A Users guide to BFD

This list is arranged in a semi-serial order. You'll use the earlier commands a few times at first and then the later in a cyclic manner.

Initialize your shell environment

The BFD installation script should setup a lot of what BFD needs to run. However, the following must be set. They can be set in your login .rc files:

- **BFDROOT**
  This should point at the top level directory where the Icecube and BFD tools where installed at.

  Example: `>setenv BFDROOT /home/icecube/Icecube_tools/bfd-tools/bfdroot`

- **JAVA_HOME**
  Ask your Sys-Admin where the most current production release of Java is located on the system.

  Example: `>setenv JAVA_HOME /usr/local/java2/sdk/java`

- **CVSROOT**
  Whoever is in charge of software development should be aware of the location. It may also be a remote machine you connect to.

Initialize the BFD workspace

If you installed BFD from the installation script (See the download page), you do not need to perform this step. Otherwise, you will need to run this command once. If you happen to run it multiple times, even after you started to populate your workspace, no damage will occur.

```
> bfd init <Absolute/Path/to/Icecube_tools/>
```

If you do not include an absolute path, bfd will attempt to resolve the BFDROOT env. variable. If this fails, bfd will not initialize.

Before each session with using BFD you will need to source the *setup.sh* or *setup.csh* scripts (depending on the shell you use). This will setup some of the more esoteric paths and environment variables. It is not recommended that sourcing of this file occurs each time a login shell is launched. It may cause uknown behaviour with other applications.

Creating a project

Before you start adding code to your project, it is essential that you prepare your project environment to be used by BFD. This is a one time step, and you will only need to modify the files created here slightly as your project grows.

While BFD is a big time saver with it's automazation of the build process, it has little to no knowledge of the underlying tools that do the builds. Hence its power to handle various language code bases. The following example uses **Ant** for creating a project space for a Java project. Support is forthcoming for C and C++ projects.

Lets start by having Ant create its framework for the project. The following commands will all be done in the top level of the Bfd workspace you just created above.

```
> mkdir foo
> ant -DPROJECT=foo createProject
```

These two commands are essentialy all you need to create a viable project space that BFD knows how to build. This would be boring, as there are no files to compile or targets to build to. Lets create some dummy packages and classes in them.

```
> ant -DPROJECT=foo -DPACKAGE=foo.bar createPackage
> ant -DPROJECT=foo -DPACKAGE=foo.bar -DCLASS=baz createClass
```

If you go into the *foo* directory, you will see the structure created by ant. As you dig down into the **src** directory, you will see the package and class structure that you just created. From here you can get a good idea on how to create the structure of a real project. There are numerous books out there on Software Design, Design Patterns, and Refactoring, that will help you make intelligent decisions on how your project is designed.

Once you have your project framework all in place, you now need to check it into CVS (BFD is designed to use CVS and is a requirement for use).

```
> cd foo
> cvs import -m"Some message" foo someVendorName start
> cd ..
> mv foo foo_save # Or some other safe directory/location
```

Those familiar with CVS should be familiar with this sequence. You are creating a new project and importing the files. Be sure you have the proper permission for the CVS realm you are using.

Checking out a project

Once you have your bfd workspace initialized and created your projects, you can start checking out projects to build.

```
> bfd co|checkout [-a cvs root] [-r release tag] project
```

The -a option allows you to override the existing CVSROOT env. variable. If you do not include this and the CVSROOT variable is undefined, the project will not be checked out.
The -r option allows you to declare which revision level of the project you want checked out. By default bfd will check out the HEAD revision.

Not all projects out there available via CVS are usable by bfd. Specific Ant files are directives need to be included with the CVS project in order for it work with bfd.

Refer to the ICSDE document to understand what needs to be added.

Using the example in the previous step:

```
> bfd checkout foo
```

Delivering a project

Once bfd has successfully built a project, You must 'stamp it' as ready for use by other projects or for inclusion in a *meta-project** inside bfd.

```
> bfd dl|deliver [-b|-n|-j|-r tag] project
```

BFD allows you to define what *type* of delivery you are making. The most common **[-b]** denotes a bug fix. **[-n]** denotes that this delivery has extended the interface of the project (Nothing destructive has been added to the project). **[-j]** is used only in the case where you are delivering a major release. If the project is removing/changing interfaces and may break current implementations, this flag should be used. If you need to deliver a project from a previous production cycle, the **[-r tag]** may be used. This will reconstitute the project from CVS and deliver it.

If a project depends on other bfd controlled projects, attempts to build the primary project will look for the last 'stamped' releases of the required projects to build against. Even if a build of an updated required project was successful, it will not be considered to be built against until it has been stamped.
The same goes for meta-projects. If a meta-project requires other meta-projects to build, it will only consider the last 'stamped' versions of each contained meta-project.

