FCM User Guide

Release: 1-5

Latest content update: 22 January 2010.

Questions regarding this document or permissions to quote from it should be directed to:

IPR Manager
Met Office
FitzRoy Road
Exeter, Devon
EX1 3PB
United Kingdom

Contents

Introduction

This is the User Guide for the *Flexible Configuration Management* system which is known as *FCM*. It is designed to tell you everything you need to know if you want to develop code which has been configured within FCM. In addition it also provides the extra information you will need if you are system manager of a project within FCM.

Note: some hyperlinks in this document lead to pages that are internal to the Met Office, and so will generally not work when this document is viewed externally. Please accept our apologies if this causes you any inconvenience.

This guide consists of the following sections:

System Overview

The FCM system is designed to simplify the task of managing and building source code. It consists of three main components.

Code Management (CM) System

> This system provides facilities for making changes to source code in a controlled and straightforward manner.
>
> Version control is provided by the open source tool Subversion. The source code and its history are stored in a central database which is called the repository. Support for parallel working is provided through the use of branches.
>
> The open source web-based tool Trac allows changes to be examined and documented. It provides an integrated issue tracker, wiki and repository browser.

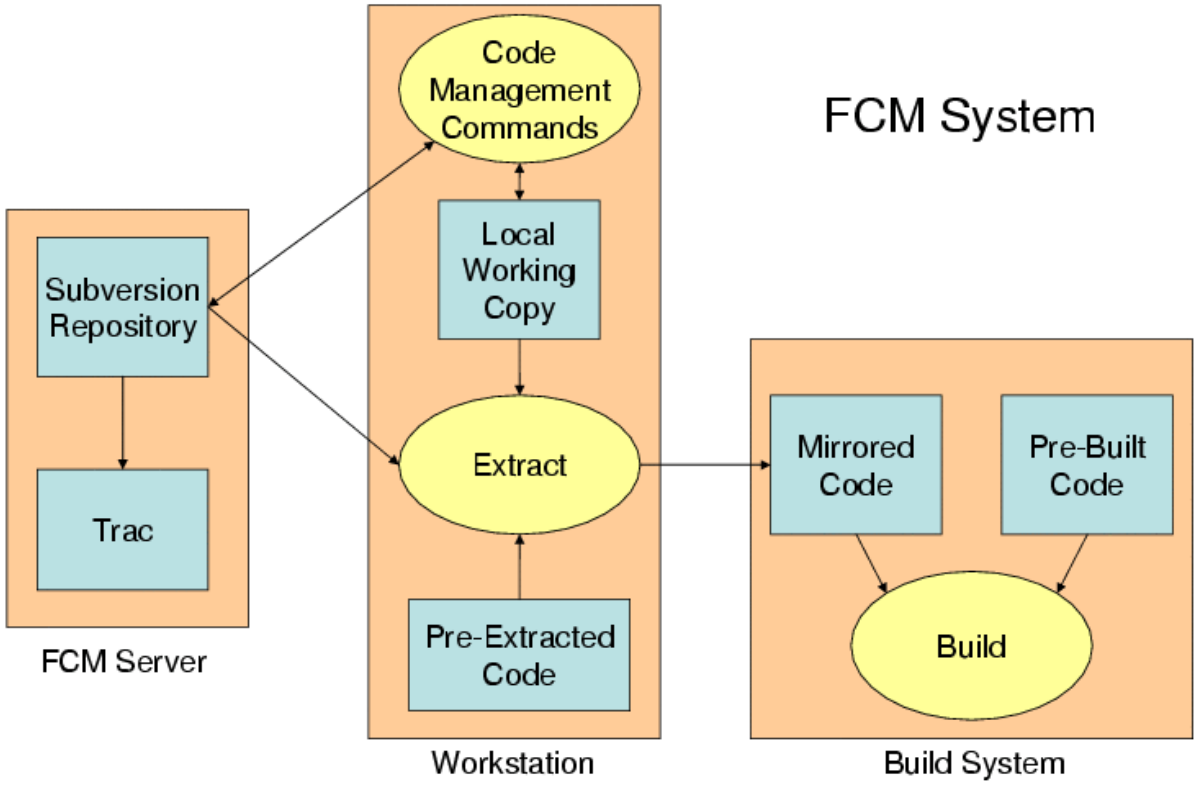
Build System

> This system allows source code to be compiled with a minimal amount of configuration. Compilation time can be minimised through the use of pre-compiled code and by using the parallel make facilities provided by the open source tool *GNU Make*.
>
> It provides a number of powerful features aimed primarily at building Fortran 9x code.

Extract System

> This system provides the interface between the CM and build systems. Code is extracted and presented in a suitable form for the build system. Code can be mirrored to a different build platform if necessary.

The diagram below illustrates how these components fit together.



The following sections discuss these components in more detail.

Code Management System

The CM system is built using a number of open source tools, in particular Subversion and Trac.

Subversion is a modern version control tool with a large and rapidly expanding user base. For a summary of its main features please refer to the What is Subversion? section in the Version Control with Subversion book.

Subversion is a generalised tool which can be used in lots of different ways. This makes some day-to-day tasks more complex than they need be. FCM defines a simplified process and appropriate naming conventions. It then adds a layer on top of Subversion to provide a natural interface which is specifically tailored to this process. Where appropriate it simply makes use of the command line tools provided by Subversion. However, in other cases it provides significant additional functionality, for example:

- By making some assumptions about the repository layout (i.e. by imposing a standard working practise) FCM simplifies the task of creating branches and enforces a standard branch naming convention.
- Having defined working practises and standard log messages allows FCM to greatly simplify the process of merging changes between branches.
- FCM makes use of xxdiff (a graphical merge tool) to simplify the process of resolving any conflicts which result from a merge.
- Code changes can be examined in graphical form using xxdiff FCM also allows you to easily examine the changes made on a branch.
- FCM allows you to check where any particular branch has been used and which version is being used.

FCM also provides a simple GUI which allows easy access to most of the common commands which you will need.

Trac is the other main component of the CM system. It is a powerful web based tool which helps you to manage your software project. It includes the following features:

- A flexible issue tracker which can be used to keep track of bugs, feature requests, etc. Each issue (known as a “ticket” within Trac) can be given a priority and assigned to a particular person. Changes made to your Subversion repository can easily be traced to the relevant ticket. Where appropriate, tickets can be used to record information about who has reviewed each change.
- A “roadmap” feature which helps you to plan and manage project releases. Each ticket can be associated with a particular milestone. Trac can then easily show you what features or fixes went into a particular release or what work remains before a particular milestone is reached.
- A “wiki” which can be used for project documentation.
- A browser for viewing your Subversion repository which allows you to browse the project tree / files and examine revision logs and changesets.
- A timeline view which summarises all the activity on a project (changes to the tickets, wiki pages or the Subversion repository).

Build System

The build system provides the following features:

- Automatic generation of Makefile’s at build time based on a simple configuration file.
- Full dependency analysis at build time.
- Automatic generation of Fortran 9X interface files at build time.
- Support for non-standard source code. You can override automatic dependency and compile rules in order to deal with code which does not conform to the necessary coding standards.
- Flexible control over compiler flags. Changes to compiler flags trigger the appropriate re-compilation.
- Support for Pre-processor directives. Changes to Pre-processor flags trigger the appropriate re-compilation.
- Support for pre-compiled object code to speed compilation time.
- GNU `make` is used to perform the build. Build times on multi-processor systems can be reduced by running parallel processes.

Extract System

The extract system provides the following features:

- Extract code to a directory tree suitable for feeding into the build system. Code can be combined from multiple repositories and branches. Local user code can also be included.
- Either a complete set of source code may be extracted or just a set of changes relative to pre-compiled code.
- A simple configuration file defines what code is required (and what compile options are required). Typically, standard versions of these configuration files are maintained within the repository. Users can then define changes relative to these standard versions.
- If necessary, code can be transferred to a different platform ready for building.

Getting Started

Introduction

This chapter takes a *hands-on* approach to help you set up your FCM session, and familiarise yourself with some of the system's basic concepts and working practices. It is designed to complement other sections of the User Guide.

You may also find it useful to refer to the Annex: Quick reference.

How to set yourself up to run FCM

It is easy to set yourself up to run FCM. Simply follow the steps below:

Setting up your PATH

> The full FCM system is already available for you to use on a Met Office Scientific Desktop. On other Met Office systems (e.g. the Met Office HPC) only the build component is functional.
>
> On a Met Office Scientifc Desktop, FCM is automatically available in the standard *PATH*. On a Met Office HPC, it is also automatically added to your *PATH* by most user interfaces of Met Office scientific application jobs. However, if you intend to run FCM from the command line on the Met Office HPC then you will need to add the following to your `$HOME/.profile` script on that platform:
>
> ```
> . ~fcm/FCM/bin/env.sh  # Add FCM environment
> ```

Setting up the FCM GUI to work with Konqueror

> If you like to use a graphical user interface for some common code management commands, you can set it up for launching from your desktop Konqueror file manager by typing:
>
> ```
> (SHELL PROMPT)$ fcm_setup_konqueror
> ```
>
> See the section on Accessing the GUI from Konqueror for further information.
>
> Note that the first time you issue a command which requires authentication you may need to supply a password or run it from the command line. See the section on GUI Commands for further information.

Configure your editor for Subversion

> When you attempt to create a branch or commit changes to the repository, you will normally be prompted to edit your commit log message using a text editor. The system chooses its editor by searching for a non-empty string through a hierarchy of environment variables in this order: *SVN_EDITOR*, *VISUAL*, and *EDITOR*. If none of these environment variables are set, the default is to use `nedit`. If you set your editor with an environment variable, it is worth bearing in mind that it must be able to run in the foreground. For example, you can add one of the followings in your `$HOME/.kshrc` (ksh) or `$HOME/.bashrc` (bash):

```
# GVim
export SVN_EDITOR='gvim -f'

# Emacs
export SVN_EDITOR=emacs

# NEdit client "nc"
export SVN_EDITOR='nc -wait'
```

Register your user name

> At the Met Office, a small number of projects managed by FCM grant write accesses to their Subversion repositories and Trac ticket create/modify privileges to authorised users only. If you are developing code for such a project, please contact the project's system manager, who will arrange with the FCM team to put your user name in the register.

Configure your e-mail address in Trac

> Trac can be configured to send automatic e-mail notifications to authors of any ticket whenever there are changes to that ticket (and we would expect most systems to be configured in this way). You should check that the settings for your name and e-mail address are correct. To do this you need to go to the Settings page once you are logged into Trac. (Click on **`Settings`** just above the menu bar). Check that your settings are entered correctly. Note: at the Met Office, these settings are set up and maintained automatically, and so you should report any errors to the FCM team.

Configure your web browser

> FCM assumes that you are using Firefox as your default web browser. If you use another web browser such as Mozilla, you should configure it in your `$HOME/.fcm` file. See the section on fcm trac for further information.

Tutorial

Introduction

This tutorial leads you through the basics of using FCM to make changes to your source code, and demonstrates the recommended practices for working with it. A tutorial Subversion repository, with its own Trac system, is available for you to practice for working with the FCM system. You will work through the following activities:

- Create a new ticket
- Launch the GUI
- Create a branch
- Checkout a working copy
- Make changes to files in your working copy
- Commit your changes to the repository
- Test your changes
- Merge changes from the trunk and resolve conflicts
- Review changes
- Commit to the trunk
- Extra activities on the extract and build systems
- Delete your branch
- Final comments

We recommend that you create a work area in your filespace, for example, `$HOME/tutorial/work` for your working copy, and `$HOME/tutorial/build` for your build.

If you have not already done so, you should set up your desktop environment as described above in the How to set yourself up to run FCM section.

It is also worth knowing that the Subversion Book is a great source of reference of Subversion features. In particular, the Fundamental Concepts and Basic Usage chapters are well worth reading.

Create a new ticket

Trac is an integrated web-based issue tracker and wiki system. You will use it to manage and keep track of changes in your project. The issue tracker is called the ticket system. When you want to report a problem or submit a change request, you will create a new ticket. In a typical situation, you and/or your colleagues will make changes to your system in order to resolve the problem or change request, and you will monitor these changes via the ticket.

After completing this sub-section, you will learn how to:

- launch a Trac system,
- create a new Trac ticket,
- search for a Trac ticket, and
- accept a Trac ticket.

Further reading:

- System Overview
- Code Management System > Basic Command Line Usage
- Code Management System > Using Trac
- FCM Command Reference > fcm trac

Launch a Trac system

To launch the Trac system for the tutorial: type and **`Enter`** the following command:

```
(SHELL PROMPT)$ fcm trac fcm:tutorial
```

This is probably the first time you have used the `fcm` command. The command has the general syntax:

```
fcm <sub-command> [<options...>] <arguments>
```

For example, if you type `fcm help`, it will display a listing of what sub-commands are available, and if you type `fcm help <sub-command>`, it will display help for that particular sub-command.

The `trac` sub-command launches the corresponding Trac system browser for a Subversion URL specified in your argument. In this case, we are asking it to display the Trac system browser for the tutorial. The argument `fcm:tutorial` is a FCM URL keyword and will be expanded by FCM into a real Subversion URL (e.g. `svn://fcm1/tutorial_svn/tutorial`). You are encouraged to use FCM URL keywords throughout the tutorial, as it will save you a lot of typing.

Note: Although we use the Trac system as a browser for a Subversion repository, they do not interact in any other ways. Having access to a Trac system does not guarantee the same privilege to a Subversion repository. In particular, you should note the differences between the URLs of a Subversion repository path and its equivalence in a Trac browser.

There are other ways to launch the Trac system for a project. If you know its URL, you can launch the Trac system by entering it in the address box of your favourite browser. If you often access a Trac system for a particular project, you should bookmark it in your favourite browser.

Create a new Trac ticket

Click on **`Login`** just above the menu bar, enter your Unix/Linux user ID as your user name and leave the password empty. Then click on **`OK`** to proceed.

Once you have logged in, the **`New Ticket`** link will become available on the menu bar. Click on it to display a new ticket form, where you can enter details about your problem or change request. In the tutorial, it does not matter what you enter, but you should feel free to play around with wiki formatting when entering the *Full description*. (Click on **`WikiFormatting`** to see how you can use it.) For example:

- Short summary:

```
Tutorial to change repository files and resolve conflicts with the trunk
```

- Full description:

```
In this tutorial, I shall:
 1. try out the FCM GUI and its functions
 2. play with WikiFormatting in Trac tickets
 3. create a branch and checkout a working copy
 4. make changes to files in it
 5. commit my changes and assign the ticket for review
 6. record the review and assign the ticket back to the author
 7. merge in the trunk, and resolve any conflicts
 8. merge my changes back to the trunk
 9. close the ticket
 10. delete my branch
```

- Feel free to select an option you desire for each of the other ticket properties: Type, Component, Priority, Version and Milestone.

At the bottom of the page, click the **`Preview`** button to see what the description would look like. When you are happy, click the **`Submit changes`** button. Trac will create the new ticket and return it in a state where you can append to it.

When the ticket is created, you should get an automatic e-mail notication from the Trac system. In real life, depending on the setting, the owner of your Trac system may also get a similar e-mail notification. It is worth noting that each time the ticket is modified, the Trac system will send out an e-mail notification to you (the reporter) and anyone who modified the ticket subsequently.

Search for a Trac ticket

You should remember the number of your new ticket, as you will have to revisit it later.

In real work, it is often not practical to have to remember the numbers of all the tickets you have created. Trac provides a powerful custom query for searching a ticket. You can search for the ticket you have just created by clicking the **`View Tickets`** link. Feel free to play with the custom query tool. Add or remove filters and try grouping your results by different categories.

In addition, you can search your ticket using the keyword **`Search`** utility at the top right hand corner of each Trac page. (If you enter **`#<number>`** in the search box, it will take you directly to that ticket.) In the tutorial, however, it may be easiest if you simply leave the tutorial Trac system open, so that you do not have to login again when you come back to your ticket.

Start work on your Trac ticket

The status of the ticket is *new*. When you start working on a problem reported in a ticket it is good practice to change the status to *in_progress* to indicate that you are working on it. For the purpose of the tutorial, however, this is entirely optional since you know you will be doing all the work any way.

To start work on a ticket, click on **`start work`** in the *Action* box at the bottom of the page, and then click on **`Submit changes`**.

Launch the GUI

The FCM GUI is a basic graphical wrapper for some of the common code management commands. Most examples in this tutorial can be done via the GUI. You can skip this section and use only the command line if that is what you prefer. Where appropriate, usage examples will be given for both the command line and the FCM GUI.

After completing this sub-section, you will learn how to:

- launch the GUI from the command line, and
- launch the GUI from Konqueror.

Further reading:

- Code Management System > Using the GUI
- FCM Command Reference > fcm gui

Launch the GUI from the command line

You can launch the GUI from the command line. Change directory to your work area and then type `fcm gui`.

Launch the GUI from Konqueror

Alternatively, open Konqueror and navigate to your work area. Right-click in the file-manager window to bring up a menu. Select **`Open with > FCM GUI`**.

Create a branch

You create a branch by making a copy of your project at a particular revision. Most often, this will be a particular revision of the trunk, i.e. the main branch/development line in your project. A branch resides in the repository. It allows you to work in parallel with your colleagues without affecting one another, while keeping your changes under version control.

After completing this sub-section, you will learn how to:

- create a branch in a Subversion repository, and
- update a ticket with a link to a branch.

Further reading:

- Code Management System > Branching & Merging > Creating Branches
- Code Management Working Practices > Branching & Merging
- FCM Command Reference > fcm branch

Create a branch in a Subversion repository

Important note: please ensure that your branch is created from revision 1 of the trunk here, or the tutorial on merge will fail to work later.

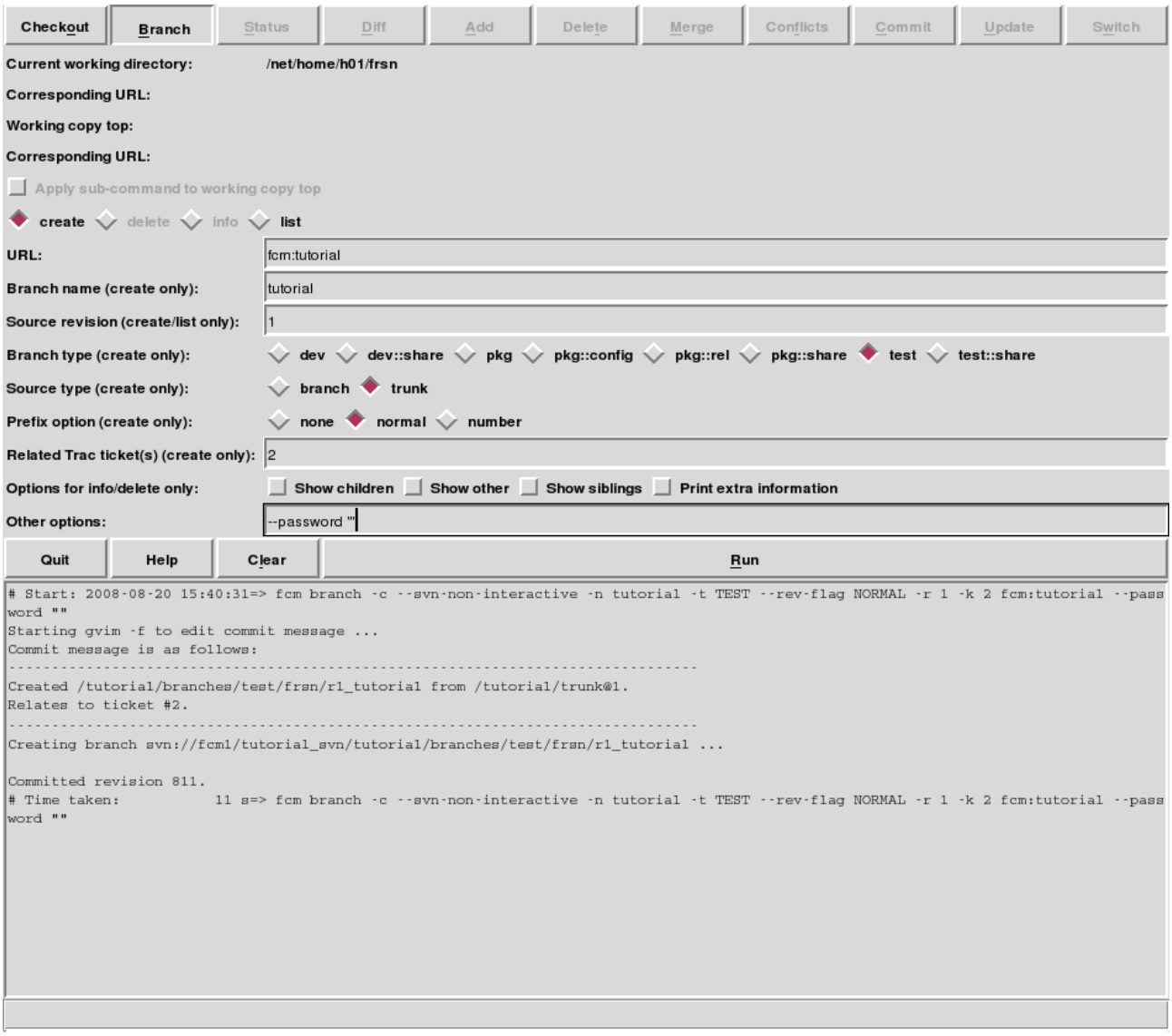
Command line: issue the `fcm branch --create` (or simply `fcm br -c`) command. E.g.:

```
(SHELL PROMPT)$ fcm br -c -n tutorial -r 1 --type test -k 2 fcm:tutorial
```

FCM GUI: click on **`Branch`** on the top menu bar of the GUI. Check the **`create`** radio button and configure your branch details as follow:

- Enter `fcm:tutorial` for the URL of the tutorial repository.
- Enter the short branch name. This must contain only alpha-numeric characters and/or underscores, e.g. `tutorial`.
- Enter **`1`** for the source revision. The trunk was last changed at revision 2. To facilitate the generation of conflicts when you merge with it, you will branch from revision 1 of the trunk.
- Set the branch type. In this case click on **`test`**. This will ensure the branch you create is a user test branch.
- Leave the source type as **`trunk`** and the prefix option as **`normal`**.
- Enter the related Trac ticket number for the ticket you created earlier.
- As this is probably the first time you access the tutorial repository, you should also enter **`--password ""`** (i.e. `--password` followed by a pair of quotes) in the **`Other Options`** entry box.

Click on **`Run`** when you are ready.



You will be prompted to edit the message log file. A standard template is automatically supplied for the commit. However, if you want to add extra comment for the branch, please do so **above** the line that says `--Add your commit message ABOVE - do not alter this line or those below--`. When you are ready, save your change and exit the editor. Answer **`Yes`** when you are prompted to go ahead and create the branch.

Note: If you are on the command line, the Subversion command will prompt you for a password the first time you access a repository. The password will normally be cached by the client, and you will not have to specify a password on subsequent access. If you are using the GUI, please refer to the section on GUI Commands in the next chapter for further information.

When creating branches for the first time, you will notice that FCM will create and commit any missing sub-directories it needs to set up your branch inside the repository, before creating your branch and commiting it.

Take a note of the revision number the branch was created at, and its branch name. (The revision number is the number following the last output that says "Committed revision". In the example above, the branch created at `[811]` is called `branches/test/frsn/r1_tutorial`, which is a branch of the `tutorial` project in the `svn://fcm1/tutorial_svn` repository.)

Update your ticket with a link to your branch

If you wish, you can update your ticket with details of the branch. Note that this step is entirely optional. It is useful for developments which will take a long time to complete. For short lived branches, this step is probably unnecessary.

In the ticket you have created, refer to the revision number in the **`Add/Change`** box, for example:

```
Created the branch [source:tutorial/branches/test/frsn/r1_tutorial@811] at [811].
```

Note:

- `[source:tutorial/branches/test/frsn/r1_tutorial@811]` is a Trac wiki link. In this syntax, you do not have to put in the root URL, (e.g. `svn://fcm1/tutorial_svn/`), but you should specify your branch using the project name (`tutorial`), the branch name (`branches/test/frsn/r1_tutorial`), and a revision number. Trac will translate this into a link to that branch.
- Trac will translate the syntax `[<number>]` into a link to the numbered changeset.

Click on **`Preview`** and check that the links work correctly, and on **`Submit changes`** when you are ready.

Checkout a working copy

A Subversion working copy is an ordinary directory tree on your local system, containing a collection of files. It is your private working area in which you can make changes before publishing them back to the repository. You create a working copy by using the checkout command on some subtree of the repository.

After completing this sub-section, you will learn how to:

- checkout a Subversion working copy.

Further reading:

- Code Management System > Basic Concepts
- FCM Command Reference > Other Subversion Commands

Checkout a Subversion working copy

Command line: issue the `fcm checkout` (or simply `fcm co`) command. E.g.:

```
(SHELL PROMPT)$ fcm co fcm:tutorial_br/test/frsn/r1_tutorial
```

FCM GUI: click on **`Checkout`** in the GUI, and enter the URL of your branch, e.g. `fcm:tutorial_br/test/frsn/r1_tutorial`. Note:

- In the example, we have replaced the leading part of the Subversion URL `svn://fcm1/tutorial_svn/tutorial/branches` with the FCM URL keyword `fcm:tutorial_br`. This is mainly to save you from having to type in the full URL. However, you may find it easier to copy-and-paste the full Subversion URL from the output generated when you created the branch.
- If you do not specify a local directory *PATH* in the `checkout` command, it will create a working copy in your current working directory, using the basename of the URL you are checking out. For example, when you checkout the branch you have just created, the command should create the working copy in `$PWD/r1_tutorial`. Make a note of the location of your working copy, in case you forget where you have put it.

- If you do not specify a revision to checkout, it will checkout the HEAD, i.e. the latest, revision.

Click on **Run** - a working copy pointing to your branch will be created. The GUI will automatically change directory to the top of your new working copy.

Example:

```
=> svn co --revision HEAD svn://fcm1/tutorial_svn/tutorial/branches/test/frsn/r1_tutorial
A    r1_tutorial/cfg
A    r1_tutorial/cfg/ext.cfg
A    r1_tutorial/doc
A    r1_tutorial/doc/hello.html
A    r1_tutorial/src
A    r1_tutorial/src/subroutine
A    r1_tutorial/src/subroutine/hello_c.c
A    r1_tutorial/src/subroutine/hello_sub.f90
A    r1_tutorial/src/module
A    r1_tutorial/src/module/hello_constants.f90
A    r1_tutorial/src/program
A    r1_tutorial/src/program/hello.f90
Checked out revision 811.
```

Make changes to files in your working copy

Subversion provides various useful commands to help you monitor your working copy. The most useful ones are "diff", "revert" and "status". You will also find "add", "copy", "delete" and "move" useful when you are rearranging your files and directories.

After completing this sub-section, you will learn how to:

- make and revert changes,
- add and remove files,
- inspect the status of a working copy, and
- display changes in a working copy.

Further reading:

- Code Management System > Basic Command Line Usage
- FCM Command Reference > fcm add, fcm diff, fcm delete, Other Subversion Commands

Make and revert changes

For the later part of the tutorial to work, you must make the following modification:

- Change to the `src/module/` sub-directory in your working copy.
- Edit **hello_constants.f90**, using your favourite editor, and change: `Hello World!` to **`Hello Earthlings!`**. Save your change and exit the editor.

Try the following so that you know how to restore a changed file:

- Change to the `src/subroutine/` directory of your working copy.
- Make a change in file **hello_c.c**, using your favourite editor.
- To see that you have **M**odified this file: *command line*: issue the `fcm status` command; *FCM GUI*: click on **`Status`** and then on **`Run`**
- Run the `revert` command to get the file back unmodified:

```
(SHELL PROMPT)$ fcm revert hello_c.c
```

Add and remove files

You may also want to try the following FCM commands in your `doc/` sub-directory. You can safely make changes here since they will not interfere with your code changes.

- change to the `doc/` directory of your working copy.
- Echo some text into a new file and then run the `add` command, which lets the repository know you're adding a new file at the next commit. For example:

```
(SHELL PROMPT)$ echo 'Some text' >new_file.txt
(SHELL PROMPT)$ fcm add new_file.txt
```

- Make a copy of `hello.html` and remove the original, using the `copy` and `delete` commands. For example:

```
(SHELL PROMPT)$ fcm copy hello.html add.html
(SHELL PROMPT)$ fcm delete hello.html
```

- You can use a simple `move` sub-command for the above `copy` and `delete`.

Inspect the status of a working copy

Command line: issue the `fcm status` (or simply `fcm st`) command.

FCM GUI: click on **`Status`** and then on **`Run`** to see what has changed.

Example:

```
=> svn status
D       doc/hello.html
A       doc/new_file.txt
A  +    doc/add.html
M       src/module/hello_constants.f90
```

This confirms the actions you have taken. You have **D**eleted a file, **A**dded a new file, **A**dded a file with history (**+**) and **M**odified another. It also confirms the action of the `revert` command.

Display changes in a working copy

You can view the changes you have made to your working copy.

Command line: issue the `fcm diff --graphical` (or simply `fcm di -g`) command.

FCM GUI: click on **`Diff`** and then on **`Run`**.

A listing of the files you have changed will be displayed, and a graphical diff tool will open up for each modified file.

Commit your changes to the repository

The change in your working copy remains local until you commit it to the repository where it becomes permanent. If you are planning to make a large number of changes, you are encouraged to commit regularly to your branch at appropriate intervals.

After completing this sub-section, you will learn how to:

- commit your changes, and
- inspect your changes using Trac.

Further reading:

- Code Management System > Basic Command Line Usage
- FCM Command Reference > fcm commit

Commit changes

Command line: issue the `fcm commit` (or simply `fcm ci`) command.

FCM GUI: click on **`Commit`** and then on **`Run`**.

A text editor will appear to allow you to edit the commit message. You must add a commit message to describe your change **above** the line that says `--Add your commit message ABOVE - do not alter this line or those below--`. (A suggestion is given as the highlighted text in the example below.) Your commit will fail if you do not enter a commit message.

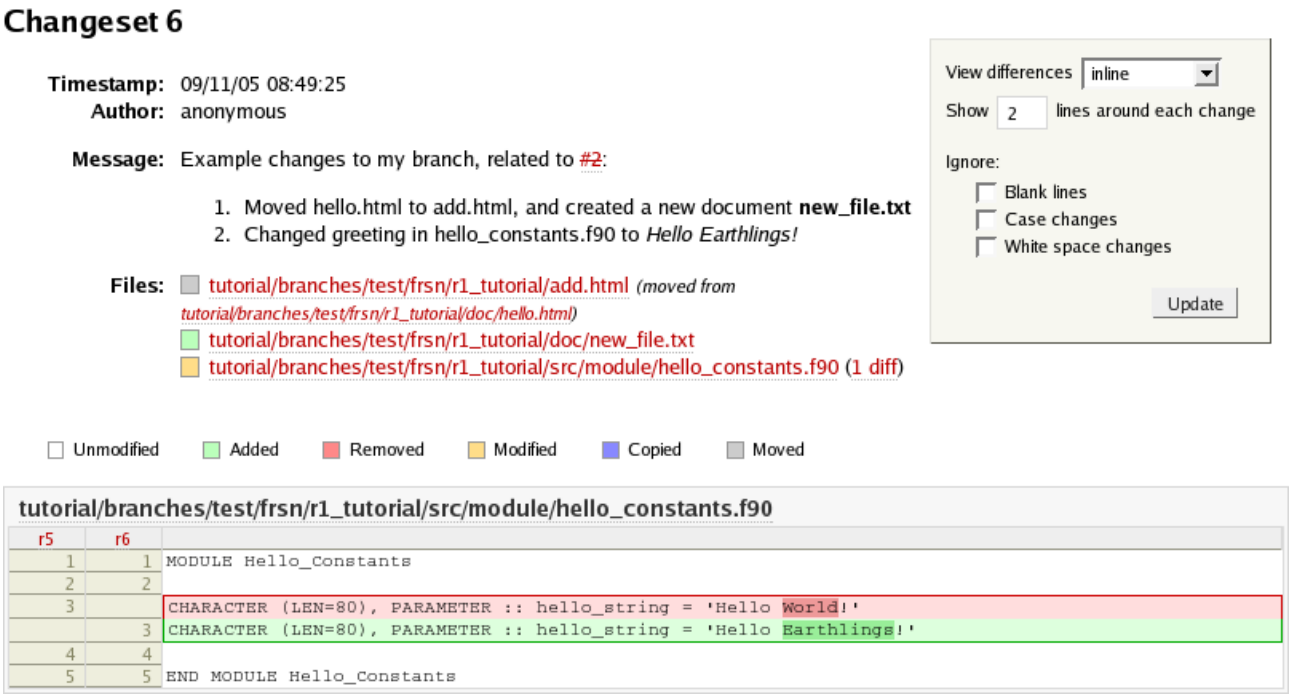
Save your change and exit the editor. Answer **`Yes`** when you are prompted to confirm the commit. For example:

```
Starting nedit to create commit message ...
Change summary:
------------------------------------------------------------------------
[Branch : branches/test/frsn/r1_tutorial]
[Sub-dir: <top>]

D      doc/hello.html
A      doc/new_file.txt
A  +   doc/add.html
M      src/module/hello_constants.f90
------------------------------------------------------------------------
Commit message is as follows:
------------------------------------------------------------------------
In my tutorial branch, for #2:
  1. Moved hello.html to add.html, and created a new document '''new_file.txt'''
  2. Changed greeting in hello_constants.f90 to ''Hello Earthlings!''
------------------------------------------------------------------------
Adding         doc/add.html
Deleting       doc/hello.html
Adding         doc/new_file.txt
Sending        src/module/hello_constants.f90
Transmitting file data ..
Committed revision 812.
=> svn update
At revision 812.
```

Inspect changes using Trac

Click on **`Timeline`** in Trac. Drill down to your changeset and see how it appears. (Alternatively, if you enter `[<number>]` into the search box at the top right, it will take you directly to the numbered changeset.) For example:



Note:

- Wiki Formatting, used in the commit message, has customised the changeset message.
- All your changes are listed.

Test your changes

You should test the changes in your branch before asking a colleague to review them. FCM features a build system that allows you to build your code easily. As your changes may be located in a repository branch and/or a working copy, you should work with the extract system to extract the correct code to build. The extract system allows you to extract code from the repository, combining changes in different branches and your working copy, before generating a configuration file and a suitable source tree for feeding into the build system.

In this sub-section of the tutorial, you will be shown how to extract and build the code from your branch. (There are some extra activities on the extract and build systems in a later sub-section of the tutorial should you want to explore the extract and build systems in more depth.) In the example here, the extract and build systems will be shown to you in their simplest form. In real life, the managers of the systems you are developing code for will provide you with more information on how to extract and build their systems.

After completing this sub-section, you will learn how to:

- set up a simple extract configuration file, and
- perform simple extracts and builds.

