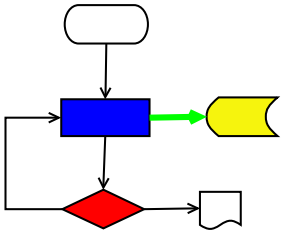
VICI



VISUAL CHART INTERPRETER

Programmer's Guide

Publication History

Date	Who	What Changes
4 October 2012	Brenton Ross	Initial version.
14 May 2014	Brenton Ross	Converted for the VICI project
16 January 2016	Brenton Ross	Updated infrastructure section.
28 February 2016	Brenton Ross	Added instructions for Eclipse include paths.
14 June 2016	Brenton Ross	Added checklist for end of increment.
20 August 2016	Brenton Ross	Added web site maintenance.
9 December 2016	Brenton Ross	Added staged build for make all, and support for config.h
12 February 2017	Brenton Ross	Added section for gth-editor.

Table of Contents

1 Introduction

This is the Programmer's Guide for the VICI project.

The aim is to provide some guidance for the developers so that there is some consistency in the code, etc. that is developed both over time, and between developers.

1.1 Scope

The document covers the configuration of the development environment, the coding standards, and the usage of some of the supporting libraries that are part of VICI.

1.2 Overview

The first sections are for the configuration of the development environment so that all developers can set up the Eclipse environment in a similar manner.

The second sections are for standards to be used when creating and updating code and other source files.

The final sections describe the usage of the VICI supporting libraries.

1.3 Audience

This document is intended to be used by the developers of the VICI system.

2 Configuring Eclipse

The VICI project uses Eclipse CDT and Autotools for developing and building the system. This section describes the configuration settings that are to be used.

2.1 The Workspace

The VICI project will have its own dedicated workspace under Eclipse.

The configuration for the workspace is accessed from the “Window | Preferences” menu. The following are the changes from the default values.

Under “C/C++ | Build | Environment” add an environment variable:

- VICI /home/brenton/dev/vici (modify to suit your build environment)

Under “C/C++ | Build | Settings | Discovery” add “ -std=c++11” to the “CDT GCC Builtin Compiler Settings” so that the Eclipse indexer will understand the latest version of C++ which we will be using for this project.

Under “C/C++ | General | Paths and Symbols” add __cplusplus with a value of 201409L as a G++ symbol.

Under “C/C++ | Code Style | Formatter” select the BSD style for code formatting. **This is the project standard.**

Under “C/C++ | Code Style | Code Templates | Comments | Files” add the following text for the pattern:

```
/*
 *             ${file_name}
 *
 *      Copyright 2012 - ${year} Brenton Ross
 *
 *      This file is part of VICI.
 *
 *   VICI is free software: you can redistribute it and/or modify
 *   it under the terms of the GNU General Public License as published by
 *   the Free Software Foundation, either version 3 of the License, or
 *   (at your option) any later version.
 *
 *   VICI is distributed in the hope that it will be useful,
 *   but WITHOUT ANY WARRANTY; without even the implied warranty of
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *   GNU General Public License for more details.
 *
 *   You should VICI received a copy of the GNU General Public License
 *   along with VICI.  If not, see <http://www.gnu.org/licenses/>.
 */
```

2.2 Subversion

The project will be using a Subversion repository for version control. Eclipse has good support built in for Subversion. It is only necessary to use Team | Share to add a project, and then add the source controlled files to Version Control. Be careful to only include versioned files when committing or it will

add every damn file it can find. The URL for Subversion on Source Forge for VICI is: svn+ssh://brenton-ross@svn.code.sf.net/p/ocratato-vici/code

2.3 Re-establishing an Eclipse Workspace

After setting up a new computer (or replacing the HDD) and installing the latest version of Eclipse there is some work to do to get it working with subversion. (Eclipse has its own subversion client and the latest is not compatible with the one used when creating the original VICI project.)

Start by renaming the workspace directory which will be our backup copy.

Create a new workspace directory, $VICI/src

Start Eclipse and select the new workspace.

Perform the actions for Workspace as described above.

For each project we will need to perform the following:

Select File|Import. Select Project from SVN.

Set the url (svn+ssh://brenton-ross@svn.code.sf.net/p/ocratato-vici/code) and browse to the trunk and then the particular project.

Set the private key (~/.ssh/id_dsa).

