Wire Cell Toolkit Manual

Brett Viren

April 15, 2017

Contents

1 Installation

1.1 Installation

The Wire Cell Toolkit (WCT) should be easy to build on any POSIX'y system with a recent C++ compiler. This section describes how to build releases and development branches, it gives guidance for supplying the few software dependencies, and documents how releases are made.

1.1.1 Toolkit installation

Warning: This assumes you already have available the required dependencies. See section 1.1.2.

Installation requires four steps:

1. get the source

2. configure the source
3. build the code
4. install the results

Source code WCT source is composed of several packages (see section 4) and all source is available from the Wire Cell GitHub organization. Releases of each package are made and documented on GitHub (*eg* here) and can be downloaded as archives. However, using git to assemble a working source area is recommended and easier. Releases and development branches are handled slightly differently.

To obtain a release requires no GitHub authentication:

```
$ git clone --recursive --branch 0.5.x https://github.com/WireCell/wire-cell-build.git
```

This gets the tip of the release branch for the `0.5.x` series. If a specific release is desired a few more commands are needed. For example, if the `0.5.0` release that started the series is wanted:

```
$ git checkout -b 0.5.0 0.5.0
$ git submodule init
$ git submodule update
$ git submodule foreach git checkout -b 0.5.0 0.5.0
```

To obtain the development branch requires SSH authentication with GitHub:

```
$ git clone --recursive git@github.com:WireCell/wire-cell-build.git wct
```

Which ever way the source is obtained, enter the resulting directory

```
$ cd wire-cell-build/
```

Hint: At some time later if there is a need to switch between HTTP or SSH a `switch-git-urls` script is available in this directory.

Configuring the source At a minimum, the source must be configured with an installation location for the build results and to allow it to find its dependencies. This, and the remaining steps are done with the provided `wcb` script which is an instance of Waf.

```
$ ./wcb --prefix=/path/to/install configure
```

This will print the results of the attempts to detect required and optional dependencies. Missing but optional dependencies will not cause failure. See below for guidance on installing dependencies if this step fails or if desired optional dependencies are not found.

Dependencies are first located in standard system locations, barring that those that are traditionally installed with support for `pkg-config` may be found by suitably setting the `PKG_CONFIG_PATH` environment variable. To force a particular location for any given dependency a `--with-*` option may be used.

Note: As shown in the example below, the flags to locate Boost are slightly different.

```
$ ./wcb configure --prefix=/path/to/install \
   --boost-includes=$WCT_EXTERNALS/include --boost-libs=$WCT_EXTERNALS/lib --boost-mt
   --with-eigen=$WCT_EXTERNALS --with-jsoncpp=$WCT_EXTERNALS --with-tbb=$WCT_EXTERNALS
   --with-root=$WCT_EXTERNALS --with-fftw=$WCT_EXTERNALS --with-root=$WCT_EXTERNALS
```

This example assumes all externals are available in a directory set by the `WCT_EXTERNALS` variable. This variable is **not** used by the build and is only used to make this example brief.

Building the source After a successful source configure step, the results are cached and any long command line need not be repeated. To build the source simply run:

```
$ ./wcb
```

If there are build failures more information can be obtained by repeating the build with more verbosity:

```
$ ./wcb -vv
```

The build will try to run tests which can be avoided to save time:

```
$ ./wcb --notests
```

Install the results To install the build results into the location given by `--prefix` simply issue:

```
$ ./wcb install
```

Other build commands These other commands may be useful:

```
$ ./wcb clean          # clean build products
$ ./wcb distclean      # also clean configuration
                       # build with debug symbols
$ ./wcb configure --build-debug=-ggdb3 [...]
                       # to save some time, just
                       # rebuild the given test
                       # and don't run any tests
$ ./wcb --notests --target=test_xxx
$ ./wcb --help         # see more options.
```

1.1.2 Guide for installation of dependencies

Manual DIY

Spack Installation and setup with Spack Views or with EM.

