OpenCMISS Build Environment Documentation

Specifications and Techdocs for building the OpenCMISS Modelling Suite with CMake.

This document specifies the components of the OpenCMISS Modelling Suite including Iron, Zinc, their respective dependencies and (build-)utilities. The GitHub repository structure can be found here, and this document explains the layout of the different components and the overall build process. The OpenCMISS main "logical" components are iron, zinc, examples, dependencies, utilities and documentation. Those components are managed by the setup project, which downloads (& manages) the sources, sets up build trees and according installation directories.

Building the OpenCMISS Suite

OpenCMISS user groups:

The build process is of different complexity, depending on whether you are simply using OpenCMISS components or develop some of them. Thus far there are two types of users:

Opencmiss-api-user:

A person who is looking to use OpenCMISS via the provided API. This person is not interested in software development of the OpenCMISS codebase. Will provide feedback on experience using the API and suggest enhancements in a non-formal setting e.g. mailing list.

For an opencmiss-api-user ideally there are precompiled binaries or packages (or a virtual machine image) from which to get started. In lieu of this a simple configuration and build process that has a high probability of satisfying their goals should be available.

Opencmiss-developer:

A person who is looking to develop the OpenCMISS codebase Iron, Zinc, dependencies, utilities the whole shebang.

For an opencmiss-developer a simple configuration and build process that will create local git repositories is available, also including registration of remote repositories from their own forks. Any component of the OpenCMISS Suite can be checked out in "developer mode", while the others are treated same as the api-user case.

Prerequisites

In order to build OpenCMISS or any part of it, you need:

1. A compiler toolchain (gnu/intel/ibm/...)
2. A local MPI implementation - some are shipped with OpenCMISS, if you want a different one, get it.
3. CMake 3.2.0 or higher. (http://www.cmake.org/download/)
   If you are on Linux/Mac and already have an older version installed (higher than 2.6), the build procedure will automatically download & install the required CMake version and prompt you to re-start the configuration with the new binary. On windows just download the current installer (http://www.cmake.org/files/v3.2/cmake-3.2.0-win32-x86.exe)
4. Linux/Mac only: OpenSSL. This is required to enable CMake to download files via the https protocol from GitHub. OpenSSL will automatically be detected & built into CMake if the setup script triggers the build, for own/a-priori cmake builds please use the `-DCMAKE_USE_OPENSSL=YES` flag at cmake build configuration time.
5. GIT version control (http://git-scm.com/downloads) **ONLY** if you intend to clone the setup script from GitHub or develop any components.

Building on Linux

Default steps for api-users (terminal/command line):

1. Create the OPENCMISS_ROOT folder somewhere and enter it
2. Clone the setup git repo into that folder via
   `git clone https://github.com/OpenCMISS-Utilities/manage.git`
   - or -
   Fetch and extract the zipped sources there and extract them into the `OPENCMISS_ROOT/manage` folder.
3. Enter the `OPENCMISS_ROOT/manage/build` folder
4. Type "cmake .."
5. Type "make | nmake | .. " whatever native build system you have around. Multithreading is used automatically, no "-j4" or so needed.
6. Have a coffee.

This will compile everything using the default compiler and default mpi. Basic warnings will be in place for all known erroneous system configurations.

Default steps for opencmiss-developers (terminal/command line):

The default steps are the same as for users with one intermediate step:
After step 2., copy the `OPENCMISS_ROOT/manage/Config/OpenCMISSDeveloper.template.cmake` file into the `OPENCMISS_ROOT/manage` directory and make any desired adjustments as described below. You can also plainly create the file as empty, but the template contains useful information. The build system will detect the file exists and include it at configure time.
Ideally, the first step for developers is to fork any components of OpenCMISS that should be

worked on at GitHub (or to some other git-aware location) and modify the developer template accordingly to have the build system checkout the repos from your own location.

Building on Windows (64bit) (experimental!)

I'm currently experimentig with MSYS2/MinGW-w64 to build stuff on windows.