Further reading:

- The Extract System
- The Build System

You should extract and build your code in a different directory to your working copy. For example, you may want to create a sub-directory `$HOME/tutorial/build/` and change to it:

```
(SHELL PROMPT)$ mkdir -p $HOME/tutorial/build
(SHELL PROMPT)$ cd $HOME/tutorial/build
```

Set up an extract configuration file

To set up an extract configuration file from scratch, launch your favourite editor and add the following lines:

```
# Extract configuration, format version 1.0
cfg::type              ext
cfg::version           1.0

# Extract destination root directory
dest                   $HERE

# Location of the source in the "r1_tutorial" branch
repos::tutorial::base  fcm:tutorial_br/test/$LOGNAME/r1_tutorial

# Extract all sub-directories under the above URL
expsrc::tutorial::base src

# Fortran and C compiler commands respectively
# You may need to redefine these for different platforms
bld::tool::fc          ifort
bld::tool::cc          gcc
```

Note:

- The `dest` declaration is set to `$HERE` the directory containing the extract configuration file. If you decide to extract to a different directory, you should modify its value. Please do not put your build in the same location as your working copy.
- Similarly, the `repos::tutorial::base` declaration is set to `fcm:tutorial_br/test/$LOGNAME/r1_tutorial`. If you have named your branch differently, you should modify its value.

Save the file as `ext.cfg` and exit your editor.

Perform an extract and a build

Issue the command `fcm extract` and you should get an output similar to the following:

```
(SHELL PROMPT)$ fcm extract
Extract system started on Tue Apr 24 13:54:06 2007.
->Parse configuration: start
Config file (ext): /net/home/h01/frsn/tutorial/build/ext.cfg
->Parse configuration: 0 second
->Setup destination: start
Destination: /net/home/h01/frsn/tutorial/build
->Setup destination: 0 second
->Extract: start
Destination status summary:
  No of files added: 4
Source status summary:
  No of files from the base: 4
->Extract: 1 second
->TOTAL : 1 second
Extract finished on Tue Apr 24 13:54:07 2007.
```

If nothing goes wrong, you should end up with the sub-direcories `src/` and `cfg/` in your working directory. The `src/` contains a source tree to be built, and `cfg/` should contain two configuration files: `ext.cfg` and `bld.cfg`. The former is an expanded version of your extract configuration file and the latter is a build configuration file. You can now build your code by running the `fcm build`

command:

```
(SHELL PROMPT)$ fcm build 2>err
Build command started on Fri Oct 14 09:15:38 2005.
->Parse configuration: start
Config file (bld): /net/home/h01/frsn/tutorial/build/cfg/bld.cfg
->Parse configuration: 0 second
->Setup destination: start
Destination: /net/home/h01/frsn/tutorial/build
->Setup destination: 0 second
->Setup build: start
->Setup build: 0 second
->Pre-process : start
->Pre-process : 0 second
->Scan dependency : start
No. of files scanned for dependency: 4
/net/home/h01/frsn/tutorial/build/Makefile: updated
->Scan dependency : 1 second
->Generate interface : start
No. of generated Fortran interface: 1
->Generate interface : 0 second
->Make : start
ifort -o hello_constants.o -I/home/h01/frsn/tutorial/build/inc -c
  /home/h01/frsn/tutorial/build/src/tutorial/src/module/hello_constants.f90
ifort -o hello.o -I/home/h01/frsn/tutorial/build/inc -c
  /home/h01/frsn/tutorial/build/src/tutorial/src/program/hello.f90
ifort -o hello_sub.o -I/home/h01/frsn/tutorial/build/inc -c
  /home/h01/frsn/tutorial/build/src/tutorial/src/subroutine/hello_sub.f90
gcc -o hello_c.o -I/home/h01/frsn/tutorial/build/inc -c
  /home/h01/frsn/tutorial/build/src/tutorial/src/subroutine/hello_c.c
ifort -o hello.exe /home/h01/frsn/tutorial/build/obj/hello.o
  -L/home/h01/frsn/tutorial/build/lib -l__fcm__hello
->Make : 3 seconds
->TOTAL : 4 second
Build command finished on Fri Oct 14 09:15:42 2005.
```

The executable you have built is `hello.exe`, which is located in the `bin/` sub-directory. You can test your executable by running it. You should get an output similar to the following:

```
(SHELL PROMPT)$ bin/hello.exe
Hello: Hello Earthlings!
Hello_Sub: Hello Earthlings!
Hello_Sub: maximum integer: 2147483647
Hello_C: Hello World!
```

Merge changes from the trunk and resolve conflicts

Your branch is normally isolated from other development lines in your project. However, at some point during your development, you may need to merge your changes with those of your colleagues. In some cases, it is desirable to merge changes regularly from the trunk to keep your branch up to date with the latest development. The automatic merge provided by FCM allows you to do this easily.

A merge results in a conflict if changes being applied to a file overlap. FCM uses a graphical merge tool to help you resolve conflicts in text files.

After completing this sub-section, you will learn how to:

- merge changes from the trunk into your working copy, and
- resolve conflicts in your working copy.

Further reading:

- Code Management System > Basic Command Line Usage > Resolving Conflicts
- Code Management System > Branching & Merging
- Code Management Working Practices > Branching & Merging
- FCM Command Reference > fcm conflicts
- FCM Command Reference > fcm merge

Merge changes from the trunk into a working copy

Perform the merge in your working copy.

Command line: issue the `fcm merge` command. E.g.

```
(SHELL PROMPT)$ fcm merge trunk
```

FCM GUI: click on **`Merge`**. Enter **`trunk`** into the Source entry box and click on **`Run`** to proceed.

If there is more than one revision of the source that you can merge with, you will be prompted for the revision number you wish to merge from. You will not be prompted in this case, because there is only one revision of the source that you can merge with.

Answer **`Yes`** when you are prompted to go ahead with the merge.

Example:

```
Available Merges From /tutorial/trunk: 2
About to merge in changes from tutorial/trunk@2 compared with tutorial/trunk@1
This merge will result in the following changes:
------------------------------------------------------------------------
U    src/subroutine/hello_c.c
C    src/module/hello_constants.f90
------------------------------------------------------------------------
Performing merge ...
U    src/subroutine/hello_c.c
C    src/module/hello_constants.f90
```

Resolve conflicts in a working copy

The `C` status indicates that the file you changed is now in conflict. If you run `status`, you will see extra files created by the merge, which enable you to resolve the conflict using the 3-way difference tool xxdiff:

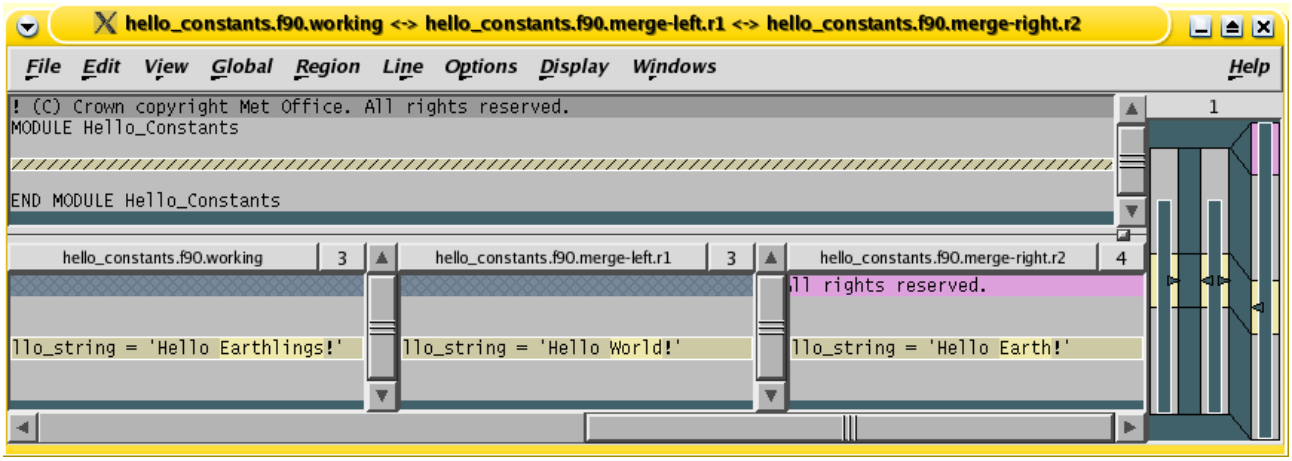
```
=> svn status
M       src/subroutine/hello_c.c
?       src/module/hello_constants.f90.merge-left.r1
?       src/module/hello_constants.f90.merge-right.r2
?       src/module/hello_constants.f90.working
C       src/module/hello_constants.f90
```

You will now have to resolve the conflicts.

Command line: issue the `fcm conflicts` (or simply `fcm cf`) command.

FCM GUI: click on **`Conflicts`** and then on **`Run`**.

The xxdiff program comes into play:



See the sub-section on resolving conflicts, or the *xxdiff User's Manual* (click on *Help*) to guide you through this process. (If you do not want to learn how to use `xxdiff` now, you can just click on the highlighted line in the left hand column, and select **`Exit with MERGE`** from the *File* menu. This saves the file you are merging in as the result of the merge, i.e. you have *merged* the changes).

On resolving the conflict, you will be asked to run `svn resolved`. Answer **`Yes`**.

If you now run `status`, you will notice that these extra conflict files have disappeared.

Example:

```
Conflicts in file: src/module/hello_constants.f90
All merge conflicts resolved
Resolved conflicted state of 'hello_constants.f90'
=> svn status
M       src/subroutine/hello_c.c
M       src/module/hello_constants.f90
```

It is important to remember that the `merge` command only applies changes to your working copy. Therefore, you must now commit the change in order for it to become permanent in the repository. Similar to other changes, it is a good practice to use `diff` to inspect the changes before committing.

When you run `commit`, you will be prompted to edit the commit log as usual. However, you may notice that a standard template is already provided for you by the `merge` command. In most cases, the standard message should be sufficient. However, if you want to add extra comment to the commit, please do so **above** the line that says `--Add your commit message ABOVE - do not alter this line or those below--`. This is useful, for example, if there were significant issues addressed in the merge.

Review changes

For the purpose of this tutorial, we assume that your changes are complete, have been tested and committed to the repository, and are now ready for review. You should assign the ticket to the reviewer and inform him/her where to find the changes you wish him/her to review. The reviewer will record any issues in the ticket, perhaps linking to other documents as required. Once completed, he/she will record the outcome in the ticket and assign it back to the you.

After completing this sub-section, you will learn how to:

- display changes in a branch, and
- re-assign a ticket.

Further reading:

- Code Management System > Branching & Merging > Getting Information About Branches
- Code Management System > Using Trac
- Code Management Working Practices > Using Tickets
- FCM Command Reference > fcm diff

Display changes in a branch

Before you ask someone to review your code, it is often a good idea to have a look at the changes one more time. To view the changes in a branch, you can look at all the changes relative to its base.

Command line: issue the `fcm diff --branch --graphical` (or simply `fcm di -b -g`) command.

FCM GUI: click on **`Diff`**. Check the box **`Show differences relative to the base of the branch`**, and click on **`Run`**.

You should be presented with the differences between the branch and the trunk (since the last merge).

Note: you can also use the `--trac` (`-t`) option instead of `--graphical` (`-g`) to view the changes in a branch using Trac rather than using a graphical diff tool.

Command line: issue the `fcm diff --branch --trac` (or simply `fcm di -b -t`) command.

FCM GUI: click on **`Diff`**. Check the box **`Show differences relative to the base of the branch`**, and select to display diff in Trac. Click on **`Run`**.

Take note of the Trac URL for displaying the differences. The part that begins with `diff:` is of particular interest to you, as it is a Trac link that can be inserted into a Trac wiki/ticket. In the above example, the Trac link would look like:
`diff:/tutorial/trunk@2///tutorial/branches/test/frsn/r1_tutorial@813`.

Re-assign a ticket to a reviewer

Back in your ticket, add an appropriate comment showing where to find your changes, in the *Add/Change* box. Include a link to your branch and a diff link (see above) in the comment. For example:

```
The [log:tutorial/branches/test/frsn/r1_tutorial@811:813] branch proposes
changes to the greeting in hello_constants.f90. It also contains some new
documents. See
[diff:/tutorial/trunk@2///tutorial/branches/test/frsn/r1_tutorial@813] for the
changes.

Fred, could you review the change, please?
```

Note: the syntax `[log:tutorial/branches/test/frsn/r1_tutorial@811:813]` will be translated by Trac into a link to the revision log browser to display the log between revision 811 and 813 of the `branches/test/frsn/r1_tutorial` branch in the `tutorial` project; and the syntax `[diff:/tutorial/trunk@2///tutorial/branches/test/frsn/r1_tutorial@813]` will be translated into a link to display the differences between the trunk at revision 2 and the branch at revision 813. Click on **`Preview`** and check that the links work correctly.

To re-assign a ticket to your reviewer, click on the **`reassign to`** button in the *Action* box section and enter the reviewer's User ID.

When you are ready, click on **`Submit changes`**.

Reassign the ticket back to the author

For the purpose of this tutorial, you will act as the reviewer of the changes you have made. Following the review, you should record its outcome and re-assign the ticket back to the author. Enter the comment **`No issues were found during the review`**. Click on the **`reassign to`** button in the *Action* box section, and enter your guest account name. Click on **`Submit changes`** when you are ready.

Commit to the trunk

Your changes in the branch have been tested and reviewed. It is now time to merge and commit it to the trunk. Once you have committed your change, you will close your ticket to complete the work cycle.

After completing this sub-section, you will learn how to:

- switch a working copy,
- merge and commit your changes into the trunk, and
- close a ticket

Further reading:

- Code Management System > Branching & Merging
- Code Management Working Practices > Branching & Merging
- FCM Command Reference > fcm switch

Switch a working copy to point to the trunk

Command line: issue the `fcm switch` (or simply `fcm sw`) command. E.g.:

```
(SHELL PROMPT)$ fcm sw trunk
```

FCM GUI: click on **`Switch`**. Enter **`trunk`** as the URL and then click on **`Run`**.

To check that your working copy is pointing to the trunk, you should: *command line*: issue the `fcm info` command; *FCM GUI*: inspect the corresponding URL of your working copy.

Merge and commit your changes into the trunk

Command line: issue the `fcm merge` command. E.g.

```
(SHELL PROMPT)$ fcm merge branches/test/frsn/r1_tutorial
```

FCM GUI: click on **`Merge`**. Enter the name of your branch in the Source entry box, (e.g. `branches/test/frsn/r1_tutorial`). Click on **`Run`** to proceed.

Example:

```
Available Merges From /tutorial/branches/test/frsn/r1_tutorial: 813 812
About to merge in changes from /tutorial/branches/test/frsn/r1_tutorial@813
  compared with /tutorial/trunk@2
This merge will result in the following changes:
------------------------------------------------------------------------
```

```
D    doc/hello.html
A    doc/new_file.txt
U    src/module/hello_constants.f90
A    add.html
------------------------------------------------------------------------
Performing merge ...
D    doc/hello.html
A    doc/new_file.txt
U    src/module/hello_constants.f90
A    add.html
```

Since there is more than one revision available for merging, you will be prompted for the revision number you wish to merge from. The default is the last changed revision of your branch. which is the revision you want to merge with, so you should just proceed with the default.

Answer **`Yes`** when you are prompted to go ahead with the merge.

Since we merged in the latest changes from the trunk into the branch, there should be no conflicts from this merge.

Once again, please remember that the merge command only changes your working copy. You need to commit the change before it becomes permanent in the repository. Before you commit to the trunk, however, it is often sensible to have a last look at what you are going to change using the `diff` command.

Note: We have set up the repository to prevent any commits to the trunk to preserve the tutorial for other users, so your commit to the trunk will fail. However, you should try doing it any way to complete the exercise.

Command line: issue the `fcm commit` (or simply `fcm ci`) command.

FCM GUI: click on **`Commit`** and then on **`Run`**.

A text editor will appear to allow you to edit the commit message. You must add a commit message to describe your change **above** the line that says `--Add your commit message ABOVE - do not alter this line or those below--`. Since you are going to commit changes to the trunk, you should provide a useful message, including a link to your ticket. For example:

```
For #2: complete the tutorial:
  1. Changed greeting in hello_constants.f90 to "Hello Earthlings!"
  2. Moved hello.html to add.html, and created a new document ''new_file.txt''
```

When you are ready, save your change and exit the editor.

As we have said before, the command will fail when you try to proceed with the commit.

Close your ticket

As you have completed your work, you should now update and close your ticket. In real life, you will typically include a closing comment with an appropriate Trac wiki link to the changeset in the trunk that fixes the ticket.

Since you cannot commit to the trunk in the tutorial, you can include a Trac link to the latest changeset in your branch. For example, you can put `Fixed at changeset [813].` in the comment. To mark the ticket as *fixed*, move down to the *Action* box section, click on **`resolve as`** and choose **`fixed`**. Use **`Preview`** to ensure that your links work correctly. When you are happy, click on **`Submit changes`**.

Extra activities on the extract and build systems

The extract and build systems are very flexible. If you have time, you may want to explore their uses in more depth.

After completing this sub-section, you will learn how to:

- extract from a working copy,
- change a compiler flag, and
- extract from a particular branch and/or revision from the repository.

Further reading:

- The Extract System
- The Build System

Extract from a working copy

Modify the source files in your working copy and commit the changes back to your branch in the repository. Re-run `fcm extract` and `fcm build` and see the results of the changes. [The file(s) you have changed should be updated by *extract*, and *build* should only re-build the necessary code.]

In fact, you can test changes in your working copy directly using a similar extract and build mechanism. In such case, you need to modify the REPOS declaration. For example:

```
repos::tutorial::base  $HOME/fcm/work/r1_tutorial
```

Change a compiler flag

Modify the compiler flags, and re-run `fcm extract` and `fcm build` and see the results of the changes. To modify the compiler flags, edit your extract configuration file, and add the declarations for changing compiler flags. For example:

```
# Declare extra options for Fortran compiler
bld::tool::fflags  -i8 -O3
```

For further information on how to set your compiler flags, please refer to the sub-section on Setting the compiler flags.

Extract from a particular branch and/or revision

Try extracting from an earlier revision of your branch. Suppose the HEAD of your branch is revision 813, and the branch was created at an earlier revision. You can extract your branch at, say, revision 811 by adding a declaration in your extract configuration file:

```
revision::tutorial::base  811
```

You can also try extracting from the trunk. In such case, you will need to modify the REPOS declaration in your extract configuration file. For example:

```
repos::tutorial::base  fcm:tutorial_tr/src

# Extract with and without the following line and note the difference!
revision::tutorial::base  1
```

Delete your branch

You should remove your branch when it is no longer required. When you remove it, it becomes invisible from the HEAD revision, but will continue to exist in the repository, should you want to refer to it in the future.

After completing this sub-section, you will learn how to:

- list branches owned by you, and
- delete a branch.

Further reading:

- Code Management System > Branching & Merging > Listing Branches Created by You or Other Users
- Code Management System > Branching & Merging > Deleting Branches

List branches owned by you

If you forget what your branch is called and/or what other branches you have created, you can get a listing of all the branches you have created in a project.

Command line: issue the `fcm branch --list` (or simply `fcm br -l`) command

FCM GUI: click on **`Branch`**, and then on the **`list`** radio button. Click on **`Run`**.

Delete a branch

Switch your working copy to point back to your branch. Before you do so, revert any changes you have made in the working copy by issuing the `fcm revert -R .` command. If a `#commit_message#` file exists, remove it by issuing the `rm '#commit_message#'` command.

Command line: issue the `fcm switch <URL>` (or simply `fcm sw <URL>`) command.

FCM GUI: click on **`Switch`**. Enter the name of your branch as the URL and click on **`Run`** to proceed.

You can continue your work in the branch if you wish, but once you have finished all the work, you should delete it. *Command line*: issue the `fcm branch --delete` (or simply `fcm br -d`) command. *FCM GUI*: click on **`Branch`** in the GUI. Check the **`delete`** radio button, and click **`Run`** to proceed.

Example:

```
URL: svn://fcm1/tutorial_svn/tutorial/branches/test/frsn/r1_tutorial
Repository Root: svn://fcm1/tutorial_svn
Repository UUID: cb858ce8-0f05-0410-9e64-efa98b760b62
Revision: 813
Node Kind: directory
Last Changed Author:
Last Changed Rev: 813
Last Changed Date: 2005-11-09 09:11:57 +0000 (Wed, 09 Nov 2005)
------------------------------------------------------------------------
Branch Create Rev: 811
Branch Create Date: 2005-11-09 08:34:22 +0000 (Wed, 09 Nov 2005)
Branch Parent: svn://fcm1/tutorial_svn/tutorial/trunk@1
------------------------------------------------------------------------
Last Merge From Trunk: /tutorial/branches/test/frsn/r1_tutorial@813
                       /tutorial/trunk@2
```

```
Avail Merges Into Trunk: 813 812
Starting nedit to create commit message ...
Change summary:
------------------------------------------------------------------------
D    svn://fcm1/tutorial_svn/tutorial/branches/test/frsn/r1_tutorial
------------------------------------------------------------------------
Commit message is as follows:
------------------------------------------------------------------------
Deleted tutorial/branches/test/frsn/r1_tutorial.
------------------------------------------------------------------------
Deleting branch
  svn://fcm1/tutorial_svn/tutorial/branches/test/frsn/r1_tutorial ...

Committed revision 8.
```

You will be prompted to edit the commit message file. A standard template is automatically supplied for the commit. However, if you want to add extra comment for the branch, please do so **above** the line that says `--Add your commit message ABOVE - do not alter this line or those below--`. Save your change and exit the editor.

Answer **`Yes`** when you are prompted to go ahead and delete this branch.

Your working copy is now pointing to a branch that no longer exists at the HEAD revision of the repository. If you want to try the tutorial again, you may want to create another branch, and switch your working copy to point to the new branch. Otherwise, you can remove your working copy by issuing a careful `rm -rf` command.

Final comments

We have guided you through the basics of the complete change process, using recommended ways of working. Most of the basic and important commands have been covered by the tutorial. (The exceptions are `fcm log` and `fcm update`, which you may have to use regularly. For information on these commands, please refer to the section on svn log and Update Your Working Copy in the Subversion book.) You should now be in a position to continue with your development work with FCM. However, if at any time you are unsure about any aspect of using FCM, please consult the relevant section of the FCM User Guide.

Feel free to use the tutorial, at any time, for testing out any aspect of the system. You may wish to do this rather than use your own repository and ticket system, to avoid cluttering them with unwanted junk.

Code Management

Using Subversion

One of the key strengths of Subversion is its documentation. Version Control with Subversion (which we'll just refer to as the *Subversion book* from now on) is an excellent book which explains in detail how to use Subversion and also provides a good introduction to all the basic concepts of version control. Rather than trying to write our own explanations (and not doing as good a job) we will simply refer you to the *Subversion book*, where appropriate, for the relevant information.

In general, the approach taken in this section is to make sure that you first understand how to perform a particular action using the Subversion tools and then describe how this differs using FCM.

Basic Concepts

In order to use FCM you need to have a basic understanding of version control. If you're not already familiar with Subversion or CVS then please read the chapter Fundamental Concepts from the *Subversion book*. In particular, make sure that you understand:

- The "Copy-Modify-Merge" approach to file sharing.
- Global Revision Numbers.

Note that this chapter states that "working copies do not always correspond to any single revision in the repository". However, the FCM working practises do not encourage this and the wrapper scripts provided by FCM should ensure that your working copy (a local copy of the repository's files and directories where you can prepare changes) always corresponds to exactly one revision.

CVS users should already be familiar with all the basic concepts. This is not surprising since Subversion was designed as a replacement for CVS and it uses the same development model. However, there are some important differences which may confuse those more familiar with CVS. Fortunately, the appendix in the *Subversion book* Subversion for CVS Users is specifically written for those moving from CVS to Subversion and you should read this if you are a CVS user.

Basic Command Line Usage

Before we discuss the FCM system you need to have a good understanding of how to perform most of the normal day-to-day tasks using Subversion. Therefore, unless you are already familiar with Subversion, please read the chapter Basic Usage from the *Subversion book*.

So, now you have an understanding of how to do basic tasks using Subversion (you did read the *Basic Usage* chapter didn't you?), how is using FCM different? Well, the key thing to remember is that, instead of using the command `svn` you need to use the command `fcm`. The advantages of this are as follows:

- `fcm` implements all of the commands that `svn` does (including all the command abbreviations).
- In some cases `fcm` does very little and basically passes on the command to `svn`.
- In other cases `fcm` has a lot of additional functionality compared with the equivalent `svn` command.
- `fcm` also implements several commands not provided by `svn`.
- `fcm` provides support for URL and revision keywords.
- Most of the additional features and commands are discussed later in this section or in the following sections.

Full details of all the `fcm` commands available are provided in the FCM Command Reference section.

URL & Revision Keywords

URL keywords can be used to specify URLs in `fcm` commands. The syntax is `fcm:<keyword>`. Keywords can be defined globally (see the file `../etc/fcm.cfg` where-ever the `fcm` command has been installed) or individually in a user configuration file `$HOME/.fcm`. See the FCM Command Reference for further details about configuration files.

For example, if you define a keyword in your configuration file as follows:

```
set::url::fcm      svn://fcm1/FCM_svn/FCM
```

then you can abbreviate the URL as in the following examples:

```
# fcm ls svn://fcm1/FCM_svn/FCM
fcm ls fcm:fcm

# fcm ls svn://fcm1/FCM_svn/FCM/trunk
fcm ls fcm:fcm_tr  # OR: fcm ls fcm:fcm-tr

# fcm ls svn://fcm1/FCM_svn/FCM/branches
fcm ls fcm:fcm_br  # OR: fcm ls fcm:fcm-br

# fcm ls svn://fcm1/FCM_svn/FCM/tags
fcm ls fcm:fcm_tg  # OR: fcm ls fcm:fcm-tg
```

Using URL keywords has two advantages.

- They are shorter and easier to remember.
- If the repository needs to be moved then only the keyword definitions need to be updated (although any working copies you have will still need to be *relocated* by issuing a `fcm switch --relocate` command).

In a similar way, revision keywords can be used to specify revision numbers in `fcm` commands. The keyword can be used anywhere a revision number can be used. Each keyword is associated with a URL keyword and can only be used when referring to that repository.

For example, if you define a keyword in your configuration file as follows:

```
set::revision::fcm::vn1.0  112
```

then the following commands are equivalent:

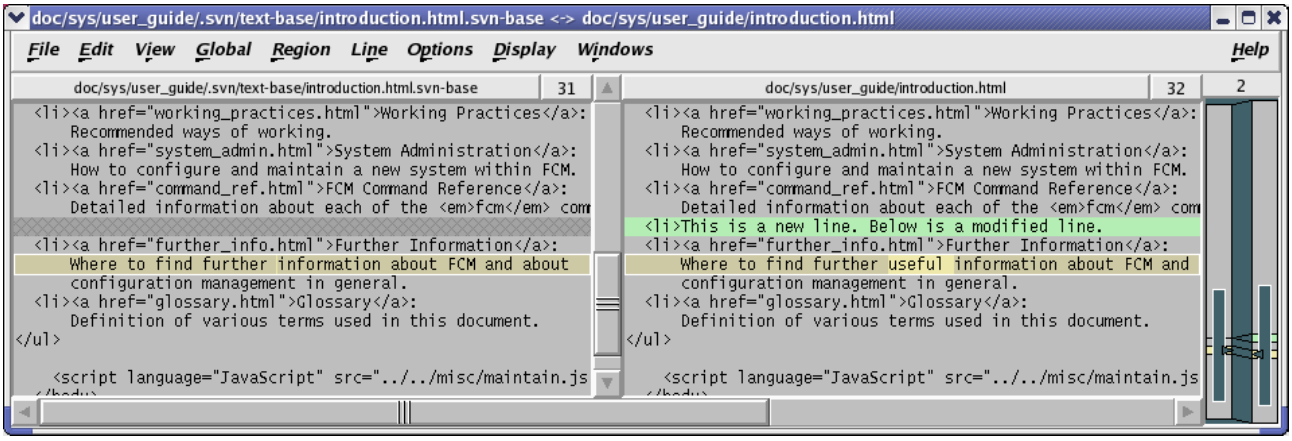
```
fcm log -r 112 svn://fcm1/FCM_svn/trunk
fcm log -r vn1.0 fcm:fcm_tr
```

You can use the `fcm keyword-print` command to print all registered URL keywords. You can also print the URL keyword, its implied keywords and the revision keywords of a particular project. For example, to print the keywords for the `UM` project, you can type `fcm keyword-print fcm:um`.

Examining Changes

Code differences can be displayed graphically using `xxdiff` by using the `--graphical` (or `-g`) option to `fcm diff`. This option can be used in combination with any other options which are accepted by `svn diff`.

An example display from `xxdiff` is shown below.



`xxdiff` 2-way display

Points to note:

- By default `xxdiff` is configured to show horizontal differences. This means that the parts of the line which have changed are highlighted (e.g. the text `useful` is highlighted in the example above).
- The number shown to the right of each file name shows the current line number. The number on the far right is the number of differences found (2 in the example above).
- You may find the following keyboard shortcuts useful.
  - **`N`** - move to the next difference
  - **`P`** - move to the previous difference
  - **`Ctrl-Q`** - exit
- If you want to use another diff tool instead of `xxdiff` to examine changes, you can either set the *FCM_GRAPHIC_DIFF* environment variable or define the `set::tool::graphic_diff` setting in your `$HOME/.fcm` file. For example, to use `tkdiff`, you will do:

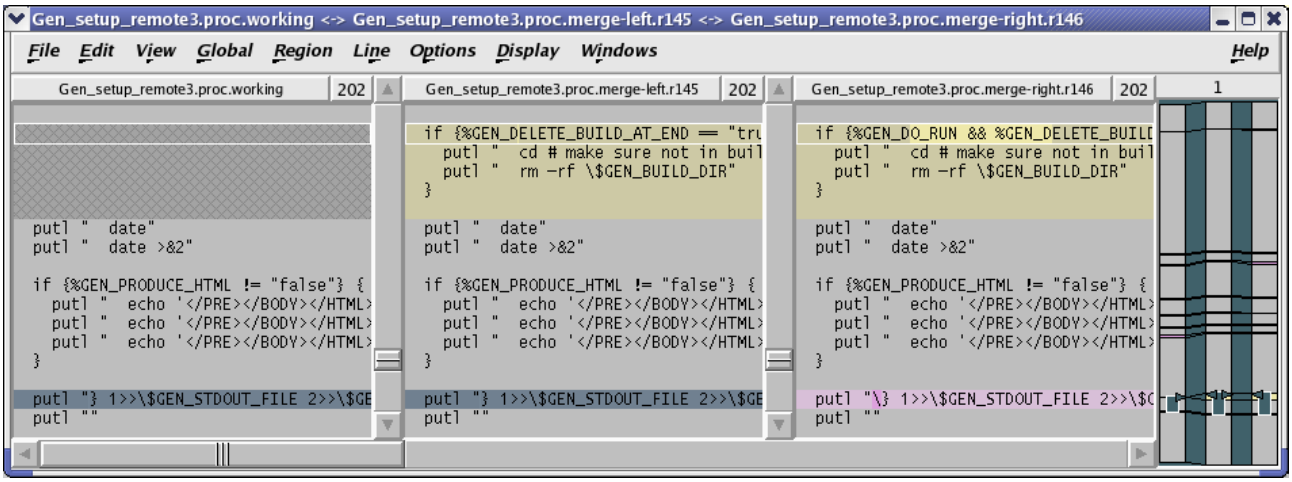
```
# EITHER: in bash/ksh:
(SHELL PROMPT)$ export FCM_GRAPHIC_DIFF=tkdiff

# OR: in your $HOME/.fcm:
set::tool::graphic_diff  tkdiff
```

Resolving Conflicts

Your working copy may contain files *in conflict* as a result of an update or a merge (covered later). Conflicts arise from the situation where two changes being applied to a file *overlap*. For conflicts in text files, the command `fcm conflicts` can be used to help resolve them. (A discussion on binary files is given in the section Working with Binary Files later in this document.) The `fcm conflicts` command calls a graphical merge tool (i.e. `xxdiff` by default) to display a 3-way diff for each of the files in conflict.

An example display from `xxdiff` is shown below.



`xxdiff` 3-way display

Points to note:

- The file in the middle is the common ancestor from the merge. The file on the left is your original file and the file on the right is the file containing the changes which you are merging in.
- `xxdiff` is configured to automatically select regions that would end up being selected by an automatic merge (e.g. there are only changes in one of the files). Any difference *hunks* which cannot be resolved automatically are left *unselected*.
- Before you can save a merged version you need to go through each unselected difference hunk and decide which text you wish to use.
  - Selecting a diff hunk can be carried out by clicking on it with the left mouse button (or refer to the keyboard shortcuts shown under the **`Region`** menu). The colours update to display which side is selected for output. You can select individual lines with the middle mouse button.
  - If you want to select more than one side, you have to invoke the **`Region->Split/swap/join`** command (keyboard shortcut: **`S`**). This will split the current diff hunk so you can select the pieces you want from both sides. Further invocations of this command will cause swapping of the regions, looping through all the different ordering possibilities, and finally joining the regions again (preserving selections where it is possible).
- The number on the far right is the number of unselected difference hunks (1 in the example above). Once this number is 0 then you are ready to save the merged file.
- If you want to see how the merged file will look with the current selections then select **`Windows->Toggle Merged View`** (keyboard shortcut: **`Alt+Y`**). An extra window then appears showing the merged output that updates interactively as you make selections.
- You may find the following keyboard shortcuts useful.
  - **`B`** - move to the next unselected hunk
  - **`O`** - move to the previous unselected hunk
- There are several different ways to exit the 3-way diff (available from the **`File`** menu):
  - Exit with MERGE (keyboard shortcut: **`M`**) - This saves the merge result. If there are any unselected difference hunks remaining then you will be warned and given the option of saving the file with conflict markers.
  - Exit with ACCEPT (keyboard shortcut: **`A`**) - This saves the file you are merging in (i.e. the middle one) as the merge result (i.e. you have *accepted* all the changes).
  - Exit with REJECT (keyboard shortcut: **`R`**) - This saves the original working copy file (i.e. the left one) as the merge result (i.e. you have *rejected* all the changes).

If you just want to exit without making any decisions you can also just close the window.

- For further details please read the `xxdiff` users manual (available from the **`Help`** menu). In particular, read the section *Merging files and resolving conflicts*.

If you have resolved all the conflicts in a file then you will be prompted on whether to run `svn resolved` on the file to signal that the file is no longer in conflict.

```
(SHELL PROMPT)$ fcm conflicts
Conflicts in file: Gen_setup_local1.proc
You have chosen to ACCEPT all the changes
Would you like to run "svn resolved"?
Enter "y" or "n" (or just press <return> for "n"): y
Resolved conflicted state of 'Gen_setup_local1.proc'
Conflicts in file: Gen_setup_remote2.proc
Merge conflicts were not all resolved
Conflicts in file: Gen_setup_remote3.proc
All merge conflicts resolved
Would you like to run "svn resolved"?
Enter "y" or "n" (or just press <return> for "n"): y
Resolved conflicted state of 'Gen_setup_remote3.proc'
```

It is important to realise that there are some types of merge that *xxdiff* will not be able to help you with.

