The ADCIRC Developers Guide

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

1 Introduction

This document describes how to build ADCIRC and ADCIRC+SWAN, as well as the guidelines that ADCIRC developers should use when developing new code for ADCIRC and its related utilities. It also presents a short tutorial for Subversion (svn), the version control software that we use to coordinate development and ease the task of merging in new features.

2 Build It

This section contains details for building ADCIRC executables and describes the directories and files related to the build process. For a more complete description of the files and directories that are included with the ADCIRC source code, see the official ADCIRC documentation on adcirc.org:

```
http://adcirc.org/home/documentation/generic-adcirc-compile-time-operations/
```

The ADCIRC source tree contains a directory called *work* where the ADCIRC and ADCIRC+SWAN executables are built. This directory contains the following files:

DEFINITIONS

config.guess
: A shell script that is called automatically by make during the build process to guess the hardware and operating system that are being used in the build process. You can execute this shell script yourself on the command line to see what it guesses and confirm that it is guessing correctly. The resulting platform guess is then used automatically in the *cmplrflags.mk* file.

cmplrflags.mk
: Contains sets of compiler flags that are appropriate for various platforms. The build process uses the output from *config.guess* to select the appropriate set of compiler flags. In the past, in the days of many varieties of proprietary unix, the hardware type and operating system type were sufficient to determine which compiler flags to select. Now that most installations of ADCIRC are on linux, it is often necessary to provide additional hints about which set of compiler flags to use, and these additional hints are provided on the command line when make is executed, as described below.

makefile
: The makefile contains instructions for building *adcirc*, *padcirc*, *padcswan* (if the SWAN source code is also present), *adcprep*, *adcpost*, *hstime*, and *aswip* on linux or unix.

When building ADCIRC for the first time on a new (to me) platform, I normally begin by going to the ADCIRC *work* directory

```
cd work
```

and running *config.guess* manually to see what it guesses about the platform architecture.

```
./config.guess
```

Then I open up the *cmplrflags.mk* file, to see if the relevant platform and/or compiler is defined in the file, and if it is, how it is triggered on the make command line with the `compiler=` notation.

I also test the availability of the compilers that I will be using by executing the `which` command. For example, if I'm using the Intel compilers, I'll type `which ifort`. If I'll be running ADCIRC in parallel, I'd also try `which icc` and `which mpif90` to make sure they're all available. If they're not, I take it up with my local sysadmin.

Looking in the *cmplrflags.mk*, I would find (for example) that I can use a set of Intel-appropriate compiler flags by setting `compiler=intel` on the make command line. As a result, my first tentative step in building all the ADCIRC executables would be

```
make adcirc compiler=intel
```

Assuming that this completes successfully, I would try

```
make padcirc compiler=intel
```

Now that padcirc has been built successfully, the parallelization modules used by parallel SWAN are available (they share the same libraries). In other words, **the *padcswan* build depends on the *padcirc* build**, so *padcirc* must be built before *padcswan*. If you'd like to build ADCIRC+SWAN, it is time to switch to the SWAN subdirectory.

```
cd ../swan
```

SWAN has a completely separate build system, since it is normally a completely separate code. But the SWAN build system must solve the same problems as the ADCIRC build system, that is, detecting the nature of the underlying platform and suggesting an appropriate set of compiler flags.

The SWAN build system has a perl script called *platform.pl* that performs much the same function as *config.guess* does for ADCIRC. The difference is that *plaform.pl* picks the compiler flags and writes them to a file called *macros.inc* for use during the build process.

Several sets of *macros.inc* files are already available in the ADCIRC source code for use with certain platforms; type `ls macros*` to see the currently available choices. To write a set of compiler flags for your platform, type

```
perl ./platform.pl
```

You may want to have a look at the resulting *macros.inc* file to be sure that it represents the compiler you're using. If it looks ok, the next step is to create a test build of *punswan*. You won't actually use *punswan*; in fact, you'll completely delete it, even if it builds successfully. You're just trying to build it to test the validity of your compiler flags.

```
make punswan
```

Once *punswan* builds successfully, go ahead and get rid of it as well as any object files that were built along the way with

```
make clobber
```

Now you're ready to switch back to the work subdirectory and build *padcswan*, if you're interested in that:

```
cd ../work
make padcswan compiler=intel
```

Once *padcswan* has been built successfully, you can build *adcprep*, the program that prepares your input files for a parallel run. If you also plan to run ADCIRC+SWAN in parallel, then add the `SWAN=enable` string to the make command line, as follows:

```
make adcprep compiler=intel SWAN=enable
```

You'll see the invokation of your C compiler during the build process for *adcprep*, since *adcprep* uses the metis library (which is written in C) to perform domain decomposition.

Once you've successfully built *adcprep*, the main set of executables is complete. However, there are a couple utility programs you may like to build as well. The first of these is a very small program called *hstime* whose only purpose is to read the time in seconds from an ADCIRC hotstart file and write it out to the console window. Use a command like *make hstime compiler=intel* to build this utility.