1. Get MSYS2!
    a. Get installer from http://sourceforge.net/projects/msys2/
    b. Install (assume here: C:\MSYS2_64), dont use spaces in the installation folder!
    c. Follow the instructions in Section III to update your version http://sourceforge.net/p/msys2/wiki/MSYS2%20installation/
2. Get MinGW 64!
    a. Get installer from http://sourceforge.net/projects/mingw-w64/
    b. Choose you GCC version and threading model; the installer automatically suggests a suitable subfolder for your selection so you can have multiple versions in parallel.
    c. install, (assume here: C:\mingw-w64\...)
    d. Create a directory junction to include the mingw64-folder into the msys directory tree
        i. Open a windows command prompt IN ADMINISTRATOR MODE
        ii. Go into C:\MSYS2_64
        iii. Remove the old mingw64-folder (it should only contain an /etc folder)
        iv. Type
        mklink /J mingw64 C:\mingw-w64\<your selection>\mingw64
        v. Windows will confirm e.g.
        Junction created for mingw64 <<===>>
        C:\mingw-w64\x86_64-4.9.2-posix-seh-rt_v4-rev2\mingw64
        vi. If you want to switch to another toolchain version/model later, install mingw-w64 with that config and repeat the symlink steps.

3. Get an MPI implementation!
    a. http://www.mpich.org/downloads/ for MPICH2 (unofficial binary packages section, i used 64bit version http://www.mpich.org/static/tarballs/1.4.1p1/mpich2-1.4.1p1-win-x86-64.msi)
    b. https://msdn.microsoft.com/en-us/library/bb524831%28v=vs.85%29.aspx for MS MPI
4. Use the C:\MSYS2_64\`mingw64_shell.bat` to open an mingw64-pathed msys2 console/command (a.f.a.i.k. all that does is adding mingw64/bin to the path)
5. Install necessary packages: `pacman -S git make flex bison` (flex/bison for ptscotch builds)
6. Follow the build instructions for linux, with the only change of invoking `cmake -G "MSYS Makefiles" <args> ..`

Note:

- Get ssh keys if you want to make a development checkout of sources (i copied my existing id.pub etc into the ~/.ssh folder (absolute path C:\MSYS2_64\home\<windows-username>), otherwise find out how to create them and notify github, see https://help.github.com/articles/generating-ssh-keys/)
- MSYS comes with mingw32/64 packages (which must still be installed using packman, (i.e. `pacman -S mingw-w64-x86_64-gcc`), but i found that those packages don't come with gfortran (yet). thus, use the procedure above.
- Parmetis builds: get http://sourceforge.net/p/mingw-w64/code/HEAD/tree/experimental/getrusage/ to have resource.h header (followed source forge link) - or - comment out the line. does not seem to matter (for compilation :-))

Building on OS X 10.10

For building OpenCMISS-Iron on OS X install the following prerequisites:

- CMake >= version 3.2.0
- From CMake GUI install for command line use in the Tools menu
- XCode from the AppStore
- From XCode install the command line utilities
- Install Homebrew
- Using brew in install gfortran with openmp support using the --without-mutlilib flag

Build customization

OpenCMISS comes with a default build behaviour that is intended be sufficient for most cases of api-users. However, if you need to make (**well-informed!**) changes to the default build configuration, here is what we offer. For now, we'll use `<manage>` as shorthand to `OPENCMISS_ROOT/manage`.

Attention: Due to the way CMake is designed, one cannot change the compiler once the configuration phase is run (without a big fuss or deleting the directory contents). Moreover, changing the MPI implementation turned out to be very error-prone as well. Hence, we decided to include an intermediate layer to the build system: The top level CMakeLists.txt is only used to configure a template file with the given toolchain and mpi choices. The configured files are stored in subfolders `<manage>/build/compiler.<toolchain>-mpi.<mpi-type>`*. The "all" target of the top level simply invokes the CMake configuration and native build system on that subdirectory.*

Compiler choice

With CMake and a "normal" toolchain setup, one shouldn't have to change any compilers as CMake finds the default ones and uses them. However, if for some reason your default compiler setup is messed up or you need a specific compiler, there are two ways to change them:

1. Define the TOOLCHAIN variable on the command line via `-DTOOLCHAIN=[GNU|Intel|IBM]` or specify the corresponding quantities in your CMake GUI application. The values listed are currently supported and the build system has some included logic to locate and find the correct toolchain compilers.
2. Specify the desired compilers for any language explicitly using the `CMAKE_<lang>_COMPILER` variable

See the `<manage>/CMakeCScripts/ToolchainSetup.cmake` file for more background on option one.

