CMake のつかいかた

- １ー CakeへのIntroduction
- ２ー ILCソフトで CMakeを使う
- ３ー CMakeを使ってILCをインストールする

Jan　Engels

ジャン　エンゲルス
DESYにて
20th September 2007

CMakeとはなにか

- CMake
  - 本来のビルド環境を作る

● UNIX/Linux -> Makefiles
● Windows -> VS Projects/Workspaces
● Apple -> Xcode

にそれぞれ対応している

- オープンソースであること
- クロスプラットフォームであること

CMakeの特長

・Cmake には沢山のいいところがある：

ー複雑で大きなビルド環境を管理出来る

ー非常に柔軟性があり且つ拡張性ガ高い

- ・マクロをサポートしている
- ・ソフトウエア（既に利用可能なモジュール の集まり）を見つけたり構成するモジュールがある
- ・新たなコンピューターや言語に拡張出来る
- ・新たなカスタムコマンドやターゲットを作る ことが出来る
- ・外部のプログラムを実行できる

ー単純で、わかりやすい表記方法である

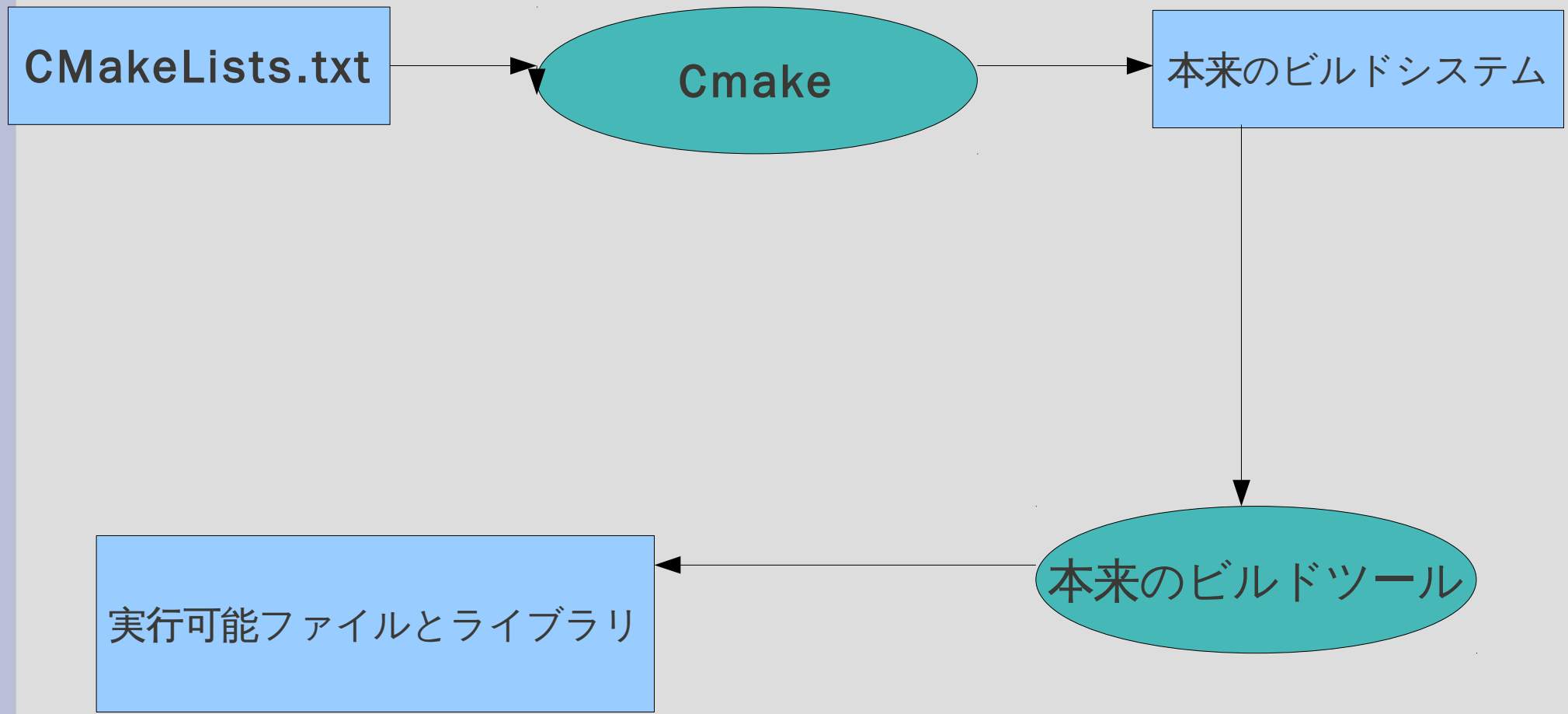
ー正規表現を使える（*nix style)

