

CCP4 Programmers' Handbook

Version of 14th November 2008

CCP4 Programmer's Handbook: Contents

1 Introduction.....	6
1.1 Aims of the handbook.....	6
1.2 Contents of the handbook.....	6
1.3 Other useful documents.....	6
1.4 Philosophy: why bother with support?.....	7
1.5 History.....	7
1.6 Credits.....	8
2 Technical Support – Helpdesk Duties.....	9
2.1 Introduction.....	9
2.2 Responsibilities of the Duty Programmer.....	9
2.3 Dealing with technical support queries.....	9
2.3.1 Procedure for dealing with requests.....	10
2.3.2 Assignment of bugs logged in Bugzilla.....	10
2.3.3 Available resources for resolving problems.....	10
2.3.4 Resolving bugs and resulting actions.....	11
2.3.5 Reporting bugs and solutions to the user community.....	11
2.3.6 Bugs that cannot be resolved.....	11
2.3.7 Issues with monitoring and control.....	12
2.4 Administration of the CCP4 Majordomo lists.....	12
2.4.1 Introduction.....	12
2.4.2 Tasks involved in administering the list manager account.....	12
2.4.3 Tasks involved in administering the lists.....	13
2.4.4 Administering the list using the web interface “MajorCool”.....	14
2.4.5 Majordomo commands for manual list administration.....	14
2.4.6 Lists and Passwords.....	14
2.5 Administration of CCP4bb via JISCmail.....	15
2.5.1 Introduction.....	15
2.5.2 Overview of administrative procedure.....	15
2.5.3 Logging into the JISCmail site.....	15
2.5.4 Procedure for adding a new address to the list.....	16
2.5.5 Checking the details of a subscriber.....	18
2.6 Administration of the CCP4 web pages.....	19
2.6.1 MySQL databases.....	19
3 Reporting Problems on the Website.....	20
3.1 Overview of web pages used to report problems.....	20
3.2 General guidelines for reports.....	20
3.3 Reporting Problems on the CCP4 Problems Pages.....	21
3.4 Making code fixes available on the FTP server.....	21
3.4.1 Background: patch files versus patched files.....	21
3.4.2 Making source code patches manually.....	22
3.4.3 Making source code patches from CVS using make patch.sh.....	22
3.4.4 Placing fixes on the FTP server and maintaining the index files.....	23
3.4.5 Naming conventions for patch and patched files.....	24
3.4.6 Making binary patches.....	24

3.4.7 Other resources.....	24
3.5 Alerting users through the Update Alert Mechanism.....	24
Adding and updating programs.....	26
3.6 Updating programs.....	26
3.6.1 Overview of the procedure for updating programs.....	26
3.6.2 Updating the code.....	26
3.6.3 Tagging new versions of programs in CVS.....	26
3.6.4 Testing updates.....	27
3.7 Adding New Programs To The Suite.....	27
3.8 Special Procedures.....	28
3.8.1 Generate Refmac5 harvesting template for RCSB.....	28
3.8.2 Updating mmCIF dictionary used by CCIF libraries.....	28
4 Software development.....	30
4.1 Style Guides.....	30
4.2 Programming Practice.....	30
4.3 Documentation.....	31
4.4 Other resources.....	31
4.4.1 Template Programs.....	31
4.4.2 Porting Guide.....	31
4.4.3 CCP4i Developers Area.....	31
4.4.4 The CCP4 Manual.....	31
5 Tools and resources.....	33
5.1 The CCP4 CVS Patch Branch.....	33
5.1.1 Preliminaries: about CVS Concurrent Version System.....	33
5.1.2 Introduction: what is the Patch Branch?.....	33
5.1.3 Which code fixes should be included?.....	33
5.1.4 Procedure for adding fixes to the patch branch.....	34
5.1.5 Making patch files from the patch branch.....	34
5.1.6 Merging the patch branch back into the main branch.....	34
5.2 Bugzilla.....	36
5.2.1 Introduction: what is Bugzilla?.....	36
5.2.2 Information that should be recorded in Bugzilla.....	36
5.2.3 Definitions of RESOLUTIONs (and when to use them).....	37
5.3 Autobuilds.....	39
5.3.1 Introduction: what are the CCP4 Autobuilds?.....	39
5.3.2 Autobuild CVS archive and documentation.....	39
5.3.3 Deployment of the autobuilds at DL.....	39
5.3.4 Maintaining the autobuilds.....	40
5.3.5 Hints for deploying an autobuild on a new machine.....	40
5.3.6 Setting up autobuilds for patch branches.....	41
5.3.7 Common problems with the Autobuilds (and how to fix them).....	42
5.3.8 Limitations.....	42
5.4 Exporting “onlylibs” distributions of CCP4.....	44
5.5 Other tools and resources.....	44
5.5.1 Valgrind.....	44
5.5.2 mtzdiff, pdbdiff and mapdiff.....	44
5.5.3 Test suite and test data.....	44
5.5.4 Code checkers and debuggers.....	44

<u>6 Book List.....</u>	<u>46</u>
--------------------------------	------------------

1 Introduction

This document is intended to be a reference manual for the CCP4 programmers based at CCLRC Daresbury Laboratory. The aim is to lay down guidelines, provide protocols for common tasks and document useful knowledge.

1.1 *Aims of the handbook*

The aim of this document is formalise the provision of support offered by the CCP4 “duty person”.

- Ensure that everyone knows what is expected
- Provide a common standard for the provision of support
- Address problem areas by explicitly stating the protocols used
- Limit the amount of time that each individual spends on support activities
- Act as a repository of general knowledge about supporting and developing the core CCP4 suite.

1.2 *Contents of the handbook*

The handbook is broken down into a number of different sections.

Section 2 outlines the duties and responsibilities involved in providing technical support for the CCP4 suite, and gives overviews of the procedures involved in dealing with technical support requests (**section 2.3**) and administering the CCP4-maintained email lists (**section 2.4**).

The following sections then deal with some of the more general aspects of maintaining the software suite, namely reporting problems on the website (**section**), adding and updating programs within the suite (**section**) and general software development within the CCP4 framework (**section 4**). These topics are separated from the sections on technical support as any member of staff, not just those on duty, may perform them.

Finally **section 5** gives details of the various software tools that are used in the maintenance of the suite.

1.3 *Other useful documents*

Two other documents are recommended as companions to this handbook:

- **CCP4 Manual:** although becoming somewhat dated in places, the CCP4 manual remains an invaluable reference for many of the technical aspects of the suite.

- **CCP4 Release Handbook:** which details the procedures involved in making releases of the CCP4 suite.

1.4 *Philosophy: why bother with support?*

Context: Software seems to be subject to entropy. Things that used to work stop working. Ideas that seemed good at the time, seem poor 3 months later. Interfaces get ugly. It's easy to find yourself surrounded by code that has niggling things wrong with it, or that's just plain bad.

The problem: When you're surrounded by ugly things, your attitude and outlook change. You become pessimistic, and your expectations are lowered. Soon you start accepting that "that's just how things are." You start producing ugly things yourself. Researchers¹ have discovered a trigger that leads to urban decay. Once a window in a building is broken and left unrepaired, the building starts to go downhill rapidly. A car can be left on a street for a week, but break one of its windows, and it will be gutted in hours.

Therefore: Don't live with broken windows. Fix things when you see them. Refactor when you can. If you can't make the change right then, put some kind of flag in place (like a [FixmeComment](#)) to at least acknowledge that you've recognized a problem, and to tell people that someone cares and is on top of it. Stop rot while it's isolated.

The Flipside: People respect well maintained things. Beautifully restored vintage cars are treated with reverence. People take off their shoes when entering well-cared for houses. A good way to get other developers to treat your systems with respect is to keep it looking cared-for.

