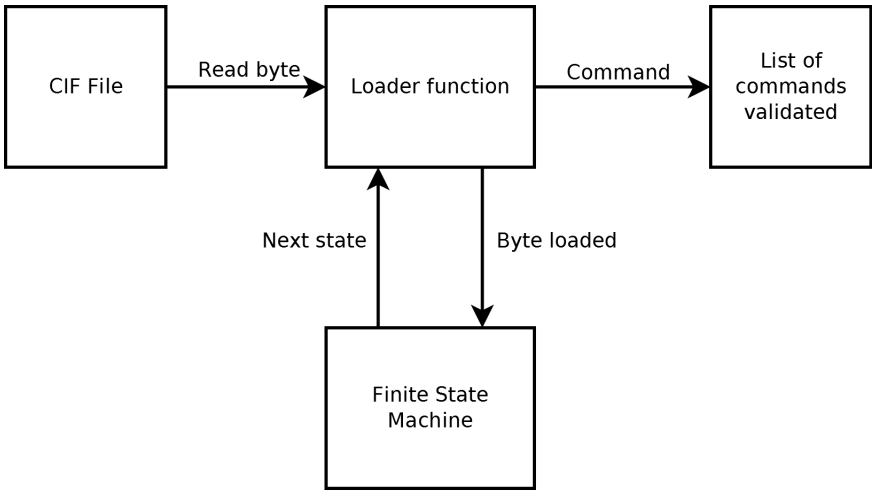


Figure 3.1: Basic diagram of the loading process of a CIF file by this library.

As can be seen, there is important to note that the CIF format defines that all the commands need to end using a semicolon[7] and that most of the commands can be identified by their first character (an upper case character or a digit in the Expansion command).

After defining the contents of a CIF file, the next step is to understand how this library is able to load and validate the commands found in it.

3.2 Validation of a CIF file

This library has the purpose of loading into memory the contents of a CIF file. Such operation requires a validation of the contents loaded to be sure that they are correct and that they meet the format considered. The validation is performed loading the CIF file byte by byte and passing each one into a **Finite State Machine** which is designed to validate the possible structure of the CIF file.

The basic idea of the loading and validation process can be seen in the figure 3.1.

The loading is performed reading one byte from the file and sending such byte to the Finite State Machine, which, in turn, having control of the current state, validates the transition needed, returning the new state to the loading function. The loading function, in turn, validates the state returned by the Finite State Machine. If the state is not an Error State, then the byte is correct. Also, the loading function validates if the returned state means that the loaded byte completed a valid command (which was being stored in a buffer). If such situation is reached, the command is moved to the list of valid commands and the buffer is cleared.

Such process is repeated until the end of the file is reached, the End command is found or an error is detected.

3.2.1 Finite state machine used

Even more important that the process of how to read a byte from a file, the truly crucial detail, when validating a file using a Finite State Machine, is how such Finite State Machine is designed. In this case, the design was carefully made, trying to include as much information as possible.

[7] In the End command, the semicolon is optional but recommended.

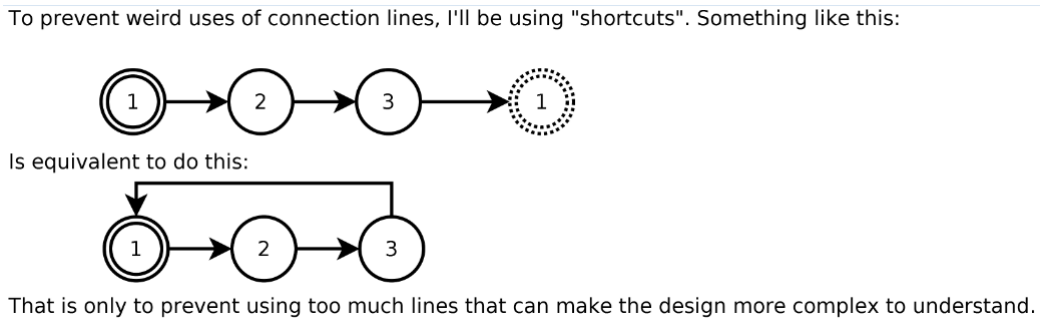


Figure 3.2: Basic diagram of the loading process of a CIF file by this library.

The diagram of such design is too big for this document, so, the image must be available alongside this document.

The first thing the user will be able to see is a little indication of how to interpret the connections between states in the diagram (see figure 3.2).

After that, comes a little section intended to understand the different groups of characters considered:

D Digits, characters from '0' to '9'.

U Uppercase characters, from 'A' to 'Z'.

B Blank characters, any ASCII character except digits, uppercase, minus (-), parentheses or semicolon.

C Comment characters, any ASCII character except parentheses.

S Separator characters, an uppercase character or Blank.

L Layer name characters, any digit, uppercase character or '_' character.

After checking the diagram, the user can see three terminal states: 1, 91 and 92. The state 1 is the starting point of the diagram. Such state is in charge of skipping Blank characters (characters found between commands that are of no use for the design). Every row represents a possible command. All of them start in the state 1 when it detects the starting character of one of it. All the commands end with semicolon. The End command is special, it doesn't need to end with semicolon (state 91) or, after finding it, is just matter of skipping characters (state 92) until the end of the file.

3.2.2 Considerations

There are some considerations that must be keep in mind at all times, because without them, the diagram can be considered to have flaws:

1. The comment command (state 89) isn't considering the fact that most applications support that a comment have internal parentheses. That is, that, meanwhile the parentheses are all closed before the comment command truly ends, there is no problem. That problem is solved in the implementation using some simple counters to know how many parentheses are left to close.

3.3 Loading of a CIF file

The loading of a CIF file, as considered, requires some options to be available to the user. One of such options is the process of loading a CIF file into memory and make the commands available to the user in string form and in class instance form. This design decision is intented to make the life of the user more easy, providing it with the required elements so the results can be used as needed.

3.3.1 Cleaning process of commands

One of the major options provided to the user is the cleaning of the commands. Even if the library can provide to the user the raw (but validated) commands, they can be hard to process because the extra characters that can be found. So, one of the biggest implementations done is the cleaning process. Such option was partitioned according to small groups of commands that can be cleaned in the same way.

Numeric commands

The numeric commands are those that only contain their identification character and, after that, just numeric values. The commands considered to be numeric are:

- Polygon
- Box
- Wire
- RoundFlash

These commands can be cleaned in the same way. The basic operation is very easy to implement. The first step is to backup the identification character and assign it to a new empty string with a whitespace after it. The second step is to iterate over the characters of the command and replace all the characters that are not digits or minus ('-') with whitespaces. This process will remove any not required character from the string, leaving only the good ones. Since the command string was already validated by the Finite State Machine, we can safely assume that all the remaining characters are always command characters.

