Environment Modules

Environment modules are used on the LRZ HPC systems to manage the environment variables in shells and scripting languages. They also permit to provide multiple versions of packages.

- Introduction
- Module command options
- Spack Generated Modules
  - Naming conventions
  - Spack Branches and Releases
- Package Variables
- Specifics on the setup for LRZ HPC systems
  - Module classes
- Shells and scripting languages
  - Supported Shells
  - Using modules from Perl or Python
  - Generating new shells
  - Using modules in batch jobs
  - Additional utilities
  - Writing your own module files
- Versioning policies
- Support and sources of information on modules

Introduction

Modules provide an easy mechanism for managing the user's environment (PATH, MANPATH, and LD_LIBRARY_PATH, to name a few).

To make the application-specific settings for <package> (the denotation is abstract) available in the current shell, the user types the command

```bash
module load <package>
```

If, subsequently, the user types

```bash
module unload <package>
```

the application-specific settings for <package> will be removed again, and the application typically will become inaccessible or unusable.

On the LRZ HPC systems, setting up of the modules package is performed automatically at login. This includes

- making the `module` command (as described below) available
- loading a default set of module files. The exact set depends on the platform used.

Module command options

This section supplies an overview of the most important command options. Note that the `module` command is defined as an alias.

`module list:` This lists all the modules which are currently loaded into the users' environment, in the order in which they were loaded. With an additional -t option, only a single column is printed.

`module avail:` This option lists all the modules which are available to be loaded. Notice that most have version numbers associated with them. One of the versions is marked out as default; this is the version that is referred to when the version number is omitted in a module command. Adding the -d switch will only list the default versions of each module. Furthermore, limited pattern matching will also work, so

```bash
module avail gs
```

will print the list of available modules that start with the letters "gs". The -t switch will display the modules as a single column.

`module help:` This option requires a module name (with or without version number) as an argument. It will display additional information for the specified module; at minimum, a pointer to web documentation for the package should be delivered.

`module show:` This option requires a module name (with or without version number) as an argument. It will display the changes to the environment that would be done if the module were loaded.

`module load:` This option requires one or more (space separated) module names (with or without version number) as an argument. For example,

```bash
module load gaussian
```

will make the default version of the package "gaussian" available for your use. If you need a specific supplied version that is non-default, you need to specify the version number explicitly, for example:

```bash
module load gaussian/09,D,02
```

Please also note that
some modules may automatically load others if a package depends on some other package to be available. Unloading dependent modules may give you trouble!
• certain default modules should also not be unloaded, since your environment will not function properly without the settings performed there
• if you specify multiple modules, they will be loaded in the supplied order
• instead of "load", you can also say "add".

module unload: This option requires one or more (space separated) module names (with or without version number) as an argument. It will unload the specified modules, removing all previously supplied environment settings. Please note that
• for multiple modules to be removed, it is recommended to do this in the reverse order of loading, especially if there exist dependencies between two such modules
• it is not necessary to specify a version number - the loaded version will be removed even if not explicitly specified
• instead of "unload", you can also say "rm".

module search: This option requires a text pattern as an argument. Each module contains a module-whatis entry with a keyword list that will be searched for the provided pattern. All modules that contain a matching entry will then be listed. In particular, the module classes (with a colon appended) are always used as a keyword, so this facility can be used to search the module system by application classes:

module search libraries:

You can also say "apropos" or "keyword" instead of "search".

module switch: This option requires exactly two (space separated) module names (with or without version number) as an argument. The command performs a transition from the module specified as the first argument to that specified by the second one with a single command. Note that this is not exactly the same as an unload followed by a load, since all modules which were loaded after the first specified will also be reloaded, to assure that dependencies are correctly resolved. If this causes difficulties, you may need to revert to explicitly perform an unload/load sequence.

module purge: This command removes all modules from your environment. On the LRZ HPC systems, this causes the environment to become rather unusable. Therefore, after a purge, you should issue the command

unset LRZ_LOAD_DEFAULTS # or unsetenv for csh-like shells
module load lrz

before loading or unloading any further modules.

module use: This option requires an existing directory name (or multiple directory names) as an argument. It causes its argument to be added to the modules path, with the result that all module files stored within that directory become visible for the module command. The entries are all stored in the $MODULESPATH environment variable. The -a option can be used to append the directory to the path instead of prepending it. It is also possible to remove a path again by specifying it as an argument for the module unuse command.

module path: This option requires one module names (with or without a version number) as an argument. The complete pathname of the module-file will be printed as a result.

Spack Generated Modules

With the installation of SuperMUC-NG, LRZ gradually changes its software deployment process from manually built packages to automated builds with Spack.

Spack is a tool to automatize and manage builds of software packages. Collections of packages built with Spack will be released on LRZ clusters periodically and are available through the environment module system. The latest release of packages will be listed at the bottom when using the 'module av' command.