- It you have 2 versions of a file, both with substantial changes to the same piece of code, then the `xxdiff` display will be extremely colourful and not very helpful.
- In these cases it is often easier to start with one version of the file and manually re-apply the changes from the other version. It might not be obvious how to do this and you may need to speak to the author of the other change to agree how this can be done. Fortunately this situation should be very rare.
- For a more detailed discussion please refer to Chapter 3: File Merge in the online book called Source Control HOWTO.

Adding and Removing Files

If your working copy contains files which are not under version control then you can use the command `fcm add --check` to add them. This will go through each of the files and prompt to see if you wish to put that file under version control using `svn add`. For each file you can enter **`y`** for yes, **`n`** for no or **`a`** to assume yes for all following files.

```
(SHELL PROMPT)$ fcm add -c
?      xxdiff1.png
?      xxdiff2.png
?      xxdiff3.png
?      xxdiff4.png
Add file 'xxdiff1.png'?
Enter "y", "n" or "a" (or just press <return> for "n"): y
A         xxdiff1.png
Add file 'xxdiff2.png'?
Enter "y", "n" or "a" (or just press <return> for "n"): n
Add file 'xxdiff3.png'?
Enter "y", "n" or "a" (or just press <return> for "n"): a
A         xxdiff3.png
A         xxdiff4.png
```

Similarly, if your working copy contains files which are missing (i.e. you have deleted them without using `svn delete`) then you can use the command `fcm delete --check` to delete them. This will go through each of the files and prompt to see if you wish to remove that file from version control using `svn delete`.

As noted in the Subversion FAQ, it can be dangerous using these commands. If you have moved or copied a file then simply adding them would cause the history to be lost. Therefore take care to only use these commands on files which really are new or deleted.

Committing Changes

The command `fcm commit` should be used for committing changes back to the repository. It differs from the `svn commit` command in a number of important ways:

- Your working copy *must* be up to date. `fcm commit` will abort if it finds that any files are out of date with respect to the repository. This ensures that your working copy reflects how the repository will be after you have committed your changes.
  - This helps to ensure that any tests you have done prior to committing are valid.
  - `fcm commit` is not suitable if you need to commit changes from a working copy containing mixed revisions. However, you are very unlikely to need to do this.
  - Actually there is a small chance that your working copy might not be up to date when you commit if someone else is committing some changes at the same time. However, this should very seldom happen and, even if it does, the commit would fail if any of the files being changed became out of date (i.e. it is not possible to lose any changes).
- If it discovers a file named `#commit_message#` in the top level of your working copy it uses this to provide a template commit message (which you can then edit).
  - If you have performed a merge then a message describing the merge will have been added to this file. It is important that you leave this included in the commit message and do not change its format, as it is used by the `fcm branch` command.
  - You can, if you wish, add entries to this file as you go along to record what changes you have prepared in your working copy. You can also use the command `fcm commit --dry-run` to allow you to edit the commit message without committing any changes.
  - `#commit_message#` is ignored by Subversion (so you won't see it show up as an unversioned files when you run `fcm status`).
- It always operates from the top of your working copy. If you issue the `fcm commit` command from a sub-directory of your working copy then it will automatically work out the top directory and work from there.
  - This ensures that any template commit message gets picked up and that you do not, for example, accidently commit a partial set of changes from a merge.
- It always commits *all* the changes in your working copy (it does not accept a list of files to commit).
  - Once again, this avoids any danger of accidently committing a partial set of changes.
  - You should only work on one change within a working copy. If you need to prepare another, unrelated change then use a separate working copy.
- It runs `svn update` after the commit to ensure that your working copy is at the latest revision and to avoid any confusion caused by your working copy containing mixed revisions.

```
(SHELL PROMPT)$ fcm commit
Starting editor to create commit message ...
Change summary:
------------------------------------------------------------------------
[Project: GEN]
[Branch : branches/test/frsn/r123_foo_bar]
[Sub-dir: <top>]

M      src/code/GenMod_Control/GenMod_Control.f90
M      src/code/GenMod_Control/Gen_SetupControl.f90
------------------------------------------------------------------------
Commit message is as follows:
------------------------------------------------------------------------
An example commit.
------------------------------------------------------------------------
```

```
Would you like to commit this change?
Enter "y" or "n" (or just press <return> for "n"): y
Sending        src/code/GenMod_Control/GenMod_Control.f90
Sending        src/code/GenMod_Control/Gen_SetupControl.f90
Transmitting file data ..
Committed revision 170.
=> svn update
At revision 170.
```

Branching & Merging

Branching is a fundamental concept common to most version control systems. For a good introduction please read the chapter Branching and Merging from the *Subversion book*. Even if you are already familiar with branching using other version control systems you should still read this chapter to see how branching is implemented in Subversion.

Having read this chapter from the *Subversion book* you should understand:

- Why each project directory has sub-directories called *trunk*, *branches* and *tags*. This structure is assumed by `fcm` (Subversion recommends it but doesn't insist on it).
- That when you make a branch you are taking a copy of the entire project file tree. Fortunately, the design of the Subversion repository means that these copies are "cheap" - they are quick to create and take very little space.
- That Subversion doesn't (currently) track merge information for you - this has to be done manually.
- That each revision of your repository can also be thought of as a *changeset*.
- That once a change is committed to a repository it cannot be removed (only reversed). Therefore you must take care not to committ a sensitive document or a large data file unintentionally.

FCM provides various commands which make working with branches easier (as described in the following sections).

Creating Branches

The command `fcm branch --create` should be used for creating new branches. It provides a number of features:

- It applies a standard naming convention for branches. The branch name is automatically constructed for you depending on the option(s) supplied to the command. The full detail of these options are described in the FCM Command Reference > fcm branch section.
- By default, it assumes that you are branching from the last changed revision of the *trunk*.
  - You can use the `--branch-of-branch` option if you need to create a branch of a branch. A branch of a branch can be useful in many situations. For example, consider a shared branch used by several members of your team to develop, say, a new science scheme, and you have come up with some different ideas of implementing the scheme. You may want to create a branch of the shared branch to develop your idea before merging it back to the shared branch. Note that you can only merge a branch of a branch with it's parent or with another branch created from the same parent. You can't, for example, merge it with the trunk.
  - You can use the `--revision <rev>` option if you need to create a branch from an earlier revision of the source.
- Each branch always contains a full copy of the trunk (or its parent branch) - you cannot create a branch from a sub-tree.
  - There would be no reason to only include a sub-tree in a branch.
- It applies a standard commit message which defines how the branch has been created. If a

Trac ticket is specified using the `--ticket <number>` option, it is added to the commit log message. If you need to add anything to the commit log message, please do so **above** the line that says `--Add your commit message ABOVE - do not alter this line or those below--`.

The following is a list of the different types of branches available:

User development branches
: `branches/dev/<Userid>/<Branch_Name>` These are for changes which are intended to be merged back to the trunk once they are complete. Most branches will belong to this type. e.g. branches/dev/frdm/vn6.1_ImprovedDeepConvection, branches/dev/frdm/r2134_NewBranchNamingConvention.

Shared development branches
: `branches/dev/Share/<Branch_Name>`

User test branches
: `branches/test/<Userid>/<Branch_Name>` These are for changes which are *not* intended for the trunk. e.g. Proof of concept work, temporary code written for dealing with a one-off problem, etc.

Shared test branches
: `branches/test/Share/<Branch_Name>`

User packages
: `branches/pkg/<Userid>/<Branch_Name>` These are branches which combine together a number of different development branches. Sometimes this will simply be for testing purposes (i.e. for testing a branch in combination with other branches). Other times it may be the package which eventually gets merged to the trunk (rather than the development branches). e.g. branches/pkg/frdm/vn6.1_TestImprovedDeepConvection

Shared packages
: `branches/pkg/Share/<Branch_Name>` E.g. branches/pkg/Share/vn6.1_NewConvectionScheme.

Configurations
: `branches/pkg/Config/<Branch_Name>` These are major packages which combine together a number of different packages and development branches. e.g. branches/pkg/Config/vn6.1_HadGEM1a.

Releases
: `branches/pkg/Rel/<Branch_Name>` These may be bug-fix branches for system releases, if required. They can also be branches on which stable releases are prepared if you don't do this on the trunk (although you lose the ability to branch from stable releases if you work this way). e.g. branches/pkg/Rel/vn6.1_BugFixes.

```
(SHELL PROMPT)$ fcm br -c -n my_test_branch -k 23 fcm:test
Starting nedit to create commit message ...
Change summary:
------------------------------------------------------------------------
A    svn://fcm1/repos/OPS/branches/dev/frsn/r118_my_test_branch
------------------------------------------------------------------------
Commit message is as follows:
------------------------------------------------------------------------
Create an example branch to demonstrate branch creation for the user guide.
Created /OPS/branches/dev/frsn/r118_my_test_branch from /OPS/trunk@118.
Relates to ticket #23.
------------------------------------------------------------------------
Would you like to go ahead and create this branch?
Enter "y" or "n" (or just press <return> for "n"): y
Creating branch svn://fcm1/repos/OPS/branches/dev/frsn/r118_my_test_branch ...

Committed revision 169.
```

Listing Branches Created by You or Other Users

The command `fcm branch --list` can be used to list the branches you have created at the HEAD of a repository. If you specify the `--user <userid>` option, the branches created by <userid> are listed instead. You can specify multiple users with multiple `--user <userid>` options, or with a colon (:) separated list to a single `--user <userid:list>` option. Note that you can also list shared branches by specifying <userid> as `Share`, configuration branches by specifying <userid> as `Config` and release branches by specifying <userid> as `Rel`. The command returns 0 (success) if one or more branches is found for the specified users, or 1 (failure) if no branch is found.

```
(SHELL PROMPT)$ fcm branch --list fcm:gen
1 branch found for frsn in svn://fcm1/GEN_svn/GEN
fcm:GEN-br/dev/frsn/r1191_clean_up/
(SHELL PROMPT)$ echo $?
0
(SHELL PROMPT)$ fcm branch --list --user frbj --user frsn fcm:gen
2 branches found for frbj, frsn in svn://fcm1/GEN_svn/GEN
fcm:GEN-br/dev/frbj/r1177_gen_ui_for_scs/
fcm:GEN-br/dev/frsn/r1191_clean_up/
(SHELL PROMPT)$ echo $?
0
(SHELL PROMPT)$ fcm branch --list --user frva fcm:gen
0 branch found for frva in svn://fcm1/GEN_svn/GEN
(SHELL PROMPT)$ echo $?
1
```

Getting Information About Branches

The command `fcm branch --info` can be used to get various information about a branch. In particular, it summarises information about merges to and from the branch and its parent.

```
(SHELL PROMPT)$ fcm branch --info
URL: svn://fcm1/FCM_svn/FCM/branches/dev/frsn/r1346_merge
Repository Root: svn://fcm1/FCM_svn
Revision: 1385
Last Changed Author: frsn
Last Changed Rev: 1385
Last Changed Date: 2006-04-20 11:08:45 +0100 (Thu, 20 Apr 2006)
------------------------------------------------------------------------
Branch Create Author: frsn
Branch Create Rev: 1354
Branch Create Date: 2006-04-04 14:27:47 +0100 (Tue, 04 Apr 2006)
Branch Parent: svn://fcm1/FCM_svn/FCM/trunk@1346
Last Merge From Parent, Revision: 1444
Last Merge From Parent, Delta: /FCM/trunk@1439 cf. /FCM/trunk@1395
Merges Avail From Parent: 1445
Merges Avail Into Parent: 1453 1452 1449 1446 1444 1443 1441 1434 1397 1396 ...
```

If you need information on the current children of the branch, use the `--show-children` option of the `fcm branch --info` command. If you need information on recent merges to and from the branch and its siblings, use the `--show-siblings` option of the `fcm branch --info` command.

To find out what changes have been made on a branch relative to its parent you can use the command `fcm diff --branch`.

- You can combine this with the options:
  `--graphical`
  : to display the differences using a graphical *diff* tool

`--trac`
: to display the differences using Trac

`--wiki`
: to print a wiki syntax suitable for inserting into Trac

- The base of the difference is adjusted to account for any merges from the branch to its parent or vice-versa.

Switching your working copy to point to another branch

The command `fcm switch` can be used to switch your working copy to point to another branch. For example, if you have a working copy at `$HOME/work`, currently pointing to the trunk or a branch of a project at `svn://fcm1/FCM_svn/FCM/trunk`, you can switch the working copy to point to another branch of same project:

```
(Shell prompt)$ cd $HOME/work
(Shell prompt)$ fcm sw dev/frsn/r959_blockdata
-> svn switch --revision HEAD svn://fcm1/FCM_svn/FCM/branches/dev/frsn/r959_blockdata
U    doc/user_guide/getting_started.html
U    doc/user_guide/code_management.html
U    doc/user_guide/command_ref.html
U    src/lib/Fcm/SrcFile.pm
U    src/lib/Fcm/Util.pm
U    src/lib/Fcm/Build.pm
U    src/lib/Fcm/Cm.pm
U    src/lib/Fcm/SrcPackage.pm
U    src/bin/fcm_internal
U    src/bin/fcm_gui
Updated to revision 1009.
```

Unlike `svn switch`, `fcm switch` does extra checking to ensure that your whole working copy is switched to the new branch at the correct level of sub-directory. In addition, you can specify only the *branch* part of the URL, such as `trunk`, `branches/dev/fred/r1234_bob` or even `dev/fred/r1234_bob` and the command will work out the full URL for you.

Deleting Branches

The command `fcm branch --delete` can be used to delete branches which are no longer required. Before being asked to confirm that you want to delete the branch, you will first see the same output as from `fcm branch --info`. This allows you to check, for example, whether your branch is being used anywhere else or whether the latest changes on your branch have been merged to the trunk. You will be prompted to edit your commit log message. If you need to add anything to the commit log message, please do so **above** the line that says `--Add your commit message ABOVE - do not alter this line or those below--`.

Merging

As mentioned earlier, Subversion doesn't track merge information (although, in the longer term, there are plans to add this feature). However, `fcm` *does* track a limited amount of merge information. It does this by making a number of assumptions:

- That all merges are performed using FCM and are identified using a standard template in the commit log message.
- That you only ever merge all the changes available on the source branch up to a chosen point (i.e. you can't only include a subset of the changes made to the branch).
- That the source and target are both branches (or the trunk) in the same FCM project.
- That the source and target are directly related, i.e. they must either have a parent/child relationship or they are siblings from the same parent branch.

Note that the term *source branch* and *target branch* referred to above can also mean the trunk.

To perform a merge, use the command `fcm merge <source>`. This includes a number of important features:

- If it finds any local modifications in your working copy then it checks whether you wish to continue (in most cases you won't want to mix a merge with other changes).
- It determines the base revision and path of the *common ancestor* to be used for the merge, taking into account any merges from the *source* to the *target* or vice-versa.
- Before doing the merge, (unless you specify the `--non-interactive` option), it reports what changes will result from performing the merge and checks that you wish to continue.
- It adds details of the merge, using a standard template, into the commit message file (`#commit_message#`). If you need to add any extra comment, you should do so **above** the line that says `--Add your commit message ABOVE - do not alter this line or those below--`.
  - If you decide to revert the merge, you should remove the template line manually from the commit message file, making sure that you do not alter the standard template by accident.

```
(SHELL PROMPT)$ fcm merge trunk # merge changes from the trunk into the branch
Available Merges From /FCM/trunk: 1383 1375
Please enter the revision you wish to merge from
  (or just press <return> for "1383"):
About to merge in changes from /FCM/trunk@1383 compared with /FCM/trunk@1371
This merge will result in the following changes:
------------------------------------------------------------------------------
A    doc/standards/fortran_standard.html
U    src/lib/Fcm/ReposBranch.pm
------------------------------------------------------------------------------
Would you like to go ahead with the merge?
Enter "y" or "n" (or just press <return> for "n"): y
Performing merge ...
A    doc/standards/fortran_standard.html
U    src/lib/Fcm/ReposBranch.pm
```

Using the GUI

So far, all the tools described have been command line tools. Many people will be happy with these but, for those who prefer it, there is also a simple Graphical User Interface (GUI).

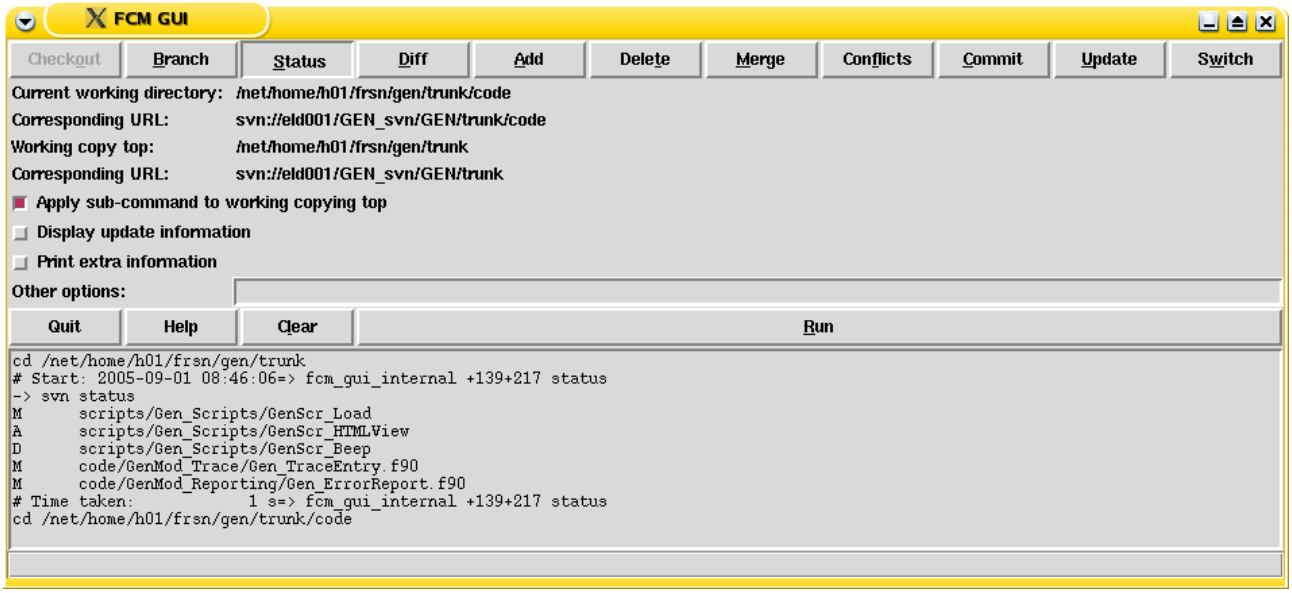
Starting the GUI

To run the GUI simply issue the command `fcm gui` from the directory you want as your working directory. You can also start the GUI from within Konqueror (see Accessing the GUI from Konqueror).

The GUI consists of several sections:

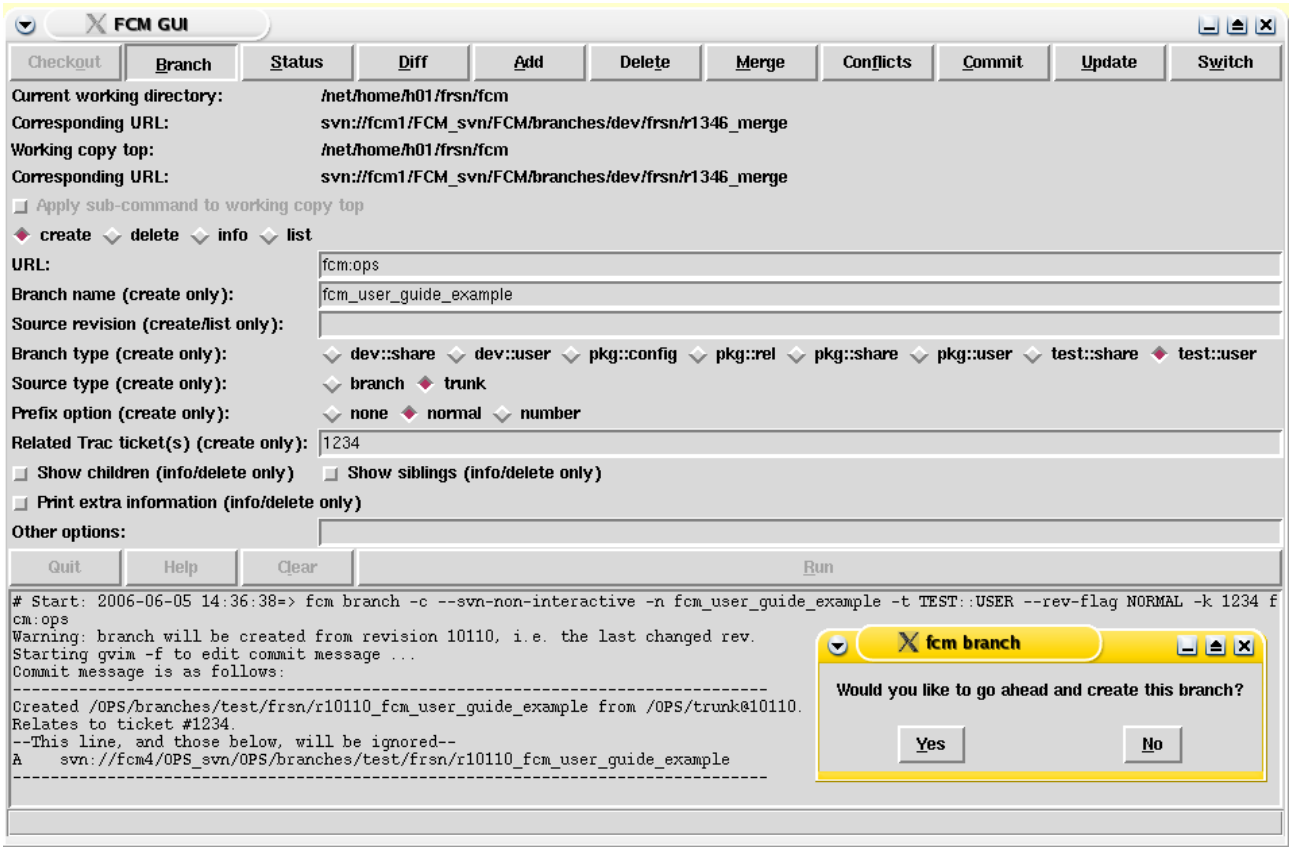
- The top section contains a row of buttons to allow you to select which command you want to run.
- Beneath this is shown the current working directory and the top level directory of your working copy (these may be the same).
- Beneath this come various buttons and entry boxes to allow you to configure the command you have selected. These vary according to the command.
- Beneath this comes a further row of buttons
  - *Quit* - this exits the GUI.
  - *Help* - this displays the help message for the selected command.
  - *Clear* - this empties the text window.
  - *Run* - this allows you to run your command.

- Beneath this comes a scrolling text window where the output from the commands is displayed.
- The bottom section displays help information when you position the cursor over various parts of the GUI.



Example GUI screen with the **`Status`** commands selected

If you run a more complicated command, like `fcm branch`, which prompts for input then extra entry windows will pop up.



Example GUI pop-up window

GUI Commands

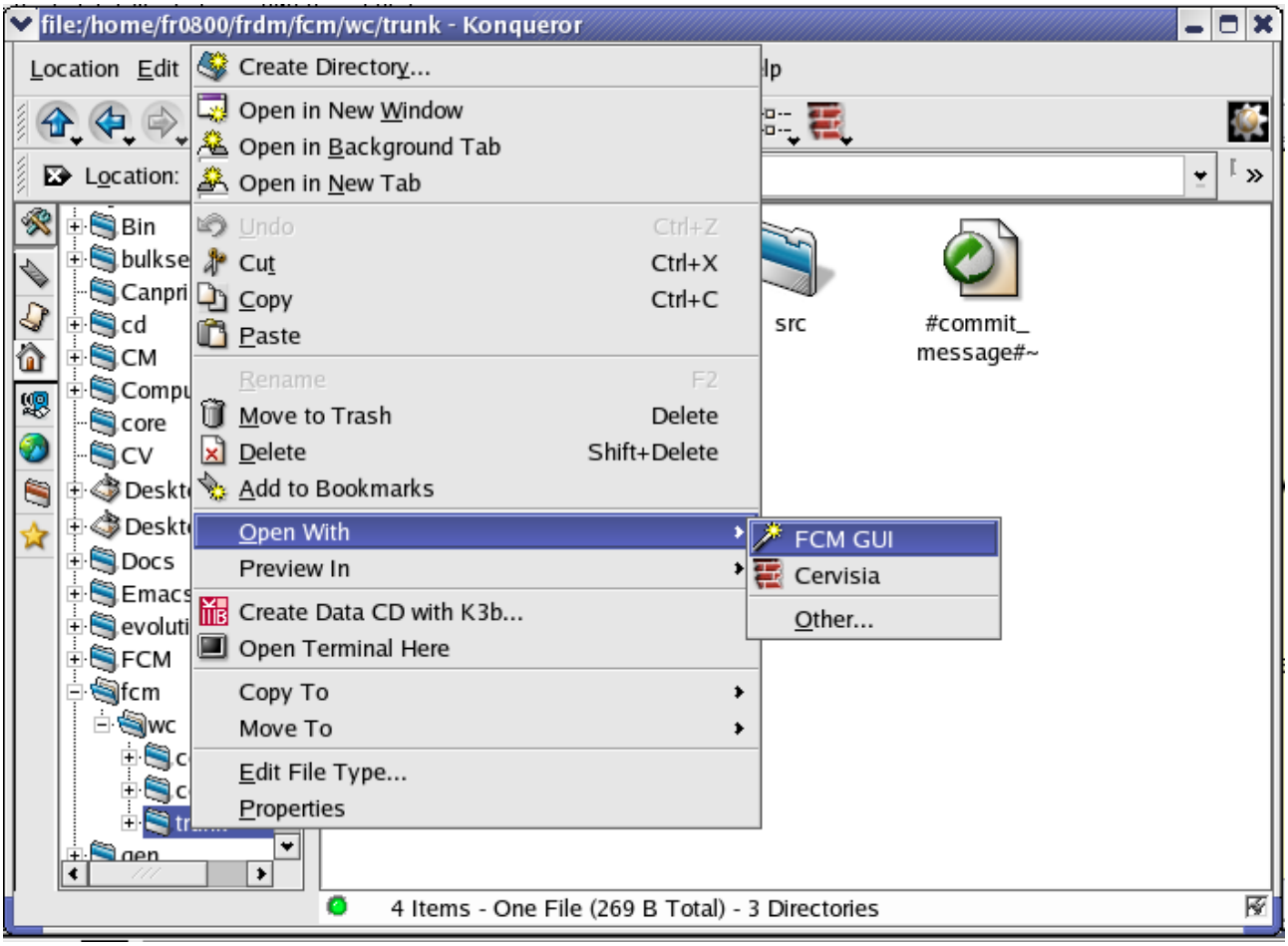
The commands available from the GUI should be self explanatory. A few points to note:

- If the current directory is not a working copy, you will only be able to Checkout a working copy or create a branch from the GUI.
- The **`Checkout`** command is only available if you start the GUI in a directory which is not already a working copy. After successfully running a checkout the GUI automatically sets the working directory to the top of this new working copy.
- With some commands (Status, Diff, Add, Delete, Conflicts) you can choose whether to run from the top level of your working copy or from your working directory. With the remaining commands this would not make sense and they can only be run from the top level.
- You can only issue commands from the GUI if they do not need to prompt you for authentication (i.e. the Subversion command can be run with the `--non-interactive` option).
  - If authentication is required then the command issued by the GUI will fail. For the `branch --create`, `branch --delete` and `commit` commands, which support the `--password` option, you should specify your password in **`Other options`** and click **`Run`** again. For other commands, you should run the command in interactive mode on the command line. Use the command displayed in the GUI text window but remove the `--non-interactive` option.
  - Most repositories will be configured so that you only need authentication for writing (not reading). Therefore, the first command requiring authentication will probably be creating a branch or commiting to the trunk.
  - You should only need to do this the first time you ever issue such a command on a each repository (unless the repository is moved to a new location) since the Subversion client caches this information for future comamnds .

Accessing the GUI from Konqueror

To enable access from Konqueror run the script `fcm_setup_konqueror`. You can then use the Konqueror file manager to select the directory which you want as your working directory. To run the GUI click the right mouse button and select **`Open With => FCM GUI`**.

- Make sure that your current selection is the directory and not a file within that directory.



Running the GUI from within Konqueror

Known Problems with Subversion

There are some limitations with Subversion v1.3 which you should be aware of:

- The `svn rename` command is not a true rename/move operation, but is implemented as a copy and delete. As a result, if you rename an item in a branch, and later attempt to merge it back to the trunk, the operation may not be handled correctly by `svn merge` (see subversion issue 898 for further details). Support for a “true rename” will hopefully be available in Subversion in the near future. Until this is implemented, you should avoid renaming of files or directories unless you can ensure that no-one is working in parallel on the affected areas of the project.

Using Trac

Trac has a simple and intuitive web interface which is relatively easy to pick up. It also includes a User and Administration Guide which is full of helpful information (and is referred to extensively in this section).

Trac contains a menu bar at the top of each page (which we will refer to as the *Trac menu*). This provides access to all the main features.

Logging In

Although different projects may choose their own rules, we expect that most systems will have Trac configured so that all the information is viewable by anyone. However, in order to make any changes you will need to login. This ensures that any changes are identified with the appropriate userid.

In the rest of this section it is assumed that you have logged in to Trac and are therefore able to make changes.

If you haven't yet got a Trac userid (which should be the same as the userid you use for committing changes to Subversion) then please contact your system manager.

Using the Wiki Pages

A wiki enables documents to be written in a simple markup language using a web browser. See the Trac Guide for information on the Trac Wiki Engine. Make sure that you read the information provided on:

- Wiki Formatting which explains how to format your wiki pages.
- Wiki Page Names which explains how *CamelCase* is used to create New Wiki Pages.
- Trac Links which allow hyperlinking between Trac entities (tickets, reports, changesets, Wiki pages, milestones and source files). This is a fundamental feature of Trac which makes it easy, for example, to link a bug report (ticket) to the changeset which fixed the bug (and vice-versa).

Whenever you are viewing a wiki page in Trac you should see several buttons at the bottom of the page:

- **`Edit This Page`** - Clicking this will bring up a page where you can edit the page contents. Before saving your changes you can preview how the modified page will appear. You can also leave a comment explaining what changes you made.
- **`Attach File`** - Allows you to attach files to a page, e.g. an image.
- If you have admin rights then you will also see
  - **`Delete This Version`** - Delete the particular version of the page you are viewing.
  - **`Delete Page`** - Delete the page and all its history.

  Use with care - these operations are irreversible!

At the top of each wiki page at the right hand side you can select **`Page History`**. This shows you the full history of each page with details of when each change was made, who made the change and what the changes were.

Using the Repository Browser

The Trac Browser is used to view the contents of your repository. To get to it just select **`Browse Source`** from the Trac menu. You can view directories and files at any version, see their revision histories and view changesets. Any wiki formatting in log messages is recognised and interpreted so you can easily link a changeset to a Trac ticket by using Trac Links.

Using the Issue Tracker

The Trac issue database provides a way of tracking issues within a project (e.g. bug reports, feature requests, software support issues, project tasks). Within Trac an issue is often referred to as a *Ticket*.

Please refer to the Trac Guide for the following information:

- The Trac Ticket System - Creating and modifying tickets.
  - Only Trac accounts with admin rights can modify ticket descriptions.
- Trac Ticket Queries - List tickets matching your chosen criterion.

Using the Roadmap

Each ticket can be assigned to a milestone. The Trac Roadmap can then be used to provide a view on the ticket system. This can useful to see what changes went into a particular system release or what changes are outstanding before a milestone can be reached.

Please refer to the Trac Guide for further information on the Trac Roadmap.

- Only Trac accounts with admin rights can add, modify and remove milestones using the web interface.

Using the Timeline

The Trac Timeline allows you to list all the acitivity on a project over any given period. It can list:

- Creation and changes to wiki pages.
- Creation, closure and changes to tickets.
- Commits to the Subversion repository.
- Milestones reached.

Code Management Working Practices

Introduction

The previous chapter described how to use the various parts of the FCM code management system. They also described aspects of working practises which are enforced by the system. This section discusses other recommended working practises. They are optional in the sense that you don't have to follow them to use FCM. It is a matter for individual projects to decide which working practises to adopt (although we expect most projects/systems at the Met Office to adopt similar practises).

Making Changes

This sub-section gives an overview of the recommended approach for preparing changes. Particular topics are discussed in more detail in later sub-sections where appropriate.

The recommended process for making a change is as follows:

1. Before work starts on any coding you should make sure that there is a Trac ticket open which explains the purpose of the change.
   - Make sure that you set the ticket milestone to indicate which release of the system you are aiming to include your change in.
   - Accept the ticket to indicate that you are working on the change.
   - For further advice on using tickets see Trac Tickets later in this section.
2. Create a branch
   - For very simple changes you may be happy to prepare your changes directly on the trunk. For further details see When to Branch later in this section.
   - Create your branch either from the latest revision or from a stable release (see Where to Branch From later in this section).
3. Prepare your code changes on the branch
   - Commit interim versions to your branch on a regular basis as you develop your change. This makes it much easier to keep track of what you're changing and to revert changes if necessary.
   - You may wish to merge in changes from the trunk. For further details see Merging From the Trunk later in this section.
     - Make sure that you always commit any local changes to your branch before doing a merge. Otherwise it becomes impossible to distinguish your changes from those you have merged in. It is also impossible to revert the merge without losing your local changes.
     - Likewise, always commit the merge to your branch (after resolving any conflicts) before making any further changes.
   - Don't include unrelated changes. If you want to make some changes which aren't really associated with your other changes then use a separate ticket and branch for these changes.
4. Once your changes are ready for review, update the Trac ticket to record which revision of the branch is to be reviewed and assign the ticket to your reviewer.
5. If the reviewer is happy with the change then he/she should update the ticket to record that the change is approved and assign the ticket back to you.
   - The reviewer can use the command `fcm diff --branch <branch_name>` to examine all of the changes on the branch.
   - If changes are necessary then these should be prepared and then the ticket updated to refer to the new revision under review.

6. Once the change is approved it can be merged back to the trunk
    - If you have been merging the latest changes from the trunk onto your branch then the merge should be automatic. If not you may have conflicts to resolve.
    - Make sure that each merge is a separate commit to the trunk. i.e. Don't combine changes from several branches in one commit. This makes it easier to reverse changes if necessary. It also makes the changeset easier to understand.
    - Make sure that you use a good log message to describe your change. For further details see Commit Log Messages later in this section.
    - Once the changes are commited, update the ticket to refer to the changeset. Then the ticket can be closed.
7. Once you are finished with the branch it should be deleted.

Working Copies

Some points to consider regarding working copies:

1. In general we recommend that you keep your working copies in your home directory. This ensures that any local changes which you accidently delete can be recovered via the snapshot facility (on Met Office Exeter and Reading based systems).
2. If the size of your project is small then you will probably find it easiest to work with a complete copy of the project (either the trunk or your branch). This means that you always have immediate access to all the files and that you are always able to perform merges using your normal working copy.
3. If you have a large project then you may prefer to work on a sub-tree of your project.