The following string is an example of a "dirty" command:

```
W1000lol20000,,,this,is,valid20000xoxoxoxoxox-10000,-10000,-500example-4000;
```

And so, the result of this first step is the following string:

```
 1000   20000                 20000             -10000 -10000 -500       -4000
```

Now, as can be seen, the command might be cleaner, but it still has lots of whitespaces and it doesn't have a semicolon. The idea is to convert it into the following string:

```
W 1000 20000 20000 -10000 -10000 -500 -4000 ;
```

The goal is to remove all the whitespaces and add one after the identification character and before the semicolon. Using the previously created string (that contains the identification character and a whitespace), we will start concatenating to it the remaining pieces of command (separated by single whitespaces). To do such operation, we start with a *istringstream* class instance. We create it with the partially cleaned command as constructor argument.

Using the previously created instance, we start to iterate over the process of extracting a word (a string) from the instance, until we reach the End Of File of such stream. Each word extracted will be contactenated to the temporal string (the one that has the identification character), with a trailing whitespace.

After this loop, we just need to concatenate the final semicolon with a trailing whitespace.

The last step on the cleaning process is to return the newly formed string to the caller.

Layer commands

The Layer commands are those that only contain their identification character and, after that, only uppercase characters, underscores and digits.

The basic operation is very easy to implement. The first step is to backup the identification character and assign it to a new empty string with a whitespace after it. Then, we manually replace the first character with a whitespace. The third step is to iterate over the characters of the command and replace all the characters that are not digits, uppercase characters or underscores with whitespaces. This process will remove any not required character from the string, leaving only the good ones. Since the command string was already validated by the Finite State Machine, we can safely assume that all the remaining characters are always command characters.

The following string is an example of a ”dirty” command:

```
Lthis.is,,and.examplePOLY_OTHERx;
```

And so, the result of this first step is the following string:

```
POLY_OTHER
```

Now, as can be seen, the command might be cleaner, but it still has lots of whitespaces and it doesn't have a semicolon. The idea is to convert it into the following string:

```
L POLY_OTHER ;
```

The goal is to remove all the whitespaces and add one after the identification character and before the semicolon. Using the previously created string (that contains the identification character and a whitespace), we will start concatenating to it the remaining pieces of command (separated by single whitespaces). To do such operation, we start with a *istringstream* class instance. We create it with the partially cleaned command as constructor argument.

Using the previously created instance, we start to iterate over the process of extracting a word (a string) from the instance, until we reach the End Of File of such stream. Each word extracted will be contactenated to the temporal string (the one that has the identification character), with a trailing whitespace.

After this loop, we just need to concatenate the final semicolon with a trailing whitespace.

The last step on the cleaning process is to return the newly formed string to the caller.

Multiple words in a layer name

The CIF format doesn't allow layer names to contain spaces (so, ”POLY METAL” is invalid). The idea of using a istringstream isn't invalid since the command was already validated by the Finite State Machine, and is guaranteed that it has only one word.

Call commands

The Call commands are those that only contain their identification character and, after that, only uppercase characters and digits.

The basic operation is very easy to implement. The first step is to backup the identification character and assign it to a new empty string with a whitespace after it. Then, we manually replace the first character with a whitespace. The third step is to iterate over the characters of the command and replace all the characters that are not digits, uppercase characters or minus with whitespaces. This process will remove any not required character from the string, leaving only the good ones. Since the command string was already validated by the Finite State Machine, we can safely assume that all the remaining characters are always command characters.

The following string is an example of a "dirty" command:

```
C1T20000 -20000R1000000 -59999MXexampleMY MXMXMXMYMXR100 100R100 100;
```

And so, the result of this first step is the following string:

```
 1T20000 -20000R1000000 -59999MX       MY MXMXMXMYMXR100 100R100 100
```

Now, as can be seen, the command might be cleaner, but it still has lots of whitespaces and other components of the command (like the Mirror options) are just too close (we need more whitespaces), and also it doesn't have a semicolon. The idea is to convert it into the following string:

```
C 1 T 20000 -20000 R 1000000 -59999 M X M Y M X M X M X M Y M X R 100 100 R 100 100 ;
```

The goal is to remove all the whitespaces and add one after the identification character, before the semicolon and between the call options. Now, the cleaning process of this step is a little bit more complicated that the previousn ones. In this case, we need to split a string without clear indications of wherer we need to cut. To process the string, we will use some properties of the command format:

1. The numbers found in the command can't be togheter, that is, that between each one must be a whitespace or an uppercase character.
2. There can't be two whitespaces togheter. If two or more whitespaces are found, we will remove all but the first.
3. The uppercase characters (the call options) require to have whitespaces before and after them.
4. The minus character requires a whitespace before it.

The previously defined properties can be used to define a code block that is able to process the whole command, adding the required whitespaces and removing the extra ones:

Listing 3.1: Implemented block of code using the properties of the Call Command.

```
1 // "command" is the partially cleaned command (string type)
2 // "tmp" is a temporal variable (string type)
3 // "final_command" is the result of the whole process, it
4 //                  already has the identification character.
5
6 for ( unsigned int i = 1; i < command.size (); i++ )
7 {
8     // Get the current character to validate.
9     tmp = command[ i ];
```

```
10
11      // If the character is a digit, append it directly into
12      // the final command.
13      if ( std::isdigit ( command[ i ] ) )
14      {
15          final_command += tmp;
16      }
17      // Validate if the current character isn't a whitespace.
18      else if ( command[ i ] != ' ' )
19      {
20          // It isn't, so, whatever it is, validate if the final
21          // command doesn't have a whitespace at its end. If it
22          // doesn't have one, add one.
23          if ( final_command[ final_command.size () - 1 ] != ' ' )
24          {
25              final_command += " ";
26          }
27
28          // Append the current character (it can be a minus or an
29          // uppercase character.
30          final_command += tmp;
31
32          // If the current character is an uppercase character, add
33          // a whitespace after it.
34          if ( std::isupper ( command[ i ] ) )
35          {
36              final_command += " ";
37          }
38      }
39      // This else-if means that the current character is a whitespace.
40      // If the final command doesn't have one at its end, add one.
41      else if ( final_command[ final_command.size () - 1 ] != ' ' )
42      {
43          final_command += " ";
44      }
45  }
```

Using the previous code block, we can correctly format the contents of the Call Command. After this loop, we just need to concatenate the final semicolon with a trailing whitespace.