(From <http://c2.com/cgi/wiki?FixBrokenWindows>)

1.5 *History*

The content of this document has been updated over time. The most recent significant additions are:

- *10th July 2007:* Added section 2.5 with information on administering the CCP4bb list on JISCmail.
- *5th October 2006:* Updated section 2.4 with additional information on the Majordomo lists and passwords at Daresbury.
- *27th January 2006:* Updated section 2.5.5 with information on the automation patching procedure to be used in release 6.0. Updated section 4 on software development within CCP4. Updated section 5 on Tools and Resources.
- *24th September 2007:* Updated section on making fixes available as patches and patched files (section 3.4). Added information on updating the mmCIF dictionary in a new section on "Special procedures" (section 3.8). Added some notes on merging the patch branch back into the main branch (section 5.1)

¹ Wilson/Kelling, *Atlantic Monthly*, March '82

- *22nd January 2008*: Added information on how to update the autobuilds to use new machines and include new CVS modules.
- *5th February 2008*: Added link to webinterface for Majordomo.

1.6 Credits

The bulk of this document was originally written and edited by Peter Briggs, with contributions (intentional and otherwise) from Martyn Winn, Alun Ashton and Charles Ballard. Subsequently material has been provided by Ronan Keegan for incorporation by Peter Briggs.

2 Technical Support – Helpdesk Duties

2.1 Introduction

The following sections outline the duties and responsibilities of the “duty programmer” – the member of staff who is responsible for dealing with CCP4 technical support (the “helpdesk”) and other administrative tasks – and the procedures for performing these duties.

2.2 Responsibilities of the Duty Programmer

Each week of cover officially begins at 5pm on Fridays to 5pm, which should avoid messages received at the weekend “falling between the cracks”. The duty roster is on the web at http://www.ccp4.ac.uk/group/duty_roster.html.

The duty programmer has responsibility during this time for:

- Dealing in the first instance with technical support messages, updates and general queries sent to ccp4@ccp4.ac.uk and ccp4gui@ccp4.ac.uk. These are covered in section 2.3.
- Administering the CCP4 mailing lists ccp4bb and ccp4-dev. This is covered in sections 2.4 and 2.5).

2.3 Dealing with technical support queries

Technical support queries come in from the ccp4@ccp4.ac.uk and ccp4gui@ccp4.ac.uk email addresses. The person currently providing technical support cover (“the duty programmer”) has the responsibility for ensuring that the queries are addressed.

The requests include:

- Problems with software installation or usage
- Technical advice
- Requests for new or improved features
- Patches or updated versions of existing packages
- Offers of new software
- Licensing questions and enquiries
- Problems or issues to do with the mailing lists

The protocols for dealing with each of these are outlined in separate sections in this document.

Programmers not currently providing technical support cover may unilaterally decide to respond to queries sent to these addresses, in which case they should ensure that that they inform the duty programmer to avoid duplication of effort.

An individual not providing duty cover who receiving a general technical support query may bounce or forward the message to ccp4@ccp4.ac.uk, so that the duty person can deal with it.

At the discretion of the duty person relevant technical queries sent to the bulletin board (ccp4bb and ccp4-dev) may also be considered as technical support queries.

2.3.1 Procedure for dealing with requests

The person providing duty cover should send an initial response to the sender within two working days of receiving the message, and the request should be logged in the Bugzilla issue tracker (see section 5.2 for a detailed description of using Bugzilla). The initial response can be a “holding message” or acknowledgement.

The initial response should be copied (i.e. cc'ed) to the ccp4@ccp4.ac.uk list. At the duty person's discretion the number assigned by Bugzilla should be included in the subject line of the reply.

Before responding the queries should be entered into Bugzilla (see section 5.2.2 on logging and tracking in Bugzilla for details of what information should be recorded). At the duty person's discretion certain common types of request may be exempted from logging, for example:

- Problems with the mailing lists (ccp4bb, ccp4-dev)
- Requests for licence information
- Other queries where a single reply is sufficient to deal with the request.

2.3.2 Assignment of bugs logged in Bugzilla

Bugs logged in Bugzilla as a result of a query should be assigned in the first instance to the duty programmer. It is the responsibility of the duty programmer to ensure that bugs are resolved or else that the responsibility for resolving them is taken on by someone else.

2.3.3 Available resources for resolving problems

It is recommended that the duty programmer check the available resources before launching into resolving the problem. A rough protocol is:

- Check the problems page for the current release
- Check the general problems pages for the specific component and operating system (we need a list of these)
- Search Bugzilla

- Check with the person who has specialist knowledge of this aspect of the suite (in this case they may be expected to address the problem, for example in the case of programs such as REFMAC).

2.3.4 Resolving bugs and resulting actions

Bugzilla offers a number of different “resolutions” for bugs. Section 5.2.3 has suggested guidelines for what these mean in the context of the CCP4 project, and what actions might follow.

2.3.5 Reporting bugs and solutions to the user community

The resolution of a query may lead to the identification and resolution of a problem that should be advertised to the wider community. To an extent it is a judgement call as to how serious a problem must be for this action to be taken. A rough guide is that if it could be expected that a large number of people will be affected by the problem then it should be advertised as follows:

- A note made on the CCP4 problems page for the current release (see section 2.5.5 for details on how to do this)
- If appropriate, the necessary code changes required to address the problem should be made available on the ftp server (see section 3.4)
- The fix should be added into the CVS patch branch for the current release (see section 5.1 for information on the patch branch).

The procedures for doing these things are outlined in section 2.5.5 of this handbook.

2.3.6 Bugs that cannot be resolved

Note that it is not always possible to resolve problems that arise from technical support queries. This can happen for a number of reasons, for example:

- The user supplies insufficient information, or doesn't respond to messages requesting further information
- The programming staff lack sufficient knowledge to trace the problem or resolve it
- The time it would take to resolve the problem is not commensurate with its seriousness

In these cases the programmer should use their discretion to resolve the problem, for example by documenting known bugs or disabling unfixed functions.

“Lapsed” problems (when the reporter hasn't responded to confirm a fix or provide more information) should have some time-out period (say, one month from the last contact) after which the bug report should be closed.

We should endeavour to do the best that we can under the circumstances, and recognise that these circumstances might often vary depending on other commitments, available manpower, and our own skills and knowledge.

2.3.7 Issues with monitoring and control

There are currently no procedures for monitoring the bugs logged in Bugzilla, and so there remains the danger that Bugzilla becomes a bin for unresolved bugs. Some monitoring procedure needs to be in place to ensure that bug reports logged in the system are resolved. In some cases this may mean deciding to close minor unresolved bugs in order that programmers are not overwhelmed with a backlog of bugs.

There are also no formal procedures for “load balancing” – recognising that some weeks are busier than others (particularly in the period immediately following a release), or that some jobs should be reallocated to others (because for example it's their particular area of expertise).

2.4 Administration of the CCP4 Majordomo lists

2.4.1 Introduction

Majordomo is a program used to manage email lists. The following sections outline the email lists that CCP4 maintains from Daresbury, and the tasks involved in administering both the list manager account and the lists themselves.

The CCP4bb (general protein crystallography) mailing list is no longer a Majordomo list - see section 2.5 for information on managing CCP4bb via the JISCmail service. The remaining key list managed at Daresbury for the wider community is now only CCP4-dev (protein crystallography software development list).

However there are a number of other lists that are also administered here – see section 2.4.6. Note that for these two main lists, only people with email addresses subscribed to the lists are able to post to them but otherwise the lists are unmoderated.

2.4.2 Tasks involved in administering the list manager account

In addition to dealing with specific queries about the mailing lists there are some general administrative duties.

The mail administrator account should be cleared out regularly by logging into ccp4h as ccp4null and starting up the **pine** mail browser. This has filters set up to clear out much of the junk email that accumulates in the administrator's inbox. (Use **ssh** and not the **su** command, as the filters won't work in the latter case.)

The password for the ccp4null account is “ccp4ondlpx1”.

The filters move messages to the following folders:

- owner-ccp4bb
- BOUNCE
- majordomosends
- spam
- others ...

Certain actions should be taken for certain types of messages, see below. It is important that this is done regularly in order to prevent the inbox filling up with an intractable amount of mail.

2.4.3 Tasks involved in administering the lists

Majordomo lists are administered by sending “approve” requests to the list via email. Usually this type of administration duty arises as the result of a specific support query. The table below outlines the most common queries, the likely causes, and the suggested action to take in order to resolve it.

