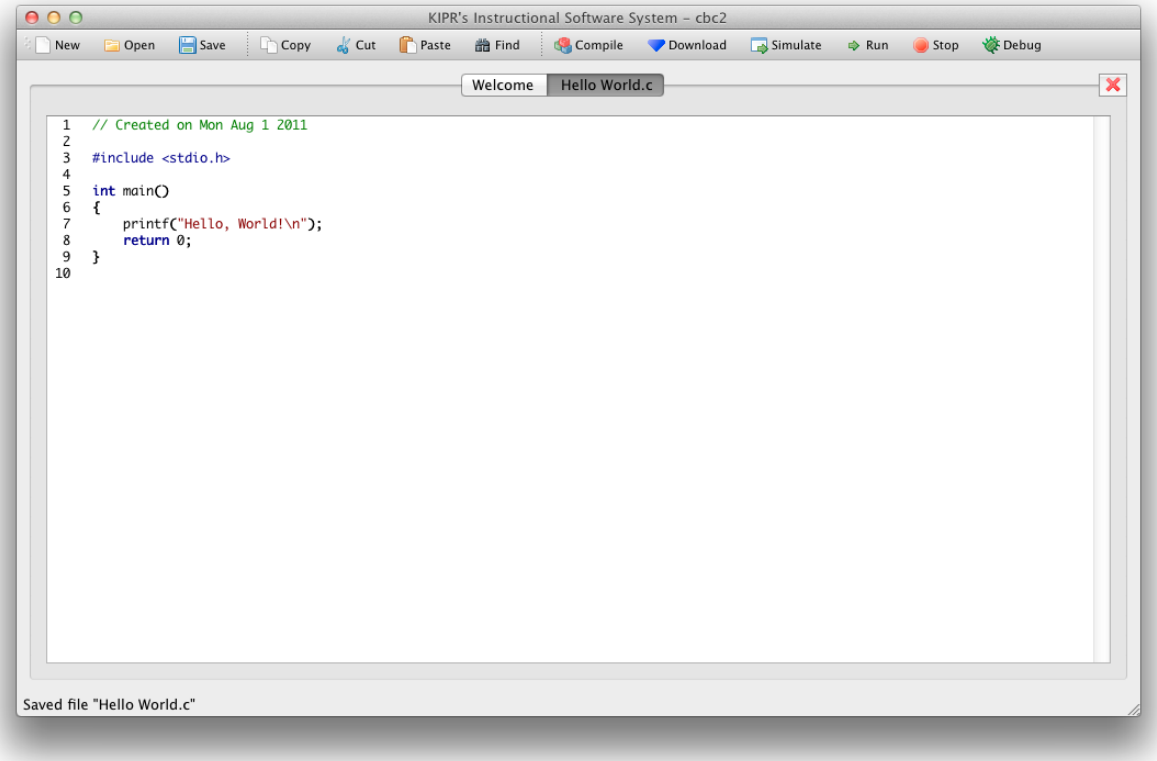
Braden McDorman

KISS Institute for Practical Robotics

KISS IDE

1 Overview

KISS IDE is an extensible IDE aimed at Robotics Development in C, C++, and Java. This document explains the internals of KISS, and should serve as a guide for anyone needing to modify its source.



2 Tabs

A Tab is a class that may be managed and added to the `MainWindow` for viewing and interaction with the user. For a class to identify itself as a Tab, it must implement the `Tab` interface. A Singleton is a global instance of a class, which can be retrieved using the `Singleton::ref()` static method. **`MainWindow` casts between `Tab` and `QWidget`, so a `Tab` needs to also implement `QWidget`.** These classes are implement `Tab` in KISS IDE:

2.1 Tab Classes

`SourceFile` - The most important Tab. Allows editing of text, and interfacing with Targets.

`WebTab` - Implements a simple Web Browser. Used for manuals and Botball Community.

`WelcomeTab` - Extends `WebTab`, removing unnecessary functionality and loading the built-in Welcome HTML.

`VideoPlayerTab` - Implements a Video Player on top of Phonon. To be used for lessons and other media.

`Repository` - A graphical front-end to `KISSArchive`. Also allows downloading of packages from Repositories.

