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 (planned) 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 OpenCMISS manage project, which downloads (& manages) the sources, sets up build trees and according installation directories.

Building the OpenCMISS Suite

ATTENTION! If you encounter any troubles, don't miss BUILD SUPPORT section!
The base for the installation is a folder called OPENCMISS_ROOT. We'll use `<manage>` as shorthand to `<OPENCMISS_ROOT>/manage`.

User groups

Shortly, we have two major groups of people using OpenCMISS: Users and Developers. Users (surprise!) only use the OpenCMISS components Iron/Zinc but may of course create/develop their own examples. Developers are people that intend to make changes to the OpenCMISS codebase itself, i.e. add functionality to Iron, Zinc or any other component. Consequently, some parts of this documentation apply only to certain user groups.

Prerequisites

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

1. A compiler **toolchain** (gnu/intel/clang/...)
2. A **MPI** implementation - some are shipped with OpenCMISS and can be automatically built for you. If you want a different one, no one holds you back.
3. [Optional] **Git** version control (http://git-scm.com/downloads). This is recommended as cloning the repositories makes contributing back easier from the start!
4. **CMake 3.3.1** 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 provide a target "cmake" to build 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/v3.3/cmake-3.3.1-win32-x86.exe)
   a. Only for Linux/Mac **without** Git: 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.

Building on Linux

Default steps for 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/manage`
    - 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. [optional] Make changes to the configuration through the OpenCMISSLocalConfig.cmake file in the environment specific build directory.
6. Build the "opencmiss" target via "make | nmake | .. opencmiss" (or whatever native build system you have around). Multithreading is used automatically, no "-j4" or so needed.
7. Have a coffee.
8. Coming back from the coffee and something failed? Checkout the support section.

This will compile everything using the default compiler and default mpi. Basic warnings will be in place for all known erroneous system configurations. *The OpenCMISS-Examples are a competely different package/project and if you want to build them after you've finished building the OpenCMISS libraries please see here.*

Default steps for Developers (terminal/command line):

The default steps are the same as for users, but with two changes:

1. At step 4, invoke `cmake`
   `-DEVIL=<your_freely_chooseable_evilness_value> ..`
2. In addition to the changes you can make at step 5, change the file
   `OPENCMISS_ROOT/manage/OpenCMISSDeveloper.cmake` according to your
   OpenCMISS development needs. A detailed description of options can be found [below](#).

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 config script accordingly to have the build system checkout the repos from your own location. You can still change repository locations later, however that might require a complete re-build.

Building on Windows (64bit) (experimental!)

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

1. Get CMake >= 3.3.1! An issue has been fixed upon request that messed up the FortranInterface verification. This is included as of 3.3.1.
2. 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
3. Get MinGW 64!
   a. Get installer from http://sourceforge.net/projects/mingw-w64/
   b. Choose you GCC version and threading model (use posix); 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.

4. 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

c. Install to a location WITHOUT spaces!

5. 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)

6. Install necessary packages: `pacman -S git make flex bison` (flex/bison for ptscotch builds)

7. Follow the build instructions for linux, with the only change of invoking `cmake -G "MSYS Makefiles" <args> ..`

a. Most likely you will need to specify MPI_HOME when running the main build configuration.

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 Windows (32bit) (experimental!)

Its basically the same as for 64 bit, but obviously using the msys2-32bit and mingw32-packages. Attention: The most current version of mingw32 comes with a pthread package, but unfortunately there is a severe error (or here) on GNULib's side: the struct "timespec" is also defined for mingw32 versions and conflicts whenever "unistd.h" is also included. Either apply the patch or simply uncomment the struct definition in `mingw32-root\include\pthread.h:320`.

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 install gfortran with openmp support using the --without-mutlilib flag

Available build targets

Just building "all" is not enough for OpenCMISS, as the install step is important to create the information about the OpenCMISS build that is needed by any examples or applications. Therefore, the build system's main target is called "opencmiss" and should be invoked for any build.

Moreover, there are some convenience targets at your disposal:

- update
  Goes through all OpenCMISS components that are locally build and fetches the newest commit on the configured version branches.
- support
  See the support section.
- gitstatus
  This target is intended for developers, who would like a quick way of obtaining the current status of all components that are build locally. Only available if Git is found.
- test
  Run all the tests for all current components. Lengthy!
- examples
  Convenience target to download & build all the examples registered as submodule of the OpenCMISS-Examples/examples repository.
- examples-test
  Uses CTest to simply execute all the examples (if successfully built). Currently they're invoked without arguments which may break some of them due to that.
- reset
  Removes everything from the current build root but the OpenCMISSLocalConfig file. Also invokes the following (independently usable) targets:
    - reset_featuretests
      Triggers a re-build of the feature tests
    - reset_mpionly
      Blows away all the build and install data of components with MPI capabilities.
- utter_destruction
  Removes the complete `build/` and `install/` root directories created by any architecture build.