The final executable that you may like to build is *aswip*, the Asymmetric Wind Input Preprocessor. This utility program is required for the use of an asymmetric vortex model for tropical cyclones in ADCIRC. One wrinkle for this program is that it calls many subroutines inside ADCIRC, therefore **there is a dependency of *aswip* on *adcirc***, meaning that *adcirc* executable must be

built first in order for *aswip* to be built successfully. It is built analogously with the other ADCIRC executables, with a command like *make aswip compiler=intel*.

Another issue with the building of *aswip* is that it is built inside the same directory as *adcirc*, and both programs have a "main" program section inside them, creating a conflict. Therefore, in order to rebuild *adcirc* after *aswip* has been built, *aswip* its object file must be deleted first, to avoid a build error when *adcirc* is rebuilt.

All of the above instructions will build the ADCIRC executables without support for NetCDF input or output. This was done intentionally, to simplify the build process and avoid complications resulting from the vagaries of building with NetCDF.

On the other hand, the use of NetCDF is recommended as a result of the many advantages of NetCDF files. ADCIRC actually supports the use of three different NetCDF capabilities: NetCDF3, NetCDF4 (classic model), and NetCDF4 with internal compression.

If you've already built all the executables without NetCDF, its necessary to get rid of them so that you can build with NetCDF.

```
make clobber
```

To enable compilation with NetCDF3, the command line listed at the beginning of these instructions for building ADCIRC would be modified as follows:

```
make adcirc compiler=intel NETCDF=enable NETCDFHOME=/usr
```

with the NETCDFHOME variable indicating the directory where NetCDF was installed. The ADCIRC build system expects this directory to contain `include/netcdf.inc`.

If NetCDF4 without internal compression is installed on your platform, the above would change to the following:

```
make adcirc compiler=intel NETCDF=enable NETCDFHOME=/usr NETCDF4=enable
```

in which case the directory specified in the NETCDFHOME variable would also be expected to contain the `include/netcdf.mod` Fortran module file.

Finally, if your NetCDF4 installation supports internal compression (only introduced in NetCDF4 version 4.1), and you want ADCIRC to automatically make use of internal compression, change the above command line as follows:

```
make adcirc compiler=intel NETCDF=enable NETCDFHOME=/usr NETCDF4=enable
NETCDF4_COMPRESSION=enable
```

These changes in the make command line must be made for the build of each of the ADCIRC executables so that they are all consistent.

In general, building ADCIRC with NetCDF support is tedious and usually requires several rounds of emails to local sysadmins to determine the right combination of compiler flags and options, because they vary siginificantly from platform to platform. Have a look at the NetCDF-related compiler flags in *cmplrflags.mk* for ideas to get you started.

3 Development Strategy

The ADCIRC development strategy consists of the maintenance of two separate versions at any given time: the stable version and the development version. The stable version only receives bug fixes, while the development version receives bug fixes as well as new features.

At some point, the development version becomes stable. At that time, a stable branch is created from it, and then new features are added to the previosly stable code, creating a new development version. At that time, the major version number will be incremented, e.g., stable v48.xx will become the next development version as v49.00.

The minor version number is changed each time there is a change in the code and the modified code is committed back to the repository. Details of the implementation of this strategy with Subversion are provided in the section entitled "Subversion".

4 Coding Standards

ADCIRC has had many individual contributors and has received code accretions over many years. A set of uniform coding standards has not been defined, and as a result, ADCIRC contains many different styles of Fortran. This section provides a set of style guidelines for contributing code to ADCIRC.

4.1 The Basics

- IMPLICIT NONE at the beginning of each subroutine.
- Fixed form, never exceeding the 72 column limit, even for comment lines.
- When adding code to an existing subroutine, make the new code match the style of the surrounding code, even if you'd prefer another style.

4.2 Maintainability

When adding code that will be used in slightly different ways in different contexts, make it a subroutine, rather than cutting and pasting several times and making small changes to each cut-and-pasted section. Although it is faster to write code with cut-and-paste, the resulting code is harder to maintain, since each cut-and-pasted section will have to be modified individually later. Also, it is easier to make mistakes when working with cut-and-pasted code.

Rick has expressed a desire for greater modularity ... particularly with the number of variables in the global module. When adding a major new feature, please consider the modularity of the data it requires. In other words, if new variables can be made private to a particular module rather than global, please do so.

5 Release Process

The release process informal, but generally includes the following steps:

- Set and stick to a list of new features as the goal for the new release.
- Have a consensus among ADCIRC developers that the goal has been achieved.
- Run tests that cover the new features as well as making sure old features still work.
- Fix issues revealed by failed tests.
- Distribute the release candidate code to collaborative users to make sure their runs perform as expected on the new code.
- Fix issues revealed by user dismay.
- Update the docs to reflect the changes. If input files or file formats have changed, produce release notes to highlight the changes.
- Publicly announce, release, and brag about a shiny new version of ADCIRC.

