Mars Configuration Management Plan

1. Overview

Mars is an infrastructure library for simulation development. Its primary clients are the JNEM and Minerva projects. Mars uses Subversion for version control, and Bugzilla for package tracking; this document explains how this works in practice.

2. Mars Version Numbering

Versions of the Mars software are numbered *x.y* where *x* is the major version number and *y* is the build number. The value of *y* increments with each build; the value of *x* increments only with significant infrastructure upgrades, i.e., when moving from one version of Tcl/Tk to the next.

Note that Mars is not released on its own, but only as part of a client project.

3. Project Tree

The project directory tree is as follows

`<root>/`	Root of project tree
`bin/`	Executables (both binaries and scripts)[1]
`docs/`	Project documentation
`dev/`	Miscellaneous development documentation
`man1/`	Man page[2] section (1): Executables
`man5/`	Man page section (5): File Formats
`mann/`	Man page section (n): Tcl Commands[3]
`mani/`	Man page section (i): Tcl Interfaces
`lib/`	Tcl libraries
`<name>/`	Tcl source code for Tcl package <name>[4]
`app_<name>/`	Tcl source code for mars(1) application mars_*name*(1).
`src/`	C/C++ source directory – includes toplevel Makefile
`include/`	C/C++ library header files
`lib/`	C/C++ library files
`lib<name>/`	C/C++ source code for library lib<name>.a
`<name>/`	C/C++ source code for executable <name>

[1] All Mars executables are development tools; none are delivered with client projects.
[2] Mars manual pages are written in mars_man(5) format, an extended HTML format translated into HTML for display by mars_man(1).
[3] Theoretically, manual section (n) is for "new" commands, which properly belong in some other section and will be moved there eventually. Practically, section (n) hasn't gotten much use—except that man pages for Tcl commands have always gone in section (n). We're just following the established Tcl convention.
[4] The structure of Tcl package directories is described in the Mars Tcl Coding Standard.

4. Version Control

JNEM uses Subversion more or less as described in *Pragmatic Version Control using Subversion*.

4.1 Subversion Repository

The Mars Subversion repository URL is

```
https://oak.jpl.nasa.gov/svn
```

For developers at JPL, this may be abbreviated `https://oak/svn`.

In the remainder of this document, the string *repository* will mean the repository's base URL in either of the above forms.

The repository contains multiple projects; the project URLs for Mars and JNEM are

repository`/mars/`
repository`/jnem/`

The project has the following internal structure:

repository`/mars/trunk/`
repository`/mars/branches/`
repository`/mars/tags/`

The `trunk` directory contains the project's files on the main line of development. Subdirectories of the `branches` directory represent development branches. Subdirectories of the `tags` directory represent snapshots of development, e.g., the 1.3 build.

4.2 Subversion Branches and Tags

This section explains how Mars uses Subversion branches and tags to facilitate development of Mars and its clients. Procedures appear in Section 7.

Branch/Tag	Example	Description
`trunk/`	`trunk`	All current development
`tags/mars_`*x.y*	`tags/mars_1.3`	Snapshot of Mars build 1.3
`branches/`*user*`/`*yyyymmdd_name*	`branches/will/20081002_sample`	Experimental branch
`branches/bug/`*nnnn*	`branches/bug/1776`	Bug fix branch

Build Tags: At each official build during normal development, the trunk is tagged for the build as "`tags/mars_`*x.y* ". This serves as a snapshot of the build, and allows it to be recovered at any time.

Experimental Branches: Developers can create branches on which to do experimental coding; the results can later be merged with the trunk or abandoned. Experimental branches are created with branch name "`branches/`*user*/*yyyymmdd_name*", where *user* is the developer's user name, *yyyymmdd* is the current date, and *name* is some brief name for the experiment. Experimental branches can then be checked out just like the trunk.

Bug Fix Branches: When fixing complicated bugs over a period of time, it is sometimes useful to do the fix on a separate branch. This allows multiple commits without any chance of destabilizing the trunk. Such branches should be called "`branches/bug/`*nnnn*", where *nnnn* is the bug number. The completed fix can be merged back to the trunk just as with an experimental branch.