    Pros:

    - Subversion operations on your working copy are faster.
    - Your working copies use up less disk space. Remember that you may be working on several changes at once on separate branches so you may wish to have several working copies.

    Cons:

    - You cannot always perform merge operations in sub-trees (if the changes which need to be merged include files outside of your sub-tree). To handle this we suggest that if you need to perform a merge using a complete copy of your project you check it out in your *$LOCALDATA* area (local disk space which is not backed up) to be used purely for doing the merge.
    - You may find that your change involves more files than you originally thought and that some of the files to be changed lie outside of your working copy. You then have to make sure that you have committed any changes before checking out a larger working copy.

Branching & Merging

When to Branch

If you are making a reasonably large change which will take more than a hour or two to prepare then there are clear advantages to doing this work on a branch.

- You can commit intermediate versions to the branch.
- If you need to merge in changes from the trunk then you have a record of your files prior to the merge.
- The version of the code which gets reviewed is recorded. If subsequent changes are required

then only those changes will need reviewing.

However, if you are only making a small change (maybe only one line) should you create a branch for this? There are two possible approaches:

Always Branch

ALL coding changes are prepared on branches.

Pros: Same process is followed in all cases.

Cons: The extra work required to create the branch and merge it back to the trunk may seem unnecessary for a very small change.

Branch When Needed

Small changes can be committed directly to the trunk (after testing and code review).

Pros: Avoids the overhead of using branches.

Cons: Danger of underestimating the size of a change. What you thought was a small change may turn out to be larger than you thought (although you can always move it onto a branch if this happens).

This is a matter for project policy although, in general, we would recommend the *Branch When Needed* approach.

Where to Branch From

When you create a new branch you have two choices for which revision to create the branch from:

The latest revision of the trunk

This is the preferred choice where possible. It minimised the chances of conflicts when you need to incorporate your changes back onto the trunk.

An older revision of the trunk

There are a number of reasons why you may need to do this. For example:

- You are using a stable version to act as your *control* data.
- You need to know that your baseline is well tested (e.g. scientific changes).
- Your change may need to be merged with other changes relative to a stable version for testing purposes or for use in a package (see Creating Packages later in this section).

Merging From the Trunk

Once you've created your branch you need to decide whether you now work in isolation or whether you periodically merge in the latest changes from the trunk.

- Regularly merging from the trunk minimises the work involved when you are ready to merge back to the trunk. You deal with any merge issues as you go along rather than all at the end (by which time your branch and the trunk could have diverged significantly).
- One downside of merging from the trunk is that the baseline for your changes is a moving target. This may not be what you want if you have some *control* results that you have generated.
- Another downside of merging from the trunk is that it may introduce bugs. Although any code

on the trunk should have been tested and reviewed it is unlikely to be as well tested as code from a stable release.

- Unless you originally created your branch from the latest revision of the trunk it is unlikely that you are going to want to merge in changes from the trunk. The exception to this is once your change is complete when it may make sense to merge all the changes on the trunk into your branch as a final step. This is discussed in Merging Back to the Trunk below.

So, there are basically three methods of working:

Branch from a stable version and prepare all your changes in isolation
: Necessary if you need to make your change relative to a well tested release.

Branch from the latest code but then prepare all your changes in isolation
: Necessary if you need a stable baseline for your *control* data.

Branch from the latest code and then update your branch from the trunk on a regular basis
: This is considered *best practice* for parallel working and should be used where possible.

Merging Back to the Trunk

Before merging your change back to the trunk you will need to test your change and get it reviewed. There are two options for what code to test and review:

Test and review your changes in isolation, then merge to the trunk and deal with any conflicts

: This may be the best method if:

  - Your changes have already been tested against a stable baseline and re-testing after merging would be impracticable.
  - Your branch needs to be available for others to merge in its changes in isolation.

Merge in the latest code from the trunk before your final test and review

: This has the advantage that you are testing and reviewing the actual code which will be committed to the trunk. However, it is possible that other changes could get committed to the trunk whilst you are completing your testing and review. There are several ways of dealing with this:

  - Use locking to prevent it happening. The danger with this is that you may prevent others from being able to get their change tested and reviewed, hence inhibiting parallel devlopment.
  - Insist that the change is re-tested and reviewed. The problem with this is that there is no guarantee that the same thing won't happen again.
  - Merge in the new changes but don't insist on further testing or review.
    - In most cases any changes won't clash so there is little to worry about.
    - Where there are clashes then, in most cases, they will be trivial with little danger of any side-effects.
    - Where the clashes are significant then, in most cases, this will be very obvious whilst you are resolving the conflicts. In this case you should repeat the testing and get the updates reviewed.

    This is the recommended approach since it doesn't inhibit parallel development and yet the chances of a bad change being committed to the trunk are still very small.

  You should also consider what can be done to minimise the time taken for testing and review.

  - Try to keep your changes small by breaking them down where possible. Smaller changes are easier and quicker to review. This also helps to minimise merge problems by getting changes back onto the trunk earlier.
  - Automate your testing as far as possible to speed up the process.

Most projects will require the developer who prepared the change to merge it back to the trunk once it is complete. However, larger projects may wish to consider restricting this to a number of experienced / trusted developers.

- This makes it easier to control and prioritise the merges.
- It applies an extra level of quality control.
- It minimises the risk of mistakes being merged back on to the trunk by less experienced developers
- Scientific developers can concentrate on the scientific work.
- One issue is that the person doing the merge to the trunk may need help from the original developer to prepare a suitable log message.

When to Delete Branches

Once you are finished with your branch it is best to delete it to avoid cluttering up the directory tree (remember that the branch and all its history will still be available). There are two obvious approaches to deleting branches:

Delete the branch as soon as it has been merged back to the trunk (prior to closing any associated Trac ticket)
: This is the tidiest approach which minimises the chances of old branches being left around.

Delete the branch once a stable version of the system has been released which incorporates your change
: If a bug is found in your change during integration testing then you can prepare the fix on the original branch (without having to do any additional work to restore the branch).

Working with Binary Files

The `fcm conflicts` command and `xxdiff` can only help you resolve conflicts in text files. If you have binary files in your repository you need to consider whether conflicts in these files would cause a problem.

Resolving Conflicts in Binary Files

Conflicts in some types of binary files can be resolved manually. When you are satisfied that the conflicts are resolved, issue the `fcm resolved` command on the file to remove the conflict status. (You will be prevented from committing if you have a conflicting file in your working copy.)

If you have a conflicting MS Office 2003+ document, you may be able to take advantage of the **`Tools > Compare and Merge Documents`** facility. Consider a working copy, which you have just updated from revision 100 to revision 101, and someone else has committed some changes to a file `doument.doc` you are editing, you will get:

```
(SHELL PROMPT)$ fcm conflicts
Conflicts in file: document.doc
document.doc: ignoring binary file, please resolve conflicts manually.
(SHELL PROMPT)$ fcm status
=> svn st
?      document.doc.r100
?      document.doc.r101
C      document.doc
```

Open `document.doc.r101` with MS Word. In **`Tools > Compare and Merge Documents...`**, open `document.doc`. You will be in Track Changes mode automatically. Go through the document to accept, reject or merge any changes. Save the document and exit MS Word when you are ready. Finally, issue the `fcm resolved` command to remove the conflict status:

```
(SHELL PROMPT)$ fcm resolved document.doc
=> svn resolved document.doc
Resolved conflicted state of 'document.doc'
(SHELL PROMPT)$ fcm status
=> svn st
M      document.doc
```

Another type of conflict that you may be able to resolve manually is where the binary file is generated from another file which can be merged. For instance, some people who use LaTeX also store a PDF version of the document in the repository. In such cases it is easy to resolve the conflict by re-generating the PDF file from the merged LaTeX file and then issuing the `fcm resolved` command to remove the conflict status. Note that, in this particular case, a better approach might be to automate the generation of the PDF file outside of the repository.

Using Locking

For files with binary formats, such as artwork or sound, it is often impossible to merge conflicting changes. In these situations, it is necessary for users to take strict turns when changing the file in order to prevent time wasted on changes that are ultimately discarded.

Subversion supports "locking" to allow you to prevent other users from modifying a file while you are preparing changes. For details please refer to the chapter Locking from the Subversion book. Note that:

- FCM does not add any functionality to the locking commands provided by Subversion.
- If you need to lock a file you must do this in a working copy of the trunk. There is nothing to stop you preparing the changes in a branch (maybe you want to prepare the change in combination with a number of other changes which do not require locking). However, you must always remember to lock the file in the trunk first to prevent other users from preparing changes to the file in parallel.
- Locking isn't the only way of preventing conflicts with binary files. If you only have a small project team and a small number of binary files you may find it easier to use other methods of communication such as emails or just talking to each other. Alternatively, you may have a working practise that particular files are only modified by particular users.

Commit Log Messages

Certain guidelines should be adhered to when writing log messages for code changes when committing to the trunk:

- Try to start off the log message with one line indicating the general nature of the change. This helps developers to tell whether a change is important to them when viewing the Trac timeline view.
- If you want to use bullets in your message then make them compatible with Wiki Formatting. For example:

  ```
  No bullet
   * First level bullet (single space at beginning)
     * Second level bullet (three spaces at beginning)
   1. Numbered item instead of a bullet
  ```

  This will ensure that the log message is displayed with proper bullets in the Trac changeset view. You can also include other types of wiki formatting but please be aware that the message still needs to be readable when simply viewed as text (e.g. via `fcm log`).

- If your changes close a Trac ticket, make sure that your log message refers to this using Trac Links, e.g. `Closes issue #26.`
- Don't leave blank lines at the end of your log message since they get included in the message and, therefore, get included in the ouptut from `fcm log`.
- Take care to avoid making mistakes in your log messages since correcting them involves additional work. However, if you realise that that you've made a mistake don't leave it - get it corrected.
  - A log message can be corrected using the `propedit` command,
    e.g. `fcm propedit svn:log --revprop -r REV`
    Take care since this is an "unversioned" property so you run the risk of losing information if you aren't careful with your edits.
  - By default, FCM repositories are configured such that all users can update log messages. If you are not the original author of the changeset then the original author will be sent an e-mail informing them of the change. Other users can also be informed of log message changes if they wish (see the section Watching changes in log messages for details).

There are two possible approaches to recording the changes to individual files:

Maintain history entries in file headers

> *Pros:* You don't need access to the Subversion repository in order to be able to view a files change history (e.g. external collaborators).
>
> *Cons:*
>
> - History entries will produce clashes whenever files are changed in parallel (although these conflicts are trivial to resolve).
> - Source files which are changed regularly can become cluttered with very long history entries.
> - It is not possible to include history entries in some types of file.

Record which files have changed in the commit log message

> The log message should name every modified file and explain why it was changed. Make sure that the log message includes some sort of description for every change. The value of the log becomes much less if developers cannot rely on its completeness. Even if you've only changed comments, note this in the message. For example:
>
> ```
> * working_practices.html:
>   Added guidelines for writing log messages.
> ```
>
> If you make exactly the same change in several files, list all the changed files in one entry. For example:
>
> ```
> * code_management.html, system_admin.html, index.html:
>   Ran pages through tidy to fix HTML errors.
> ```
>
> It shouldn't normally be necessary to include the full path in the file name - just make sure it is clear which of the changed files you are referring to. You can get a full list of the files changed using `fcm log -v`.

When you're committing to your own branch then you can be much more relaxed about log messages. Use whatever level of detail you find helpful. However, if you follow similar guidelines then this will help when it comes to preparing the log message when your change is merged back to the trunk.

Trac Tickets

Creating Tickets

There are two different approaches to using the issue tracker within Trac:

All problems should be reported using Trac tickets

> *Pros:* The issue tracker contains a full record of all the problems reported and enhancements requested.
>
> *Cons:* The issue tracker gets cluttered up with lots of inappropriate tickets, (which can make it much harder to search the issues and can slow down the response to simple issues).
>
> - Duplicate tickets.
> - Issues already discussed in the documentation.
> - Problems which turn out to be unrelated to the system.
> - Problems which are poorly described.
> - Things which would be better solved by a quick conversation.

A Trac ticket shouldn't be created until the issue has been agreed

> Problems and issues should first be discussed with the project team / system maintainers. Depending on the project, this could be via email, on the newsgroups or through a quick chat over coffee.
>
> Nothing is lost this way. Issues which are appropriate for the issue tracker still get filed. It just happens slightly later, after initial discussion has helped to clarify the best description for the issue.

Using Tickets

This sub-section provides advice on the best way of using tickets:

1. In general, mature systems will require that there is a Trac ticket related to every changeset made to the trunk. However this doesn't mean that there should be a separate ticket for each change.
   - If a change is made to the trunk and then a bug is subsequently found then, if this happens before the next release of the system, the subsequent change can be recorded on the same ticket.
   - There can often be changes which don't really affect the system itself since they are just system administration details. One way of dealing with this is to open a ticket for each release in which to record all such miscellaneous changes. It will probably be acceptable to review these changes after they have been committed, prior to the system release.
2. Whenever you refer to source files/directories in tickets, make sure that you refer to particular revisions of the files. This ensures that the links will work in the future, even if those files are no longer in the latest revision. For example:
   ```
   Changes now ready for review:
   source:/OPS/branches/dev/frdm/r123_MyBranch@234
   ```
3. For some types of information, simply appending to the ticket may not be the best way of working. For example, design notes or test results may be best recorded elsewhere, preferably in a wiki page. If using wiki pages we recommend using a naming convention to identify the wiki page with the associated ticket, for example:
   ```
   Please refer to [wiki:ticket/123/Design design notes]
   See separate [wiki:ticket/123/TestResults test results]
   ```
   Note that the square brackets have to be used since a page name containing numbers is not

recognised automatically.

Creating Packages

Sometimes you may need to combine the changes from several different branches. For example:

- Your branch is just part of a larger change which needs to be tested in its entirety before committing to the trunk.
- You have some diagnostic code stored on a branch which you want to combine with another branch for testing purposes.

We refer to this as creating a *package*.

To create a package you simply create a new branch as normal. The *type* should be a *package* or possibly a *configuration* branch to help you distinguish it from your other branches. You then simply merge in all of the branches that you want to combine using `fcm merge`.

- The chance of conflicts will be reduced if the branches you are combining have been created from the same point on the trunk. Your package branch should also be created from the same point on the trunk.
  - *Currently, `fcm merge` will not work unless this is true.*
- If further changes are made on a branch you are using in a package then you can incorporate these changes into your package using `fcm merge`. Note, however, that if you have a branch which is being used in a package then you should avoid merging changes from the trunk into your branch. If you do then it will be very difficult to get updates to your branch merged into the package.

The `fcm branch --info` command is very useful for maintaining packages. It tells you all of the branches which have been merged into your package and whether there are any more recent changes on those branches.

Preparing System Releases

There are two ways of preparing system releases:

A system release is simply a particular revision of the trunk

In order to do this it will be necessary to restrict changes on the trunk whilst the release is being prepared.

- Users can continue to develop changes not intended for inclusion in this release on branches.
- This may be a problem if preparing the release takes too long.

Create a release branch where the release is finalised

You then lose the ability to be able to branch from the release.

It may be harder to identify what changes have been made between releases (since you can’t simply look at all the changesets made between two revisions of the trunk).

Rapid vs Staged Development Practises

Most of this section on working practises has focussed on projects/systems which are quite mature. Such systems are likely to have regular releases and will, for example, insist that all changes to the trunk are reviewed and tested.

If your system is still undergoing rapid development and has not yet reached any sort of formal release then you will probably want to adopt a much more relaxed set of working practises. For example:

- Changes don’t need to be reviewed.
- More changes will be committed to the trunk. Only very large changes will be prepared on branches.
- No requirement to have a Trac ticket associated with each change.

We have tried to avoid building too many assumptions about working practises into the FCM system. This gives projects the flexibility to decide which working practises are appropriate for their system. Hopefully this means that FCM can be used for large or small systems and for rapidly evolving or very stable systems.

The Extract System

Introduction

The extract system provides an interface between the revision control system (currently Subversion) and the build system. Where appropriate, it extracts code from the repository and other user-defined locations to a directory tree suitable for feeding into the build system. In this chapter, we shall use many examples to explain how to use the extract system. At the end of this chapter, you will be able to extract code from the local file system as well as from different branches of different repository URLs. You will also learn how to mirror code to an alternate destination. Finally, you will be given an introduction on how to specify configurations for the build system via the extract configuration file. (For further information on the build system, please see the next chapter The Build System.) The last section of the chapter tells you what you can do in the case when Subversion is not available.

The Extract Command

To invoke the extract system, simply issue the command:

```
fcm extract
```

By default, the extract system searches for an extract configuration file `ext.cfg` in `$PWD` and then `$PWD/cfg`. If an extract configuration file is not found in these directories, the command fails with an error. If an extract configuration file is found, the system will use the configuration specified in the file to perform the current extract.

If the destination of the extract does not exist, the system performs a new full extract to the destination. If a previous extract already exists at the destination, the system performs an incremental extract, updating any modifications if necessary. If a full (fresh) extract is required for whatever reason, you can invoke the extract system using the `-f` option, (i.e. the command becomes `fcm extract -f`). If you simply want to remove all the items generated by a previous extract in the destination, you can invoke the extract system using the `--clean` option.

For further information on the extract command, please see FCM Command Reference > fcm extract.

Simple Usage

The extract configuration file is the main user interface of the extract system. It is a line based text file. For a complete set of extract configuration file declarations, please refer to the Annex: Declarations in FCM extract configuration file.

Extract from a local path

A simple example of a basic extract configuration file is given below:

```
# Example 1
# ------------------------------------------------------------------------
cfg::type          ext        # line 1
cfg::version       1.0        # line 2
                              # line 3
dest               $PWD       # line 4
                              # line 5
repos::var::user   $HOME/var  # line 6
                              # line 7
expsrc::var::user  code       # line 8
```

The above demonstrates how to use the extract system to extract code from a local user directory. Here is an explanation of what each line does:

- *line 1*: the label `CFG::TYPE` declares the type of the configuration file. The value `ext` tells the system that it is an extract configuration file.
- *line 2*: the label `CFG::VERSION` declares the version of the extract configuration file. The current default is `1.0`. Although it is not currently used, if we have to change the format of the configuration file at a later stage, we shall be able to use this number to determine whether we are reading a file with an older format or one with a newer format.
- *line 3*: a blank line or a line beginning with a `#` is a comment, and is ignored by the interpreter.
- *line 4*: the label `DEST` declares the destination root directory of this extract. The value `$PWD` expands to the current working directory.
- *line 5*: comment line, ignored.
- *line 6*: the label `REPOS::<pck>::<branch>` declares the top level URL or path of a repository. The package name of the repository is given by <pck>. In our example, we choose `var` as the name of the package. (You can choose any name you like, however, it is usually sensible to use a package name that matches the name of the project or system you are working with.) The branch name in the repository is given by <branch>. (Again, you can choose any name you like, however, it is usually sensible to use a name such as `base`, `user` or something that matches your branch name.) In our example, the word `user` is normally used to denote a local user directory. Hence the statement declares that the repository path for the `var` package in the `user` branch can be found at `$HOME/var`.
- *line 7*: comment line, ignored.
- *line 8*: the label `EXPSRC::<pck>::<branch>` declares an *expandable* source directory for the package <pck> in the branch <branch>. In our example, the package name is `var`, and the branch name is `user`. These match the package and the branch names of the repository declaration in line 6. It means that the source directory declaration is associated with the path `$HOME/var`. The value of the declaration `code` is therefore a sub-directory under `$HOME/var`. By declaring a source directory using an `EXPSRC` label, the system automatically searches for all sub-directories (recursively) under the declared source directory.

Invoking the extract system using the above configuration file will extract all sub-directories under `$HOME/var/code` to `$PWD/src/var/code`. Note: the extract system ignores symbolic links and hidden files, (i.e. file names beginning with a `.`). It will write a build configuration file to `$PWD/cfg/bld.cfg`. The configuration used for this extract will be written to the configuration file at `$PWD/cfg/ext.cfg`.

Note - incremental extract
: Suppose you have already performed an extract using the above configuration file. At a later time, you have made some changes to some of the files in the source directory. Re-running the extract system on the same configuration will trigger an incremental extract. In an incremental extract, the system will update only those files that are modified. If the last modified time (or last commit revision) of a source file in the current extract differs from that in the previous extract, the system will attempt a content comparison. The system updates the destination only if the content and/or file access permission of the source differs from that of the destination.

Extract from a Subversion URL

The next example demonstrates how to extract from a Subversion repository URL:

```
# Example 2
# ------------------------------------------------------------------------
cfg::type           ext                     # line 1
cfg::version        1.0                     # line 2
                                            # line 3
dest                $PWD                    # line 4
                                            # line 5
repos::var::base    svn://server/var/trunk  # line 6
revision::var::base 1234                    # line 7
                                            # line 8
expsrc::var::base   code                    # line 9
```

- *line 1-5*: same as example 1.
- *line 6*: the line declares the repository location of the `base` branch of the `var` package to be the Subversion URL `svn://server/var/trunk`.
- *line 7*: the label `REVISION::<pck>::<branch>` declares the revision of the repository associated with the package <pck> in the branch <branch>. The current line tells the extract system to use revision 1234 of `svn://server/var/trunk`. It is worth noting that the declared revision must be a revision when the declared branch exists. The actual revision used is the last changed revision of the declared one. If the revision is not declared, the default is to use the last changed revision at the HEAD of the branch.
- *line 8*: comment line, ignored.
- *line 9*: the line declares an expandable source directory in the repository `svn://server/var/trunk`.

Invoking the extract system using the above configuration file will extract all sub-directories under `svn://server/var/trunk/code` to `$PWD/src/var/code`. It will write a build configuration file to `$PWD/cfg/bld.cfg`. The configuration used for this extract will be written to the configuration file at `$PWD/cfg/ext.cfg`.

EXPSRC or SRC?

> So far, we have only declared source directories using the `EXPSRC` statement, which stands for *expandable source directory*. A source directory declared using this statement will trigger the system to search recursively for any sub-directories under the declared one. Any sub-directories containing regular source files will be included in the extract. Symbolic links, hidden files and empty directories (or those containing only symbolic links and/or hidden files) are ignored.
>
> If you do not want the system to search for sub-directories underneath your declared source directory, you can declare your source directory using the `SRC` statement. The `SRC` statement is essentially the same as `EXPSRC` except that it does not trigger the automatic recursive search for source directories. In fact, the system implements the `EXPSRC` statement by expanding it into a list of `SRC` statements.

Package and sub-package

> The second field of a repository, revision or source directory declaration label is the name of the container package. It is a name selected by the user to identify the system or project he/she is working on. (Therefore, it is often sensible to choose an identifier that matches the name of the project or system.) The package name provides a unique namespace for a file container. Source directories are automatically arranged into sub-packages, using the names of the sub-directories as the names of the sub-packages. For example, the declaration at line 9 in example 2 will put the source directory in the `var/code` sub-package automatically.

Note that, in additional to slash `/`, double colon `::` and double underscore __ (internal only) also act as delimiters for package names. Please avoid using them for naming your files and directories.

You can declare a sub-package name explicitly in your source directory statement. For example, the following two lines are equivalent:

```
src::var::base                      code/VarMod_Surface
src::var/code/VarMod_Surface::base  code/VarMod_Surface
```

Explicit sub-package declaration should not be used normally, as it requires a lot more typing (although there are some situations where it can be useful, e.g. if you need to re-define the package name).

Currently, the extract system only supports non-space characters in the package name, as the space character is used as a delimiter between the declaration label and its value. If there are spaces in the path name to a file or directory, you should explicity re-define the package name of that path to a package name with no space using the above method. However, we recommend that only non-space characters are used for naming directories and files to make life simpler.

The expanded extract configuration file

At the end of a successful extract, the configuration used by the current extract is written in `cfg/ext.cfg` under the extract destination root. This file is an *expanded* version of the original, with changes in the following declarations:

- All revision keywords are converted into revision numbers.
- If a revision is not defined for a repository, it is set to the corresponding revision number of the HEAD revision.
- All URL keywords are converted into the full URLs.
- All `EXPSRC` declarations are expanded into `SRC` declarations.
- All other variables are expanded.

With this file, it should be possible for a later extract to re-create the current configuration even if the contents of the repository have changed. (This applies only to code stored in the repository.)

Mirror code to an alternate location

The next example demonstrates how to extract from a repository and mirror the code to an alternate location. It is essentially the same as example 2, except that it has three new lines to describe how the system can mirror the extracted code to an alternate location.

```
# Example 3
# ------------------------------------------------------------------------
cfg::type           ext
cfg::version        1.0

dest                $PWD

rdest::machine      tx01                          # line 6
rdest::logname      frva                          # line 7
rdest               /scratch/frva/extract/example3 # line 8

repos::var::base    svn://server/var/trunk
revision::var::base 1234

expsrc::var::base   code
```

Here is an explanation of what each line does:

- *line 6*: `RDEST::MACHINE` declares the target machine to which the code will be mirrored. The example mirrors the code to the machine named `tx01`.
- *line 7*: `RDEST::LOGNAME` declares the user name of the target machine, to which the user has login access. If this is not declared, the system uses the login name of the current user on the local machine.
- *line 8*: `RDEST` declares the root directory of the alternate destination, where the mirror version of the extract will be sent.

Invoking the extract system on the above configuration will trigger an extract similar to that given in example 2, but it will also attempt to mirror the contents at `$PWD/src/var/code` to `/scratch/frva/extract/example3/src` on the alternate destination. It will also mirror the expanded extract configuration file `$PWD/cfg/ext.cfg` to `/scratch/frva/extract/example3/cfg/ext.cfg` and `$PWD/cfg/bld.cfg` to `/scratch/frva/extract/example3/cfg/bld.cfg`. It is also worth noting that the content of the build configuration file will be slightly different, since it will include directory names appropriate for the alternate destination.

Note - mirroring command

> The extract system currently supports `rdist` and `rsync` as its mirroring tool. The default is `rsync`. To use `rdist` instead of `rsync`, add the following line to your extract configuration file:
>
> ```
> rdest::mirror_cmd  rdist
> ```
>
> If `rsync` is used to mirror an extract, the system needs to issue a separate remote shell command to create the container directory of the mirror destination. The default is to issue a shell command in the form `ssh -n -oBatchMode=yes LOGNAME@MACHINE mkdir -p DEST`. The following declarations can be used to modify the command:
>
> ```
> # Examples using the default settings:
> rdest::rsh_mkdir_rsh         ssh
> rdest::rsh_mkdir_rshflags    -n -oBatchMode=yes
> rdest::rsh_mkdir_mkdir       mkdir
> rdest::rsh_mkdir_mkdirflags  -p
> ```
>
> In addition, the default `rsync` shell command is `rsync -a --exclude='.*' --delete-excluded --timeout=900 --rsh='ssh -oBatchMode=yes' SOURCE DEST`. The following declarations can be used to modify the command:
>
> ```
> # Examples using the default settings:
> rdest::rsync       rsync
> rdest::rsyncflags  -a --exclude='.*' --delete-excluded --timeout=900 \
>                    --rsh='ssh -oBatchMode=yes'
> ```

Advanced Usage

Extract from multiple repositories

So far, we have only extracted from a single location. The extract system is not much use if that is the only thing it can do. In fact, the extract system supports extract of multiple source directories from multiple branches in multiple repositories. The following configuration file is an example of how to extract from multiple repositories:

```
# Example 4
# ------------------------------------------------------------------------
cfg::type           ext
cfg::version        1.0

dest                $PWD

repos::var::base    fcm:var_tr              # line 6
repos::ops::base    fcm:ops_tr              # line 7
repos::gen::base    fcm:gen_tr              # line 8

revision::gen::base 2468                    # line 10

expsrc::var::base   src/code                    # line 12
expsrc::var::base   src/scripts                 # line 13
expsrc::ops::base   src/code                    # line 14
src::gen::base      src/code/GenMod_Constants   # line 15
src::gen::base      src/code/GenMod_Control     # line 16
src::gen::base      src/code/GenMod_FortranIO   # line 17
src::gen::base      src/code/GenMod_GetEnv      # line 18
src::gen::base      src/code/GenMod_ModelIO     # line 19
src::gen::base      src/code/GenMod_ObsInfo     # line 20
src::gen::base      src/code/GenMod_Platform    # line 21
src::gen::base      src/code/GenMod_Reporting   # line 22
src::gen::base      src/code/GenMod_Trace       # line 23
src::gen::base      src/code/GenMod_UMConstants # line 24
src::gen::base      src/code/GenMod_Utilities   # line 25
```

Here is an explanation of what each line does:

- *line 6-8*: these lines declare the repositories for the `base` branches of the `var`, `ops` and `gen` packages respectively. It is worth noting that the values of the declarations are no longer Subversion URLs but are FCM URL keywords. These keywords are normally declared in the central configuration file of the FCM system, and will be expanded into the corresponding Subversion URLs by the FCM system. For further information on URL keywords, please see Code Management System > Using Subversion > Basic Command Line Usage > Repository & Revision Keywords.
- *line 10*: this line declares the revision number for the `base` branch of the `gen` package, i.e. for the `fcm:gen_tr` repository. It is worth noting that the revision numbers for the `var` and `ops` packages have not been declared. By default, their revision numbers will be set to the last changed revision at the HEAD.
- *line 12-14*: these line declares the source directories for the `base` branches of the `var` and `ops` packages. For the `var` package, we are extracting everything from the `code` and the `scripts` sub-directory. For the `ops` package, we are extracting everything from the `code` directory.
- *line 15-25*: these line declares the source directories for the `base` branch of the `gen` package. The source directories declared will not be searched for sub-directories underneath the declared directories.

We shall end up with a directory tree such as:

```
$PWD
   |
   |--- cfg
   |      |
   |      |--- bld.cfg
   |      |--- ext.cfg
   |
   |--- src
          |
          |--- gen
          |      |
```

```
|      |--- code
|              |
|              |--- GenMod_Constants
|              |--- GenMod_Control
|              |--- GenMod_FortranIO
|              |--- GenMod_GetEnv
|              |--- GenMod_ModelIO
|              |--- GenMod_ObsInfo
|              |--- GenMod_Platform
|              |--- GenMod_Reporting
|              |--- GenMod_Trace
|              |--- GenMod_UMConstants
|              |--- GenMod_Utilities
|
|--- ops
|      |
|      |--- code
|              |
|              |--- ...
|
|--- var
       |
       |--- code
       |       |
       |       |--- ...
       |
       |--- scripts
               |
               |--- ...
```

Note - revision number

As seen in the above example, if a revision number is not specified for a repository URL, it defaults to the last changed revision at the HEAD of the branch. The revision number can also be declared in other ways:

- Any revision arguments acceptable by Subversion are allowed. You can use a valid revision number, a date between a pair of curly brackets (e.g. `{2005-05-01T12:00}`) or the keyword HEAD. However, please do not use the keywords BASE, COMMITTED or PREV as these are reserved for working copy only.
- FCM revision keywords are allowed. These must be defined for the corresponding repository URLs in either the central or the user FCM configuration file. For further information on revision keywords, please see Code Management > Using Subversion > Basic Command Line Usage > Repository & Revision Keywords.
- Do not use the keyword USER, as it is used internally by the extract system.

If a revision number is specified for a branch, the actual revision used by the extract system is the last changed revision of the branch, which may differ from the declared revision. While this behaviour is useful in most situations, some users may find it confusing to work with. It is possible to alter this behaviour so that extract will fail if the declared revision does not correspond to a changeset of the declared branch. Make the following declaration to switch on this checking:

```
revmatch  true
```

Extract from multiple branches

We have so far dealt with a single branch in any package. The extract system can be used to *combine* changes from different branches of a package. An example is given below:

```
# Example 5
# ------------------------------------------------------------------------
cfg::type               ext
cfg::version            1.0

dest                    $PWD

repos::var::base        fcm:var_tr
repos::ops::base        fcm:ops_tr
repos::gen::base        fcm:gen_tr

revision::gen::base     2468

expsrc::var::base       src/code
expsrc::var::base       src/scripts
expsrc::ops::base       src/code
src::gen::base          src/code/GenMod_Constants
src::gen::base          src/code/GenMod_Control
src::gen::base          src/code/GenMod_FortranIO
src::gen::base          src/code/GenMod_GetEnv
src::gen::base          src/code/GenMod_ModelIO
src::gen::base          src/code/GenMod_ObsInfo
src::gen::base          src/code/GenMod_Platform
src::gen::base          src/code/GenMod_Reporting
src::gen::base          src/code/GenMod_Trace
src::gen::base          src/code/GenMod_UMConstants
src::gen::base          src/code/GenMod_Utilities

repos::var::branch1     fcm:var_br/frva/r1234_new_stuff   # line 27
repos::var::branch2     fcm:var_br/frva/r1516_bug_fix     # line 28
repos::ops::branch1     fcm:ops_br/opsrc/r3188_good_stuff # line 29
```

The configuration file in example 5 is similar to that of example 4 except for the last three lines. Here is an explanation of what they do:

- *line 27*: this line declares a repository URL for the `branch1` branch of the `var` package. From the URL of the branch, we know that the branch was created by the user `frva` based on the trunk at revision at 1234. The description of the branch is `branch1`. The following points are worth noting:
  - By declaring a new branch with the same package name to a previously declared branch, it is assumed that both branches reside in the same Subversion repository.
  - No revision is declared for this URL, so the default is used which is the last changed revision at the HEAD of the branch.
  - No source directory is declared for this URL. By default, if no source directory is declared for a branch repository, it will attempt to use the same set of source directories as the first declared branch of the package. In this case, the source directories declared for the `base` branch of the `var` package will be used.
- *line 28*: this line declares another branch called `branch2` for the `var` package. No source directory is declared for this URL either, so it will use the same set of source directories declared for the `base` branch.
- *line 29*: this line declares a branch called `branch1` for the `ops` package. It will use the same set of source directories declared for the `ops` package `base` branch.

When we invoke the extract system, it will attempt to extract from the first declared branch of a package, if the last changed revision of the source directory is the same in all the branches. However, if the last changed revision of the source directory differs for different branches, the system will attempt to obtain an extract priority list for each source directory, using the following logic:

1. The system looks for source directory packages from the first declared branch to the last declared branch.
2. The branch in which a source directory package is first declared is the `base` branch of the source directory package.
3. The last changed revision of a source directory package in a subsequently declared repository branch is compared with that of the base branch. If the last changed revision is the same as that of the base branch, the source directory of this branch is discarded. Otherwise, it is placed at the end of the extract priority list.