Naming conventions

Environment modules generated by Spack use the naming scheme

<package name>/<full version number>[-<suffix1>][-<suffix2>][...]

It is highly recommended to use the complete module name when using the spack generated modules. If you need a specific supplied version, you will have to specify the version number explicitly.

For example, asking for how many FFTW modules are available, do the following

> module av fftw

You will get a list which looks (can differ depending on which system and software stack a user is using) like the following,

fftw/3.3.8-intel
fftw/3.3.8-intel-impi
fftw/3.3.8-intel-impi-openmp
To load a specific module of FFTW, as recommended, a user must use a complete module name, like the following,

```bash
> module load fftw/3.3.8-intel-impi
```

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>-intel</td>
<td>compiled with the Intel compiler suite (default is gcc; no suffix)</td>
<td>gsl/2.5-intel</td>
</tr>
<tr>
<td>-mpi</td>
<td>with MPI support (Intel MPI)</td>
<td>mpibash/1.2-mpi</td>
</tr>
<tr>
<td>-openmpi</td>
<td>with MPI support (OpenMPI)</td>
<td>&lt;none available yet&gt;</td>
</tr>
<tr>
<td>-openmp</td>
<td>with threading support by OpenMP</td>
<td>fftw/3.3.8-intel-openmp</td>
</tr>
<tr>
<td>-i32 / -i64</td>
<td>32/64 bit integer/float support</td>
<td>metis/5.1.0-intel-i64-r64</td>
</tr>
<tr>
<td>-r32 / -r64</td>
<td>32/64 bit integer/float support</td>
<td></td>
</tr>
<tr>
<td>-X11</td>
<td>with X11 support</td>
<td>r/3.4.4-X11</td>
</tr>
<tr>
<td>-python</td>
<td>with python support</td>
<td>libxml2/2.9.8-python</td>
</tr>
</tbody>
</table>

Currently, most Spack generated packages are not supplied with a default version, so a `module load <package name>` will just give you the highest lexical version.

**Spack Branches and Releases**

- **release**: stable version for long term support
- **staging**: addon to current release or preview of a future release
- **master**: development version of LRZ - accessible for test installations, may be subject to full rebuild

LRZ will periodically release a new version of the software stack built with Spack and change the default version to the new release. Older versions will remain available by loading specific modules, e.g.

```bash
module switch spack/release/18.2
```

Sometimes we will provide new software prior to a new Spack software-stack release in a staging branch. To access such versions, load the staging module

```bash
module switch spack/staging/[<version>]
```

<table>
<thead>
<tr>
<th>Release number (click for details)</th>
<th>Release Date</th>
<th>Compiler Version</th>
<th>Platforms</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>GCC</td>
<td>Intel</td>
</tr>
<tr>
<td>18.2</td>
<td>10/2018</td>
<td>7.3.0</td>
<td>17.0.6</td>
</tr>
</tbody>
</table>
Please be aware that not all platforms will have the same set of packages and modules. In particular, commercial software may be restricted to certain machines.

Package Variables

When loading a module, package specific environment variables follow the conventions given in this section. In the following table, the placeholder PAC is used for the package name. These variables can be used in scripts and Makefiles e.g.

```bash
mpif90 ... $MKL_LIB
```

<table>
<thead>
<tr>
<th>Generic Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAC_BASE</td>
<td>base directory for package</td>
</tr>
<tr>
<td>PAC_LIBDIR</td>
<td>library directory for package, usually $PAC_BASE/lib</td>
</tr>
<tr>
<td>PAC_LIB</td>
<td>library setting for (static) linking, typically $PAC_BASE/lib/&lt;library_name.a&gt;</td>
</tr>
<tr>
<td>PAC_SHLIB</td>
<td>library setting for dynamic linking, typically -L$PAC_BASE/lib -l&lt;library_name&gt;</td>
</tr>
<tr>
<td>PAC_INC</td>
<td>include setting for compilation, usually -I$PAC_BASE/include</td>
</tr>
<tr>
<td>PAC_DOC</td>
<td>directory containing documentation (in other than man or info format)</td>
</tr>
<tr>
<td>PAC_WWW</td>
<td>URL referring to location of package-specific documentation on the LRZ WWW server.</td>
</tr>
</tbody>
</table>

Note that occasionally a module may contain subpackages e.g., SUB, in which case additional variables of the form PAC_SUB_LIB etc. may be defined.