MPI

MPI is a crucial dependency to OpenCMISS and is required by many components (well, it's the backbone to Iron's computational engine!). By default, CMake looks and detects the system's default MPI (if present) and configures the build system to use that.
If you need a different MPI version (and you should know WHY you need it!), there are several ways to achieve that:

1. Use the `MPI` variable and set it to one of the values `[mpich, mpich2, openmpi, intel, mvapich2, msmpi]`. CMake is "aware" of those implementations and tries to find according compiler wrappers at pre-guessed locations.
2. Set the `MPI_HOME` variable to the root folder of your MPI installation. CMake will then exclusively look there and try to figure the rest by itself.
3. Specify the compiler wrappers directly via `MPI_<LANG>_COMPILER`, which should ideally be an absolute path or at least the binary name. Possible values for `<LANG>` are `C`,`CXX` and `Fortran` (case sensitive!).
4. Set `OCM_SYSTEM_MPI` to `NO` and let the build system download and compile the MPI specified by `MPI`. *Note that this is only possible for selected implementations and environments that use GNU makefiles, as most MPI implementations are not "cmakeified" yet.*

At a later stage, the option `MPI=none` is planned to build a sequential version of opencmiss.

The `OpenCMISSLocalConfig.cmake` files

Everything else but the toolchain and MPI type can be configured in config files. The OpenCMISS default configuration is set by the `<manage>/Config/OpenCMISSDefaultConfig.cmake` file. Any value defined there can be overridden by re-definition in the local configuration files, which are the central point to change the build behaviour or add/remove components. The template file can be found at `<manage>/Config/OpenCMISSLocalConfig.template.cmake`. This template will be automatically processed and copied into the `<manage>/build/compiler.<toolchain>-mpi.<mpi-type>` folder (if not already there), where it will be read and processed by the main CMake script. *Yes, correctly, you* *have a possibly different local configuration file for each toolchain/mpi combination!*

Build precisions

The flags “sdcz” are available to be set in the `BUILD_PRECISIONS` variable. It is initialized as cache variable wherever suitable and can be passed via command line in the opencmiss-buildenv.
Note: Currently LAPACK/BLAS is always built using `dz` minimum, as suitesparse has test code that only compiles against both versions (very integrated code). `SCALAPACK` is always built with `s` minimum.

OpenMP

Have global flag `OCM_USE_MT`, that controls if “local” multithreading should be enabled/used. Thus far only OpenMP is implemented in the build system, so this controls the `WITH_OPENMP` flag being passed to any dependencies that can make use of it. If used, the architecture path will also contain an extra segment “mt” between mpi and compiler.

Single component configuration

A central concept of the build system is a *component.* A list of all components known to the setup process can be found in `<manage>/Config/Variables.cmake#OPENCMISS_COMPONENTS`. We will abbreviate a placeholder for a component by `<COMPNAME>` in the following.

- To enable/disable the use of a component in OpenCMISS, use the `OCM_USE_<COMPNAME>` variable. This will, however, only disable the build of the component and does **not** check for violation of interdependencies.
- To specify a certain version of a component, use the `<COMPNAME>_VERSION` variable. See the default configuration file for a set of interoperable versions of all components.
- Component interconnections are realized via variables like `SUPERLU_DIST_WITH_PARMETIS`. For a list of all possible component connections see the `<manage>/Config/OpenCMISSDefaultConfig.cmake` file.Those default settings can be overwritten by re-definition in the local config file.

Testing

For testing, the variable `BUILD_TESTS` can be set and is turned on by default for each dependency.

Local packages inclusion policy

Many libraries are also available via the default operating system package managers or downloadable as precompiled binaries. OpenCMISS allows to use those packages, however, the default policy is to download & build every required/selected package from our repositories as they are known to be compatible with any other OpenCMISS component at any published version of the setup script. To allow the local search for a component, set the `OCM_SYSTEM_<COMPNAME>` flag to `YES` in the local config file. Note that the search scripts for local packages (CMake: find_package command and Find<COMPNAME>.cmake scripts) are partially very immature and unreliable; cmake is improving them continuously and we also try to keep our own written ones up-to-date and working on many platforms. This is another reason why the default policy is to rather build our own packages than tediously looking for binaries that might not even have the right version and/or components.