ビルドの時“In-Source”と “ Out-of-Source”を両方使える

ークロスコンパイルが出来る

ー総合的なテストとパッケージングが出来る（test,CPack)

ビルドシステムの生成


CMakeLists.txt
Cmake
本来のビルドシステム
本来のビルドツール
実行可能ファイルとライブラリ

CMakeの基本的考え方

・CmakeLists.txt
- 入力用のテキストファイルで、プロジェクトパラメーターとかビルドプロセスのフロー制御を記述しており、単純にはCMake言語である

・Cmake モジュール
- 特別なcmakeファイルであるソフトを探したり、そのライブラリーやインクルードファイルやその適切な変数の定義などを設定し、結果として、その他のプロジェクトのビルド処理で使われる様になる物である

(例えば、FindJava.cmake, FindZLIB.cmakeやFindQt4.cmake などである)

CMakeの基本的考え方

・ソースコードツリーは次の物を含む
  - CMake の入力ファイル（CmakeLists.txt)
  - プログラムソースファイル（例えば　hello.cc)
  - プログラムヘッダーファイル（例えば　hello.h)

・バイナリーツリーは次の物を含む
  - 本来のビルドシステムファイル（Makefiles)
  - ビルド処理の出力
      ・ライブラリー（複数）
      ・実行ファイル
      ・その他ビルドが生成するファイル
・ソースツリーとバイナリツリーは次のとおり
      - 同じディレクトリーの中（in-source build)
      - 異なるディレクトリーの中（out-of -source build)

Cmake の基本的考え方

- CMAKE_MODULE_PATH（CMAKE モジュールのパス）
  –Cmakeモジュールのある場所へのパス

- CMAKE_INSTALL_PREFIX（CMAKEのインストール先のプレフィックス）
  –make installが呼ばれたときにファイルをおく場所（インストール先）

- CMAKE_BUILD_TYPE　（cmakeの種類)
  -ビルドの種類（Debug,Reliase,・・・・）デバッグ、リリース

- BUILD_SHARED_LIBS
  -共有ライブラリーとスタティックライブラリの間ノ切り替え

- 変数は（(CmakeLists.txt)を直接変更して行える。すなわち '-D':で始まるコマンド行を変更して行える。
  –cmake -DBUILD_SHARED_LIBS=OFF

- GUI also available: ccmake　（ccmake を使えばGUIを利用できる）

CMake Cache (キャッシュの使いかた)

- ビルドツリーの中に作成される (CMakeCache.txt)
- エントリーを含む VAR:TYPE=VALUE
- コンフィグレーション中にPopulated/Updatedを行う
- ビルド処理を高速化する
- cmakeを用いてイニシャライズされる
- 次の様に行う　　cmake -C <file>
- GUI は変数値を変更するために使用される
- これらは手動で編集必要がない!!