For the `var` package in the above example, let us assume that we have three source directory packages X, Y and Z under `code`, and their last changed revisions under `base` are 100. Let's say we have committed some changes to X and Z in the `branch1` branch at revision 102, and other changes to Y and Z in the `branch2` branch at revision 104, the extract priority lists for X, Y and Z will look like:

- X: base (100, base), branch1 (102), branch2 (100, discarded)
- Y: base (100, base), branch1 (100, discarded), branch2 (104)
- Z: base (100, base), branch1 (102), branch2 (104)

Once we have an extract priority list for a source directory, we can begin extracting source files in the source directory. The source directory of the base branch is extracted first, followed by that in the subsequent branches. If a source file in a subsequent branch has the same content as the that in the base branch, it is discarded. Otherwise, the following logic determines the branch to use:

1. If a source file is modified in only one subsequent branch, the source file in that branch is extracted.
2. If a source file is modified in two or more subsequent branches, but their modifications are the same, then the source file in the first modification is used.
3. If a source file is modified in two or more subsequent branches and their modifications differ, then the behaviour depends on the "conflict mode" setting, which can be `fail`, `merge` (default) and `override`. If the conflict mode is `fail`, the extract fails. If the conflict mode is `merge`, the system will attempt to merge the changes using a tool such as `diff3`. The result of the merge will be used to update the destination. The extract fails only if there are unresolved conflicts in the merge. (In which case, the conflict should be resolved using the version control system before re-running the extract system.) If the conflict mode is `override`, the change in the latest declared branch takes precedence, and the changes in all other branches will be ignored. The conflict mode can be changed using the `CONFLICT` declaration in the extract configuration file. E.g:

   ```
   conflict  fail
   ```

Once the system has established which source files to use, it determines whether the destination file is out of date or not. The destination file is out of date if it does not exist or if its content differs from the version of the source file we are using. The system only updates the destination if it is considered to be out of date.

The extract system can also combine changes from branches in the Subversion repository and the local file system. The limitation is that there can only be one branch from the local file system. (By convention, the branch is named `user`.)

It is also worth bearing in mind that the `user` branch always takes precedence over branches residing in Subversion repositories. Hence, source directories from a `user` branch are always placed at the end of the extract priority list.

Extracting from a mixture of Subversion repository and local file system is demonstrated in the next example.

```
# Example 6
# ------------------------------------------------------------------------
cfg::type               ext
cfg::version            1.0

dest                    $PWD

repos::var::base        fcm:var_tr
repos::ops::base        fcm:ops_tr
repos::gen::base        fcm:gen_tr

revision::gen::base     2468

expsrc::var::base       src/code
expsrc::var::base       src/scripts
expsrc::ops::base       src/code
src::gen::base          src/code/GenMod_Constants
src::gen::base          src/code/GenMod_Control
src::gen::base          src/code/GenMod_FortranIO
src::gen::base          src/code/GenMod_GetEnv
src::gen::base          src/code/GenMod_ModelIO
src::gen::base          src/code/GenMod_ObsInfo
src::gen::base          src/code/GenMod_Platform
src::gen::base          src/code/GenMod_Reporting
src::gen::base          src/code/GenMod_Trace
src::gen::base          src/code/GenMod_UMConstants
src::gen::base          src/code/GenMod_Utilities

repos::var::branch1     fcm:var_br/frva/r1234_new_stuff
repos::var::branch2     fcm:var_br/frva/r1516_bug_fix
repos::ops::branch1     fcm:ops_br/opsrc/r3188_good_stuff

repos::var::user        $HOME/var                       # line 31
repos::gen::user        $HOME/gen                       # line 32
```

Example 6 is similar to example 5 except that it is also extracting from local directories. Here is an explanation of the lines:

- *line 31-32*: these line declare the repositories for the `user` branches of the `var` and `gen` packages respectively. Both are local paths at the local file system. There are no declarations for source directories for the `user` branches, so they use the same set of source directories of the first declared branches, the `base` branches in both cases.

Note - the INC declaration

You have probably realised that the above examples have many repeated lines. To avoid having repeated lines in multiple extract configuration files, you can use `INC` declarations to include other extract configuration files. For example, if the configuration file of example 5 is stored in the file `$HOME/example5/ext.cfg`, line 1 to 29 of example 6 can be replaced with an `INC` declaration. Example 6 can then be written as:

```
inc                     $HOME/example5/ext.cfg

repos::var::user        $HOME/var
repos::gen::user        $HOME/gen
```

Note: the `INC` declaration supports the special environment variable *$HERE*. If this variable is already set in the environment, it acts as a normal environment variable. However, if it is not set, it will be expanded into the container directory of the current extract configuration file. This feature is particularly useful if you are including a hierarchy of extract configurations from files in the same container directory in a repository.

Inherit from a previous extract

All the examples above dealt with standalone extract, that is, the current extract is independent of any other extract. If a previous extract exists in another location, the extract system can inherit from this previous extract in your current extract. This works like a normal incremental extract, except that your extract will only contain the changes you have specified (compared with the inherited extract) instead of the full source directory tree. This type of incremental extract is useful in several ways. For instance:

- It is fast, because you only have to extract and mirror files that you have changed.
- The subsequent build will also be fast, since it will use incremental build.
- You do not need write access to the original extract. A system administrator can set up a stable version in a central account, which developers can then inherit from.
- You want an incremental extract, but you need to leave the original extract unmodified.

The following example is based on example 4 and example 6. The assumption is that an extract has already been performed at the directory `~frva/var/vn22.0` based on the configuration file in example 4.

```
# Example 7
# ------------------------------------------------------------------------
cfg::type               ext
cfg::version            1.0

dest                    $PWD

use                     ~frva/var/vn22.0                  # line 6

repos::var::branch1     fcm:var_br/frva/r1234_new_stuff   # line 8
repos::var::branch2     fcm:var_br/frva/r1516_bug_fix     # line 9
repos::ops::branch1     fcm:ops_br/opsrc/r3188_good_stuff # line 10

repos::var::user        $HOME/var                         # line 12
repos::gen::user        $HOME/gen                         # line 13
```

- *line 6*: this line replaces line 1 to 25 of example 6. It declares that the current extract should inherit from the previous extract located at `~frva/var/vn22.0`.

Running the extract system using the above configuration will trigger an incremental extract, as if you are running an incremental extract having modified the configuration file in example 4 to that of example 6. The only difference is that the original extract using the example 4 configuration will be left untouched at `~frva/var/vn22.0`, and the new extract will contain only the changes in the branches declared from line 8 to 13.

Note: extract inheritance allows you to add more branches to a package, but you should not redefine the `REPOS`, `REVISION`, `EXPSRC` or `SRC` declarations of a branch that is already declared (and already extracted) in the inherited extract. Although the system will not stop you from doing so, you may end up with an extract that does not quite do what it is supposed to do. For example, if the `base` branch in the `foo` package (`repos::foo::base`) is already defined and extracted in an extract you are inheriting from, you should not redefine any of the `*::foo::base` declarations in your current extract. However, you are free to add more branches for the same package with new labels (e.g. `repos::foo::b1`), and indeed new packages that are not already defined in the inherited extract (e.g. `repos::bar::base`).

If you are setting up an extract to be inherited, you do not have to perform a build. If you don't you will still gain the benefit of incremental file extract, but you will be performing a full build of the code.

Note - inherit and mirror

It is worth bearing in mind that `rdest::*` settings are not inherited. If mirroring is required in the inheriting extract, it will require its own set of `rdest::*` declarations.

The system will, however, assume that a mirrored version of the inherited extract is available for inheritance from the mirrored destination of the current extract.

E.g.: Consider an extract at `/path/to/inherited/` and an inheriting extract at `/path/to/current/`. If the former does not have a mirror, the latter should not have one either. If the former mirrors to `machine@/path/to/inherited/mirror/` and the latter mirrors to `machine@/path/to/current/mirror/`, the system will assume that the subsequent build at `machine@/path/to/current/mirror/` can inherit from the build at `machine@/path/to/inherited/mirror/`. This is illustrated below:

```
/path/to/current/       => at machine: /path/to/current/mirror/
use /path/to/inherited/ => at machine: use /path/to/inherited/mirror/
```

Extract - Build Configuration

Configuration settings for feeding into the build system can be declared through the extract configuration file using the `BLD::` prefix. Any line in an extract configuration containing a label with such a prefix will be considered a build system variable. At the end of a successful extract, the system strips out the `BLD::` prefix before writing these variables to the build configuration file. Some example entries are given between line 17 and 22 in the following configuration file:

```
# Example 8
# ------------------------------------------------------------------------
cfg::type           ext
cfg::version        1.0

dest                $PWD

repos::var::base    fcm:var_tr
repos::ops::base    fcm:ops_tr
repos::gen::base    fcm:gen_tr

revision::gen::base 2468

expsrc::var::base   src/code
expsrc::var::base   src/scripts
expsrc::ops::base   src/code
src::gen::base      src/code/GenMod_Constants
src::gen::base      src/code/GenMod_Control
src::gen::base      src/code/GenMod_FortranIO
src::gen::base      src/code/GenMod_GetEnv
src::gen::base      src/code/GenMod_ModelIO
src::gen::base      src/code/GenMod_ObsInfo
src::gen::base      src/code/GenMod_Platform
src::gen::base      src/code/GenMod_Reporting
src::gen::base      src/code/GenMod_Trace
src::gen::base      src/code/GenMod_UMConstants
src::gen::base      src/code/GenMod_Utilities

bld::target         VarProg_AnalysePF.exe   # line 27

bld::tool::fc       sxmpif90                # line 29
bld::tool::cc       sxmpic++                # line 30
bld::tool::ld       sxmpif90                # line 31
```

The above example is essentially the same as example 4, apart from the additional build configuration. The following is a simple explanation of what the lines represent: (For detail of the build system, please see the next chapter on The Build System.)

- *line 27*: the line declares a default target of the build.
- *line 29-31*: the lines declare the Fortran compiler, the C compiler and the linker respectively.

Note - use of variables

When you start using the extract system to define compiler flags for the build system, you may end up having to make a lot of long and repetitive declarations. In this case, you may want to define variables to replace the repetitive parts of the declarations.

Environment variables whose names contain only upper case latin alphabets, numbers and underscores can be referenced in a declaration value via the syntax `$NAME` or `${NAME}`. For example:

```
repos::um::base    ${HOME}/svn-wc/um
bld::tool::fflags  $MY_FFLAGS
```

You can define a user variable by making a declaration with a label that begins with a percent sign `%`. The value of a user variable remains in memory until the end of the current file is reached. You can reference a user variable in a declaration value via the syntax `%NAME` or `%{NAME}`. For example:

```
# Declare a variable %fred
%fred                     -Cdebug -eC -Wf,-init heap=nan stack=nan

bld::tool::fflags         %fred
# bld::tool::fflags       -Cdebug -eC -Wf,-init heap=nan stack=nan

bld::tool::fflags::foo    %fred -f0
# bld::tool::fflags::foo  -Cdebug -eC -Wf,-init heap=nan stack=nan -f0

bld::tool::fflags::bar    -w %fred
# bld::tool::fflags::bar  -w -Cdebug -eC -Wf,-init heap=nan stack=nan
```

Further to this, each declaration results in an internal variable of the same name and you can also refer to any of these internal variables in the same way. So, the example given above could also be written as follows:

```
bld::tool::fflags         -Cdebug -eC -Wf,-init heap=nan stack=nan
bld::tool::fflags::foo    %bld::tool::fflags -f0
bld::tool::fflags::bar    -w %bld::tool::fflags
```

Note - as-parsed configuration

If you use a hierarchy of `INC` declarations or variables, you may end up with a configuration file that is difficult to understand. To help you with this, the extract system generates an as-parsed configuration file at `cfg/parsed_ext.cfg` of the destination. The content of the as-parsed configuration file is what the extract system actually reads. It should contain everything in your original extract configuration file, except that all `INC` declarations, environment variables and user/internal variables are expanded.

Diagnostic verbose level

The amount of diagnostic messages generated by the extract system is normally set to a level suitable for normal everyday operation. This is the default diagnostic verbose level 1. If you want a minimum amount of diagnostic messages, you should set the verbose level to 0. If you want more

diagnostic messages, you can set the verbose level to 2 or 3. You can modify the verbose level in two ways. The first way is to set the environment variable *FCM_VERBOSE* to the desired verbose level. The second way is to invoke the extract system with the `-v <level>` option. (If set, the command line option overrides the environment variable.)

The following is a list of diagnostic output at each verbose level:

Level 0

- Report the time taken to extract the code.
- Report the time taken to mirror the code.
- If `rdist` is used to mirror the code, run the command with the `-q` option.

Level 1

- Everything at verbose level 0.
- Report the name of the extract configuration file.
- Report the location of the extract destination.
- Report date/time at the beginning of the extract step.
- If the revision specified for a repository branch is not its last changed revision, print an information statement to inform the user of the last changed revision of the branch.
- Summarises the destination status and the source status.
- Report date/time at the beginning of the mirror step.
- Report the location of the alternate destination.
- Report total time.

Level 2

- Everything at verbose level 1.
- If the revision specified for a repository branch is not current (i.e. the specified revision number is less than the revision number of the last commit revision), print an information statement to inform the user of the last commit revision of the branch.
- Report the detail of each change in the destination.
- If `rdist` is used to mirror the code, run the command without the `-q` option.

Level 3

- Everything at verbose level 2.
- Report all shell commands invoked by the extract system with timestamp.
- If `rdist` is used to mirror the code, print the `distfile` supplied to the command.
- If `rsync` is used to mirror the code, invoke the command with the `-v` option.

When Subversion Is Not Available

The extract system can still be used if Subversion is not available. Clearly, you can only use local repositories. However, you can still do incremental extract, mirror an extract to an alternate location, or combine code from multiple local repositories.

If you are using Subversion but your server is down then clearly there is little you can do. However, if you already have an extract then you can re-run `fcm extract` as long as the extract configuration file only refers to fixed revisions. If this is not the case then you can always use the expanded extract configuration file which can be found in `cfg/ext.cfg` under the extract destination root. This means that you can continue to makes changes to local code and do incremental extracts even whilst your Subversion server is down.

The Build System

Introduction

The build system analyses the directory tree containing a set of source code, processes the configuration, and invokes `make` to compile/build the source code into the project executables. In this chapter, we shall use many examples to explain how to use the build system. At the end of this chapter, you should be able to use the build system, either by defining the build configuration file directly or by using the extract system to generate a suitable build configuration file.

The Build Command

To invoke the build system, simply issue the command:

```
fcm build
```

By default, the build system searches for a build configuration file `bld.cfg` in `$PWD` and then `$PWD/cfg`. If a build configuration file is not found in these directories, the command fails with an error. If a build configuration file is found, the system will use the configuration specified in the file to perform the build. If you use the extract system to extract your source tree, a build configuration should be written for you automatically at the `cfg/` sub-directory of the destination root directory.

If the root directory of the build does not exist, the system performs a new full build at this directory. If a previous build already exists at this directory, the system performs an incremental build. If a full (fresh) build is required for whatever reason, you can invoke the build system using the `-f` option, (i.e. the command becomes `fcm build -f`). If you simply want to remove all the items generated by a previous build in the destination, you can invoke the build system using the `--clean` option.

The build system uses GNU `make` to perform the majority of the build. GNU `make` has a `-j jobs` option to specify the number of *jobs* to run simultaneously. Invoking the build system with the same option triggers this option when the build system invokes the `make` command. The argument to the option *jobs* must be an integer. The default is `1`. For example, the command `fcm build -j 4` will allow `make` to perform 4 jobs simultaneously.

For further information on the build command, please see FCM Command Reference > fcm build.

Basic Features

The build configuration file is the user interface of the build system. It is a line based text file. You can create your own build configuration file or you can use the extract system to create one for you. For a complete set of build configuration file declarations, please refer to the Annex: Declarations in FCM build configuration file.

Basic build configuration

Suppose we have a directory at `$HOME/example`. Its sub-directory at `$HOME/example/src` contains a source tree to be built. You may want to have a build configuration file `$HOME/example/cfg/bld.cfg`, which may contain:

```
# Example 1
# ------------------------------------------------------------------------
cfg::type     bld                           # line 1
cfg::version  1.0                           # line 2

dest          $HOME/example                 # line 4
```

```
target        foo.exe bar.exe               # line 6

tool::fc      ifort                         # line 8
tool::fflags  -O3                           # line 9
tool::cc      gcc                           # line 10
tool::cflags  -O3                           # line 11

tool::ldflags -O3 -L$(HOME)/lib -legg -lham # line 13
```

Here is an explanation of what each line does:

- *line 1*: the label `CFG::TYPE` declares the type of the configuration file. The value `bld` tells the system that it is a build configuration file.
- *line 2*: the label `CFG::VERSION` declares the version of the build configuration file. The current default is `1.0`. Although it is not currently used, if we have to change the format of the configuration file at a later stage, we shall be able to use this number to determine whether we are reading a file with an older format or one with a newer format.
- *line 4*: the label `DEST` declares the root directory of the current build.
- *line 6*: the label `TARGET` declares a list of *default* targets. The default targets of the current build will be `foo.exe` and `bar.exe`.
- *line 8*: the label `TOOL::FC` declares the Fortran compiler command.
- *line 9*: the label `TOOL::FFLAGS` declares the options to be used when invoking the Fortran compiler command.
- *line 10*: the label `TOOL::CC` declares the C compiler command.
- *line 11*: the label `TOOL::CFLAGS` declares the options to be used when invoking the C compiler command.
- *line 13*: the label `TOOL::LDFLAGS` declares the options to be used when invoking the linker command.

When we invoke the build system, it reads the above configuration file. It will go through various internal processes, such as dependency generations, to obtain the required information to prepare the `Makefile` of the build. (All of which will be described in later sections.) The `Makefile` of the build will be placed at `$HOME/example/bld`. The system will then invoke `make` to build the targets specified in line 6, i.e. `foo.exe` and `bar.exe` using the build tools specified between line 8 to line 13. On a successful build, the target executables will be sent to `$HOME/example/bin/`. The build system also creates a shell script called `fcm_env.sh` in `$HOME/example/`. If you source the shell script, it will export your *PATH* environment variable to search the `$HOME/example/bin/` directory for executables.

N.B. You may have noticed that the `-c` (compile to object file only) option is missing from the compiler flags declarations. This is because the option is inserted automatically by the build system, unless it is already declared.

N.B. You can declare the linker using `TOOL::LD`. If it is not specified, the default is to use the compiler command for the source file containing the main program.

Note - declaration of source files for build

> Source files do not have to reside in the `src/` sub-directory of the build root directory. They can be anywhere, but you will have to declare them using the label `SRC::<pcks>`, where `<pcks>` is the sub-package name in which the source belongs. If a directory is specified then the build system automatically searches for all source files in this directory. E.g.
>
> ```
> # Declare a source in the sub-package "foo/bar"
> src::foo/bar  $HOME/foo/bar
> ```

By default, the build system searches the `src/` sub-directory of the build root directory for source files. If all source files are already declared explicitly, you can switch off the automatic directory search by setting the `SEARCH_SRC` flag to false. E.g.

```
search_src  false
```

As mentioned in the previous chapter, the name of a sub-package <pcks> provides a unique namespace for a file. The name of a sub-package is a list of words delimited by a slash `/`. (The system uses the double colons `::` and the double underscores __ internally. Please avoid using `::` and __ for naming your files and directories.)

Currently, the build system only supports non-space characters in the package name, as the space character is used as a delimiter between the declaration label and its value. If there are spaces in the path name to a file or directory, you should explicity re-define the package name of that path to a package name with no space using the above method. However, we recommend that only non-space characters are used for naming directories and files to make life simple.

In the build system, the sub-package name also provides an *inheritance* relationship for sub-packages. For instance, we may have a sub-package called `foo/bar/egg`, which belongs to the sub-package `foo/bar`, which belongs to the package `foo`.

- If we declare a global build tool, it applies to all packages.
- If we declare a build tool for `foo`, it applies also to the sub-package `foo/bar` and `foo/bar/egg`.
- If we declare a build tool for `foo/bar`, it applies also to `foo/bar/egg`, but not to other sub-packages in `foo`.

Build configuration via the extract system

As mentioned earlier, you can obtain a build configuration file through the extract system. The following example is what you may have in your extract configuration in order to obtain a similar configuration as example 1:

```
# Example 2
# ------------------------------------------------------------------------
cfg::type          ext                              # line 1
cfg::version       1.0                              # line 2

dest               $HOME/example                    # line 4

bld::target        foo.exe bar.exe                  # line 6

bld::tool::fc      ifort                            # line 8
bld::tool::fflags  -O3                              # line 9
bld::tool::cc      gcc                              # line 10
bld::tool::cflags  -O3                              # line 11

bld::tool::ldflags -O3 -L$(HOME)/lib -legg -lham # line 13

# ... and other declarations for source locations ...
```

It is easy to note the similarities and differences between example 1 and example 2. Example 2 is an extract configuration file. It extracts to a destination root directory that will become the root directory of the build. Line 6 to line 13 are the same declarations, except that they are now prefixed with `BLD::`. In an extract configuration file, any lines prefixed with `BLD::` means that they are build configuration setting. These lines are ignored by the extract system but are parsed down to the output build configuration file, with the `BLD::` prefix removed. (Note: the `BLD::` prefix is optional for declarations in a build configuration file.)

N.B. If you use the extract system to mirror an extract to an alternate location, the extract system will assume that the root directory of the alternate destination is the root directory of the build, and that the build will be carried out in that destination.

Naming of executables

If a source file called `foo.f90` contains a main program, the default behaviour of the system is to name its executable `foo.exe`. The root name of the executable is the same as the original file name, but its file extension is replaced with `.exe`. The output extension can be altered by re-registering the extension for output EXE files. How this can be done will be discussed later in the sub-section File Type.

If you need to alter the full name of the executable, you can use the `EXE_NAME::` declaration. For example, the declaration:

```
bld::exe_name::foo  bar
```

will rename the executable of `foo.f90` from `foo.exe` to `bar`.

Note: the declaration label is `bld::exe_name::foo` (not `bld::exe_name::foo.exe`) and the executable will be named `bar` (not `bar.exe`).

Setting the compiler flags

As discussed in the first example, the compiler commands and their flags can be set via the `TOOL::` declarations. A simple `TOOL::FFLAGS` declaration, for example, alters the compiler options for compiling all Fortran source files in the build. If you need to alter the compiler options only for the source files in a particular sub-package, it is possible to do so by adding the sub-package name to the declaration label. For example, the declaration label `TOOL::FFLAGS::foo/bar` will ensure that the declaration only applies to the code in the sub-package `foo/bar`. You can even make declarations down to the individual source file level. For example, the declaration label `TOOL::FFLAGS::foo/bar/egg.f90` will ensure that the declaration applies only for the file `foo/bar/egg.f90`.

N.B. Although the prefix `TOOL::` and the tool names are case-insensitive, sub-package names are case sensitive in the declarations. Internally, tool names are turned into uppercase, and the sub-package delimiters are changed from the slash `/` (or double colons `::`) to the double underscores __. When the system generates the `Makefile` for the build, each `TOOL` declaration will be exported as an environment variable. For example, the declaration `tool::fflags/foo/bar` will be exported as `FFLAGS__foo__bar`.

N.B. `TOOL` declarations for sub-packages are only accepted by the system when it is sensible to do so. For example, it allows you to declare different compiler flags, linker commands and linker flags for different sub-packages, but it does not accept different compilers for different sub-packages. If you attempt to make a `TOOL` declaration for a sub-package that does not exist, the build system will exit with an error.

The following is an example setting in an extract configuration file based on example 2:

```
# Example 3
# ------------------------------------------------------------------------
cfg::type               ext
cfg::version            1.0

dest                    $HOME/example

bld::target             foo.exe bar.exe
```

```
bld::tool::fc          ifort
bld::tool::fflags      -O3    # line 9
bld::tool::cc          gcc
bld::tool::cflags      -O3

bld::tool::ldflags     -L$(HOME)/lib -legg -lham

bld::tool::fflags::ops -O1 -C # line 15
bld::tool::fflags::gen -O2    # line 16

# ... and other declarations for repositories and source directories ...
```

In the example above, line 15 alters the Fortran compiler flags for `ops`, so that all source files in `ops` will be compiled with optimisation level 1 and will have runtime error checking switched on. Line 16, alters the Fortran compiler flags for `gen`, so that all source files in `gen` will be compiled with optimisation level 2. All other Fortran source files will use the global setting declared at line 9, so they they will all be compiled with optimisation level 3.

Note - changing compiler flags in incremental builds

> Suppose you have performed a successful build using the configuration in example 3, and you have decided to change some of the compiler flags, you can do so by altering the appropriate flags in the build configuration file. When you trigger an incremental build, the system will detect changes in compiler flags automatically, and update only the required targets. The following hierarchy is followed:
>
> - If the compiler flags for a particular source file change, only that source file and any targets depending on that source file are re-built.
> - If the compiler flags for a container package change, only source files within that container package and any targets depending on those source files are re-built.
> - If the global compiler flags change, all source files are re-built.
> - If the compiler command changes, all source files are re-built.

N.B. For a full list of build tools declarations, please see Annex: Declarations in FCM build configuration file > list of tools.

Automatic Fortran 9X interface block

For each Fortran 9X source file containing standalone subroutines and/or functions, the system generates an interface file and sends it to the `inc/` sub-directory of the build root. An interface file contains the interface blocks for the subroutines and functions in the original source file. In an incremental build, if you have modified a Fortran 9X source file, its interface file will only be re-generated if the content of the interface has changed.

Consider a source file `foo.f90` containing a subroutine called `foo`. In a normal operation, the system writes the interface file to `foo.interface` in the `inc/` sub-directory of the build root. By default, the root name of the interface file is the same as that of the source file, and is case sensitive. You can change this behaviour using a `TOOL::INTERFACE` declaration. E.g.:

```
bld::tool::interface  program # The default is "file"
```

In such case, the root name of the interface file will be named in lower case after the first program unit in the file.

The default extension for an interface file is `.interface`. This can be modified through the input and output file type register, which will be discussed in a later section on File Type.

In most cases, we modify procedures without altering their calling interfaces. Consider another source file `bar.f90` containing a subroutine `bar`. If `bar` calls `foo`, it is good practice for `bar` to have an explicit interface for `foo`. This can be achieved if the subroutine `bar` has the following within its declaration section:

```
INCLUDE 'foo.interface'
```

The source file `bar.f90` is now dependent on the interface file `foo.interface`. This can make incremental build very efficient, as changes in the `foo.f90` file will not normally trigger the re-compilation of `bar.f90`, provided that the interface of the `subroutine foo` remains unchanged. (However, the system is clever enough to know that it needs to re-link any executables that are dependent on the object file for the `subroutine bar`.)

By default, the system uses its own internal logic to extract the calling interfaces of top level subroutines and functions in a Fortran source file to generate an interface block. However, the system can also work with the interface generator `f90aib`, which is a freeware obtained from Fortran 90 texts and programs, assembled by Michel Olagnon at the French Research Institute for Exploitation of the Sea. To do so, you need to make a declaration in the build configuration file using the label `TOOL::GENINTERFACE`. As for any other `TOOL` declarations, you can attach a sub-package name to the label. The change will then apply only to source files within that sub-package. If `TOOL::GENINTERFACE` is declared to have the value `NONE`, interface generation will be switched off. The following are some examples:

```
# Example 4
# ------------------------------------------------------------------------
# This is an EXTRACT configuration file ...

# ... some other declarations ...

bld::tool::geninterface       f90aib # line 5
bld::tool::geninterface::bar  none   # line 6

# ... some other declarations ...
```

In line 5, the global interface generator is now set to `f90aib`. In line 6, by setting the interface generator for the package `bar` to the `none` keyword, no interface file will be generated for source files under the package `bar`.

Switching off the interface block generator can be useful in many circumstances. For example, if the interface block is already provided manually within the source tree, or if the interface block is never used by other program units, it is worth switching off the interface generator for the source file to speed up the build process.

Automatic dependency

The build system has a built-in dependency scanner, which works out the dependency relationship between source files, so that they can be built in the correct order. The system scans all source files of known types for all supported dependency patterns. Dependencies of source files in a sub-package are written in a cache, which can be retrieved for incremental builds. (In an incremental build, only changed source files need to be re-scanned for dependency information. Dependency information for other files are retrieved from the cache.) The dependency information is passed to the `make` rule generator, which writes the `Makefile`.

The `make` rule generator generates different `make` rules for different dependency types. The following dependency patterns are automatically detected by the current system:

- The `USE <module>` statement in a Fortran source file is the first pattern. The statement has two implications: 1) The current file compiles only if the module has been successfully compiled, and needs to be re-compiled if the module has changed. 2) The executable depending on the current file can only resolve all its externals by linking with the object file of the compiled module. The executable needs to be re-linked if the module and its dependencies has changed.
- The `INCLUDE '<name>.interface'` statement in a Fortran source file is the second pattern. (The default extension for an interface file is `.interface`. This can be modified through the input and output file type register, which will be discussed in a later section on File Type.) It has two implications: 1) The current file compiles only if the included interface file is in the INCLUDE search path, and needs to be re-compiled if the interface file changes. 2) The executable depending on the current file can only resolve all its externals by linking with the object file of the source file that generates the interface file. The executable needs to be re-linked if the source file (and its dependencies) associated with the interface file has changed. It is worth noting that for this dependency to work, the root <name> of the interface file should match with that of the source file associated with the interface file. (Please note that you can use pre-processor [#include "<name>.interface] instead of Fortran INCLUDE, but it will not work if you switch on the pre-processing stage, which will be discussed in a later section.)
- The `INCLUDE '<file>'` statement (excluding the INCLUDE interface file statement) in a Fortran source file is the third pattern. It has two implications: 1) The current file compiles only if the included file is in the INCLUDE search path, and needs to be re-compiled if the include file changes. 2) The executable needs to be linked with any objects the include file is dependent on. It needs to be re-linked if these objects have changed.
- The `#include '<file>'` statement in a Fortran/C source or header file is the fourth pattern. It has similar implications as the Fortran INCLUDE statement. However, they have to be handled differently because `#include` statements are processed by the pre-processor, which may be performed in a separate stage of the FCM build process. This will be further discussed in a later sub-section on Pre-processing.

If you want your code to be built automatically by the FCM build system, you should also design your code to conform to the following rules:

1. Single compilable program unit, (i.e. program, subroutine, function or module), per file.
2. Unique name for each compilable program unit.
3. Always supply an interface for subroutines and functions, i.e.:
   - Put them in modules.
   - Put them in the CONTAINS section within the main program unit.
   - Use interface files.
4. If interface files are used, it is good practise to name each source file after the program unit it contains. It will make life a lot simpler when using the Automatic Fortran 9X interface block feature, which has already been discussed in the previous section.
   - The problem is that, by default, the root name of the interface file is the same as that of the source file rather than the program unit. If they differ then the build system will create a dependency on the wrong object file (since the object files are named according to the program unit).
   - This problem can be avoided by changing the behaviour of the interface file generator to use the name of the program unit instead (using a `TOOL::INTERFACE` declaration).

Note - setting build targets

> The `Makefile` generated by the build system contains a list of targets that can be built. The build system allows you to build (or perform the actions of) any targets that are present in the generated `Makefile`. There are two ways to specify the targets to be built.

Firstly, you can use the `TARGET` declarations in your build configuration file to specify the default targets to be built. These targets will be set as dependencies of the `all` target in the generated `Makefile`, which is the default target to be built when `make` is invoked by FCM. It is worth noting that `TARGET` declarations are cumulative. A later declaration does not override an earlier one - it simply adds more targets to the list.

Alternatively, you can use the `-t` option when you invoke the `fcm build` command. The option takes an argument, which should be a colon `:` separated list of targets to be built. When the `-t` option is set, FCM invokes `make` to build these targets instead. (E.g. if we invoke the build system with the command `fcm build -t foo.exe:bar.exe`, it will invoke `make` to build `foo.exe` and `bar.exe`.)

If you do not specify any explicit targets, the system will search your source tree for main programs:

- If there are main programs in your source tree, they will be set as the default targets automatically.
- Otherwise, the default is to build the top level library archive containing objects compiled from the source files in the current source tree. (For more information on building library archives, please see the section on Creating library archives.)

Advanced Features

Further dependency features

Apart from the usual dependency patterns described in the previous sub-section, the automatic dependency scanner also recognises two special directives when they are inserted into a source file:

- The directive `DEPENDS ON: <object>` in a comment line of a Fortran/C source file: It states that the current file is dependent on the declared external object. The executable depending on the current file needs to link with this external object in order to resolve all its external references. It needs to be re-linked if the declared external object (and its dependencies) has changed.
- The directive `CALLS: <executable>` in a comment line of a script: It states that the current script is dependent on the declared executable file, which can be another script or a binary executable. The current script can only function correctly if the declared executable is found in the search path. This directive is useful to ensure that all dependent executables are built or copied to the correct path.

Another way to specify external dependency is to use the `EXE_DEP` declaration to declare extra dependencies. The declaration normally applies to all main programs, but if the the form `EXE_DEP::<target>` is used, it will only apply to <target>, (which must be the name of a main program target). If the declaration is made without a value, the main programs will be set to depend on all object files. Otherwise, the value can be supplied as a space delimited list of items. Each item can be either the name of a sub-package or an object target. For the former, the main programs will be set to depend on all object files within the sub-package. For the latter, the main programs will be set to depend on the object target. The following are some examples:

```
# Example 5
# ------------------------------------------------------------------------
cfg::type          ext
cfg::version       1.0

bld::exe_dep::foo.exe  foo/bar egg.o # line 4
bld::exe_dep                         # line 5
# ... some other declarations ...
```

Here is an explanation of what each line does:

- *line 4*: this line declares the dependency on the sub-package `foo/bar` and the object target `egg.o` for building the main program target `foo.exe`. The target `foo.exe` will now depends on all object files in the `foo/bar` sub-package as well as the object target `egg.o`.
- *line 5*: this line declares that all other main program targets will depend on all (non-program) object files in the build.

Note - naming of object files

> By default, object files are named with the suffix `.o`. For a Fortran source file, the build system uses the lower case name of the first program unit within the file to name its object file. For example, if the first program unit in the Fortran source file `foo.f90` is `PROGRAM Bar`, the object file will be `bar.o`. For a C source file, the build system uses the lower case root name of the source file to name its object file. For example, a C source file called `egg.c` will have its object file named `egg.o`.
>
> The reason for using lower case to name the object files is because Fortran is a case insensitive language. Its symbols can either be in lower or upper case. E.g. the `SUBROUTINE Foo` is the same as the `SUBROUTINE foo`. It can be rather confusing if the subroutines are stored in different files. When they are compiled and archived into a library, there will be a clash of namespace, as the Fortran compiler thinks they are the same. However, this type of error does not normally get reported. If `Foo` and `foo` are very different code, the user may end up using the wrong subroutine, which may lead to a very long debugging session. By naming all object files in lower case, this type of situation can be avoided. If there is a clash in names due to the use of upper/lower cases, it will be reported as warnings by the build system, (as *duplicated targets* for building `foo.o`).

It is realised that there are situations when an automatically detected dependency should not be written into the `Makefile`. For example, the dependency may be a standard module provided by the Fortran compiler, and does not need to be built in the usual way. In such case, we need to have a way to exclude this module during an automatic dependency scan.