2.2 Tab Usage

Listing 1: The Tab Class

```
class Tab
{
public:
        virtual void activate() = 0;
        virtual void addActionsFile(QMenu* file) = 0;
        virtual void addActionsEdit(QMenu* edit) = 0;
        virtual void addActionsHelp(QMenu* help) = 0;
        virtual void addOtherActions(QMenuBar* menuBar) = 0;
        virtual void addToolbarActions(QToolBar* toolbar) = 0;
        virtual bool beginSetup() = 0;
        virtual void completeSetup() = 0;
        virtual bool close() = 0;

        virtual void refreshSettings() = 0;
};
```

`activate()` - Called every time the tab becomes visible.

`addActionsFile(QMenu* file)` - Passes `QMenu*` to add actions to the File menu.

`addActionsEdit(QMenu* edit)` - Passes `QMenu*` to add actions to the Edit menu.

`addActionsHelp(QMenu* help)` - Passes `QMenu*` to add actions to the Help menu.

`addOtherActions(QMenuBar* menuBar)` - Allows other menus to be added to the menu bar.

`addToolbarActions(QToolBar* toolbar)` - Allows actions to be placed in the tool bar.

`beginSetup()` - A Tab should return `true` if setup was successful. `false` will prevent the Tab from opening.

`completeSetup()` - Useful for setting Tab name. Tab has been added to the `MainWindow` at this point.

`close()` - A Tab should return `true` if it may be closed. `false` otherwise. Prompt for save here.

`refreshSettings()` - Should be a public slot in a Tab's implementation. Called when settings are updated.

2.3 WebTab KISS URLs

WebTab allows HTML to modify the IDE's state through the use of special URLs. These URLs begin with `kiss://`. `kiss://command#param`, where the scheme is `kiss`, `command` is the authority, and `param` is the URL's fragment.

`kiss://new` - Create new file with template.

`kiss://open` - Shows open file dialog.

`kiss://settings` - Show settings dialog.

`kiss://newbrowser#http://google.com/` - Creates a new browser, and loads google.com

`kiss://openfile#path/to/file` - Open file in new source tab.

`kiss://video#path/to/video` - Plays video in new video tab.

`kiss://external#path/to/file` - Opens a file in its default editor. (e.g. PDFs opened in Preview)

3 Singletons

The Concept of a Singleton is an important one in KISS IDE. Several classes use Singletons as their base. A Singleton is a global instance of a class, which can be retrieved using the `Singleton::ref()` static method. These classes implement `Singleton` in KISS IDE:

3.1 Singleton Classes

`MainWindow` - Manages Tabs and Opens Files.

`PluginManager` - An interface for plugin loaders. This is implemented by `LexerManager` and `TargetManager`.

`LexerManager` - Manages Loading of `LexerSpec` plugins.

`TargetManager` - Manages loading of Target plugins.

`SourceFileShared` - A Shared object used to store some common functionality for all `SourceFile` tabs. For example, Pixmaps and the Debugger are shared resources among all `SourceFile` instances.

Singletons for Plugins are necessary, as plugins are inherently single instances. Thus, plugins each have one instance for the entire execution of KISS.

4 Plugins

`PluginManager` is the interface used by both `TargetManager` and `LexerManager` to load plugins. An implementing class must implement `getExpectedLocation(const QString& name)`, which returns the directory to look for a plugin with the given `name`. In `TargetManager`, this returns `targets/${name}`, while in `LexerManager` this returns `lexers`. PluginManager also relies on the implementer to `unloadAll()` plugins on destruction.

4.0.1 Plugin Naming

`PluginManager` expects plugins to follow the naming scheme `lib${name}_plugin.${os_lib_ext}`. This is the default on Mac OS X and Unix, but Windows requires some modification. Here is an example of how this is achieved in the python target:

Listing 2: Python Target

```
TARGET = $$qtLibraryTarget(python_plugin)
win32:TARGET = $$qtLibraryTarget(libpython_plugin)
```

4.1 Lexers

The Lexer system has several parts, including `Lexer`, `LexerProvider`, `LexerSpec`, and `LexerManager`. This may be quite confusing at first glance, but provides a robust way to interface with QScintilla. `Lexer` implements `QsciLexer` for wrapping `LexerSpec` classes. `LexerSpec` is a struct that a plugin developer should implement to highlight syntax. `LexerProvider` is the interface for plugins, which gives you a `LexerSpec` to configure in the `init()` method of the plugin. `LexerManager` manages the loading of `LexerProvider`s.

