Building Unix Squeak (>= 3.2) from source

Ian Piumarta
<ian.piumarta@squeakland.org>

Last edited: 2009-08-12 11:17:55 by piumarta on emilia.local
Translated to .ps/.pdf/.html/.txt: August 12, 2009

Contents

0 The easy way

Beginning with version 3.7-7, everything you need is included with the Unix sources regardless of how you obtained them – either as a tarball or by repository checkout. (This was done to eliminate frequent problems encountered by people combining the repository Unix and Cross sources with a set of incompatible generated sources.)

If you extract a tarball then you will have a top-level directory named `Squeak-X.Y-Z` (for some values of X, Y, and Z). If you are checking out from a repository, you can call the direcory anything you like; for example:

```
$ svn co http://squeakvm.org/svn/squeak/trunk squeak
```

will leave you with a directory called `squeak`. (We'll assume from now on that the directory is called `squeak`.)

Next change to the 'unix' directory within the sources that you just checked out (or extracted from the tarball):

```
$ cd squeak/platforms/unix
```

Build the VM and plugins by running `make`:

```
$ make
```

Then install the VM, plugins and manual pages by running `make` again (with superuser privileges):

```
$ sudo make install
```

To delete the temporary files created during the build process, run `make` one last time:

```
$ make clean
```

That's all there is to it.

1 The hard way: configure, build, install

Unix Squeak is built using the (almost) universal "`configure;make;makeinstall`". If you haven't come across this before, read on...

Create a build directory (which we will call '`blddir`' from now on) and then '`cd`' to it:

```
$ mkdir blddir
$ cd blddir
```

A convenient place is just next to the `platforms` directory, like this:

```
$ cd squeak
$ ls
platforms src ...
$ mkdir bld
$ cd bld
```

Create the build environment by running the script `configure` which lives in the `platforms/unix/config` directory.

> Note: The `configure` script accepts lots of options. To see a list of them, run: '`configure --help`'

Assuming you've created the `blddir` next to `platforms`, this would be:

```
$ ../platforms/unix/config/configure
```

> Note: This assumes that the VMMaker sources are in '`../src`'. However, since the Unix Squeak support code is independent of the image version from which VMMaker generated the interpreter/plugin sources, it is possible that your source distribution comes with more than one `src` directory (corresponding to more than one image version used to generate the sources). In such cases you will have to tell `configure` which source version to use, via the '`--with-src`' option. For example, if there are two source directories called `src-3.2gamma-4857` and `src-3.3.alpha-4881` then you would use *one* of the following commands:
>
> ```
> $ .../configure --with-src=src-3.2gamma-4857
>       or
> $ .../configure --with-src=src-3.3alpha-4881
> ```

Build the VM and plugins by running `make`:

```
$ make
```

> Note: If you want to build just the VM (without external plugins) or just the external plugins (without the VM) then you can use: '`make squeak`' or '`make plugins`' respectively.

Finally install the VM, plugins and manual pages:

```
$ su root
$ make install
```

2 Generating your own VM and plugin sources

Generating your own VM/plugin sources might be necessary for various reasons:

- you want to change the mix of internal vs. external plugins
- you want to remove some plugins from the VM that you will never use
- you've pulled in some updates that modify the Interpreter or plugins
- you've filed-in (or written) a whole new plugin
- etc. . .

Version 3.2 (and later) of Unix Squeak use VMMaker to generate the core interpreter and plugin sources.

Start Squeak in the top-level directory (the one containing the `platforms` directory); for example:

```
$ ls
src platforms ...
$ squeak MyCoolPlugin.image
```

Open a VMMakerTool and modify the setup to your liking.

> Note: The VMMaker configuration used to build the distributions of Unix Squeak is available in `platforms/unix/config/VMMaker.config`.

Then click on the relevant "generate ..." button. You can now '`configure;make;makeinstall`' in your `blddir` (as described above).

> Note: You only need to run `configure` **once** for a given `blddir` (on the same host). If you modify the choice of plugins (or change whether they're internal/external) then you can update the build environment by running the `config.status` script in the `bldddir`, like this:
>
> `$ squeak MyCoolPlugin.image ... generate new sources ... $ cd blddir $ ./config.status $ make`
>
> This is *much* faster than running `configure` all over again. (In fact, `make` should detect any changes to the plugin configuration and re-run `config.status` for you automatically.)

> Note: '`configure`' doesn't actually create any files. The last thing it does is run '`config.status`' to create the configured *files* in `blddir` from the corresponding *file.ins* in the `unix/config` directory. So in the remainder of this document the phrase 'during configuration' means *either* when running '`configure`' for the first time *or* running '`config.status`' to update an already `configured` build environment.

