

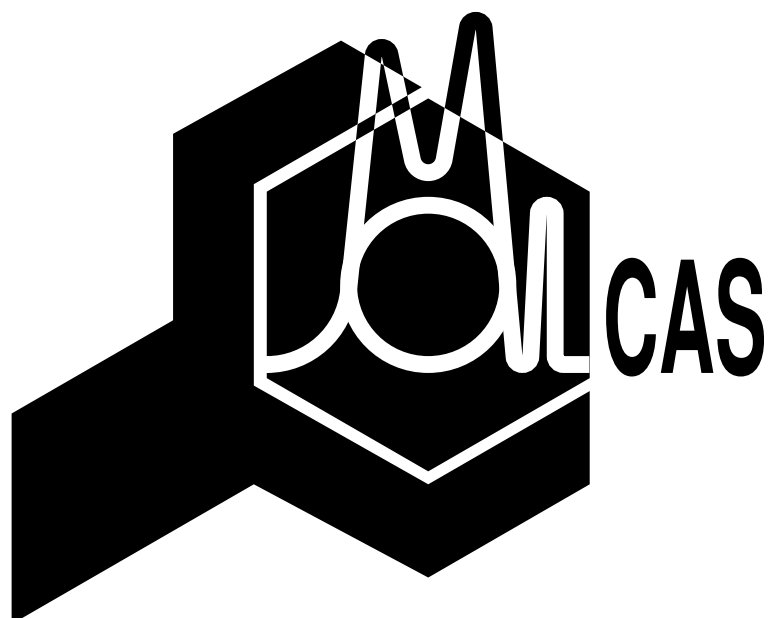
---

MOLCAS version 7.6

Molcas Programming Guide

Krogh, J. W., Lindh, R., Malmqvist, P.-Å., Roos, B. O.,  
Veryazov, V., Widmark, P.-O.

---



---

©Lund University 2003



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Who should read this . . . . .	1
1.2	Developers' home page . . . . .	1
<b>2</b>	<b>General File Structure</b>	<b>3</b>
2.1	Versions . . . . .	3
2.2	Directory structure . . . . .	3
2.3	Internal Tools . . . . .	5
2.4	Multiple versions of MOLCAS . . . . .	7
<b>3</b>	<b>Building Molcas</b>	<b>9</b>
3.1	The configuration file . . . . .	10
3.2	Generation of Makefiles . . . . .	11
3.3	What happens, when you type <i>make</i> . . . . .	12
3.4	Preprocessing the code . . . . .	13
3.5	The structure of prgm-files . . . . .	14
3.6	Packages . . . . .	15
3.7	Examples of package definition files . . . . .	16
3.8	Rebuilding MOLCAS . . . . .	18
3.9	Installation of GUI . . . . .	19
<b>4</b>	<b>The Patch System</b>	<b>23</b>
4.1	Why use a patch system? . . . . .	23
4.2	How the patch system works . . . . .	24
4.3	Make reservation for a patch . . . . .	25
4.4	Making the patch available to all . . . . .	31
4.5	Retrieving of patches . . . . .	31

<b>5</b>	<b>Verification</b>	<b>33</b>
5.1	How does it work? . . . . .	34
5.2	How to generate a test . . . . .	35
5.3	How to add new information for checking . . . . .	35
5.4	Conventions . . . . .	36
5.5	Performance tests . . . . .	36
5.6	Automatic verification . . . . .	37
<b>6</b>	<b>Documentation</b>	<b>41</b>
6.1	Users guide . . . . .	41
6.2	Examples in manual . . . . .	42
6.3	Source code documentation . . . . .	43
6.4	XML formatted description of keywords . . . . .	44
<b>7</b>	<b>Use of Utilities</b>	<b>51</b>
7.1	List of utilities . . . . .	51
7.2	Memory manager . . . . .	52
7.3	I/O Utilities . . . . .	54
7.4	Parallelisation . . . . .	55
7.5	Cholesky decomposition . . . . .	59
<b>8</b>	<b>Creation of a New Program</b>	<b>65</b>
8.1	Your first MOLCAS program . . . . .	65
<b>9</b>	<b>Debugging and Code Developing</b>	<b>67</b>
9.1	Running through a debugger . . . . .	67
9.2	Other ways to debug . . . . .	67
9.3	Core generation . . . . .	68
9.4	Parallel debugging . . . . .	68
9.5	Problems with aggressive optimization . . . . .	69
9.6	Utilities . . . . .	69
9.7	Memory allocation profiling . . . . .	69
9.8	How to check inter procedure consistency . . . . .	70

<b>A</b>	<b>Makefiles</b>	<b>71</b>
A.1	How to define a target . . . . .	71
A.2	00sources . . . . .	72
A.3	00dependencies . . . . .	73
A.4	stdsuffix . . . . .	73
<b>B</b>	<b>Glossary</b>	<b>75</b>



# Section 1

## Introduction

### 1.1 Who should read this

This manual is addressed to those who intend to modify MOLCAS source code. Also this document is useful for advanced users who would like to tune MOLCAS code. The reader is assumed to be well acquainted with the programming languages and operating systems.

- People supplying external packages.
- MOLCAS developers.
- MOLCAS administrators.

### 1.2 Developers' home page

All developers of the MOLCAS code can (and should) get access to the to the Developers' home page, <http://www.teokem.lu.se/molcas/dev>. If you do not already have access to the home page, you should contact the MOLCAS group. On this page, you can always obtain the newest version of the program, retrieve and upload patches, see the status of the verification runs, get access to bug reports, etc. These webpages are self documented, however some aspects will also be discussed on this guide.



## Section 2

# General File Structure

This section describes the MOLCAS structure such as conventions about naming of the version, and the directory structure.

### 2.1 Versions

The MOLCAS version numbers consist of a version number, a release number, and patch level, e.g. 6.0.007 where 6 is the version number, 0 is the release number, and 007 is the patch level. The release number has an additional meaning other than just for counting. An even revision number specifies a distribution (stable) version, whereas an odd revision number specifies an developer (unstable) version. The only changes that is allowed on the stable version are bug fixes - it is not allowed to introduce new features and programs into the stable version!

On the other hand it is allowed to introduce new code into the developer version. A revision with an odd number will never be released - it will be renumbered instead, e.g. the developer version 6.1 will be released as revision 6.2.

The revision numbers are also written into the file headers (see section 4.2 for details on the header), which help preventing mixing of files from two different revisions.

### 2.2 Directory structure

Below is a list of the directories and their contents. All directories are listed relative to the MOLCAS main directory, which as default will be *\$MOLCASHOME/7.6.dev/*, if one is working on the 7.6 revision. For the distribution version the top MOLCAS directory is renamed to *molcas76/*; however the content of this directory is identical to *7.6.dev/*. Some of the directories will be examined more throughly in a later section.

Before we start defining the characteristics of the MOLCAS environment, we need to take briefly a look at the general structure of the directory created after the installation. Plain files and directories are included into the main directory, for instance *7.6.dev/*, with the names:

<i>File</i>	<i>Contents</i>
<i>configure</i>	This file is a script that is used to check the system (operating system, compilers, etc.) and prepare the Makefiles.
<i>Makefile.in</i>	The template for the top level <i>Makefile</i> .
<i>Symbols</i>	Contains the translation between the symbolic names used in, e.g. <i>Makefile</i> and the real names. This file is generated by <i>configure</i> . <i>Symbols.bak</i> is the backup for a replaced <i>Symbols</i> file.
<i>molcas.rte</i>	File contains runtime environment used by molcas scripts
<i>basis_library/</i>	Contains the basis set information.
<i>bin/</i>	Contains the binary executables and some shell scripts. This directory is created during installation.
<i>cfg/</i>	Contains the configuration files for each specific platform. the <i>configure</i> script will pick up the configuration file for this machine and compiler (if necessary). All special keys, flags, location of compilers, etc. are taken from the configuration file.
<i>data/</i>	Some database files generated during installation.
<i>doc/</i>	The MOLCAS documentation, e.g. the manual, is located in this directory. The detailed description of this directory is in section 6.
<i>g/</i>	The GLOBAL ARRAYS ( <a href="http://www.emsl.pnl.gov/docs/global/ga.html">http://www.emsl.pnl.gov/docs/global/ga.html</a> ), i.e. memory manager and parallel stuff. GLOBAL ARRAYS is a separate program/package that is distributed together with MOLCAS. Usually one should not need to worry about GLOBAL ARRAYS.
<i>lib/</i>	This directory is also generated during build. It will contain the the common library ( <i>libmolcas.a</i> ) and libraries for each module (e.g. <i>libseward.a</i> ).
<i>packages/</i>	MOLCAS supports packages/plugin made by e.g. external developers. These packages will be located here. A package will contain the necessarily source code and documentation needed for the module delivered by the package. Packages are described in further detail in section 3.6.
<i>nopackages/</i>	A “fake” directory, which is only used in the developer version. It could contain some packages that one does not wish to compile for the moment being.
<i>patch/</i>	Contains the information about patches. By using the information in this directory and its subdirectories one can undo changes and read the <i>README</i> files that comes with all patches. See section 4.2 for details.
<i>Test/</i>	Contains the files needed for verification of MOLCAS. Detailed description can be found in section 5.

<i>shell/</i>	Contains some executable scripts like <i>molcas.sh</i> and <i>run.sh</i> .
<i>sbin/</i>	Contains scripts which are usually system scripts. Scripts in <i>sbin/</i> are not preprocessed (unlike the scripts in <i>bin/</i> ), so the scripts can be used without building MOLCAS. The scripts are either written in Bourne shell or Perl.
<i>src/</i>	Contains the source code and other informations used during the build of MOLCAS. All source code must be written in FORTRAN 77 or C. There are three different classes of subdirectories: 'Special', 'Utilities', and 'Modules'.

**Special** The naming of the special directories starts with an upper case letter, and the directories include a file called *Contents* (text file with a short description of the files in this directory). *src/Driver/* contains templates for everything used during building of MOLCAS. These directories always contain a *Makefile.in* (which is used to create the real *Makefile*).

**Utilities** The name always ends in "util".

The meaning of the utilities should be read from the name of the directory, e.g. *src/memory\_util/* contains files for memory handling. During compilation all files in the utility directories go to the MOLCAS library (*lib/libmolcas.a*) and can be used by all the programs (and other utilities). The *Makefile* is generated from the template *src/Driver/utildefault\_Makefile.in* if *Makefile.in* is missing. The detailed description of the utilities can be found in section 7.

**Modules** The name of the directories is always in lower case and is not ending in "util". The directories include a *main.f* or *main.c*. The *Makefile* is generated from *src/Driver/progdefault\_Makefile.in* if *Makefile.in* is missing. The library file is stored in the *lib/* directory.

## 2.3 Internal Tools

It's noteworthy to mention some utilities provided by the MOLCAS environment to deal with MOLCAS code. These utilities are located in the *sbin/* directory.

<i>File</i>	<i>Contents</i>
<i>buildpatch</i>	This a script to create a new patch. See Chapter 4.3.1 for details.
<i>copy</i>	A script to create a copy of MOLCAS source tree
<i>checksum</i>	A script to generate/compare checksums of molcas source file (even across different versions). See 2.4 for further information

<i>defdoc</i>	Script for creation of the template for source code documentation. See 6.3 for further information.
<i>edit</i>	A script to get fast access to a file or a subroutine in the MOLCAS source. See Section 9.6
<i>extract</i>	Generates the source code documentation from the FORTRAN source code. This script is called automatically during the installation of this guide.
<i>find</i>	A script to locate a subroutine or a file in the MOLCAS source. See Section 9.6
<i>firstrun</i>	A script called during the installation of MOLCAS. This will run the first verification test and generate some database files.
<i>getpatch</i>	Used to retrieve patches from the MOLCAS home page. See Section 4.5.
<i>help</i>	A code of help system. The script can be used to get help for keyword usage in molcas, and to find a source code documentation (6.3)
<i>install</i>	A script to create a new module or install a package. See Sections 3.6 and 8.1.
<i>mem_ana</i>	A script to analyze memory reports 9.7
<i>monolith</i>	A script that will make monoliths out of module source files as well as utility source files. The purpose it to be able to check if subroutine calls are consistent as well as other checks. See section 9.8.
<i>police</i>	A script to check violations of molcas programming rules
<i>revert</i>	A script to revert patches. See also Chapter 4.
<i>snooper</i>	A script to locate a file which require detuning. See Section 9.5.
<i>src_help</i>	A script to search and extract source code documentation.
<i>timing</i>	A script to create a report with benchmark timings. See Section 5.5.
<i>uninclude.plx</i>	A script which inlining all include files into a source. See Section 9.6 for more information
<i>uninstall</i>	A script to create a package tar-file from an installed module. See Section 3.6.
<i>verify</i>	A script to perform a verification test of the installation. See Chapter 5.

---

All these scripts should be run through the MOLCAS environment, e.g.

```
molcas getpatch [flags]
```

A detailed description of all MOLCAS tools could be obtained via command `molcas name_of_script -h`.

## 2.4 Multiple versions of Molcas

Often a developer ends up with several versions of MOLCAS. This can for example be a copy of the master version for productive calculation, and a developing version (of both the stable and unstable revisions). So one needs a way to define which version should be used, when the `molcas` script is called.

In MOLCAS there are several ways to do this:

- One can redefine the environmental variable `MOLCAS`. The directory assigned to `MOLCAS` will then be considered as the MOLCAS root directory.
- One can stay in a subdirectory (up to 3rd level) of the MOLCAS root directory. The `molcas` script will then overwrite the `MOLCAS` environmental variable and use the version in which tree you are standing.
- If `MOLCAS` variable is not defined, and current directory is outside MOLCAS tree, the last installed MOLCAS will be used.

Because it is so easy to maintain several versions of MOLCAS, one should never do developing on a working code.

The simplest way to create another copy of installed MOLCAS is an execution of a command `molcas copy new_directory_name`. The script creates a new MOLCAS installation with proper timestamps.



## Section 3

# Building Molcas

If you make development of MOLCAS on Linux platform you can fetch prebuild (exe) version of the code from the developers homepage. This is fully functional version, build every day, which allows to continue development (submit patches, or revert them) immediately after unpacking the file.

In more general situations (you make a development on another platform, or you have very different set of compiler libraries under Linux), you should download the source code, and install it.

For the normal user, building MOLCAS just involves two steps:

```
./configure or ./configure -setup  
make
```

Normally `configure` does not require any flags. However, as a developer it is better to know some more details of what's going on, and which flags can be used to customize the installation.

The flags which are specific for administrators and programmers are:

---

---

<i>File</i>	<i>Contents</i>
<code>-compiler COMPILER</code>	Specify type of compiler. There must be a corresponding configuration file <code>cfg/COMPILER.comp</code> .
<code>-demo</code>	Build a demo version of MOLCAS. This build will allow users to do calculations using MOLCAS, although there will be some restrictions.
<code>-parallel MSGPASS</code>	Make a parallel installation with MSGPASS as the message passing library. See also the installation guide.
<code>-fake_distributed</code>	An option which can be used only in combination with <code>-PARALLEL</code> in order to build a parallel version of molcas, which can run on a single CPU computer.
<code>-path PATH</code>	Specify an extra path to the search path of <code>configure</code> . This option can be useful to build MOLCAS using different versions of compilers.