The last step on the cleaning process is to return the newly formed string to the caller.

Definition control commands

The Definition control commands are those that contain their identification characters and, after that, only digits.

The basic operation is very easy to implement. The first step is to backup the first identification character and assign it to a new empty string with a whitespace after it. Then, we manually replace the first character with a whitespace. The third step is to iterate over the characters of the command and replace all the characters that are not digits or uppercase characters. This process will remove any not required character from the string, leaving only the good ones. Since the command string was already validated by the Finite State Machine, we can safely assume that all the remaining characters are always command characters.

The following string is an example of a "dirty" command:

```
DxxxxxSlol100im,totally,correct;
```

And so, the result of this first step is the following string:

```
S   100
```

Now, as can be seen, the command might be cleaner, but it still has lots of whitespaces and it doesn't have a semicolon. The idea is to convert it into the following string:

```
D S 100 ;
```

The goal is to remove all the whitespaces and add one after the identification character, before the semicolon and between the call options. Now, the cleaning process of this step is a little bit more complicated that the previousn ones (but similar to the one found for the Call Command, just a little bit more simple). In this case, we need to split a string without clear indications of wherer we need to cut. To process the string, we will use some properties of the command format:

1. The numbers found in the command can't be togheter, that is, that between each one must be a whitespace or an uppercase character.
2. There can't be two whitespaces togheter. If two or more whitespaces are found, we will remove all but the first.
3. The uppercase characters (the second identification character) require to have whitespaces before and after them.

The previously defined properties can be used to define a code block that is able to process the whole command, adding the required whitespaces and removing the extra ones:

Listing 3.2: Implemented block of code using the properties of the Call Command.

```
1  // "command" is the partially cleaned command (string type)
2  // "final_command" is the result of the whole process, it
3  //                  already has the identification character.
4
5  for ( unsigned int i = 1; i < command.size (); i++ )
6  {
7     // If the character is a digit, append it directly into
8     // the final command.
9     if ( std::isdigit ( command[ i ] ) )
10    {
11       final_command += command[ i ];
12    }
13    // Validate if the current character isn't a whitespace.
14    else if ( command[ i ] != ' ' )
15    {
16       // Append the current character and also a whitespace.
17       final_command += command[ i ];
18       final_command += " ";
19    }
20    // This else-if means that the current character is a whitespace.
21    // If the final command doesn't have one at its end, add one.
22    else if ( command[ i ] == ' ' && final_command[ final_command.size () - 1 ] != '
          ' )
23    {
24       final_command += " ";
25    }
26 }
```

Using the previous code block, we can correctly format the contents of the Definition Commands. After this loop, we just need to concatenate the final semicolon with a trailing whitespace.

The last step on the cleaning process is to return the newly formed string to the caller.

Expansion commands

The Expansion commands are those that contain their identification characters and, after that, any character that the user might require.

These commands are the easiest to process, since they only require to have their identification charcters separated. The followin is an example of valid command:

```
9 CORE;
```

In this example, the Expansion command ID is ”9”, and its content ”CORE” must be processed and validated by the application loading the file. The CIF format itself doesn't give any special meaning to the Expansion commands, so, in this library they are being as they are defined.

3.4 Converting strings into class instances

Once the commands are totally validated and cleaned, they can be converted into class instances to be used by the user as needed. The classes are designed to be as simple as possible and to be able to be stored in a single *vector* class instance (using polymorphism to call derivated instance member functions from base class pointers).

All the classes are gouped in 3 main groups (depending on their usage): Commands, Validation and Loading.

The classes will be explained according to their usage and importance. The first group to be explained is the Validation group since they are independent from the other two groups and are just a few classes (and, also, they are a dependence of the third group). The second group will be the Command classes, since they are the most numerous and are a dependence of the third group. Finally, the Loading group will be last one since it requires all the other classes.

In the same way, in each group, the user can find the most basic classes explained first, and then, the most complex ones.

3.4.1 Validation group of classes

The first group to be explained, the Validation group, is the one used to validate the contents of the CIF file. This group is composed by the State, FiniteStateMachine and CIFFSM classes, that provides the functionality required to validate the format and contents of the file.

Is important to note that these classes, by themselves, doesn't validate the file of the user, but they are used by other classes that perform the opening and reading of the user's file.

Class: State

The first class to analyse is the State class. After analysing the problem of validation of a file, the final design demanded a Finite State Machine (FSM) to be used to correctly parse and validate the contents of the file. Being working in C++, the FSM required to be a class, and as a result of this, its states would be also classes. Also, having in mind the use of the whole ASCII set (validation of a file), the State class was designed with plain text files in mind. The UML diagram representing such class can be found in the figure 3.1.

A State was defined as a singular point in the whole FSM. Internally, the State will be able to recognize and support the complete set of ASCII characters (even the extended ones), and to be able to represent such set of options, the class will contain an array of *int* values, 256 of them. Every element of the array will represent a single character in the ASCII set, that is, that the ASCII value of a character

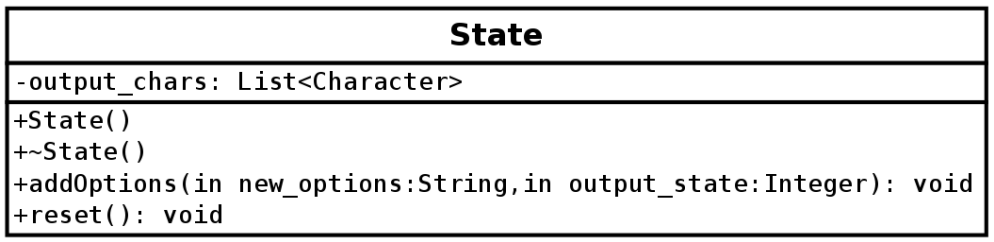


Figure 3.1: UML diagram of the **State** class.

will be used to access directly the corresponding index in the array (for example, a whitespace, having an ASCII value of 32, will access the index 32). The value stored in each position will represent the next state in the FSM in case that such character is loaded when loading the file. By default, all the elements of the array will be initialized with -1, being such value a transition to an error.

In the class found in the library, such array is named **output_chars**, and is of type *int*. This is done in this way because we need to be able to redirect to a negative state (never really added to the FSM, its value just means error), and, if we use the *char* type for the array, we would need to work with signed characters, limiting the total ammount of states to wich we can redirect to just 128. Since the FSM might change in the future, in the library was used the *int* type instead to be able to work with as many states as needed.