4.2 LexerManager Notes

`LexerManager` keeps track of lexer extensions in a thin layer on top of `PluginManager`. This is necessary because lexers can be registered to multiple extensions, while `PluginManager` expects a 1:1 mapping. It is recommended you use the `lexerSpec(const QString& ext)` function to return the appropriate LexerProvider for a given extension rather than `get(const QString& name)`.

5 KISS Archives

KISS Archives are a way to install and uninstall optional components into KISS. All targets, lexers, and video lessons build KISS Archives, some of which are preinstalled for distribution. The `KissArchive` class provides several static methods for the manipulation and creation of these archives. KISS also has a basic CLI for these commands. The currently installed packages are kept track of in an `installed` file located in the working directory of KISS. Just to be safe, installing/uninstalling KISS Archives unloads all plugins. The easiest way to reload KISS at this point is a restart of KISS.

Listing 3: KISS Archive Header

```
struct KissReturn
{
        bool error;
        QString message;
};

class KissArchive
{
public:
        static KissReturn create(const QString& name, unsigned version,
                const QStringList& platforms, const QStringList& files,
                QIODevice* out);
        static KissReturn install(QIODevice* in);
        static KissReturn uninstall(const QString& name);
        static QStringList list(QIODevice* in);
        static const unsigned version(const QString& name);
        static QStringList installed();
};
```

5.1 CLI Interface

`KISS --createArchive name version platforms contents output.kiss` - Creates a KISS Archive

`KISS --uninstall name` - Uninstalls a kiss archive in the working directory.

`KISS --install file.kiss` - Installs a kiss archive in the working directory.

`KISS --list` - Lists installed KISS Archives

5.1.1 Example Creating KISS Archive

In this example, we create a `contents` file containing newline delimited files to add to a KISS Archive.

Listing 4: Building KISS Archive for Windows

```
find * -type f > contents
${KISS}/deploy/KISS --createArchive name 1 win contents test_archive_win.kiss
```

In this example, we use `win,osx,nix` rather than `win` to specify we want all platforms to be supported.

Listing 5: Building KISS Archive for All Platforms

```
find * -type f > contents
${KISS}/deploy/KISS --createArchive name 1 win,osx,nix contents test_archive.kiss
```

On Mac OS X, the path of KISS is ${KISS}/deploy/KISS.app/Contents/MacOS/KISS

5.2 Repositories

A repository simply is a remote directory served over HTTP. This directory is required to have an `available.lst` file in it, which specifies the packages available for download. The format of the `available.lst` file is as follows: `os<tab>name<tab>version<tab>file`

5.2.1 Example available.lst

Listing 6: Example available.lst

```
win     cbc2     1     cbc2_target_win.kiss
osx     cbc2     1     cbc2_target_osx.kiss
nix     cbc2     1     cbc2_target_nix.kiss
win     c lexer  1     c_lexer_win.kiss
```

6 Targets

Targets implement the `TargetInterface`, which allows the target to specify the actions it can perform, and perform them. The TargetInterface passes a port name with most functions, as Targets are not bound to any specific file or port. The `Target` class stores a port, and provides wrappers for all TargetInterface functions. It is recommended code wishing to use a Target use the `Target` class instead of a `TargetInterface` directly.

Listing 7: Target Class

```
class Target : public QObject
{
public:
        Target(QObject *parent = 0);

        bool setTargetFile(const QString& filename);
        QMap<QString, QString> targetManualPaths();
```

```
        QString requestFilePath();

        QStringList      errorMessages();
        QStringList      warningMessages();
        QStringList      linkerMessages();
        QStringList      verboseMessages();
        QList<QAction*> actionList();

        QStringList sourceExtensions();
        QString defaultExtension();
        bool cStyleBlocks();

        bool hasDownload();
        bool hasCompile();
        bool hasRun();
        bool hasStop();
        bool hasSimulate();
        bool hasDebug();
        bool hasUi();

        bool compile(const QString& filename);
        bool download(const QString& filename);
        bool run(const QString& filename);
        void stop();
        bool simulate(const QString& filename);
        DebuggerInterface* debug(const QString& filename);
        Tab* ui();

        bool hasRequestFile();
        QStringList requestDir(const QString&);
        QByteArray requestFile(const QString&);

        bool error();

        bool hasPort();
        void setPort(const QString& port);
        const QString& port() const;
};
```