<code>-profiling</code>	Build MOLCAS with profiling, e.g. using the <code>-pg</code> flag when building with the Gnu compiler. Currently this option only exists when using the Gnu compiler.
<code>-speed SPEED</code>	Choose to what extent you want to optimize the code. The default value for <code>SPEED</code> is 'safe' which gives reasonable optimized code (i.e. <code>-O2</code> for most compilers). 'fast' increases the optimization level if possible, and 'debug' add informations used by debuggers.
<code>-debug</code>	Switch on debug statements in the code. See Chapter 9.
<code>-trace</code>	Switch on tracing of calling routines. See Chapter 9.
<code>-bound</code>	To build a version with boundary violation checks (this version currently works only with g95 compiler, and require a manual editing of <code>WrkSpc.inc</code> file)
<code>-garble</code>	Compile a version with memory garbling

---

The first step during the configuration involves locating programs, e.g. the compilers, and other standard commands. This is done in the way that `configure` tries to figure out what system it is, and then it will use the configuration file (located in `cfg/` that has been made for this system. Alternatively one can specify the operating system, compiler, etc.

### 3.1 The configuration file

`Configure` uses the files located in the `cfg` directory. There is a `.cfg` for each supported operating system and a `.comp` for each supported compiler.

There will be two passes of the configuration file:

1. The paths in which to search for the programs are set. These are predefined according to what is normal for the operation system at hand.
2. Now the configuration file tries to find a number of settings, e.g. the compiler flags:

**F77FLAGS** are the flags used for the FORTRAN 77 compiler.

**CFLAGS** are the flags used for the C compiler.

**CPPFLAGS** are the flags used by the preprocessor.

**LDFLAGS** are the flags used by the linker.

The configuration script will analyze the configuration file, and it will generate the file `Symbols`, which is placed in the MOLCAS root directory and contains all the information needed, i.e. which programs to use, compiler flags, and some more variables. Runtime environment will be saved to the file `molcas.rte`.

Now we have reached the same point in the building process, as one would end at, if one specified the '-Symbols' flag to the `configure` script (which is not recommended).

The complete list of flags for `configure` one can get running `configure -help`.

Different *configure* flags are analyzed in corresponding *.cfg* file. For example, flag '-fast' changes 'OPT' variable in configuration script, so it contains compilers flags for aggressive optimization.

Especially as a developer it can be necessary to apply a couple of tricks to make MOLCAS compile the way you want it.

If some of the programs, like GNU make, or a non-standard compiler is located in an unusual place, these directories should be listed in the environmental variable 'MOLCAS\_EXE\_PATH'. Alternatively one can add '-path' flag as a parameter for configure:

```
./configure -path /usr/freeware/bin/
```

If you want to include more than one directory, then you simply separate them with a colon. This can for example be useful, if you want to use a special compiler:

```
./configure -path /opt/gcc/bin:/usr/freeware/bin
```

If a user defined path is specified, then configure will search this path for programs before it tries to locate programs with the default path.

Note, that if you rerunning configure script without parameters, all parameters which you have specified in a previous run of configure will be used.

## 3.2 Generation of Makefiles

The next step during the building of MOLCAS is generation of the Makefiles, which also can be accomplished with:

```
./configure -makefiles
```

These are generated from templates with a name something like *Makefile.in*. There are a program (and one for utilities as well) default template located in *src/Driver/*, but it is also possible to put one in the program (or utility) directory. If configure finds a *Makefile.in* in the program directory, this will be used. There are a couple of reasons to use a separate template for a directory:

- The code in this directory needs to be compiled with special compiler flags.
- The program should be linked in some special way, e.g. if it needs to pick up subroutines from another program.

But in the majority of cases (and the preferred one), it is not necessary to have a *Makefile.in* for a program. In this case configure will use either *progdefault\_Makefile.in* or *utildefault\_Makefile.in* - both located in *src/Driver/*. If you take a look inside one of these, you'll see several constructions like

```
@MOLCAS@
```

which is a reference to the corresponding variable in the *Symbols* file. Using the default templates, all information regarding the source code is assumed to be located in *00sources*

(in the directory containing the program or the utility). The same when it comes to dependencies (*00dependencies*). If *00sources* and/or *00dependencies* do not exist, then the program MkDep will be called, and this will generate them.

All information on suffixes (for more on suffixes and Makefiles see Appendix A) is located in *src/Driver/stdsuffix*.

There is also an alternative way to modify 'standard' *Makefile* for a code. Instead of *Makefile.in* file, a directory may contain a file, called *Makefile.add* with a redefinition of variables in a makefile.

```
# $Revision: 7.5 Patch(7.5): 012_add $
# Makefile.add for loprop
PLIB   = $LIBDIR/lib@module@.a -L$LIBDIR -lmolcas -lscf -l Rasscf -l caspt2
EXTRA_DEPS = $MOLCASROOT/src/scf/.stamp $MOLCASROOT/src/rasscf/.stamp $MOLCASROOT/src/caspt2/.stamp
```

*Makefile* in the MOLCAS root directory is generated from *Makefile.in*, which also is located in the MOLCAS root directory.

### 3.3 What happens, when you type *make*

1. First the special directories (with a name starting with upper case letter) are build.
2. Next Global Arrays (*g/*) is build.
3. All utilities (in alphabetical order) are build.
4. All programs are build.
5. Documentation is build.
6. The script *firstrun* is executed. This includes the first verification test and generating of database files (which goes to the *data* directory).

If you want the compilation to be faster, then you can type 'make' in the directory that you want to be compiled, but be aware that if you make changes to a utility directory, then you must both compile the utility and the program (in order to generate binary containing the new utility) again.

If you want to be absolutely sure that everything are build, then type 'make' in the MOLCAS root directory.

There are two main version of make: standard Unix make, and GNU make. Our Makefiles uses some non standard features, like conditional include statement, so GNU make should be installed in your system to build MOLCAS.

On some platforms GNU make has a special name, e.g. *gmake* or *gnumake*, but it can also just be named *make*. *configure* should find the real GNU make location, so if you type *make* in the MOLCAS root directory, then there shouldn't be any problems (Makefile in this directory could be interpreted by both versions of make!), but be careful if you type 'make' in a source directory.

## 3.4 Preprocessing the code

The content of the directories with source code can be of different degrees of complexity. In the simplest case, you just have FORTRAN files and nothing else. An example of this is *src/alaska/*. In more complicated cases the directory will also contain include files, e.g. *src/grid\_it/*. Include files can both be located in the local source directory and in *src/Include/*, where the global include files are stored. In a few directories, you can have a mixture of FORTRAN and C code together with include files, e.g. *src/util/*.

The include files can have two types of extension:

**.inc** is normal include files for FORTRAN.

**.fh** is include files to be used together with the preprocessor. These files contain some preprocessor commands, and the reason they have their own extension is that else the preprocessor doesn't know that it is necessarily to preprocess it.

In MOLCAS only quite simple preprocessor code is used, because the default preprocessing is done through the FORTRAN compiler. The typical constructions are similar to:

```
#ifdef _I8_
...
#else
...
#endif
```

The '#' character must be placed in the first column of the line. In the case above, you test whether the variable '\_I8\_' is defined (if it is, then this computer uses 8 byte integers). If it is, then you add some information to your code; if it is not, then you add some other information.

All the definitions of the preprocessor variable are done in the configuration file, e.g.

```
PPFLAGS = -D_IRIX64 -D_I8_
```

'PPFLAGS' is transferred to be a part of 'F77FLAGS', 'CPPFLAGS', and 'CFLAGS'. In this case '\_IRIX64\_' and '\_I8\_' are defined.

You can also include preprocessor instructions in the source code itself, e.g. *src/util/fndlnk.c*, but this is mostly used in the utility source code.

There are three different ways to do the preprocessing in MOLCAS. Which way it is done is defined through the 'FPREPROG' variable in the configuration file. This variable can take three values:

**f** : All preprocessing is done by the FORTRAN compiler (modern FORTRAN compilers are able to handle this).

**F** : The FORTRAN source files (\*.f) is copied to \*.F, and then the FORTRAN compiler is able to do the job.

**cpp** : The C-preprocessor does things directly (for 'stupid' compilers).

### 3.5 The structure of prgm-files

In MOLCAS we have a translation table between FORTRAN file names and UNIX file names.

These files are located in *data/* and are named as the program name followed by '.prgm' extension, e.g. the translation table for the program *grid\_it* is located in the file *data/grid\_it.prgm*.

The *.molcas\_info* file is used in verification, and the returncode file is used in the MOLCAS shell scripts.

Below is listed the prgm file for *grid\_it*:

```
# $Revision: 7.5 Patch(7.5): $
(prgm) "$MOLCAS"/bin/grid_it.exe      executable
(file) M2MSI "$WorkDir/$Project."M2Msi  rwm
(file) RUNFILE "$WorkDir/$Project."RunFile  rw
(file) ONEINT "$WorkDir/$Project."OneInt  rw*
(file) VBWFN "$WorkDir/$Project."VbWfn    ro
```

During building of MOLCAS some extra file definitions are added. These are common for all modules.

The first line is described in chapter Patch system.

All the remaining lines set up the correspondence between the logical and the physical names of files. The first parameter is either '(prgm)' or '(file)'.

If the first parameter is '(prgm)', then the second one defines the location of the executable. The last parameter is then either 'executable' or 'script'.

If the first parameter is '(file)', then the second one is the FORTRAN file name, the third one is the UNIX file name (variables are allowed in order to describe the location).

Last is a parameter which defines the file's attributes.

- ro - file is accessed for reading
- rw - file is accessed for reading and writing
- s - file will be saved (copied) from the scratch area
- m - file will be saved (moved) from the scratch area
- g - file can be visualized by program **GV**
- t - file is an ASCII text
- \* - name of the file is constructed by a filename mask.

Historical attributes (not used for the moment) are 'rw' for read and write, 'ro' for read only. Attribute 's' or 'm' means that after an execution of a module, the file should be copied or moved from WorkDir. A new location of the file can be set by *MOLCAS\_OUTPUT* environment variable. If it is not set - files will be copied to current directory, if it is set to 'WorkDir' - files will remain in WorkDir. If a file has a descriptor 's' or 'm' it also has to get an attribute of the file type: 'g' stands for an input for **GV** (grid file, molden file), or/and

't' - if it is an ASCII formatted text file. Attribute \* stands for multifile declaration. E.g. name *ONEINT77* will be translated to *\$WorkDir/\$Project.OneInt77*.

Sometime, a code generates a custom file, which is not listed in prgm. In order to apply actions, like saving or removing, the code can create a file called *extra.prgm* with the same structure as a normal prgm file. The file will be processed immediately after run, and deleted.

## 3.6 Packages

Each executable program in MOLCAS could be considered as a separate package or plug-in. There is a simple mechanism which allows to cut off a module from the MOLCAS source code, so this package could be maintained and distributed independently on MOLCAS itself. It means that the source code could be updated in any way (without the patch system), and the code could be distributed separately from the rest of the MOLCAS code.

There are two sets of commands to operate with packages: *install/uninstall* and *installpkg/uninstallpkg*. In both cases the result is the same - a package becomes visible or hided from molcas. However, *uninstall* simply mark the corresponding directory as a package (by creating a file with name *PACKAGE*. This directory will be ignored during the installation, however, all files still can be modified by the patch system. *uninstallpkg*, in contrary creates a tar file in *package* directory, so this package can be removed from Molcas, and e.g. distributed separately. But the patch system can't be used to make modifications in these files.

A command

```
molcas install
```

is used to create a new module. This script runs interactively and guide the programmer to create new modules including some template files, required for a new module, such as: *main.f*, *module.tex* and *module.prgm*.

The separation process is quite simple. First of all the source code (as well as tests, documentation, prgm file) must be placed into a MOLCAS tree. To extract a program from molcas one should issue the command

```
molcas uninstall program_name
```

This command will create a tar file (*packages/program\_name.tar.gz*) which contains the source code, tests related to this module, documentation from users guide, and *prgm* file. To put the package back, you should execute:

```
molcas install program_name
```

After both installation or uninstallation procedure the *configure* script should be executed to rebuild the list of programs.

By default during uninstallation of a module several questions will be asked - in particular, there is a possibility either to rename tests or keep these names as the global ones. To avoid these questions (and to create more complicated packages, for example, containing several modules) one should create some package configuration files *src/module/module.def*, *src/module/module\_install.def*, *src/module/module\_uninstall.def*. These files contain a set of UNIX commands (usually it is a set of environment variables which could change the default settings for a package). File *src/module/module.def* is executed in both

the installation and uninstallation process. And if `src/module/module_install.def` exists, it will be executed during installation (as well as `src/module/module_uninstall.def` for uninstallation).

A package definition file can contain a line

```
# Guess=YES
or
# Guess=NO
```

to specify a suggestion for the user - is it a required package or not. A line, started with

```
#
```

could be used to specify a very short description of the package.

If a package should contain several directories, one of them is called the parent (or main), and the others are child directories. In this case `src/module/module_uninstall.def` files should exist in all directories.

The set of environment variables one can use in these scripts is:

---



---

<i>File</i>	<i>Contents</i>
<code>MOLCAS_PACKAGE_TEST</code>	One gives a value 'keep' to keep the absolute names for tests, say test001.input against test\$module001.input
<code>MOLCAS_PACKAGE_TEX</code>	to change the name of the L <sup>A</sup> T <sub>E</sub> X file in the users guide (or set it as blank, if there is no documentation for the content of this directory).
<code>MOLCAS_PACKAGE_NAME</code>	set up the name of package (the name of parent directory in the package)
<code>MOLCAS_PACKAGE_CHAIN</code>	space separated list of child directories.

---

Most complicated (and not recommended) way of making packages includes using `module_install.def`, `module_install.def` files. If you had to create a package from an utility directory, you must create a dummy wrapper for these functions, and activate it (for example, by renaming a file dummy.wrapper to dummy.f) during removing package from MOLCAS

### 3.7 Examples of package definition files

An example of simple package definition file which allows to keep test numeration:

```
# src/casvb/casvb.def $Revision: 7.5 Patch(7.5): $
# Molcas Package version 1.1
# Guess=NO
#MOLCAS_PACKAGE_TEST=keep
export MOLCAS_PACKAGE_TEST
```

An example of simple package definition file for utility directory:

```
# src/blas_util/blas_util.def $Revision: 7.5 Patch(7.5): $
```

```
# Molcas Package version 1.1
# Guess=YES
#echo '***** Blas utilities      *****'
MOLCAS_PACKAGE_TEX=""
export MOLCAS_PACKAGE_TEX
```

An example of a package containing several directories.

File *src/casvb/casvb.uninstall.def* (parent directory):

```
# $Revision: 7.5 Patch(7.5): $
# set the name of .tex file
MOLCAS_PACKAGE_TEX=casvb
export MOLCAS_PACKAGE_TEX
# set the name of child directories (space separated)
MOLCAS_PACKAGE_CHAIN="casvb_util"
export MOLCAS_PACKAGE_CHAIN
# keep test names
MOLCAS_PACKAGE_TEST="keep"
export MOLCAS_PACKAGE_TEST
```

File *src/casvb\_util/casvb\_util.def* (child directory):

```
# $Revision: 7.5 Patch(7.5): $
# Molcas Package version 1.1
echo '*** This directory should be uninstalled as package casvb ***'
exit 1
```

The *MOLCAS\_PACKAGE\_CHAIN* variable defines the packages which are connected to the *ccsd* directory, so that when this one will be uninstalled, all the others in the list will be uninstalled too. The packages in the list, instead, cannot be uninstalled by themselves, because they are part of a more general package.

Note, that during automatic verification of MOLCAS all packages could be installed and tested, if you run *configure* as *configure -packages*

Finally, the most complicated case if you want to create a package from an utility directory. In this case we have an obvious problem - if you remove a source code for some 'utility' calls - you will not be able to build some of executables. So, you have to substitute calls to these routines by a dummy calls.

The complete example you can find out in *src/aces2\_util* directory. All routines with dummy calls can be placed into *src/clones\_util*, so they can be swapped during installation and uninstillation.