Also, since the design requires to have a vast ammount of possible input characters to be considered, there must be a way to add transitions more easily. To simplify the process of adding transitions, there is a member function named **addOptions** that takes as argument a constant *string* class instance as reference and a single integer, also as constant reference[8] . The integer is the jump to perform in the FSM when any of the characters in the string is found, also passed as a constant reference. The following code block shows this idea applied in the library:

Listing 3.3: **addOptions** code example.

```
1  void OpenCIF::State::addOptions ( const std::string& new_options ,
2                                    const int& exit_state )
3  {
4     for ( unsigned int i = 0; i < new_options.size (); i++ )
5     {
6        state_options[ (int)(new_options[ i ]) ] = exit_state;
7     }
8
9     return;
10 }
```

The first value passed, a *string* class instance, named **new_options**, and the second, an *int* value named **exit_state**, are the values required to add new transitions to a single state. When called, a *for* cicle will iterate over the string characters. Each character will be converted into an integer value (to prevent warnings when compiling about convertion of types when accessing the array) and used to access the transitions array at some position (the ASCII value of the character). In such position (that, by default, can be found a -1, meaning transition to error) will be assigned the new jump to the following state.

The member function *addOptions* will be in charge of adding transitions to the State, but the complex part will be accesing such transitions. The idea behind the State is that the FSM will *ask* to the current

[8]The library uses constant references to avoid creating innecessary copies of instances in memory, and to avoid accidental modifications of data in memory.

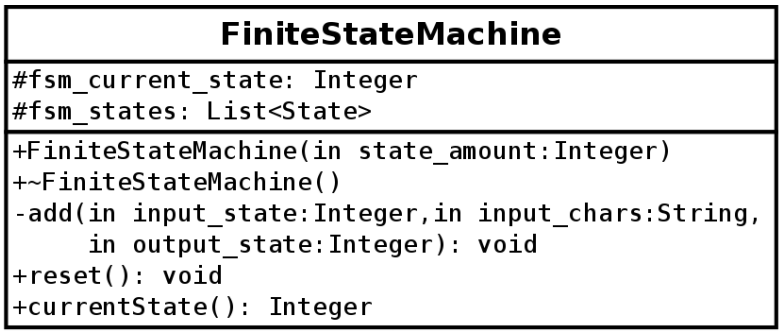


Figure 3.2: UML diagram of the **FiniteStateMachine** class.

State where to go with the current character loaded from the file. To access the list of the transitions, an overloaded operator was added to the State class. Such operator (brackets) will be used to access the internal array of the State, returning the integer value stored in the expected position. For example, when a whitespace is loaded, it has an ASCII value of 32, and, as such, will be accessed the internal array in its index 32 (33rd position), and the value of such integer will be returned. The whole idea can be seen in the following code block:

Listing 3.4: **addOptions** code example.

```
1 int OpenCIF::State::operator[] ( const char& input_char )
2 {
3     return ( state_options[ (int)input_char ] );
4 }
```

As can be seen, a *char* value is expected to be provided as argument (the current loaded character from the file). Such value is converted into an *int* value to access without warnings the array of transitions. The accessed value is then returned to the caller.

Class: FiniteStateMachine

The following class to explain is the *FiniteStateMachine* class. It doesn't provide specific functionality related to the CIF format. This class will only provide a generic mechanism to represent and manage the FSM needed to validate the CIF format. The UML diagram found in the figure 3.2 details the main idea behind the class.

As can be seen, the class is intented to be a base class for the CIFFSM class, wich specializes it and adds the required functionality.

The *FiniteStateMachine* class provides a new protected attribute named **fsm_current_state**. This protected attribute will be used to keep track of the current state when validating the CIF file. This attribute must be of *int* type to match the type used in the State class.

A second protected attribute, **fsm_states**, represents a list (a *vector* class instance) of State class instances that will form the whole set of states in the FSM. This array doesn't have a specific size, so, when creating an instance of this class, it will required for the caller to set the ammount of states to be created (and added to this list).

To control the FSM, the public member fucntion **reset** is intented to reset (or restart) the whole FSM (done by setting the current state to 1). This will help in case that such functionality is required.

The public member function **currentState** can be used to return the current state of the FSM and keep track of its functionality.

The private member function **add** is expected to be used only by those classes that extend this one. This member function takes as arguments the current state from which to perform a jump (as a constant reference to an *int* value), a constant reference to a *string* class instance that represents the set of input characters to perform the jump with, and the destination state of the jump, as a constant reference to an *int* value. Is important to note that, even if this member function is implemented, the whole class is expected to be just a "skeleton" for others with more specific logic (so, this member function is private to prevent direct usage of the class). In the following code block, can be seen the main idea behind this member function:

Listing 3.5: **add** code example.

```
1 void OpenCIF::FiniteStateMachine::add ( const int& input_state ,
2                                         const std::string& input_chars ,
3                                         const int& output_state )
4 {
5     fsm_states[ input_state ].addOptions ( input_chars , output_state );
6
7     return;
8 }
```

As can be seen, having the **fsm_states** array of *State* class instances, the first argument, **input_state**, is used to access the right state from the whole FSM. Then, from such state, its member function **addOptions** is accessed to pass to it, as argument, the values of **input_chars** and **output_state**.

Finally, this class is expected to provide a similar functionality to the caller as the *State* class, providing access to an overloaded operator (brackets) to test a character with the current state and return the new state. All the previous functionality can be found in the following code block, wich performs all those operations in a single line:

Listing 3.6: **add** code example.

```
1 int OpenCIF::FiniteStateMachine::operator[] ( const char& input_char )
2 {
3     return ( fsm_current_state = fsm_states[ fsm_current_state ][ input_char ] ,
          fsm_current_state );
4 }
```

The line operates using some commodities allowed by the language, like the *comma* operator. Using such operator, we will execute the instruction found in the left, and returning the value to its right. In the left, we have, first, an access to the **fsm_states** array using the value of **fsm_current_state**. With that, the code is accessing the correct and actual state in the FSM. From such accessed state, its brackets overloaded operator is called to test the current character (**input_char**). This call to its overloaded operator will return the next state in the FSM according to the configuration done and the current state and input. Such returned value (the next state), is then stored in the **fsm_current_state** attribute, and, as a final step in the whole line, the right side of the **comma** operator, returning the value of the **fsm_current_state** value itself.

Class: CIFFSM

As mentioned previously, the *FiniteStateMachine* class only provides the basic structure of the FSM required to validate a CIF file. The class *CIFFSM* does that and specializes it to add the required functionality. The UML diagram found in the figure 3.3 details the idea behind the class.

Internally, the CIFFSM class will integrate the functionality provided by the FiniteStateMachine class, adding some new components.