6.1 Layout of a Target

`kiss-targets/name` - Target directory

`kiss-targets/name/name.pro` - Qt project file for building

`kiss-targets/name/name.target` - Loaded by KISS at runtime for target information

`kiss-targets/name/name.api` - Used for auto-completion in the editor.

`kiss-targets/name/src/*` - Source Files.

`kiss-targets/name/templates/*` - Templates for New File dialog

6.1.1 Compiled Target Layout

`targets/name` - Target directory (`kiss-targets/root/targets`)

`targets/name/libname_plugin.dylib` - Plugin shared library (dylib on Mac OS X)

`targets/name/name.target`

`targets/name/name.api`

`kiss-targets/name/templates/*` - Templates for New File dialog

6.2 Target File

Listing 8: Example Target File

```
[General]
description=A Description of the Target
display_name=CBCv2
extensions=C Sources (*.c *.h)|C++ Sources (*.cpp *.h *.hpp)
name=cbc2
port_dialog=true
default_extension=c
c_style_blocks=true
request_file_path=/mnt/browser/stage

[Manuals]
Manual=targets/cbc2/manual/cbcmanual.html
Sensors and Motors Manual=targets/cbc2/manual/Sensor_and_Motor_Manual_BB2011.pdf
Video Lessons=videos/videos.html

[win]
cflags=-Wimplicit -include stdio.h ...
include_dirs=targets/gcc/include targets/cbc2/include
lflags=-lcbc2_sim -lkiss -lglfw -lGLee -lopengl32 ...
lib_dirs=targets/gcc/lib targets/cbc2/lib

[osx]
cflags=-arch i386 -Wimplicit -include stdio.h ...
include_dirs=targets/gcc/include targets/cbc2/include
lflags=-arch i386 -lcbc2_sim -lGLee -lkiss ...
lib_dirs=targets/gcc/lib targets/cbc2/lib

[nix]
cflags=-Wimplicit -include stdio.h ...
include_dirs=targets/gcc/include targets/cbc2/include
lflags=-lcbc2_sim -lkiss ...
lib_dirs=targets/gcc/lib targets/cbc2/lib
```

6.2.1 Target File Options

`General/description` - Description to appear on hover in `TemplateDialog`.

`General/display_name` - Name to appear in `TemplateDialog`.

`General/extensions` - Pipe delimited source files this target is allowed to open.

`General/name` - Name.

`General/port_dialog` - True if needs port dialog (for downloading, running, etc.)

`General/default_extension` - Default lexerspec to use, unless the template specifies otherwise.

`General/c_style_blocks` - If the language uses curly brackets in code. Used to turn on/off "Indent All".

`General/request_file_path` - The remote path to look for files by default.

`Manuals` - Holds list of manuals, in the format `Name=Path/to/Manual`. Will not show non-existent manual.