Problem	Likely Cause	Suggested Action
Subscriber cannot unsubscribe	• User no longer has access to an old account	Unsubscribe the old email address on the user's behalf
Subscriber cannot post to the list	• User isn't actually subscribed to the list • User is sending a message from a different account from the one that they are subscribed with	Tell the user to (re)subscribe, or subscribe them yourself. Or send the message to the list on their behalf.
Subscriber reports that they no longer receive messages from the list	• Subscriber has changed email address • Subscriber's address is no longer on the list	Tell the user to subscribe with their new address, or subscribe them yourself. NB sometimes this is a sign of problems so it is wise to check that the list still has subscribers!

Subscriber reports that they receive multiple copies of the same posting to the list	They are subscribed with multiple email addresses	Check the list of subscribers and search for likely duplicated email addresses.
	There is a dead email address in the same block of addresses as the subscriber	Check the header of the duplicated messages for clues as to which address may be dead, and remove it from the list.

2.4.4 Administering the list using the web interface “MajorCool”

There is a web interface that provides a simple way of issuing list commands, it can be accessed via

<https://dlmail.dl.ac.uk/majorcool?module=modify>

Otherwise you can issue the commands to Majordomo “manually” via email, which is described in the following section.

2.4.5 Majordomo commands for manual list administration

Commands are issued to Majordomo by sending an email to majordomo@dl.ac.uk with the required commands in the message body. There are two key commands used for administering the lists:

who *list*

Get a list of all addresses subscribed to *list*

approve *password* **[un]subscribe** *list address*

(Un)subscribe *address* from list

Majordomo has a large number of other commands that can be used for example to retrieve archives of old messages, change the banner or welcome message, and configure other attributes of the list. The full list of commands can be obtained from Majordomo using **help** *list*.

2.4.6 Lists and Passwords

The following table gives the current passwords for the main Majordomo lists administered at DL:

List name	Password	Description
Ccp4	4pcc	CCP4 general email address
Ccp4staff	c1c2p34	CCP4 STFC staff local information list

Ccp4-admin	adminofc4	CCP4 and STFC staff dealing with commercial companies & invoicing
Ccp4-dev	ved4pcc	Protein crystallography software development
Ccp4wg1	4pc-awg1	CCP4 Working Group 1 circulation
Ccp4wg2	4pc-awg2	CCP4 Working Group 2 circulation
Ccp4-auto	-	CCP4 Automation Project forum
3dccp4	-	(Defunct) Mailing list for CCP4 molecular graphics

2.5 Administration of CCP4bb via JISCmail

2.5.1 Introduction

As of late 2006 the CCP4bb list was moved from Majordomo to the National academic mailing service at JISCmail (<http://www.jiscmail.ac.uk>). As a result the procedures for administering CCP4bb are different from those required to maintain the Majordomo list; the procedure is described here in detail.

2.5.2 Overview of administrative procedure

The JISCmail system requires that the list administrators explicitly approve requests for new users to be added to the list by the list owner. There are two list owners: one is Charles Ballard, the other is the general ccp4null account. The ccp4null account is the one that is used for list administration.

To check the requests to be added to the list, log into the ccp4null account on ccp4h (password is "ccp4ondlpx1") and check the mailbox for this account using the pine email program from the command line. Pine will automatically filter the messages so you should only need to check for requests by looking in the **majordomosends** folder.

There should be two types of request:

- **Notifications of subscribers leaving the list:** these can be ignored; they require no action on the administrator's part.
- **Request to add a new address to the list:** this does require action. This is outlined in the following section.

2.5.3 Logging into the JISCmail site

To log into the JISCmail site to administer the list:

- Go to <http://www.jiscmail.ac.uk>
- Click the *List Management* link
- Enter the e-mail address ccp4null@dl.ac.uk and the password "ccp4ondlpx1"
- Select *Login using cookie* if you want to remain logged in for future sessions

2.5.4 Procedure for adding a new address to the list

There are two ways to add someone to the list:

- **Follow the link given in the email request:**

In the email request the highlighted link in figure 1 can be pasted into a browser in order to action the request:

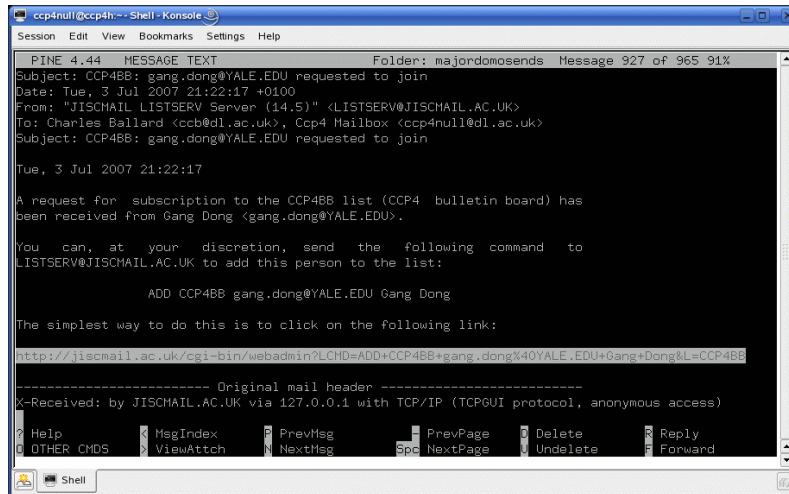


Figure 1: Pine window with example email requesting to join the list

- **Execute the ADD command via the JISCmail web interface**

Go directly to the JISCmail site, go to the *List Management* screen (see figure 2):

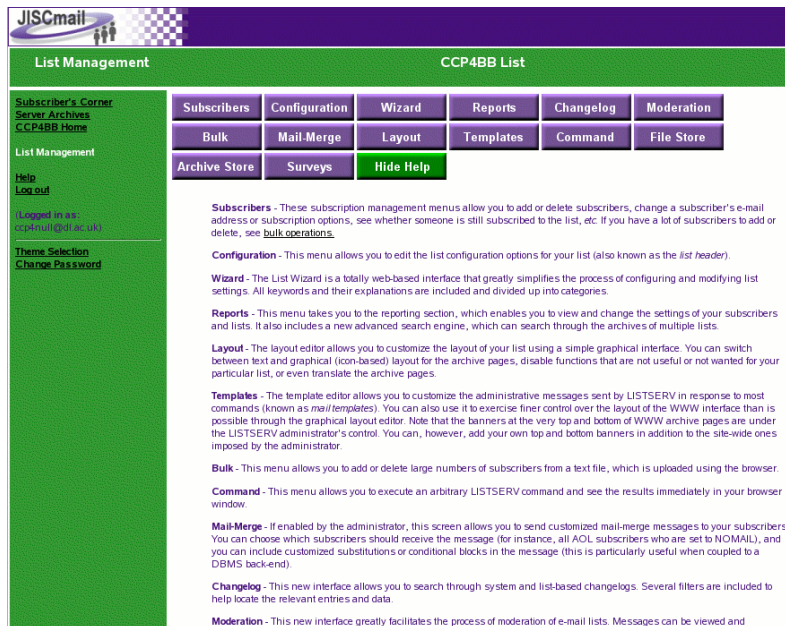


Figure 2: the JISCmail "List Management" screen

Select the **Command** button link (one of the purple buttons at the top of the screen shown in figure 2) to execute the “ADD...” command that is included in the email request body.

(Note: the Command button is one of a large cluster of buttons at the top of the screen and may not be immediately obvious on first inspection!).

The web interface for executing the command is shown in figure 3.