Check out as project with name, e.g. cfi

Convert to C/C++ project and allow it to build makefiles etc.

Create a build directory and set it as the build location.

Confirm that you can build the project.

Confirm the you can commit changes

2.4 The top Project

This is the master project for building the entire suite. Eclipse treats each program and library as a stand-alone project, each with its own configure script and makefiles. The top project collects these together so that a single distribution package can be created, or a full suite build can be performed.

From “File | New | C++ Project” select the GNU Autotools | Empty Project and name it “top”. Then select “Next” and “Advanced settings”.

Under “Autotools | Configure Settings” set the following Directory Specifiers:

- --prefix: ${VICI}

The configure command should also have “ --enable-testing” appended during development, and while developing each module it should be “--enable-testing=VICI_XXX” where “XXX” is your module.

The configure.ac file should include AC_CONFIG_SUBDIRS for each subdirectory and Makefile.am should also include just a SUBDIRS= directive that lists the other projects. You will need to manually add soft-links from the

top project to the other projects since autotools expects a strict directory hierarchy.

You should add a separate build directory under top, (and for each project) so that object files,and other temporary files do not litter the source directories.

The top project enables the "make all" target by modifying it to also do an install into a staging directory. The sub-projects need to include this staging directory in their search paths for libraries and include files.

The configure.ac file for the top project includes the following:

```
# staged build of subdirs
stagedir=`pwd`/stage
AC_SUBST(stagedir)

ac_configure_args="$ac_configure_args stagedir=$stagedir"
```

This declares the staging directory in the top project's build directory and passes this to the configuration step of the sub-projects so that they know where to install to.

The Makefile.am for the top project includes the following:

```
SUBDIRS=cfi gth persist interfaces config ebnf syntax admin search
SUBDIRS += symbol canvas editor interpreter vici

.PHONY: subdirs $(SUBDIRS)

all-recursive: subdirs

subdirs: $(SUBDIRS)

$(SUBDIRS):
    $(MAKE) -C $@
    $(MAKE) -C $@ DESTDIR=$(stagedir) install

distclean-local:
    rm -rf $(stagedir)
```

This sets up the build order for the sub-projects and redefines the all-recursive target to build and then install each sub-project into the staging directory.
A local target adds the removal of the staging directory during the distribution clean process.

2.5 Library Projects

Start by creating a new project: File | New | C++ Project and then selecting GNU Autotools | Autotools Shared Library Project and set the project name. Then select Next and set the details, especially the library name. Then select Next and Advanced Settings, under “Autotools | Configure Settings” set the

following Directory Specifiers:

- --prefix: ${VICI}

Under Advanced add the following for the compiler options:

CXXFLAGS="-g -O2"

Then select OK and Finish.

Eclipse will have generated C source, but we need C++, so the following changes need to be made:

- In configure.ac change the project parameter of AC_INIT to “vici” and AC_PROG_CC to AC_PROG_CXX.
- Rename the generated source files from .c to .cpp and adjust the names in the corresponding Makefile.am files.

Add -L$(VICI)/lib/vici and -I$(VICI)/include to the Makefile.am files so that our libraries can be referenced.

Add AM_CXXFLAGS = -std=c++11 to the Makefile.am files so that the correct and latest version of the C++ language is used.

Eclipse needs to be told to use the recent additions to the compiler, plus the location of our headers:

Add __cplusplus 201409L to Paths and Symbols | #Symbols

Add ${VICI}/include/ to Paths and Symbols | Includes

Use the pkg prefix for the lib and include prefixes in the Makefile.am files so that the headers and libraries are installed in the vici subdirectories.

To support “make all” from the top project the configure.ac file for each sub-project needs to include the following:

```
# Support for staged install during top level make all
VICI_INCLDS='-I.. -I$(top_srcdir)/include -I$(stagedir)/$(includedir)
-I$(includedir)'
VICI_LDPATHS='-L$(stagedir)/$(libdir)/vici -L$(libdir)/vici'
AC_SUBST(stagedir)
AC_SUBST(VICI_INCLDS)
AC_SUBST(VICI_LDPATHS)
```

This defines a couple of variables that specify the locations to look for includes and libraries. The stage directory will default to an empty string when building from the sub-project but will be defined when the top level build is being performed.

The Makefile.am files for the various libraries and binaries in the sub-projects should then use $(VICI_INCLDS) and $(VICI_LDPATHS) when specifying the search paths.