ソースの構造

- SUBDIRS/ADD_SUBDIRECTORYの様にサブディレクトリーを追加出来る

- 親ディレクトリーカラ子ディレクトリへの継承がある（コノ形質は伝統的なMakefilesでは欠けていたものである）

- 処理ノ順番はDir1;Dir3;Dir4;Dir2 である（Cmakeがサブディレクトリーのコマンドを見つけると現在のファイル処理を停止してツリーを降って処理する）

CMakeを使う

- ビルドディレクトリーを作る (“out-of-source-build” concept)
  - mkdir build ; cd build
- 各自のシステムに合わせたパッケージを構成する
  - cmake [options] <source_tree>
- パッケージを作る
  - make
- Install it:
  - make install
- 最後の2行は一つにまとめることが出来る (just “make install”)

Similar to Auto Tools
自動ツールと同様

Hello World プログラムを CMakeで作る

- プロジェクトのトップディレクトリー
  - CMakeLists.txt
  - Sub-directory Hello:
    - CMakeLists.txt
    - hello.h
    - hello.cc
  - Sub-directory Test:
    - CmakeLists.txt
    - test.cc

```
/*hello.h*/
#ifndef _hello_h
#define _hello_h

class#endif
 Hello {
public:
void Print();
};
#endif
```

```
/*hello.cc*/
#include "hello.h"
#include <iostream>
using namespace std;
void Hello::Print() {
cout<<"Hello, World!”<<endl;
 }
```

Library Hello

```
/*test.cc*/
#include <iostream>
#include "hello.h"

int main() {
Hello().Print();
return 0;
}
```

Test Binary

Hello World プログラムを CMakeで作る

```
# Top-Level CmakeLists.txt
PROJECT( HELLO )
ADD_SUBDIRECTORY( Hello )
ADD_SUBDIRECTORY( Test )
```

CmakeList.txtはトップレベルにある

CmakeLists.txt はhelloデレクトリーにある

Hello(linux下のlibHello.a)と呼ばれるライブラリーにソースファイルhello.ccから追加する

```
# CmakeLists.txt in Hello dir
# Adds a library called Hello (libHello.a under Linux) from the source file hello.cc
ADD_LIBRARY( Hello hello )
```

```
# makeLists.txtはTestディレクトリにある
# コンパイラーは確かに自分のHello ライブラリからインクルードファイルを見つける異が出来た
INCLUDE_DIRECTORIES(${HELLO_SOURCE_DIR}/Hello)
#  ”test.cc”と言うソースファイルからビルドされた”helloWorld”と呼ばれるバイナリを追加する
# エクステンションは自動的に発見される
ADD_EXECUTABLE(helloWorld test)
# hello ライブラリの実行可能ファイルにリンクを張る
TARGET_LINK_LIBRARIES(helloWorld Hello)
```

CmakeLists.txt ファイル

- Very simple syntax: 非常に単純な表記法
  - # This is a comment これはコメント行です
  - コマンド表記法: COMMAND(arg1 arg2 ...)
  - Lists A;B;C # セミコロンは値を分離するものである。
  - Variables ${VAR} 変数の表記
  - Conditional constructs 条件の構築
    - IF() ... ELSE()/ELSEIF() ... ENDIF() IF文が使える
      - 非常に便利 I: IF(APPLE); IF(UNIX); IF(WIN32) の表記が可能
    - WHILE() ... ENDWHILE() While文が使える
    - FOREACH() ... ENDFOREACH() FOR文も使える
  - 通常の表記について、詳しくはCMakeのFAQを参照
    (check CMake FAQ for details...)

CmakeLists.txt ファイル

- INCLUDE_DIRECTORIES(“dir1” “dir2” ...) 含まれるディレクトリー
- AUX_SOURCE_DIRECTORY(“source”) ソースディレクトリーのアクセサリ
- ADD_EXECUTABLE 実行ファイルの追加
- ADD_LIBRARY ライブラリの追加
- ADD_CUSTOM_TARGET カスタムターゲットの追加
- ADD_DEPENDENCIES(target1 t2 t3) target1 depends on t2 and t3
  依存性の追加
- ADD_DEFINITIONS(“-Wall -ansi -pedantic”) 定義の追加
- TARGET_LINK_LIBRARIES(target-name lib1 lib2 ...)
  各ターゲットの個別設定
- LINK_LIBRARIES(lib1 lib2 ...)ライブラリーの同じセットに関する全ターゲットへのリンク
- SET_TARGET_PROPERTIES(...) 多くのプロパティ... OUTPUT_NAME, VERSION, ...
- MESSAGE(STATUS|FATAL_ERROR “message”)
- INSTALL(FILES “f1” “f2” “f3” DESTINATION .)
  - ${CMAKE_INSTALL_PREFIX}に関連したDESTINATION