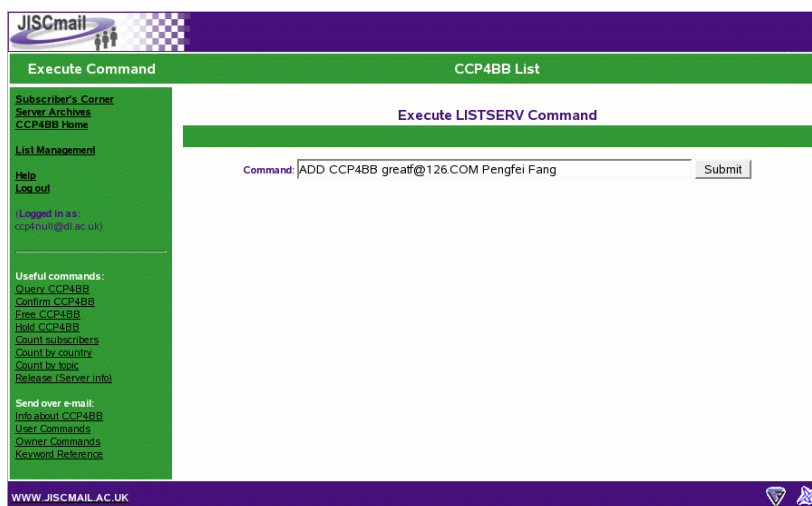


Figure 3: the JISCmail web interface for executing a listserv command

To execute the command to add the new subscriber, copy the whole ADD command from the Pine window and paste it into the form, then hit the **Submit** button.

In either case, if the addition is successful then the web interface will report this:

Execute LISTSERV Command

```
P J Briggs <p.j.briggs@DL.AC.UK> has been added to the CCP4BB list .
Notification has been sent to p.j.briggs@DL.AC.UK .
```

New Command:

Figure 4: JISCmail reporting success after adding a new email address

If the user already exists then you will be told that they are already on the list, so there is no problem if you make a mistake and accidentally try to add the same address more than once:

Execute LISTSERV Command

```
greatf@126.COM is already subscribed to the CCP4BB list under the name
"Pengfei Fang".
```

New Command:

Figure 5: JISCmail reporting that an email address is already subscribed to the list

It is recommended that you delete the request emails from the **majordomosends** folder after actioning them.

2.5.5 Checking the details of a subscriber

Sometimes it may be necessary to check the details of existing subscribers. The JISCmail web interface allows you to look up a name or email address by selecting the **Subscribers** button on the **List Management** screen shown in figure 2. This will bring up a screen that allows the list to be easily searched (figure 6).

The screenshot shows the JISCmail web interface for the CCP4BB List. On the left is a green sidebar with navigation links: Account Management, Subscriber's Corner, Server Archives, CCP4BB Home, List Management, Help, Log out, Logged in as: admin@dl.ac.uk, Subscriber Reporting, and Bulk Operations. The main content area has a purple header with the JISCmail logo and 'CCP4BB List'. Below this is a section titled 'Examine or delete a subscription' with a 'Name or Address:' input field and 'Search in CCP4BB' and 'Clear' buttons. A horizontal line separates this from the 'Add a new user to the list' section, which has an 'E-mail Address and Name:' input field, radio buttons for 'Send welcome message' (selected) and 'Do not notify the user in any way', and 'Add to CCP4BB' and 'Clear' buttons. Another horizontal line separates this from the 'Review the list membership' section, which contains a note: 'You can review the list membership either in a browser window or by e-mail. You should avoid using a browser window if the list has more than about 500 subscribers - it will be slow and impractical, and could even crash your browser.'

Figure 6: JISCmail web interface for searching the list of subscribers

2.6 Administration of the CCP4 web pages

2.6.1 MySQL databases

A lot of the information displayed on the CCP4 web pages is actually held in a MySQL database and accessed via PHP scripts. For example, the people information displayed at <http://www.ccp4.ac.uk/staff/people.php> is held in a database called "staff". The people information can be updated via a form available at <http://www.ccp4.ac.uk/group/staff/editpage.php>. The required MySQL password is "SecureEdit".

3 Reporting Problems on the Website

3.1 Overview of web pages used to report problems

The CCP4 website has a number of resources relating to problems using the software:

- Problems with the current release
<http://www.ccp4.ac.uk/problems.php>

This has subsections for each patch release within the current release “family”, and is further broken down into the following sections:

- o Installation
 - o Programs
 - o Libraries
 - o Graphical user interface
 - o Documentation, Tutorials and Examples
- Non-version-specific problems
<http://www.ccp4.ac.uk/problems/general-probs.php>
- Non-version-specific platform-specific problems for Windows, Linux, Mac, Sun, OSF1, IRIX, AIX and others (a number of pages linked from the other problem pages)
- CCP4i Troubleshooting Page (general problems with CCP4i)
http://www.ccp4.ac.uk/ccp4i/trouble_shooting.html

There are also links to e.g. MOSFLM problems pages but these are external links and are not maintained by CCP4 at Daresbury.

3.2 General guidelines for reports

There is a lot of information on these pages and it is not always well organised either for CCP4 staff or for external users trying to find information.

This will always be a balance between what is convenient and practical for staff to maintain, as against what is easy and useful for users of the software. These guidelines try to set out some general principles.

- **Always date the information you add**
Recent problem reports (and associated fixes) are likely to be more relevant than older ones.
- **Make summaries as concise and descriptive as possible**

This helps people to find the information that they want quickly.

- **Reference any relevant bug reports logged in Bugzilla**
This will make it easier for other CCP4 staff members to look up more detailed information if they need it.
- **Include links to patch or replacement files**
See section 3.4 below for how to make patches and make them available on the FTP server.
- **Try to be consistent with the format of the report page**
Each of the web pages above has its own format for reporting problems and fixes. User expectations are that even if the format varies between pages, there will be consistency within each page, so please observe consistent formatting when adding new problems and fixes.

3.3 Reporting Problems on the CCP4 Problems Pages

Decide which of the above areas the problem comes under (specific to the current release, specific to a particular platform etc) in order to decide which page the information should be added to, and which section (if relevant).

The pages themselves are held on the CCP4 web server and can be accessed via `dlpx1` at `/ccpdisk/ccp4/public/www/`.

For the main problems page at `/ccpdisk/ccp4/public/www/problems.php`, there is a template (within a html comment) which should be copied and pasted into the document. The fields in the template should then be filled in; in doing so the various pieces of information outlined above should automatically be taken care of.

3.4 Making code fixes available on the FTP server

3.4.1 Background: patch files versus patched files

For those problems that require code fixes in order to be resolved, the code fixes should be made available on the FTP server. Traditionally a code fix for CCP4 software takes one of two forms, either a patch file or a patched file.

A **patch file** (or **patch** for short) is a file with just the changes between the original and the fixed versions of the file. Patches are typically generated using the UNIX diff program and are applied using the patch program. Patches are normally used for relatively small (as in number of lines affected) fixes to source code files for compiled programs or libraries. It is possible to make binary patches but we don't normally do this in practice.

Patches are good because they (generally) allow the fix to be applied over any other changes that have already been made in the file in question; however they are not so good for Windows platforms since there is no equivalent of the patch program available as a default.

Conversely, a **patched file** is a complete file with the appropriate fixes already applied. Basically it's just an updated copy of the file with the bug in it. Patched files are normally used for cases where the changes to the source code file are relatively large, for binaries, and for fixes that are required to be applied across Linux/UNIX and Windows platforms (for example fixes to Tcl/Tk files for CCP4i).

Following CCP4 6.0.2 a policy has been adopted to keep patches and patched files in separate locations on the FTP server. However the suggested procedure in both cases is otherwise broadly similar:

1. Make either a patch file or a patched file (whichever seems most appropriate) containing the fix. Patch files can be generated either manually or using the `make_patch.sh` script; patched files are usually made in an ad-hoc fashion.
2. Move the patch file to the appropriate directory on the FTP server and update the relevant index file.
3. If the patch or patched file provides a new problem fix, also add a link to the file on the problems pages.

The following sections describe the procedures in more detail.

3.4.2 Making source code patches manually

Source code patches that can be applied using the patch program are most usefully made using the "context" mode of the diff command (the `-c` option), with the old version of the file appearing the command line before the newer version:

```
% diff -c oldfile newfile > patchfile.diff
```

The CVS diff command supports the same syntax, so for example:

```
% cvs diff -c -r1.1 -r1.2 myfile > patchfile.diff
```

Context diffs are more robust than "standard" diffs because the patch program can use the additional lines of context in order to correctly apply changes even if the line numbers don't match. The order that the files appear in is important because the diff tells patch how to transform the first file into the second – if the order is reversed, then so is the transformation.²

3.4.3 Making source code patches from CVS using `make_patch.sh`

² In practice patch implementations are usually smart enough to detect an incorrect ordering and give some warning. However it's better to get it right in the first place.