The libtool program imposes a limitation on this build approach.

Normally when building a library you can specify other libraries that it is dependent on. This simplifies the list of libraries when building a binary since

you do not have to specify all the dependent libraries, but just the ones directly used. The libtool program does this by recording the dependent libraries in its little configuration file that it keeps for each library (with the suffix of .la). Unfortunately it does not record the actual location of the libraries, but rather the location they will have once installed (as determined by the "prefix" variable).

Thus, if you include a VICI library as a dependency for another VICI library the build will fail because libtool expects the dependent library to have been installed into its final location and therefore cannot find it.

The projects should use the config.h file generated by the configure process as it significantly reduces the command line for the compile.

Each project should include the following in its configure.ac file:

```
AC_CONFIG_HEADERS([config.h])
```

All .cpp files should then have #include <config.h> as their first include. There should be no reference to config.h in include files as it will probably refer to the wrong one when included by other projects.

Add a build directory so that object files, etc. do not clutter up the source directories. Tell Eclipse to use this directory using the Properties | C/C++ Build dialog's Build Location – just press the Workspace button and select the build directory. Adjust the exampleProgram_LDFLAGS entry in the Makefile.am to $(top_builddir)/lib...

A build will now run configure and create the Makefiles. However we need to set the run configuration before we can run the exampleProgram. The autotools will generate a shell script called exampleProgram which you can run from the command line, but Eclipse gets confused and needs to run the actual executable which is located at exampleProgram/.libs/exampleProgram. Unfortunately the library will not be found until we set the Environment for the run configuration to include LD_LIBRARY_PATH set to the path to the build location for the library.

Do a build and ensure both the library and exampleProgram build successfully.

From the Run menu, select Run Configurations... then select the New icon (top left). Give the run configuration a name, then press the “Search Project” button to select the exampleProgram. Add any arguments that you need for testing, and under the Environment tab add LD_LIBRARY_PATH and a value similar to ${project_loc}/build/libxxx/.libs – substituting you library.

2.6 GUI Projects

The Qt libraries expect to be built using the Qt build system. We are not using that, but are using autotools instead. The following changes need to be made to the configure.ac and Makefile.am files in order to support building Qt based programs and libraries.

2.6.1 Macros

A set of configure macros have been developed to confirm that Qt is correctly installed. Add a directory called m4 to the project, and copy the file brqt.m4 into it. This file should be updated to refer to the current version of Qt.

Modify the configure.ac file to include

```
AC_CONFIG_MACRO_DIR([m4])
...
# Qt checks
BR_QT_TEST
```

Modify the top level Makefile.am to include

```
ACLOCAL_AMFLAGS = -I m4
```

2.6.2 Makefile

There will be generated code consisting of an application.cpp file based on the application.qrc file, and moc_xxx.cpp for each header that includes a Q_OBJECT macro. We tell the make system about these with

```
# stuff generated by the moc and rcc programs go here
nodist_command_SOURCES = moc_mainwindow.cpp  application.cpp
```

Add the following defines to the AM_CXXFLAGS:

```
$(QTCXXFLAGS)
```

Add the required libraries to the LDFLAGS, such as $(QTLDFLAGS) and $ (QTLIBS).

Add the required includes to the CPPFLAGS, such as

```
$(QTINC)
```

The commands to build the Qt files are:

```
# instructions for the moc pre-processor
moc_%.cpp: %.h
            $(QT_MOC) $< -o $@

# instructions for the resource compiler
application.cpp:          $(top_srcdir)/src/application.qrc $(top_srcdir)/src/images/*.png
            rcc -name application $(top_srcdir)/src/application.qrc -o application.cpp
```

Eclipse needs to be told of the location for the Qt libraries:

Add /usr/include/QtGui, etc to Paths and Symbols | Includes

2.6.3 GUI Programs

Start by creating a new project: File | New | C++ Project and then selecting GNU Autotools | Hello World C++ Autotools Project and set the project name.

Then select Next and set the details. Then select Next and Advanced Settings, under “Autotools | Configure Settings” set the following Directory Specifiers:

- --prefix: ${VICI}

Under Advanced add the following for the compiler options:

CXXFLAGS="-g -O2"

Then select OK and Finish.