Check www.cmake.org -> Documentation

CmakeLists.txt ファイル

- SET(VAR value [CACHE TYPE DOCSTRING [FORCE]])
- LIST(APPEND|INSERT|LENGTH|GET|REMOVE_ITEM|REMOVE_AT|SORT ...)
- STRING(TOUPPER|TOLOWER|LENGTH|SUBSTRING|REPLACE|REGEX ...)
- SEPARATE_ARGUMENTS(VAR) convert space separated string to list
- FILE(WRITE|READ|APPEND|GLOB|GLOB_RECURSE|REMOVE|MAKE_DIRECTORY ...)
- FIND_FILE
- FIND_LIBRARY
- FIND_PROGRAM
- FIND_PACKAGE
- EXEC_PROGRAM(bin [work_dir] ARGS <..> [OUTPUT_VARIABLE var] [RETURN_VALUE var])
- OPTION(OPTION_VAR “description string” [initial value])

Check www.cmake.org -> Documentation
ウエッブサイトの文書をチェック

CMake手引書

- 2 – Using CMake for the ILC Software

CMake手引書

- IMPORTANT:
  - ILSソフトノCmake files はr out-of-source buildsに完全に排除する感じで設計され、作成され、テストされている、その訳は in-source buildsを強く推奨する為である

  - パッケージハまず (with 'make install')のコマンドを使ってインストールするべきである。他のパッケージに使用される前にである。したがって、著者等は次の様に推奨する、あるパッケージから “installation directory”を使って他のパッケージにバイナリーツリーを通してみることをやってみることである。

- CMake (build)サポートを使ったパッケージ:
  - Marlin, MarlinUtil, MarlinReco, CEDViewer, CED, LCIO, GEAR, LCCD, RAIDA, PandoraPFA, LCFIVertex, SiliconDigi, Eutelescope
- CMake modulesは外部パッケージのために作成されている:
  - CLHEP, CERNLIB, CondDBMySQL, GSL, ROOT, JAVA, AIDAJNI

特別な変数

- BUILD_WITH=”CLHEP GSL”
  - ー　これらのパッケージから使うライブラリー、インクルードファイルや定義をパッケージにしらせるものである

- <PKG>_HOME

  - ー　あるパッケージ（pkg）からのホームパスを定義する為の変数である。
    - Standard CMake Find modules differ slightly from ILC Find modules
    - 標準的なCMakeはILCが発見したモジュールから若干異なるモジュールを発見する為のものである

      - ILC Find modules require PKG_HOME variable set
      - ー　ILCはPKG＿HOME　変数セットニ必要なモジュールを見つけ出すものである。
        - バージョンの整合性を確認 (get rid of setting globalローカルな依存性定義するためにグローバルな環境の設定を取り除く）
          - ー　Could instead be called Config<PKG>.cmake
          - ー　Config<PKG>.cmake　と呼ばれているものの代わりに使う

Macros（マクロズ）

- MacroLoadPackage.cmake
  - To be able to use a package by using a
    パッケージを使える様にするには次の
    “Find<PKG>.cmake” モジュールか、または
    “<PKG>Config.cmake” ファイルを使って行う
  - PKG_HOME　変数は適切に設定されている物と仮定する

- MacroCheckDeps.cmake
  - 依存性をチェックするために　MacroLoadPackage.cmake　を使う

Find<PKG>.cmake　モジュール

- the cmake build　で生成された
  <PKG>Config.cmake　と同じことを行う

- パッケージを使うために返される変数

  ー <PKG>_LIBRARIES　　:インクルードディレクトリー
  ー <PKG>_INCLUDE_DIRS　　:ライブラリー
  ― <PKG>_DEFINITIONS　　:　定義