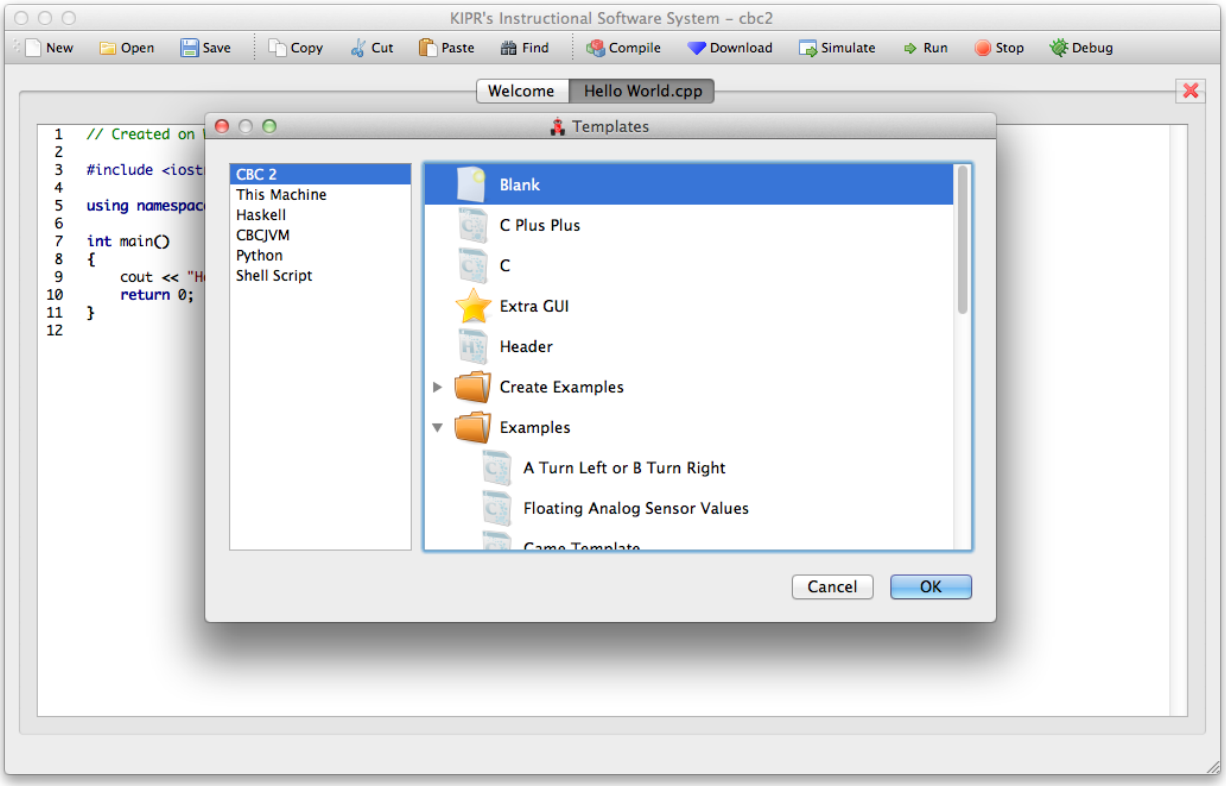
The other sections are left to the target to process, such as compiler and linker flags.

6.3 CBCv2

The CBCv2 target is very similar to GCC, and requires GCC to be installed to work. The CBCv2 target uses cbcserial to communicate with the CBC.

7 Templates

Templates are the files that appear in the `TemplateDialog` when creating a new file. A template has the extension `.template` and has the same name that will be displayed in the `TemplateDialog`. For example, `Hello World.template` will appear in the `TemplateDialog` as "Hello World". Templates also have `.png` icon by the same name in the same directory. If this icon is not found, `TemplateDialog` falls back on a `Default.png` in the same directory. If `Default.png` doesn't exist, no icon is displayed. Templates may be located in the `target/templates` directory, or one subdirectory lower. For example, `target/templates/Examples/Turn Left.template` is also valid, and will appear in an "Examples" folder in the `TemplateDialog`.



7.1 Template Metadata

Template files usually contain exactly the text you want loaded into the editor, but KISS allows us to specify some information in a template file.

`KISS_LEXER <lexer_name>` - Specifies the lexerspec to use instead of the target's default.

`KISS_END_META` - Ends the meta section of a template file, which is always the top of the file. If no metadata is necessary, this keyword may be omitted.

`KISS_DATE` - Inserts the date at any place in the file.

Listing 9: Example Template with Metadata

```
KISS_LEXER cpp
END_KISS_META
// Created on KISS_DATE

#include <iostream>

using namespace std;

int main()
{
        cout << "Hello, World!" << endl;
        return 0;
}
```

Listing 10: Example Template without Metadata

```
// Created on KISS_DATE

#include <stdio.h>

int main()
{
        printf("Hello, World!\n");
        return 0;
}
```

8 Serial Communication with the CBC

Serial Communication has been largely rewritten in KISS IDE. You may link to the `cbcserial` library in `kiss-targets/libraries` to use serial communication. `SerialClient` exposes a few useful functions for serial interaction:

`setPort(const QString& port)` - Set port to communicate over.