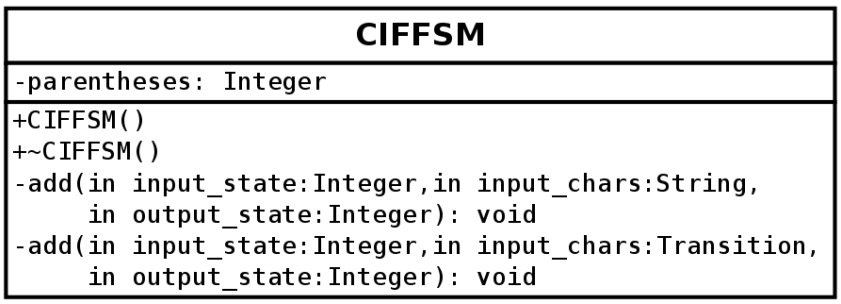


Figure 3.3: UML diagram of the **CIFSM** class.

The first new element found is the private attribute **parentheses**, a counter of *int* type used internally to keep track of open parentheses in the Comment command when validating such command.

Next, can be found the private member function **add**. This member function is the same as the one found in the *FiniteStateMachine* class. The only difference (or modification) is that its prototype is being re-defined in the interface of the class. This is done to keep it hidded from the caller, but to make it available to class itself[9] .

The second **add** member function (that is also private) is new, and is intended to be used when configuring the FSM for the CIF format. The difference with the previous member function is that this one takes as second argument an *enum* instance variable that represents a predefined set of characters. Internally, the member functins uses a *switch* structure to validate the value of the argument, and to perform the required operations. The *enum* can be found with the following values defined:

Digit These are the characters '0' to '9'.

UpperChar These are the characters 'A' to 'Z'.

LowerChar These are the characters 'a' to 'z'.

BlankChar These are all the characters in the ASCII set that are not a digit, an uppercase character, minus ('-'), parentheses or semicolon.

UserChar These are all the characters in the ASCII set that are not a semicolon.

CommentChar All ASCII characters (from 0 to 255).

SeparatorChar LowerChar or BlankChar.

LayerNameChar Digit, UpperChar or underscore.

ExtentionChar All ASCII characters, except semicolon.

The main idea behind these values is to make it easier to configure the FSM in the class instance. Using such values, it becomes easier to configure the states.

The **Constructor** is in charge of calling the father class, initializing the ammount of states for this FSM (92 in this case). Also, the constructor is in charge of effectively calling the **add** member function in order to configure and set all the states and jumps defined by the FSM itself. The following code block demonstrates how the states are set in the constructor:

[9]If class A has a private member function, and class B performs a public heritage from A, by default, class B will not be able to access the private member function of A, to, just re-defining such member of A as private in B, makes it available to B, but keeps it private to the outside.

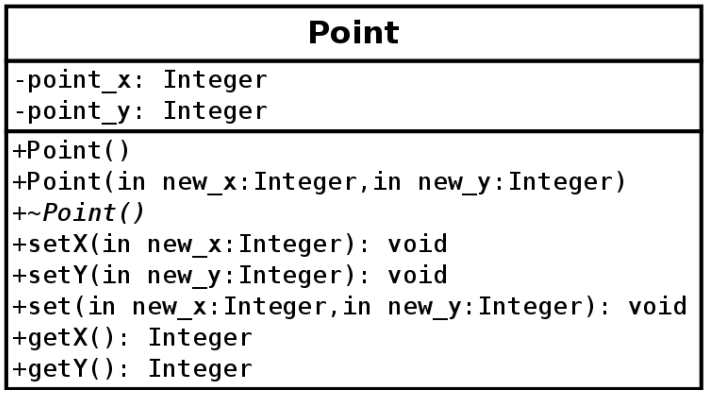


Figure 3.4: UML diagram of the **Point** class.

Listing 3.7: Constructor code example.

```
1  add ( 1 , BlankChar , 1 );
2  add ( 1 , "P" , 2 );
3  add ( 1 , "B" , 14 );
4  add ( 1 , "R" , 31 );
5  add ( 1 , "W" , 40 );
6  add ( 1 , "L" , 54 );
7  add ( 1 , "D" , 57 );
8  add ( 1 , "C" , 70 );
9  add ( 1 , Digit , 88 );
10 add ( 1 , "(" , 89 );
11 add ( 1 , "E" , 91 );
```

In the library, the calls are grouped by command. That is, that all the states required to validate a specific command are groupes using comments.

3.4.2 Command group of classes

The Command group of classes is intended to group those classes that represents commands from the CIF format. These classes are intended to provide a more abstract and simple way to access the information contained in the commands themselves.

These classes are used when converting a *vector* class instance of *string* instances is ready to be converted into command instances.

Class: Point

The first class in this group is a very basic class that represents a single point in space of the design. This point can be a position or rotation angle. The UML diagram found in the figure 3.4 details the idea behind the class.

As can be seen, the class just represent the X and Y coordinates of the desired point. The class provides two constructors. The firs one being the default constructor (without arguments), initializing both coordinate values in zeros. The second constructor initializes the point with the values passed as arguments.

The class also provides access to set the values after creation. This is done using one of the three member functions provided:

Negative coordinates
Is important to remember that the coodinates of a point can be negative. By this reason, the coordinate values are defined as *signed long int* values.

- **setX** This member function takes as argument a constant reference to a *signed long int* value. The value is stored directly into the X component of the point.
- **setY** This member function takes as argument a constant reference to a *signed long int* value. The value is stored directly into the Y component of the point.
- **set** This member function takes as argument two constant references to a *signed long int* values. The values are stored directly into the X and Y components of the point.

Also, the class provides two member functions intented to retrieve the values of X and Y coordinate components. Those member functions are **getX** and **getY**. These member functions return signed *long int* values when called.

The Point class also provides two overloaded operators for flow extraction and insertion. The overloaded operator » is used to load a Point from a **istream** class instance (such instance can be an open file, or the standard input). The second overloaded operator « is used to write a Point to a **ostream** class instance (also, like an open file or the standard output). The following code block shows such operators as defined in the library.

Listing 3.8: Overloaded operators of the Point class.

```
1  std::istream& operator >> ( std::istream& input_stream ,
2                              OpenCIF::Point& point )
3  {
4     long int x , y;
5
6     input_stream >> x >> y;
7     point.set ( x , y );
8
9     return ( input_stream );
10 }
11
12 std::ostream& operator << ( std::ostream& output_stream ,
13                              const OpenCIF::Point& point )
14 {
15    output_stream << point.getX () << " " << point.getY ();
16
17    return ( output_stream );
18 }
```

As can be seen, when extracting, the expected format is two values separed by a blank character (a single space, enter, tabulator, etc). When inserting, the behaviour is similar: First, the X coordinate value is inserted, then a single whitespace and, finally, the Y coordinate value.