Specifics on the setup for LRZ HPC systems

Module classes

In order to impose some structure on the large number of provided modules, a number of module classes has been defined, one of which each module falls into:

- **spack**: modules compiled with the spack building system
- **applications**: third party application programs will be entered here
- **compilers**: programs for converting computer language source code to (executable) machine code
- **environment**: modules for providing LRZ specific environment variables
- **graphics**: modules for graphics and visualization related functionality
- **io tools**: modules for I/O related libraries and tools
- **libraries**: performance and functionality libraries
- **parallel**: access to parallelization software, especially MPI
- **tools**: development, tuning and debugging aids

There is a further, special extfiles category which will be used e.g., for applications with a rapidly changing profile or for testing. However, this class is not made available by default. You will need to activate it by issuing

```bash
module use -a /lrz/sys/share/modules/extfiles
```

Please note that modules in this area are not guaranteed any support; also they may be modified or removed without any prior warning, on short order.

Shells and scripting languages

Supported Shells

The following shells are supported by the module system: bash, csh, ksh, sh, tcsh

Using modules from Perl or Python

It is possible to use the module command also from non-shell scripting languages. In the case of Perl, a LRZ specific package is available. Here is an example on how to use this:

```perl
#!/usr/bin/perl
use lib "/lrz/sys/share/modules/init";
use Environment_Modules;
module("load","intel mkl");
module("list");
system("make myprog");
system("./dosomething");
```

Note the following details:
• Use standard perl function call syntax with brackets, especially when specifying more than one argument
• In contrast to the shell setup, only the LRZ administrative modules are loaded. Please load all other required modules yourself.
• Within Perl, the environment variables are accessible via the %ENV hash.
• All environment variables will be propagated to processes called e.g., via system. Hence, you can use these variables in Makefiles and scripts which are executed/called from Perl.

Generating new shells

While the environment generated within a shell should be inherited to newly generated ones (either the same shell, or other shells), the module command itself may not be. We recommend inserting the following lines into your shell init script (e.g. .bashrc for bash, or $ENV for ksh):

```shell
otypeset -f module > /dev/null
if [ $? != 0 -a -r /etc/profile.d/modules.sh ] ; then
    source /etc/profile.d/modules.sh
fi
```

Note that this will preserve the environment set up in the calling shell.

Using modules in batch jobs

The modules package is loaded by default at login. However, the typical batch systems are not able to reliably load the modules package. Hence, if you want to use modules within batch scripts, please do not forget to insert the following command as the first line of your script:

```shell
source /etc/profile.d/modules.sh
```
or in case your script is csh derived.

```shell
source /etc/profile.d/modules.csh
```

Additional utilities

Some additional utilities which are provided specifically with the LRZ installation are described in this section.

1. `ismoduleloaded`: if issued with an argument, will tell whether that module is loaded. If the argument is a versioned module name, the exact version will be checked, otherwise only whether any version of the given module is loaded. Output is a “YES” or “NO” to stdout as well as an error exit of 0 or 1, respectively. If the -s command line switch is provided, output to stdout is suppressed. If no module argument is given, “YES” will be written to stdout if any module is loaded, “NO” otherwise.

2. `defaultversion`: if issued with a module name as argument (with or without a version number), this will return the default version of the module on STDOUT. Error 1 is returned if the module does not exist.

3. `loadedversion`: if issued with a module name as argument (with or without a version number), this will return the presently loaded version of the module on STDOUT. Error 1 is returned if a module of the given name is not loaded. The version number on the argument need not be the correct or even an existing one.

4. `removemodules.csh`: this can be used to completely remove all module settings from your environment by sourcing the appropriate version into your running shell; all modules are unloaded, and the `module` alias as well as all module-related variables are unset. Beware that this will rather cripple your environment since many important settings are performed via LRZ administrative modules.

Writing your own module files

A user can perform her/his own module initializations at login time using a shell-specific script in the `$HOME/.modules` subdirectory. This script must have the same name as your login shell, i.e., `$HOME/.modules/$SHELL`, and will typically contain

- extension of the `MODULEPATH` environment variable (e.g., via the `module use` command) to contain some subdirectory where a user's own module files can reside
- loading of additional LRZ or self-written modules at login as needed

Versioning policies

Please note that software packages as well as the module system undergo a maintenance and update process. As a consequence, the following may happen at LRZ HPC staff's discretion and - unless the user impact is considered very large - without special announcement:

- the default version of a software package may change; the method of loading the default module will however normally stay the same.
- a specific module version may become invisible. This means that it will not be listed any more by the `module avail` command, because the name or the version of the module begins with a dot. To see all versions of a module including the hidden ones, please use e.g., the bash command

```bash
ls -a $(dirname $(module path <package>))
```

We apologize for the potential impact on processing of user jobs, but given the large number of packages we make available we presently see no alternative. For a given software package, please check out its documentation page on the LRZ web server for updates and/or use the `module avail` command to see the presently available releases.

Support and sources of information on modules

If you experience problems using the module package, please notify the Support for HPC.
Further information on modules can be found by reading the man pages on the LRZ HPC systems (module (1), modulefile (4)). The command "module help" will also provide some information.

The sources can be found on the Sourceforge git repository. For the use of modules, a Tcl (http://www.tcl.tk/) installation is needed as a prerequisite.