Add a nodist to the Makefile.am for the generated code

```
nodist_vici_SOURCES = moc_vici.cpp application.cpp
```

Add linker options to the Makefile.am for our libraries and the Qt libraries.

```
# Linker options for vici
vici_LDFLAGS =  $(VICI_LDPATHS) $(QTLDFLAGS)
vici_LDFLAGS += $(QTLIBS) -lcfi -lcc
```

Add compiler options to the Makefile.am for our libraries and the Qt libraries.

```
# Compiler options for vici
vici_CPPFLAGS = $(VICI_INCLDS) -I/usr/include/libxml2
vici_CPPFLAGS += $(QTINC)

AM_CXXFLAGS = -std=c++11 $(QTCXXFLAGS)
```

Add the Qt code generator commands to the makefile.am

```
# instructions for the moc pre-processor
moc_%.cpp: %.h
            $(QT_MOC) $< -o $@

# instructions for the resource compiler
application.cpp:          $(top_srcdir)/src/application.qrc $(top_srcdir)/src/images/*.png
            rcc -name application $(top_srcdir)/src/application.qrc -o application.cpp
```

Copy the png image files for the toolbar icons into an images directory, and create an application.qrc file to list them.

Add a build directory so that object files, etc. do not clutter up the source directories. Tell Eclipse to use this directory using the Properties | C/C++ Build dialog's Build Location – just press the Workspace button and select the build directory.

Add the VICI::XINI::config string to your main file to define the search path for your configuration data file.

A build should now produce a “Hello world” program.

Add the vici library path to LD_LIBRARY_PATH and the program should then run.

Tip: If some of the newer features of g++ are not recognised by Eclipse then go to “Properties | C/C++ General | Paths and Symbols” and under the Symbols tab for GNU C++ add the symbol __GXX_EXPERIMENTAL_CXX0X__

3 Using gth-editor To Create Test Scripts

This section describes how to use gth-editor to create a test script for a GUI program.

3.1 Select a Script

Use Open to bring up a list of the tests defined in vici.xml (or whatever the config file is called). Select the test and the program will open the script if it can find one and it has an xml extension.

If there is no existing script you will need to run the test harness to initialise it.

Note that the test harness puts its own path in the script so that the editor can run it (and be sure of getting the correct version of the program).

3.2 Tests for Windows

The Tests for Windows tab allows you to assign test cases to windows.

The tab shows the windows that the test harness has discovered on the left and the test cases on the right. Windows that have test cases assigned to them have a tick in the associated check box. Test cases may have a tick to show that its a test case defined in the test harness and that it has been assigned to a window, or a bar to indicate that it has not been assigned to a window. You can also create test default test cases in this tab which have neither a tick nor a bar.

First select a window to assign tests to. You can also set the Sequence# which allows a window to have several sets of tests assigned to it. In the test harness the sequence number is incremented after the tests for the current sequence number have all been run.

You can assign existing tests to the window by double clicking the test. This will insert the test above the test selected in the centre panel. Alternatively you can type the name of a test case directly into the centre panel.

Press the OK button to save to the internal document (DOM), and Save to write to disk.

3.3 Actions for Test

The Actions for Test tab allows you to define the actions that are run in the constructor and destructor of each test case.

Start by selecting a test case from the left panel, and then select either constructor or destructor.

The table allows you to select an action. The editing is done in the fields below it. Tooltips show what is expected to be entered.

The first field is for a label. This is the destination for a jump – see below.

The next field is time to wait before executing the action in milliseconds.

The “...” button will bring up a list of the widgets (and a tooltip will show the type). Selecting a widget will insert it into the field and set the options in the command drop down.

When you select a command the number of fields on the next row will be adjusted according to the parameters required. The tooltips show what data to enter in the parameter fields.

The third row has two fields, the first is for a regular expression and the second for a comman separated list of variable names. The regular expression should have sub-expressions for extracting values that will be assigned to the variables.

The bottom row has two pairs of fields, each may contain a regular expression and a label. If the regular expression matches the result of the command execution resumes at the action that has the corresponding label. Such jumps are confined to the same test case constructor or destructor and can only be in the forward direction.

The combo box next to Apply button is used to select how the filds are applied to the actions in the table.

Once the actions have been defined press OK to save to the DOM and then Save to write to disk.

3.4 Scripts for Windows

Each window may have a Lua script. This enables the test script to perform complex logic should that be necessary.

Select the window then enter the script in the text entry area. The script may have variables that will remember state between invocations of the functions.