File *src/aces2\_util/install.def* contains lines:

```
rm -f $MOLCAS/src/clones_util/bwread.f
touch $MOLCAS/src/aces2_util/bwread.f
```

File *src/aces2\_util/uninstall.def* contains lines:

```
cp $MOLCAS/src/clones_util/bwread.wrapper $MOLCAS/src/clones_util/bwread.f
touch $MOLCAS/src/clones_util/bwread.f
```

File *src/clones\_util/bwread.f* contains a dummy routine:

```
*fordeck ACES2 $Revision: 7.5 $
*
* collection of wrappers for aces2
* SUBROUTINE bwread(invcc,iuhf)
```

```

logical invcc
Call SysAbendMsg('aces2_util','ACES2 package is not installed',
& 'Happy landing')
return
end

```

### 3.8 Rebuilding Molcas

Sometimes you need to rebuild the code, e.g. if you need to change the flags used during compilation. To be able to do this, you first have to clean the installation by using either `make veryclean` or `make distclean` in the top directory.

If you just need to rebuild one module, you can go to the source directory and execute `make clean` there.

---



---

<i>File</i>	<i>Contents</i>
<code>make clean</code>	The module library will be deleted as well as the executable, temporary files used during compilation, the stamp file, and files used for solving dependencies. This procedure is necessary, if you add or delete a file to the directory or remove a subroutine from a file.
<code>make veryclean</code>	Deletes everything created during the installation except for the files used for solving dependencies, and <i>Symbols</i> file. The script used for solving dependencies is quite slow, so this is the reason, you are given the possibility to leave these files.
<code>make extraclean</code>	In addition to <code>veryclean</code> option, delete <i>Symbols</i> and <i>molcas.rte</i> files.
<code>make distclean</code>	The same as <code>make veryclean</code> except that the files for solving dependencies are also deleted. This command actually takes the MOLCAS distribution back to the same state, as it was just after unpacking it.

---

If you run `make veryclean` or `make distclean` you have to specify proper flags for configuration.

## 3.9 Installation of GUI

There are several possibilities to visualize coordinates used in Molcas calculation, and the results: molecular orbitals and densities, geometry optimization, frequencies.

In order to install graphical user interface (GUI) codes for MOLCAS you have to install some extra libraries, which are different for different operating systems.

The complete set of GUI codes in MOLCAS includes:

- gv - geometry and grid viewer and editor.
- ming - molcas input generator
- molgui - molecular builder and frontend for ming.

You can use either a combination of **GV** and **MING**, or **MOLGUI** to create input, and visualize output files calculated by molcas.

### 3.9.1 gv: Molcas Grid Viewer

Grid Viewer (**GV**) is an OpenGL based code, running under Linux, Windows and MacOS.

First of all, you must check that your Linux installation contains all packages which are needed for Mesa development. If configure can't locate include file `glut.h`, ask your system administrator to install required packages. The recommended source for Glut libraries is: <http://www.mesa3d.org/>

Under MS Windows you have to install Cygwin framework, and check that you have installed opengl packages for development.

If during the linking of **GV** you got messages about unresolved `_glut` names, it means that you have a conflict between glut header files, and libraries. In this case, locate all `glut.h` files, which are installed in your Cygwin environment. All these files are located in **GL** subdirectories. You have to delete (or rename) all **GL** subdirectories, except of `w32api/GL`.

### 3.9.2 ming: Molcas input generator

**MING** uses several graphical libraries, including wxWidgets. If the standard installation doesn't work (e.g. due to incompatibility with version of wxWidgets) you have to install these libraries manually.

#### installation for Linux

- Download wxGTK 2.8.4 <http://www.wxwidgets.org/>, unpack it, and cd into unpacked directory
- mkdir buildwx
- cd buildwx

- select an installation directory, here we assume it is `$WX_INSTALL_DIR`, e.g. `/opt`, please replace this variable with a local path in your computer
- `./configure --prefix=$WX_INSTALL_DIR/wx/2.8 --with-gtk=2 --with-gnomeprint --with-opengl --disable-debug -enable-geometry --enable-graphics_ctx -enable-sound --with-sdl --enable-mediactrl --enable-display --enable-unicode --enable-debugreport`

Note: If you try to link wxWidgets libraries statically, please add `-disable-shared`

The default wxWidgets requires `gtk+-2.0` or above. If you are not sure, please execute `'gtk-config -version'` to see the output. If it is 1.2, please use `-with-gtk=1` instead. But we suggest you install wxWidgets with `gtk+-2.0`

- `make`
- `make install`

Note: If the installation directory `$WX_INSTALL_DIR` needs root privilege, switch to root before run this command

- modify environment setting

```
WX_HOME=$WX_INSTALL_DIR/wx/2.8
LD_LIBRARY_PATH=$WX_HOME/lib
PATH=$PATH:$WX_HOME/bin
export PATH LD_LIBRARY_PATH
```

Expat library is included by most linux distributions. If not, please install it from <http://expat.sourceforge.net/>. The libcurl can be downloaded from <http://curl.haxx.se/>. Out tested version is 7.18.2.

To compile MING you can execute `make` command in `ming` directory, or configure molcas with flag `-ming`.

### installation for Windows

First, you have to install wxDev-C++ 6.10.2. Download `wxdevcpp-6.10.2_setup.exe` from <http://wxdsgn.sourceforge.net/>, and install it.

Installation of expat:

- Download `expat-win32bin-2.0.1.exe` from <http://expat.sourceforge.net/>, and install it. For example, we install it at `C:\Program Files\Expat 2.0.1`
- copy all files in `C:\Program Files\Expat 2.0.1\Source\lib` to `ming\include\expat`
- copy all files in `C:\Program Files\Expat 2.0.1\Bin` to `ming\lib`

Now, we can compile MING:

open `ming\ide\wxdev\ming.dev` with wxDev-C++, click menu `Execute/Compile`. After successful complication, an executable file `ming\ming.exe` will be generated.

**installation for Mac**

- Download wxMAC 2.8.4 <http://www.wxwidgets.org/>, unpack it, and cd into unpacked directory
- mkdir buildwx
- cd buildwx
- select an installation directory, here we assume it is `$WX_INSTALL_DIR`, e.g. `/opt`, please replace this variable with a local path in your computer
- `../configure --prefix=$WX_INSTALL_DIR/wx/2.8 --with-mac --with-gnomeprint --with-opengl --disable-debug -enable-geometry --enable-graphics_ctx -enable-sound --with-sdl --enable-mediactrl --enable-display --enable-unicode --enable-debugreport`

Note: If you try to link wxWidgets libraries statically, please add `--disable-shared`

- make
- make install

Note: If the installation directory `$WX_INSTALL_DIR` needs root privilege, switch to root before run this command

- modify environment setting

```
WX_HOME=$WX_INSTALL_DIR/wx/2.8
LD_LIBRARY_PATH=$WX_HOME/lib
PATH=$PATH:$WX_HOME/bin
export PATH LD_LIBRARY_PATH
```



## Section 4

# The Patch System

### 4.1 Why use a patch system?

MOLCAS provides its own system to deal with patches. There are two reasons for using a patch system together with MOLCAS:

- to have the possibility to undo changes to the code (i.e. have a backup),
- to be “safe” when two or more developers work on the same file at the same time.

In MOLCAS anyone is allowed to make changes to all the files included in the program, even if it is a file that this person does not own. So we need to have a system that gives us the possibility to check whether we can include both changes (in case of two working on the same file). That is, it should not be possible to overwrite changes made by some other developer without knowing it.

This is called a patch system, and it automatically takes care of the backup part. The most used system is called CVS, which only includes the differences between different versions of the files, so it keeps the patches very compact.

We use our own patch system for various reasons:

- Historical reasons.
- Because this patch system has been used for quite a while, it is well known by the developers (at least those that have been for some time).
- Several MOLCAS specific things are included in the system, e.g. if you change a file, the patch system will take care of *.stamp*, *00dependencies*, and *00sources*, if it is necessary (e.g. deleting them).
- Our patch system knows the directory structure of MOLCAS.

## 4.2 How the patch system works

If you need to make a change to a file, you pick it up from the master version, and you make the changes. Maybe in the meantime, someone else applied changes to the same file. In order to make sure you'll be notified about this, all files contain a comment line of the form:

```
# $Revision: 7.5 ... $
```

The start of the line must be a `#` for shell scripts (including Perl), `C` or `*` for FORTRAN code, and `/*` for C code. This line must be the first or the second in the file (first line source code and second line for shell scripts). Also note that when one adds the first part of the line by hand, then the final `'$'` should be added as well. The line consist of three parts:

**# \$Revision: 7.6:** This part must be added manually for new files.

**...:** Will be generated by the patch system and will have the form: “patch(7.6): 044\_cygwin 045\_structure”, where '044' is a patch number, and 'cygwin' is the label of patch number 044. This information is also used as part of the backup system.

**\$:** Should also be added manually, if the first part is.

The patch system only considers the part between the two `$s`. So when you try to apply your change, there'll be a conflict, so you'll need to pick up the new file and try again.

When the patch is applied, it is located in the `patch/` directory. This directory contains two different types of subdirectories:

- Those which name starts with three digits followed by an underscore and a label, e.g. '044\_cygwin'. These subdirectories contain the real patches.
- Those which name is just the label. These are caused by a mistake by a developer.

The subdirectories contain 6 files:

---



---

<i>File</i>	<i>Contents</i>
<i>The patch definition file</i>	This file has the same name as the directory followed by '.def'. In other words, the definition file in the directory '044_cygwin' will have the name '044_cygwin.def'. We will return to the contents of this file in section 4.3, where we address the problem of generating a patch.
<i>old.tar.gz</i>	Contains the backup of the files replaced by this patch.
<i>dependents</i>	Contains information on which patches depend on this patch. Due to the way how the patch system works, all these patches must be younger than '044_cygwin', and these are called the children of this patch. The common thing of the children is that they all uses files which where changed by this patch. If there are any children, we can't revert this patch until the children have been reverted as well. See the end of this section for more details.

<i>revert</i>	A script which can be executed in order to restore the code as it were, before this patch was applied. The script should be executed for the MOLCAS root directory as “patch/044_cygwin/revert” (or with whatever patch name in question).  Why do we want to be able to revert a patch? Maybe testing of the patch, before it is applied to the master version, shows errors in the patch. Then it is necessarily to revert the patch, before it can be applied again.
<i>files</i>	A list of all the files that have been changed.
<i>README</i>	This file is generated from the definition file. This file is not generated for ”fake” patches (”fake” means that this patch came from a file with a broken header). The existence of this file in the corresponding patch directory is used by some scripts, e.g. <code>getpatch</code> .

---

The problem with our patch system is that the `old.tar.gz` files become huge. In order to solve this problem there is a “light” version of MOLCAS, where all the `old.tar.gz` files have been excluded. The light version can not be reverted, but it is in all other ways fully operational. The gain is that the size of the light version download file (`molcas.tgz`) can be down to half the size of the full version.

There is no build in requirements for names of the patches, because the patch system knows everything about children and parents. This, of course, makes it hard for the users to see the relationship, but the numbering of the patches provides a guide: children can *never* have a patch number lower than it’s parents. This also results in that patches always are applied in number order.

Patches in MOLCAS are not cumulative. Separate patches could be reverted, if they have any children, all tree of dependent patches will be reverted by a command `commandmolcas revert patch PATCH_NUMBER`

Unless of very special occasion, developers should only create patches that are revertible!

### 4.3 Make reservation for a patch

Because patches should have a sequential and unique number, there is the following procedure for reserving patches:

(A note for external developers who do not have access to the developers homepage: they can use the patch system, but they could choose an arbitrary name of the patch, if they are sure that the sequence of applying the patches does not matter).

- Go to <http://www.teokem.lu.se/molcas/dev>. The details of the MOLCAS developers homepage can be found by using the help links next to the items on the page. Accessing this page require that you have a user name and password.
- Go back to the MOLCAS developer page and follow the link ’Reserve/apply patch’. This will take you to the reservation page, where you’ll be given three different options for reserving patches:

**Only reservation** Do everything manually. You should create the patch definition file, build patch, and upload it using the respective link. This is usually not recommended.

**Prepare template** This is the default way. A script will prepare a template for the patch definition file, and you'll just need to add some information about the files you want to patch. You will need to build the patch and upload it using the web. This method will be discussed in detail a few lines below.

**Auto** Fully automatic. This method should only be used for very simple patches, such as change of a print statement. Using this option, you have no possibility to check your patch, and the patch system will send the patch and notify the patch master automatically.

### 4.3.1 Prepare template

- The next screen is where the template is generated. The screen contains five fields, which are:

**Patch name:** The patch label (the unique number and the underscore will be added automatically), e.g. 'cygwin'. Don't use any special characters in the label except for letters, numbers, and underscore.

**Patcher:** Find your name on the list. If you are not on the list, please contact the MOLCAS webmaster.

**Version:** Which version do you want to apply your patch to. Patches only work for one version, so it is important to choose the correct version.

**Keywords:** Some useful (please) keywords description your patch, e.g. 'RASSCF' if you change something in the RASSCF module. The keywords are searchable. This field can be edited later on.

**README:** Again, please write something useful, describing what this patch does. The README field is also searchable. This field can be edited later on.

- Press 'Go!'.
- The script now sets up a unique patch number and displays the template.
- Save the template ('File' -> 'Save Page As...' in Netscape). It is recommended to give a name like *patch.def*, which should result as the default name on most of the browsers.

Now you are ready to do the actual work of generating the patch:

- Pick up the file (or files) that you want to change from the master version, if you have access to it. If you don't, download the newest version from the MOLCAS developer page (or run `molcas getpatch` - see section 4.5).
- Make the changes in the file(s).

- Append the file name(s) under the field 'files' in the definition file. If you have made changes in *seward.f* from the *src/seward/* directory, the line should read:

```
files
seward.f      src/seward/
main.f        src/seward/
```

The last slash is important, because if it is not there, the program that builds the patch will believe you are trying to rename the file. If you actually do want to rename the file, you should write:

```
seward.f      src/seward/new_file_name.f
```

instead.

All the files, you have changed, must be located in the same directory as the definition file, which in turn must be the current directory.

Note, that in order to avoid some typos, the patch system refuses to create a new directory. If you would like to create a new directory you have to create this directory in a separate patch.

- Execute 'buildpatch'. This script will also make some preliminary testing of the validity of the patch:
  - If the definition file is named *patch.def* (default) and MOLCAS is located in */temp1/7.6.dev/* type:
 

```
export MOLCAS=/temp1/7.6.dev
molcas buildpatch
```
  - If you have chosen another name for the definition file, then you'll need to give this name as an argument to buildpatch.
  - During build, buildpatch will notify you, if there is any conflicts between your file and the patches already in MOLCAS. The error message will contain both the name of your file and the number and name of the patch. This will cause buildpatch to exit.
  - You will also be warned, if you try to create a new file. buildpatch will continue, but you should check whether you really want to create this file, or the message is caused by an error, e.g. misspelling of the file name.
  - If the patch is already installed (e.g. from a previous validating attempt), you'll get the error:
 

```
"7.6.044_cygwin.sh.gz has already be installed."
```

 So you'll need to revert the old patch, before you can rebuild.
- If everything went well, a file named *7.6.044\_cygwin.sh.gz* has been created (of course with the proper substitutions for version, patch number, and patch name).
- If you check the file you changed, you can see that your patch number and name have been added to the header.
- If you rerun buildpatch, the header will not be altered any further.
- Now apply the patch to a local version:
  - Copy the patch file (e.g. *7.6.044\_cygwin.sh.gz*) to the MOLCAS root directory.