It is expected that source code patches should be based on fixes added to the CVS tree, and a tool is available to help minimise the amount of effort involved. The script `make_patch.sh` is in `$CETC` in the CVS checked-out version of CCP4 (it is not in final exported releases).

Usage is:

```
% make_patch.sh [-d CVSROOT] old_rev new_rev subdir file
```

`old_rev` and `new_rev` are CVS revision numbers or tags. The `subdir` is the path from `$CCP4` to the directory containing `file`, which is the file to generate the diff from. The script generates a context diff file called `file-old_rev-new_rev.diff`, which contains the patch required to go from revision `old_rev` to revision `new_rev`.

If the file is in `$CCP4` then use `“.”` as the `subdir` argument.

The script uses the `rdiff` option of CVS so you can run this script in any location provided that the `CVSROOT` environment variable is set correctly – if `CVSROOT` is not set then you can use the `-d` option (analogous to the `-d` option of the CVS command) to set it explicitly.

The script also generates the line that needs to be added to the `index.patches` file, or if such a file exists in the current directory, will actually add the appropriate line automatically. See section 3.4.4 for information on the `index.patches` file.

3.4.4 Placing fixes on the FTP server and maintaining the index files

- **Patch files**

Anything that is a patch (`.diff`) file should be placed in the `patches` subdirectory of the current release area on the FTP server:

```
/public2/ftp/pub/ccp4/current/patches/
```

The patch files can be linked from the CCP4 problems pages as described in section 3.3. Note that this area is used only for patches to that particular version of the release.

The ftp area should also contain a file called `index.patches`, which lists each of the patch files followed by the subdirectory (relative to `$CCP4`) where the patch should be applied. The index file supports the `patch_ccp4.sh` update script, which can fetch all `.diff` files plus the index file, and use the information to apply source code patches automatically to the user's installation.

Patches are references in the index file as follows:

```
<patch_file_name> <subdir of CCP4>
```

For example:

```
cmtzlib_f.c-10Aug2004.diff lib/src
```

Important: *no leading spaces or other characters are permitted on the line before the patch file name, and the name of the file and the subdirectory must be separated by a single space.*

The index.patches file in the CVS checked-out \$CETC directory contains comments that are intended to clarify the usage and gotchas involved in this process.

Note that using the make_patch.sh script should address many of these issues automatically. If the script is actually run in the appropriate patches directory then there should be no user intervention required.

- **Patched files**

Patched files (mostly patched executables and updated ccp4i files) should be placed in the "patched_files" directory of the current release, i.e.:

```
/public2/ftp/pub/current/patched_files
```

This contains an "index.files" file analogous to the "index.patches" file in the patches directory; see the notes on index.patches for information on updating this file appropriately. This file isn't yet used for anything but may be used in the future.

3.4.5 Naming conventions for patch and patched files

There is no official naming scheme - the only requirement for the automated patching mechanism that the patch files end with a .diff extension. A naming scheme such as that used by the make_patch.ch script described in section 3.4.3 is recommended for consistency only.

3.4.6 Making binary patches

At present there is no standard mechanism for making updates to the precompiled binaries. When needed these fixes are currently supplied as completely new binary executables.

3.4.7 Other resources

The CCP4 Release Handbook (draft) has some sections on the FTP server that are complementary to the notes in this section.

3.5 Alerting users through the Update Alert Mechanism

From CCP4 6.1, users will have a script `$CCP4/etc/ccp4_updates.py` which they can run to find out about available updates (exactly when and how they run this script is not yet mandated). This script downloads a list of available updates from http://www.ccp4.ac.uk/problems/ccp4_update_DB.txt and compares it to a local copy, saved from a previous query, and alerts the user to new or outstanding events.

The list of available updates on the web site (`ccp4_update_DB.txt`) therefore needs to be kept up-to-date. The format of this file is described at the top of the file, but briefly there is one line per update, and each line has a number of fields describing aspects of the update. Here "update" is a very broad term relating to anything the user might like to know about, e.g. patch for a bug, new executable available, new version available, etc. The second field describes the kind of update.

The first field of the updates file includes a tag which should correspond to the html tag used on the Problems Page. For example, if the problem is described at:

http://www.ccp4.ac.uk/problems.php#6.1-matthews_MOLWT-21Apr08

then the first field should have the tag

`XX_6.1-matthews_MOLWT-21Apr08`

where XX can be anything, and can be used for additional classification. If there is no entry on the Problems Pages, then the user will be given a broken link when `ccp4_updates.py` is run, but nothing worse. The fourth field may contain the location of a patch file, an updated file or another file such as an executable. A patch file (identified by PATCH in the second field) is assumed to be at

`ftp://ftp.ccp4.ac.uk/<fourth_field>`

Adding and updating programs

3.6 *Updating programs*

3.6.1 Overview of the procedure for updating programs

This section outlines a procedure to follow when adding updates to existing programs. Essentially the procedure is:

- Update the code:
 - Apply patch to an up-to-date working copy of the file(s) in question, or
 - Replace working copy of files with updated versions
- Check that the update doesn't overwrite any fixes made previously in CVS
- Check that the updated code builds and runs
- For significant updates add the details to CHANGES

Each of these steps is covered in more detail below.

Remember to add details of significant changes to the \$CCP4/CHANGES file. These form the release notes for the next release.

3.6.2 Updating the code

Program updates may be received as patches or as completely new files. In either case it is a good idea to check which versions of the files were used as the base for the changes – for example by looking for the CVS “\$Date: ...\$” or “\$Id: ...” tags.

If the changes are against an older version of the code then it is a good idea to revert the working copy of the code to that older version, apply the patch (or overwrite the file) and then use the CVS update mechanism to apply the changes from CVS and bring the working copy back up-to-date.

For patches, it is simplest to use the patch program to apply the changes.

For whole file updates it is wise to ensure that you have converted any DOS line-ends to Unix format (for example using a tool like `to_unix`) before updating the CVS repository.

3.6.3 Tagging new versions of programs in CVS

CVS tag old version of “large” programs, commit the raw updated files, then retag. For example:

- Update of Refmac is received from Garib

In the \$CPROG/refmac5_ directory tag the files as e.g. "refmac_dl_05_03_10" using CVS. Then overwrite with the latest files from Garib, CVS commit and tag as e.g. "refmac-5_2_0009". Then compile, perform tests and so on.

The advantages to using this approach are:

- It is possible to see what the changes were between the last version at DL and the latest version from the program author, and therefore to check that any fixes provided at DL have been incorporated into the official version
- It is easy to produce differences against a particular version of the program to send back to the author if necessary

Use CVS to check that any fixes made at DL have been incorporated into the author's official version.

3.6.4 Testing updates

Program updates should be tested, for example: programs should compile successfully and run using standard examples, changes to CCP4i shouldn't crash the interface on start-up, and so on.

For programs in \$CPROG a basic testing procedure might consist of running the available example scripts using the last stable version of the program, and then comparing the output when using the new version with the same script. Comparison of the log files can be done by eyeballing the output from diff; there are tools such as mtzdiff, pdbdiff and mapdiff to help compare CCP4 format output.

After CVSing changes to compiled programs it is also worth checking the autobuilds to see if there are any problems arising from the update of the program (see section 5.3).

3.7 Adding New Programs To The Suite

Assume that we have a procedure for obtaining legal authority to release contributed software. There is likely to be a proforma or other document that the contributing author will need to complete.

1. Add the appropriate licence banner to source code files where appropriate. Licence banners can be obtained from the licence resource page: http://www.ccp4.ac.uk/group/licence/licence_index.html
2. Add CVS keywords
3. Add to the build system.
4. Add html documentation. Note that there are special tags that must be inserted into documentation in order for the program indices to be correctly updated, and for references to be extracted. See the information at http://www.ccp4.ac.uk/dev/html_guide.html

5. Add runnable examples.
6. Update the \$CCP4/CHANGES file.

3.8 Special Procedures

This section documents some special procedures for making updates to particular programs or libraries.

3.8.1 Generate Refmac5 harvesting template for RCSB

This isn't normally requested these days but the information is kept here for historical interest. The aim is to generate a file that contains the CIF tokens that Refmac might output, so that the RCSB can determine whether a new version of the program has any new or modified tokens.