2.1 How `configure` finds the `src` directory

Starting with version 3.7 `configure` looks in two places for the `src` directory, in the following order, and uses the first one that it finds:

- the top-level directory (the one containing the `platforms` directory);
- the `platforms/unix` directory.

In other words, if you want to use your own generated sources without deleting the built-in generated sources, generating them into a `src` directory next to `platforms` (and then re-running `configure`) will do what you want.

3 Adding your own plugins

> Note: This section is intended primarily for plugin developers.

If your plugin requires no platform-specific tweaks then there's nothing for you to do. `configure` (and `config.status`) will provide a default `Makefile` for it that should work. If your plugin requires only platform-independent tweaks (and/or additional hand-written code) then these go in `platforms/Cross/plugins`, and there's nothing for you to do (in Unixland).

On the other hand, if you require special `configure` tests or additional declarations/rules in your plugin's `Makefile` then you need to specify them explicitly.

> Note: Unix Squeak subscribes to the following philopsophy:
>
> *Absolutely everything that is specific to Unix (sources, headers, `configure` and `Makefile` extensions, etc.) lives under `platforms/unix`.*
>
> In other words: there is not (nor aught there be) *any* Unix-related information under the `platforms/Cross` directory. (Unix Squeak is entirely encapsulated under `platforms/unix` and is utterly immune to "random junk" elsewhere in the `platforms` tree.)

First you must create a new directory under `platforms/unix/plugins` named after your plugin. This directory will hold the files describing the additional configuration checks and/or `Makefile` contents. For example, if your plugin is called "MyCoolPlugin" then

```
$ mkdir platforms/unix/plugins/MyCoolPlugin
```

would be the thing to do. (The following sections will refer to this directory as `platdep` since the full path is quite a mouthful of typing for my lazy fingers.)

3.1 Plugin-specific configuration

Your plugin can ask `configure` to run additional tests (and to set additional variables in its output files) simply by including a file called `acinclude.m4` in it's `platdep` directory.

> Note: The `configure` script is 'compiled' from several other files. If you create a '`platdep./acinclude.m4`' file then you *must* 'recompile' `configure`. You can do this by '`cd`'ing to `unix/config` and running '`make`', or (if you have GNU `make`) from the `blddir` like this:
>
> ```
> $ make -C ../platforms/unix/config
> ```

In addition to the usual `autoconf` macros, the following macros are available specifically for Squeak plugins to use:

3.1.1 `AC_PLUGIN_CHECK_LIB(`*lib*`,`*func*`)`

This is similar to the `autoconf` '`AC_CHECK_LIB`' macro.

func is the name of a function required by the plugin, defined in the external (system) library *lib*. The macro checks that the library is available (via '`-l`*lib*') and then adds it to the list of libraries required by the plugin (see the explanation of `[plibs]` in Section 3.2.1 for a description of how library dependencies for plugins are handled).

If *func* cannot be found in *lib* then the plugin will be disabled and a message to that effect printed during configuration. (The VM can still be built, *without* rerunning VMMaker or reconfiguring, and the plugin will simply be ommitted from it.)

3.1.2 `AC_PLUGIN_DEFINE_UNQUOTED(`*keyword*`,`*text*`)`

This is similar to the `autoconf` '`AC_DEFINE_UNQUOTED`' macro.

keyword is a `Makefile` keyword (usually of the form '[*name*]') and *text* is arbitrary text to be associated with it. Calling this macro causes `mkmf` to substitute *text* for all occurrences of *keyword* in the `Makefile` generated for the plugin.

3.1.3 Plugin-specific variables

The following variables are also set during the execution of a plugin-specific `acinclude.m4`:

`${plugin}` is the name of the plugin;

`${topdir}` is the path to the top-level directory (containing `platforms`);

`${vmmdir}` is the path to the VMMaker '`src`' directory.

3.2 Plugin-specific `Makefile` declarations and rules

Three mechanisms are avilable for this:

1. scanning additional dirrectories for sources and headers;
2. including a few additional lines into the default `Makefile`; and
3. replacing entirely the default `Makefile` with a hand-written one.

(The last option isn't as scary as it might sound: read on. . .)

3.2.1 The anatomy of a plugin's `Makefile`

Before proceeding, let's take a minute to understand how Unix Squeak compiles and links files in its default `Makefile` for plugins. The default `Makefile` is shown in Figure 1.