Class: Size

The second class in this group is a very basic class that represents the size of a component (like a figure). The size represents length in X and Y coordinates, so it is very similar to the Point class previously explained. The UML diagram found in the figure 3.5 details the idea behind the class.

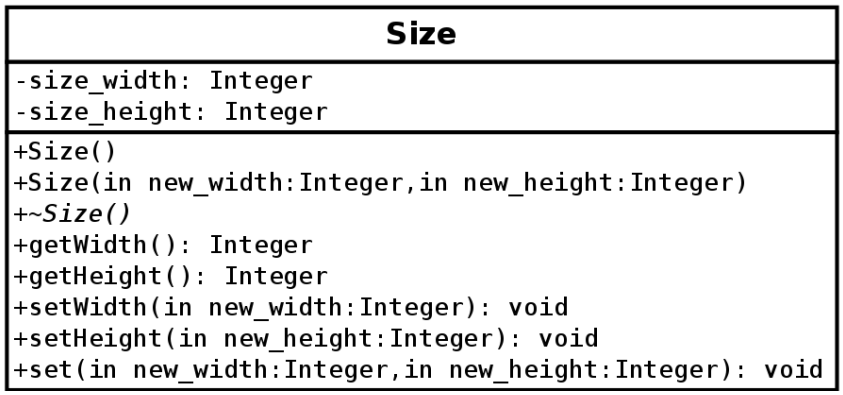


Figure 3.5: UML diagram of the **Size** class.

The main difference between the Size and Point classes are just the names and data types. The Point class is focused in storage and management of an X-Y coodinate with the possibility of using negative values. The Size class is focused in storage and management of X-Y size components of a figure with the restriction of using unsigned values.

The Size class also provides two overloaded operators for flow extraction and insertion. The overloaded operator » is used to load a Size from a **istream** class instance (such instance can be an open file, or the standard input). The second overloaded operator « is used to write a Size to a **ostream** class instance (also, like an open file or the standard output). The following code block shows such operators as defined in the library.

Listing 3.9: Overloaded operators of the Size class.

```
1  std::istream& operator>> ( std::istream& input_stream ,
2                             OpenCIF::Size& command )
3  {
4     unsigned long int x , y;
5
6     input_stream >> x >> y;
7     command.size_height = y;
8     command.size_width = x;
9
10    return ( input_stream );
11 }
12
13 std::ostream& operator<< ( std::ostream& output_stream ,
14                            const OpenCIF::Size& command )
15 {
16    output_stream << command.getWidth () << " " << command.getHeight ();
17
18    return ( output_stream );
19 }
```

As can be seen, when extracting, the expected format is two values separed by a blank character (a single space, enter, tabulator, etc). When inserting, the behaviour is similar: First, the widh value is inserted, then a single whitespace and, finally, the height value.

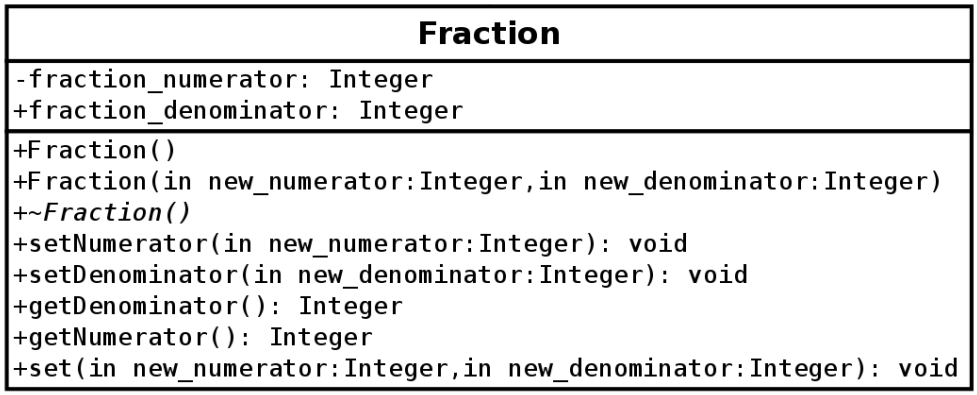


Figure 3.6: UML diagram of the **Fraction** class.

Class: Fraction

The third class in this group is the one that represents the scaling factor of size and position values in a Definition. Since the CIF format doesn't allow the use of floating point values, there is the need to represent fractionary values without decimals, and in this case, when defining a new Definition, there is possible to define the values of such fraction, wich is going to be applied to the values found within the Definition itself. The UML diagram found in the figure 3.6 details the idea behind the class.

In this case, the Fraction class contains two attributes of type *unsigned long int*. This is because the fraction is intended to transform other integer values (that might already have sign) into floating point values. Because of that, the fractions don't require to have sign.

The class has a **set** member function that takes as argument two constant references to *unsigned long int* values. The first value is going to be stored as the **enumerator** of the fraction, while the second is going to be stored as the **denominator**.

There is possible to set and get the denominator and numerator values independently. The class has a member function named **setNumerator** that takes as argument a constant reference to a *unsigned long int* value, and a member function named **setDenominator** that takes the same value type as argument. In the same way, there is a member function **getNumerator** and a member function **getDenominator**. Both of these member functions return, when called, an *unsigned long int* value.

The Fraction class also provides two overloaded operators for flow extraction and insertion. The overloaded operator » is used to load a Fraction from a **istream** class instance (such instance can be an open file, or the standard input). The second overloaded operator « is used to write a Fractio to a **ostream** class instance (also, like an open file or the standard output). The following code block shows such operators as defined in the library.