The `EXCL_DEP` declaration can be used to do just that. The following extract configuration contains some examples of the basic usage of the `EXCL_DEP` declaration:

```
# Example 6
# ------------------------------------------------------------------------
cfg::type          ext
cfg::version       1.0

bld::excl_dep  USE::YourFortranMod             # line 4
bld::excl_dep  INTERFACE::HerFortran.interface # line 5
bld::excl_dep  INC::HisFortranInc.inc          # line 6
bld::excl_dep  H::TheirHeader.h                # line 7
bld::excl_dep  OBJ::ItsObject.o                # line 8

# ... some other declarations ...
```

Here is an explanation of what each line does:

- *line 4*: this line declares that the Fortran module `YourFortranMod` should be excluded. The value of each `EXCL_DEP` declaration has two parts. The first part is a label that is used to define the type of dependency to be excluded. For a full list of these labels, please see the dependency types table in the Annex: Declarations in FCM build configuration file. The label `USE` denotes a Fortran module. The second part of the label is the dependency itself. For instance, if a Fortran source file contains the line: `USE YourFortranMod`, the dependency scanner will ignore it.
- *line 5*: this line declares that the include statement for the Fortran 9X interface file `HerFortran.interface` should be excluded. The label `INTERFACE` denotes a Fortran INCLUDE statement for a Fortran 9X interface block file. For example, if a Fortran source file contains the line: `INCLUDE 'HerFortran.interface'`, the dependency scanner will ignore it.
- *line 6*: this line declares that the include statement for `HisFortranInc.inc` should be excluded. The label `INC` denotes a Fortran INCLUDE statement other than an INCLUDE statement for an interface block file. For example, if a Fortran source file contains the line: `INCLUDE 'HisFortranInc.inc'`, the dependency scanner will ignore it.
- *line 7*: this line declares that the header include statement `TheirHeader.h` should be excluded. The label `H` denotes a pre-processing #include statement. For example, if a source file contains the line: `#include 'TheirHeader.h'`, the dependency scanner will ignore it.
- *line 8*: this line declares that the external dependency for `ItsObject.o` should be excluded. The label `OBJ` denotes a compiled binary object. These dependencies are normally inserted into the source files as special comments. For example, if a source file contains the line: `! depends on: ItsObject.o`, the dependency scanner will ignore it.

An `EXCL_DEP` declaration normally applies to all files in the build. However, you can suffix it with the name of a sub-package, i.e. `EXCL_DEP::<pcks>`. In such case, the declaration will only apply while scanning for dependencies in the source files in the sub-package named <pcks>.

You can also exclude all dependency scan of a particular type. To do so, simply declare the type in the value. For example, if you do not want the build system to scan for the `CALLS: <executable>` directive in the comment lines of your scripts, you can make the following declaration:

```
bld::excl_dep  EXE
```

The opposite of the `EXCL_DEP` declaration is the `DEP::<pcks>` declaration, which you can use to add a dependency to a source file (in the package name `<pcks>`). The syntax of the declaration is similar to that of `EXCL_DEP`, but you must specify the package name of a source file for DEP declarations. Please also note that a `DEP` declaration only works if the particular dependency is supported for the particular source file - as it makes no sense, for example, to specify a USE dependency for a shell script.

If you need to switch off dependency checking completely, you can use the `NO_DEP` declaration. For example, to switch off dependency checking for all but the `foo/bar` sub-package, you can do:

```
bld::no_dep           true
bld::no_dep::foo/bar  false
```

Linking a Fortran executable with a BLOCKDATA program unit

If it is required to link Fortran executables with BLOCKDATA program units, you must declare the executable targets and the objects containing the BLOCKDATA program units using the `BLOCKDATA::<target>` declarations. For example, if `foo.exe` is an executable target depending on the objects of the BLOCKDATA program units `blkdata.o` and `fbk.o`, you will

make the following declarations:

```
bld::blockdata::foo.exe  blkdata fbk
```

If all your executables are dependent on `blkdata.o` and `fbk.o`, you will make the following declarations:

```
bld::blockdata  blkdata fbk
```

Creating library archives

If you are interested in building library archives, the build system allows you to do it in a relatively simple way. For each sub-package in the source tree, there is a target to build a library containing all the objects compiled from the source files (that are not main programs) within the sub-package. If the sub-package contains children sub-packages, the object files of the children will also be included recursively. By default, the library archive is named after the sub-package, in the format `lib<pcks>.a`. (For example, the library archive for the package `foo/bar/egg` will be named `libfoo__bar__egg.a` by default.) If you do not like the default name for the sub-package library, you can use the `LIB::<pcks>` declaration to rename it, as long as the new name does not clash with other targets. For example, to rename `libfoo__bar__egg.a` to `libham.a`, you will make the following declaration in your extract configuration file:

```
bld::lib::foo/bar/egg  ham
```

In addition to sub-package libraries, you can also build a global library archive for the whole source tree. By default, the library is named `libfcm_default.a`, but you can rename it using the `LIB` declaration as above. For example, to rename the library to `libmy-lib.a`, you will make the following declaration in your extract configuration file:

```
bld::lib  my-lib
```

When a library archive is created successfully, the build system will automatically generate the relevant exclude dependency configurations in the `etc/` sub-directory of the build root. You will be able to include these configurations in subsequent builds that utilise the library. The root names of the configuration files match those of the library archives that you can create in the current build, but the extension `*.a` is replaced with `*.cfg`. For example, the exclude dependency configuration for `libmy-lib.a` is `libmy-lib.cfg`.

Pre-processing

As most modern compilers can handle pre-processing, the build system leaves pre-processing to the compiler by default. However, it is recognised that there are code written with pre-processor directives that can alter the argument list of procedures and/or their dependencies. If a source file requires pre-processing in such a way, we have to pre-process before running the interface block generator and the dependency scanner. The `PP` declaration can be used to switch on this pre-processing stage. The pre-processing stage can be switched on globally or for individual sub-packages only. The following is an example, using an extract configuration file:

```
# Example 7
# ------------------------------------------------------------------------
cfg::type          ext
cfg::version       1.0

bld::pp::gen       true                  # line 4
bld::pp::var/foo   true                  # line 5

bld::tool::cppkeys GOOD WEATHER FORECAST # line 7
bld::tool::fppkeys FOO BAR EGG HAM       # line 8

# ... some other declarations ...
```

Here is an explanation of what each line does:

- *line 4-5*: these switches on the pre-processing stage for all sub-packages under `gen` and `var/foo`.
- *line 7*: this declares a list of pre-defined macros `GOOD`, `WEATHER` and `FORECAST` for pre-processing all C files.
- *line 8*: this declares a list of pre-defined macros `FOO`, `BAR`, `EGG` and `HAM` for pre-processing all Fortran files that require processing.

Source files requiring pre-processing may contain `#include` statements to include header files. For including a local file, its name should be embedded within a pair of quotes, i.e. `'file.h'` or `"file.h"`. If the header file is embedded within a pair of <file.h> angle brackets, the system will assume that the file can be found in a standard location.

The build system allows header files to be placed anywhere within the declared source tree. The system uses the dependency scanner, as described in the previous sub-section to scan for any header file dependencies. All source files requiring pre-processing and all header files are scanned. Header files that are required are copied to the `inc/` subdirectory of the build root, which is automatically added to the pre-processor search path via the `-I<dir>` option. The build system uses an internal logic similar to `make` to perform pre-processing. Header files are only copied to the `inc/` sub-directory if they are used in `#include` statements.

Unlike `make`, which only uses the timestamp to determine whether an item is out of date, the internal logic of the build system does this by inspecting the content of the file as well. In an incremental build, the pre-processed file is only updated if its content has changed. This avoids unnecessary updates (and hence unnecessary re-compilation) in an incremental build if the changed section of the code does not affect the output file.

Pre-processed code generated during the pre-processing stage are sent to the `ppsrc/` sub-directory of the build root. It will have a relative path that reflects the name of the declared sub-package. The pre-processed source file will have the same root name as the original source file. For C files, the same extension `.c` will be used. For Fortran files, the case of the extension will normally be dropped, e.g. from `.F90` to `.f90`.

Following pre-processing, the system will use the pre-processed source file as if it is the original source file. The interface generator will generate the interface file using the pre-processed file, the dependency scanner will scan the pre-processed file for dependencies, and the compiler will compile the pre-processed source.

The `TOOL::CPPKEYS` and `TOOL::FPPKEYS` declarations are used to pre-define macros in the C and Fortran pre-processor respectively. This is implemented by the build system using the pre-processor `-D` option on each word in the list. The use of these declarations are not confined to the pre-process stage. If any source files requiring pre-processing are left to the compiler, the declarations will be used to set up the commands for compiling these source files.

The `TOOL::CPPKEYS` and `TOOL::FPPKEYS` declarations normally applies globally, but like any other `TOOL` declarations, they can be suffixed with sub-package names. In such cases, the declarations will apply only to the specified sub-packages.

Note - changing pre-processor flags

> As for compiler flags, the build system detects changes in pre-processor flags (`TOOL::CPPFLAGS` and `TOOL::FPPFLAGS`) and macro definitions (`TOOL::CPPKEYS` and `TOOL::FPPKEYS`). If the pre-processor flags or the macro definitions have changed in an incremental build, the system will re-do all the necessary pre-processing. The following hierarchy is followed:
>
> - If the pre-processor flags or macro definitions for a particular source file change, only that source file will be pre-processed again.
> - If the pre-processor flags or macro definitions for a particular container package change, only source files within that container will be pre-processed again.
> - If the global pre-processor flags or macro definitions change, all source files will be pre-processed again.
> - If the pre-processor command changes, all source files are pre-processed again.

File type

The build system only knows what to do with an input source file if it knows what type of file it is. The type of a source file is normally determined automatically using one of the following three methods (in order):

1. If the file is named with an extension, its extension will be matched against a set of registered file extensions. If a match is found, the file type will be set according to the register.
2. If a file does not have an extension or does not match with a registered extension, its name is compared with a set of pre-defined patterns. If a match is found, the file type will be set according to the file type associated with the pattern.
3. If the above two methods failed and if the file is a text file, the system will attempt to read the first line of the file. If the first line begins with a `#!` pattern, the line will be compared with a set of pre-defined patterns. If a match is found, the file type will be set according to the file type associated with the pattern.

In addition to the above, if a file is a Fortran or C source file, the system will attempt to open the source file to determine whether it contains a main program, module (Fortran only) or just standalone procedures. All these information will be used later by the build system to process the source file.

The build system registers a file type with a set of type flags delimited by the double colons `::`. For example, a Fortran 9X source file is registered as `FORTRAN::FORTRAN9X::SOURCE`. (Please note that the order of the type flags in the list is insignificant. For example, `FORTRAN::SOURCE` is the same as `SOURCE::FORTRAN`.) For a list of all the type flags used by the build system, please see the input file extension type flags table in the Annex: Declarations in FCM build configuration file.

The following is a list of default input file extensions and their associated types:

`.f .for .ftn .f77`
: `FORTRAN::SOURCE` Fortran 77 source file (assumed to be fixed format)

`.f90 .f95`
: `FORTRAN::FORTRAN9X::SOURCE` Fortran 9X source file (assumed to be free format)

`.F .FOR .FTN .F77`
: `FPP::SOURCE` Fortran 77 source file (assumed to be fixed format) that requires pre-processing

`.F90 .F95`
: `FPP::FPP9X::SOURCE` Fortran 9X source file (assumed to be free format) that requires pre-processing

`.c`
: `C::SOURCE` C source file

`.h .h90`
: `CPP::INCLUDE` Pre-processor `#include` header file

`.o .obj`
: `BINARY::OBJ` Compiled binary object

`.exe`
: `BINARY::EXE` Binary executable

`.a`
: `BINARY::LIB` Binary object library archive

`.sh .ksh .bash .csh`
: `SHELL::SCRIPT` Unix shell script

`.pl .pm`
: `PERL::SCRIPT` Perl script

`.py`
: `PYTHON::SCRIPT` Python script

`.tcl`
: `TCL::SCRIPT` Tcl/Tk script

`.pro`
: `PVWAVE::SCRIPT` IDL/PVWave program

`.cfg`
: `CFGFILE` FCM configuration file

`.inc`
: `FORTRAN::FORTRAN9X::INCLUDE` Fortran INCLUDE file

`.interface`
: `FORTRAN::FORTRAN9X::INCLUDE::INTERFACE` Fortran 9X INCLUDE interface block file

N.B. The extension must be unique. For example, the system does not support the use of `.inc` files for both `#include` and Fortran `INCLUDE`.

The following is a list of supported file name patterns and their associated types:

`*Scr_* *Comp_* *IF_* *Suite_* *Interface_*`
: `SHELL::SCRIPT` Unix shell script, GEN-based project naming conventions

`*List_*`
: `SHELL::SCRIPT::GENLIST` Unix shell script, GEN list file

`*Sql_*`
: `SCRIPT::SQL` SQL script, GEN-based project naming conventions

The following is a list of supported `#!` line patterns and their associated types:

`*sh* *ksh* *bash* *csh*`
: `SHELL::SCRIPT` Unix shell script

`*perl*`
: `PERL::SCRIPT` Perl script

`*python*`
: `PYTHON::SCRIPT` Python script

`*tclsh* *wish*`
: `TCL::SCRIPT` Tcl/Tk script

The build system allows you to add or modify the register for input file extensions and their associated type using the `INFILE_EXT::<ext>` declaration, where <ext> is a file name extension without the leading dot. If file extension alone is insufficient for defining the type of your source file, you can use the `SRC_TYPE::<pcks>` declaration, (where <pcks> is the package name of the source file). For example, in an extract configuration file, you may have:

```
# Example 8
# ------------------------------------------------------------------------
cfg::type                 ext
cfg::version              1.0

bld::infile_ext::foo      CPP::INCLUDE                 # line 4
bld::infile_ext::bar      FORTRAN::FORTRAN9X::INCLUDE  # line 5
bld::src_type::egg/ham.f  FORTRAN::FORTRAN9X::INCLUDE  # line 6

# ... some other declarations ...
```

Here is an explanation of what each line does:

- *line 4*: this line registers the extension `.foo` to be of type `CPP::INCLUDE`. This means that any input files with `.foo` extension will be treated as if they are pre-processor header files.
- *line 5*: this line registers the extension `.bar` to be of type `FORTRAN::FORTRAN9X::INCLUDE`. This means that any input file with `.bar` extension will be treated as if they are Fortran 9X INCLUDE files.
- *line 6*: this line declares the type for the source file in the package `egg::ham.f` to be `FORTRAN::FORTRAN9X::INCLUDE`. Without this declaration, this file would normally be given the type `FORTRAN::SOURCE`.

The `INFILE_EXT` declarations deal with extensions of input files. There is also a `OUTFILE_EXT::<type>` declaration that deals with extensions of output files. The declaration is opposite that of `INFILE_EXT`. The file <type> is now declared with the label, and the extension is declared as the value. It is worth noting that `OUTFILE_EXT` declarations use very different syntax for <type>, and the declared extension must include the leading dot. For a list of output types used by the build system, please see the output file extension types table in the Annex: Declarations in FCM build configuration file. An example is given below:

```
# Example 9
# ------------------------------------------------------------------------
cfg::type                   ext
cfg::version                1.0

bld::outfile_ext::mod       .MOD   # line 4
bld::outfile_ext::interface .intfb # line 5

# ... some other declarations ...
```

Here is an explanation of what each line does:

- *line 4*: this line modifies the extension of compiled Fortran 9X module information files from the default `.mod` to `.MOD`.
- *line 5*: this line modifies the extension of INCLUDE Fortran 9X interface block files from the default `.interface` to `.intfb`.

N.B. If you have made changes to the file type registers, whether it is for input files or output files, it is always worth re-building your code in full-build mode to avoid unexpected behaviour.

Inherit from a previous build

As you can inherit from previous extracts, you can inherit from previous builds. The very same `USE` statement can be used to declare a build, which the current build will depend on. The only difference is that the declared location must contain a valid build configuration file. In fact, if you use the extract system to obtain your build configuration file, any `USE` declarations in the extract configuration file will also be `USE` declarations in the output build configuration file.

By declaring a previous build with a `USE` statement, the current build automatically inherits settings from it. The following points are worth noting:

- Build targets are not normally inherited. However, you can switch on inheritance of build targets using an `INHERIT::TARGET` declaration, such as:

  ```
  inherit::target  true
  ```

- The build root directory and its sub-directories of the inherited build are placed into the search paths. For example, if we have an inherited build at `/path/to/inherited`, and it is used by a build at `/path/to/my_build`, the search path of executable files will become `/path/to/my_build/bin:/path/to/inherited/bin`, so that the `bin/` sub-directory of the current build is searched before the `bin/` sub-directory of the inherited build. If two or more `USE` statements are declared, the `USE` statement declared last will have higher priority. For example, if the current build is *C*, and it USEs build *A* before build *B*, the search path will be `C:B:A`.
- Source files are inherited by default. If a source file is declared in the current build that has the same package name as a source file of the inherited build, it will override that in the inherited build. Any source files missing from the current build will be taken from the inherited build.

  You can switch off inheritance of source files using an `INHERIT::SRC` declaration. This declaration can be suffixed with the name of a sub-package. In such case, the declaration applies only to the inheritance of the sub-package. Otherwise, it applies globally. For example:

  ```
  # Switch off inheritance of source files in the gen sub-package
  inherit::src::gen  false
  ```

- `BLOCKDATA`, `DEP`, `EXCL_DEP`, `EXE_DEP`, `INFILE_EXT`, `LIB`, `OUTFILE_EXT`, `PP`, `TOOL` and `SRC_TYPE` declarations are automatically inherited. If the same setting is declared in the current incremental build, it overrides the inherited declaration.

As an example, suppose we have already performed an extract and build based on the configuration in example 2, we can set up an extract configuration file as follows:

```
# Example 10
# ------------------------------------------------------------------------
cfg::type            ext
cfg::version         1.0

use                  $HOME/example               # line 4

dest                 $HOME/example10             # line 6

bld::inherit::target true                        # line 8
bld::target          ham.exe egg.exe             # line 9

bld::tool::fflags    -O2 -w                      # line 11
bld::tool::cflags    -O2                         # line 12

# ... and other declarations for repositories and source directories ...
```

Here is an explanation of what each line does:

- *line 4*: this line declares a previous extract at `$HOME/example` which the current extract will inherit from. The same line will be written to the output build configuration file. The subsequent build will then inherit from the build at `$HOME/example`.
- *line 6*: this declares the destination root directory of the current extract, which will become the root directory of the current build. Search paths of the build sub-directories will be set automatically. For example, the search path for executable files created by the current build will be `$HOME/example10/bin:$HOME/example/bin`.
- *line 8*: this line switches on inheritance of build targets. The build targets in example 1, i.e. `foo.exe` and `bar.exe` will be built as part of the current build.
- *line 9*: this declares two new build targets `ham.exe` and `egg.exe` to be added to the inherited ones. The default build targets of the current build will now be `foo.exe`, `bar.exe`, `ham.exe` and `egg.exe`.
- *line 11-12*: these lines modify options used by the Fortran and the C compilers, overriding those inherited from example 1.

Build inheritance limitation: handling of include files

The build system uses the compiler/pre-processor's `-I` option to specify the search path for include files. For example, it uses the option to specify the `inc/` sub-directories of the current build and its inherited build.

However, some compilers/pre-processors (e.g. `cpp`) search for include files from the container directory of the source file before searching for the paths specified by the `-I` options. This behaviour may cause the build to behave incorrectly.

Consider a source file `egg/hen.c` that includes `fried.h`. If the directory structure looks like:

```
# Sources in inherited build:
egg/hen.c
egg/fried.h

# Sources in current build:
egg/fried.h
```

The system will correctly identify that `fried.h` is out of date, and trigger a re-compilation of `egg/hen.c`. However, if the compiler searches for the include files from the container directory of the source file first, it will wrongly use the include file in the inherited build instead of the current one.

Some compilers (e.g. `gfortran`) do not behave this way and others (e.g. `ifort`) have options to prevent include file search in the container directory of the source file. If you are using such a compiler you can avoid the problem for Fortran compilation although this does not fix the problem entirely if you have switched on the pre-processing stage. Otherwise you may have to work around the problem, (e.g. by making a comment change in the source file, or by not using an inherited build at all).

Building data files

While the usual targets to be built are the executables associated with source files containing main programs, libraries or scripts, the build system also allows you to build *data* files. All files with no registered type are considered to be *data* files. For each container sub-package, there is an automatic target for copying all *data* files to the `etc/` sub-directory of the build root. The name of the target has the form `<pcks>.etc`, where <pcks> is the name of the sub-package (with package names delimited by the double underscore __). For example, the target name for sub-package `foo/bar` is `foo__bar.etc`. This target is particularly useful for copying, say, all namelists in a

sub-package to the `etc/` sub-directory of the build root.

At the end of a successful build, if the `etc/` sub-directory is not empty, the `fcm_env.sh` script will export the environment variable *FCM_ETCDIR* to point to the `etc/` sub-directory. You should be able to use this environment variable to locate your data files.

Diagnostic verbose level

The amount of diagnostic messages generated by the build system is normally set to a level suitable for normal everyday operation. This is the default diagnostic verbose level 1. If you want a minimum amount of diagnostic messages, you should set the verbose level to 0. If you want more diagnostic messages, you can set the verbose level to 2 or 3. You can modify the verbose level in two ways. The first way is to set the environment variable *FCM_VERBOSE* to the desired verbose level. The second way is to invoke the build system with the `-v <level>` option. (If set, the command line option overrides the environment variable.)

The following is a list of diagnostic output at each verbose level:

Level 0
- Report the time taken at the end of each stage of the build process.
- Run the `make` command in silent mode.

Level 1
- Everything at verbose level 0.
- Report the name of the build configuration file.
- Report the location of the build destination.
- Report date/time at the beginning of each stage of the build process.
- Report removed directories.
- Report number of pre-processed files.
- Report number of generated F9X interface files.
- Report number of source files scanned for dependencies.
- Report name of updated `Makefile`.
- Print compiler/linker commands.
- Report total time.

Level 2
- Everything at verbose level 1.
- For incremental build in archive mode, report the commands used to extract the archives.
- Report creation and removal of directories.
- Report pre-processor commands.
- Print compiler/linker commands with timestamps.

Level 3
- Everything at verbose level 2.
- Report update of dummy files.
- Report all shell commands.
- Report pre-processor commands with timestamps.
- Report any F9X interface files generated.
- Report number of lines and number of automatic dependencies for each source file which is scanned.
- Run `make` on normal mode (as opposed to silent mode).
- Report start date/time and time taken of `make` commands.

Overview of the build process

The FCM build process can be summarised in five stages. Here is a summary of what is done in each stage:

1. *Parse configuration and setup destination*: in this pre-requisite stage, the build system parses the configuration file. The `src/` sub-directory is searched recursively for source files. For full builds, it ensures that the sub-directories and files created by the build system are removed. If you invoke `fcm build` with a `--clean` option, the system will not go any further.
2. *Setup build*: in this first stage, the system determines whether any settings have changed by using the cache. If so, the cache is updated with the current settings.
3. *Pre-process*: if any files in any source files require pre-processing, they will be pre-processed at this stage. The resulting pre-processed source files will be sent to the `ppsrc/` sub-directory of the build root.
4. *Generate dependency*: the system scans source files of registered types for dependency information. For an incremental build, the information is only updated if a source file is changed. The system then uses the information to write a `Makefile` for the main build.
5. *Generate interface*: if there are Fortran 9X source files with standalone subroutines and functions, the build system generates interface blocks for them. The result of which will be written to the interface files in the `inc/` sub-directory of the build root.
6. *Make*: the system invokes `make` on the `Makefile` generated in the previous stage to perform the main build. Following a build, the *root* directory of the build may contain the following sub-directories (empty ones are removed automatically at the end of the build process):

   `.cache/.bld/`
   : Cache files, used internally by FCM.

   `bin/`
   : Executable binaries and scripts.

   `cfg/`
   : Configuration files.

   `done/`
   : Dummy *done* files used internally by the `Makefile` generated by FCM.

   `etc/`
   : Miscellaneous data files.

   `flags/`
   : Dummy *flags* files used internally by the `Makefile` generated by FCM.

   `inc/`
   : Include files, such as `*.h`, `*.inc`, `*.interface`, and `*.mod`.

   `lib/`
   : Object library archives.

   `obj/`
   : Compiled object files.

   `ppsrc/`
   : Source directories with pre-processed files.

   `src/`
   : Source directories. This directory is not changed by the build system.

   `tmp/`
   : Temporary objects and binaries. Files generated by the compiler/linker may be left here.

System Administration

Introduction

This chapter provides an administration guide for managers of projects or systems which are using FCM.

Note that, where this section refers to the *FCM team* this applies only to Met Office users. External users will either need to refer to the equivalent team within their organisation or will need to perfom these tasks themselves.

Subversion

Repository design

The FCM system assumes that each project directory has sub-directories called *trunk*, *branches* and *tags* (Subversion recommends it but doesn't insist on it). We recommend that each project within a Subversion repository is in a sub-directory of the repository root.

```
<root>
    |
    |-- <project 1>
    |       |
    |       |-- trunk
    |       |-- branches
    |       |-- tags
    |
    |-- <project 2>
    |       |
    |       |-- trunk
    |       |-- branches
    |       |-- tags
    |
    |-- ...
```

In theory you could also have the project as the root directory or several levels below the root directory. However, this is not tested and could cause problems with some `fcm` commands so is best avoided.

You will need to decide whether to use a single project tree for your system or whether to use multiple projects.

Advantages of a single project tree:

- Changes to any part of the system can always be committed as a single logical changeset. If you split your system into multiple projects then you may have occasions when a logical change involves more than one project and hence requires multiple commits (and branches).

Disadvantages of a single project tree:

- If you have a large system then your working copies may become very large and unwieldy. Basic commands such as `checkout` and `status` can become frustratingly slow if your working copy is too large.
- Depending on how you work, you may end up doing lots more merges of files that are unrelated to your work.

One common approach is to split the admin type files (things that only the system manager should need to touch) into a separate project from the core system files (which all the developers need access to). If you include any large data files under version control you may also want to use a separate project for them to avoid making your working copies very large when editing code.

Note that there is often no obvious right or wrong answer so you just have to make a decision and see how it works out. You can always re-arrange your repository in the future (although be aware that this will break any changes being prepared on branches at the time).

You also need to decide whether your system requires its own repository (or multiple repositories) or whether it can share with another system.

- The main disadvantage of having separate repositories for each system is the maintenance overhead (although this is almost all automated by the FCM team so is not a big deal).
- We normally configure a single Trac system per repository. If the repository contains multiple systems then it makes it difficult to use the Trac milestones to handle system releases. However, Trac now supports restricting itself to a sub-directory within a repository so, again, this is not a big deal.
- If you share a repository with other systems then your revision numbers can increase even when there are no changes to your system. This doesn't matter but some people don't like it.

For simplicity, in most cases you will probably want your own repository for your system.

You will not normally want to have multiple repositories for a system. One exception may be if you are storing large data files where you might not want to keep all the old versions for ever. Removing old versions can't be done without changing all the revision numbers which would mess up all your code history and Trac tickets. Storing the large data files in a separate repository reduces the impact if you do decide to remove old versions in the future. One disadvantage of this approach is that, for the moment at least, Trac only handles one repository so you will need a separate Trac system for the data files.

For further details please see the section Planning Your Repository Organization from the Subversion book.

Creating a repository

Normally the FCM team will help you to set up your initial repository. However, it is quite simple if you need to do it yourself. First you need to issue the command `svnadmin create /path/to/repos`. This creates an empty repository which is now ready to accept an initial import. To do so, you should create a directory tree in a suitable location, and issue the `svn import` command. At the top level of your directory tree should be the project directories. Each project should then contain three directories `trunk`, `branches` and `tags`. The directories `branches` and `tags` should be empty. The directory `trunk` should contain your source files in a directory structure you have chosen. For example, if your directory tree is located at `$HOME/foo`, you will do the following to import it to a new repository:

```
(SHELL PROMPT)$ svnadmin create FOO_svn
(SHELL PROMPT)$ svn import $HOME/foo file://$PWD/FOO_svn -m 'Initial import'
Adding         FOO
Adding         FOO/trunk
Adding         FOO/trunk/doc
Adding         FOO/trunk/doc/hello.html
Adding         FOO/trunk/doc/world.html
Adding         FOO/trunk/src
Adding         FOO/trunk/src/bar
Adding         FOO/trunk/src/bar/egg.f90
Adding         FOO/trunk/src/bar/ham.f90
```

```
Adding         FOO/branches
Adding         FOO/tags

Committed revision 1.
```

Note that the `svnadmin` command takes a *PATH* as an argument, as opposed to a URL for the `svn` command.

For further details please see the section Planning Your Repository Organization from the Subversion book.

Access control

You will not normally want to allow anonymous write access to your repository since this means that changes do not get identified with a userid. Therefore system managers need to provide the FCM team with a list of users who should have write access to their repositories. You may also want to arrange passwords for these users although this is only necessary if you need to prevent malicious access. Further restrictions such as preventing anonymous read access or restricting write access to the trunk to a limited set of users can be arranged if necessary.

Repository hosting

The FCM team will organise the hosting of your repository. A number of facilities will be set up for you as standard.

- Your repository will be set up on a central FCM server and access will be provided via `svnserve` (which we use in preference to *Apache* for performance reasons). The FCM team will advise you of the URL.
- Each night a full backup of your repository will be taken. An integrity check will also be performed using the `svnadmin verify` command.
- Standard hook scripts will be set up to handle the following post-commit tasks:
  - Each time a changeset is successfully committed an incremental dump of the new revision is taken. This would allow the repository to be recovered should the live repository become corrupted for whatever reason.
  - A file is updated which records the latest revision of your repository. This can be used by system managers to regularly check for new commits in a cron job and perform any required actions (updating files on a remote platform for instance). The FCM team can advise you of the location of this file and show you some example scripts which make use of it.
- Standard hook scripts will be set up to handle the following tasks for changes in revision properties (pre-revprop-change/post-revprop-change):
  - If a user attempts to modify the log message of a changeset and he/she is not the original author of the changeset, the pre-revprop-change hook script will e-mail the original author. You can also set up a watch facility to monitor changes of log messages that affect particular paths in the repository. (See the next sub-section for details.)
  - The post-revprop-change hook script updates the Trac SQLite database following a successful change in the log message.

Additional hook scripts can be put in place if you have a requirement. The use of hook scripts is discussed in the section Repository Creation and Configuration from the Subversion book.

Note that if you want to use a Subversion repository for your own individual use there is no need to get the FCM team to host it. You can simply create your repository and then use a `file://` URL to access it.

Watching changes in log messages

You can set up a watch facility to monitor changes in revision log messages in your repository. An obvious use of this facility is for system managers to monitor changes of log messages affecting the trunks of their projects. To set up the facility, you will need to add a `watch.cfg` file to the root of your repository. (To avoid checking out the whole repository, including every branch, make sure that you checkout the root of your repository non-recursively, i.e. `fcm checkout -N URL`.) The `watch.cfg` file is an INI-type file, with the basic format:

```
[repos_base]

path/in/repos = list,of,watchers
```

For example, if your repository is `svn://fcm1/FCM_svn/`, and you want set up particular users to monitor changes of the log messages affecting several areas within the repository, you may have something like this:

```
[FCM_svn]

FCM/trunk/src             = fred,bob
FCM/trunk/doc             = fred,bob,alice
FCM/branches/dev/*/*/src = fred
```

Later on, if `dave` attempts to modify the log message of a changeset that affects the path `FCM/trunk/src`, `fred` and `bob` will both be notified by e-mail.

Trac

Trac configuration

Normally the FCM team will set up your Trac system for you (using a script which helps automate the configuration). This section describes some things you may wish to be configured. This can be done when the Trac system is set up or later if you are unsure what you will require at first.

Access control

You will not normally want to allow anonymous users to make changes to your Trac system since this means that changes may not get identified with a userid. The FCM team will normally set up your Trac system such that any authenticated users can make changes. Further restrictions such as restricting write access to named accounts or preventing anonymous read access can be arranged if necessary.

The system manager will normally be given *TRAC_ADMIN* privileges. This allows them to do additional things which normal users cannot do such as:

- Delete wiki pages (the latest version or the entire page).
- Add or modify milestones, components and versions.
- Modify ticket descriptions and delete ticket attachments.
- Make wiki pages read-only.
- Alter the permissions.

For further details please see the section Trac Permissions from the Trac documentation.

Email notification

By default, each Trac system is configured such that the owner and reporter and anyone on the *CC* list are notified whenever a change is made to a ticket. If system mangers wish to be notified of all ticket changes then this can also be configured. Alternatively, email notifications can be disabled if they are not wanted.

Milestones

Milestones are useful for identifying when tickets need to be resolved. Typically, milestones may be particular system releases or time periods. The FCM team can configure milestones for you when they set up your Trac system. However, this is not strictly necessary since milestones can be set up via the web interface using the admin account (go to the “Roadmap” page).

Other configurable items

There are lots of other things that can be configured in your Trac system such as:

- Custom fields
- System icon
- Stylesheets

For further details please see the sections The Trac Configuration File and The Trac Ticket System from the Trac documentation.

Trac hosting

The FCM team will organise the hosting of your Trac system. It will be set up on the same server that hosts your Subversion repository and access will be provided via a web server. The FCM team will advise you of the URL. Each night a backup of your Trac system will be taken in case the live system should become corrupted for whatever reason.

FCM keywords

When you set up a repository for a new project, you will normally want the FCM team to set up a URL keyword for it in the FCM central configuration file. The name of the project should be a short string containing only word characters.

Individual projects can store revision keywords using the Subversion property `fcm:revision` at registered URLs. Using the UM as an example: if `UM` is a registered URL keyword, you can add the `fcm:revision` property at the head of the UM project by doing a non-recursive checkout. E.g.:

```
(prompt)$ fcm co -q -N fcm:um um
(prompt)$ fcm pe fcm:revision um
```

In the editor, add the following and `fcm commit`:

```
VN6.3 = 402
VN6.4 = 1396
VN6.5 = 2599
VN6.6 = 4913
VN7.0 = 6163
```

In a subsequent invocation of `fcm`, if a revision keyword is specified for a URL in the UM namespace, the command will attempt to load it from the `fcm:revision` property at the head of the UM project. Revision keywords can also be defined in the FCM central configuration file if you prefer.

If the project has an associated Trac browser, you can also declare browser URL mapping in the central configuration file. This allows FCM to associate the Subversion URL with a Trac browser URL. There is an automatic default for mapping URLs hosted by the FCM team at the Met Office. External users of FCM may want to adjust this default for their site.