> Note: The keywords appearing between '`[ square brackets ]`' are substituted during configuration by a preprocessor called '`mkmf`' according to the kind of plugin (internal/external) being built.

```
# default Makefile for Unix Squeak plugins

[make_cfg]
[make_plg]

XINCLUDES       = [includes]
OBJS            = [targets]
TARGET          = [target]
PLIBS           = [plibs]

[make_inc]

$(TARGET) : $(OBJS) Makefile
        $(LINK) $(TARGET) $(OBJS) $(PLIBS)

[make_targets]

.force :
```

Figure 1: Default `Makefile` "template" for plugins.

`[make_cfg]` is the configured variable section. It contains the platform-specific information gleaned by `configure` while it was figuring out which compiler you have, what flags your linker needs, where to install stuff, and so on.

`[make_plg]` contains a handful of definitions which depend on whether the plugin is being compiled as internal or external:

`o`	the extension for object files
`a`	the extension for plugins
`COMPILE`	the command to compile a source file into an object file
`LINK`	the command to link one or more object files into a plugin

For internal plugins: `$o` is '`.o`' and `$a` is '`.a`'. `$(COMPILE)` is the C compiler ('`$(CC) ... -o`', so the first thing after the command *must* be the output filename) and `$(LINK)` is archiver ('`ar -rc`', again requiring the output file to follow immediately). Note that internal plugins are built as '`ar`' archives before being linked into the final binary.

For external plugins: `$o` is '`.lo`', `$a` is '`.la`', and `$(COMPILE)` and `$(LINK)` are invocations of '`libtool`' to create position-independent objects and shared libraries (with a '`-o`' appearing right at the end, so the first thing after the command *must* be the output filename).

`[includes]` is a list of '`-I\emph{dir}`' compiler flags, one for each of the directories

`src/plugins/`*`name`* `src/vm/intplugins/`*`name`* `platforms/Cross/plugins/`*`name`*
`platforms/unix/plugins/`*`name`*

in which at least one header file is present.

`[targets]` is a list of object files corresponding to the source (`.c`) files found in the directories:

`src/plugins/`*`name`*`/*.c` `src/vm/intplugins/`*`name`*`/*.c` `platforms/Cross/plugins/`*`name`*`/*.c`
`platforms/unix/plugins/`*`name`*`/*.c`

where each source file has been stripped of the directory name and had the '`.c`' converted into '`$o`'.

`[target]` is the name of the plugin, including the `$a` extension.

`[plibs]` is a list of zero or more libraries on which the plugin depends (as detected using the macro `AC_PLUGIN_CHECK_LIB` in the plugin-specific `acinclude.m4`). If the plugin is being built internally then this list is empty and the required libraries are included in the final link command. If the plugin is being built externally then the plugin itself (a shared object) is linked against these libraries (via `[plist]`) rather than with the main VM binary.

(This is to ensure that a missing shared object needed by an external plugin will only affect the operation of that plugin and not prevent the rest of the VM from running, which would be the case if the entire VM were linked against it.)

`[make_inc]` is the contents of the `Makefile.inc` file in your plugin's `platdep` directory (or empty if this file doesn't exist).

`[make_targets]` is a list of rules for building the files listed in `[targets]`. Each rule looks like this:

```
name$o : original/source/dir/name.c
        $(COMPILE) name$o original/source/dir/name.c
```

3.2.2 A note about `$(COMPILE)` and `$(LINK)` commands

You should *never* pass additional flags to these commands explicitly. This is because you cannot know how they are defined. (Their definitions depend on whether the plugin is being built internally or externally — and might even change radically in future releases of Unix Squeak.)

Instead you should pass additional compiler/linker flags to these commands by setting the following variables in '`Makefile.inc`' or '`Makefile.in`':

`XCPPFLAGS`	'`-I`' flags for `cpp`
`XDEFS`	'`-D`' flags for `cpp`
`XCFLAGS`	anything to be passed to the compiler
`XLDFLAGS`	anything to be passed to the linker

> Note: '`mkmf`' already uses '`XINCLUDES`' to pass the list of directories containing plugin header files to `cpp`. You can redefine it if you like, but make sure that '`[includes]`' appears in its definition (or in the definition of '`XCPPFLAGS`').

3.2.3 Specifying additional source directories

`mkmf` looks for a file in your plugin's `platdep` directory called '`mkmf.subdirs`'. If this file exists then it should contain a list of directory names relative to the top-level directory (the one containing the `src` and `platform` directories). These directories will be added to the list of locations searched for '`.c`' and '`.h`' files while constructing the substitutions for '`[includes]`', '`[targets]`' and '`[make_targets]`'.

3.2.4 Including additional material in the default `Makefile`

If the file `platdep/Makefile.inc` exists then `mkmf` will substitute its contents into the `Makefile` in place of the `[make_inc]` keyword.