Component-level build targets

Besides the top-level targets, each OpenCMISS component also provides targets for direct invocation. In the following, <compname> stands for any OpenCMISS component (lowercase).

- <compname>
  Trigger the build for the specified component, e.g. "make iron"
- <compname>-clean

Invoke the classical "clean" target for the specified component and triggers a re-run of CMake for that component.
- **<compname>-update**
  Update the sources for the specified component. *Please note that you should ALWAYS use the top level "make update" command to ensure fetching a compatible set of components - single component updates are for experienced users only.*
- **<compname>-gitstatus**
  Get a current git status report. Only available if Git is used.
- **<compname>-test**
  Run any tests provided by the component.

Building for multiple architectures/configurations

If you want to compile OpenCMISS using different compilers, MPI implementations etc., the build environment is aware of most common "architecture" choices and automatically places builds in appropriate architecture paths. Most importantly:
- All architecture choices except toolchain and MPI configuration can be made inside the OpenCMISSLocalConfig file explained later. Toolchain and MPI setup are explained below. See also the main `CMakeLists.txt` file.
- You need a different base binary directory for each intended toolchain/mpi combination! The build instructions above are laid out by default for only one (=the default) toolchain and mpi choice. The name and place of the different binary directories are up to you, we recommend to put them inside the `<OPENCMISS_ROOT>/manage` directory for simple "cmake .."-style invocations. If you decide to put them somewhere else (e.g. `<OPENCMISS_ROOT>/build/my_toolchain_mpi_combo`), you will need to invoke cmake at that location and pass an absolute or relative path to `<OPENCMISS_ROOT>/manage` as argument.

Example directory layout and cmake invocation:
<OPENCMISS_ROOT>/manage/
     build/
     >> `cmake -DTOOLCHAIN=GNU -DMPI=mpich ..`
     build_intel/
     >> `cmake -DTOOLCHAIN=intel -DMPI=intel ..`

Note: As the sources for one OpenCMISS installation are the same for each different architecture build, you cannot have two different source versions (e.g. branches) for any component for two or more different architectures. For example, you cannot have the current release version of IRON within a build using Intel MPI and the devel version of IRON with OpenMPI.
To solve this, you need two different OpenCMISS root installations.

Toolchain/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` variables.

See the `<manage>/CMakeScripts/OCToolchainCompilers.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 want a specific MPI version, 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]`, e.g. "cmake -DMPI=mpich". The build system is aware of those implementations and tries to find according compiler wrappers at pre-guessed locations.
   a. If you want the build environment to build the specified MPI for you, set `SYSTEM_MPI` to `NO` and let the build system download and compile the specified implementation. *Note that this is only possible for selected implementations and environments that use GNU makefiles, as most MPI implementations are not "cmakeified" yet.*
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!).

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

Building OpenCMISS examples (Unix/terminal)

The instructions here are for building a single example. For developers/testers, see the instructions for building all examples. Building an example requires some effort in order to have compatible settings to your OpenCMISS installation. Luckily, the OpenCMISS build system can manage that for you. But this, in turn, means that a local OpenCMISS installation (=manage

repository) is always required, even if you want to use a remote installation and only build examples yourself. So, make sure you have that first, as it will look for (and possibly build) at least a local MPI implementation.

1. Download the desired example from PMR or GitHub. (In the current transition phase, clone the https://github.com/OpenCMISS-Examples/examples repository and navigate to the folder of your desired example
2. Create a build folder within the example source, e.g. `build.` and change to it
3. Invoke `cmake -DOPENCMISS_INSTALL_DIR=<OPENCMISS_ROOT>/install ..`
4. Invoke `make install`

Now you should have a binary "run" in your example source that can be executed.

Build support

A word! Having a smoothly working build is what we aim to provide for you. Yet, facing all those different systems, there are still situations where stuff goes woo. In order to help you out as good and fast as possible we implemented a build report system for support. The report system is realized via a "support" target that can be built (e.g. "`make support`"). This will collect build information and create a zip file, which you can attach to any support email. Nice, heh?

OpenCMISS remote installations

If an entire work group uses OpenCMISS, it is desireable to have a pre-compiled set of OpenCMISS libraries and dependencies at a central location on the network. While this can save a lot of disk space and reduce maintenance efforts, good care needs to be taken to find and use a matching set of libraries depending on your local architecture, compiler and MPI versions. The OpenCMISS build system tries to find those matches automatically by using an architecture path.

User instructions (="Client-side")

Specify the `OPENCMISS_REMOTE_INSTALL_DIR` in the OpenCMISSLocalConfig file. The build system will then automatically search for a matching OpenCMISS installation at that remote directory.
The above procedure is recommended as it will use an architecture path to find compatible installations. If that fails for some reason and you need to override that mechanism, specify `OPENCMISS_REMOTE_INSTALL_DIR_FORCE` instead and have it point to the mpi-dependent architecture sub-path of the remote installation which contains the "`context.cmake`" file.