These tests are often referred to as "regression tests," i.e., a means to detect a regression in correctness or functionality due to changes in the code. A regression test suite can be built up pretty easily. There are a number of small test cases on adcirc.org, so the initial set of tests should consist of those. We may also want to differentiate between tests for correctness and tests for functionality. The former is making sure you're getting the expected model results and the latter is making sure that nothing is broken (i.e., adcprep, i/o, message passing, etc).

The test suite gets built up based on different purposes. One may want to exercise more options and more cases, but it is also important to test for known bugs that might have been reintroduced. In otherwords, once a bug is discovered and fixed, a test should be created specifically for that bug to make sure it doesn't appear again.

And lastly, a small test suite should be included with each distribution and a more comprehensive should be created that runs during development and before a release. A step in this direction has been made using ant in the autotest directory.

6 Subversion

Subversion is a version control system. That is, it provides a means for many software developers to collaborate on a single software project.

A subversion repository is a storehouse of code for a particular project. Subversion repositories consist of a trunk, which is the mainline code that everyone shares, and any number of branches that are created by and for individual users.

A branch may be created by a developer that has a lot of changes to make over a longer period of time. This lone developer can work on a branch without affecting the mainline code. Once the changes in the branch are satisfactory, the branch can be merged back to the trunk, making the changes available to all.

A further advantage of Subversion is that it automates the process of merging new features into ADCIRC. For example, in the past, without Subversion, developers modified their local copies of ADCIRC and wanted to merge the modifications into the mainline code. This required hours or days of painstakingly examining each change they had made and cutting and pasting the changes one by one into the up-to-date version of ADCIRC. With Subversion, the software examines the changes that were made locally and makes corresponding changes to the mainline version in an automated way.

6.1 Policies

Working with subversion requires greater coordination among ADCIRC developers. For example, if two developers change the same line of code in two different branches in two different ways, this will create a conflict that will have to be resolved manually.

Furthermore, if one developer makes changes to trunk that prevent the code from compiling, it will be difficult for other developers to continue working if they update to the latest code. As a result, ADCIRC development with Subversion will adhere to the following policies:

- Communicate. Jason Fleming is responsible for making sure ADCIRC development is smooth, pain-free and productive---when in doubt, email him (jason.fleming@seahorsecoastal.com).
- Also use the adcirc-dev mailing list to keep everyone informed of what you are doing.
- If you are not on the adcirc-dev mailing list and would like to be, email Jason Fleming.
- Make a branch to develop new features, rather than making changes to trunk.
- Don't commit changes to trunk that prevent the code from compiling, at least on your platform. You should also confirm that your code compiles with and without NetCDF.

6.2 Instructions

Comprehensive documentation for Subversion is available: http://svnbook.red-bean.com/.

The ADCIRC code is hosted at the following Subversion repository location:

```
https://adcirc.renci.org/svn/adcirc
```

To check out code from the trunk, please type

```
svn checkout https://adcirc.renci.org/svn/adcirc/trunk
```

Here are a few examples of things that we commonly do with svn.

6.2.1 Compare Revisions

In order to see the changes between revisions of the repository, type (for example)

```
svn diff  -r 15:16 owiwind.F
```

6.2.2 Package Up Code From Repository

In order to create a tar file that contains just the ADCIRC code and excludes the .svn directories, the first step is to export the code to a new subdirectory. For example:

```
svn export ./trunk ./adc99_99
```

Would copy the local files under version control from the trunk subdir to the adc99_99 subdir, excluding the .svn files. Particular revisions of the svn repository can also be extracted, see the svn documentation for details on how to do this.

6.2.3 Pull Old Version from Repository

One advantage of using svn is that it is easy to go back to an older version of the code, if necessary. The first thing to do in this case is usually to look at the svn log to see what changes were made, so that you can select the right version of the repository to extract. Go to the subdirectory of the code tree that you want an older version of and issue the command

```
svn log
```

Look at the log entries, and note the revision that you'd like to extract. Let's say you want to retrieve adcirc48/trunk at revision 65 of the repository (for example). Issue the following command:

```
svn checkout -r 65 https://adcirc.renci.org/svn/adcirc/trunk
```

6.2.4 Make a Branch

To make a branch off trunk to add a new feature and call it the myfeature branch:

```
svn copy https://adcirc.renci.org/svn/adcirc/trunk
 https://adcirc.renci.org/svn/adcirc/branches/myfeature
 --message "Creating branch to add myfeature"
```

Then to start using the newly created branch:

```
svn checkout https://adcirc.renci.org/svn/adcirc/branches/myfeature
```

A This Document

This document was prepared from the text file ADCIRCDevGuide.txt using software called asciidoc (http://www.methods.co.nz/-asciidoc/). The document can be formatted as an html page with the command

```
asciidoc -a toc2 ADCIRCDevGuide.txt
```

or formatted as a pdf with the command

```
a2x --format=pdf -a toc2 ADCIRCDevGuide.txt
```