`sendCommand(quint16 command, const QByteArray& data = "")` - Sends a command (which is a unique unsigned short) and an associated byte array.

`waitForResult(quint16 command, QByteArray& data)` - Waits for a command with the specified unsigned short to be sent from the CBC. Data is set to the CBC's response

`sendFile(const QString& name, const QString& destination)` - Just a wrapper for `KISS_SEND_FILE_COMMAND`. Name is the path to a file, while destination is the local CBC path to write to.

8.1 Commands

`KISS_SEND_FILE_COMMAND 1` - Sends file to CBC

`KISS_REQUEST_FILE_COMMAND 2` - Request file response from CBC

`KISS_LS_COMMAND 3` - Request ls of given directory

`KISS_RUN_COMMAND 4` - Run given file

`KISS_STOP_COMMAND 5` - Stop currently running program

`KISS_EXECUTE_COMMAND 6` - Execute arbitrary shell command

`KISS_COMPILE_COMMAND 7` - Compile given file

`KISS_CREATE_PROJECT_COMMAND 8` - Create given project name

`KISS_PRESS_A_COMMAND 9` - Simulate A press

`KISS_PRESS_B_COMMAND 10` - Simulate B press

`KISS_PRESS_LEFT_COMMAND 11` - Simulate Left press

`KISS_PRESS_RIGHT_COMMAND 12` - Simulate Right press

`KISS_PRESS_UP_COMMAND 13` - Simulate Up press

KISS_PRESS_DOWN_COMMAND 14 - Simulate Down press

KISS_RELEASE_A_COMMAND 15 - Simulate A release

KISS_RELEASE_B_COMMAND 16 - Simulate B release

KISS_RELEASE_LEFT_COMMAND 17 - Simulate Left release

KISS_RELEASE_RIGHT_COMMAND 18 - Simulate Right release

KISS_RELEASE_UP_COMMAND 19 - Simulate Up release

KISS_RELEASE_DOWN_COMMAND 20 - Simulate Down release

KISS_GET_STATE_COMMAND 21 - Request State information (sensors and motor values)

KISS_GET_STDOUT_COMMAND 22 - Get stdout change since last request

KISS_DELETE_FILE_COMMAND 23 - Delete file at given file path

KISS_MKDIR_COMMAND 24 - Make given directory

CBC_REQUEST_FILE_RESULT 127 - Response to Request File

CBC_LS_RESULT 128 - Response to ls

CBC_EXECUTE_RESULT 129 - Response to arbitrary command

CBC_COMPILE_SUCCESS_RESULT 130 - Response to compile

CBC_STATE_RESULT 131 - Response to state request

CBC_STDOUT_RESULT 132 - Response to stdout request

8.2 Useful References for Serial Communication

kiss-targets/libraries/cbcserial/SerialClient.cpp - Communicates with the CBCv2

kiss-targets/libraries/cbcserial/QSerialPort.cpp - Makes serial communication cross-platform

cbc/cbcui/src/Serial/SerialServer.cpp - The CBC side of serial communication. Shows how each command's data should be packed.

9 Building KISS

9.1 Mac OS X

Step 1: Download and Install the Cocoa Version (Carbon for PPC) of Qt from
`http://qt.nokia.com/downloads/qt-for-open-source-cpp-development-on-mac-os-x`

Step 2: Download and Extract QScintilla from
`http://www.riverbankcomputing.co.uk/software/qscintilla/download`

Step 3: `cd ${qscintilla}/Qt4`

Step 4: `nano qscintilla.pro`
Change `dll` to `staticlib` under `CONFIG`.
Remove `QSCINTILLA_MAKE_DLL` from `DEFINES`.
`Ctrl-X, Y, Enter` to Save and Exit.