- MacroLoadPackage を使ってこれは実行者の代わりに自動的に行われる

BuildSetup.cmakeについて

- 先行するキャッシング の変数やオプションに関する記述
  – SET(VAR “value” CACHE TYPE “description” FORCE)
- ビルドパラメーターを、コマンドラインに通すことなしに変更する簡単な方法
- Use simple steps to build a package:ビルドパッケージを作る単純なステップを使うべきである。
  - mkdir build ; cd build
  - cmake -C ../BuildSetup.cmake ..
  - make install
- それでも、まだコマンド行でオプションを乗り換える事は可能である。

BuildSetup.cmake

- ーC　オプション以外で使用できるか：
  - cmake -C ../BuildSetup.cmake -C ~/ILCSoft.cmake
  - 次のファイルは以前のファイルを上書きする
  - 'more global'ファイルに定義されている上書きされているパスは便利である。
    - Cmake はグローバルファイルの中の冗長な変数を無視するだけである。
    - ILCをインストールするとILCsoft.cmakeと呼ばれるグローバルファイルを生成する。

- /afs/desy.de/group/it/ilcsoft/v01-01/ILCSoft.cmake を例として参照してください

自身のプロセッサーをCMakeに適合させる

- $Marlin/examples/mymarlin　からコピーを作成する
  - CmakeLists.txt
    - プロジェクト名を変更してなくなっている依存性を（デフォルトは Marlin;LCIO）を追加する

–mymarlinConfig.cmake.in
  - ●名前を変更して <MyProcessor>Config.cmake とする

–BuildSetup.cmake
  - ●自分のシステム構成にしたがって変更する

–cmake_uninstall.cmake.in
  - 'make uninstall'をターゲットとする記述を生成するa tar
    - –変更の必要無し!

CmakeLists.txt のテンプレート

```
# cmake file for building Marlin example Package                cmakeでMarlinのパッケージをビルドする例
# CMake compatibility issues: don't modify this, please!        Cmake の互換性の問題:これはどうか変更しないﾃﾞください。
CMAKE_MINIMUM_REQUIRED( VERSION 2.4.6 )
MARK_AS_ADVANCED(CMAKE_BACKWARDS_COMPATIBILITY)
# allow more human readable "if then else" constructs           より人間の読みやすい"if then else" の構造をしている
SET( CMAKE_ALLOW_LOOSE_LOOP_CONSTRUCTS TRUE )
# User section                                                  ユーザーの変更部分
PROJECT( mymarlin )
# project version                                               プロジェクトの版
SET( ${PROJECT_NAME}_MAJOR_VERSION 0 )
SET( ${PROJECT_NAME}_MINOR_VERSION 1 )
SET( ${PROJECT_NAME}_PATCH_LEVEL 0 )
```

You can add here your own options,
but don't forget at the end of the file to
display them with a MESSAGE(STATUS)
and to also write them properly to cache!

```
# project options                                               プロジェクトのオプション
OPTION( BUILD_SHARED_LIBS "Set to OFF to build static libraries" ON )
OPTION( INSTALL_DOC "Set to OFF to skip build/install Documentation" ON )
# project dependencies e.g. SET( ${PROJECT_NAME}_DEPENDS "Marlin MarlinUtil LCIO GEAR CLHEP GSL" )  依存性の記述
SET( ${PROJECT_NAME}_DEPENDS "Marlin LCIO" )
# set default cmake build type to RelWithDebInfo (None Debug Release RelWithDebInfo MinSizeRel)  デフォルトのビルドの種類
IF( NOT CMAKE_BUILD_TYPE )
SET( CMAKE_BUILD_TYPE "RelWithDebInfo" )
ENDIF()
# set default install prefix to project root directory          デフォールトのインストール先を記述
IF( CMAKE_INSTALL_PREFIX STREQUAL "/usr/local" )
SET( CMAKE_INSTALL_PREFIX "${PROJECT_SOURCE_DIR}" )
```