Developer instructions (="Server-side")

To set up a remote OpenCMISS installation, at first setup the OpenCMISSDeveloper file and enable the (in this case mandatory) `OC_USE_ARCHITECTURE_PATH` setting. Please also make

sure to fill in your eMail address into `OC_INSTALL_SUPPORT_EMAIL`. Next, build and install all the different configurations that you want to provide for the consuming clients. Note the [build customization](#) section below for instructions to create builds for different compilers and MPI versions. Finally, publish the installation root directory `OPENCMISS_ROOT/install` (check for different mount paths!) to anyone wanting to use the remote installation.

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.

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 exclude those settings from those you can "easily" change. See [here](#) for information on toolchain/mpi configuration.

Common Options

All the OpenCMISS options and their defaults are listed [here](#).

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 your current build binary directory (if not already existing - default: `<manage>/build`), 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 (and not for every component), 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 `OC_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 `OC_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")`.

FAQ

Why do you have different folder structures for debug/release builds?

By design, users/developers should be able to build a debug version of their example or even iron while having optimized MPI and dependencies. In order to ensure to have CMake find the correct builds, separate directories are employed.
Moreover, while the new cmake package config file system allows to store library information for multiple configurations, different include directories are not yet natively supported. As some packages provide fortran module files (which are different for debug/release), they need to be

stored at different paths using different include directories.

How do i use precompiled dependencies like in the old build system?
See the [remote installations](#) section.

Troubleshooting

This section is intended to be used as first place of collection for common errors and mishaps

Im getting errors from CMake claiming "You have changed variables that require your cache to be deleted [...] The following variables have changed: CMAKE_C_COMPILER=/usr/bin/gcc (or similar)

One case where this occurs is if you try to configure an OpenCMISS build using the same compiler but different aliases/symlinks like "gcc" or "cc", where "cc" just is the system's default compiler pointing to e.g. "gcc". Assume your first configure run was just `cmake ..`, and later you've decided to re-configure using `cmake -DTOOLCHAIN=GNU ...` Now, if selecting GNU as toolchain effectively selects the same compiler, but the binary path is different, CMake (correctly) assumes you've got a new compiler and needs to delete the cache. The trouble comes in as the build environment (correctly) detects that your default compiler in fact was from the GNU toolchain and hence placed the builds for the external projects under the same architecture path - boom.
Solution: Run `make reset` in the current architecture main build directory and re-build using `make`.

MPI detection mismatch. Aborting.
This occurs when the automatic MPI detection/selection system fails. A known case is e.g. if you have GNU/Intel toolchains with mpich/intel mpi installed, and there is a `mpigcc` wrapper within the intel MPI path, but the mpich gcc wrapper is simply called `mpicc`.
Solution: Instead of choosing the `MPI` value, specify the `MPI_HOME` directly.

Could not find a package configuration file provided by "OpenCMISS" with any of the following names / opencmiss-config.cmake is missing

Two scenarios commonly happen here:
1. For remote installations: You specified an incorrect remote directory - see [here](#).
2. The "install" target was not run. This most commonly occurs if you did not build the "opencmiss" target.
   Solution: Run the "install target, which installs the `opencmiss-config.cmake` files. Then, your example / application can use that to find your opencmiss installation. As the "install" target is a part of the "opencmiss" target, we recommend building "opencmiss"

from the start as described [here](#).

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`
    Default build tree folder for top level project.
  - build_intel
    [more build folders for different toolchain/mpi configs - suggestion only]
  - build_intel_mpi_debug
  - build_my_naming
- `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 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/`)

- `CMakeLists.txt`
  Main CMake build config file.
- `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`
  - `OCSetupBuildMacros.cmake`
  - ...
- `CMakeFindModuleWrappers/`
  Own wrappers for find_package() calls in OpenCMISS
  - FindXXX.cmake
- `CMakeModules/`
  Own provided MODULE mode search scripts
  - FindSUNDIALS.cmake
  - …
- Templates/
  Files that are configured at some stage of the configuration or build phase
  - `OpenCMISSLocalConfig.template.cmake`
  - `...`

Examples structure

Similar to the old build system, the examples available for OpenCMISS are kept separate. Ultimately, all the available examples will be hosted in their own GIT repository, and a central examples repository will collect all working examples for any OpenCMISS release. In the process of conversion, however, there still only exists one global examples repository (initialized with the old examples svn repo) at https://github.com/OpenCMISS-Examples/examples, branch "v1.0".

The current global project can generate the CMakeLists.txt files automatically (not very clever though) for each example. The detection is done simply via looking if the according folder contains a "Makefile" file. This is far from ideal, but a quick way to see what's working and what not.