Listing 3.10: Overloaded operators of the Fraction class.

```
1  std::istream& operator>> ( std::istream& input_stream ,
2                             OpenCIF::Fraction& command )
3  {
4     unsigned long int numerator, denominator;
5
6     input_stream >> numerator >> denominator;
7
8     command.set ( numerator , denominator );
9
10    return ( input_stream );
```

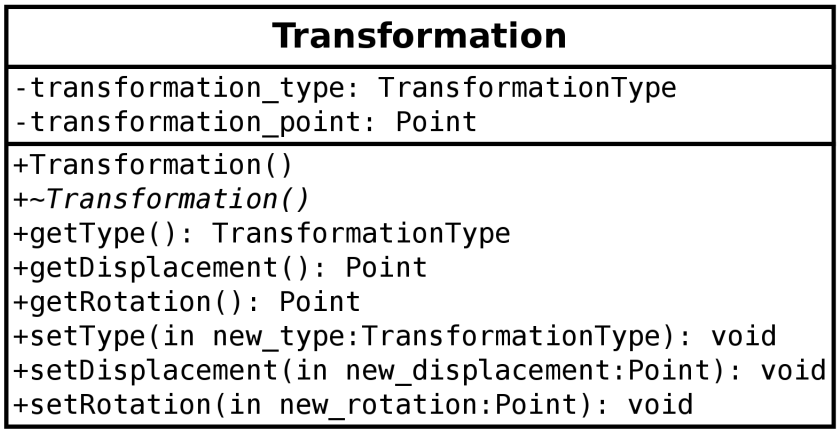


Figure 3.7: UML diagram of the **Transformation** class.

```
11 }
12
13 std::ostream& operator<< ( std::ostream& output_stream ,
14                             const OpenCIF::Fraction& command )
15 {
16    output_stream << command.getNumerator () << " " << command.getDenominator ();
17
18    return ( output_stream );
19 }
```

As can be seen, when extracting, the expected format is two values separed by a blank character (a single space, enter, tabulator, etc). When inserting, the behaviour is similar: First, the numerator value is inserted, then a single whitespace and, finally, the denominator value.

Class: Transformation

The Transformation class represents a single transformation required to be applied to a Cell in a Call command. All the possible transformations (Rotation, Displacement, Vertical Mirror and Horizontal Mirror) can be represented with this class. The UML diagram found in the figure 3.7 details the idea behind the class.

By default the only consructor available initializes the class as a neutral[10] Displacement transformation (one of the most common transformations).

To set and get the type of the instance, the class provides access to the setType and getType member functions. The set function takes as argument a constant reference to the type to set. The get function returns a copy of the type actually setted.

The types available can be found in an enum structure found within the class itself. The types available are:

- **Displacement** An addition to the coordinates of the elements found within a Call command.
- **Rotation** A rotation of the cell called by the Call command.
- **HorizontalMirroring** A multiplication by -1 of all the X coordinates of the cell called by the Call command.

[10] A displacement of zero in X and Y coordinates.

- **VerticalMirroring** A multiplication by -1 of all the Y coordinates of the cell called by the Call command.

The Transformation class contains an attribute named **transformation_point**, which is of type Point. This point is used to store the values related to the Displacement and Rotation types.

The class provides a get and set member functions to get the Point when the instance is configured as Displacement. These member functions are **setDisplacement** and **getDisplacement**. The first takes as argument a constant reference to a Point instance, while the second returns a copy of the same point.

Also, the class provides a get and set member functions to get and set the Point when the instance is configured as a Rotation. These member functions are **setRotation** and **getRotation**. The first takes as argument a constant reference to a Point instance, while the second returns a copy of the same point.

There is important to note that the previous four member functions work over the same attribute in the instance. That is, that both set member functions assign their arguments to the same attribute, and both get member functions return the same instance. This is done in order to provide a more simple interface to the user, providing more clear names depending on the type of the instance.

The Transformation class also provides two overloaded operators for flow extraction and insertion. The overloaded operator » is used to load a Transformation from a **istream** class instance (such instance can be an open file, or the standard input). The second overloaded operator « is used to write a Transformation to a **ostream** class instance (also, like an open file or the standard output). The following code block shows such operators as defined in the library.

Listing 3.11: Overloaded operators of the Transformation class.

```
1  std::ostream& operator<< ( std::ostream& output_stream ,
2                             const OpenCIF::Transformation& transformation )
3  {
4     switch ( transformation.getType () )
5     {
6        case OpenCIF::Transformation::Displacement:
7           output_stream << "T ";
8           output_stream << transformation.getDisplacement ();
9           break;
10
11       case OpenCIF::Transformation::Rotation:
12          output_stream << "R ";
13          output_stream << transformation.getRotation ();
14          break;
15
16       case OpenCIF::Transformation::VerticalMirroring:
17          output_stream << "M Y";
18          break;
19
20       case OpenCIF::Transformation::HorizontalMirroring:
21          output_stream << "M X";
22          break;
23    }
24
25    return ( output_stream );
26 }
27
28 std::istream& operator>> ( std::istream& input_stream ,
29                            OpenCIF::Transformation& transformation )
30 {
31    std::string type;
32    OpenCIF::Point point;
33
```

```
34      input_stream >> type;
35
36      switch ( type[ 0 ] )
37      {
38          case 'T':
39              input_stream >> point;
40              transformation.setDisplacement ( point );
41              transformation.setType ( OpenCIF::Transformation::Displacement );
42              break;
43
44          case 'R':
45              input_stream >> point;
46              transformation.setRotation ( point );
47              transformation.setType ( OpenCIF::Transformation::Rotation );
48              break;
49
50          case 'M':
51              input_stream >> type;
52              transformation.setType ( ( type == "X" )
53                                       ? OpenCIF::Transformation::HorizontalMirroring
54                                       : OpenCIF::Transformation::VerticalMirroring );
55              break;
56      }
57
58      return ( input_stream );
59  }
```

As can be seen, when extracting, the expected format is a character that signals the transformation type (T, R or M), followed by a point (for Displacement and Rotation) or an X or Y (for Mirror). When inserting, the behaviour is similar: First, the character that defines the type of the instance, followed by the value asociated (a Point or a character).

Class: Command

Class: Command

Class: PrimitiveCommand

Class: PrimitiveCommand

Class: PathBasedCommand

Class: PathBasedCommand

Class: PolygonCommand

Class: PolygonCommand

Class: WireCommand

Class: WireCommand

Class: PositionBasedCommand

Class: PositionBasedCommand

Class: BoxCommand

Class: BoxCommand

Class: RoundFlashCommand

Class: RoundFlashCommand

Class: ControlCommand

Class: ControlCommand

Class: DefinitionStartCommand

Class: DefinitionStartCommand

Class: DefinitionDeleteCommand

Class: DefinitionDeleteCommand

Class: CallCommand

Class: CallCommand

Class: DefinitionEndCommand

Class: DefinitionEndCommand

Class: EndCommand

Class: EndCommand

Class: RawContentCommand

Class: RawContentCommand

Class: UserExtentionCommand

Class: UserExtentionCommand

Class: CommentCommand

Class: CommentCommand

Class: LayerCommand

Class: LayerCommand

3.4.3 Loading group of classes

The final group of classes contains only one class: The File class.