UPS Setup with UPS.

1.1.3 Release management

2 Configuration

2.1 Introduction and Scope

As the Wire Cell Toolkit (WCT) is a toolkit, it is up to the application to provide some mechanism for the user to provide configuration information. Users of an application not provided by the WCT itself should refer to its documentation.

WCT itself provides a such a mechanism which is exposed to the user by the `wire-cell` command line application. Any application may easily adopt this same mechanism by making use of the `WireCell::ConfigManager` class.

This WCT configuration mechanism is described here from the point of view of **user** and **developer**. Details for each role are given in the following sections. However, both user and developer must understand one aspect of WCT internal design in order to understand configuration. One aspect of WCT is that an application is composed of a number of *component* classes. Components work together in some way to enact the job of the application. A component is specifically a C++ class which implements one or more *interface* base classes. One interface pertinent here is `IConfigurable`. A

component that implements this interface is called a *configurable component* or just *configurable*. A configurable then is the atomic unit of WCT configuration and this unit is reflected in what the user provides for configuration and what developers should expect if they write configurable components.

The user then provides an ordered list of *configuration objects* or simply *configurations*. Each configuration is associated by WCT with exactly one *instance* of a configurable component class. This association is done via two string identifiers. The pair are also used to initially construct and later locate the instance of any type of WCT component (not just configurables):

type specifies the "configurable type" which often matches the C++ class name with any C++ namespace removed. However, developers of configurable components are free to chose any unique type name.

name specifies a "configurable instance", that is an C++ object instance of the C++ class associated with the configurable type identifier. The name is free form and may be omitted in which case it defaults to the empty string. A specific name is needed if multiple instances are required or if multiple configurables require sharing a component.

Finally, configurations have a third attribute:

data specifies a data structure following a schema specific to the configurable type. This is the "payload" that WCT gives to the instance of the configurable component.

In the next section, WCT user-configuration support is described. The following section gives guidance to developers who wish to write their own configurable components.

2.2 Configuration from a user point of view USER

Users of the WCT command line interface `wire-cell` or any WCT application that uses `WireCell::ConfigManager` can provide configuration information in the form of one or more files. This files express the same ordered list of configuration objects as described above.

2.2.1 File formats

WCT supports two related configuration file formats: JSON and Jsonnet. Of the two, JSON is more fundamental while Jsonnet provides a way to

better organize and construct complex configurations. Jsonnet support is a compile-time option. The stand-alone `jsonnet` program may also be used to evaluate Jsonnet into JSON.

2.2.2 Basic command line

A user gives one or more configuration files to the `wire-cell` application each with a `-c` flag:

```
$ wire-cell -c myparameters.cfg [...]
```

If a relative path is given, the file will be searched for starting in the current working directory and then in each directory listed in a `WIRECELL_PATH` environment variable, if given. When multiple configuration are used, their top-level arrays are conceptually concatenated in the order on which they are given on the command line.

The user can also dump out the hard-coded default configuration for one or more components:

```
$ wire-cell \
    -p <plugin> \
    -D <component1> \
    -D <component2> \
    --dump-file=mydefaults.cfg
```

Here the components are specified by their "type" identifier as described above. As the example shows, a plugin must be given. The `wire-cell` application itself does not "know" about any components as they are all dynamically loaded when needed. However, the application must be told a collection of plugins in which to find components. A plugin typically takes the name of its shared library with the `lib` prefix and `.so` extension removed.

The user must know what components to dump. There is no way for the application to iterate over all possible components. In general, it is up to the provider of a plugin to catalog what component types it provides. WCT provides a simple script that will search the WCT source, determine the components and dump them out using `wire-cell`. Result of this dump, possibly out of date, is available in the wire-cell-cfg repository.