- Decompress the patch file, e.g.
 

```
gzip -d 7.6.044_cygwin.sh.gz
```

 (or use gunzip or whatever you prefer).
  - Install the patch with:
 

```
./7.6.044_cygwin.sh
```

 (on a few systems, e.g. cygwin you'll need to use the command `sh ./7.6.044_cygwin.sh`).
  - The patch is now applied, and you can see that there a new directory in `patch/` with your patch.
  - Type 'make' (in the MOLCAS root directory) to build the new version.
- Verify that the test suite works (see section 5).
  - When everything is OK, go to the homepage and upload the patch.
  - Patches are usually applied to the master version once a day (around 4 pm.).
  - If you during the day find out that there is an error, you can resend the patch, but *only* until the patch has been applied. After that you'll need to send a new patch!

### 4.3.2 Creation of more complicated patches

You can also do some more advanced manipulations like deleting files. The possibilities are:

**delete** This subsection in the patch definition file defines which files to delete.

```
delete
src/seward/seward.f
src/seward/main.f
```

This deleting is revertible (i.e. it is a safe operation). If you want to move a file from one directory to another, the proper way to do it is to delete it from the original directory and create it (in the files subsection) in the new directory.

**commands** In this subsection you can put any UNIX command. These commands are in general NOT revertible! One exception are creation of new directories. It can also be necessary to use this subsection to replace binary files, because the header is not available and for this reason the patch system will not allow you to replace the file in other ways.

Usually patches do not require the MOLCAS installation to be reconfigured. But if you create or delete directories, change configuration files (e.g. change compiler flags), or the like, you have to force the user to run `configure` manually by including a *Makefile* (to be located in the MOLCAS root directory), which tell the user to reconfigure. The Makefile should look something like:

```
# $Revision: 7.5 Patch(7.5): 030_cut_off5 145_molcasdisk $
default:
    @echo "Please run configure and make afterward!!!"
```

You must find out on your own which patch labels to include in the header (or you can just delete the *Makefile* in the root directory).

If your are really sure that your file is correct, you can manually edit the header of your file and add labels of parent patches to cheat patch system checks.

### 4.3.3 Auto

If you only need to patch a single file, or all files are located in the same directory, then it is possible (but still not recommended!) to use the automatic reservation of patches. The first part of the submission page is the same as for the template. The rest of the fields have the following meanings:

**Select input tar** Put the name of the file to patch (or in case of several files, put the name of a tar-file containing all the files).

**Single file** Select this check box, if - and only if - it is a single file (opposite of a tar-file).

**Executable** Select this check box, if the file is an executable, e.g. a shell script.

**Directory** All files must belong to the same directory. Select this from the drop down list, or type it in the field to the right.

**Extra commands** Type a 'delete' and/or 'command' block manual here, if these are needed.

### 4.3.4 Only reservation

If you choose to do everything manually, then you just type the name of the patch, choose who you are, and choose which version the patch applies for. Then the script will return the number of the patch, and after this you are on your own.

### 4.3.5 Most common problems during building of patch

Sooner or later, you will experience error messages or warnings as a response to your attempt to build a patch. Here follows a list of the most common problems and an explanation of what causes them.

**file.f (src/program) cannot be read** You have added an entry in *patch.def*, but the file specified does not exist in the directory. All the files to be applied must be located in the same directory as *patch.def*.

— **Warning : new file src/program** You probably wanted to place your file in the directory *src/program/*, but if you omit the slash at the end or misspell the name, *buildpatch* will think you want to add a new file (which of course could be what you want).

**there appears to be a problem running shar .. cannot continue** This error message is for example caused by a slash in front of the directory name, e.g.

```
file.f  /src/program/
```

**\*\*\* Conflict:** src/Driver/molcas.src already exists in patch 297\_submit\_pwd  
 \*\*\*\* patch CANNOT be applied \*\*\*\*\*

This means that you probably try to apply a patch containing a version of a file older than the one currently available in the master version. You should retrieve the latest version of the file and incorporate your changes to this file instead.

**You should apply this patch to version 6.x only, not to 6.y** You have tried to apply a patch belonging to version 6.y to MOLCAS 6.x. Check that the environmental variable MOLCAS point to the same version as your patch.

**\*\*\* Filelist is empty?** It is a warning that either means that you have forgot to add which files should be included in the patch, or you have a blank line after the 'files' statement in *patch.def*.

**Dummy dependencies** Sometime, especially if you making an update for a file, coming from different version of molcas - the header of the file contains dummy dependencies to patches, which are 'missing' in a current version. This is not a critical problem for the functionality of the patch system, however, this problem is detected by the system.

**Police** You get this message if your patch contains the code, which violating some rules for FORTRAN programming in MOLCAS . If you (for some reason) can't fix the problem with this checks (since *police* script makes only recommendation for the clean code - you may create a file *nopolice*, in order to change this error into a warning.

You should also be aware that you can have errors in your *patch.def* without getting errors. One of the most common mistakes is to have an invalid patch name (after the 'name' statement). The format have to be:

```
123_name
```

where the number consist of three (and always three) digits followed an underscore and the name of the patch. Without the number and the underscore, the patch system will not know in which order to apply the patch, and it will be considered an invalid patch.

### 4.3.6 Reverting patches

To revert a single patch, you should execute the command

```
molcas revert patch PATCH_NUMBER
```

This script will uninstall a patch, and all it's children.

An alternative way of reverting a single patch:

```
./patch/patchname/revert
```

where 'patchname' should be replaced with the complete name of the patch (including the number and underscore, e.g. 044\_cygwin). During reverting the script will check whether there is any children of patch; if this is the case, revert will refuse to revert the patch, and it will warn you about the conflicts instead.

Furthermore MOLCAS has a tool to revert several patches:

```
molcas revert 'test number'
```

where 'test number' is a number of a test in the verification suite (see Chapter 5). This command will keep reverting patches in reverse order until the test passes.

Note, that reverting patches has sense only for a complete version of molcas (not 'light'), which contains the complete information about previously installed patches.

## 4.4 Making the patch available to all

After you have uploaded the patch, it will go to the repository, and it will be available for download immediately. Later it will be applied to the master version. This is usually done once a day (in the late afternoon), and then the patch is available for downloading through the `getpatch` command (see next section).

If you realize that there is an error or conflict in your patch, you can resubmit it until it has been applied to the master version.

It could still be that there are conflicts which wasn't discovered during the building. If this is the case, you will be notified by mail.

You can check the status of your patch by choosing the link 'Check reservation' from the MOLCAS developer homepage. This will generate a list of the patches for the chosen version. The list can be generated with two formats, either as a list of the patches with the status of it, or as a description containing the keywords and the readme sections from the definition file. This is also the place to go in order to check whether your patch has been applied to the master version. (Be aware that if you change the label of the patch between reservation and application, then the script making this page will not think the patch has been applied). If your patch isn't applied, you should contact Mr. Patch.

## 4.5 Retrieving of patches

You can ensure that your version of MOLCAS is fully updated in three ways:

- You can download the newest version and install it.
- You can download the patches you need from the developers homepage via the link 'Download patches', but this is not a safe way, because patches could be resubmitted. However it is the way to retrieve patches not yet in the master version.
- Use the `molcas getpatch` utility. This is the recommended way, and it will be described in further detail below.

### 4.5.1 `getpatch`

The `getpatch` script will download new patches, so you don't need to download a complete new version. The syntax is:

```
molcas getpatch
```

or

```
molcas getpatch -unsafe
```

The first call will only download patches, which are considered safe, whereas the latter will download all patches applied to the master version. The definition of a "safe" patch is that it has been tested for some time. It is of course recommended to use a version also containing the "unsafe" patches, when you do development.

Please notice that it can take a long while before a patch is moved from unsafe status to safe status, so if you need a fully updated version, you really need to install the unsafe patches as well as the safe ones.

Not recommended, but sometime useful flag for `getpatch` is `'-crazy'`. In this case `getpatch` also retrieve all patches from patch repository, so you have a possibility to update your version with all submitted patches. A potential risk with using this option is obvious, if a patch will be resubmitted during a day (before an update of official version), `getpatch` will NOT retrieve a second (third) edition of the patch. Use this option only in case you really know what you are doing!

`getpatch` could require authorization. This will then be the same username and password as for MOLCAS developers homepage.

If you need any help with `getpatch`, then `molcas getpatch -help` gives you some help on the usage of `getpatch`.

It is recommended for developers to update their version at least every other day.

You can modify the default behavior of `getpatch` in other ways as well by adding the relevant flags (the first latter will suffice):

---



---

<i>File</i>	<i>Contents</i>
<code>-query</code>	This will cause a dummy execution. <code>getpatch</code> will contact the server and check whether there is any updates available. If this is the case, it will print a list of the patches.
<code>-verbose</code>	Instead of automatically applying the patches, <code>getpatch</code> will ask before installing each patch.
<code>-netscape</code>	If you do not have Perl installed, you need to use another client for downloading the patches. Netscape is one of the possibilities.
<code>-lynx</code>	Use Lynx to download the patches instead of Perl.
<code>-file</code>	<code>getpatch</code> will create a file called <code>getpatch.html</code> which can be used to retrieve the patches in case you are sitting behind a firewall.

---

## Section 5

# Verification

MOLCAS comes with a verification script, which will run a set of calculations in order to test whether the installation contains bugs or not. To run the default set of verification tests just type:

```
molcas verify
```

but this will take quite a while, especially on computers that are not 100 % up to date. So it is possible just to run a subset of the tests. This can be done in several ways:

You can specify the test(s) you want to run by giving the test number as argument, e.g.:

```
molcas verify 001 102
```

You can also specify a name of a program. In this case all tests including this program will be run:

```
molcas verify caspt2
```

The last way, is by specifying a keyword from the input, e.g.

```
molcas verify direct
```

will run all tests with the keyword 'direct' in the input file.

You can also choose one of the special tasks: 'all' runs also time consuming tests except of performance tests, 'small' is default, and 'performance' runs a special test suite designed for measuring the performance. Keyword 'failed' runs verification for all failed tests from previous run of verification.

The verification script also includes some other options:

Flag	Example	Description
-l	molcas verify -l direct	Gives all the test with the keyword 'direct' as output.
-help	molcas verify -help	Gives a complete list of flags.
-k	molcas verify 001 -k	Execute test 001 and keep the files in <i>Test/tmp/test001/</i> .
-d	molcas verify 001 -d	Execute test 001 and flush all output to STDOUT.
-v	molcas verify failed -v	Verbose mode. Will immediately print whether a test passed or failed.
-i	molcas verify 001 -i	Ignore any errors in check values.
-q	molcas verify 001 -q	Quiet run. Returns the return code, which can be useful in scripts.
-generate	molcas verify 001 -generate	Generate new check values for test 001. See also Section 5.2.

## 5.1 How does it work?

If you take a look at the contents of the directory *Test/*, you'll find (some of the files/directories will not appear until after the verification script has been run at least once):

<i>File</i>	<i>Contents</i>
<i>Result</i>	This is a file that contains the summary on which tests passed and which failed.
<i>Failed Test</i>	This directory will contain log- and error files for the tests that have failed.
<i>input</i>	This directory contains the inputs (normal auto inputs) for all the tests.
<i>check</i>	<p>This directory contains one file for each test with informations of which values to check. These check files are auto generated as described below. The structure of these files are such that there is a "block" for each module executed in the given test. Each of these blocks contains lines with a label, a reference number, and a tolerance. The tolerance should be given as an integer: a positive tolerance (e.g. 5) means that is should be interpreted as 10 to the negative of this number (e.g. <math>10^{-5}</math>). A negative tolerance means that the tolerance should be taken as it is written. An example of the check of how many iterations done in a SCF calculation can be found in <i>Test/check/test001.check</i>:</p> <pre>#&gt; SCF_ITER="1.000000000000"/8</pre> <p>where the label is "SCF_ITER", the value is "1.000000000000", and the tolerance is "8" (meaning <math>10^{-8}</math>).</p>
<i>timing</i>	This directory contains timing information of the tests. These data are used for benchmarking (see 5.5).

<i>tmp</i>	This directory contains the files needed during execution. Each test will make a subdirectory with the name of the test, and this subdirectory will be deleted after the test unless the '-k' or '-d' flag has been specified. The location of this directory can be redefined, if \$ProjectDir is defined.
------------	---

---

The program check will be run between all the modules listed in the input. This program will compare the check data generated by the verification test, and the check data listed in the appropriate check file in *Test/check/*.

So why is it necessary to allow for a difference between the data in the check file and the data generated? Unfortunately one can not expect to get the exact same result on different machines and with different compilers. First of all, reals are not represented with the same number of figures on all computers, so there might be rounding errors arising from this. Second, the compilers reorder the code to make it more efficient. This reordering might make rounding errors as well (especially seen with very high optimization). Third, the differences can of course arise from changes in the source code.

The test suite is run regularly on some machines, and the results of these verifications can be found on the MOLCAS developers web page. Here it is also possible to get the output from the failed tests.

## 5.2 How to generate a test

To generate the check files, you simply execute the verification with the flag '-generate':

```
molcas verify 001 -generate
```

You should replace '001' with the number of the test, you want to generate new check files for.

If you want to create a new test, you simply copy your auto input file to the *Test/input/* directory giving it the next available number, and then run the command given above for this test.

One should be aware that when you create a new check file, then the header is empty, so if the previous check file had labels in the header, then these labels should be added by hand.

And be *very* careful, if you substitute a check file. A wrong check file can mislead people for a long time!

All new tests should be reasonable (we don't want to spend a long time on verification just for the sake of it). The tests should correspond to some real system. Do not use situations with numerical instability. Tests should test a particular keyword (or set of keywords). In order to prevent unnecessary numerical problems make sure the results are not "too small".

## 5.3 How to add new information for checking

This is very simple. In the source code for the program, you simply put a call to the subroutine called 'Add.Info':

```
Call Add_Info('label', value, size_of_array, tolerance)
```

The arguments have the following meaning:

**label** is a string containing the label you want displayed in the check file.

**value** is the value or values you want to check. In the latter case you simply specify an array. All values must be reals, so if you want to check an integer, e.g. the number of iterations, then you must convert this to a real.

**size\_of\_array** specifies how many values there are. If you just want to check a scalar, you simply put 1 here. If the 'value' argument contains an array, then put the size of the array.

**tolerance** is the measure for how much you allow the tests to deviate from the value in the check file. See section 5.1 to find information on the conventions for how to specify the tolerance.

It is quite safe to add extra information. If the verification includes more informations than there are in the check file, then this information will just be ignored.

Example:

```
Call Add_Info('PotNuc',PotNuc,1,6)
```

## 5.4 Conventions

The convention (at least for the time being) is that all check files must be generated on an IRIX machine or a Linux computer with GNU compiler version 3.0.4.

The convention on the test number is:

Test number	Description
001 - 899	These are the "normal" tests. The test number just increases with one each time a new test is added. Holes in the list ordering are caused by packages (see section 3.6 for more information).
900 - 999	Benchmark tests.

The test numbers for external packages could be either static in the same way as for normal tests, or they should contain the name of the package. See section 3.6 for details.

## 5.5 Performance tests

For performance tests you can have a look at the homepage (<http://www.teokem.lu.se/molcas/benchmark.html>).

Benchmark tests are based on (currently) 4 different calculations of real systems. There kind of results are analyzed:

- FPU benchmark
- I/O benchmark
- integrated test with heavy memory request.

Benchmark test presented in the Table 1 (on the web page) could be used for comparing of different computers from the point of view of FPU. These results reflect the time of the mckinley module (no I/O and low memory request).

Table 2 (on the web page) presents the results for the caspt2 module (with heavy unbuffered I/O). These results could be used to estimate the disk performance and role of RAM for disk caching.