Release Branches: Mars is not released on its own, and so release branches are not required. When Mars is released as part of a client project, the desired build is copied into the client's release branch and maintained there. See Section 8 for information on how Mars is released with client projects.

5. Package Tracking

Both bugs and new requirements will be tracked using Bugzilla. The JNEM Bugzilla Server resides at

https://oak.jpl.nasa.gov/bugzilla

This may be abbreviated `https://oak/bugzilla` or even `https://oak`.

5.1 Product

Bugzilla relates bugs to a specific Product, Version, and Component. The product for Mars is "Mars 1.0".

5.2 Version

The "Version" field indicates the build of Mars, "*x.y*", in which the bug was found.

5.3 Component

The component will be one of:

- CM Process
- Code
- Data
- Documentation

5.4 Status Whiteboard

By convention, the Status Whiteboard field is used to record the build in which the bug was fixed or otherwise resolved. The following text is used:

`Fixed in x.y`

5.5 Severity

Every bug has a severity; Bugzilla defines the following categories:

Blocker	Blocks development and/or testing work
Critical	Crashes, loss of data, severe memory leak
Major	Major loss of function
Minor	Minor loss of function, or other problem where easy workaround is present
Trivial	Cosmetic problem like misspelled words or misaligned text
Enhancement	Request for enhancement

New features are tracked as “Enhancement” bugs.

5.6 Life Cycle

The “Status” and “Resolution” Fields track a bug through its life cycle, as shown in the table on the following pages, which was pulled from the Bugzilla documentation.

5.7 Report Contents

Bug reports should include enough specific information to allow the bug to be reproduced. When the bug is fixed, the comment should include the precise test needed to verify the fix.

STATUS	RESOLUTION
The **status** field indicates the general health of a bug. Only certain status transitions are allowed.	The **resolution** field indicates what happened to this bug.
UNCONFIRMED: This bug has recently been added to the database. Nobody has validated that this bug is true. Users who have the "canconfirm" permission set may confirm this bug, changing its state to **NEW**. Or, it may be directly resolved and marked **RESOLVED**. **NEW:** This bug has recently been added to the assignee's list of bugs and must be processed. Bugs in this state may be accepted, and become **ASSIGNED**, passed on to someone else, and remain **NEW**, or resolved and marked **RESOLVED**. **ASSIGNED:** This bug is not yet resolved, but is assigned to the proper person. From here bugs can be given to another person and become **NEW**, or resolved and become **RESOLVED**. **REOPENED:** This bug was once resolved, but the resolution was deemed incorrect. For example, a **WORKSFORME** bug is **REOPENED** when more information shows up and the bug is now reproducible. From here bugs are either marked **ASSIGNED** or **RESOLVED**.	No resolution yet. All bugs which are in one of these "open" states have the resolution set to blank. All other bugs will be marked with one of the following resolutions.

STATUS	RESOLUTION
RESOLVED: A resolution has been taken, and it is awaiting verification by QA. From here bugs are either re-opened and become **REOPENED**, are marked **VERIFIED**, or are closed for good and marked **CLOSED**. **VERIFIED:** QA has looked at the bug and the resolution and agrees that the appropriate resolution has been taken. Bugs remain in this state until the product they were reported against actually ships, at which point they become **CLOSED**. **CLOSED:** The bug is considered dead, the resolution is correct. Any zombie bugs who choose to walk the earth again must do so by becoming **REOPENED**.	**FIXED:** A fix for this bug is checked into the tree and tested. **INVALID:** The problem described is not a bug. **WONTFIX:** The problem described is a bug which will never be fixed. **LATER:** The problem described is a bug which will not be fixed in this version of the product. **REMIND:** The problem described is a bug which will probably not be fixed in this version of the product, but might still be. **DUPLICATE:** The problem is a duplicate of an existing bug. Marking a bug duplicate requires the bug# of the duplicating bug and will at least put that bug number in the description field. **WORKSFORME:** All attempts at reproducing this bug were futile, and reading the code produces no clues as to why the described behavior would occur. If more information appears later, the bug can be reopened.

