Using the SALOME configuration and building system environment

Version 0.3

Patrick Goldbronn C.E.A.
Marc Tajchman C.E.A.

Successive versions

Date	Version	Description	Author(s)
10/07/2001	0.0	Initial version	PG
25/07/2001	0.1	English traduction, rewriting	MT
29/08/2001	0.2	Add source creation, some precision	PG
24/05/2002	0.3	Add instruction to do installation correctly	PG

Abstract

This document contains rules and advices to configure, build and extend the SALOME platform.

Contents

1 SALOME Configuration

1.1 Directories organisation

We suppose here that you unpack the SALOME distribution from scratch. The path to the SALOME sources will be named “top source directory” or `SALOME_ROOT`.

It is possible, but not advised, to build the set of binaries and libraries in the same subtree. Instead, we suppose you have choosen a different subtree where to put builded files (you can so build to multiples architectures from the same source tree). The root of the build subtree will be named “top build directory”.

At the end of configuration and compilation processs, you may install builded files in a separate subtree, name “installation subtree”. The root of the installation subtree will be named “top installation directory”.

The figure 1 shows subtrees organisation.

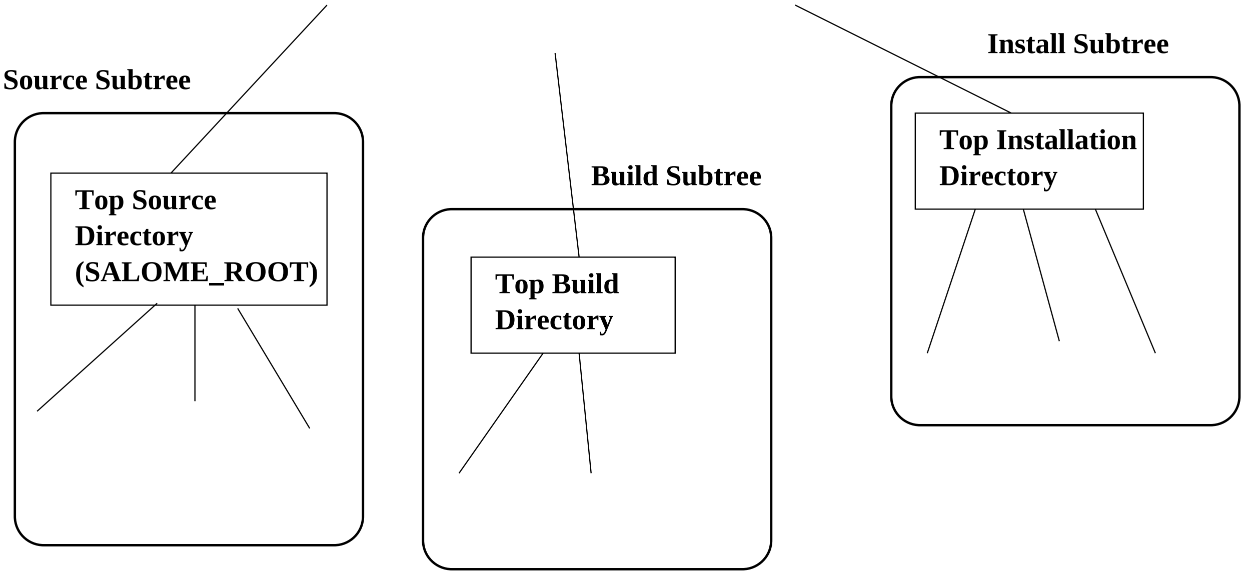

Figure 1: Subtrees organisation

1.2 PreConfiguration step

SALOME needs some environment variables (to be defined for example in a .cshrc or .bashrc file in your home directory) :

variable	*set value and check*
`QTDIR`	root directory of qt distribution (`$QTDIR/lib` must contain libqt.so)
`HDF5HOME`	root directory of hdf5 distribution (`$HDF5HOME/lib` must contain libhdf5.so)
`VTKHOME`	root directory of vtk distribution (`$VTKHOME/common` must contain libVTKCommon.so)
`CASROOT`	root directory of OpenCascade distribution (`$CASROOT/Linux/lib` must contain libTKernel.so)
`PYTHONHOME`	root directory of python distribution (`$PYTHONHOME/lib/pythonXXX/config` must contain libpythonXXX.a)
`OMNIORB_CONFIG`	path to the omniORB.cfg file (this file contains default options to omniORB, see below)

Create a file named omniORB.cfg in your root tree, containing default options to omniORB. Put in this file, the following line :

```
ORBInitRef NameService=corbaname::localhost
```

(tells omniORB that the CORBA name service is local).