CmakeLists.txt のテンプレート

```
#include directories                                    インクルードディレクトリー
INCLUDE_DIRECTORIES( "${PROJECT_SOURCE_DIR}/include" )
# install include files                               インクルードファイルのインストール
INSTALL( DIRECTORY "${PROJECT_SOURCE_DIR}/include"
DESTINATION . PATTERN "*~" EXCLUDE PATTERN "*CVS*" EXCLUDE )
# require proper c++                                  通常のc++が必要
ADD_DEFINITIONS( "-Wall -ansi -pedantic" )
# add debug definitions                               デバッグの定義を追加
#IF( CMAKE_BUILD_TYPE STREQUAL "Debug" OR             仮にビルドタイプがデバッグか空欄ならば
# CMAKE_BUILD_TYPE STREQUAL "RelWithDebInfo" )
# ADD_DEFINITIONS( "-DDEBUG" )                        デバッグの定義を追加
#ENDIF()
# get list of all source files                        全ソースファイルのリストが欲しい場合
AUX_SOURCE_DIRECTORY( src library_sources )
( .... )

# DEPENDENCIES: this code has to be placed before adding any library or
# executable so that these are linked properly against the dependencies
IF( DEFINED ${PROJECT_NAME}_DEPENDS OR DEFINED BUILD_WITH OR DEFINED LINK_WITH )
# load macro
   IF( NOT EXISTS "${CMAKE_MODULE_PATH}/MacroCheckDeps.cmake" )
      MESSAGE( FATAL_ERROR
         "\nSorry, could not find MacroCheckDeps.cmake...\n"
         "Please set CMAKE_MODULE_PATH correctly with: "
         "cmake -DCMAKE_MODULE_PATH=<path_to_cmake_modules>" )
   ENDIF()
   INCLUDE( "${CMAKE_MODULE_PATH}/MacroCheckDeps.cmake" )
   CHECK_DEPS()
ENDIF()
```

インクルードディレクトリー

デバッグ定義を追加

もしもっとソースがある場合
はここに追加する(次の例を参照.
LCFIVertex CMakeLists.txt)

依存性はここで
チェックされる!

CmakeLists.txt のテンプレート

```
# LIBRARY                                                    ライブラリー
ADD_LIBRARY( lib_${PROJECT_NAME} ${library_sources} )
# create symbolic lib target for calling target lib_XXX      ターゲットlib_XXXを呼び出すライブラリーを生成
ADD_CUSTOM_TARGET( lib DEPENDS lib_${PROJECT_NAME} )
# change lib_target properties                               ターゲットライブラリーのプロパティを変更
SET_TARGET_PROPERTIES( lib_${PROJECT_NAME} PROPERTIES
# create *nix style library versions + symbolic links       nix スタイルノライブラリとシンボリックリンクノ作成
VERSION ${${PROJECT_NAME}_VERSION}
SOVERSION ${${PROJECT_NAME}_SOVERSION}
# allow creating static and shared libs without conflicts   スタテックライブラリート共有ライブラリを衝突無く生成
CLEAN_DIRECT_OUTPUT 1
# avoid conflicts between library and binary target names   ライブラリーとバイナリのターゲット名称間の衝突ヲ避ける
OUTPUT_NAME ${PROJECT_NAME} )

# install library                                            ライブラリーのインストール
INSTALL( TARGETS lib_${PROJECT_NAME} DESTINATION lib PERMISSIONS
OWNER_READ OWNER_WRITE OWNER_EXECUTE
GROUP_READ GROUP_EXECUTE
WORLD_READ WORLD_EXECUTE )
# create uninstall configuration file                        アンインストールの構成ファイルを作成
CONFIGURE_FILE( "${PROJECT_SOURCE_DIR}/cmake_uninstall.cmake.in"
"${PROJECT_BINARY_DIR}/cmake_uninstall.cmake"
IMMEDIATE @ONLY )
# create uninstall target                                    アンインストールの対象を作成
ADD_CUSTOM_TARGET( uninstall
"${CMAKE_COMMAND}" -P "${PROJECT_BINARY_DIR}/cmake_uninstall.cmake" )
# create configuration file from .in file
CONFIGURE_FILE( "${PROJECT_SOURCE_DIR}/${PROJECT_NAME}Config.cmake.in"
"${PROJECT_BINARY_DIR}/${PROJECT_NAME}Config.cmake" @ONLY )
# install configuration file                                 構成ファイルのインストール
INSTALL( FILES "${PROJECT_BINARY_DIR}/${PROJECT_NAME}Config.cmake" DESTINATION . )
```