The results in Table 3 (on the web page) show the benchmark in a real calculation with heavy memory request (test 902, calculation with different modules). These results also could be used to estimate the role of RAM

For testing purpose MOLCAS compiled with standard optimization level (-O2) has been used.

To perform a test on a new computer, one should issue the commands:

- molcas verify performance
- molcas timing (to generate the report in the *Test/timing/user.timing* file)

timing script also can be used to check the time of molcas tests with a reference timings.

A command `molcas timing -generate -file=/path/filename` creates a reference file with timings. Execution of command `molcas timing -file=/path/filename` will check actual timings against reference data. In order to force timing command to use normal (not performance) tests, you should add a flag `-all`.

## 5.6 Automatic verification

There is yet another way to verify the installation. There is a script (*checkinstall*) which is not part of the MOLCAS distribution, but which can be downloaded from the MOLCAS developers homepage.

The procedure for using the script follows here:

- Download the *checkinstall* script from the test page <http://www.teokem.lu.se/molcas/dev/test>.
- The script needs a little configuration. This should be self explain.

---



---

<i>File</i>	<i>Contents</i>
<i>CONTACT</i>	Information on who use the script including e-mail address.
<i>MOLCASHOME</i>	Where to install MOLCAS.
<i>PASSHOME</i>	Where the <i>.pass</i> file is located (see below).

<i>NAME</i>	A short description of the computer and/or compiler.
<i>RETRIEVER</i>	Can take of the two values: <code>wget</code> or <code>webget</code> . This defines how to retrieve MOLCAS. If you choose <code>'wget'</code> , the script will use the <code>wget</code> program, whereas <code>'webget'</code> will cause the build in Perl script to be used.
<i>MAIL</i>	Specify which mail program to use for mailing the result of the verification. You can choose between <code>'mail'</code> (the normal mail program) or <code>'perlmial'</code> (a build in Perl script). In the later case, you should also specify which server to use as SMTP-server; this is done with the <code>SERVER</code> variable.
<i>GUNZIP</i>	The command needed in order to <code>gunzip</code> .
<i>UNTAR</i>	The command to <code>untar</code> files.
<i>MAKE</i>	The location of <code>make</code> .
<i>HNAME</i>	The command to retrieve the hostname. Usually <code>hostname -s</code> or <code>hostname</code> .
<i>UNAME</i>	The command to get the name of the operating system in short form.
<i>LNAME</i>	The command to get the name of the operating system in long form.
<i>PATH</i>	Specify the <code>PATH</code> to use during the script. Notice that the <code>PATH</code> used via the cron very well can be different from your normal <code>PATH</code> .
<i>RUN</i>	Specify which version you want to test. You should write the version in a space separated list, e.g. <code>'6.0.dev 6.1.dev'</code> .
<i>TESTS</i>	Specify which tests you want to run. See the first part of this chapter for the various possibilities.
<i>WWW</i>	The location of MOLCAS. It shouldn't be necessary to change this.
<i>LANG</i>	Define the default language used in the <code>checkinstall</code> script. Currently the only accepted value is <code>'en'</code> .
<i>DoDownload</i>	Do you want to download MOLCAS? If you answer anything else than <code>'yes'</code> , downloading will be skipped.
<i>DoMake</i>	Answer <code>'yes'</code> if you want to build MOLCAS.
<i>DoTest</i>	Answer <code>'yes'</code> if you want to run the tests. If you answered anything else than <code>'yes'</code> in <code>DoMake</code> , then tests will automatically be skipped.
<i>DoSendMail</i>	If you want to send a report automatically, then answer <code>'yes'</code> . Otherwise you have to send the report manually.
<i>DoKeepFiles</i>	Answer <code>'yes'</code> if you want to keep the files after the execution of the script.
<i>CONFIGUREFLAG</i>	This variable is located a little further down the script. Here you can specify extra flags to configure. See Chapter 3 for help.

---

- It is nice to run the script through the cron.
- The script works as follows:
  - The script downloads MOLCAS (including the unsafe patches) with the method specified in the variable `$RETRIEVER`.
  - Installs MOLCAS.
  - Runs the tests.
  - The results are sent to *test@signe.teokem.lu.se*.
  - The test page (<http://www.teokem.lu.se/molcas/dev/test/>) is automatically updated.
- In order for the script to retrieve MOLCAS we need some authentication:
  - The password is specific for the IP-address, but otherwise the transfer is not secure.
  - The password is stored in `.pass`, which must have the permissions 400.
  - The password is generated from web, so the person who generate the password should have access to the developers part of the website.
- A remark: one day the checkinstall script must be rewritten so it includes support for auto updating of the checkinstall script, e.g. by adding it to the MOLCAS package, but this may be a security problem.

The nicest way to run `checkinstall` is to run it via the cron, in which case the script is executed automatically.



## Section 6

# Documentation

The MOLCAS environment provides several kind of documentation for its modules and programs. The MOLCAS manual is located in *doc/manual/* directory and consists of 4 parts: installation guide, users guide, tutorial and programmers guide. Installation guide and tutorial are 'static' documents, but the rest could be modified by developers.

The main source of documentation exist in  $\text{\LaTeX}$  format. However, not all  $\text{\LaTeX}$  constructions are allowed, because the manual is also converted to HTML format (in particular, the only allowed style file is *molcas.sty*). Some valid, but rare  $\text{\LaTeX}$  construction could be misinterpreted by *latex2html* converter.

### 6.1 Users guide

Each module has its own file (usually with the same filename as a program) which collect informations about the keywords and the data files produced by it.

At the same time the  $\text{\LaTeX}$  files are used to produce a documentation in plain text (ASCII) format used in the MOLCAS help system and in the GUI.

More peculiar, instead, is the way the MOLCAS environment produces the ASCII documentation. An interpreter program, called *mipi*, picks up such information from the  $\text{\LaTeX}$  files, provided that the proper syntax has been used.

During the installation process '*molcas mipi -l*' is executed, so the file *data/keyword.db* is generated from the documentation and used for fast access for the MOLCAS help system and the GUI to the documentation in plain text format.

If a line in a file in the users guide is started with `%%` signs, it is considered as a command.

Valid commands are:

---

---

<i>File</i>	<i>Contents</i>
<code>%%Description:</code>	a header for description of the program
<code>%%Keyword:</code>	a header for a keyword definition followed by the name of the keyword and a flag (basic—advanced—obsolete) which defines the priority of the keyword. This flag is used in MOLCAS GUI to separate compulsory and other keywords.

%%+                   Line which defines a free text with values for the above keywords. If a '+' sign is followed by space, this line inside a paragraph will be not reformatted.

---

The following example shows the general description of the MCLR module and of a specific keyword:

```
%%Description:
%%+This program calculates the response of a SCF or MCSCF wave function
%%+and related second order properties.
\item[LOWMemory]
%%Keyword: LowMemory advanced
%%+Lowers the amount of memory used, by paging out the CI vectors on disk.
%%+This will lower the performance, but the program will need less memory.
Lowers the amount of memory used, by paging out the CI vectors on disk.
This will lower the performance, but the program will need less memory.
```

The ASCII part of the documentation could be located in any place in the L<sup>A</sup>T<sub>E</sub>X documentation, but it is recommended to place it close to the definition of corresponding keyword. Note, that quite often the L<sup>A</sup>T<sub>E</sub>X and the ASCII description of a keyword are not identical - the definition in ASCII format should be independent on other text in the manual, and of course, shouldn't contain L<sup>A</sup>T<sub>E</sub>X constructions. Also, the name of a keyword in the L<sup>A</sup>T<sub>E</sub>X documentation and in the ASCII could be written with different cases, if it can improve the readability.

The reason we at all want to border with two formats in the documentation is that MOLCAS comes with a command line help tool, which requires the ASCII format in order to be easily readable. If you want to use this help tool, you simply issue a command like:

```
molcas help scf
```

to get a list of available keywords for the SCF module, or

```
molcas help scf charge
```

to get further help on the keyword CHARGE in the SCF module.

The plan is to extend this feature, so it can be used to make a consistency check of an input file without running MOLCAS itself, e.g. to get information whether inclusion of one keyword requires additional keywords or data.

## 6.2 Examples in manual

It is possible to place examples directly to the molcas manual, and they will be extracted during the installation.

In order to mark the area in a L<sup>A</sup>T<sub>E</sub>X file use markers `%%To_extract{filename}` (to begin) and `%%To_extract` (to finish selection). The filename can contain subdirectories. If the filename started from / it will be placed into directory, related to /molcas/ home. Otherwise, it will be placed into directory, related to `$MOLCAS/doc/examples/`.

In order to include the \$ sign, e.g. when linking \$WorkDir , use the `{\}`.

All line inside the marked area started from back slash will be removed, as well as percent signs.

Example:

```
%%To_extract{SCF/SCF.input}
\begin{inputlisting}
  &SCF &END
%%* This line is invisible in documentation
End of input
\end{inputlisting}
%%To_extract
```

### 6.3 Source code documentation

In MOLCAS we have the possibility to put documentation inside the source code, so it could be extracted, and it has actually been used during preparation of this manual.

To include this kind of documentation into the source code, you have to put some comment lines into your FORTRAN or C code. This kind of documentation should consist of separated fields defined in 'SGML-like' ('HTML-like') format.

```
* <TAG_NAME> text </TAG_NAME>
```

Tag names are case insensitive. The treatment of these tags is done in the same way as interpretation of HTML code: if the tag name is unknown for the system (or there is no the closing tag) this field is ignored.

The main tag - `< DOC >` describes a documentation section in the code, and all other tags must be located inside the `< DOC > ... < /DOC >` region. It is possible to have several documentation sections inside a single file.

Text inside each field could contain L<sup>A</sup>T<sub>E</sub>X constructions.

The list of valid tags is:

---



---

<i>File</i>	<i>Contents</i>
<i>Name</i>	a field to define the name of the subroutine
<i>Syntax</i>	a field to define the syntax of routine call
<i>Arguments</i>	a field to define the list of arguments
<i>Purpose</i>	a field to describe the purpose of the routine
<i>Dependencies</i>	a field to list dependencies
<i>Author</i>	a field to define the name of author(s)
<i>Modified_by</i>	a field to define the list of modifications
<i>Side_Effects</i>	a field to describe side effects

*Description* free text with description, examples, etc.

---

An example:

```
*****
*
* <DOC>
*   <Name>Check</Name>
*   <Syntax>Call Check(ireturn)</Syntax>
*   <Arguments>
*     \Argument{ireturn}{short description}{Integer: the return code}{out}
*   </Arguments>
*   <Purpose>Driver for check module</Purpose>
*   <Dependencies>Call Process</Dependencies>
*   <Author>V. Veryazov</Author>
*   <Modified_by></Modified_by>
*   <Side_Effects>none</Side_Effects>
*   <Description>
*     Main driver for check module
*   </Description>
* </DOC>
*
*****
```

There is a script *defdoc* to create a default template with source documentation for a FORTRAN code. The command `molcas defdoc filename.f` put a template for source documentation inside file *filename.f*. The original file is copied to *filename.f.bak*. Please notice that this template has the tag `< DummyDOC >` instead of `< DOC >` so it will ignored until this is changed.

By default the source code documentation is not regenerated by `make`. To force updating of this part of documentation one should use the command `make doc`.

The source code documentation is used for two purposes: first of all, it is included into this guide; secondly it can be retrieved by using the command

```
molcas help >src name
```

where 'name' is a name of a subroutine.

## 6.4 XML formatted description of keywords

New specification of XML formatted description of keywords is needed to generate GUI widgets. Note, that the following specification is preliminary, and is a subject of modifications and improvements.

General rules:

- All XML tags are located in documentation. (doc/manual/users.guide/\*.tex)

- All XML related lines started from
- All XML fields are UPPERCASED
- XML tags contains specifiers `< TAGKEY1 = "Value1" KEY2 = "Value2" >`

During an installation all XML information is extracted from documentation and placed to a file `data/keyword.xml`

### 6.4.1 Specification of XML input in molcas

**Main XML tags:** MODULE, KEYWORD, SELECT, GROUP, HELP

---



---

<i>Keyword</i>	<i>Meaning</i>
MODULE	definition of a module (can contain subtags HELP, KEYWORD, SELECT, GROUP)
KEYWORD	definition of a keyword (can contain subtags HELP, VALUES)
SELECT	definition of a group of keywords (radio box selection) (can contain subtags HELP, KEYWORD)
GROUP	definition of a group of keywords (like CHOLESKY/End of CHOLESKY) (can contain subtags HELP, KEYWORD)
HELP	help text

---

Tag MODULE contains specifiers:

---



---

<i>Keyword</i>	<i>Meaning</i>
NAME	(compulsory, UPPERCASED name of a module)
APPEAR	(optional - if missed equal to NAME, on-screen name of the module)
MEMBER	(optional, coma separated list of parent screens with predefined scenario)

---

Tag `< HELP >`any text`< /HELP >` contains a help description.

Example:

```

... in the beginning of scf.tex
%%%<MODULE NAME="SCF" APPEAR="HF/DFT" MEMBER="HF,HF-GEOMETRY,CAS,CAS-GEOMETRY">
%%%<HELP>
%%+this is a text as it is now in description of module
%%%</HELP>
...
... at the end
%%%</MODULE>

```

Tag KEYWORD contains specifiers:

---



---

<i>Keyword</i>	<i>Meaning</i>
MODULE	(same as module NAME, this field is using for double check and might be removed in a future)
NAME	(compulsory, UPPERCASED name of a keyword. one WORD only!)
APPEAR	(optional (if omitted - the same as NAME), on-screen name of the keyword)
LEVEL	(compulsory, indicates the complexity of a keyword)
KIND	(compulsory, one of the predefined types)
SIZE	(optional/compulsory depending on TYPE, size of array)
LIST	(compulsory if TYPE=CHOICE, coma separated list of choices)
MEMBER	(optional, name of GROUP or SELECT block. This field is using for double check and might be removed in a future)
EXCLUSIVE	(optional, coma separated list of keywords, which are exclusive for this keyword)
REQUIRE	(optional, coma separated list of keywords, which must be set, in order to use this keyword, or (if a keyword NAME starts with '!') must be unset)  Warning! undecided feature: a possibility to have && and    operators for the value.
DEFAULT_VALUE	(optional, default value (applies for all numbers in array))
MIN_VALUE	(optional, min value (applies for all numbers in array))
MAX_VALUE	(optional, max value (applies for all numbers in array))
DEFAULT_VALUES	(optional, default values)
MIN_VALUES	(optional, min values)
MAX_VALUES	(optional, max values)
INPUT	(optional keyword with the value REQUIRED)
WINDOW_SIZE	(optional, the number of showing rows in UI (applies for keywords who need more than one row in UI))

---

Attribute KIND is used to specify the type of the information which follows after a keyword.

KIND must be one of the following:

---



---

<i>Keyword</i>	<i>Meaning</i>
SINGLE	single (no values)
INT	integer
INTS SIZE=N	integer array(constant size)
INTS_COMPUTED SIZE=CONST	integer array(computed size): N, A(CONST*N)
INTS_LOOKUP SIZE=FIELD_NAME	integer array(lookuped size)
REAL	real
REALS SIZE=N	real array(constant size)
REALS_COMPUTED SIZE=CONST	real array(computed size): N, A(CONST*N)
REALS_LOOKUP SIZE=FIELD_NAME	real array(lookuped size)
CHOICE LIST=LIST	string(fixed values)
STRING	string(any)
STRINGS SIZE=N	string array

Computed size arrays are used for a common type of input in molcas. On a first line user specifies a number, and on the following lines - array, which has a dimension (N\*Constant), where Constant is specific for the keyword.