Package version management

OpenCMISS uses ‘git’ and version-number named branches to maintain consistency and interoperability throughout all components. Each dependency as well as iron and zinc has branches like “v3.5.0”, and the `OpenCMISSDefaultConfig.cmake` file contains the respective version numbers “3.5.0” [that will/can also be used to look for local versions]. Those quantities are not intended to be changed by api-users, but might be subject to changes for development tests. Assuming the existence of the respective branches on GitHub, all that needs to be done to change a package version is to set the version number accordingly. The setup will then checkout the specified version and attempt to build and link with it. Warning: Having a consistent set of interoperable packages (especially dependencies) is a nontrivial task considering the amount of components, so be aware that changes will most likely break the builds!

The `OpenCMISSDevelopers.cmake` configuration file

As OpenCMISS-Developers will mainly work only on a selection of components, the configuration file is intended to tell the build system which those components are and have the setup checkout the correct git repos instead of downloading plain sources.
At first, a flag `<COMPNAME>_DEVEL` must be set in order to notify the setup that this component (iron, zinc, all dependencies etc) should be under development.
As we recommend OpenCMISS development via GitHub forks, we recommend to set the variable `GITHUB_USERNAME`. if so, the setup will automatically compute the repositories location (assuming you wont change the forked repos names) and that’s it. if you have an SSL Key registered on GitHub for your local machine, set `GITHUB_USE_SSL` to YES to have the setup clone via SSL instead of HTTPS.

Alternatively, for every component there is a pair of variables `<COMPNAME>_REPO` and `<COMPNAME>_BRANCH` which can be set to any value. The setup will then clone the repositories from there and switch to the specified branch. If no `<COMPNAME>_REPO` or `GITHUB_USERNAME` is given, the setup chooses the default public locations at the respective GitHub organizations (OpenCMISS, OpenCMISS-Dependencies etc). If no `<COMPNAME>_BRANCH` is given, the setup automatically derives the branch name from the `<COMPNAME>_VERSION (pattern “v<COMPNAME>_VERSION”).`

Techdocs for build system developers

File System Layout

The top level folder subsequently called `OPENCMISS_ROOT` is the base folder for the build environment. Its subfolders are as follows:

- `build/`
  Global root for all build trees. Every build tree is optionally (default:no) further organized using an architecture path `arch-dir` that separates binaries and libraries build with different mpi/compiler versions etc; see below for its specifications.
  - `[arch-dir-short/]mpi/`
    Contains the builds of different mpi implementations (if no system ones are used)
    - `openmpi`
    - `mpich`
  - `[arch-dir/]`
    Contains the actual components of OpenCMISS, sorted into subfolders according to the logical grouping. The last level of each component is the build configuration, to keep consistent with multi-configuration generators like xcode or visual studio
    - `dependencies`
      - `blas[/release|/debug/|...]`
      - …
      - `petsc[/release/|/debug|...]`
    - `iron[/release/|/debug|...]`
    - `zinc[/release/|/debug|...]`
    - `examples[/release/|/debug|...]`
  - `utilities/`
    - `gtest`
- `manage/`
  Contains the main setup project and scripts that organize source downloads and builds. For details see the developer tech notes here. The only relevant folder for api-users is
  - `build`
    CMake main invocation folder, sets toolchain and MPI choice and generates an appropriate subfolder with those choices fixed.
    - `compiler.<toolchain>-mpi.<mpi-type>`
- `src/`
  Contains all component and utilities sources, again sorted into subfolders by logical groups.

    - `iron/`
    - `zinc/`
    - `dependencies/`
      this folder “collects” all dependencies sources into one folder for tidyness
        - `blas/`
        - `...`
        - `zlib/`
    - `utilities/`
        - `cmake/`
        - `git/`
        - `openmpi/`
    - `examples/`
        - `ex1/`
        - `ex2/`
- `install/`
  In order to be able to discard of any build/source folders when necessary, all binaries, includes (and config files) are installed under a global install directory, subjected to the architecture path used for the current build. In order to have consistent behaviour for linux and windows, the build type (release/debug) is the last path component.
    - `[/arch-dir][/release|/debug|...]`
        - `bin`
            - `cmgui-exe`
            - …
        - `lib`
            - `iron.mod`
            - `zinc.a`
            - `blas.a`
            - `zlib.a`
        - `include`
        - `cmake`
          Contains the cmake package config files. If the libraries are given relative to the install prefix (which is a good thing), unfortunately we cant have the config.cmake files *outside* the install_prefix. that would be suitable as the naming convention for package files is to append -release or -debug automatically, thus we’d ideally have one folder “cmake” on the parent level along “debug|release”. this is, however, not implemented in current cmake versions.
    - `utilities`
        - `gtest`
        - `cmake`