1.3 Configuration step

1. There are two cases :
   - There is a `configure` file in the top source directory, and you didn't change the SALOME structure (adding a module or unit, see sections 3 or 4 below). Go to point 3.
   - You don't have a `configure` file or you add a module/unit in the SALOME system. Go to point 2
2. Go to the top source directory and type :
   ```
   ./reconfigure
   ```
   This script find all file with suffix `.in` (which will be generate by `configure` script) and add them in `configure.in` file, launch `aclocal` and `autoconf` to generete `configure` script. Continue with point 3
3. Go to the top build directory you choose.
   If you plan to install SALOME files after building in a non-standard location (i.e. different from /usr/local), type :
   ```
   <path to the top source directory>/configure \
                     --prefix=<installation directory>
   ```

otherwise, type :

```
<path to the top source directory>/configure
```

where “path to the top source directory” is to be replaced by the path to the SALOME sources.

For other options to the configure command, type :

```
<path to the top source directory>/configure --help
```

This will create a mirror subtree of the sources into the top build directory where object files, binaries and libraries will be builded. Also a makefile system will be created into the build tree.

1.4 PostConfiguration step

This phase is optional, to be used only if the compilation process (see next section) fails to use `libtool` script.

On some systems, the `libtool` script generated by the configure command will not operate correctly during compilation (see next section). If you encounter this situation, copy the local libtool script in your system (e.g. in the /usr/bin directory) to the top build directory after configuration and before compilation phases.

Check the following line in libtool script :

```
deplibs_check_method=...
```

If needed, replace this line by

```
deplibs_check_method="pass_all"
```

2 SALOME compilation

From the top build directory, type

```
make
```

After some time (be patient ...), it will create various binaries. Building SALOME is split in several phases :

- `make inc` : copy/update header files exported by development units in the directory `inc` of the build tree ;
- `make depend_idl` : determine dependencies between idl files (useful when recompiling SALOME after idl modification);
- `make depend (make dep)` : determine dependencies between source files and header files (useful when recompiling SALOME after source modification);
- `make lib` : generate libraries, put a copy/link into the `lib` directory of the build tree;

- `make bin` : generate binaries;
- `make tests (make check)` : build and run tests (not yet implemented).

After building, testing, the user may install the system in a choosen directory (different from and not included in the top source directory and the top build directory).

From the top build directory, type :

> `make install` : install libraries, header and idl files, binaries, resource files in the installation directory

3 Module creation

In this section, the new module will be named `<Module>`. Replace each occurence with the real name of your module.

1. In the source tree root `SALOME_ROOT`, create a new directory `<Module>` :
   ```
   cd SALOME_ROOT
   mkdir <Module>
   ```
2. Modify the `Makefile.in` file in the `SALOME_ROOT` directory to add the new module :
   Append to the line beginning with
   ```
   SUBDIRS =
   ```
   the name of the new module.
3. In the module root directory, create two subdirectories `src` and `resources` and create a file `Makefile.in` (e.g. copy the corresponding file in `GEOM` module for example) :
   ```
   cd <Module>
   mkdir src
   mkdir resources
   cp ../GEOM/Makefile.in .
   ```
4. In the `src` subdirectory, copy a `Makefile.in` file (e.g. from the corresponding file in `GEOM/src` subdirectory for example) :
   ```
   cd src
   cp ../../GEOM/src/Makefile.in .
   ```
5. Edit this file and replace the line
   ```
   MODULE = GEOM
   ```
   with
   ```
   MODULE = <Module>
   ```
6. Edit this file and replace the line
   ```
   SUBDIRS = GEOMDS GEOM GEOMGUI
   ```
   with
   ```
   SUBDIRS =
   ```

(empty list of development units in this module).