Lookuped array is most controversial construction. Some molcas inputs (might) depend on a calculation, for example, the input for Occupied, RAS2 depends on the number of symmetry operations. It is assumed, that this information is available at the moment of generation of user input.

For KIND=CHOICE LIST can be written as a coma separated list, or with aliases in a format "value:explanation" in this case explanation will be visible on screen, but the value will go to the output example: LIST="0:silent run,1:short output,2:verbose output"

LEVEL must be one of the following

<i>Keyword</i>	<i>Meaning</i>
BASIC	keyword appears on all screens
ADVANCED	appears only if advanced mode is on
DEV	GUI should ignore this keyword
GUI	GUI only keyword (with no influence to molcas input)
NOTIMPLEMENTED	default

example:

```
%%% <KEYWORD MODULE="GRID_IT" NAME="TITLE" KIND="STRING" LEVEL="BASIC">
%%% <HELP>
%%+ One line following this one is regarded as title.
%%% </HELP></KEYWORD>
```

```
%%% <KEYWORD MODULE="GRID_IT" NAME="ORBITAL" KIND="INTS_COMPUTED"
%%%   SIZE="2" MIN_VALUE="1" LEVEL="ADVANCED">
```

SELECT tag groups KEYWORDS into radio box.

---



---

<i>Keyword</i>	<i>Meaning</i>
MODULE	(same as module NAME)
NAME	(compulsory, UPPERCASED name of a keyword)
APPEAR	(optional, on-screen name of the keyword)
CONTAINS	(compulsory, coma separated list of keywords)

---

example:

```
%%% <SELECT MODULE="SCF" NAME="ELECTRON_COUNT"
%%%   APPEAR="Electron count selection" CONTAINS="NONE,CHARGE,AUFBAU">
%%% <KEYWORD MODULE="SCF" NAME="NONE" APPEAR="Intelligent default"
%%%   KIND="GUI" LEVEL="BASIC" EXCLUSIVE="CHARGE,AUFBAU">
%%% <HELP> ... </HELP></KEYWORD>
%%% <KEYWORD MODULE="SCF" NAME="CHARGE" APPEAR="Charge"
%%%   KIND="INT" LEVEL="BASIC" EXCLUSIVE="NONE,AUFBAU">
%%% <HELP> ... </HELP></KEYWORD>
%%% <KEYWORD MODULE="SCF" NAME="AUFBAU" APPEAR="Charge"
%%%   KIND="INTS" SIZE="3" LEVEL="BASIC" EXCLUSIVE="NONE,CHARGE">
%%% <HELP> ... </HELP></KEYWORD>
%%% </SELECT>
```

Tag INPUT=REQUIRED is used to specify that the keyword is compulsory, and so processing of the input is impossible unless this keyword gets a value.

GROUP tag contains a set of grouped keywords

---



---

<i>Keyword</i>	<i>Meaning</i>
MODULE	(same as module NAME)
NAME	(compulsory, UPPERCASED name of a group)
APPEAR	(optional, on-screen name of the group)

KIND	(compulsory, have a value - RADIO, BOX, BLOCK)
LEVEL	(compulsory, have a value ADVANCED or BASIC, applies for all members)
KIND	(compulsory, have a value - RADIO, BOX, BLOCK)
WINDOW	(optional, has a value of - POPUP, INPLACE, TAB, the default value is POPUP)

---

KIND in the GROUP tag can have values:

- RADIO - all keywords inside the group are exclusive.
- BOX - all keywords inside the box are grouped together, however, this tag is used only for visualization, and there is no any specific output in generated Molcas input.
- BLOCK means that these keywords are members of labeled block in the molcas input, like 'CHOLESKY ... End Of CHOLESKY'. In this case, if at least one of keyword is GROUP is selected, all keywords in this group will have a header NAME, and a footer 'End of NAME'.

example:

```
%%% <GROUP MODULE="SEWARD" NAME="CHOLESKY"
%%%   APPEAR="Cholesky integrals" KIND="BLOCK" LEVEL="ADVANCED">
%%% <KEYWORD NAME="CHOT" APPEAR="Threshold"
%%%   KIND="REAL" >
%%% <HELP> ... </HELP></KEYWORD>
%%% </GROUP>
```

WINDOW in the GROUP tag can have values:

- POPUP - It is the default value. The UI is the same of the group and a button with label "...". After clicking the button, its subkeywords will be shown as a separate dialog.
- INPLACE - All its subkeywords will be shown in the current UI.
- TAB - All its subkeywords will be shown separately as a new tab.

EMIL commands also have XML specification.

example:

```
%%% <COMMAND NAME="DO" APPEAR="Start loop" FORMAT="DO WHILE">
```



## Section 7

# Use of Utilities

MOLCAS contains several utility directories (see also section 2.2). Whenever a utility is available, it is preferable to use the utility rather than writing a new code yourself. Often one should call an interface rather than the utility itself; one example being `getmem`. The reason for this convention is that if one makes changes in the utility, we'll only need to adjust the call from the interface, not in several hundred places throughout the code. This of course assumes that the call to the interface stays the same.

### 7.1 List of utilities

There are different types of utilities. The main categories are:

- Common subroutines such as tools for printing, matrix multiplication, etc.
- Routines which are needed by more than one program, but which on the other hand still performs some specific task not widely needed.
- Utilities which serve as an interface to another program such as interfaces for Global Arrays and Aces2.

Currently the MOLCAS distribution contains the following utility directories:

---

---

<i>File</i>	<i>Contents</i>
<code>aces2_util</code>	Interface to the ACES II.
<code>amfi_util</code>	Utilities for calculating atomic mean field integrals.
<code>blas_util</code>	Mathematical library with optimised routines for common tasks such as matrix multiplication.
<code>casvb_util</code>	CASVB code.
<code>cholesky_util</code>	Core and interface routines for Cholesky decomposition.
<code>clones_util</code>	Wrappers for MOLPRO.

<i>dtraf_util</i>	Direct transformation utilities.
<i>egp_util</i>	Effective Group Potentials utilities.
<i>essl_util</i>	Another mathematical library.
<i>integral_util</i>	Utilities for integral calculations.
<i>io_util</i>	I/O related utilities.
<i>lapack_util</i>	A mathematical library.
<i>lucia_util</i>	Code from the LUCIA program. Needed for the CI-routines.
<i>memory_util</i>	Wrappers for Global Arrays (i.e. memory).
<i>molcas_ci_util</i>	Our own CI related stuff.
<i>molpro_util</i>	MOLPRO utilities.
<i>nq_util</i>	Numerical quadrature utilities.
<i>parallel_util</i>	Utilities used in parallel calculations.
<i>pcm_util</i>	PCM code.
<i>property_util</i>	Code for calculations on properties.
<i>quadpack_util</i>	Some other quadrature utilities.
<i>runfile_util</i>	Utilities used for manipulating the <i>runfile</i> .
<i>rys_util</i>	Integration stuff.
<i>system_util</i>	Utilities for system related operations.
<i>transform_util</i>	Utilities for transformation of the two-electrons integrals.
<i>util</i>	Unsorted routines - should be moved into specific directories.

---

## 7.2 Memory manager

In MOLCAS we use a memory manager, because it is hard to make dynamic allocations of memory in FORTRAN. One can create a memory manager in two ways:

1. You can create one big global array and decide the size on compile time.
2. You can include C-routines, which can do malloc calls, in the program:

```
pointer = (int) malloc(sizeof(int)*length_of_array);
```

to allocate an array with length `length_of_array`. If successful malloc returns a pointer to the allocated memory.

Similarly you free memory with

```
free(pointer);
```

When the memory manager is created this way, a FORTRAN wrapper is needed.

Which arrays should be allocated dynamical? Even arrays with fixed size preferably should be allocated via the memory manager if their size exceed say, 50 elements, because the stack can be of limited size. The stack can be used for static allocations, whereas dynamic allocations goes to the free memory.

MOLCAS uses Global Arrays for memory managing (the manager is of the second kind described above). This memory manager takes care of holes in the memory (e.g. from deallocation) and boundary check.

### 7.2.1 How to get memory

```
Call GetMem('Label', 'Action', 'Type', Pointer, Length)
```

where

**Label** is a string of your choice.

**Action** is either 'Allo', 'Free', or 'List' (check of memory).

**Type** is either 'Real' or 'Inte' depending on whether the variables to be stored in the array are reals or integers.

**Pointer** is an integer with the address of the array.

**Length** is an integer describing the number of elements.

A real example taken from *src/rasscf/rasscf.f* is:

```
Call GetMem('FI', 'Allo', 'Real', LFI, NTOT1)
```

### 7.2.2 How to use the memory

The use of an array allocated with the memory manager depends on whether it is an array of reals or integers:

**Real array** Address with Work(Pointer).

**Integer array** Address with iWork(Pointer).

Be aware that the addresses in Work and iWork are in C-counting style, so the first element of the array is Work(Pointer + 0), the second element is Work(Pointer + 1), etc.

An example is:

```
Length = 100
Call Getmem('Array', 'Allo', 'Real', iPointer, Length)
Do I = 0, Length - 1
  Work(iPointer + I) = 1.0D0
Enddo
...
Call Getmem('Array', 'Free', 'Real', iPointer, Length)
```

### 7.3 I/O Utilities

Why do we use I/O utilities instead of making direct statements in the code? Let's first take a look at the two alternatives:

**Use standard open statement** This is useful for small files. The file name must be a symbolic link and not the UNIX name of the file, e.g.

```
open(Unit=Lu0rb, File='INPORB', ...)
```

This is the way to do it in MOLCAS, because then porting to new platforms is much easier.

Function `isFreeUnit(Unitnumber)` - Unitnumber is an integer - returns the first free unit greater than the unit number.

Why is this standard method not good in general? If you have a binary file with direct access, then the buffering in FORTRAN is not perfect, so it is useful to use a C-program to do this.

**Use of utilities** One can use either a home made set of utilities or a standard packages like Global Arrays (in the same manner as with `getmem`).

For historical reasons in MOLCAS we use home made utilities. These are placed in `src/io_util/`. The routines in the I/O utilities can be divided into three levels:

**Lowest level** `cio.c` is the file containing the actual utility written in C-code. There should not be any reason to be concerned with these routines.

**Middle level** Files like `aixcls.f`, `aixrd.f`, and `aixopn.f`. These routines is a wrapper between the FORTRAN code and the C-code. In addition some extra checking is done.

**Highest level** Files like `ddafile.f`, `idafile.f`, and `daname.f`. These are the routines that is actually called from your code.

#### 7.3.1 How to open a file

Sometimes it is necessary to split files into pieces due to platform limitations (defined by `$MOLCASKDISK`). For this reason there exists two routines for opening files: `daname.f` (for single files), and `daname_mf.f` (for multifiles).

In case of a multifile, the utilities will automatically calculate which subfile to use, but it must (currently) be specified upon opening of the file whether the file is a multifile or a single file.

#### 7.3.2 Reading and writing

Depending on what kind of data you want to read or write, you should use one of the following files:

**cdafle.f** The data consists of characters.

**idafile.f** The data consists of integers.

**ddafile.f** The data consists of real\*8 (double precision) numbers.

Except for the data type the subroutines are identical. Some of the parameters in these routines are actually obsolete now.

The call to these routines is:

```
call iDaFile (Lu, iOpt, Buf, lBuf, iDisk)
```

**Lu** The FORTRAN unit number.

**iOpt** Which kind of read or write do you want? The main choices are 1 for write and 2 for read.

1 Synchronous write.

2 Synchronous read.

0 Dummy write, which can be used to find some info in the file, and the file pointer will then be located after the info. It is not recommended to use dummy write, though; it is much better to set up a formula.

99 Dummy read. If something is wrong then you usually get an error message. Dummy read ensures that the code does not hang, it just return 0 or 1 depending on success or not.

## 7.4 Parallelisation

In MOLCAS we do not use MPI or some other means of parallelisation directly. Instead we use it through Global Arrays, because it hides the architecture dependent stuff, e.g. on shared memory machines, you can sometimes do some fancy stuff, because access times to other nodes memory are very fast compared to those on a distributed machine.

When we parallelise the code, we will always assume that the program will be run on a distributed machine to guarantee that the parallelisation works on all parallel computers. Global Arrays will help improve the performance on SMP machines, if this is possible.

Furthermore, we will never call Global Arrays directly, but call wrapper routines instead. These wrapper routines include C preprocessing commands so the parallel overhead is left out for pure serial installations. Furthermore this also allows MOLCAS to be compiled on architectures, where Global Arrays might not have been ported to yet.

For the time being all parallelisation is coarse grain to allow for maximum performance on distributed machines, but we might later on add some fine grain parallelisation, e.g. by using the OpenMP directives.

The aim at increase performance on distributed machines also makes it very important to avoid unnecessary communication. Communicated items should take at least one to two orders of magnitude longer to calculate than communicate!

### 7.4.1 Example: A very simple one

Let us start with a very simple serial example:

```
Do i = 1,n
  a(i) = ...
Enddo
```

where the determination of  $a(i)$  could involve a call to a subroutine.

The same piece of code written for parallel execution would be:

```
Call Init_Tsk(id, n)
Call FZero(a, n)
98 If (.NOT. Rsv_Tsk(id,i)) Goto 99
C Do i = 1,n
  a(i) = ...
C Enddo
  Goto 98
99 Continue
Call Free_Tsk(id)
Call GADsum(a,n)
```

Step by step, what is happening is:

**Call Init\_Tsk(id, n)** This will initialise the task list.  $id$  is some unique integer specifying which task list, and  $n$  is the number of tasks. The task list will be accessible by all the nodes. At this stage all tasks are marked by 0. Each time a task is distributed to a node, this will no longer be true for this task, and it is in this way possible to keep track of which tasks already have been done, and which still remain to be done.

**Call Fzero(a, n)** Because each node will only calculate a few of the entries in the array  $a$ , then we need to make sure all the rest are zero. This is done by initialise all entries to zero.

**If (.NOT. Rsv\_Tsk(id,i)) Goto 99** Rsv\_Tsk will return the .TRUE., if there is anymore tasks to do for the task list with identification  $id$ , and the task number will be stored in  $i$ . If all tasks have been done, then Rsv\_Tsk returns .FALSE. and forces the program out of the loop. Global Arrays tries to find the next zero in the task list by means of atomic read.

Notice that the original serial loop (commented out) has been replaced by two Goto-statements.

**Call Free\_Tsk(id)** Tells that the task list  $id$  will not be needed any longer.

**Call GADsum(a, n)** Makes a global double precision summation of all nodes contributions to the  $a$  array. This way of collecting the various contributions is the reason to initialise the array before the loop.

### 7.4.2 Example: including non-parallelisable task

Sometimes one has a loop which is not parallelisable, because the calculation of it is too fast compared with the communication overhead, which would otherwise be created. Sometimes, though, it is still possible to parallelise it.

Take a look at this serial code:

```

Do j = 1,m
  b(j) = ...
Enddo
Do i = 1,n
  a(i) = ...
Enddo

```

where the first loop over  $b(j)$  is "very fast". An example which would fit this could be one electron integrals in  $b$  and two electron integrals in  $a$ .

By including the whole  $b$ -loop as one single task, it is possible to parallelise both loops:

```

Call Init_Tsk(id, n+1)
Call Fzero(a, n)
Call Fzero(b, m)
98 If (.NOT. Rsv_Tsk(id,i)) Goto 99
If (i .LE. n) Then
  a(i) = ...
Else
  Do j = 1, m
    b(j) = ...
  Enddo
Endif
Goto 98
99 Continue
Call Free_Tsk(id)
Call GADsum(a,n)
Call GADsum(b,m)

```

Notice that in the call to `Init_Tsk` the number of tasks is not 'n+1'.