6. Development Documentation

Mars development documentation consists of:

- Software manual pages (man pages)
- Memos
- Other documents: CM plan, Mars Analyst's Guide, etc.

Software manual pages are stored in the repository project tree, as described in Section 3, and are written in mars_man(5) format.

Other documents may be written in HTML or in MS Word; note that Word documents should be saved in both Word and PDF formats. HTML documents will usually use the mars_doc(5) extended HTML format.

Memos will be saved on the JNEM web page:

```
http://eis.jpl.nasa.gov/jaws
```

Because memos are date-stamped, they need not be version-controlled in the repository.

Other documents, such as this CM plan, will be stored in the repository in the `mars/docs/dev` directory.

7. Procedures

We expect that Mars development will generally take place in a Mars work area embedded within a client workarea; see Section 8 for details. This section describes procedures for standalone Mars development.

7.1 Performing Official Builds

1.	Make sure all developers have committed all changes needed for the build.	
2.	Determine the version number, e.g., 1.3.	
3.	Check the trunk out of the repository.	`$ svn checkout` *repository*`/mars/trunk ~/mars`

4.	Update the Build Notes file, `docs/build_notes.ehtml`, with the details about the current build, and checks it into the repository.	`$ cd ~/mars/docs`
		`$ emacs build_notes.ehtml`
		`$ svn commit -m"Updated for build 1.3"`

5.	Build and test the software, resolving any problems found until the build is clean, and committing any changes.	`$ cd ~/mars` `$ make build`

6.	Tag the trunk with the build number.	`$ cd ~/mars`
		`$ svn cp .` *repository*`/mars/tags/mars_1.3`
		
		or
		
		`$ cd ~/mars`
		`$ make MARS_VERSION=1.3 tag`

7.	Check out the tagged version out of the repository as `~/mars_`*version*.	`$ cd ~` `$ svn checkout` *repository*`/mars/tags/mars_1.3`
8.	Perform the official build. Because Mars is typically built with each client that uses it, this "official" build simply verifies that the tagged version of Mars is viable.	`$ cd ~/mars_1.3` `$ make MARS_VERSION=1.3 build`
9.	If desired, the Mars documentation tree can be copied to some central location.	

7.2 Checking Out The Trunk

Mars is often checked out as an embedded work area within the client project. If it is to be checked out stand-alone, it can be checked out anywhere; this procedure assumes that it is checked out simply as "`~/mars`".

1.	Check out the trunk.	`$ svn checkout` *repository*`/mars/trunk ~/mars`
6.	Do a development build.	See Section 7.5.
8.	Go to work.	`$ cd ~/mars` `$ ...`

7.3 Checking Out A Tag

Tags are usually created only for particular builds, and have names like "`tags/mars_`*x.y*". Consequently, tags can usually be checked out without specifying a directory name. Don't commit changes to tags.

1.	Check out the tag.	`$ svn checkout` *repository*`/mars/tag/mars_1.3 ~/mars_1.3`
4.	Go to work.	`$ cd ~/mars_1.3` `$ ...`

7.4 Checking Out An Experimental Branch

Experimental branches are named for the developer, the date on which the branch was created, and the experiment itself, e.g., "`branches/will/20081010_widget`" was created on 10 October 2008 by "`will`" and has something to do with widgets. It's usually necessary to pick a directory name when checking out an experimental branch; a common strategy is to use the directory name "`mars_`*name*" where *name* is the experiment name, e.g., "`mars_widget`".

1.	Check out the branch.	`$ svn checkout` *repository*`/mars/branches/will/20081010_widget \` `  mars_widget`
2.	Go to work.	`$ cd ~/mars_widget` `$ ...`

7.5 Performing a Development Build

A development build is simply a build performed during the course of development. It has three purposes: it builds C/C++ libraries, it builds the documentation, and it runs all unit tests. This procedure presumes that the version to be built is the current version.