7. Edit this file and replace the line

   ```
   RESOURCES_FILES = arc.png \
   ...
   ```

   with

   ```
   RESOURCES_FILES =
   ```

   (list of all ressources for this module).

8. Add the new `Makefile.in` files in the global list of .in files.

   In the root directory of the source tree, execute the `reconfigure` script or manually :

   (a) edit the configure.in file in the source tree root, add `Makefile.in` files into the `AC_OUTPUT` list,

   (b) from the source tree root directory, run the `genconf` script which launch `aclocal` and `autoconf`.

Figure 2 summarize these changes.

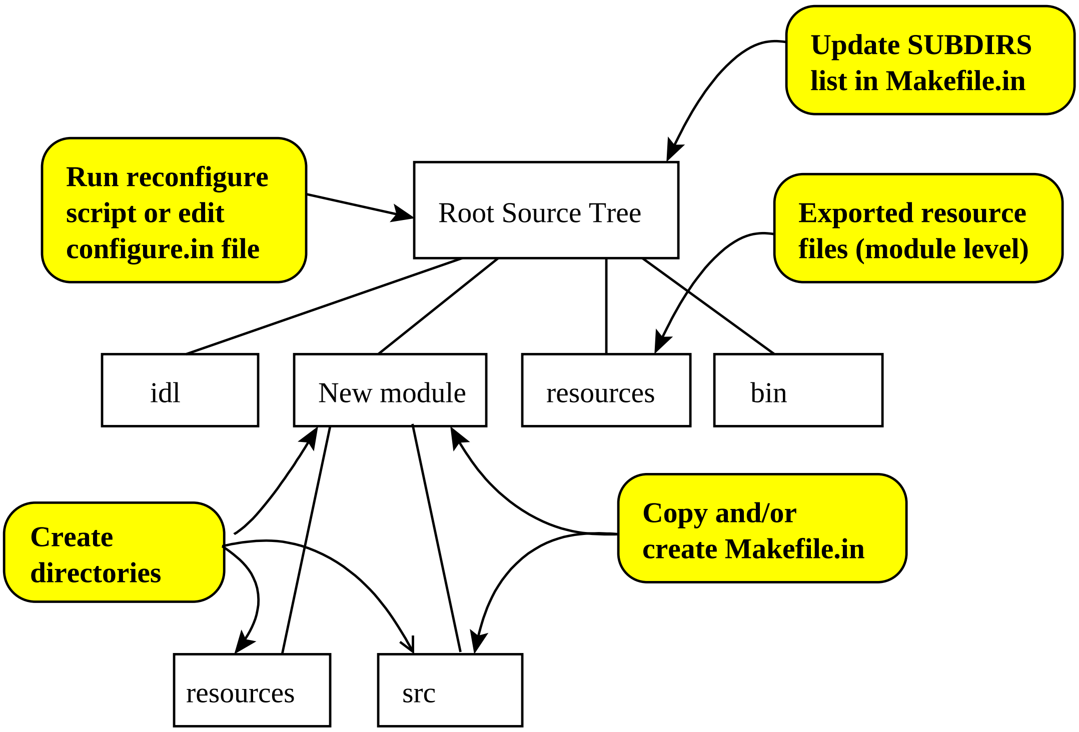
Update SUBDIRS list in Makefile.in
Run reconfigure script or edit configure.in file
Root Source Tree
Exported resource files (module level)
idl
New module
resources
bin
Create directories
Copy and/or create Makefile.in
resources
src

Figure 2: Source tree : modification when adding an new module

4 Development unit creation

Here we want to add a development unit named `<Unit>` in the existing module `<Module>` (replace the names `<Unit>` and `<Module>` with real ones).

1. In the src subdirectory of <Module>, create a subdirectory named <Unit> :

   ```
   cd <path to <Module> >/src
   mkdir <Unit>
   ```

   Modify then Makefile.in file in the src directory to add the new unit to the compilation process :

   Complete the line beginning with

   ```
   SUBDIRS = ...
   ```

   with the name of the new directory

   ```
   SUBDIRS = ... <Unit>
   ```

2. Create a Makefile.in file in the new <Unit> directory (you can copy a Makefile.in file from the corresponding subdirectory in GEOM module : GEOM/src/GEOMGUI subdirectory for example, and modify as you need)

   ```
   cd <Unit>
   ... create Makefile.in
   ```

   The details of Makefile.in creation is detailed in the next section.