2.2.3 Diving into JSON

An example configuration dump from this command

```
$ wire-cell -D TrackDepos -p WireCellGen
```

produces:

```
[
   {
      "data" : {
 "clight" : 1,
 "step_size" : 0.10000000000000001,
 "tracks" : []
      },
      "name" : "",
      "type" : "TrackDepos"
   }
]
```

Here we see an array holding one element which is an object with the `type`, (instance) `name` and payload `data` structure as described above. If `wire-cell` were to load this configuration it would create a default instance of the component type `TrackDepos` which happens to correspond to the C++ class `WireCell::Gen::TrackDepos` (see the simulation package manual for more information). This component is responsible for produces deposition (`IDepo`) objects using a simple linear source model.

The `tracks` array in this example is empty and no depositions would be produced. The user most certainly should specify a nonempty set of tracks. In principle, the user may produces a huge `tracks` array. WCT support bzip2 compressed JSON files (see the section on persistence in the util package manual.

2.2.4 Limitations of JSON

As the complexity of a `wire-cell` job grows, hand crafting JSON becomes tedious and error prone. Splitting the files and/or using `WIRECELL_PATH` can provide some rudimentary means of organizing a large, complex configuration.

However, a user will quickly outgrow direct authoring of JSON files. An accomplished one will likely turn to some form of JSON generation using a more expressive language. Or, some configuration may need to be extracted or converted from other source. For example, Geant4 steps might be extracted and fed into `TrackDepos` as a long `tracks` array. The user is free to generate JSON in this manner in any way they desire as long as the result conforming to the required schema.

Another limitation is that any numerical quantities **must** be expressed in the base units used by the WCT *system of units* (see the section on units in the Utilities manual). This places a burden on the configuration author and is a source of error.

WCT provides a more powerful JSON-like configuration file format as described next.

2.2.5 Learning Jsonnet

WCT provides support for configuration files following the Jsonnet data templating language. This language is evaluated to produce JSON. If WCT is compiled with support it will evaluation Jsonnet files directly. Otherwise the user may install and run the `jsonnet` command line program to produce JSON.

To learn how to write Jsonnet in general, the user should refer to its documentation which is excellent. There is no one right way to write Jsonnet, however, the wire-cell-cfg package provides a number of examples and support files that can help the user craft their configuration in Jsonnet. In particular the WCT system of units and some common data structures used by WCT are exported to Jsonnet in wirecell.jsonnet. Some of this exported functionality is illustrated below.

WCT locates Jsonnet files as it does JSON files and in particular using the environment variable `WIRECELL_PATH`. However, it does not (currently) support compressed Jsonnet files.

System of units Wire Cell provides an internal system of units as described in the section on units in the Utilities manual). As stated above, users must take care to give numerical quantities JSON in base WCT units. If writing Jsonnet this is less trouble as once can label a quantity by multiplying it with a symbolic unit. For example:

```
local wc = import "wirecell.jsonnet";
[
    {
    type:"TrackDepos",
    data: {
step_size: 1.0 * wc.millimeter,
// or could abreviate with wc.mm
}
    }
]
```

Functions Some data sub-structures are needed in multiple laces and it can be laborious to write them by hand. Jsonnet provides functions to assist in this. A number of functions are defined to assist in representing common data types. For example `point()` and `ray()`:

```
{
  // ...
  tracks : [ wc.ray(wc.point(10,0,0,wc.cm),
             wc.point(100,10,10,wc.cm)) ]
},
```

Default Structures Some common structures are defined with default objects so that they may be extended/overridden. For example, the `Node` object defines a default `type`, `name` and `port` to be used in a graph connection. It is typical to override at least the `type`:

```
graph:[
{
  tail: wc.Node {type:"TrackDepos"},
  head: wc.Node {type:"DumpDepos"}
},
//...
]
```

Commas One of the most irritating aspect of crafting JSON files by hand is that any array or object must not have a internal trailing comma. Jsonnet allows this otherwise extraneous comma. For this reason alone and if no other features are used, writing Jsonnet is worth the added dependency!

2.2.6 Specific detector support

The `wire-cell-cfg` package also provides support for popular LArTPC detectors. You can find these files under a directory named for the experiment (such as `./uboone/`).