To change the default browser URL mapping, you need to make 3 `SET::URL_BROWSER_MAPPING_DEFAULT::<key>` declarations in your `$FCM/etc/fcm.cfg` file. There are 3 keys to this declaration: *LOCATION_COMPONENT_PATTERN*, *BROWSER_URL_TEMPLATE* and *BROWSER_REV_TEMPLATE*. The *LOCATION_COMPONENT_PATTERN* is a Perl regular expression, which is used to separate the scheme-specific part of a version control system URL into a number of components by capturing its substrings. These components are then used to fill in the numbered fields in the *BROWSER_URL_TEMPLATE*. The template should have one more field than the number of components captured by *LOCATION_COMPONENT_PATTERN*. The last field is used to place the revision, which is generated via the *BROWSER_REV_TEMPLATE*. This template should have a single numbered field for filling in the revision number. This is best demonstrated by an example. Consider the declarations:

```
%pattern       ^//([^/]+)/(.*)$
%url_template  http://{1}/intertrac/source:{2}{3}
%rev_template  @{1}
set::url_browser_mapping_default::location_component_pattern  %pattern
set::url_browser_mapping_default::browser_url_template        %url_template
set::url_browser_mapping_default::browser_rev_template        %rev_template
```

If we have a Subversion URL `svn://repos/path/to/a/file`, the *LOCATION_COMPONENT_PATTERN* will capture the components [`repos`, `path/to/a/file`]. When this is applied to the `BROWSER_URL_TEMPLATE`, *{1}* will be translated to `repos` and *{2}* will be translated to `path/to/a/file`. A revision is not given in this case, and so *{3}* is inserted with an empty string. The result is `http://repos/intertrac/path/to/a/file`. If the revision is `1357`, the *BROWSER_REV_TEMPLATE* will be used to translate it to `@1357`, which is then inserted to *{3}* of the *BROWSER_URL_TEMPLATE*. The result is therefore `http://repos/intertrac/path/to/a/file@1357`.

For more information on how to set up the keywords, please refer to Repository & Revision Keywords and the Annex: Declarations in FCM central/user configuration file.

Extract and build configuration

The extract and build systems are very flexibile and can be used in lots of different ways. It is therefore difficult to give specific advice explaining how to configure them. However, based on experience with a number of systems, the following general advice can be offered.

- Standard extract configuration files should be defined and stored within the repository. Users then include these files into their configurations, before applying their local changes.
- The files should be designed to include one another in a hierarchy. For example, you may have one core file which defines all the repository and source locations plus a series of platform/compiler specific files which include the core file. More complex setups are also possible if you need to cater for other options such as different optimisation levels, 32/64 bit, etc.
- When including other configuration files, always make use of the special *$HERE* variable (rather than, for instance, referring to a fixed repository location). When your configuration file is parsed, this special variable is normally expanded into the container directory of the current configuration file. This means that the include statements should work correctly whether you are referring to configuration files in the repository trunk, in a branch or in a local working copy.
- Make good use of user variables (e.g. `%fred`) to simplify repetitive declarations and make

your configuration files easier to maintain.
- Use continuation lines to split long lines and make them easier to read (see the FCM Command Reference section for further details about configuration files).

Probably the best advice is to look at what has already been set up for other systems. The FCM team can advise on the best systems to examine.

When you create a stable build you should keep an extract configuration file that can reproduce the build. One easy way to do this is to create your build using the standard configuration files and the latest versions of the code. You can then save the expanded extract configuration file which is created when you run the extract. To help document your stable build you can use the command `fcm cmp-ext-cfg` to show what has changed.

Maintaining alternate versions of namelists and data files

Sometimes it is useful to be able to access particular revisions of some directories from a FCM repository without having to go via Subversion. Typical examples are namelist or data files used as inputs to a program. The script `fcm_update_version_dir.pl` is designed to help with this. It can be used to maintain a set of extracted version directories from a FCM repository. The script has the following options and arguments:

`-f [--full]`
: Specify the full mode, in which the versioned directories of each specified item will be removed before being re-extracted.

`-d [--dest] arg`
: Specify a destination *arg* for the extraction. If not specified, the current working directory will be used as the base path.

`-u [--url] arg`
: Specify the source repository URL. This option is compulsory.

If an argument is specified, it must be the location of a configuration file. Otherwise, the command reads its configuration from the standard input. The configuration file is a line-based text file. Blank lines and lines beginning with a `#` are ignored.

Each configuration line must contain the relative path of a sub-directory under the specified source repository URL. If the path ends in `*` then the path is expanded recursively and any sub-directories containing regular files are added to the list of relative paths to extract.

Optionally, each relative path may be followed by a list of space separated conditions. Each condition is a conditional operator (`>`, `>=`, `<`, `<=`, `==` or `!=`) followed by a revision number or the keyword HEAD.

The command uses the revision log to determine the revisions at which the relative path has been updated in the source repository URL. If these revisions also satisfy the conditions set by the user, they will be considered in the extraction. In full mode, everything is re-extracted. In incremental mode, the version directories are only updated if they do not already exist.

Example:

```
(SHELL PROMPT)$ fcm_update_version_dir.pl -u fcm:ver_tr <<EOF
namelists/VerNL_AreaDefinition   >1000 !=1234
namelists/VerNL_GRIBToPPCode     >=600 <3000
namelists/VerNL_StationList
elements/*                       >1000
EOF
```

N.B.

1. Each time a sub-directory is revised, the script assigns a sequential *v* number for the item. Each *v* number for a sub-directory, therefore, is associated with a revision number. For each extracted revision directory, there is a corresponding *v* number symbolic link pointing to it.
2. The system also creates a symbolic link `latest` to point to the latest extracted revision directory.

Defining working practises and policies

Some options on working practises and policies are defined in the chapter on Code Management Working Practices. Individual projects should document the approach they have adopted. In addition, each project may also need to define its own working practices and policies to suit its local need. For example each project may need to specify:

- Whether changes are allowed directly on the trunk or whether branches have to be used in all cases.
- Whether all users are allowed to make changes to the trunk.
- Whether Trac tickets have to be raised for all changes to the trunk.
- Whether Trac tickets should be raised for all support queries or whether a Trac ticket should only be raised once there is an agreed "issue".
- Whether branches should normally be made from the latest code or from a stable release.
- Whether a user is allowed to resolve conflicts directly when merging a branch into the trunk or whether he/she should merge the trunk into the branch and resolve the conflicts in the branch first.
- Whether all code changes to the trunk need to be reviewed.
- What testing is required before changes can be merged to the trunk.
- Whether history entries are maintained in source files or whether individual source files changes need to be described in the Subversion log message.
- Branch deletion policy.
- Whether any files in the project require locking before being changed.

FCM Command Reference

fcm Configuration File

The FCM system uses simple line based text files to store configuration settings. All configuration files used by FCM are based on the same principles:

- All configurations are stored in plain text files.
- A line in a file may contain a configuration setting or a comment.
- Blank lines are ignored.
- A line that begins with a `#` is a *comment* line.
- Each configuration line has a *label*, a *value* and optionally a trailing *comment*. For example:

```
my::label  this is the value  # some comment
```

- A *label* in a configuration line may contain any non-space character. A space character marks the end of the *label*.
- Words or fields in the *label* are delimited by a double colon `::` or slash `/`. To improve readability, the convention is to only use slash as the delimiter when referring to package names.
- The first non-space character after the *label* is the beginning of the *value*. The *value* may contain space characters. The newline character or the character sequence `" # "` marks the end of the *value*.
- The first non-space character after the character sequence `" # "` at the end of the *value* is the *comment*. The *comment* is normally ignored by the parser of the configuration file.
- If it is indicated in the documentation that a declaration is expecting a `true` or `false` value, the following can all be used to indicate a value of `false`: an empty string, the numeric `0`, the string `off` or the string `no`. All other values will be considered `true`.
- If the last character of the *value* is a backslash `\`, the next non-comment line will be the continuation of the current line. Please note that trailing spaces before the continuation mark are preserved, and leading spaces are removed from the beginning of a continuation. If you want to have leading spaces in a continued line, start the line with a backslash `\` before the leading spaces. It is also worth bearing in mind that the backslash `\` character is only significant if it appears at the end of a value or the beginning of a continuation line. It is not a special character if it appears elsewhere. For example:

```
foo bar\
    egg\
    ham
# will become:
foo bareggham

foo bar \
    egg \
    ham
# will become:
foo bar egg ham

foo bar\
    \ egg\
    \ ham
# will become:
foo bar egg ham

foo bar\
    \ egg\ham
# will become:
foo bar egg\ham
```

The FCM central and user configuration files can be used to add or modify some of the default settings of FCM. When the `fcm` command is invoked, it normally attempts to search for a central configuration file at `$BINDIR/../etc/fcm.cfg` and then `$BINDIR/fcm.cfg`, where `$BINDIR` is the container directory of the `fcm` command. If a central configuration file can be located, the settings in the file will replace the pre-defined ones. After searching/reading the central configuration file, the system will attempt to search for a user configuration file located at `$HOME/.fcm` of the current user. If such a file can be found, its settings will replace the pre-defined ones as well as those defined in the central configuration file.

For information on the valid entries in the central and user configuration files, please refer to the Annex: Declarations in FCM central/user configuration file.

fcm build

Usage

```
fcm build [OPTIONS...] [CFGFILE]
```

Description

`fcm build` invokes the FCM build system.

The path to a valid build configuration file *CFGFILE* may be provided as either a URL or a pathname. Otherwise, the build system searches the default locations for a build configuration file.

If no option is specified, the system uses the `-s 5 -t all -j 1 -v 1` by default.

`-a [--archive]`
: This option can be specified to switch on the archive mode. In archive mode, sub-directories produced by the build will be archived in `tar` format at the end of a successful build. This option should not be used if the current build is intended to be re-used as a pre-compiled build.

`--clean`
: If this option is specified, the build system will parse the configuration file, remove contents generated by the build system in the destination and exit.

`-f [--full]`
: If this option is specified, the build system will attempt to perform a full/clean build by removing any previous build files. Otherwise, the build system will attempt to perform an incremental build where appropriate.

`--ignore-lock`
: When the build system is invoked, it sets a lock file in the build root directory to prevent other extracts/builds taking place in the same location. The lock file is normally removed when the build system exits. (However, a lock file may be left behind if the user interrupts the command, e.g. by typing **`Ctrl-C`**.) You can bypass the check for lock files by using this option.

`-j [--jobs] arg`
: This option can be used to specify the number of parallel jobs that can be handled by the `make` command. The argument *arg* must be a natural integer to represent the number of jobs. If not specified, the default is to perform serial `make` (i.e. 1 job).

`-s [--stage] arg`
: This option can be used to limit the actions performed by the build system, up to a named stage determined by the argument *arg*. If not specified, the default is 5. The stages are:
  - *1, s or setup*: Stage 1, read configuration and set up the build
  - *2, pp or pre_process*: Stage 2, perform pre-processing for source files that require pre-processing
  - *3, gd or generate_dependency*: Stage 3, scan source files for dependency

information and generate `make` rules for them
- *4, gi or generate_interface*: Stage 4, generate interface files for Fortran 9X source files
- *5, m or make*: Stage 5, invoke the `make` command to build the project

`-t [--targets] arg`
This option can be used to specify the targets to be built. The argument *arg* must be a colon-separated list of valid targets. If not specified, the default to be built is the `all` target.

`-v [--verbose] arg`
This option can be specified to alter the level of diagnostic output. The argument *arg* to this option must be an integer greater than or equal to 0. The verbose level increases with this number. If not specified, the default verbose level is 1.

For further details, please refer to the chapter on The Build System.

Alternate Names
bld

fcm extract

Usage
```
fcm extract [OPTIONS...] [CFGFILE]
```
Description

`fcm extract` invokes the FCM extract system.

The path to a valid extract configuration file *CFGFILE* may be provided as either a URL or a pathname. Otherwise, the extract system searches the default locations for an extract configuration file.

`--clean`
If this option is specified, the extract system will parse the configuration file, remove contents generated by previous extract in the destination and exit.

`-f [--full]`
If this option is specified, the extract system will attempt to perform a full extract by removing any previous extracted files. Otherwise, the extract system will attempt to perform an incremental extract where appropriate.

`--ignore-lock`
When the extract system is invoked, it sets a lock file in the extract destination root directory to prevent other extracts/builds taking place in the same location. The lock file is normally removed when the extract system exits. (However, a lock file may be left behind if the user interrupts the command, e.g. by typing **`Ctrl-C`**.) You can bypass the check for lock files by using this option.

`-v [--verbose] arg`
This option can be specified to alter the level of diagnostic output. The argument *arg* to this option must be an integer greater than or equal to 0. The verbose level increases with this number. If not specified, the default verbose level is 1.

For further details, please refer to the chapter on The Extract System.

Alternate Names
ext

fcm cmp-ext-cfg

Usage

```
fcm cmp-ext-cfg [--verbose (-v) arg] [--wiki (-w) arg] CFG1 CFG2
```

Description

`fcm cmp-ext-cfg` compares the extract configurations of two similar extract configuration files *CFG1* and *CFG2*. It reports repository branches and source directories that are declared in one file but not another. If a source directory is declared in both files, it compares their versions. If they differ, it uses `svn log` to obtain a list of revision numbers at which changes are made to the source directory. It then reports, for each declared repository branch, the revisions at which changes occur in their declared source directories.

The list of revisions for each declared repository branch is normally printed out as a simple list in plain text.

`--verbose ARG`
: You can use this option to print the log of each revision, by setting *ARG* to 2.

`--wiki`
: Alternatively, you can use this option to change that into an tabular output suitable for inserting into a Trac wiki page. This option must be specified with an argument, which must be the Subversion URL or FCM URL keyword of a FCM project associated with the intended Trac system. The URL allows the command to work out the correct wiki syntax to use.

fcm gui

Usage

```
fcm gui [DIR]
```

Description

`fcm gui` starts up the FCM GUI. If *DIR* is specified then this is used as the working directory.

For further details, please refer to the section Using the GUI.

fcm keyword-print

Usage

```
fcm keyword-print [TARGET]
```

Description

If no argument is specified, `fcm keyword-print` prints all the registered FCM location keywords. Otherwise, it prints the location and revision keywords according to the argument *TARGET*, which must be a FCM URL keyword, a Subversion URL or a path to a Subversion working copy.

Alternate Names

kp

FCM Code Management Commands

This section describes all of the code management commands supported by `fcm`.

- In some cases `fcm` simply passes the command directly on to `svn` (after expanding any URL keywords). These commands are listed in the Other Code Management Commands section.
- Where `fcm` adds functionality these commands are discussed individually.
- In all cases, all the command abbreviations supported by `svn` work with `fcm`.

fcm add

Usage

```
fcm add --check (-c)
fcm add <any valid svn add options>
```

Description

`fcm add` supports all of the switches and arguments supported by `svn add` (refer to the Subversion book for details).

In addition, `fcm add` supports a `--check` switch (no other switches or arguments). When this is specified then `fcm` checks for any files which are not currently under version control (i.e. those marked with a `?` by `svn status`) and prompts to see if you wish to schedule them for addition at the next commit (using `svn add`).

For further details refer to the section Adding and Removing Files.

fcm branch

Usage

```
fcm branch [--info (-i)] [<info-options>] [TARGET]
fcm branch --delete (-d) [<info-options>] [<commit-options>] [TARGET]
fcm branch --create (-c) --name (-n) arg [--revision arg]
[<create-options>] [<commit-options>] [TARGET]
fcm branch --list (-l) [<branch-list-options>] [TARGET]
```

Description

If *TARGET* is specified, it must either be a URL or a path to a local working copy. Otherwise, the current working directory must be a working copy. For `--info` and `--delete`, the specified URL or that of the working copy must be a valid branch (including the trunk) in a standard FCM project. For `--create` and `--list`, it must be a valid URL of a standard FCM project.

`fcm branch --info`

Displays information about a branch. This is the default if no options are specified. It performs the following actions:

- It reports the basic information of the branch URL, as returned by `svn info`.
- If `--verbose` is set, it also prints the log message of the last change revision.
- If the URL is not the trunk:
  - It reports the branch creation information, including the revision, author and date. It also reports the parent URL@REV of the branch. If `--verbose` is set, it prints the log message of the branch creation revision.
  - If the branch does not exist at the HEAD, it reports the revision at which it is deleted.
  - It reports the last merges into and from the parent branch. If `--verbose` is set, it also prints the log message of these merges.
  - It reports the revisions available for merging into and from the parent branch. If `--verbose` is set, it also prints the log message of these revisions.
- If `--show-children` is specified, it lists the current children of the branch and their

create revisions. Where appropriate, it reports the revision of each child, which is last merged from/into the current branch. It also reports the available merges from/into each child into the current branch.

- If `--show-siblings` is specified, it reports recent merges from/into sibling branches. It also reports the available merges from/into sibling branches where recent merges are detected. If `--verbose` is set, it also prints the log message of these merges.
- If `--show-other` is specified, it reports all custom and reverse merges into the current branch.
- You can turn on `--show-children`, `--show-siblings` and `--show-other` simultaneously by specifying `--show-all`.

For further details refer to the section Getting Information About Branches.

`fcm branch --delete`

Deletes a branch. This command performs the following actions:

- Firstly, it provides exactly the same output as `fcm branch --info`.
- If you do not specify the `--non-interactive` option, it starts an editor (using a similar convention as commit) to allow you to add further comment to the commit log message. A standard commit log template and change summary is provided for you below the line that says `--Add your commit message ABOVE - do not alter this line or those below--`. If you need to add any extra message to the log, please do so **above** this line. When you exit the editor, the command will report the commit log before prompting for confirmation that you wish to proceed with deleting the branch (it aborts if not).

If you specify the `--non-interactive` option, the command will not prompt you for anything. (The `--svn-non-interactive` option is set automatically when you specify `--non-interactive`.)

Subversion may prompt you for authentication if it is the first time you access the repository. The command fails if the authentication fails. If you specify the `--svn-non-interactive` option, Subversion will not prompt you for authentication. In such case, the command will simply fail if authentication is required. You can use the `--password` option to specify a password. Please note that the `--svn-non-interactive` option is always specified if you are running `branch --delete` from the FCM GUI. If authentication is required, you should specify your password using the `--password` option in **`Other options`**.

For further details refer to the section Deleting Branches.

`fcm branch --create`

Creates a new branch.

You have to choose a name for your branch. This must be specified as the argument of the `--name (-n)` option. The name of the branch must contain only characters in the set [A-Za-z0-9_-.].

You can specify the type of branch you are creating using the `--type (-t)` option. The argument to the option must be one of the following:

`DEV::USER`
: A development branch for the current user (e.g. `branches/dev/<user_id>/<branch_name>`)

`DEV::SHARE`
: A shared development branch (e.g. `branches/dev/Share/<branch_name>`)

`DEV`
: Same as `DEV::USER`

`TEST::USER`
: A test branch for the current user (e.g. `branches/test/<user_id>/<branch_name>`)

`TEST::SHARE`
: A shared test branch (e.g. `branches/test/Share/<branch_name>`)

`TEST`
: Same as `TEST::USER`

`PKG::USER`
: A package branch for the current user (e.g. `branches/pkg/<user_id>/<branch_name>`)

`PKG::SHARE`
: A shared package branch (e.g. `branches/pkg/Share/<branch_name>`)

`PKG::CONFIG`
: A configuration branch (e.g. `branches/pkg/Config/<branch_name>`)

`PKG::REL`
: A release branch (e.g. `branches/pkg/Rel/<branch_name>`)

`PKG`
: Same as `PKG::USER`

`CONFIG`
: Same as `PKG::CONFIG`

`REL`
: Same as `PKG::REL`

`SHARE`
: Same as `DEV::SHARE`

`USER`
: Same as `DEV::USER`

If the `--type` option is not specified, it defaults to DEV::USER.

Your branch name will normally be prefixed by the revision number from which it is branched. (E.g. if the branch name is `my_branch` and you are branching from revision 123 of the trunk, the final name will be `r123_my_branch`.) If this revision number is associated with a revision keyword, the keyword will be used in place of the revision number. (E.g. if revision 123 is associated with the keyword vn6.1, `r123_my_branch` will become `vn6.1_my_branch`.) You can alter this behaviour using the argument to the `--rev-flag` option. If `NORMAL` is specified, it uses the default behaviour. If `NUMBER` is specified, it will always use the revision number as the prefix, regardless of whether the revision number is defined as a keyword or not. If `NONE` is specified, it will not add a prefix to your branch name.

The command will normally create your branch from the last changed revision of the trunk of the specified project.

- You can use the `--revision` option to specify an earlier revision of the source.
- If the source URL is a valid URL of a branch in a standard FCM project, you can use the `--branch-of-branch` option to create a branch of the source branch.

The `--ticket` option can be used to specify one or more Trac ticket numbers, which the branch relates to. Multiple ticket numbers can be set by specifying this option multiple times, or by using a comma-separated list of ticket numbers as the argument to the option. If set, the line `Relates to ticket #<number>[, #<number>...]` will be

added to the template commit log.

If you specify the `--non-interactive` option, the command will not prompt you for anything. (The `--svn-non-interactive` option is set automatically when you specify `--non-interactive`.)

Subversion may prompt you for authentication if it is the first time you access the repository. The command fails if the authentication fails. If you specify the `--svn-non-interactive` option, Subversion will not prompt you for authentication. In such case, the command will simply fail if authentication is required. You can use the `--password` option to specify a password. Please note that the `--svn-non-interactive` option is always specified if you are running `branch --create` from the FCM GUI. If authentication is required, you should specify your password using the `--password` option in **`Other options`**.

This command performs the following actions:

- It determines the last changed revision of the trunk/source branch at the HEAD (or the specified) revision.
- It constructs the branch name from the option you have specified and reports it.
- It checks that the chosen branch name does not currently exist. If so, the command aborts with an error.
- If you do not specify the `--non-interactive` option, it starts an editor (using a similar convention as commit) to allow you to add further comment to the commit log message. A standard commit log template and change summary is provided for you below the line that says `--Add your commit message ABOVE - do not alter this line or those below--`. If you need to add any extra message to the log, please do so **above** this line. When you exit the editor, the command will report the commit log before prompting for confirmation that you wish to proceed (it aborts if not).
- It uses `svn mkdir` to create any sub-directories required for the copy to succeed.
- It uses `svn copy` to create the branch.

For further details refer to the section Creating Branches.

`fcm branch --list`

Lists the branches created by you (and/or other users) at the HEAD revision of a standard FCM project.

By default, it lists the branches created by you at the HEAD revision. To display the branches at a different revision, you can use the `--revision arg` option.

You can specify a list of users with the `--user arg` option, where *arg* is a colon separated list of users. (Alternatively, you can specify this option multiple times.) When this option is set, the command lists the branches created by the specified list of users instead. Note that you can also list shared branches by specifying the user as `Share`, configuration branches by specifying the user as `Config` and release branches by specifying the user as `Rel`.

You can list all branches in the project by specifying the `--show-all` option. (This option overrides the `--user arg` option.)

By default, it lists the branches in the FCM URL keyword format. If you want to print the full Subversion URL of the branches, you can use the `--verbose` option.

The command returns 0 (success) if one or more branches is found for the specified users, or 1 (failure) if no branch is found.

Alternate Names

br

fcm commit

Usage

```
fcm commit [--dry-run] [--svn-non-interactive] [--password arg]
[PATH]
```

Description

`fcm commit` sends changes from your working copy in the current working directory (or from *PATH* if it is specified) to the repository.

This command performs the following actions:

- It checks that the current working directory (or *PATH* if it is specified) is a working copy. (If not, it aborts with an error).
- It always commits from the top level of the working copy.
- It checks that there are no files in conflict, missing or out of date (it aborts if there are).
- It checks that any files which have been added have the *svn:executable* property set correctly (in case a script was added before the execute bit was set correctly).
- It reads in any existing commit message.
  - The commit message is stored in the file `#commit_message#` in the top level of your working copy.
- It adds the following line to the commit log message: `--Add your commit message ABOVE - do not alter this line or those below--`. This line, and anything below it, is automatically ignored by `svn commit`. If you need to add any extra message to the log, please do so **above** this line.
- If you have run the merge command before the commit, you will get a standard commit log template below a line that says `--FCM message (will be inserted automatically)--`. Please do not try to alter this message (your changes will be ignored if you do).
- It adds current status information to the commit message showing the list of modifications below a line that says `--Change summary (not part of commit message)--`.
- It starts an editor to allow you to edit the commit message.
  - If defined, the environment variable *SVN_EDITOR* specifies the editor.
  - Otherwise the environment variable *VISUAL* specifies the editor.
  - Otherwise the environment variable *EDITOR* specifies the editor.
  - Otherwise the editor `nedit` is used.
- It reports the commit message that will be sent to Subversion and then asks if you want to proceed (it aborts if not).
- It calls `svn commit` to send the changes to the repository.
- It calls `svn update` to bring your working copy up to the new revision.

Subversion may prompt you for authentication if it is the first time you access the repository. The command fails if the authentication fails. If you specify the `--non-interactive` option, Subversion will not prompt you for authentication. In such case, the command will simply fail if authentication is required. You can use the `--password` option to specify a password. Please note that the `--svn-non-interactive` option is always specified if you are running `commit` from the FCM GUI. If authentication is required, you should specify your password using the `--password` option in **`Other options`**.

The `--dry-run` switch prevents the command from committing any changes. This can be used to allow you to add notes to your commit message whilst you are still preparing your change.

For further details refer to the section Committing Changes.

Alternate Names

ci

fcm conflicts

Usage

```
fcm conflicts [PATH]
```

Description

`fcm conflicts` helps you to resolve any text files in your working copy which have conflicts by using the graphical merge tool `xxdiff`. If *PATH* is set, it must be a working copy, and the command will operate in it. If *PATH* is not set, the command will operate in your current working directory.

This command performs the following actions:

- For each text file reported as being in conflict (i.e. marked with a `C` by `svn status`) it calls `xxdiff`.
- If `xxdiff` reports all conflicts resolved then if asks if you wish to run `svn resolved` on that file.

For further details refer to the section Resolving Conflicts.

Alternate Names

cf

fcm delete

Usage

```
fcm delete --check (-c)
fcm delete <any valid svn delete options>
```

Description

`fcm delete` supports all of the switches, arguments and alternate names supported by `svn delete` (refer to the Subversion book for details).

In addition, `fcm delete` supports a `--check` switch (no other switches or arguments). When this is specified then `fcm` checks for any files which are missing (i.e. marked with a `!` by `svn status`) and prompts to see if you wish to schedule them for deletion at the next commit (using `svn delete`).

Subversion may prompt you for authentication if it is the first time you access the repository. The command fails if the authentication fails. If you specify the `--non-interactive` option, Subversion will not prompt you for authentication. In such case, the command will simply fail if authentication is required. Please note that the `--non-interactive` option is automatically specified if you are running `delete` from the FCM GUI and you have not checked the box **`Check for files and directories...`**. If authentication is required, you should run `delete` in interactive mode on a command line.

For further details refer to the section Adding and Removing Files.

fcm diff

Usage

```
fcm diff --branch (-b) [--graphical (-g) | --summarise | --wiki |
--trac (-t)] [TARGET]
fcm diff [--graphical (-g) | --summarise | <any valid svn diff
options>]
```

Description

`fcm diff` supports all of the switches, arguments and alternate names supported by `svn diff` (refer to the Subversion book for details). In addition, `fcm diff` supports the following switches:

`--graphical (-g)`
: If this option is specified, the command uses a graphical tool to display the differences. (The default graphical diff tool is `xxdiff`, but you can alter the behaviour by following the instruction discussed in the sub-section on Examining Changes.) This switch can be used in combination with all other valid switch (including `--branch`) except `--diff-cmd`, `--extensions`, `--trac` and `--wiki`.

`--summarise`
: This option is implemented in FCM as a wrapper to the Subversion `--summarize` option. It prints only a summary of the results.

`--branch`
: If this option is specified, the command displays the differences between the target branch and its parent. This should show you the differences which you would get if you tried to merge the changes in the branch into its parent. It performs the following actions:
  - If *TARGET* is specified, it must either be a URL or a path to a local working copy. Otherwise, the current working directory must be a working copy. The specified URL or that of the working copy must be a valid branch in a standard FCM project.
  - It determines the base of the branch relative to its parent. This is adjusted to account for any merges from the branch to its parent or vice-versa.
  - It reports what path and revision it is comparing against.
  - If `--trac` is specified, it launches Trac with your default web browser to display it. Note: if *TARGET* is a working copy, local changes in it will not be displayed by Trac.
  - If `--wiki` is specified, it prints a Trac wiki syntax for the differences between the base and the specified branch.
  - Otherwise, it calls `svn diff` to report the differences between the base and the specified branch (or working copy).

For further details refer to the section Examining Changes.

fcm merge

Usage

```
fcm merge [--dry-run] [--non-interactive] [--verbose (-v)] SOURCE
fcm merge --custom --revision N[:M] [--dry-run] [--non-interactive]
[--verbose (-v)] SOURCE
fcm merge --custom [--dry-run] [--non-interactive] [--verbose (-v)]
URL1[@REV1] URL2[@REV2]
fcm merge --reverse --revision [M:]N [--dry-run] [--non-interactive]
[--verbose (-v)]
```

Description

`fcm merge` allows you to merge changes from a source into your working copy.

Before it begins, the command does the following:

- If a *SOURCE* or *URL* is specified, it can be a full URL or a partial URL starting at the branches, trunk or tags level.
  - If a partial URL is given, and the path name does not begin with `trunk`, `tags` or `branches` then `branches/` is automatically added to the beginning of your path.
- It determines the *TARGET* URL by examining your working copy.
- If the current directory is not the top of your working copy, it changes the current directory to the top of your working copy.
- If your working copy is not pointing to a branch of a project managed by FCM, the command aborts with an error.
- If you do not specify the `--non-interactive` option, it checks for any local modifications in your working copy. If it finds any it reports them and asks you to confirm that you wish to continue (it aborts if not).

Automatic mode (i.e. neither `--custom` nor `--reverse` is specified)

Automatic merges are used to merge changes between two directly related branches, (i.e. the branches must either be created from the same parent or have a parent/child relationship). These merges are tracked by FCM and can be used by subsequent FCM commands. The merge delta is calculated by doing the following:

- It checks that the *SOURCE* and *TARGET* are directly related.
- It determines the base revision and path of the *common ancestor* of the *SOURCE* and *TARGET*.
- The base revision and path are adjusted to account for any merges from the *SOURCE* to the *TARGET* or vice-versa.
- It reports the revisions from *SOURCE* available for merging into *TARGET*. If the `--verbose` option is set, it prints the log for these revisions. It aborts if no revision is available for merging.
- If there are 2 or more revisions available for merging and you do not specify the `--non-interactive` target, it asks you which revision of the *SOURCE* you wish to merge from. The default is the last changed revision of the *SOURCE*. The merge delta is between the base and the specified revision of the *SOURCE*.
- If your working copy is a sub-tree of the *TARGET*, it ensures that the *SOURCE* contains only changes in the same sub-tree. Otherwise, the merge is unsafe, and the command will abort with an error.

  N.B.: The command looks for changes in the *SOURCE* by going through the list of changed files since the the *SOURCE* was last merged into the *TARGET*. (If there is no previous merge from SOURCE to *TARGET*, the common ancestor is used.) It is worth noting that there are situations when the command will regard your merge as *unsafe* (and so will fail incorrectly) even if the changes in the *SOURCE* outside of the current sub-tree will result in a null merge. This can happen if the changes are the results of a previous merge from the *TARGET* to the *SOURCE* or if these changes have been reversed. In such case, you will have to perform your merge in a working copy of a full tree.

Custom mode (i.e. `--custom` is specified)

The custom mode is useful if you need to merge changes selectively from another branch. The custom mode can be used in two forms:

- In the first form, you must specify a *SOURCE* as well as a revision (range) using the `--revision` option. If you specify a single revision *N*, the merge delta is between revision *N - 1* and revision *N* of the SOURCE. Otherwise, the merge delta is between revision *N* and revision *M*, where *N < M*.
- In the second form, you must specify two URLs. The merge delta is simply between the two URLs. (For each URL, if you do not specify a peg revision, the command will peg the URL with its last changed revision.)

N.B. Unlike automatic merges, custom merges are not tracked or used by subsequent FCM `diff` or `merge` commands, (although `branch --info` can be set to report them). Custom merges are always allowed, even if your working copy is pointing to a sub-tree of a branch. However, there is no checking mechanism to ensure the safety of your sub-tree custom merge so you should only do this if you are confident it is what you want. Therefore, it is recommended that you use automatic merges where possible, and use custom merges only if you know what you are doing.

Reverse mode (i.e. `--reverse` is specified)

The reverse mode is useful if you need to reverse a changeset (or a range of changesets) in the current branch of the working copy. In this mode, you must specify a revision (range) using the `--revision` option. If you specify a single revision *N*, the merge delta is between revision *N* and revision *N - 1* of the current branch. Otherwise, the merge delta is between revision *M* and revision *N*, where *M > N*.

N.B. Like custom merges, reverse merges are not tracked or used by subsequent FCM `diff` or `merge` commands, (although `branch --info` can be set to report them). Likewise, reverse merges in sub-trees are always allowed, although there is no checking mechanism to ensure the safety of your sub-tree reverse merge.

Once the merge delta is determined, the command performs the following:

- If you set the `--dry-run` option or if you are running in the interactive mode, it reports what changes will result from performing this merge by calling `svn merge --dry-run`.
  - It prints the actual `svn merge --dry-run` command if the `--verbose` option is specified.
  - If you specify the `--dry-run` option, it exits after reporting what changes will result from performing the merge.
- If you are running in the interactive mode, it asks if you want to go ahead with the merge (it aborts if not).
- It performs the merge by calling `svn merge` to apply the delta between the base and the *SOURCE* on your working copy.
  - It prints the actual `svn merge` command if the `--verbose` option is specified.
- It adds a standard template into the commit message to provide details of the merge. The template is written below the line that says `--FCM message (will be inserted automatically)--`. The fcm commit command will detect the existence of the template, so that you will not be able to alter it by accident.
  - The commit message is stored in the file `#commit_message#` in the top level of your working copy. It is created by the merge command if it does not already exist.

For further details refer to the section Merging.

fcm mkpatch

Usage

```
fcm mkpatch [OPTIONS] URL [OUTDIR]
```

Description

`fcm mkpatch` creates patches from the specified revisions of the specified *URL*, which must be a branch URL of a valid FCM project. If the *URL* is a sub-directory of a branch, it will use the root of the branch.

If *OUTDIR* is specified, the output is sent to *OUTDIR*. Otherwise, the output will be sent to a default location in the current directory (`$PWD/fcm-mkpatch-out/`). The output directory will contain the patch for each revision as well as a script for importing the patch.

If a revision is specified using the `--revision` option, it will attempt to create a patch based on the changes at that revision. If a revision is not specified, it will attempt to create a patch based on the changes at the HEAD revision. If a revision range is specified, it will attempt to create a patch for each revision in that range (including the change in the lower range) where changes have taken place in the URL. No output will be written if there is no change in the given revision (range).

The `--exclude` option can be used to exclude a path in the URL. The specified path must be a relative path of the URL. Glob patterns such as `*` and `?` are acceptable. Changes in an excluded path will not be considered in the patch. A changeset containing changes only in the excluded path will not be considered at all. Multiple paths can be specified by using a colon-separated list of paths, or by specifying this option multiple times.

The `--organisation` option can be used to specify the name of your organisation. The command will attempt to parse the commit log message for each revision in the patch. It will remove all merge templates, replace Trac links with a modified string, and add information about the original changeset. If you specify the name of your organisation, it will replace Trac links such as `ticket:123` with `$organisation_ticket:123`, and report the orginal changeset with a message such as `$organisation_changeset:1000`. If the organisation name is not specified then it defaults to `original`.

Within the output directory are the *patches* and the log message file for each revision. It also contains a generated script `fcm-import-patch` for importing the patches. The user of the script can invoke the script with either a URL or a working copy argument, and the script will attempt to import the patches into the given URL or working copy.