The different files of your unit must be located in several directories (see figure 3 and the list below).

- Private source and header files of your unit

  Place the only copy of these files in your unit. If you use the proposed makefile system, dont put them in subdirectories of your unit.

  Note

  > Using a non-flat directory structure for an unit, has not been tested but it should work. You must write your makefile to take care of subdirectories.

- Exported idl files from a unit

  These files are provided by the unit for CORBA communication with other units.

  Place the only copy of these files into the idl subdirectory of the root source tree.

- Exported header files from a unit

  These files are provided by the unit for direct communication from other units (using the unit's library).

  Place the master copy of these files in your unit subtree.

  Assure that these files are automatically or manually copied in the inc subdirectory of the root build tree.

5 Creating a Makefile.in file in a new unit

5.1 Using predefined make rules

Copy the following Makefile.in skeleton in the unit directory :

```
# begin copy here ==========================================
```

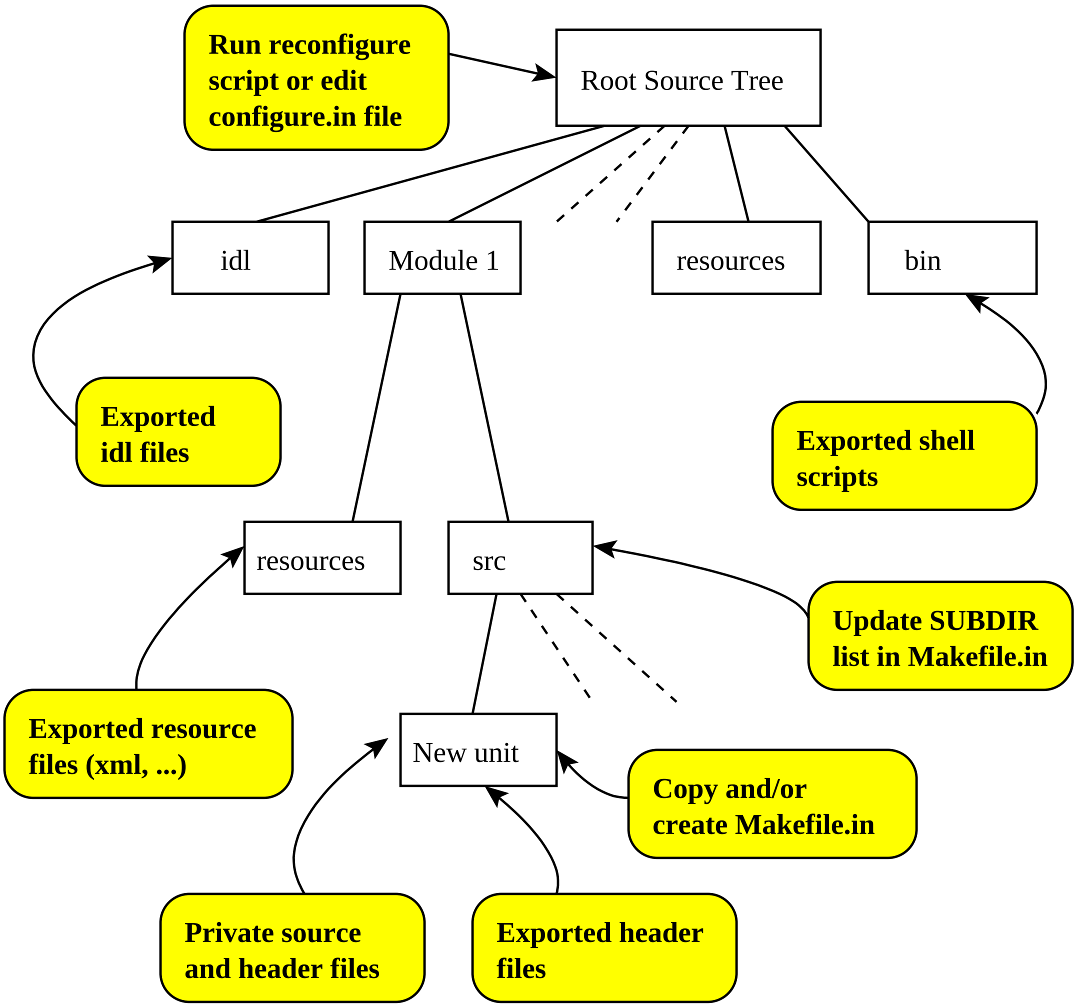
Run reconfigure script or edit configure.in file
Root Source Tree
idl
Module 1
resources
bin
Exported idl files
Exported shell scripts
resources
src
Update SUBDIR list in Makefile.in
Exported resource files (xml, ...)
New unit
Copy and/or create Makefile.in
Private source and header files
Exported header files