Use the fields at the bottom of the page to enter a command (which is the name of a Lua function in the script) and short descriptions for up to four parameters. Use the Apply button to store the command in the table.

These commands will appear as commands for the script object in the Actions for Tests tab. And will be executed similarly to aactions for widgets.

3.5 Run the Script

The editor can be used to run the script. It will display the log of the tests.

After running the script, reload the script as it may have been updated by the test harness to add new windows.

Add test cases and their actions for the new windows, and repeat.

4 End of Increment Check List

This section provides a check list of tasks that should be done after the coding for an increment has been completed.

4.1 Final Build

The make files should be adjusted so that the test harness is run when “make check” is performed. Make check should run successfully.

A “make install” must be performed so that the project's changes are visible to the other projects.

4.2 Design Documentation

If there has been any changes to the design the design documents need to be updated.

Save any changed documents as PDF and upload to VICI's web site http://ocratato-vici.sourceforge.net/dev_docs.html .

4.3 User Manual

Update the user manual to reflect the changes made in this increment.

Save as PDF and upload to to VICI's web site

http://ocratato-vici.sourceforge.net/user_guide.html .

4.4 Top Integration

All projects must be able to built from the top project. Add the project to top/setup.sh and run the script to create the soft links to the project.

Update config.ac files to reflect the increment number.

After the final commit, remove interfaces/include/svn.h so that it will be regenerated with the current revision number.

Confirm that make install, and make check work.

4.5 Create Distribution

Update the README file.

Confirm that make distcheck works from the top project.

Copy the resultant tar file to ~/Software/vici and install it.

Confirm that make install and make check work as expected.

4.6 Qt5 Validation

Rebuild the distribution for Qt5 and confirm that make check works.

Run any manual tests to confirm the GUI components work correctly.

4.7 API Documents

If the core API has been changed then run the doxygen program to create a new version.

Upload to SourceForge if changed.

4.8 Upload to SourceForge

Upload the tar file to SourceForge.

Update the SourceForge files to reflect the changes for the increment. Don't forget to update the links in the Wiki if necessary.

4.9 Add News Item

Update news.js and upload it to VICI's home page.

5 Web Site Maintenance

5.1 Fedora Forum

There is a thread on the Fedora Forums site that is providing a running commentary on the development of VICI. Please keep it going.

VICI: A Software Development Project

http://forums.fedoraforum.org/showthread.php?t=298342

5.2 SourceForge

The SourceForge site is the public location for the development of VICI.

https://sourceforge.net/p/ocratato-vici/

5.2.1 Summary

Update the Screen Shots as necessary.

Use the Admin metatdata page to update the description. Ensure the status section is kept up to date.

5.2.2 Files

Use the buttons to upload files to the directories. These files can only be downloaded, not browsed in situ.

It is possible to move files to the previous release directory, but it is probably simpler to just delete and re-upload.

The files should be the distributions, the Doxygen generated API and the user guide.

5.2.3 Reviews

Hopefully someone will write one.

5.2.4 Wiki

Use the wiki to describe the development and to discuss interesting features.

5.2.5 Tickets

This is place to put bugs and change requests.

5.2.6 Discussion

Where other developers can discuss the development.

5.2.7 Code

This is managed via Subversion.

5.2.8 External Links

To add a new External Link press the Add New... link on the menu bar.

5.3 VICI Home Page

The VICI web site is intended for documentation on the usage of the VICI application. However, since it is not possible to have browsable PDF or SVG files on SourceForge it is also the location for the development documentation.

http://ocratato-vici.sourceforge.net/index.html

5.3.1 Local Staging Site

Changes to the web site should be made on the local version in ~/public_html/vici first and then copied to the htdocs directory for the site hosted on sourceforge.net.

The following rsync command will copy the local files to the web site:

rsync -avP -e ssh ~/public_html/vici/

brenton-ross,ocratato-vici@web.sourceforge.net:htdocs/

5.3.2 SVG Files

Use the XSL script to add animation to collaboration diagrams.

Remove the height and width entries from the <svg> element so that they will be scaled to the window size by browsers.

5.3.3 News

To update the news side-bar, update the news.js file in ~/public_html/vici, and then upload it to the site.

6 Coding Standards

6.1 Introduction

This documents the coding standards to be used for the C++ code and the shell scripts for the VICI project.