CmakeLists.txt のテンプレート

```
# display status message for important variables
MESSAGE( STATUS )
MESSAGE( STATUS "-------------------------------------------------------------------------------" )
MESSAGE( STATUS "BUILD_SHARED_LIBS = ${BUILD_SHARED_LIBS}" )
MESSAGE( STATUS "CMAKE_INSTALL_PREFIX = ${CMAKE_INSTALL_PREFIX}" )
MESSAGE( STATUS "CMAKE_BUILD_TYPE = ${CMAKE_BUILD_TYPE}" )
MESSAGE( STATUS "CMAKE_MODULE_PATH = ${CMAKE_MODULE_PATH}" )
MESSAGE( STATUS "${PROJECT_NAME}_DEPENDS = \"${${PROJECT_NAME}_DEPENDS}\"" )
MESSAGE( STATUS "BUILD_WITH = \"${BUILD_WITH}\"" )
MESSAGE( STATUS "INSTALL_DOC = ${INSTALL_DOC}" )
MESSAGE( STATUS "Change a value with: cmake -D<Variable>=<Value>" )
MESSAGE( STATUS "-------------------------------------------------------------------------------" )
MESSAGE( STATUS )

# force some variables that could be defined in the command line to be written to cache
SET( BUILD_SHARED_LIBS "${BUILD_SHARED_LIBS}" CACHE BOOL

"Set to OFF to build static libraries" FORCE )
SET( CMAKE_INSTALL_PREFIX "${CMAKE_INSTALL_PREFIX}" CACHE PATH

"Where to install ${PROJECT_NAME}" FORCE )
SET( CMAKE_BUILD_TYPE "${CMAKE_BUILD_TYPE}" CACHE STRING
"Choose the type of build, options are: None Debug Release RelWithDebInfo MinSizeRel." FORCE )
SET( CMAKE_MODULE_PATH "${CMAKE_MODULE_PATH}" CACHE PATH
"Path to custom CMake Modules" FORCE )
SET( INSTALL_DOC "${INSTALL_DOC}" CACHE BOOL
"Set to OFF to skip build/install Documentation" FORCE )
# export build settings
INCLUDE( CMakeExportBuildSettings )
CMAKE_EXPORT_BUILD_SETTINGS( "${PROJECT_NAME}BuildSettings.cmake" )
# export library dependencies (keep this as the last line in the file)
EXPORT_LIBRARY_DEPENDENCIES( "${PROJECT_NAME}LibDeps.cmake" )
```

重要な変数に関して状態を表示する

プロジェクトのオプションをここで表示する

いくつかの変数についてはコマンドラインで定義されているから、それをキャッシュに書き込む

ここでも再び自身のプロジェクトオプションを追加して適正にキャッシュに書き込まれるようにする

ビルドの設定をエクスポートする

ライブラリーの依存性をエクスポート
（これは必ず子おファイルの最終行

Marlinにおけるローディングのプロセッサー

- MARLIN_DLL の環境変数
  - $ export MARLIN_DLL=”/path1/lib1.so:/path2/lib2.so：$MARLIN_DLL”
  - $ ./Marlin steer.xml
  - ILCInstall を使ってこの情報は既に生成されたファイルの中にある。
    build_env.sh for the processors found in the config file
    プロセッサーに関するbuild_env.shがconfigファイルの中にある

- その他の共有ライブラリを使ってMARLINにリンクする
  - 自身のMARINのBuildSetup.cmakeに追加する:
    - SET(LINK_WITH "MarlinReco CEDViewer” CACHE STRING “Link Marlin with these optional packages" FORCE)
  - もしくは、コマンドラインから通してやる:
    - $ cmake - C ../BuildSetup.cmake
      - DLINK_WITH="mymarlin PandoraPFA"
      - Dmymarlin_HOME="path_to_mymarlin
      - DPandoraPFA_HOME="path_to_pandora" ..