Figure 3: Source tree : modification when adding an new unit in an existing module

```
top_srcdir=@top_srcdir@
top_builddir=../../..
srcdir=@srcdir@
VPATH=.:@srcdir@

@COMMENCE@

# Libraries targets
LIB =
LIB_SRC =
LIB_MOC =
LIB_CLIENT_IDL =
LIB_SERVER_IDL =
# Executable targets
BIN =
BIN_SRC =
BIN_MOC =
BIN_CLIENT_IDL =
BIN_SERVER_IDL =

# exported header files
EXPORT_HEADERS =

# exported python executable files
EXPORT_PYSCRIPTS =

# list of files in resources directory (copy when do make install)
RESOURCES_FILES =

# po ressources files (to transform them in qm file) :
PO_FILES =

# put here additional rules, or extra compiler options ...

@CONCLUDE@

# end copy here ============================================
```

Adapt this `Makefile.in` skeleton to your particular needs :

- if you have to compile a library
  1. Complete the line

     `LIB =`
     as

     `LIB = lib<MyLibrary>.la`
     Example :

     `LIB = libGeometryGui.la`
     Notes

     (a) the library name **must** begin with `lib` and end with `.la` (this allows automatic creation of shared libraries with libtool).

     (b) there must be only one library by development unit

  2. Also add to the line :

     `LIB_SRC =`
     the list of sources files (in this unit) needed to build the library

  3. If your library uses QT MOC file, add to the line :

```
    LIB_MOC =
    ```
    the list of headers files to transform with moc.

4. If your library uses CORBA functionnalities from other units (i.e. uses idl files exported from other units), add to the line :

    ```
    LIB_CLIENT_IDL =
    ```
    the list of idl files.

5. If your unit provides CORBA functionnalities (i.e. exports idl files to the other units), add to the line :

    ```
    LIB_SERVER_IDL =
    ```
    the list of idl files.

- if you want to build one or more executables :

    1. Complete the line

        ```
        BIN =
        ```
        as

        ```
        BIN = <MyBin1> <MyBin2> ..
        .
        ```
        Note

        > For each executable in the `BIN` list, say `MyBin1`, the main function **must** be in a file named accordingly, in this example : `MyBin1.cxx` and `MyBin2.cxx`.

    2. Also add to the line :

        ```
        BIN_SRC =
        ```
        the list of source files (in this unit) needed to build **all** the executables, **excluding files containing main function(s)**.

        Notes :

        (a) The makefile system will automatically add to each executable, its main function file. That's why these files must not be included in the `BIN_SRC` list

        (b) The object files (compiled from the source files in the `BIN_SRC` list) will be properly dispatched between the executables by the linker.

    3. If your binaries uses QT MOC file, add to the line :

        ```
        BIN_MOC =
        ```
        the list of headers files to transform with moc.

    4. If your binaries uses CORBA functionnalities from other units (i.e. uses idl files exported from other units), add to the line :

        ```
        BIN_CLIENT_IDL =
        ```
        the list of idl files.

    5. If your unit provides CORBA functionnalities (i.e. exports idl files to the other units), add to the line :

        ```
        BIN_SERVER_IDL =
        ```
        the list of idl files.

- List the exported header files that your unit provides to other developments units :
    Complete the line

`EXPORT_HEADERS =`
with the list header files.

Note

> The makefile system will automatically copy these files in a subdirectory `inc` in the top build directory, and maintain coherence with your private copy inside your unit subtree. This is to assure name uniqueness of differents exported header files from different units and to write easier makefiles.

- List the python scripts files that your unit export :
  Complete the line
  `EXPORT_PYSCRIPTS =`
- To generate qm file from po file (use by QT), list po files in :
  `PO_FILES =`
  Note

  > The resulting qm files will ge generated directory which contain Makefile. It will be copied in resources directory when do `'make install'`.

5.2 Using your own makefiles in an unit

If the proposed makefile system don't suit your needs (several libraries, non flat unit subtree structure, ...). It's possible to write your own makefiles.

1. Create a file `Makefile.in`
   This file must begin with the lines

   ```
   # begin copy here ==========================================