These standards are a “living” document. It is expected that they will be extended and modified as circumstances dictate.

6.2 C Code

6.2.1 ANSI

Where possible the code should conform to ANSI C. However, unless a significant revision is being made a program should not be changed. All files that make up a binary or library (ie share a common Makefile) should use the same style.

6.2.2 Comments

A comment block at the beginning of each file describes its contents. This block should include the file name, the author, the file’s description, and a short form of the GPL license.

A comment block at the beginning of each function should explain the purpose of the function. It should include sufficient detail to be able to tell if the function satisfies the purpose (or not).

At the very minimum a comment line of hyphens should separate one function from the next.

Running comments that describe the purpose of blocks of code or explain why something is done in a specific way will have the following style:

```
/*
 *  comment text…
 *  comment text…
*/
```

Any changes to existing code should include a comment such as

```
/* BR 05DEC98 */
```

and a reference to why it was made (include reference to bug or enhancement doc).

6.2.3 Functions

Functions should be kept short: do one thing, and do it well.

6.2.4 Indenting

When you are making minor changes to existing code use the existing indenting scheme.

For new code, and major revisions the following example illustrates the format to use:

```
if ( a == b )
{
      /* indent one tab – default is 4 characters */
}
```

The opening brace is under the corresponding key word, as is the closing brace. In general braces should be used, even for single statements. (This makes adding debug statements much less error prone.)

Empty statement blocks shall have the semicolon on the following line:

```
while ( *p++ )
      ;
```

6.2.5 Switch Statements

Always have a default case. If falling through one case into the next (ie there is no `break;` statement then use the comment

```
/* fall through */
```

6.3 C++ Code

For the most part the above standards will still apply.

6.3.1 Header Files

All header files, and especially those for libraries, should test and define a variable to prevent duplicate inclusion of the file contents. This should be made up of the namespace name for the sub-project and the name of the file. It should be upper case.

```
#ifndef CFI_FILENAME_H
#define  CFI_FILENAME_H
// file contents
#endif
```

6.3.2 Comments

The // style comment should be used in preference to the /*...*/ style of comment. This allows us to use the /*...*/ style comment to disable multiple lines of code safely during debugging activities.

Where a file has code for more than one class, all code for each class shall be together. A separator at the start of the functions for each class in the following style is highly recommended:

```
//-------------------------------------------------
//                      C l a s s N a m e
//-------------------------------------------------
```

6.3.3 Pointer Members

Any member that is a pointer should have a comment which identifies it as either a referenced or an owned object. A referenced object has its own independent existence, while an owned object would need to be deleted by this object's destructor.

Where a pointer points to an owned object, then you should include a copy constructor and an assignment operator in the header. It is OK to declare but not implement these functions if you think they should never be used, but be aware that many containers rely on these methods.

6.3.4 Standard Template Library

The C++ environment includes access to a reasonable implementation of the STL. The classes provided should be used in preference to re-inventing your own container classes.

7 Common Facilities Infrastructure

This section describes how to use the CFI component for handling errors and debugging the VICI suite. These can be used by including the relevant header file and linking with libcfi.

7.1 String Functions

The stringy.h header file provides some useful routines for manipulating strings.

A typedef, csr, provides a short cut for “const std::string &” which saves a considerable amount of typing.

A trim function removes leading and trailing space characters. A split function breaks a string up into words on space boundaries and an expandMacros function expands environment variables.

A Path class is sub-classed from std::string to provide some useful routines for checking and manipulating paths.

7.2 The vx Class

The vx class provides an exception object which has stream like semantics to make it simpler to construct error messages.

The vx.h file also specifies a set of severities for use in the log messages that should be used as follows:

- Emergency: This should only be used when the program fails and is likely to cause data corruption and damage to the system.
- Alert: This is used when the system is mis-configured.
- Critical: The problem will most likely cause data corruption and demands immediate attention.
- Error: The program has detected a problem that it cannot recover from.
- Code: Some code feature is being used incorrectly. These should be reported to the development team.
- Warning: The program has detected a problem, but is able to recover.
- Notice: Some unusual event has occurred.
- Info: A normal event has occurred.
- Debug: A message used to get the internal state of the system for the purposes of diagnosing a problem.