Step 5: `qmake -spec macx-g++`

Step 6: `make`

Step 7: `sudo make install`

Step 8: `cd ${development}`

Step 9: `git clone git@github.com:kissInstitute/kiss.git`

Step 10: `git clone git@github.com:kissInstitute/kiss-targets.git`

Step 11: `git clone git@github.com:kissInstitute/kiss-lexers.git`

Step 12: `echo "KISS=${development}/kiss" > kiss-lexers/kiss.pri` (Absolute path to kiss)

Step 13: `echo "KISS=${development}/kiss" > kiss-targets/kiss.pri` (Absolute path to kiss)

Step 14: `cd kiss`

Step 15: `sh scripts/buildAll.sh ../kiss-targets ../kiss-lexers` (Deploys kiss to `kiss/deploy`)

Step 16: Open deploy/KISS.app and install the packages you want to deploy with.

Step 17: `sh scripts/osx_packager.sh version_number`

Step 18: Your KISS dmg is now ready in the releases folder.

9.2 Windows

The Windows build piggy-backs off of the unix msysgit environment, so all commands should be executed from a msysgit prompt.

Step 1: Download and Install msysgit from
`http://code.google.com/p/msysgit/downloads/list`

Step 2: Download and Install GNU Make from (**Install to C:\gnuwin32**)
`http://gnuwin32.sourceforge.net/packages/make.htm`

Step 3: Download and Install Qt from (**Install to C:\Qt**)
`http://qt.nokia.com/downloads/sdk-windows-cpp`

Step 4: Download and Extract QScintilla from
`http://www.riverbankcomputing.co.uk/software/qscintilla/download` to `C:\Projects\`

Step 5: Download and Install NSIS from
`http://nsis.sourceforge.net/Download`

Step 6: Right Click on Computer, Properties, Advanced System Settings, Environment Variables, Path, Edit... (On Windows Vista or 7)

Step 7: Append `;C:\Qt\mingw\bin;C:\Qt\Desktop\Qt\${version}\mingw\bin;C:\gnuwin32\bin`

Step 8: `mkdir -p /c/Projects`

Step 9: `cd /c/Projects`

Step 10: `cd QScintilla*/Qt4`

Step 11: `qmake`

Step 12: `make`

Step 13: `cp -R Qsci /c/Qt/mingw/include`

Step 14: `cp releases/qscintilla2.dll /c/Qt/mingw/bin`

Step 15: `cd /c/Projects`

Step 16: `git clone git@github.com:kissInstitute/kiss.git`

Step 17: `git clone git@github.com:kissInstitute/kiss-targets.git`

Step 18: `git clone git@github.com:kissInstitute/kiss-lexers.git`

Step 19: `echo "KISS=../../kiss" > kiss-lexers/kiss.pri`

Step 20: `echo "KISS=../../kiss" > kiss-targets/kiss.pri`

Step 21: `mkdir -p kiss-targets/root/targets`

Step 22: Copy distribution mingw to kiss-targets/gcc

Step 23: `cd kiss`

Step 24: `mkdir depends`

Step 25: `mkdir releases`

Step 26: You will need to populate `depends` with `libgcc_s_dw2-1.dll`, `mingwm10.dll`, `phonon_ds94.dll`, `phonon4.dll`, `qscintilla2.dll`, `QtCore4.dll`, `QtGui4.dll`, `QtNetwork4.dll`, `QtWebKit4.dll`

Step 27: `sh scripts/buildAll.sh ../kiss-targets ../kiss-lexers` (Deploys kiss to `kiss/deploy`)

Step 28: Right click on kiss/scripts/KISS.nsi, Compile NSIS Script (Choose Compressor)

Step 29: Choose LZMA (Solid) as compressor.

Step 30: Your KISS installer is now built in the releases folder.

9.3 Linux

Step 1: Install Qt4 and libqscintilla2 development packages using system package manager

Step 2: `cd ${development}`

Step 3: `git clone git@github.com:kissInstitute/kiss.git`

Step 4: `git clone git@github.com:kissInstitute/kiss-targets.git`

Step 5: `git clone git@github.com:kissInstitute/kiss-lexers.git`

Step 6: `echo "KISS=${development}/kiss" > kiss-lexers/kiss.pri` (Absolute path to kiss)

Step 7: `echo "KISS=${development}/kiss" > kiss-targets/kiss.pri` (Absolute path to kiss)

Step 8: `cd kiss`

Step 9: `sh scripts/buildAll.sh ../kiss-targets ../kiss-lexers` (Deploys kiss to `kiss/deploy`)

Step 10: Open deploy/KISS and install the packages you want to deploy with.