   top_srcdir=@top_srcdir@
   top_builddir=../../..
   srcdir=@srcdir@
   VPATH=.:@srcdir@

   @COMMENCE@

   # end copy here ============================================
   ```

   The rest of the file has the standard GNU make format.
   You must define the following targets :
   (a) `inc` : copy/update the exported header files to the `$top_builddir/inc` directory
   (b) `dep` : update dependencies
   (c) `lib` : build libraries and link them into the `$top_builddir/lib` directory
   (d) `bin` : build executables and link them into the `$top_builddir/bin` directory
   Some of these targets may be empty, if not applicable.

The line

```
@\texttt{COMMENCE}@
```

provides a number of predefined variables that you can use in your makefile rules (defining standard libraries locations, compiler options, ..., see next section).

6 Add or remove a script

If you want to add a new shell script in `SALOME_ROOT/bin`, you must edit `SALOME_ROOT/Makefile.in` to add it in `BIN_SCRIPT`.

If this script have some package dependent variable, you must create a ".in" file and add this reference to `configure.in` file.

To remove an existing script, you must of course remove it from CVS archive and also remove it from `SALOME_ROOT/Makefile.in` and if any, from `configure.in`.

If you want to add a new python script, put it in `EXPORT_PYSCRIPTS` variable. It will be copied at same place than others executables.

7 Add or remove an IDL file

If you want to add a new IDL file in `SALOME_ROOT/idl`, you must edit `SALOME_ROOT/idl/Makefile.in` and add its in `IDL_FILES`.

To remove an existing IDL file, you must of course remove it from CVS archive and also remove it from `SALOME_ROOT/idl/Makefile.in`.

8 Predefined symbols used in `Makefile.in`

You can use predefined symbols in you `Makefile.in` files. These symbols define

- compilation flags for source compiling,
- header files location in your local system,
- libraries needed for binaries linking.

For example to use the OpenCascade libraries in your unit, you will add the

- `$OCC_INCLUDES` symbol to the included header file locations,
- `$OCC_CXXFLAGS` symbol to the compilation flags,

- `$OCC_LIBS` symbol to the linker's flags

If you use the predefined make rules, add the lines