Typically the exception is thrown as follows:

```
throw vx( VICI::Alert ) << "message " << strerror( errno );
```

A common issue with exceptions and log messages is that it can be difficult to determine where they were first thrown from. To automate this a macro has been defined that includes the file name and line number:

```
throw VX(Error) << "message: " << strerror(errno);
```

7.3 The Xml Class

The Xml class is used to wrap the libxml2 library in a more C++ friendly style.

The Xml class handles the management of the underlying XML file, opening, closing, reading and writing, via the libxml2 library as necessary. It also provides C++ friendly methods for creating, reading and deleting nodes, their content and attributes.

You should derive a class from Xml, usually privately, and provide methods that are application specific for accessing the content. This will have the effect of collecting all the XML specific processing into a single place thus making it easier if the XML format needs to be updated later.

7.4 The XINI Class

The XINI class provides access to configuration data stored in an XML file. The configuration values are accessed via an XPath expression. Two access functions are provided, the first assumes the value is unique while the second retrieves a vector of values. (All values are strings from the content – currently properties are not retrievable but they can be used to select results using the Xpath expression.)

The location of the XML configuration file is determined by searching through a sequence of locations. The order to search, and what to look for is determined by a static std::string variable that must be supplied by the using programs (or a common library).

```
const std::string XINI::config =
"PECHD:-i:VICI_CONF:vici.xml:.vici.xml:/etc/vici/conf.xml";
```

The first sub-string defines the search order:

- P = path provided on the command line.
- E = path provided in the environment.
- C = the current working directory
- H = the user's home directory
- D = directories as listed.

The remaining sub-strings should be in the order specified in the first sub-string, e.g. if P is first, then the second sub-string should be the command line parameter that flags that the subsequent command line parameter is the path.

Any sub-strings beyond those specified in the first sub-string are assumed to be paths and are searched in order.

7.5 The Semaphore Class

A Semaphore class was introduced so that logging to a file could be used by multiple processes. This was necessary to support tracing which is too intensive to use the UDP logging server.

If the XINI configuration includes an entry for “//ipc/keyfile” the application is assumed to be part of a multiprocess system and a real semaphore is used, otherwise it just uses a dummy so that the same code will work for single or multiprocess systems.

Applications should use the SemaphoreLock class so that the semaphore is released when its scope is left, even if its via an exception.

7.6 The UDP Socket Class

A client and server UDP socket was introduced to support a logging server that can assemble a log file from multiple processes. This can, of course be used for other purposes.

7.7 The logging Classes

The log.h file defines a set of logging classes that can be used for different purposes. These are instantiated from a logstream class that can be given a name parameter; this name is used to determine the type of logging based on entries in the XINI configuration file.

Each log configuration is searched for in the configuration file with “/VICI-CONFIG/LOG/” concatenated with the log name.

7.7.1 The StdLogger Class

This can send formatted log messages to either cout, cerr, or clog depending on the “type” specified in the configuration file for this log name, which should be one of “cout”, “cerr”, or “clog”.

7.7.2 The FileLogger Class

This can send formatted log messages to a specified file. The “type” specified in the configuration file should be “file”, and there should be an element “file” containing the full path to the log file.

7.7.3 The PlainFileLogger Class

This is used for logging messages that have been already formatted. The “type” should be specified as “unformatted”. The test harness uses this format for logging the test results.

7.7.4 The UDPLogger Class

This sends formatted log messages to a logging server. This allows us to have a single log file for multiple programs, possibly spread over the local network.

A logging program, vicilogger, is included with libcfi. It looks for the file to write to using the “//LOG/Logger/file” Xpath in the XINI configuration file.

The vicilogger and the UDPLogger both look for the port to use using the “//LOG/*[type='udp']/port” Xpath. The host is similarly specified for the UDPLogger.

7.7.5 The SystemLogger Class

This can send log messages to the Linux system logging daemon. The “type” specified in the configuration file should be “syslog”, and there should be an element “ident” containing the identifier for the log entries.

7.7.6 The TraceLogger Class

This is used by the tracing classes to save the trace data during a test run. The “type” specified in the configuration file should be “trace”, and there should be an element “file” containing the full path to the trace log file.

7.8 The File Descriptor Stream Classes

A recurring problem when using C++ on Linux systems is that many system calls return a file descriptor (integer) but we would like to use the C++ streaming operator. The fdistream and fdostream classes provide the link.

Another recurring problem is that a file descriptor might not get closed if an exception is thrown. The FD class provides a wrapper around a file descriptor that ensures it is closed when the object goes out of scope.