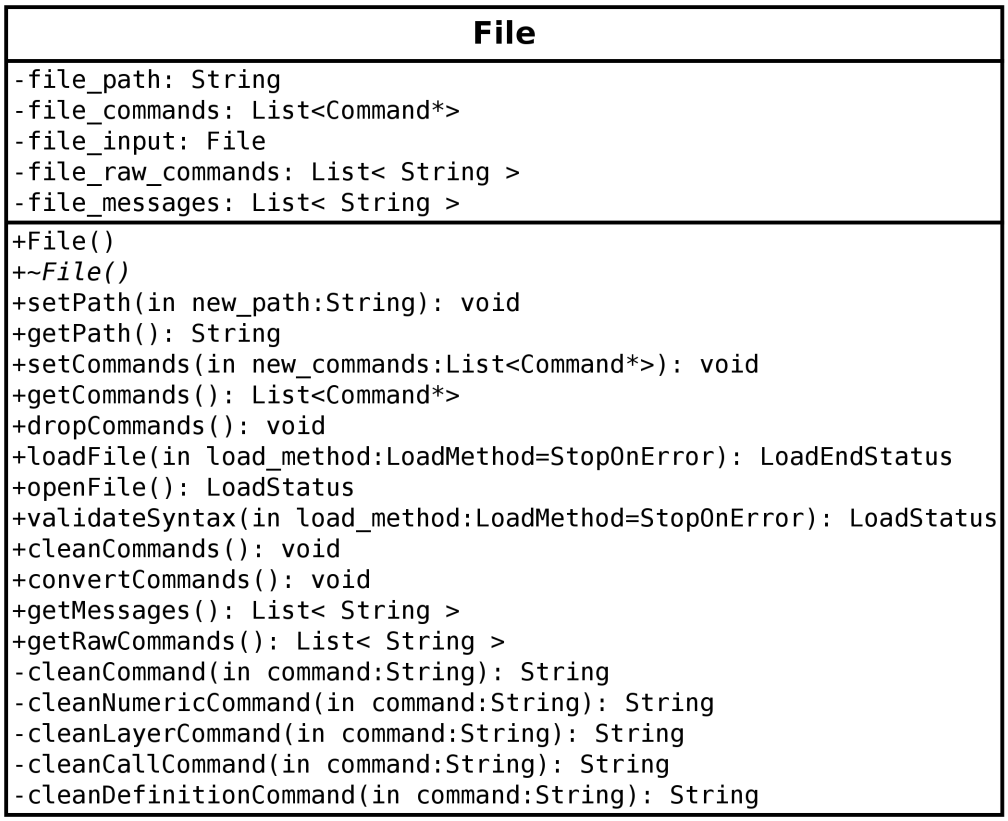


Figure 3.8: UML diagram of the **File** class.

Class: File

The File class performs the operations required to open, load, validate and transform a CIF file. The user is expected to only provide the path (full or relative) to the class and ask it to do all the job. The UML diagram found in the figure 3.8 shows the general idea behind the class. The table 3.1 shows the enum definitons found within the class itself, while the tables 3.2 and 3.3 details the interface of the class.

`LoadStatus`	Represents the final status of a loading process. • **AllOk** The loading process requested completed without errors. • **CantOpenInputFile** The opening of the input file failed (the file path might be incorrect or the file doesn't have read permissions). • **IncompleteInputFile** The validated contents are correct, but the **End** command is missing. • **IncorrectInputFile** The input file has one or more errors in its contents.
`LoadMethod`	Defines if the library should try to load as much as possible (skipping errors) or stop when an error is found. • **StopOnError** Stop the loading process if an error is found. • **ContinueOnError** Try to skip the errors found.

Table 3.1: Enum definitions in the **File** class.

`string file_path`	Stores the path to the CIF file specified by the user.
`ifstream file_input`	Used to open and access the file specified by the user once open.
`vector<Command*> file_commands`	Stores pointers to the final commands available to the user once they are converted from clean strings.
`vector<string> file_raw_commands`	Stores, in a first instance, the "dirty", but validated, commands of the CIF file. Then, when the commands are cleaned, is used to store the final and cleaned commands.
`vector<string> file_messages`	Stores all the error messages generated during the usage of the instance of the File class.

Table 3.2: Attributes of the **File** class.

`explicit File (void)`	Explicit default constructor. Doesn't initialize values.
`virtual File (void)`	Destructor. Deletes from memory the conents of **file_commands**.
`void setPath (const string&)`	Assigns to **file_path** the path to the CIF file expected to be loaded. The path is not validated when assigned. The path can be relative or full.
`string getPath (void)`	Returns the path stored in **file_path**.
`LoadStatus loadFile (const LoadMethod& = StopOnError)`	Starts the automatic loading of the input file. With the argument StopOnError the loading stops after an error is found. With ContinueOnError the loading is performed skipping incorrect commands. Returns a value of type LoadStatus according to the results.
`LoadStatus openFile (void)`	Tries to open the input file. Returns AllOk on success, CantOpenInputFile otherwize.
`LoadStatus validateSyntax (const LoadMethod& = StopOnError)`	Starts the validation of the contents of the input file. With the argument StopOnError the loading stops after and error is found. With ContinueOnError the loading is performed skipping incorrect commands. Returns a value of type LoadStatus according to the results.
`void cleanCommands (void)`	Cleans the raw commands found in the input file. The commands are taken from file_raw_commands and stored there once cleaned (the contents are replaced).
`void convertCommands (void)`	Converts the cleaned commands into class instances, storing their pointers in **file_commands**. The commands to convert must be already cleaned.
`void setCommands (const vector<Command*>&)`	Assigns to **file_commands** the vector passed as argument. The previous contents of the vector are not deleted from memory.
`vector<Command*> getCommands (void)`	Returns the value of **file_commands**.
`void dropCommands (void)`	Assigns an empty vector to **file_commands**. Doesn't delete from memory the previous contents of the vector.
`vector<string> getMessages (void)`	Returns the messages generated during the loading of the input file.
`vector<string> getRawCommands (void)`	Returns the contents of **file_raw_commands**.
`static bool isCommandValid (string)`	Validates a command (in string form) on demand. Takes as argument a command and returns true if the command is valid, false otherwize.
`static string cleanCommand (string)`	Cleans a command (in string form) on demand. Takes as argument the dirty command and returns a cleaned copy. The dirty command should be valid.

Table 3.3: Member functions of the **File** class.

Document created by Moises Chavez-Martinez •
http://moiseschavezmartinez.blogspot.com

Cover Illustration by Dusan Bicanski • http://www.public-domain-image.com