```
CPPFLAGS+=$(OCC_INCLUDES)
CXXFLAGS+=$(OCC_CXXFLAGS)
LDFLAGS+=$(OCC_LIBS)
```

in your `Makefile.in` file after the @`COMMENCE`@ line.

For each standard tool you need in SALOME (QT, python, OpenCascade, CORBA, VTK, . . .), main symbols listed below.

1. *Corba*

variable	*value*
`CORBA_ROOT`	CORBA home base
`CORBA_INCLUDES`	compiler options to include CORBA headers
`CORBA_LIBS`	libraries needed to link with CORBA
`CORBA_CXXFLAGS`	C++ compiler options to use with CORBA
`IDL`	idl compiler
`IDLCXXFLAGS`	options to the idl compiler to generate C++ stub or skeleton code
`IDLPYFLAGS`	options to the idl compiler to generate python stub or skeleton code
`IDL_CLN_H`	extension of generated CORBA header files (client side)
`IDL_CLN_CXX`	extension of generated CORBA source files (client side)
`IDL_CLN_OBJ`	extension of generated CORBA object files (client side)
`IDL_SRV_H`	extension of generated CORBA header files (server side)
`IDL_SRV_CXX`	extension of generated CORBA source files (server side)
`IDL_SRV_OBJ`	extension of generated CORBA object files (server side)

2. *python*

variable	*value*
`PYTHON`	python interpreter (absolute path to)
`PYTHON_VERSION`	python version
`PYTHONHOME`	python home base (sometimes needed to run python)
`PYTHON_INCLUDES`	compiler options to include python header files
`PYTHON_LIBS`	libraries needed to link with python

3. *QT*

variable	*value*
`MOC`	moc compiler
`UIC`	uic graphical compiler
`QTDIR`	QT home base
`QT_ROOT`	QT home base
`QT_INCLUDES`	compiler options to include QT headers
`QT_MT_INCLUDES`	same as above, for multithreaded applications
`QT_LIBS`	libraries needed to link with QT (single threaded)
`QT_MT_LIBS`	same as above, for multithreaded applications

For SALOME developments, multithreaded versions of qt options and libraries are needed.

4. *OpenGL*

variable	*value*
`OGL_INCLUDES`	compiler options to include OpenGL headers
`OGL_LIBS`	libraries needed to link with OpenGL

5. *VTK*

variable	*value*
`VTK_INCLUDES`	compiler options to include VTK headers
`VTK_LIBS`	libraries needed to link with VTK

6. *HDF (v5)*

variable	*value*
`HDF5_INCLUDES`	compiler options to include HDF headers
`HDF5_LIBS`	libraries needed to link with HDF
`HDF5_MT_LIBS`	libraries needed to link with HDF (multithreaded version)

7. *OpenCascade*

variable	*value*
`OCC_INCLUDES`	compiler options to include OpenCascade headers
`OCC_LIBS`	libraries needed to link with OpenCascade
`OCC_CXXFLAGS`	C++ compiler options to use with OpenCascade

9 Location of generated files in the build tree

A partial view of the build tree shows the location of files generated during the compilation process.

10 What's matter when launch `make install`

When all libraries and binaries files are generated, make copies all identified files as `configure` parameters `--prefix`, `bindir`, `datadir`, ... (see `configure --help` for details).

If you specify nothing, all are installed in `<prefix>=/usr/local`.

All executables (binaries and scripts) are placed in `<prefix>/bin` (see BIN and BIN_SCRIPT variables in `Makefile`).

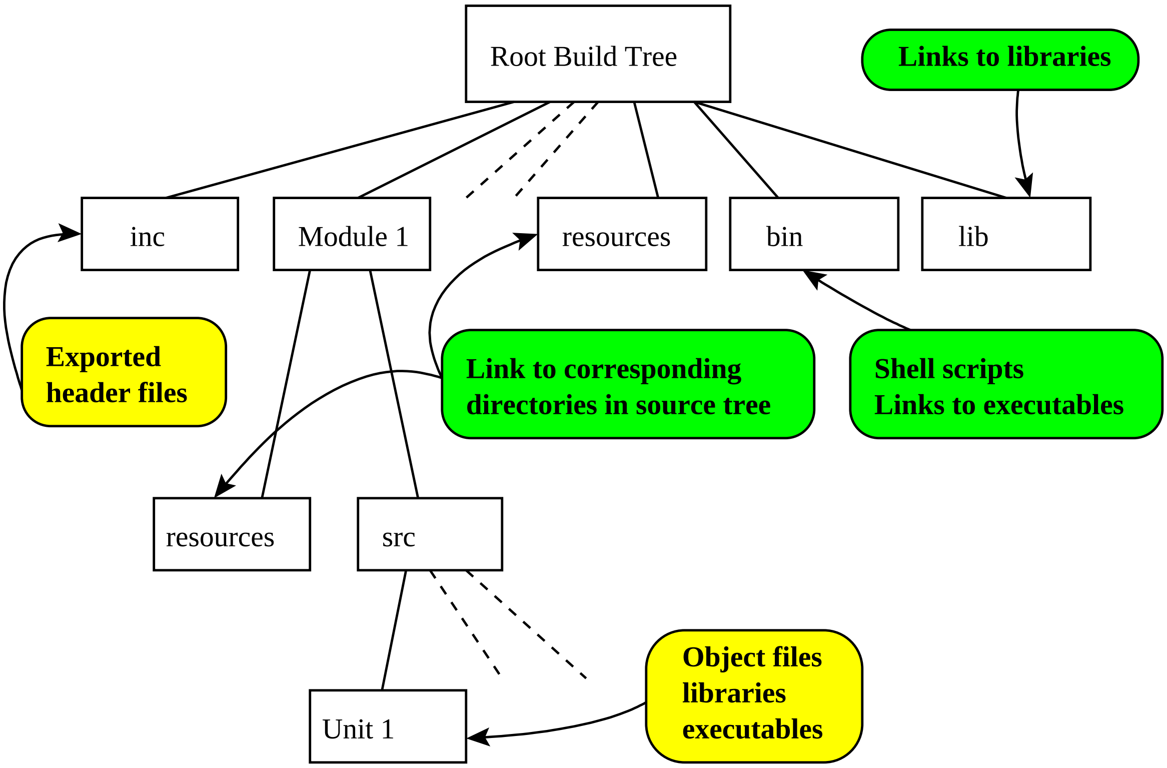
Root Build Tree
Links to libraries
inc
Module 1
resources
bin
lib
Exported header files
Link to corresponding directories in source tree