### 7.4.3 Example: a very long loop

Lets say that we have a loop with a billion iterations, then it would create an unnecessary amount of overhead. Instead we can create compound tasks, i.e. tasks where more than one entry in  $a$  is calculated per task.

Doing this, we should make sure that the first tasks are relatively big, and the later ones are relatively small. In that way, we make sure that no CPU will be idle longer time than it takes to calculate one of the small tasks, and thus create the best possible load balancing.

Another trick to ensure as good a load balancing as possible is to make the number of very big tasks a multiple of the number of CPUs.

### 7.4.4 Example: parallelisation with I/O within a loop

This situation could arise e.g. if a program is parallelised within the iteration loop. If the parallelised part consists purely of floating point operations then there is no difference compared to the examples above.

On the other hand, if the tasks involves writing to disk (and of course reusing these informations in subsequent iterations), we will end up with a very poor parallelisation, if we do not try to make sure that each node primarily does the same tasks as in the previous iteration.

This is done by creating a private priority list and picking tasks from this one rather from the complete task list. First of all, we create a task list as in the previous case, but only first time we enter the parallel section. In addition the following steps are necessary:

- In the first iteration *Init\_PPList* is called, and in the subsequent iterations *ReInit\_PPList* is called.
- In the same manner a global task list is created with *Init\_GTList* and *ReInit\_GTList*.

The private priority list describes in which order each node should try to perform the tasks for optimal performance. In case a node has done all the tasks of the previous iteration, then it will try to reserve the first task not done by any of the other nodes yet. This will of course generate some extra I/O, but this will be performed when the node otherwise would have been idle anyway, and the node will then be able to reuse the data in the next iteration. Experiences with the SCF code show that after a few iterations, the distribution of tasks will enter into a steady state mode.

The global task list will as in the previous examples keep track of which tasks have been done, and which still have to be done.

The example below has been taken from *src/scf/drv2el.scf.f*:

```

      If (FstItr) Then
        Triangular = .TRUE.
*      Init_TList is a "local" replacement of Init_Tsk
        Call Init_TList(Triangular)
        Call Init_PPList
        Call Init_GTList
      Else
        Call ReInit_PPList(Semi_Direct)
        Call ReInit_GTList
      End If
*      big loop over individual tasks, distributed over individual nodes
10 Continue
      If (.Not.Rsv_GTList(TskLw,TskHi,iOpt,W2Disc)) Go To 11
*      ... Do some work ...
      Goto 10
11 Continue

```

#### 7.4.5 Parallel housekeeping

The first issue you encounter with running a module in parallel is handling of input and output files. All parallel copying of files is handled by the program *parnell.exe*. Before a module starts, MOLCAS will distribute the files listed in *module.prgm*, with an 'i' at the end, from the master to all the slaves. When a module stops, the output is only collected from the master. Any communication during the parallel execution should be done through GA calls.

When a module is a script, e.g. executing some MM code in a QM/MM procedure, it is run in serial, that is only on the master. To ensure that any files not occurring in *module.prgm* are available for subsequent use, one can use *parnell.exe* directly to copy the necessary files:

```

#!/bin/bash
/path/to/some/code < input > output

```

```
/path/to/mpiexec -np $CPUS $MOLCAS/bin/parnell.exe c <mode> file1 file2
```

The flag `jmodej` is a number from 0 to 3 which corresponds to the EMIL commands SAVE (0), COPY (1), COLLECT (2), and CLONE (3).

Another issue is the use of `systemf()` which forks a process. Within MPI, this is not supported and should not be used as this could lead to problems on some systems. Instead, the proper way to call another process is to exit the current running module and start your program as a different module.

## 7.5 Cholesky decomposition

### 7.5.1 Overview

The Cholesky integral representation is given by ( $\alpha$ ,  $\beta$ ,  $\gamma$ , and  $\delta$  denote atomic orbital indices or, more precisely, symmetry orbital indices)

$$(\alpha\beta|\gamma\delta) = \sum_{J=1}^M L_{\alpha\beta}^J L_{\gamma\delta}^J, \quad (7.1)$$

where  $L_{\alpha\beta}^J$  denotes the  $\alpha\beta$ -element of the  $J$ th Cholesky vector. This representation can be computed by program Seward.

Due to screening, not all elements of the Cholesky vectors are stored. Rather, the dimensions of the vectors fall into so-called reduced sets in which all elements that are spanned by previous vectors have been eliminated. The largest of these sets is therefore the first set which excludes all elements that are not needed to represent the integral matrix with the specified precision (i.e. the decomposition threshold). All later reduced sets refer back to the first one which, in turn, refers back to full storage through index arrays stored on disk after the decomposition utility has been executed (from within Seward).

The interface to the Cholesky utility from other modules therefore contains three basic components,

1. initialization:
  - open vector and index storage files
  - allocate and initialize index arrays
2. vector read
3. finalization:
  - close vector and index storage files
  - deallocate index arrays

These components are described in more detail below.

### 7.5.2 The initialization step

Initialization is done by calling the routine

$$\text{Cho\_X\_Init}(\text{irc}, \text{FracMem})$$

where  $\text{irc}=0$  on exit if successful completion. Note that this routine does *not* halt execution if an error occurs. Thus, the caller must take appropriate action if a non-zero code is returned. Also note that a non-zero return code most likely implies that the information has not been properly initialized. The second argument, `FracMem`, is an input parameter (double precision, between 0.0 and 1.0) specifying the fraction of available memory that will at most be allocated as buffer to reduce the amount of I/O. The initialization routine will then allocate and read vectors into the buffer. Specifying a non-zero `FracMem` is only relevant for iterative methods (e.g. SCF or RASSCF) where the vector files are read more than once. Moreover, the buffering is likely to improve performance mainly for parallel runs where the vectors are distributed over the nodes. The Cholesky reading routines (see below) automatically detects the existence of the buffer and, whenever possible, reads vectors from there rather than from disk. Thus, except for specifying `FracMem`, there is no need to worry about the buffer.

The following include files are affected by the initialization:

- `choptr.inc`  
pointers to index arrays stored in `iWork`
- `choorb.inc`  
orbital information
- `cholesky.inc`  
misc. decomposition information, including (small) statically allocated index arrays
- `chovecbuf.inc`  
information such as memory addresses needed for the reading routines to access the vector buffer (if allocated).

Some basic information stored in `cholesky.inc`:

- `nShell`:  
the number of shells.
- `nnShl_Tot`:  
the total number of shell pairs,  $\text{nnShl\_Tot} = \text{nShell} * (\text{nShell} + 1) / 2$ .
- `nnShl`:  
the number of shell pairs contributing to the diagonal according to prescreening,  $\text{nnShl} \leq \text{nnShl\_Tot}$
- `nSym`:  
the number of symmetries (irreps).
- `NumCho(i)`:  
the number of Cholesky vectors of symmetry  $i=1,2,\dots,\text{nSym}$ .

- NumChT:  
 $\sum_{i=1}^{nSym} \text{NumCho}(i)$ .
- MaxVec:  
 $\max_i \text{NumCho}(i)$ .
- MaxRed:  
the number of reduced sets.
- InfVec\_N2:  
second dimension of array InfVec (see below).
- nnBstR( $i, k$ ):  
reduced set dimension, symmetry block  $i$ . The last index  $k$  is just a memory pointer:  $k = 1$  points to the first reduced set,  $k = 2, 3$  are scratch locations. After initialization, all three locations are identical (first reduced set indices).
- iiBstR( $i, k$ ):  
 $\sum_{j=1}^{i-1} \text{nnBstR}(j, k)$ . The last index  $k$  is just a memory pointer as for  $\text{nnBstR}(i, k)$ .
- nnBstRT( $k$ ):  
 $\sum_{i=1}^{nSym} \text{nnBstR}(i, k)$ . The last index  $k$  is just a memory pointer as for  $\text{nnBstR}(i, k)$ .

Information stored in choorb.inc:

- nBas( $i$ ):  
the number of basis functions of symmetry  $i=1,2,\dots,nSym$ .
- iBas( $i$ ):  
 $\sum_{j=1}^{i-1} \text{nBas}(j)$ .
- nBasT:  
 $\sum_{i=1}^{nSym} \text{nBas}(i)$ .

Dynamically allocated index arrays; pointers and dimensions stored in choptr.inc:

- InfRed( $i$ )= $i\text{Work}(\text{ip\_InfRed}-1+i)$ :  
returns the disk address for reading info pertaining to reduced set  $i$ . Dimension:  $1\_InfRed=MaxRed$ .
- InfVec( $i, j, k$ )= $i\text{Work}(\text{ip\_InfVec}-1+MaxVec*InfVec\_N2*(k-1)+MaxVec*(j-1)+i)$ :  
InfVec( $i, 1, k$ ) returns the column index in the integral matrix (stored as first reduced set) of vector  $i$  of symmetry  $k$ .  
InfVec( $i, 2, k$ ) returns the reduced set to which this vector belongs.  
InfVec( $i, 3, k$ ) returns the disk address for reading this vector.  
InfVec( $i, 4, k$ ) returns the WA disk address for this vector (somewhat redundant).  
Dimension:  $1\_InfVec=MaxVec*InfVec\_N2*nSym$ .
- iSP2F( $i$ )= $i\text{Work}(\text{ip\_iSP2F}-1+i)$ :  
returns the full shell pair corresponding to reduced shell pair  $i$ . Dimension:  $nnShl$ .

- $\text{nnBstRSh}(i, j, k) = \text{iWork}(\text{ip\_nnBstRSh}-1 + \text{nSym} * \text{nnShl} * (k-1) + \text{nSym} * (j-1) + i)$ :  
returns the dimension of shell pair  $j$ , symmetry block  $i$ . The last index  $k$  is a memory pointer (1,2, or 3) as defined for  $\text{nnBstR}$  etc. above. Dimension:  $\text{l\_nnBstRSh} = \text{nSym} * \text{nnShl} * 3$ .
- $\text{iiBstRSh}(i, j, k) = \text{iWork}(\text{ip\_iiBstRSh}-1 + \text{nSym} * \text{nnShl} * (k-1) + \text{nSym} * (j-1) + i)$ :  
 $\sum_{l=1}^{j-1} \text{nnBstRSh}(i, l, k)$ .  
Last index  $k$  is a memory pointer (1,2, or 3) as defined for  $\text{nnBstR}$  etc. above. Dimension:  $\text{l\_iiBstRSh} = \text{nSym} * \text{nnShl} * 3$ .
- $\text{IndRed}(i, k) = \text{iWork}(\text{ip\_IndRed}-1 + \text{nnBstrT}(1) * (k-1) + i)$ :  
(The index  $k$  identifies the memory location as above.) Returns address in first reduced set of element  $i$  in the reduced set stored at location  $k$ . For  $k=1$ , however,  $\text{IndRed}$  returns the address within the full shell pair to which element  $i$  belongs. Dimension:  $\text{l\_IndRed} = \text{nnBstrT}(1) * 3$ .
- $\text{IndRsh}(i) = \text{iWork}(\text{ip\_IndRsh}-1 + i)$ :  
Returns full shell pair to which first reduced set element  $i$  belongs. Dimension:  $\text{l\_IndRsh} = \text{nnBstrT}(1)$ .
- $\text{nDimRS}(i, j) = \text{iWork}(\text{ip\_nDimRS}-1 + \text{nSym} * (j-1) + i)$ :  
Returns dimension of reduced set  $j$ , symmetry  $i$ . Dimension:  $\text{l\_nDimRS} = \text{nSym} * \text{MaxRed}$ .
- $\text{iRS2F}(i, j) = \text{iWork}(\text{ip\_iRS2F}-1 + 2 * (j-1) + i)$ :  
Returns AO indices ( $\alpha$  index for  $i=1$ ,  $\beta$  index for  $i=2$ ) for first reduced set element  $j$ .  
Dimension:  $\text{l\_iRS2F} = 2 * \text{nnBstrT}(1)$ .
- $\text{nBasSh}(i, j) = \text{iWork}(\text{ip\_nBasSh}-1 + \text{nSym} * (j-1) + i)$ :  
Returns dimension of symmetry block  $i$  within shell  $j$ . Dimension:  $\text{l\_nBasSh} = \text{nSym} * \text{nShell}$ .
- $\text{iBasSh}(i, j) = \text{iWork}(\text{ip\_iBasSh}-1 + \text{nSym} * (j-1) + i)$ :  
Returns offset to symmetry block  $i$  within shell  $j$ . Dimension:  $\text{l\_iBasSh} = \text{nSym} * \text{nShell}$ .
- $\text{nBstSh}(i) = \text{iWork}(\text{ip\_nBstSh}-1 + i)$ :  
Returns total dimension of shell  $i$ . Dimension:  $\text{l\_nBstSh} = \text{nShell}$ .
- $\text{iSOShl}(i) = \text{iWork}(\text{ip\_iSOShl}-1 + i)$ :  
Returns the shell to which basis function  $i$  belongs. Dimension:  $\text{l\_iSOShl} = \text{nBasT}$ .
- $\text{iShlSO}(i) = \text{iWork}(\text{ip\_iShlSO}-1 + i)$ :  
Returns index (not symmetry reduced) of basis function  $i$  in its shell (i.e. shell  $\text{iSOShl}(i)$ ). Dimension:  $\text{l\_iShlSO} = \text{nBasT}$ .

### 7.5.3 The read routines and related utilities

There are currently three routines available for reading the Cholesky vectors:

- $\text{Cho\_X\_VecRd}(\text{Scr}, \text{lScr}, \text{jVec1}, \text{jVec2}, \text{iSym}, \text{jNum}, \text{iRedC}, \text{mUsed})$
- $\text{Cho\_X\_GetVec}(\text{ChoVec}, \text{LenVec}, \text{NumVec}, \text{iVec1}, \text{iSym}, \text{Scr}, \text{lScr})$
- $\text{Cho\_X\_GetVFull}(\text{irc}, \text{RedVec}, \text{lRedVec}, \text{iVec1}, \text{NumV}, \text{iSym}, \text{iSwap}, \text{iRedC}, \text{ipChoV}, \text{iSkip}, \text{DoRead})$

While `Cho_X_VecRd` reads as many vectors as fit into array `Scr` and returns them "as is", `Cho_X_GetVec` reads a specified number of vectors and returns them in the reduced set storage defined by "location 2" in the reduced set index arrays discussed above. It should be noted that `Cho_X_GetVec` implements several different models for reading the vectors. Which algorithm is employed depends on the value of the integer `CHO_IOVEC`, which is set to 3 by the initialization routine. Although not recommended, this may be changed by the programmer by modifying `CHO_IOVEC` (stored in `cholesky.inc`) at the time of calling the read routine. (Warning: some algorithms might require a memory allocation for scratch storage!) The `Cho_X_GetVFull` routine returns vectors in full storage and may be used as a read/reorder utility or as reorder only. See the documentation of these routines for more details.

To set the index arrays at location `iLoc=2` or `3` as appropriate for reduced set `iRed`, use:

$$\text{Cho\_X\_SetRed}(\text{irc}, \text{iLoc}, \text{iRed})$$

This defines the following index arrays (see above):

- `nnBstRT(iLoc)`
- `nnBstR(:,iLoc)`
- `iiBstR(:,iLoc)`
- `nnBstRSh(:, :, iLoc)`
- `iiBstRSh(:, :, iLoc)`
- `IndRed(:,iLoc)`

For safety reasons, `iLoc=1` can not be set here since this would destroy the basic assumptions of most of the decomposition core routines. On successful completion, `irc=0` is returned.