2.3 TODO Configuration from a developer point of viewDEVEL

For the C++ part of developing WCT components or applications the developer should refer to the configuration section in the manual on WCT Internals.

In addition, a developer is encouraged to provide Jsonnet files that abstract away any less important details and give users a simplified way to configure the developers components.

In particular, if the developer writes multiple components, an application component or a component that refers to another component, working example configuration files should be provided.

3 Internals

3.1 Intro

This doc describes the Wire Cell Toolkit (WCT) core internal structure and support facilities. The "batteries included" or reference implementations that are also provided as part of the toolkit are documented in the individual package manuals.

3.2 Packages

The WCT is composed of a number of *packages*. Each package has an associated with a Git source repository. Most packages produce a shared library, which may also be a WCT plugin library, C++ header files, some number of main or test applications. Others include a single package holding all Python code in various modules, a package providing support for developing WCT configuration files and the documentation package holding this document. One special type of package is a *build* package described more in section on the build package.

3.2.1 Names

Package repositories are named like `wire-cell-<name>` where `<name>` is some short identifier giving indication of the main scope of the package. In the documentation the `wire-cell-` prefix is often dropped and only this short name is used.

If a package produces a shared library it should be named in `CamelCase` with a prefix `WireCell`. For example the `gen` package produces a library `libWireCellGen.so`. As a plugin name or an entry in the build system, the `lib` and `.so` are dropped. If the package has public header files to expose to other packages they should use this same name for a subdirectory in which to hold them. Package layout is described move below.

3.2.2 Dependencies

Some of the C++ packages are designated as *core* packages. These include the packages providing the toolkit C++ structure (described later in this document) as well as the reference implementations (eg, `gen`, `sigproc`). These packages have strict requirements on what dependencies may be introduced and in particular their shared libraries are not allowed to depend on ROOT (although their apps and tests are, see sections 3.2.3 and 3.2.4).

The base package is `util` and it *must* not depend on any other WCT package. The next most basic is `iface` and it *must* not depend on any other WCT except `util`. Core implementation packages such as `gen` or `sigproc` may depend on both but should not depend on each other.

Fixme: there is a need to factor some general utility routines and data structures that depend on `iface` and which the implementation packages should use that needs to be created.

WCT also provides a number of peripheral implementation packages, which are free to have more dependencies, including ROOT, than "core" packages. These are mostly for the purpose of providing WCT components which provide file I/O. The `sst` package in particular support the so called `celltree` ROOT `TTree` format used by the Wire Cell prototype code.

Finally, there may be third-party implementation packages. They are free to mimic WCT packages but WCT itself will not depend on them. They should not make use of the `WireCell::` C++ namespace.

3.2.3 Package structure

The WCT package layout and file extensions must follow some conventions in order to greatly simplify the build system. In the description below, `WireCellName` is as described above.

`src/*.cxx` C++ source file for libraries with .cxx extensions or private headers

`inc/WireCellName/*.h` public/API C++ header files with .h extensions

`test/test_*.cxx` main C++ programs named like test$*$.cxx, may also hold Python, shell scripts, private headers

`apps/*.cxx` main application(s), one appname.cxx file for each app named `appname`, should be very limited in number

In the root of each C++ package directory must exist a file called `wscript_build`. It typically consist of a single line with a method call like:

```
bld.smplpkg('WireCellName', use='...')
```

The `bld` object is automagically available. If the package has no dependencies then only the name is given. Most packages will need to specify some dependencies via `use` or may specify a different list of dependencies just for any applications (using `app_use`) or for the test programs (via `test_use`). Dependencies are transitive so one must only list those on which the package directly depends.

Fixme: make a script that generates a dot file and show the graph.

3.2.4 Build package

To actually build WCT see the section on toolkit installation. The build system is based on Waf and uses the `wcb` command and a `wscript` file provided by the top level *build package*. Besides holding the main build instructions this package aggregates all the other packages via Git's "submodule" feature. In principle, there may be more than one build package maintained. This allows developers working on a subset to avoid having to build unwanted code. In practice there is a single build package which is at: `https://github.com/wirecell/wire-cell-build`.