7.9 The Child Process Classes

The ChildProcess class allows us to run another program. Its constructor has optional links to the stdin and stdout of the program so that we can communicate with it if necessary.

A manager class keeps track of the child processes and is primarily responsible for catching the SIGCHLD signal and handling the exit status of the program.

An abstract interface class is provided so that application objects can be notified when the child process terminates.

7.10 The Plug-in Classes

In order to support the ability to test complete programs the infrastructure library includes support for dynamically loaded libraries.

The component supports three styles of plug-in:

- Load and remain resident. This is for a library that is loaded when the program starts and remains until the program exits. It is used to provide optional additional functionality to the program.
- Demand loaded. This is for libraries that are loaded when needed and which may be unloaded after their use has finished. This would be useful if there was a large collection of libraries each of which was only used occasionally.
- Autorun. This is primarily for testing. The library is loaded and the contents executed almost immediately. The library may be unloaded after it has been executed.

The plug-in system allows us to have more than one plug-in object (even of different styles) in a single library. (The library is only removed when all uses have completed.)

Templates are used so that the plug-in component can provide an object of the type specified in the application without having to know about them during the build of the libcfi library.

8 Configuring VICI Modules

8.1 Background

One of the features of VICI is that it consists of 13 application modules (plus a bit of infrastructure). The aim was to demonstrate how to build a large project from several smaller projects that can be independently developed. Since the modules will obviously have dependencies on each other we need some way of building the modules that doesn't require the completed versions of all the other modules to exist.

The first step was to produce a library containing a stubbed version of the facade of each module. This has just enough code to satisfy the compiler, though the functions will usually return NULL pointers, zero or false results. You will need to have -lifstubs as a library to support this.

The second step is to produce test harness versions of the modules that interface to the module we are currently building. The test harness for a module uses these local versions of the surrounding modules to supply test values and check the outputs from the module under test. In total there will probably end up being several dozen of these test modules.

The final step is the actual implementation version of each module.

What we don't want is for the code in the modules to have to be changed as we change from stub to test to implementation - that would just lead to test code being left in the production version. The standard solution to that problem is to use factory classes to create the modules. These are very simple objects that just have one method that returns a new instance of the object. We can thus hide the explicit type of the module in the factory.

However, we still have to tell the factories which version we want. We could use various flags and other parameters to control the factories, but a simpler solution is to just have multiple different types of factories, each of which instantiates the appropriate version of the module. This may seem like we have come full circle back to where we started, but we have made a significant simplification: While the modules are all very different from each other, the factories are quite similar.

We still need to hide the type of module, and since we now have a factory for each type, we again need a factory, but this time a factory for making factories - the FactoryFactory class. Inside this there is an associative array that maps a module identifier (an enum) to the factory for that module. During the testing phase of development this is first populated with the factories for the stub version of the modules. A test harness program can then register its test harness version of modules, which would replace the stub version.

The problem is the final implementation version. The FactoryFactory cannot directly reference these since, for now, most of them don't even exist. Even when we have completed the coding, this will still be an issue since each of the three VICI programs will use a different subset of the modules. We don't want to load up libraries that are not going to be used. The solution is to use the system function dlsym to search for a module specific function by name - if it finds it, the function is used to create the factory - if not its just ignored. Hence the FactoryFactory first does a search for the implementation version of the factories, and then the test harnesses can overwrite them with test harness versions if they need to.

8.2 Configuration Coding

Each module must implement its makeXxxFactory() method, as defined in the extern “C” section of vici.h. This enables the configuration system to find and initialise the factories for each module.

In order for one module to instantiate another the following pattern should be used:

```
FactoryFactory &ff = FactoryFactory::instance();
VICI::EBNF::EBNF_FactoryPtr ef;
ff.getFactory(EBNF_Module, ef);
ebnf = ef->makeEBNF();
```

If a test program has built test versions of modules then they should be installed into the configuration system:

```
FactoryFactory &ff = FactoryFactory::instance();
ff.registerFactory(EBNF_Module, FactoryPtr(new EbnfSyntaxTestFactory) );
```

8.3 Configuration Building

Programs should include -lconfig in their library list.

During the development phase programs the flag -DVICI_TESTING will be defined which will force the configuration to include the stubs, so -lifstubs needs to be included.

(The library list will be provided by an autoconf macro to automate this.)

Appendix A