Marlinにおけるローディングのプロセッサー

- Linux 上でのみ有効なスタテックライブラリをリンクする
  - -DLINK_STATIC_WHOLE_LIBS="path_to_library/libMyprocessor.a"
  - Library gets fully included into the Marlin binary　ライブラリーは全体とし　　で Marinライブラリ含まれる事になる
  - For more than one library:　　1個のライブラリーだけではない
    - -DLINK_STATIC_WHOLE_LIBS="/path1/lib1.a;/path2/lib2.a"

CMake の手引書

- ３－ cmakeを使ってILCインストールを行う。

ILCInstall

```
ilcsoft = ILCSoft("/data/ilcsoft")
ilcsoft.useCMake = True
# python variable for referring the ILC Home directory      ILC Homeディレクトリーを参照するパイソン変数

ilcPath = "/afs/desy.de/group/it/ilcsoft/"
# install RAIDA v01-03      RAIDAv01-03をインストールする
ilcsoft.install( RAIDA( "v01-03" ))
# example for setting cmake variables (“ON”/”OFF” is equivalent to 1/0)   make変数の設定例
ilcsoft.module( “RAIDA” ).envcmake[“BUILD_RAIDA_EXAMPLE”] = “ON”
ilcsoft.module( “RAIDA” ).envcmake[“RAIDA_DEBUG_VERBOSE_FACTORY”] = 1
# use ROOT at: /afs/desy.de/group/it/ilcsoft/root/5.08.00   ROOTを使って処理
ilcsoft.link( ROOT( ilcPath + "root/5.08.00" ))
# use CMakeModules at: /afs/desy.de/group/it/ilcsoft/CMakeModules/v01-00   CMakeモジュールを使う
ilcsoft.use( CMakeModules( ilcPath + "CMakeModules/v01-00" ))
# use CMake at: /afs/desy.de/group/it/ilcsoft/CMake/2.4.6   CMakeをつかう。
ilcsoft.use( CMake( ilcPath + "CMake/2.4.6" ))
# End of configuration file   構成ファイルの終了
```

CMake 変数はRAIDAをビルドする時はコマンド行でパスを通す

ILCInstall

- フィルコールを作った後で
  - ilcsoft-install RAIDA.cfg (display summary)
  - ilcsoft-install RAIDA.cfg -i (install RAIDA)
- ILCSoft.cmakeはインストールスクリプトによって生成される

  - インストレーションはルートディレクトリーに置く事
  - すべてのパッケージのパスはcfg fileに定義されているいなければならない
    - Only the ones that are supported by the cmake modules!
      cmake モジュールによってサポートされているものはこの一つだけ。

ILCInstall (Dev)

- “releases”というディレクトリーにAFSがみつかる。
  インストール構成の参照用ファイルです。
  - その一つをコピーします:
- 自分で動かしたいと思うインストールパッケージを一つ選らぶ
  - ilcsoft.module("RAIDA").download.type="ccvssh"
  - ilcsoft.module("RAIDA").download.username="engels"

- パッケージの依存性を変更する　install -> link
  - ilcsoft.link(ROOT("/data/myILCSoftware/root/5.08.00"))

- 必要なオプションを設定する
  - ilcsoft.module(“RAIDA”).envcmake[“BUILD_RAIDA_EXAMPLE”] = 1
  - ilcsoft.module(“RAIDA”).envcmake[“RAIDA_DEBUG_VERBOSE_FACTORY”] = 1

References　参考文献

- 次のサイトをたどってhttp://ilcsoft.desy.de -> General Documentation ->
  “　How to use the Cmake building tool for the ILC Software”
”ILCソフトウエアに関するCmake のビルディングツールをどう使うか”
- http://www.cmake.org
  このサイトの文書類とFAQ が参考になる
  - Documentation
  - FAQ
- Mastering Cmake (書籍 Cmake マスターする)
  - Ken Martin, Bill Hoffman　（ケン・マーチンとビル・ホフマン）
  - Published by Kitware, Inc.　（キットウエアー社出版）
  - ISBN: 1-930934-16-5　（ISBN：1-930934-16-5）
- This talk: http://ilcsoft.desy.de -> General Documentation
  この発表は上のサイトで参照出来ます

Thank you!　（ご清聴有難う）