3.2.5 Adding a new code package

To add a new code package to a build package from scratch, select a `<name>` following guidance above and do something like:

```
$ mkdir <name>
$ cd <name>/
$ echo "bld.smplpkg('WireCell<Name>', use='WireCellUtil WireCellIface')" > wscript_bui
$ git init
$ git add wscript_build
$ git commit -a -m "Start code package <name>"
```

Replace `<name>` with your package name. You can create and commit actual code at this time as well following the layout in 3.2.3.

Now, make a new repository by going to the WireCell GitHub and clicking "New repository" button. Give it a name like `wire-cell-<name>`. Copy-and-paste the two command it tells you to use:

```
$ git remote add origin git@github.com:WireCell/wire-cell-<name>.git
$ git push -u origin master
```

If you made your initial package directory inside the build package move it aside. Then, from the build package directory, add this new repository as a Git submodule:

```
$ cd wire-cell-build/ # or whatever you named it
$ git submodule add -- git@github.com:WireCell/wire-cell-<name>.git <name>
$ git submodule update
$ git commit -a -m "Added <name> to top-level build package."
$ git push
```

In order to be picked up by the build the new package short name must be added to the `wscript` file.

3.3 Coding conventions

3.3.1 C++ code formatting

- Base indentation *should* be four spaces.
- Tabs *should* not be used.
- Opening braces *should not* be on a line onto themselves, closing braces *should be*.
- Class names *should* be `CamelCase`, method and function names *should be* `snake_case`, class data attributes *should be* prefixed with `m_` (signifying "member").
- Doxygen triple-slash `///` or double-star `/** */` comments *must* be used for in-source reference documentation.
- Normal comments *may* be used for implementation documentation.
- Interface classes and their types and methods *must* each have a documenting Doxygen comment.
- Header files *must* have `#ifndef/#define/#endif` protection.
- The C++ `using namespace` keyword *must not* be used at top file scope in a header.
- Unused headers *should not* be retained.
- Any =#include# need in an implementation file but not the corresponding header file *should not* be in the header file.

3.3.2 C++ namespaces

- All C++ code part of WCT proper and which may be accessed by other packages (eg, exported via "public headers") *must* be under the `WireCell::` namespace.
- WCT core code (`util` and `iface` packages) *may* exist directly under `WireCell::` but bare functions *must* be in a sub namespace.
- Non-core, WCT implementation code (eg contents of `gen` package) *must* use secondary namespace (eg `WireCell::Gen::`).
- Any third-party packages providing WCT-based components or otherwise depending on WCT *should not* use the `WireCell::` namespace.

3.4 Interfaces

A central design aspect of the WCT is that all "important" functionality which may have more than one implementation must be accessed via an *pure abstract interface class*. All such interface classes are held in the iface package. Interface classes should present a very limited number of purely abstract methods that express a single, cohesive concept. Implementations typically inherent from more than one interface. If two concepts are close but not cohesive they are best put into two interface classes. Besides defining the method interface, Interface classes may define types. They may also be templated.

After an implementation of an interface is instantiated and leaves local scope it should be referenced only through one of its interfaces. It should be held through an appropriately typed `std::shared_ptr<>` of which one should be defined as `ITheInterface::pointer`.

Interfaces are used not only to access functionality but the data model for major working data is defined in terms of interfaces inheriting from `WireCell::IData`. Once an instance is created it is immutable.

Another category of interfaces are those which express the "node" concept. They inherit from `WireCell::INode`. These require implementation of an `operator()` method. Nodes make up the main unit of code. They are somewhat equivalent to `Algorithm` concept from the Gaudi framework where the `operator()` method is equivalent to Gaudi's `execute()` method. They also require some additional instrumenting in order to participate in the data flow programming paradigm described below.