The architecture directory organziation

todo or link to external page

The main setup project organization

The setup project is the main access point for OpenCMISS builds and shows the following structure (mounted on `OPENCMISS_ROOT/manage/`)

- `build/`
  This is the main build-tree entry point where CMake is invoked. From here CMake will organize which external projects are build under `OPENCMISS_ROOT/build/`. this folder is arbitrary in principle and defaults to "build".
    - `compiler.default-mpi.mpich`
    - `compiler.default-mpi.openmpi`
      Subfolders for fixed toolchain and MPI choices. Contains the processed `CMakeLists.main.template.cmake` file as main CMakeLists.txt.
- `CMakeLists.txt`
- `Config/`
  A collection of CMake files regarding the configuration of the build process
- `CMakeScripts/`
  A collection of scripts performing a specific task. Merely created for tidyness and separation of concerns.
    - `CMakeCheck.cmake`
    - `OCMSetupBuildMacros.cmake`
    - `CMakeLists.main.template.cmake`
    - ...
- `CMakeFindModuleWrappers/`
  Own wrappers for find_package() calls in OpenCMISS
    - FindXXX.cmake
- `CMakeModules/`
  Own provided MODULE mode search scripts
    - FindSUNDIALS.cmake
    - ...

---- Discussion part!!! -----

Buildsystem/Layout alternative

What we are currently doing is mixing up the "src/build/install" structure with “iron/zinc/dependencies/examples” (and possibly an architecture path as well). one is a software-type constraint and the other is a logical grouping in “our heads”. currently we let the logical grouping go first and have separate iron/zinc/deps folders, each containing (even only partially!) of src/build/install. here are some pro arguments for the structure below:

- looking from the opencmiss as a whole piece of software, i’d expect src/build/install first.
- the architecture path can be inserted at high level instead of repeating it for deps/iron/zinc etc (the builds with same architecture path are linked exclusively against each other anyways, so why not have iron.mod amongst petsc.a and others).
- install in one folder: will require only one folder to link against for examples/applications using opencmiss.
- not everything will be installed at all times, it makes more sense to have varying subfolders.
- users “unaware” of the guts of opencmiss don’t really care if there are dependencies or not, they “expect” a source folder to build from and an install location where all “the stuff” goes that involves opencmiss as a software.
- programmers still find the logical grouping within the src/build/ folders to kick off single special builds
- no manage or config-folders needed
- no submodules needed anywhere: the default config contains versions of all packages (iron/zinc/dependencies), which will be used to check out the respective *branch* with the same name (+prefix maybe). tags are very unhandy to use as they need to be removed/re-set after each commit so that the setup uses the correct source code

[Variables specified in a section get the uppercase section name plus an underscore prepended onto it. The exception is the general section where the general name gets changed to OPENCMISS.]

Command line versions of a variable do not have the namespace prefix.

General Variables

OPENCMISS_ROOT (Command line version ROOT)	The root location of the source, dependencies and utilities
OPENCMISS_BUILD_TYPE	Release
OPENCMISS_INSTALL_PREFIX	OPENCMISS_ROOT/install

Zinc Variables

Iron Variables

Development notes Daniel

Download modes

Related config variable: OCM_DEVELOPER_MODE [YES|NO]

NO = User

The user download tries to simply load the current registered submodule revisions as zip files from GitHub and thats it. Then the build script uses the sources as-is and done.

Unfortunate: if CMake has not been built with SSL support (it has its own libcurl), the https:// urls from GitHub fail. Hence, the fallback is to use GIT’s native approach to make a ‘--depth=1’ copy only and thus have a minimal download. This will work 100% as the dependencies repo must have been obtained using git and a secure connection. The resulting .git folder could be removed for convenience of less disk space usage.

Alternative: Set up a proxy here at ABI that serves standard http requests and internally asks for the https GitHub zip file. this would not only make the download idiot-proof, but one could also check how often and where the files are requested ..

Comment: git clone with --remote does not work for GitHub :-(

YES = Developer

The developer mode checks out the complete submodules and checks out the “opencmiss”-branches each. Only the submodules corresponding to currently selected packages are tracked and loaded if need be.