There is a routine to copy between reduced set index locations. The routine is:

$$\text{Cho\_X\_RSCopy}(\text{irc}, \text{iRS}, \text{jRS})$$

where `iRS` and `jRS` refer to locations 1, 2, or 3. On successful completion, `irc=0` is returned.

Similarly, there is a routine to swap between reduced set index locations. The routine is:

$$\text{Cho\_X\_RSSwap}(\text{irc}, \text{iRS}, \text{jRS})$$

where `iRS` and `jRS` refer to locations 1, 2, or 3. On successful completion, `irc=0` is returned.

#### 7.5.4 The finalization step

Finalization is done by calling the routine

$$\text{Cho\_X\_Final}(\text{irc})$$

where `irc=0` if successful completion. Note that this routine does *not* halt execution if an error occurs. Thus, the caller must take appropriate action when a non-zero code is returned. Also note that a non-zero return code most likely implies that the information has not been properly finalized.

### 7.5.5 Cholesky decomposition utilities

There are three routines available for Cholesky decomposing positive (semi-) definite symmetric matrices:

- `CD_InCore(X,n,Vec,MxVec,NumCho,Thr,irc)`
- `CD_InCore_p(X,n,Vec,MxVec,iD,NumCho,Thr,irc)`
- `ChoDec(CD_Col,CD_Vec,Restart,Thr,Span,MxQual,Diag,Qual,Buf,iPivot,iQual,nDim,lBuf,ErrStat,Num`

As the names suggest, `CD_InCore` and `CD_InCore_p` Cholesky decompose a matrix already stored in core. If the matrix at hand is too large to store in core, `ChoDec` may be used instead. However, two external routines (`CD_Col` and `CD_Vec`) must then be supplied to `ChoDec`. See the documentation of these routines for more details.

## Section 8

# Creation of a New Program

In order to create a new program, the following files are needed:

- Source files
- .prgm file
- Tests
- Documentation
- Makefile.in (hopefully not, although)

### 8.1 Your first Molcas program

The molcas install can help you create a new module inside the MOLCAS environment. This script interactively ask you about the name of the module, the location of the source, etc. It then creates templates for the most commonly needed files, *main.f*, *module.tex* (in the documentation), and *module.prgm* (in *data*):

The steps in the creation of the program is:

1. Create a file (located in *src/your-program-name/*) with your source code, e.g.

```
Subroutine myfirst(iReturn)
Implicit Real*8 (a-h,o-z)
iReturn = 0
i = 2 + 2
If (i .ne. 4) call Abend()
Write (*,*) i
Return
End
```

There are strict rules about the value of the return code in MOLCAS. The list of available returncodes can be found in *src/Include/warnings.fh* file.

2. Run the molcas install command:
  - (a) Confirm that you want to install a new plug.in.

- (b) Choose the name of the directory containing the source code (the same name as the program). Please recall the name convention (see Section 2.2).
- (c) Specify where the source is located. Be aware that the script will only copy files with the extensions '.f' and '.inc'. These files will be copied to the destination `$MOLCAS/src/myprogram`, where 'myprogram' is the name you specified in the second step.

The script will not overwrite any files, so the template for `main.f` will only be generated, if there isn't any such file already.

The template for `main.f` have the following structure:

```
*fordeck myprogram/main $Revision: 6.1 $
  program main
  implicit double precision (a-h,o-z)
  Call Start('myprogram')
  Call myprogram(ireturn)
  Call Finish(ireturn)
end
```

- (d) The script exists with a list of suggested steps to continue with.
3. Modify the `doc/manual/users.guide/myprogram.tex` file, so it contains the relevant documentation for your program.
  4. Edit the `data/myprogram.prgm` file:
 

If you have any I/O then you need to add a line for each file used. See Section 3.5 for more details. If you do not have any I/O, you do not need to make any changes to the file created by the `install` script.
  5. Run `./configure` and `make`, and you have your program, `00sources`, `00dependencies`, `libmyfirst.a`, and `.stamp`.
  6. To run it type: `molcas myfirst inputfile`.

Now we in principle have a working program, which runs in the MOLCAS surroundings, and it can use all of the MOLCAS libraries.

## Section 9

# Debugging and Code Developing

### 9.1 Running through a debugger

If you have a problem with MOLCAS crashing during execution, it is often helpful to run the code through a debugger. You can't use the debugger directly with the executable, because the environmental is not set.

So in order to run MOLCAS through the debugger, you should execute MOLCAS as:

```
molcas MOLCAS_DEBUGGER=gdb input
```

or use other than gdb debugger.

The complete manual for using the debugger can be found for example through the man pages. Here we just want to mention the most useful commands in the GNU debugger (the same for most other debuggers) to analyze the core file.

---

---

<i>File</i>	<i>Contents</i>
<i>run</i>	Will start the program.
<i>where</i>	To print the trace of the calling of subroutines.
<i>quit</i>	Exit the debugger.

---

If you want to get more detailed information from the execution, you should recompile MOLCAS with the appropriate keywords. In order to do this, you need to run `configure` with the flag `'-speed debug'` (see also Section 3.8 for preparing the installation for rebuilding). Running this version through the debugger will supply informations on the "exact" line, where the crash occurs, you can add break points, and you can print variables, etc.

### 9.2 Other ways to debug

You can - at compile time - choose to switch on additional printing. `configure -debug` and `configure -trace` are the two possibilities:

---



---

<i>File</i>	<i>Contents</i>
<code>configure -debug</code>	<p>The code contains preprocessor code like:</p> <pre> #ifdef _DEBUG_     debug = .true. #else     debug = .false. #endif </pre> <p>This option activates the option, where <code>DEBUG</code> is set to true, which turns on additional print statements.</p>
<code>configure -trace</code>	This option activates trace of callings to subroutines.

---

A last possibility is to build MOLCAS with profiling, which is done with `configure -profiling`. Then after each execution the working directory will contain a file called `gmon.out` which can be analyzed with the `gprof` program. This generates a detailed profile of where the program spends time during the execution.

### 9.3 Core generation

Some time you might prefer to generate a core file in order to trace the problem. For example, if the code dies in some I/O operation, you don't know which routine initiated this call.

If you configured molcas with `-debug` flag (or set an environment variable `export MOLCAS_BOMB=Yes`), and the return code is higher than 20, instead of 'happy landing' molcas will generate an exception, forcing program to crash.

There is also a possibility to keep molcas job alive after finish (for example, to allow a developer to execute some commands before the job is terminating). If environment variable `MOLCAS_ZOMBIE` sets to `YES`, `auto` will sleep after the finishing of calculation until a termination of the job, or until user creates a file `zombie` in working directory.

### 9.4 Parallel debugging

When MOLCAS run in parallel, it is impossible to use `MOLCAS_DEBUGGER` variable. However, it is possible to attach debugger (e.g. `gdb`) to running program.

First of all, we need to export `MOLCAS_NAP` variable, and set it to the number of seconds, e.g. 30. This export can be done as an EMIL command before a program we would like to debug. If `MOLCAS_NAP` has been set, during initiation of a module, PID numbers for each nodes will be printed. During 'the nap time', a developer can run `gdb`, and use command `attach PID`, to connect executable and debugger. Use command `gdb` command 'continue' to run the calculation.

Note, that many problems with parallel execution of molcas are related to the fact that information, located at different working directories is different. It is a good idea to run calculation locally (but using 2 CPUs) and verify the content of `tmp_N` directories.

## 9.5 Problems with aggressive optimization

A common problem with development of any computational code is the selection of proper optimization level. Very often due to either bugs in compiler or unclear coding, a code can produce incorrect results with an aggressive level of optimization.

A script called `snooper` can locate a file (files), which require detuning or rewriting. In order to run this script, you have to create a test case, and install two versions of molcas, compiled with different optimization levels. The script builds molcas executables from the source code using different compiler flags and verifying the test.

a command `molcas snooper molcas.low_opt 555`, running from a MOLCAS installation where test number 555 fails, will use compiler options from *Makefiles* in the directory `molcas.low_opt` and search for a file which should be compiled with low optimization level.

Note, that this procedure is time consuming, and might fail if there are more than one file which must be detuned.

## 9.6 Utilities

Very useful for this purpose are the `find` and `edit` utilities. The former allows to find the location of a file inside the `src/` subdirectories. The main use of this utility is to locate the FORTRAN source files and the subroutines. In addition, the utility `edit` allows a direct access through a text editor to the desired file. Provided that the MOLCAS variable has been correctly defined, the access to these utilities is achieved typing at the prompt:

```
molcas find someName
```

where 'someName' is the name of the subroutine. `find` is a script located in the `sbin/` directory. The result of the command is the file that contains someName, e.g.

```
molcas find drvmo
```

will return "drvmo:grid\_it/drvmo.f". Currently only FORTRAN source code is supported for locating subroutines.

By executing

```
molcas edit name
```

the editor defined in the environmental variable `MOLCAS_EDITOR` (the default is `vi`) will be opened with the requested file. If 'name' is without '.f' then 'name' is considered a subroutine, and the file containing the subroutine will be opened. On the other hand if 'name' includes '.f', then 'name' will be considered a file, and then this file will be opened.

## 9.7 Memory allocation profiling

A developer can analyze memory consumption of a code, by compiling `src/memory_util/getmem.f` with flags.

If option `-D_GARBLE_` is in use, all allocated memory is filled (at the moment of allocation) by a special value, and during the release of the memory block, an information about actual use of this block is printed. Note, that this is a very time consuming operation.

Option `-D_MEM_PROF_` makes a log of all allocations and deallocations of memory. For each module, a file with name *memory\_profile* is generated. To analyze these files one can use a command `molcas mem_ana`.

## 9.8 How to check inter procedure consistency

With a file structure where all routines are in individual files some things are hard or impossible for the compiler to check. For example if two routines call one subroutine in different ways will not be detected and checking that the definition and use is consistent will not be done either. For this reason you can run the command `molcas monolith` to create a copy of the current molcas version and make a monolith for each module in the package. Each module will contain a copy of all utility routines as well, so this is not a cost effective way to organize MOLCAS.

Once the copy is made you can configure and make MOLCAS as usual. Please note that using very picky compiler flags like `pedantic` might produce too much warnings.

# Appendix A

## Makefiles

The standard structure of a Makefile is:

- Comments: the first character is '#'.  
• Variables are declared in the following way:

```
MOLCAS = /bin/7.6.dev
```

- If you want to use a variable, you must inclose it in curly brackets:

```
${MOLCAS}
```

- Targets are markers for what to do. For example 'make all' tells that you want to make what is associated with the marker 'all'. A marker is written in the Makefile as:

```
target:
```

There is a special marker 'default', which is the one that will be addressed if you execute 'make' without any arguments.

### A.1 How to define a target

You define a target by adding a line like:

```
name_of_target: ${OBJECTS}
```

where '\${OBJECTS}' is a list of objects in the order that they should be compiled, or it is a list of dependencies.

After this line you add a number of lines with the actions you want to perform for this target. Be aware that all lines with actions *must* be started with a tabulator. If you place a '@' in front of a command, e.g. @echo "Hello world" then the command will be executed without printing the command.

Each command is executed in it's own shell, i.e.

```
cd src/${@}  
${MAKE}
```

will go into a directory, and then execute 'make' in the old directory. What you probably wanted to do is:

```
cd src/${@}; ${MAKE}
```

which will execute 'make' in the “new” directory.

The target can also be a variable containing several items, e.g.

```

${UTILS}: ${DIRS} ${INC}
:
@echo "*** Building ${@}"
:

```

in *Makefile* (in the MOLCAS root directory). In the example above notice the '@' variable. It's a special variable that refers back to “current” variable - in this case the item from `${UTILS}` that is currently processed.

It is also possible to include other files in your Makefile (in the same way as you can put include files into your FORTRAN code). This can be done in two ways:

```
include file_name
-include file_name
```

The first way works both for UNIX make and for Gnu make, and this command will cause make to stop if the operation is not possible. On the other hand, the statement in the second line only works in Gnu make, and it means that it should try to include the file, but make should continue even if it is not possible.

Consider the line (in a Makefile belonging to a program, e.g. rasscf):

```

${EXE}: main.o .stamp ${LIBDIR}/libmolcas.a
      ${FC} -o ${@} ${LDFLAGS} main.o ${PLIB} ${ILIB} ${XLIB}
      ${MKEXE} ${@}

```

In the first line “main.o .stamp \${LIBDIR}/libmolcas.a” is the three things `${EXE}` (the name of the executable) depends on. 'main.o' is the result of the compilation of *main.f*, '.stamp' will be changed if any of the other FORTRAN files in the program directory have been changed, and '`${LIBDIR}/libmolcas.a`' will be changed, if any of the utilities have been changed.

The second line does the actual linking. Here `${@}` is the same as `${EXE}`, and `${LDFLAGS}` are the flags used for linking (see section 3.1). “main.o `${PLIB}` `${ILIB}` `${XLIB}`” are the files needed in order to be able to link.

## A.2 00sources

In MOLCAS *00sources* (in each of the source directories) contain the information on the source needed by make. It contains a few variable:

**FSRC** is a list of all the FORTRAN source files (excluding main.f).

**OBJ** is a list of all the object files (the list is written implicit as a rule how to come from the FORTRAN file name to the object file name).

**PLIBOBJ** tells which library the object files should go into.

**INCSRC** is a list of the include files needed in order to compile the files listed in **FSRC**.

### A.3 00dependencies

The *00dependencies* file (in each of the source directories) tells which files are needed on each object file.

### A.4 stdsuffix

This file is located in *src/Driver/* and defines the suffix rules for the makefiles (i.e. the target named '.SUFFIXES'):

```
.SUFFIXES: .c .f .o .a
```

meaning that C code must be taken care of before FORTRAN code, and this should be taken care of before objects and again before libraries. Or said in another way: libraries depend on objects, which in turn depend on FORTRAN (and C) files. One can also say that .f is younger than .o.

*stdsuffix* also contains information of how to go from a FORTRAN (or C) file to an object file:

```
%o: %f
    ${FC} -c ${FFLAGS} $<
```

This means that you should use the FORTRAN compiler ( $\{FC\}$ ) with the flags specified in  $\{FFLAGS\}$  on the current dependency ( $\{<\}$ ), i.e. the .f file.

Similarly there is a rule how to add a FORTRAN (or C) file to a library:

```
(%.o): %.f
    ${FC} -c ${FFLAGS} $<
    ${AR} cr ${@} $*.o
    @${RM} $*.o
```

The first action is the same as above. The second line tells how to add the object file to the archive (library file), and the third line deletes all the object files.

The last rule is how to add an object to a library:

```
(%.o): %.o
    ${AR} cr ${@} $*.o
    @${RM} $*.o
```

which is just the two last actions from the previous example.



## Appendix B

# Glossary

**Environmental variable** is a variable defined through the system, not internally in MOLCAS.

One can set an environmental variable with the command:

“`export MOLCAS=/usr/local/molcas/7.6.dev/`” or whatever command is used on the system at hand.

**Global Arrays** A package with utilities for e.g. memory management and parallel executions. The package is developed by Environmental Molecular Science Laboratory (financed by the U.S. government). You will need an agreement with EMSL to put it into your code, but then you get it for free and in open source. We have an agreement not only to use the toolkit, but also to include it in the distribution.

**Master version** is the official version of the particular revision of MOLCAS. A patch is for instance not officially applied until it is applied to the master version. The master version is available inside local network in Theoretical Chemistry Department in a home directory of user patch, and mirrored at MOLCAS developers’ homepage.

**Molcas root directory** is the top level directory of the MOLCAS installation, e.g.

*`/usr/local/molcas/7.6.dev/`*.

**Revision** is referring to the secondary version number, e.g. version 6.0.007 is version 6, revision 0, patch level 007.