3.5 Components

Components are implementations an interface which itself inherits from the `WireCell::IComponponent` interface class (this interface class is in `util` as a special case due to dependency issues. fixme: needs to be solved with a general package depending on both `iface and =util`). This inheritance follows CRTP.

Components also must have some tooling added in their implementation file. This is in the form of a single CPP macro which generates a function used to load a factory that can create and retain instances based on a *type* name and an *instance* name. For `WireCell::Gen::TrackDepos` the tooling looks like:

```
#include "WireCellUtil/NamedFactory.h"
WIRECELL_FACTORY(TrackDepos, WireCell::Gen::TrackDepos, WireCell::IDepoSource, WireCel
```

Note, this macro needs to appear before any `using namespace` directives. The arguments to the macro are:

1. The "type name" which is typically the class name absent any namespace prefixes. It must be unique across the entire WCT application.
2. The full class name.
3. A list of all interfaces that it implements.

A component may be retrieved as an interface using the *named factory pattern* implemented in WCT. If the component has yet to be instantiated it will be through this lookup. This is performed with code like:

```
#include "WireCellUtil/NamedFactory.h"
auto a = Factory::lookup<IConfigurable>("TrackeDepos");
// or
auto b = Factory::lookup<IConfigurable>("TrackeDepos","some instance name");
// or
auto c = Factory::lookup_tn<IConfigurable>("TrackeDepos:");
// or
auto d = Factory::lookup_tn<IConfigurable>("TrackeDepos:some instance name");
```

The four example differ in if an instance name is known and if it is known separately from the type name or in the canonical join (eg as `type:name`). The returned value in this example is a `std::shared_ptr<const IConfigurable>`.

This example accesses the `IConfigurable` interface of `TrackDepos`. Not typically required by most code but there exists also a function `lookup_factory()` to get the factory that constructs the component instance.

3.6 Configuration

One somewhat special component interface is `IConfigurable`. A class inheriting from this interface is considered a *configurable component* such as `TrackDepos` in the above example. It is *required* for any main application using the WCT toolkit to adhere to the Wire Cell Toolkit Configuration Protocol. This is a contract by which the main application promises to do the following:

1. Load in user-provided configuration information (see the configuration section of hte manual)
2. Instantiate all configurables referenced in that configuration.
3. Request the default configuration object from each instance.
4. Update that object with, potentially partial, information provided by the user.
5. Give the instance the updated configuration object.
6. Do this before entering any execution phase of the application.

If the main application uses `WireCell::Toolkit` then the protocol can be enacted with code similar to

```
using namespace WireCell;
ConfigManager cfgmgr();
// ... load up cfgmgr
for (auto c : cfgmgr.all()) {
    string type = get<string>(c, "type");
    string name = get<string>(c, "name");
    auto cfgobj = Factory::lookup<IConfigurable>(type, name); // throws
    Configuration cfg = cfgobj->default_configuration();
    cfg = update(cfg, c["data"]);
    cfgobj->configure(cfg);
}
```

FIXME: shouldn't we put this all inside `ConfigManager`?

Developers of new configurables should keep this protocol in mind and should refer to existing configurables for various useful patterns to provide their end of the exchange.

3.7 Execution Models

3.7.1 Ad-hoc

Direct calling of utility functions and concrete objects.

3.7.2 Component

Concrete components.

3.7.3 Interface

Using NamedFactory.

3.7.4 Data flow programming

Using abstract DFP

4 Packages

4.1 Utilities

Introduction.

4.1.1 Units

Describe units.

4.1.2 Persistence

Describe support for persistent files including compression and location.

4.1.3 Etc

....

4.2 Interfaces

Brief overview but it's also in `./internals.org` so don' t over do it.

4.2.1 Data

4.2.2 Nodes

4.2.3 Misc

4.3 Simulation

Scope and intro blah blah.

4.3.1 Depositions

4.3.2 Drifting

4.3.3 Response

4.3.4 Digitizing