Shell scripts Links to executables
resources
src
Unit 1
Object files libraries executables

Figure 4: Partial view of the build tree : generated files during compilation

All libraries are placed in `<prefix>/lib` (see LIB variable in `Makefile`).

All includes are placed in `<prefix>/include` (see EXPORT_HEADERS variable in `Makefile`).

All idls are placed in `<prefix>/idl` (see IDL_FILES variable in `Makefile`).

All python srcipts are placed in `<prefix>/lib/python2.1/...` (see `EXPORT_PYSCRIPTS` variable in `Makefile`).

All ressources files (icons, messages, configuration, ...) are placed in `<prefix>/share/salome/ressources` (see RESOURCES_FILES variable in `Makefile`).

11 Creating source files according to SALOME building system

Building system use dependencies between files writing in Makefile rules. We use `C` or `C++` preprocessor to automatically generate this dependencies rules.

There are some configuration and useful macro defined in header file `SALOMEconfig.h`. **All files should be included this header !** You must include it ussing <> delimiter because `SALOMEconfig.h` must not appear in dependencies rules (see below 11.1).

When a `Makefile` is regenerate with `config.status` script, all files are regenerates (in particular `SALOMEconfig.h`).It is a restriction of `autoconf 2.13` which could not regenerate only one particular file. So, all files which depend of `SALOMEconfig.h` are rebuild even if it does not change. If you effectively change `SALOMEconfig.h` file, you must clean all and rebuild.

11.1 C or C++ source files

You must name your C file `<myCFile>.c` **and header file** `<myCHeaderFile>.h`

You must name your C++ file `<myC++File>.cxx` **and header file** `<myC++HeaderFile>.hxx`

To have right dependencies rules, you must correctly write the include statement in your source files. We only take care about SALOME package header files to generate dependencies. We suppose that other header files (qt, vtk, OpenCascade, ...) are stables and are not modified when we build some SALOME modules.

According to cpp documentation, local header files must be included with `""` statement and system or tools headers files must be included with <> statement.

If you do not respect this notation, dependencies would not be true and some rebuilding trouble can appear !

11.2 idl files

We use C preprocessor to build dependencies between idl files. The same convention must be applied as C or C++ source files.

If included file is an external files, you must use statement <> because this file will not be modified during SALOME devloppement and/or building. If included file is part of SALOME files, you must use statement `""`.

If you do not respect this notation, dependencies would not be true and some building or rebuilding trouble can appear !

11.3 Included header file generated from idl file

To include header file generated from idl file, you must use macro CORBA_CLIENT_HEADER or CORBA_SERVER_HEADER defined in `SALOMEconfig.h`.

These two macros replace idl prefix into corresponding header name generated (take care if you use client part or server part)

Example :

```
#include  CORBA_CLIENT_HEADER(geom)
#include  CORBA_SERVER_HEADER(mesh)
```