It is worth noting that changes in Subversion properties, including changes in executable permissions, are not handled by the import script.

fcm switch

Usage

```
fcm switch [OPTIONS] URL [PATH]
fcm switch --relocate [OPTIONS] FROM TO [PATH]
```

Description

`fcm switch` supports the arguments and alternate names supported by `svn switch`. If `--relocate` is specified, it supports all options supported by `svn switch`. Otherwise, it supports `--non-interactive`, `-r [--revision]` and `-q [--quiet]` only. (Please refer to the Subversion book for details).

If `--relocate` is specified, FCM will pass the options and arguments directly to the corresponding Subversion command. Otherwise, FCM will ensure that your working copy switches safely through the following actions:

- If *PATH* (or the current working directory if *PATH* is not specified) is not at the top of a working copy, the command will automatically search for the top of the working copy, and the switch command will always apply recursively from that level.
- You can specify only the *branch* part of the URL, such as `trunk`, `branches/dev/fred/r1234_bob` or even `dev/fred/r1234_bob` and the command will work out the full URL for you.
- If you do not specify the `--non-interactive` option, it checks for any local modifications in your working copy. If it finds any it reports them and asks you to confirm that you wish to continue (it aborts if not).
- If you have some template messages in the `#commit_message#` file in the top level of your working copy, (e.g. after you have performed a merge), the command will report an error. You should remove the template message manually from the `#commit_message#` file before re-running `switch`.
- The command will analyse the current working copy URL and the specified URL to ensure that they are in the same project. If your working copy is a sub-tree of a project, the command will assume that you want the same sub-tree in the new URL.

For further details refer to the section Switching your working copy to point to another branch.

fcm trac

Usage

```
fcm trac [--browser (-b) ARG] [PATH]
```

Description

`fcm trac` invokes the web-browser to launch the corresponding URL of the web-based repository browser (currently Trac browser) to view the Subversion repository specified by *PATH*.

If the `--browser` option is specified, *ARG* must be a valid command to a web browser. If this option is not specified, the default is to use `firefox`, or whatever setting you have declared in the user configuration file (`$HOME/.fcm`) using the label `SET::MISC::WEB_BROWSER`. For example:

```
set::misc::web_browser  mozilla
```

If *PATH* is specified, it must be a path to a local working copying, a Subversion URL or a FCM URL keyword. If *PATH* is not specified, it is set to `.`, the current working directory. If *PATH* is a directory in the local file system, the command will determine whether it is a working copy. If so, its associated Subversion URL will be used. The command fails if the directory is not a working copy. If *PATH* is a Subversion URL or a FCM URL keyword, the URL can be *pegged* with a revision number using the `@` symbol. For example, to view the trunk of the FCM repository at revision 400, you can use `fcm:fcm_tr@400`. The URL declared by or associated with *PATH* must also be associated with a Trac browser URL, which is declared using the `SET::TRAC::<pck>` label in the FCM central/user configuration file. The command fails if an associated Trac browser URL is not found.

Alternate Names

www

fcm update

Usage

```
fcm update [OPTIONS] [PATH ...]
```

Description

`fcm update` supports the arguments and alternate names supported by `svn update`. It supports the options `--non-interactive`, `-r [--revision]` and `-q [--quiet]` only. (Please refer to the Subversion book for details).

FCM will ensure that your working copies updates safely through the following actions:

- If *PATH* (or the current working directory if *PATH* is not specified) is not at the top of a working copy, the command will automatically search for the top of the working copy, and the update command will always apply recursively from that level.
- If you do not specify the `--non-interactive` option, it uses `svn status --show-updates` to display what will be updated in your working copies and to check for local modifications (if you specify `-r [--revision]` then it just uses `svn status`). If it finds any it reports them and asks you to confirm that you wish to continue (it aborts if not).

Other Code Management Commands

Other `svn` commands are supported by `fcm` without any change in functionality, with the following minor enhancements:

- Where appropriate, FCM performs repository and revision keywords expansion.
- The `fcm checkout` command fails if you attempt to checkout into an existing working copy.
- FCM prints the corresponding `svn` command, except for `cat` and any commands with the `--xml` option specified.

The following is a list of the commands:

- svn blame
- svn cat
- svn checkout
- svn cleanup
- svn copy
- svn export
- svn import
- svn info
- svn list
- svn log
- svn lock
- svn mkdir
- svn move
- svn propdel
- svn propedit
- svn propget
- svn proplist
- svn propset
- svn resolved
- svn revert
- svn status

- svn unlock

Please refer to the Subversion Complete Reference in the Subversion book for details of these commands.

Further Information

FCM Detailed Design

- Information about the internal workings of the FCM system.
- Intended mainly for people maintaining or developing the FCM system.

Subversion

- Subversion Home Page
- Subversion Book
- Subversion FAQ
- Hacker's Guide to Subversion: Contains some useful guidelines on log message, filing issues, etc.

Other links

- Trac Home Page
- xxdiff Home Page

Annex: Quick reference

Note: some sub-commands can be invoked with alternate names. For example, `fcm help` is the same as `fcm ?`. In this annex, some favourite alternate names are listed, separated by a pipe, i.e. the above example will be given as `fcm help|?`.

Getting help

`fcm help|?`
: get list of subcommands

`fcm help|? SUBCOMMAND`
: get help on SUBCOMMAND

Maintaining the working copy

`fcm checkout|co [OPTIONS] URL [DEST]`
: Checkout URL (and create a working copy at DEST)

`fcm checkout|co -r N URL [DEST]`
: Checkout revision N of URL (and create a working copy at DEST)

`fcm info`
: Print working copy information

`fcm status|st [OPTIONS]`
: Print status of working copy

`fcm status|st -u`
: Show update information

`fcm status|st -v`
: Show verbose information

`fcm update|up`
: Update working copy with repository changes

`fcm switch|sw URL`
: Switch your working copy to point to a branch specified by URL

`fcm commit|ci`
: Commit local changes back into the repository

Preparing changes

`fcm diff|di [OPTIONS]`
: Display working copy changes in unified diff format

`fcm diff|di -b`
: Show differences relative to the base of the branch

`fcm diff|di -g`
: Display working copy changes with a graphical diff tool

`fcm diff|di -r N`
: Display working copy changes against revision N

`fcm diff|di -t`
: Display differences in Trac, (with -b only)

`fcm revert [OPTIONS] PATH`
: Restore the file PATH to the pristine copy

`fcm revert -R PATH`
: Descend PATH recursively, restoring any modified files to the pristine copy

`fcm mkdir [PATH]`
: Add a directory PATH under revision control

`fcm add [OPTIONS] [PATH]`
: Add PATH under revision control

`fcm add -c [PATH]`
: Check for items not under revision control and add them

`fcm delete|del|rm [OPTIONS] [PATH]`
: Remove PATH from revision control

`fcm delete|del|rm -c [PATH]`
: Check for missing items and remove them

`fcm copy|cp SRC DST`
: Duplicate SRC to DST, remembering history

`fcm move|mv SRC DST`
: Move or rename SRC to DST, remembering history

Browsing

`fcm log [OPTIONS] [TARGET]`
: Show the log message of a TARGET that can either be working copy or URL

`fcm log -r N[:M] [TARGET]`
: Show the log message of a range of reivsions

`fcm list|ls [OPTIONS] [TARGET]`
: List directory entries in TARGET

`fcm list|ls -r N [TARGET]`
: List directory entries of revision N

`fcm list|ls -v [TARGET]`
: List directory entries in verbose mode

`fcm list|ls -R [TARGET]`
: List directory entries recursively down the directories

`fcm trac [TARGET]`
: Open a WWW browser to browse TARGET with Trac

Branching

`fcm branch|br [OPTIONS] [URL]`
: Show branch information of URL or local working copy

`fcm branch|br -d [URL]`
: Show branch information and delete the branch

`fcm branch|br -c -n NAME [URL]`
: Create a branch

`fcm merge [SOURCE]`
: Merge changes from SOURCE to your working copy

`fcm conflicts|cf`
: Use xxdiff to resolve conflicts in your working copy

Annex: Declarations in FCM central/user configuration file

Please note that setting labels in both the central and the user configuration files are case insensitive.

N.B. almost all settings in the Fcm::Config module can be modified using the central/user configuration file. However, most users should only ever need to use the following.

SET::URL::<pck>
~~SET::REPOS::<pck>~~

This declares a URL keyword for the package <pck>. The value of the declaration must be a valid Subversion <URL>. Once declared, the URL keyword <pck> will be associated with the specified URL. In subsequent invocations of the `fcm` command, the following expansion may take place:

- `fcm:<pck>`: replaced by `<URL>`.
- `fcm:<pck>_tr` or `fcm:<pck>-tr`: replaced by `<URL>/trunk`
- `fcm:<pck>_br` or `fcm:<pck>-br`: replaced by `<URL>/branches`
- `fcm:<pck>_tg` or `fcm:<pck>-tg`: replaced by `<URL>/tags`

Example:

```
# Associate "var" with "svn://server/VAR_svn/var"
set::url::var  svn://server/VAR_svn/var

# "fcm:var" is now the same as "svn://server/VAR_svn/var"
```

SET::REVISION::<pck>::<keyword>

This declares <keyword> to be the revision number for the package <pck>. The <keyword> string can contain any characters except spaces. It must not contain only digits (as digits are treated as revision numbers). It must not be the Subversion revision keywords HEAD, BASE, COMMITTED and PREV. It cannot begin and end with a pair of curly brackets (as this will be parsed as a revision date). The package <pck> must be associated with a URL using the SET::URL::<pck> declaration described above before this declaration can make sense. Once defined, <keyword> can be used anywhere in place the defined revision number.

Example:

```
set::revision::var::v22.0  8410

# E.g. "fcm list -r v22.0 fcm:var" is now the same as
#      "fcm list -r 8410 fcm:var".
```

SET::URL_BROWSER_MAPPING_DEFAULT::<key>

These declarations are used to change the global default for mapping a version control system URL to its corresponding web browser URL. <key> can be LOCATION_COMPONENT_PATTERN, BROWSER_URL_TEMPLATE or BROWSER_REV_TEMPLATE.

Example:

```
set::url_browser_mapping_default::location_component_pattern ^//([^/]+)/(.*)$
set::url_browser_mapping_default::browser_url_template http://{1}/intertrac/source:{2}{3}
set::url_browser_mapping_default::browser_rev_template @{1}
```

SET::URL_BROWSER_MAPPING::<pck>::<key>

Similar to SET::URL_BROWSER_MAPPING_DEFAULT::<key>, but settings only apply to the specified <pck>.

Example:

```
set::url_browser_mapping::var::location_component_pattern ^//([^/]+)/(.*)$
set::url_browser_mapping::var::browser_url_template http://{1}/intertrac/source:{2}{3}
set::url_browser_mapping::var::browser_rev_template @{1}
```

SET::MISC::WEB_BROWSER

This declares a default web browser that can be used by some FCM commands to browse files and documents online. The default is `firefox`.

Example:

```
# Use Netscape instead of Firefox
set::misc::web_browser  netscape
```

INC

This declares the name of a file containing user configuration. The lines in the declared file will be included inline.

Example:

```
inc  ~fred/etc/fcm.cfg
# ... and then your changes ...
```

Annex: Declarations in FCM extract configuration file

The following is a list of supported declarations for the configuration file used by the FCM extract system. Unless otherwise stated, the fields in all declaration labels are not case sensitive.

CFG::TYPE

The configuration file type, the value should always be `ext` for an extract configuration file. This declaration is compulsory for all extract configuration files.

Example:

```
cfg::type  ext
```

CFG::VERSION

The file format version, currently `1.0` - a version is included so that we shall be able to read the configuration file correctly should we decide to change its format in the future.

Example:

```
cfg::version  1.0
```

%<name>

`%<name>` declares an internal variable *<name>* that can later be re-used.

Example:

```
%my_variable   foo
src::bar::base %my_variable
src::egg::base %my_variable
src::ham::base %my_variable
```

INC

This declares the name of a file containing extract configuration. The lines in the declared file will be included inline to the current configuration file.

Example:

```
inc  ~frva/var_stable_22.0/cfg/ext.cfg
# ... and then your changes ...
```

DEST[::ROOTDIR]

The *root* path of the destination of this extract. This declaration is compulsory for all extract configuration files.

Example:

```
dest  $HOME/project/my_project
```

USE

This declares the location of a previous successful extract, which the current extract will inherit from. If the previous extract is also a build, the subsequent invocation of the build system on the current extract will automatically trigger an inherited incremental build based on that build.

Example:

```
use  ~frva/var_stable_22.0
# ... and then the settings for your current extract ...
```

RDEST[::ROOTDIR]

The alternate destination of this extract. This declaration is compulsory if this extract requires mirroring to an alternate destination.

Example:

```
rdest  /home/nwp/da/frva/project/my_project
```

RDEST::LOGNAME

The login name of the user on the alternate destination machine. If not specified, the current login name of the user on the local platform is assumed.

Example:

```
rdest::logname  frva
```

RDEST::MACHINE

The destination machine for this extract. If not specified, the current host name is assumed.

Example:

```
rdest::machine  tx01
```

RDEST::MIRROR_CMD
~~MIRROR~~

The extract system can mirror the extracted source to an alternate machine. Currently, it does this using either the `rdist` or the `rsync` command. The default is `rsync`. This declaration can be used to switch to using `rdist`.

Example:

```
rdest::mirror_cmd  rdist
```

RDEST::RSH_MKDIR_RSH (~~RDEST::REMOTE_SHELL~~)
RDEST::RSH_MKDIR_RSHFLAGS
RDEST::RSH_MKDIR_MKDIR
RDEST::RSH_MKDIR_MKDIRFLAGS

If `rsync` is used to mirror an extract, the system needs to issue a separate remote shell command to create the container directory of the mirror destination. The default is to issue a shell command in the form `ssh -n -oBatchMode=yes LOGNAME@MACHINE mkdir -p DEST`. These declarations can be used to modify the command.

Example:

```
# Examples using the default settings:
rdest::rsh_mkdir_rsh         ssh
rdest::rsh_mkdir_rshflags    -n -oBatchMode=yes
rdest::rsh_mkdir_mkdir       mkdir
rdest::rsh_mkdir_mkdirflags  -p
```

RDEST::RSYNC
RDEST::RSYNCFLAGS

These declarations are only useful if `rsync` is used to mirror an extract. By default, the system issues the shell command `rsync -a --exclude='.*' --delete-excluded --timeout=900 --rsh='ssh -oBatchMode=yes' SOURCE DEST`. These declarations can be used to modify the command.

Example:

```
# Examples using the default settings:
rdest::rsync       rsync
rdest::rsyncflags  -a --exclude='.*' --delete-excluded --timeout=900 \
                   --rsh='ssh -oBatchMode=yes'
```

REPOS::<pck>::<branch>

This declares a URL or a local file system path for the container *repository* of a branch named <branch> in a package named <pck>. The package name <pck> must be the name of a top-level package (i.e. it must not contain the double colon `::` delimiter). The name <branch> is used internally within the extract system, and so is independent of the branch name of the code management system. However, it is usually desirable to use the same name of the actual branch in the code management system. For declaration of a local file system path, the convention is to name the branch `user`. Please note that both <pck> and <branch> fields are case sensitive. The declared URL must be a valid Subversion URL or a valid FCM URL keyword.

Example:

```
repos::var::base    fcm:var_tr
repos::var::branch1 fcm:var_br/frsn/r4790_foobar
repos::var::user    $HOME/var
```

REVISION::<pck>::<branch>
~~VERSION::<pck>::<branch>~~

The revision to be used for the URL of <branch> in the package <pck>. If specified, the revision must be a revision where the branch exists. If not specified, the revision defaults to last changed revision at the HEAD of the branch. Please note that if the declared *branch* is in the local file system, this declaration must not be used. The value of the declaration can be a FCM revision keyword or any revision argument acceptable by Subversion. You can use a valid revision number, a date between a pair of curly brackets (e.g. `{"2005-05-01 12:00"}`) or the keyword HEAD. However, please do not use the keywords BASE, COMMITTED or PREV as these are reserved for working copies only. Again, please note that both <pck> and <branch> fields are case sensitive.

Example:

```
# Declare the revision with the FCM revision keyword "vn22.0"
revision::var::base     vn22.0
# Declare the revision with a {date}
revision::var::branch1  {2006-01-01}
```

REVMATCH

If set to true, the declared revision of a branch must be a changed revision of that branch, (unless the keyword HEAD is used).

Example:

```
revmatch  true
```

SRC::<pcks>::<branch>

This declares a source directory for the sub-package <pcks> of <branch>. If the repository is declared as a URL, the source directory must be quoted as a relative path to the URL. If the repository is declared as a path in the local file system, the source directory can be declared as either a relative path to the *repository* or a full path. If the source directory is a relative path and <pcks> is a top-level package, the full name of the sub-package will be determined automatically using the directory names of the relative path as the names of the sub-packages. If the source directory is a full path, the full sub-package name must be specified. The name of the sub-package determines the destination path of the source directory in the extract.

Example:

```
src::var::base                     code/VarMod_PF
src::var/code/VarMod_PF::user   $HOME/var/code/VarMod_PF
```

EXPSRC::<pcks>::<branch>

This declares an expandable source directory for the sub-package <pcks> of <branch>. This declaration is essentially the same as the SRC declaration, except that the system will attempt to search recursively for sub-directories within the declared source directory.

Example:

```
expsrc::var::base  code
expsrc::var::user  code
```

CONFLICT
~~OVERRIDE~~

This declaration can be used to specify the conflict mode, which is relevant when a file is modified by two different branches (or more) relative to the base branch. The conflict mode can be `fail`, `merge` (default) or `override` (or 0, 1 and 2 respectively). If `fail` is specified, the extract fails when a file is modified by two branches (or more) relative to the base branch. If `merge` is specified, the system will attempt to merge the changes. It will fail only on unresolved conflicts. If `override` is specified, the changes in the last branch takes precedence and the changes in the earlier branches will be ignored. Note: the old `override true|false` declaration is deprecated. If declared, `override true` will be equivalent to `conflict override`, and `override false` will be equivalent to `conflict fail`.

Example:

```
conflict  override
```

BLD::<fields>

Declare a build configuration file declaration. The label <fields> is the label of the declaration. On a successful extract, <fields> will be added to the build configuration file. Please note that some of the <fields> may be case sensitive.

Example:

```
bld::target   VarScr_AnalysePF
bld::tool::fc sxmpif90
bld::tool::cc sxmpic++
# ... and so on ...
```

Annex: Declarations in FCM build configuration file

The following is a list of supported declarations for the configuration file used by the FCM build system. Unless otherwise stated, the fields in all declaration labels are not case sensitive. Build declarations can be made either in a build configuration file or in an extract configuration file. In the latter case, the prefix `BLD::` must be added at the beginning of each label to inform the extract system that the declaration is a build system declaration. (In a build configuration file, the prefix `BLD::` is optional.)

CFG::TYPE

> The configuration file type, the value should always be `bld` for a build configuration file. This declaration is compulsory for all build configuration files. (This declaration is automatic when the extract system creates a build configuration file.)
>
> Example:
>
> ```
> cfg::type  bld
> ```

CFG::VERSION

> The file format version, currently `1.0` - a version is included so that we shall be able to read the configuration file correctly should we decide to change its format in the future. (This declaration is automatic when the extract system creates a build configuration file.)
>
> Example:
>
> ```
> cfg::version  1.0
> ```

%<name>

> `%<name>` declares an internal variable *<name>* that can later be re-used.
>
> Example:
>
> ```
> %my_variable  -foo -bar
> tool::fflags  %my_variable
> tool::cflags  %my_variable
> ```

INC

> This declares the name of a file containing build configuration. The lines in the declared file will be included inline to the current configuration file.
>
> Example:
>
> ```
> inc  ~frva/var_stable_22.0/cfg/bld.cfg
> # ... and then your changes ...
> ```

DEST[::ROOTDIR]
~~DIR::ROOT~~

> The destination of the build. It must be declared for each build. (This declaration is automatic when the extract system creates a build configuration file. The value is normally the path of the extract destination.)
>
> Example:

```
dest  $HOME/my_build
```

USE

This inherits settings from a previous build. The value must be must be either the configuration file or the root directory of a successful build. Output of the build, the tools, the exclude dependency declarations, the file type registers declarations are automatically inherited from the declared build. Source directories and build targets declarations may be inherited depending on the INHERIT declarations. (If you have a USE declaration in an extract, the resulting build configuration file will contain an automatic USE declaration, which expects an inherited build at the extract destination.)

Example:

```
# Use VAR build 22.0
USE  ~frva/var_22.0
```

INHERIT::<name>[::<pcks>]

This declares whether build targets (<name> = `target`) or source directories (<name> = `src`) can be inherited using the USE statement. By default, source directories are inherited, while build targets are not. Use the value `true` to switch on inheritance, or `false` to switch off. For source directories declarations, the name of a sub-package <pcks> can be specified. If a sub-package pcks is specified, the declaration applies only to the files and directories under the sub-package. Otherwise, the declaration applies globally.

Example:

```
inherit::target   true
inherit::src      false
```

SRC[::<pcks>]

This declares a source file/directory. You must specify the sub-package <pcks> if the source file/directory is located outside of the `src/` sub-directory of the build destination or if you want to redefine the sub-package name of the source file/directory. The name of the sub-package <pcks> must be unique. Package names are delimited by double colons `::` or double underscores __. If you declare a relative path, it is assumed to be relative to the `src/` sub-directory of the build destination. (This declaration is automatic when the extract system creates the build configuration file. The list of declared source directories will be the list of extracted source directories.)

Example:

```
src::var/code/VarMod_PF  $HOME/var/src/code/VarMod_PF
```

SEARCH_SRC

This declares a flag to determine whether the build system should search the `src/` sub-directory of the build root for a list of source files. The automatic search is useful if the build system is invoked standalone and the `src/` sub-directory contains the full source tree of the build. The default is to search (`true`). Set the flag to `false` to switch off the behaviour. (When the extract system creates a build configuration file, it declares all source files. Searching of the source sub-directory should not be required, and so this flag is automatically set to `false`.)

Example:

```
search_src  false
```

TARGET

Specify the targets for the build. Multiple targets can be declared in one or more declarations. These targets become the dependencies of the default `all` target in the *Makefile*. It is worth noting that `TARGET` declarations are cumulative. A later declaration does not override an earlier one - it simply adds more targets to the list.

Example:

```
target  VarScr_AnalysePF VarScr_CovAccStats
target  VarScr_CovPFstats
```

TOOL::<label>[::<pcks>]

This declaration is used to specify a build tool such as the Fortran compiler or its flags. The <label> determines the tool you are declaring. A TOOL declaration normally applies globally. However, where it is sensible to do so, a sub-package <pcks> can be specified. In which case, the declaration applies only to the files and directories under the sub-package. A list of <label> fields is available later in this annex.

Example:

```
tool::fc      sxmpif90
tool::fflags  -Chopt -Pstack

tool::cc      sxmpic++
tool::cflags  -O nomsg -pvctl nomsg

tool::ar      sxar
```

EXE_DEP[::<target>]

This declares an extra dependency for either all main program targets or only <target> if it is specified. If <target> is specified, it must be the name of a main program target. The value of the declaration is a space delimited list. Each item in the list can either be a valid name of a sub-package or the name of a valid object target. If a sub-package name is used, the *make* rule for the main program will be set to depend on all (non-program) object files within the sub-package.

Example:

```
# Only foo.exe to depend on the package foo::bar and egg.o
exe_dep::foo.exe  foo::bar egg.o

# All executables to depend on the package foo::bar and egg.o
exe_dep  foo::bar egg.o

# Only foo.exe to depend on all objects
exe_dep::foo.exe

# All executables to depend on all objects
exe_dep
```

BLOCKDATA[::<target>]

This declares a BLOCKDATA dependency for either all main program targets or only <target> if it is specified. If <target> is specified, it must be the name of a main program target. The value of the declaration is a space delimited list. Each item in the list must be the name of a valid object target containing a Fortran BLOCKDATA program unit.

Example:

```
# Only foo.exe to depend on blkdata.o
blockdata::foo.exe  blkdata.o

# All executables to depend on fbd.o
blockdata  fbd.o
```

EXCL_DEP[::<pcks>]

This declaration is used to specify whether a particular dependency should be ignored during the automatic dependency scan. If a sub-package <pcks> is specified, the declaration applies only to the files and directories under the sub-package. Otherwise, the declaration applies globally. The value of this declaration must contain one or two fields (separated by the double colon `::`). The first field denotes the dependency type, and the second field is the dependency target. If the second field is specified, it will only exclude the dependency to the specified target. Otherwise, it will exclude all dependency to the specified type. The following dependency types are supported:

USE
: The dependency target is a Fortran module.

INTERFACE
: The dependency target is a Fortran 9X interface block file.

INC
: The dependency target is a Fortran INCLUDE file.

H
: The dependency target is a pre-processor #include header file.

OBJ
: The dependency target is a compiled binary object file.

EXE
: The dependency target is an executable binary or script.

N.B. The following dependency targets are in the default list of excluded dependencies:

Intrinsic Fortran modules:

- USE::ISO_C_BINDING
- USE::IEEE_EXCEPTIONS
- USE::IEEE_ARITHMETIC
- USE::IEEE_FEATURES

Intrinsic Fortran subroutines:

- OBJ::CPU_TIME
- OBJ::GET_COMMAND
- OBJ::GET_COMMAND_ARGUMENT
- OBJ::GET_ENVIRONMENT_VARIABLE
- OBJ::MOVE_ALLOC
- OBJ::MVBITS
- OBJ::RANDOM_NUMBER
- OBJ::RANDOM_SEED
- OBJ::SYSTEM_CLOCK

Dummy declarations:

- OBJ::NONE
- EXE::NONE

Example:

```
excl_dep  USE::YourFortranMod
excl_dep  INTERFACE::HerFortran.interface
excl_dep  INC::HisFortranInc.inc
excl_dep  H::TheirHeader.h
excl_dep  OBJ
excl_dep  EXE
```

DEP::<pcks>

This declaration is used to specify a dependency for a source file in <pcks>. The value of this declaration must contain two fields (separated by the double colon `::`). The first field denotes the dependency type, and the second field is the dependency target. The dependency types are the same as those for EXCL_DEP described above.

Example:

```
dep::foo/bar.f  USE::your_fortran_mod
dep::foo/bar.f  INTERFACE::her_fortran.interface
dep::foo/bar.f  INC::his_fortran_inc.inc
dep::foo/bar.f  H::their_header.h
dep::foo/bar.f  OBJ::its_object.o
dep::foo/egg    EXE::ham
```

NO_DEP::<pcks>

This declaration is used to switch off/on dependency checking. If <pcks> is specified in the label, the declaration applies to the specified sub-package only.

Example:

```
# Switch on dependency checking only for "foo"
no_dep      true
no_dep::foo false
```

EXE_NAME::<name>

This renames the executable target of a main program source file <name> to the specified value.

Example:

```
# Rename executable target of foo.f90 from "foo.exe" to "bar"
exe_name::foo  bar
```

LIB[::<pcks>]

This declares the name of a library archive target. If <pcks> is specified in the label, the declaration applies to the library archive target for that sub-package only. If set, the name of the library archive target will be named `lib<value>.a`, where <value> is the value of the declaration. If not specified, the default is to name the global library `libfcm_default.a`. For a library archive of a sub-package, the default is to name its library after the name of the sub-package.

Example:

```
# Rename the top level library "libfoo.a"
lib  foo

# Rename the library for the sub-package "egg::ham"
# from "libegg__ham.a" to "libegg-ham.a"
lib::egg/ham  egg-ham
```

PP[::<pcks>]

This declares whether a pre-processing stage is required. To switch on pre-processing, set the value to `true`. If <pcks> is specified in the label, the flag applies to the files within that sub-package only. Otherwise, the flag affects source directories in all packages. The pre-processing stage is useful if the pre-processor changes the dependency and/or the argument list of the source files. The default behaviour is skip the pre-processing stage for all source.

Example:

```
pp::gen true  # switch on pre-processing for "gen" only
pp      true  # switch on pre-processing globally
```

SRC_TYPE::<pcks>

This declaration is used to (re-)register the file type of the sub-package <pcks> to associate with different file types. The value of the declaration is a list of type flags delimited by the double colon `::`. Each type flag is used internally to describe the nature of the file. For example, a Fortran free source form containing a main program is registered as `FORTRAN::FORTRAN9X::SOURCE::PROGRAM`. A list of type flags is available later in this annex.

Example:

```
src_type::foo/bar.f  FORTRAN::FORTRAN9X::SOURCE::PROGRAM
```

INFILE_EXT::<ext>

This declaration is used to re-register particular file name extensions <ext> to associate with different file types. The value of the declaration has a similar format to that of SRC_TYPE declaration described above. A list of type flags is available later in this annex.

Example:

```
infile_ext::h90  CPP::INCLUDE
infile_ext::inc  FORTRAN::FORTRAN9X::INCLUDE
```

OUTFILE_EXT::<type>

This declaration is used to re-register the output file extension for a particular <type> of output files. The value must be a valid file extension. The following is a list of output file types in-use by the build system:

OBJ
: compiled object files
[default = .o]

MOD
: compiled Fortran module information files
[default = .mod]

EXE
: binary executables
[default = .exe]

DONE
done files for compiled source
[default = .done]

IDONE
done files for included source
[default = .idone]

FLAGS
flags files, compiler flags config
[default = .flags]

INTERFACE
interface files for F9X standalone subroutines/functions
[default = .interface]

LIB
archive object library
[default = .a]

TAR
TAR archive
[default = .tar]

Example:

```
# Output F9X interface files will now have ".foo" extension
outfile_ext::interface  .foo
```

The following is a list of <label> fields that can be used with a `TOOL` declaration. Those marked with an asterisk (*) accept declarations at sub-package levels.

FC
The Fortran compiler.
[default = `f90`]

FFLAGS *
Options used by the Fortran compiler.
[default = ""]

FC_COMPILE
The option used by the Fortran compiler to suppress the linking stage.
[default = `-c`]

FC_INCLUDE
The option used by the Fortran compiler to specify the include search path.
[default = `-I`]

FC_MODSEARCH
The option used by the Fortran compiler to specify the search path for the compiled module definition files. This option is often unnecessary as it is normally covered by the include search path.
[default = ""]

FC_DEFINE
The option used by the Fortran compiler to define a pre-processor definition macro.
[default = `-D`]

FC_OUTPUT
The option used by the Fortran compiler to specify the output file name.
[default = `-o`]

CC
The C compiler.
[default = `cc`]

CFLAGS *
Options used by the C compiler.
[default = ""]

CC_COMPILE
: The option used by the C compiler to suppress the linking stage.
[default = `-c`]

CC_INCLUDE
: The option used by the C compiler to specify the include search path.
[default = `-I`]

CC_DEFINE
: The option used by the C compiler to define a pre-processor definition macro.
[default = `-D`]

CC_OUTPUT
: The option used by the C compiler to specify the output file name.
[default = `-o`]

LD *
: Name of the linker or loader for linking object files into an executable. If not set, use the compiler of the source file containing the main program.
[default = ""]

LDFLAGS *
: The flags used by the linker or loader.
[default = ""]

LD_OUTPUT
: The option used by the linker or loader for the output file name (other than the default `a.out`).
[default = `-o`]

LD_LIBSEARCH
: The option used by the linker or loader for specifying the search path for link libraries.
[default = `-L`]

LD_LIBLINK
: The option used by the linker or loader command for linking with a library.
[default = `-l`]

AR
: The archive command.
[default = `ar`]

ARFLAGS
: The options used for the archive command to create a library.
[default = `rs`]

FPP
: The Fortran pre-processor command.
[default = `cpp`]

FPPKEYS *
: The Fortran pre-processor will pre-define each word in this setting as a macro.
[default = ""]

FPPFLAGS *
: The options used by the Fortran pre-processor.
[default = `-P -traditional`]

FPP_DEFINE
: The option used by the Fortran pre-processor to define a macro.
[default = `-D`]

FPP_INCLUDE
: The option used by the Fortran pre-processor to specify the include search path.
[default = `-I`]

CPP
: The C pre-processor command.
[default = `cpp`]

CPPKEYS *
: The C pre-processor will pre-define each word in this setting as a macro.
[default = ""]

CPPFLAGS *
: The options used by the C pre-processor.
[default = `-C`]

CPP_DEFINE
: The option used by the C pre-processor to define a macro.
[default = `-D`]

CPP_INCLUDE
: The option used by the C pre-processor to specify the include search path.
[default = `-I`]

MAKE
: The `make` command.
[default = `make`]

MAKEFLAGS
: The options used by the `make` command.
[default = ""]

MAKE_SILENT
: The option used by the `make` command to specify silent operation.
[default = `-s`]

MAKE_JOB
: The option used by the `make` command to specify the number jobs to run simultaneously.
[default = `-j`]

GENINTERFACE *
: The command/method to extract the calling interfaces of top level subroutines and functions in a Fortran 9X source. Supported values are `f90aib` and `none` (to switch off interface generation). If not specified, the system will use its own internal logic.
[default = (not specified)]

INTERFACE *
: Generate Fortran 9X interface files with root names according to either the root name of the source `file` or the name of the `program` unit.
[default = `file`]

The following is a list of type flags that are currently in-use (or * *reserved*) by the build system for TYPE and INFILE_EXT declarations:

SOURCE
: a source file containing program code of a supported language (currently Fortran, FPP, C and CPP).

INCLUDE
: an include file containing program code of a supported language (currently Fortran, FPP, C and CPP).

FORTRAN
: a file containing Fortran code.

FORTRAN9X
: a file containing the Fortran free source form. This word must be used in conjunction with the word `FORTRAN`.

FPP
: a file containing Fortran code requiring pre-processing.

FPP9X
: a file containing Fortran free source form requiring pre-processing. This word must be used in conjunction with the word `FPP`.

C
: a file containing C code.

CPP
: a file containing CPP include header.

INTERFACE
: a file containing a Fortran 9X interface block.

PROGRAM
: a file containing a main program.

MODULE
: a file containing a Fortran 9X module.

BINARY
: a binary file.

EXE
: an executable file. This word must be used in conjunction with the word `BINARY`.

LIB
: an archive library. This word must be used in conjunction with the word `BINARY`.

SCRIPT
: a file containing source code of a scripting language.

PVWAVE
: a file containing executable PVWAVE scripts. This word must be used in conjunction with the word `SCRIPT`.

SQL
: a file containing SQL scripts. This word must be used in conjunction with the word `SCRIPT`.

GENLIST
: a GEN List file.

OBJ
: *(* reserved)* an object file. This word must be used in conjunction with the word `BINARY`.

SHELL
: *(* reserved)* a file containing executable shell scripts. This word must be used in conjunction with the word `SCRIPT`.

PERL
: *(* reserved)* a file containing executable Perl scripts. This word must be used in conjunction with the word `SCRIPT`.

PYTHON
: *(* reserved)* a file containing executable Python scripts. This word must be used in conjunction with the word `SCRIPT`.

TCL
: *(* reserved)* a file containing executable TCL scripts. This word must be used in conjunction with the word `SCRIPT`.