1.	Go to the work area.	`$ cd ~/mars`
2.	Build everything from scratch.	`$ make clean all`
3.	Run all tests, if desired.	`$ make test`
4.	The previous two steps can be combined into a single command.	`$ make build`
5.	Read the header of `~/mars/Makefile` for other possibilities.	

8. Client Projects

Mars is not delivered on its own, but is to be used as an infrastructure layer by client projects. This section explains how client projects are intended to fit Mars into their CM, using JNEM as an example.

8.1 Main-Line Development

During main-line development on the client's trunk, the client should be using the latest Mars code. This is accomplished by using an `svn:externals` property to cause Mars to be checked out as part of the client code base:

```
$ svn propset svn:externals repository/jnem/trunk "mars repository/mars/trunk"
```

When the client's trunk is checked out, e.g., as `jnem/`, Mars is checked out as `jnem/mars/`. The current Mars workarea is thus embedded in the client workarea and can be referenced, built, and tested in place.

It is possible to make code changes to the Mars trunk in this embedded work area; one must be careful to commit both the Mars changes and any client changes.[5]

8.2 Bug Reporting

When a change is made that affects both Mars and a client, two bug reports are needed: one for the Mars change and one for the client change. The Mars bug report provides a record of what's changed in Mars, and the client bug report goes through I&T with the client, thus allow both the client change and the Mars change to be tested in the test lab.

When the Mars change is a new API and the client change is use of that API, completely separate bug reports are probably appropriate. When the change is actually a bug fix that affects both projects, or perhaps even a bug in Mars that manifests in the client, cloning the original client's bug report is probably appropriate.

---

[5] *Pragmatic Version Control* suggests that this is a bad idea. However, given that we have a single development team, and that Mars development is tightly coupled with client development, we think that this is OK.

8.3 Client Development Builds

When a developer performs a development build of the client, the client's makefiles should be set up to build Mars as well. In particular, the client's "`make src`", "`make docs`", and "`make test`" targets should do a "`cd mars; make` *target*".

8.4 Client Official Builds: On the Client Trunk

When an official build is to be done for the client, the build manager should do the following:

1.	Determine whether any changes have been comitted to Mars since the previous official build of Mars. If there have been, do an official build of Mars, as described in Section 7.1. Either way, we'll assume that the latest build is `mars_1.6.`	
2.	Do a development build and test of the client. Resolve any errors, and retest, until the client builds cleanly. Commit any changes.	`$ cd ~/jnem` `$ make build`
3.	Check out the trunk or branch of the client for which the build will be done.	`$ cd ~/jnem_pkgs` `$ svn checkout` *repository*`/jnem/tags/jnem_4.0.9`

4.	Tag the build, and update the work area to the tagged version.	`$ cd ~/jnem_pkgs/jnem_4.0.9`
		`$ svn copy .` *repository*`/jnem/tags/jnem_4.0.9`
		`$ svn switch` *repository*`/jnem/tags/jnem_4.0.9 .`

5.	Link the tagged version to the desired version of Mars. This sets the `svn:externals` property, and updates the embedded version of Mars to the desired version.	`$ cd ~/jnem_pkgs/jnem_4.0.9` `$ mars import 1.6`
6.	Proceed with the client build. This version of the client is now permanently linked with the desired version of Mars.	

8.5 Client Branches

Client branches can be linked to any desired version of Mars via `svn:externals`, just as the client trunk is linked to the trunk of Mars. However, it is more likely that a given client branch, and particularly any release branch, should be tied to a particular version of Mars, and that any bug fixes to Mars should be applied within the context of the release branch as well. Consequently, when creating a client release branch the `svn:externals` property should be removed and `svn copy` should be used to pull the desired version of Mars into the release branch. The procedure is the same as that used when doing an official build on the client's trunk.

9. References

Bugzilla, http://www.bugzilla.org

Subversion, http://subversion.tigris.org

Pragmatic Version Control: Using Subversion, by Mike Mason, Pragmatic Bookshelf, 2006.