On UNIX/Linux:

```
% grep REMARK make_dummy.f > harvest_template.txt
```

3.8.2 Updating mmCIF dictionary used by CCIF libraries

The following notes describe how to make updates to the CCP4-distributed mmCIF dictionary:

1. Any application which reads or writes mmCIF files using Peter Keller's libccif library has the full file in memory. Thus all the mmCIF tags in the file must be recognised by libccif, which means they must be in the CCP4-distributed mmCIF dictionary.
2. Add the dictionary entry to the file \$CLIBD/cif_mm.dic, and CVS it. Just add a text block at the end.
3. The minimal entry is something like:

```
save _refln.pdbx_HL_C_iso
  _item.description.description
;      The isomorphous Hendrickson-Lattman coefficient
      C~iso~for this reflection.
;
  _item.name                '_refln.pdbx_HL_C_iso'
  _item.category_id         refln
  _item.mandatory_code      no
  _item.type.code           float
  save_
```

4. The mmCIF category is the bit before the period. That should have its own save statement. If not, add as:

```
save_refln
  _category.description
;      Data items in the REFLN category record details about
```

```

        the reflection data used to determine the ATOM_SITE
        data items.
;
  _category.id                refln
  _category.mandatory_code    no
  loop_
    _category_key.name        '_refln.index_h'
                              '_refln.index_k'
                              '_refln.index_l'

  loop_
    _category_group.id        'inclusive_group'
                              'refln_group'
  save_

```

Be careful with the number and placement of underscores.

5. Generate the binary mmCIF dictionary by running "make install" in the ccif directory.

4 Software development

Strictly speaking, software development is beyond the scope of this handbook. However this section outlines some resources that might be useful.

4.1 Style Guides

Adherence to programming style guides are recommended when developing software as it results in code that is easier to read and understand – both good qualities for maintainable software. It is suggested that the details of the style guide matter less than adhering to whatever style guide is chosen for the project.

For the majority of CCP4 software there is no official style guide (one exception is CCP4i, for which a style guide is being developed – see http://www.ccp4.ac.uk/peter/ccp4/ccp4i_style_guide.html). Furthermore there are clearly a variety of different styles within the suite (and even within some programs).

However there are some general principles:

- **Adopt a style which is consistent with the existing code**
For example, using the same depth of indents, or adopting lowercase Fortran commands if that is what the program already uses. This will make it easier for programmers who may have to understand the code in future, as they will only need to spend time adjusting to one style of programming for the whole program.
- **Break lines longer than 80 characters**
Use the appropriate continuation characters to break long lines. This is helpful to other programmers who might view code or code differences in an 80 character terminal window, as the default wrapping for longer lines on these screens makes it harder to understand.

Aside from this it has been suggested that the exact details of a style guide are less important than the fact that once you have chosen one, you should stick to it faithfully.

4.2 Programming Practice

Some general guidelines for good programming practice include:

- Avoiding or minimising the use of global variables as much as possible (especially in libraries)
- Maintaining backwards compatibility, for example by changing interfaces or functionality – it is better to provide new APIs to new functionality and preserve the old API (even if it simply wraps the new functionality).
- Avoiding duplication (also known as the DRY principle, short for “don’t repeat yourself”). For example, reuse existing code where possible, and if the

same code is used in different parts of a program or script, then consider moving that a single function.

4.3 Documentation

Traditionally individual CCP4 programs have been documented using HTML formatted pages with a style based on the UNIX man pages, and this remains the preferred way of documenting the compiled program. Task interfaces within CCP4i are documented within the CCP4i help tree, usually by module.

There is a page on “How to Write CCP4 Documentation” at http://www.ccp4.ac.uk/dev/html_guide.html, which covers the background and important technical details involved in writing HTML documentation for individual programs – this includes information on the tags to include for references and for inclusion in the program index.

(Unfortunately there is currently no similar documentation for CCP4i.)

4.4 Other resources

The CCP4 Developers web area at <http://www.ccp4.ac.uk/dev> has a number of different resources that may be useful when developing software within CCP4.

4.4.1 Template Programs

The template programs can be found via <http://www.ccp4.ac.uk/dev/templates/templates.html>. These are example programs which illustrate how to use a number of the CCP4 Fortran libraries, for example to read and write MTZ programs. These programs can be downloaded, compiled and should run.

4.4.2 Porting Guide

The “Developer’s Guide to Upgrading Programs to CCP4 5.0 Libraries” can be found at http://www.ccp4.ac.uk/dev/changes_for_developers_5.0.html. Although principally aimed at programmers using the CCP4 4.2 software libraries who wished to upgrade to version 5.0, this has some useful background information.

4.4.3 CCP4i Developers Area

The CCP4i Developers Resource Page is at <http://www.ccp4.ac.uk/ccp4i/developers.html>. As well as links to the CCP4i programmer’s documentation, it also has information on how to package up task interfaces for redistribution, and short and extended tutorial materials for people wishing to write new interfaces.

4.4.4 The CCP4 Manual

The infamous “blue book” is perhaps now rather dated but continues to be a surprisingly useful reference manual for many different aspects of CCP4. Sections II.3 and IV are particularly valuable.

5 Tools and resources

This section outlines the main tools and resources that are available to CCP4 programmers maintaining the suite. These are:

- The CVS patch branch (section 5.1)
- Bugzilla (section 5.2)
- Autobuilds (section 5.3)

There are a number of other general programming tools that might also be useful, and these are covered briefly in section 5.4.

5.1 *The CCP4 CVS Patch Branch*

5.1.1 Preliminaries: about CVS Concurrent Version System

CVS is a system that allows multiple developers to work simultaneously on the same set of files. Without such a system it is likely that the CCP4 project as it exists today would not be possible.

If you are not already familiar with CVS then before reading the rest of this section it would be useful to read generally about CVS. The following links are recommended for background on CVS:

- The CCP4 developer's area has a guide to version control with CVS at <http://www.ccp4.ac.uk/dev/cvs.html>
- A slightly more comprehensive (but still manageable) “quick and dirty” guide is at <http://www.cs.umb.edu/~srevelak.cvs.html>

The rest of this section focuses on one aspect of CVS, which is the ability to manage “branching” of the code base. However familiarity with the key CVS commands (check out, tagging, differencing and so on) are also required.

5.1.2 Introduction: what is the Patch Branch?

The patch branch is a branch in the CVS tree which is used to make patch releases of the CCP4 suite after the first patch release of any major (“point zero”) or minor (“point something”) release.

Patch branches associated with a release x.y.z are tagged within CVS using the tag name “release-x_y_z_patch”, for example for release 5.0.1 the patch branch had the tag name “release-5_0_1_patch”.

5.1.3 Which code fixes should be included?

There is a simple rule: any code fixes that are advertised on the problems page should also be added to the patch branch.

5.1.4 Procedure for adding fixes to the patch branch

This is a description of the procedure for updating the patch branch manually. It assumes that you already have a checked-out version of the suite with an up-to-date version of the file to be patched.

The procedure for adding a code change to the patch branch is as follows.

1. Switch the file to patched over to the patch branch version:

```
% cvs update -r <patch_tag> <file>
```

2. Apply the patch – this can be done manually or by previously preparing a patch from the main branch using the diff -c option of CVS.

3. Commit the change to the patch branch:

```
% cvs commit -m "message" <file>
```

4. Switch the file back to the main trunk:

```
> cvs update -A <file>
```

5.1.5 Making patch files from the patch branch

There is more information on how to do this in section 3.4.

5.1.6 Merging the patch branch back into the main branch

The first piece of advice is: consider if you can avoid doing this at – while adding changes to both the main and patch branches at the time of making the fix might feel like an annoying duplication of effort, it might pay off later because merging complicated changes (that the authors have forgotten about) shouldn't be necessary.