> Note: `Makefile.inc` is read into the `Makefile` under construction *before* `mkmf` performs substitutions on the '`[keyword]`'s. In other words, your `Makefile.inc` can use the above keywords to include relevant declarations and rules without worrying about whether the plugin is internal or external.

3.2.5 Replacing the default `Makefile` entirely

If neither of the above are sufficient then you can create a complete `Makefile` template called `platdep/Makefile.in`. `mkmf` will use this template instead of the default `Makefile` template shown earlier, and will perform keyword substitutions on it as described above to create the final `Makefile`. (In other words, simply copying the default template shown earlier will result in a `Makefile` identical to the one that `mkmf` would have produced by default.

3.3 Examples taken from existing plugins

By way of example we'll look at how two existing plugins specialise their configuration and `Makefile`s.

3.3.1 Configuration

The `B3DAcceleratorPlugin` requires OpenGL in order to compile. The file `unix/plugins/B3DAcceleratorPlugin/acinclude.m4` contains a single call to an `autoconf`-style macro:

```
AC_PLUGIN_SEARCH_LIBS(glIsEnabled, GL)
```

This works similarly to the `autoconf` '`AC\_SEARCH_LIBS`' macro: If a library `libGL.\{a,so\}` (OpenGL) exists and exports the function `glIsEnabled()` then '`-lGL`' is added to the final VM link command. Otherwise the plugin is disabled (and a message warning of the fact is printed).

> Note: There's a bug here. This should also check for '`GL\_VERSION\_1\_1`' in headers.

3.3.2 Customising the `Makefile`

The `Mpeg3Plugin` requires a (modified) `libmpeg` to be compiled along with it. The sources for this library are in (several) subdirectories of `Cross/Meg3Plugin` and they require additional `cpp` definitions in order to compile correctly.

To cope with the additional directories, `unix/plugins/Mpeg3Plugin/mkmf.subdirs` simply lists them:

```
platforms/Cross/plugins/Mpeg3Plugin/libmpeg
platforms/Cross/plugins/Mpeg3Plugin/libmpeg/audio
platforms/Cross/plugins/Mpeg3Plugin/libmpeg/video
```

To cope with the additional `cpp` definitions, we could have written a tiny `Makefile.inc` containing:

```
XDEFS    = -DNOPTHREADS
```

Unfortunately the additional source directories contain various utility and test programs (which *must not* be built) so we cannot rely on `mkmf` generating the correct `[targets]` list.

Instead we just copy the default `Makefile` "template" (shown above) as `Mpeg3Plugin/Makefile.in` and insert the required list of targets (and `cpp` definition) manually. The end result is shown in Figure 2.

> Note: The default '`[make\_targets]`' will contain additional rules for the objects that we're trying to avoid building (because it's built from an exhaustive list of '`.c`' files in the source directories). This does no harm since the offending rules can never be triggered (their targets are not listed in '`OBJS`').

```
# Makefile.in for Mpeg3Plugin in Unix Squeak

[make_cfg]
[make_plg]

TARGET  = Mpeg3Plugin$a

PLUGIN  = Mpeg3Plugin$o
VIDEO   = getpicture$o headers$o idct$o macroblocks$o etc...
AUDIO   = dct$o header$o layer1$o layer2$o layer3$o etc...
LIBMPEG = bitstream$o changesForSqueak$o libmpeg3$o etc...

OBJS    = $(PLUGIN) $(VIDEO) $(AUDIO) $(LIBMPEG)

XINCLUDES       = [includes]
XDEFS           = -DNOPTHREADS

$(TARGET) : $(OBJS) Makefile
	$(LINK) $(TARGET) $(OBJS)

[make_targets]

.force :
```

Figure 2: `unix/plugins/Mpeg3Plugin/Makefile.in`

3.4 Coping with VMMaker quirks

VMMaker will refuse to compile a plugin if it thinks the plugin requires platform support. This is "all-or-nothing": if platform support is required on *one* platform then it is required on *all* platforms (even if the plugin compiles quite happily without platform support in Unix).

The easiest way to add "null" platform support is to place an empty '`Makefile.inc`' in the plugin's `platdep` directory. (To see this in action, look in `unix/plugins/JPEGReadWriter2Plugin`.)

3.5 If all else fails

(Where "all else failing" is defined as: "after trying for 20 minutes and still getting nowhere".)

If you're writing a plugin that needs platform support (beyond dumb inclusion of a few additional '`.c`' files) and this document has been of no help at all (or if you understood it but you're still suffering from "all else failing") then send me mail and I'll be happy to help you with the various platdep files.

Index