Following our *foo* project:

```
> bfd deliver {-b|-n|-j} foo
```

You will be prompted to OK the delivery tag, if this is what you want, say 'y'. Otherwise you will need to pass in the -r arg with the tag you want to deliver to.

The flags here represent which type of delivery you are making.

> -b: A **b**ug fix. Nothing is done to the interface.
> -n: A mi**n**or fix. The new interface is extended
> -j: A ma**j**or fix. The new interface is not backwards compatible.

Producing a project

More specifically, make a release of this project which has the main task of building the projet.

```
> bfd prod|produce [-r release tag] [-l build dir] project
```

```
> bfd prod|produce [-b build dir] project architecture
```

Where **[-r release tag]** is used to denote which delivered release version of the project you want to produce. A project must be delivered before it can be produced.

As you saw above, a delivery of a project will return its release string. **[-l build dir]** is supplied by BFD to make a sepereate build directory for the project. If it is missing, all the build and produce generated files will be stored in the project directory. This will sometimes make for a messy directory and using the -l is highly recommended.

If your project can be built for multiple architectures, you need not re-deliver the project. BFD allows you to reproduce the project by simply noting the local build dir you initially used with the **[-b build dir]** flag. As you can see, by not using the -l flag initially, you can end up with multiple architecture builds in the same project directory. Yuck! There may also be problems with .h and conf files being overwritten for each arch. Again, we highly recommend that the -l flag is used for all your BFD projects that are produced.

In a nutshell, this will instruct bfd to run the proper Ant command on the project required to make it a deliverable and thus usable by other projects.

Concluding our *foo* project example:

```
> bfd produce -r V00-00-01 -l foo_debug foo
```

Note that the release tag was the default value returned when we delivered it above.

Creating a MetaProject

If you are using BFD, you will understand how large software projects grow more complex with both the amount of code to maintain and their intricate dependencies. Simply using BFD to keep track of each software component in a serial manner doesn't adhere to the hierarchial structure that each component contributes to. To solve this, BFD is able to create and manage what is known as a *MetaProject*.

Consider 5 software components. Each is being developed by a separate effort. There is a main component that utilizes exposed API from 2 others, and those two both require the two remaining components.
For a developer or seasoned software manager, this is easy to visualize. While this group of projects are of equal importance as the other components in the entire project, they play no role beside interacting in this group. By creating a MetaProject within BFD, the user can define both the build and dependency order of these components. At the highest level. BFD sees this MetaProject as simply another project to build. As an added bonus, BFD will also allow you to define a MetaProject as constituent of yet another MetaProject. As you have surmised, this allows you to replicate large, complex build trees by breaking them down into logical [sic] components. Enough verbage, lets look at the process:

In all cases the []'s should be ignored once a value has been inserted.

1. Create the empty meta project repository in CVS from your workspace:

   ```
   > mkdir [META-PROJECT]
   ```

   By convention, meta project names are all upper-case. This helps to distinguish between projects and the meta projects that contain them.

Use the same name of the directory for the name of the new CVS repo. At this time, the repository will be empty of user files.

2. Have Ant create the build framework.

```
> ant -DPROJECT=[META-PROJECT] createProject
```

This will tell Ant to populate the new directory with the build framework that BFD will use.

3. Use BFD to create a new metaproject

```
> bfd meta [META-PROJECT] [list of needed BFD project names]
```

The list of projects must all be existing projects that have been created with BFD by the aforementioned steps.

4. Inform CVS of the new repository.

```
> cd [META-PROJECT]
> cvs import -m"Some meaningful initial message" [META-PROJECT] BFD
start
```

5. Convert metaproject into CVS directory.

```
> cd ..
> rm -rf [META-PROJECT]
> cvs co [META-PROJECT]
```

Building a MetaProject

Once you have a metaproject organized, you will need to deliver it periodically 'deliver' it so it can be built with the latest stable releases of its constituent projects.

In order to do so, you will need to coordinate with the individual project developers and have them Deliver their projects before you can proceede. Once done, each developer will provide you with a BFD release tag (e.g. V01-00-06). Each project will then need to be Checked out in your workspace.

```
> bfd co -r [release tag] [project]
```

Repeat the above for each project with its tag.
The metaproject can now be delivered.

```
> bfd deliver {-b|-n|-j} [meta project]
```

** A project that is solely composed of other interdependent projects*

This page maintained by Martin Stoufer
Page last modified: *Tuesday, 16-Mar-2004 10:30:44 PST*