However if you do decide you need to do it then here's the strategy that was employed for merging the changes on the release-6_0_patch branch back into the main CVS trunk:

1. Tag the files on the patch branch so that you know where the merge point is (this will make it possible to do multiple merges should that ever be required), e.g.:

```
cvs rtag -r release-6_0_patch merge-6_0_patch_date ccp4
```

2. CVS checkout a new working copy of the suite from the main branch, then merge the changes from the branch into the working copy by doing:

```
cvs update -j release-6_0_patch
```

“-j” stands for join. There is a special flag that you can specify that tells CVS to ignore changes where the only differences are in CVS keywords (things like \$Id ...\$ and \$Date ...\$) – I think that this is “-kk”. If you forget to use this then most likely you will get a lot of conflicts on merge that are actually only different keyword substitutions, and this will save you some work.

3. It is very likely that you will get a number of conflicts on merge – this is where CVS is unable to figure out how to apply the changes to the trunk because the relevant code has changed too much for patch to locate a context.

So your first job is to identify and resolve the conflicts – unfortunately this has to be done by examining each file, the conflicts and the change log, to figure out which changes to keep and which to throw away. Your aim is to get all the conflicts resolved so that you left with only “locally modified” or “up-to-date” files in your working copy.

Remember that a file that contains conflicts may also contain other changes that merged without any problems. This means that you can’t necessarily just throw away the conflicted file and take the last change in the main trunk (though sometimes this is the right thing to do) – you might be throwing away real changes at the same time.

4. Once you’ve resolved the conflicts, you need to examine any remaining files that have local modifications, to verify that the changes really are worth keeping. This might mean more references to the change logs in CVS to figure out whether the changes should kept. If yes then commit them, if not then remove them.

***Addenda:** I found it useful to keep track of the status of the files at each stage in an Excel spreadsheet. I also developed a Python script to help examine CVS in a more sophisticated manner than the raw CVS commands seem to allow – I will add this to the suite once I have found a suitable place to put it – PJB 24th September 2007*

5.2 Bugzilla

5.2.1 Introduction: what is Bugzilla?

Bugzilla is the issue tracker that was developed during the Mozilla project, for logging and tracking of bugs and other software issues. It is one of a number of available issue trackers and is the one used within the Daresbury CCP4 group for logging issues and their outcomes.

Bug tracking has a number of advantages, for example:

- It allows others to see how problems were solved in the past and provides a knowledge base for future reference.
- It allows auditing of outstanding bugs during the release process.

This section outlines how Bugzilla should be used within CCP4.

5.2.2 Information that should be recorded in Bugzilla

The exact information entered into Bugzilla at each stage of the bug's life cycle is left to the discretion of the person dealing with the query. However the minimum information is:

Life-cycle stage	Information and actions
Logging a query in the system	• Include a copy of the original message plus the name and email address of the person sending the query, and any attachments that were included with the original query. • If the "bug" is actually a request for additional functionality or other feature request then it should be flagged as an <i>enhancement</i>. • Set the fields for platform, operating system, package version and component as accurately as possible.
Whilst dealing with a query	• You should always assign a bug to yourself and then accept it (changing its status from NEW to ASSIGNED) when you start working on it • The information entered into the Bugzilla system in the course of resolving the query is at the discretion of the programmer dealing with the query (my preference is to add a copy of each email sent to and received from the originator of the query).

	The full file name (i.e. including the path) and revision number of any changes made in CVS in the course of resolving the query.
Resolving a query	A summary comment briefly explaining the resolution

The “summary” (title) of the defect logged in Bugzilla should also be sensible. Since each of us has our definition of what this means in practice, these can only be guidelines:

- Don't duplicate information that is recorded elsewhere, such as platform or operating system (unless this adds additional useful detail like flavour of Linux)
- Try to keep the length of the summary text within the length of the entry box

Bugzilla will assign an id number to each new bug that is registered. It is convenient to use this number for example in the subject line when replying to the person who originally submitted the bug.

5.2.3 Definitions of RESOLUTIONs (and when to use them)

The table below outlines the different resolutions in Bugzilla, and suggested guidelines for what these mean in the context of the CCP4 project, and what actions might follow:

Resolution	Meaning	Additional actions
INVALID	The report wasn't a real bug (e.g. user error)	None.
FIXED	- There were changes to source code, documentation, installation scripts etc that mean that the buggy behaviour will not occur in future as a result, or - Other resources (web pages, files on FTP server) have been updated so that the bug will not occur again.	If relevant the fix should be reported on the problems page for the current release and added to the patch branch.
	The report was an enhancement that has now been added.	Consider adding to the CHANGES file
WONTFIX	The report is a bug but it will never be fixed.	Consider reporting the bug on one of the general problems pages.
LATER	The report is a bug and should be fixed at some unspecified later date.	Try to avoid using this.

Other resolutions should be avoided: for example, WORKSFORME is ambiguous (it is taken to mean that a putative fix has been implemented but not confirmed, whereas actually it means that the bug could not be reproduced by the programmer trying to fix it).

Do not flag bugs as CLOSED after resolving them. Closing bugs implies that the resolutions have passed a quality assurance procedure that CCP4 does not use.

5.3 Autobuilds

5.3.1 Introduction: what are the CCP4 Autobuilds?

The autobuild scripts are a set of shell scripts that checkout the CCP4 suite from CVS on various machines, configure and build it, run the available tests, and report the results in a set of log files. There are also scripts for prettifying these logs and making them available on the web.

The scripts are run regularly (most of them each night) by cron jobs. The logs can be checked to look for obvious problems in compilation of the suite and running the standard examples. It is useful to check the build logs the day after making a program or library update, to see if the fix causes any problems on platforms besides the one the update was made on.

The build logs can be accessed at the URL:

<http://www.ccp4.ac.uk/group/auto-test/summary.html>.

5.3.2 Autobuild CVS archive and documentation

The autobuild scripts plus a number of tests are held in a CVS repository called `auto-build` in `/ccpdisk/xtal/CVSR00T`. To obtain copies of the scripts, tests and documentation do e.g.

```
cvs -d /ccpdisk/xtal/CVSR00T co auto-build
```

This should create a working copy of the `auto-build` module, with the scripts and documentation in the `bin` and `html` subdirectories respectively.

The documentation outlines the technical details of how the scripts work together to form the build system, how the logs get published to the web, and how to make changes to the system (including adding new build platforms and new components of CCP4) – so this is not covered in this handbook. Instead the specific details of how the build system is actually deployed and how to perform day-to-day maintenance are covered in the following sections.

5.3.3 Deployment of the autobuilds at DL

Autobuilds are set to run as cron jobs on a number of machines at DL. At the time of writing (2nd September 2008) these machines are:

- `dlpx1`
- `ccp4e`
- `ccp4h`
- `ccp4t`

- `ccp4serv3`

Typically each cron job runs a build by executing the `cron-test-ccp4.csh` script, with additional options that allow testing of the `--shared-lib` and Intel compiler build options.

In addition cron jobs run regularly on `dlpx1` to copy the logs generated by the builds on these machines to the web server, by periodically executing the `copy-autotest-logs.tcl` script.

In the past these cron jobs have been run by user “`pjx`” (i.e. Peter Briggs); however I am now attempting to migrate these to the “`ccp4null`” user.

5.3.4 Maintaining the autobuilds

The autobuilds must be monitored as sometimes maintenance activities are required. It is a good idea to check the build logs on a regular basis to make sure that the builds are running on the days and times that are expected.

If logs for particular builds become seriously out of date then this may indicate that there is a problem with that build (see the section on “Common problems”). Note that in this case a problem refers to the running of the build scripts, not a problem with the configuration, building or running of the CCP4 suite.

Other maintenance activities involve periodically logging into the account that runs the builds and clearing out the emails that cron sends to that account; these emails contain the standard output from each autobuild run as a cron job, and can provide useful diagnostics in the event that the build has problems. Otherwise generally these logs are verbose and uninformative, and should be deleted.

It may also be necessary to change the days or times that autobuilds run. This is a simple case of logging into the account that runs the builds and updating the entries in the crontab file appropriately.

Finally, it is usually the case that the builds are setup to use the autobuild scripts in `/ccpdisk/xtal/auto-build/bin`. If this is not the case and instead a machine uses local copies, then these local copies will need to be kept up to date with any changes made to the CVS repository.

5.3.5 Hints for deploying an autobuild on a new machine

The autobuild documentation gives details of how to update the scripts to add a new build machine to the system. Once the scripts have been updated, deployment is a case of setting up cron jobs to run the `cron-test-ccp4.csh` script at the required dates and times.

(It is assumed that the machine mounts `ccpdisk`, in order to be able to access the CVS repository at `/ccpdisk/xtal/CVSROOT` and also to be able to write autobuild log files to the `/ccpdisk/xtal/auto-build` area for collection and publication on the web area.)

An example crontab line running the most basic autobuild might look like:

```
30 19 * * 0-5 /ccpdisk/xtal/auto-build/bin/cron-test-ccp4.csh
```

which would execute the default autobuild for this platform at 7:30pm each night Sunday to Friday. (See the crontab man page for an explanation of the crontab format.)

In addition we might want to also run a shared autobuild on Saturdays on this machine, in which case we might add another line:

```
30 19 * * 6 /ccpdisk/xtal/auto-build/bin/cron-test-ccp4.csh -shared
```

Since at the weekends we can assume that the machine is not in use by anyone else, maybe we will also add a test using the Intel compilers (assuming that they are installed and configured):

```
30 7 * * 6 /ccpdisk/xtal/auto-build/bin/cron-test-ccp4.csh -shared
30 7 * * 6 /ccpdisk/xtal/auto-build/bin/cron-test-ccp4.csh -intel \
-shared
```

and so on.

In practice for most machines I set them up to build on the local disk and only write the logs to the /ccpdisk/xtal/auto-build area, by using the -log_dir and -build_dir options of cron-test-ccp4.csh. For example:

```
30 19 * * 0-5 /ccpdisk/xtal/auto-build/bin/cron-test-ccp4.csh \
-build_dir /home/pjx/AUTO-BUILD -log_dir /ccpdisk/xtal/auto-build
```

Notes:

- Both the specified directories must exist before the script is run, otherwise the test will fail to run;
- The build and log areas are created as subdirectories of those specified by the -log_dir and -build_dir arguments;
- The logs must be written to /ccpdisk/xtal/auto-build in order for them to be published on the web area.

Finally I tend to try and slightly stagger the start times for builds across the different machines, and to ensure that only one build is scheduled to run on each machine at any one time. (The reason for staggering the builds is to try and avoid several builds competing for read access to CVS at the same time, which still works but slows them all down.)

5.3.6 Setting up autobuilds for patch branches

After a release it is useful to run builds for the patch branch as well as the main CVS trunk. This can be done very simply by setting up new cron jobs (or editing existing

ones) to add the `-tag` option to the `cron-test-ccp4.csh` invocation, and using this to specify the CVS tag corresponding to the branch in question. For example:

```
30 19 * * 0-5 /ccpdisk/xtal/auto-build/bin/cron-test-ccp4.csh \
-build_dir /home/pjx/AUTO-BUILD -log_dir /ccpdisk/xtal/auto-build \
-tag release-6_0_patch
```

In this case the build and log areas will actually be created within a subdirectory called `release-6_0_patch` (or whatever CVS tag is actually used), to avoid clashes with builds and logs made using the CVS trunk.

In order to publish the logs it is also necessary to run a variant of the `copy-autotest-logs.tcl` script on `dlpx1`, again specifying the CVS tag. For example:

```
30 19 * * 0-5 /ccpdisk/xtal/auto-build/bin/copy-autotest-logs.tcl \
-tag release-6_0_patch
```

This will copy the logs to a subdirectory of the web area called `release-6_0_patch`.

Note: as before all the specified directories must exist before the script is run, otherwise the test will fail to run (the same applies for the web area).

5.3.7 Common problems with the Autobuilds (and how to fix them)

The most common problems encountered so far are:

- **Builds fail to run because of left over lock files**
In this case the solution is to manually remove the lock file `LOCK.machine` for the particular machine in question.
- **Builds leave lock files in the CVS repository that block other autobuilds and users from checking out the suite**
The solution is to identify where in the repository the lockfiles are, and then manually delete them – however one should always exercise extreme caution when deleting files from the CVS repository, as under UNIX there is no way to get them back unless you make a copy first.
- **Builds fail due to lack of disk space**
The only options here are to either change where the build runs to somewhere with sufficient disk space, or stop running the build.

5.3.8 Limitations

The autobuild scripts have a number of limitations:

- The runnable scripts test only a small fraction of the parameter space of the suite, so the tests are quite cursory
- The autobuilds don't test any components of CCP4i
- The requirement to mount `/ccpdisk` is cumbersome.
-

5.4 Exporting “onlylibs” distributions of CCP4

In the past nightly exports of the “library” components have been exported from CVS and made available via the ftp site for developers to download and install even if they did not want the full suite. This service has now been withdrawn due to lack of demand.

The exporter script **export-onlylibs.csh** can still be used to make one-off updates of the library distributions. It is in the autobuild CVS archive – when it is run it will automatically tag, export, tar and gzip the onlylibs part of the core CCP4 archive and will then move it to the ftp area (hardcoded into the script) using **scp**. It can also be used to export against a pre-existing CVS tag in the archive.

See the script for more details of usage.

5.5 Other tools and resources

The remaining tools are mentioned here for completeness and are not covered in any significant detail. References to external resources are provided where appropriate.

5.5.1 Valgrind

Valgrind is a suite of tools for debugging and profiling Linux programs; most usefully for CCP4 it was used to detect memory management errors during the upgrade to the 5.0 C-based libraries. See <http://valgrind.org> for more information.

5.5.2 mtzdiff, pdbdiff and mapdiff

These are differencing utilities that can be used to do a quick comparison of the key CCP4-formatted file formats, and are part of the main CCP4 release.

5.5.3 Test suite and test data

As part of the initial development of CCP4 6.0 a CVS module called “ccp4-tests” was created to hold miscellaneous test data and examples for various programs. This was intended to supplement the data and examples distributed to all users with the suite, and which was kept separate for reasons of size (for example, image files for use with the MOSFLM test scripts) or because CCP4 does not have permission to redistribute the data.

There are also a number of files with particular properties, for example maps consisting of a single section for testing mapslicer or MTZ with usual or problematic spacegroups.

5.5.4 Code checkers and debuggers

These are not widely used within CCP4 but may be worth looking at in some cases:

- ftnchk (a static code analyser for Fortran 77 programs)
<http://www.dsm.fordham.edu/~ftnchek/>
- TclPro (a code checker and graphical debugger for Tcl)
<http://www.tcl.tk/software/tclpro>
- PyChecker (a python source code checking tool)
<http://pychecker.sourceforge.net/>
- GDB (The GNU Project Debugger)
<http://www.gnu.org/software/gdb/gdb.html>

6 Book List

This is a rather random list of books that I have found either interesting or useful (and sometimes even both), which have informed the content of this handbook.

- **“Software Project Survival Guide”** *Steve McConnell*: an overview of how to run software development projects.
- **“The Pragmatic Programmer”** *Andrew Hunt and David Thomas*: an inspiring overview of the philosophy of good software management and development.
- **“The Practice of Programming”** *Brian W. Kernighan and Rob Pike*: “an excellent treatise on writing high-quality programs surely destined to become a classic of the field.”³
- **“The Art of UNIX Programming”** *Eric S. Raymond*: a wide-ranging overview of software development within the Unix tradition; it mixes history, Unix philosophy and practical information and makes for fascinating reading.
- **“Effective Tcl/Tk Programming”** *Mark Harrison and Michael McLennan*: excellent book on using Tcl/Tk to develop applications, with lots of useful good-quality examples. It is clear that the development of CCP4i was heavily influenced by the concepts in Harrison and McLennan’s book.
- **“About Face 2.0: The Essentials of Interaction Design”** *Alan Cooper and Robert Reimann*: “despite some occasional quirks and crotchets, this book is a trenchant and brilliant analysis of what’s wrong with software interface designs, and how to put it right.”⁴
- **“Essential CVS”** *Jennifer Vesperman*: good overview of usage of CVS.
- **“Software Maintenance: Concepts and Practice”** *Penny Grubb and Armstrong A Takang*

³ Raymond, “The Art of Unix Programming” p487

⁴ *Ibid*, p485