How to build and run Squawk3G

This version of the Squawk VM is very much a pre-alpha, "work in progress" release that is being made available to certain interested parties within Sun Microsystems. It contains application programs that we may legally use, but we must not distribute outside Sun.

Therefore please to not give this distribution to anybody else.
Please refer any interested parties to either
nik.shaylor@sun.com or doug.simon@sun.com

The current version of Squawk uses a temporary core virtual machine called the 'Slow VM'. This is an interpreter for the Squawk3G Java virtual machine. As the name suggests it is not optimized for speed and runs at about 30% that of other interpreter based Java virtual machines. This is a temporary program written in C that will be replaced once the full Squawk3G architecture is complete. The full Squawk3G architecture will include an interpreter and compiler that are all written in Java and can compile itself in such a way that no C code will be required.

Current system requirements:

Windows 2000 / XP

- PC computer running Windows 2000 / XP
- Java SDK 1.4.2
- Microsoft Visual C++ 6.0

Sparc Solaris

- Solaris 9
- Java SDK 1.4.2
- cc (Sun WorkShop 6 update 2 C 5.3 2001/05/15)

Mac OS X

- Mac OS 10.3
- Java SDK 1.4.2
- gcc
-

Our primary development platform is **Windows 2000** but the steps outlined in this document should be reproducible on the other platforms. There is also a chance that these steps may work on Linux/x68/gcc but it has not been tested.

1, Build the Java components

This uses the standard Squawk build procedure to compile all the Java source code using javac and to preverify the J2ME classes. Note that is important that the **java** executable used below is the one located in the **bin** directory of the JDK installation (as opposed to the executable that is often in a different location if you also have the JRE installed on your system).

```
D:\w\work\zcvs\Squawk3G>java -jar build.jar
os=Windows XP
java.home=c:\j2sdk1.4.2_01
java.vm.version=1.4.2_01-b06
C compiler=msc
Cleaning...
Building bco...
Building j2me...
Building jasmin...
Generated: jasmin\classes\jasmin\JasminTests.class
Building graphics...
Building samples...
Building prototypecompiler...
Building compiler...
Building translator...
Building j2se...
Note: temp\src\com\sun\squawk\vm\Channel2.java uses or overrides a deprecated API.
Note: Recompile with -deprecation for details.
Building vmgen...
Building mapper...
Building romizer...

D:\w\work\zcvs\Squawk3G>
```

Note that an alternate debugging version can be built instead using the following command:

```
D:\w\work\zcvs\Squawk3G>java -jar build.jar -debug
```

This will result in a system that is many times slower, but it can be very useful to test the system if the core system classes are modified.

2, Build the ROM image

Before the Squawk VM can run, an image of the Squawk system classes must be made. A number of different possible system images can be made, but the following is the one we use for demonstrations. It will build an image that includes the system classes, the bytecode translator, and the graphics library. (Note that the term 'ROM' image is because this is a read-only image of the classes and methods - you do not need any special hardware for this procedure.)

```
D:\w\work\zcvs\Squawk3G>java -jar build.jar -prod -o2 rom j2me translator graphics
os=Windows XP
java.home=c:\j2sdk1.4.2_01
java.vm.version=1.4.2_01-b06
C compiler=msc
Running romizer...
[translating suite squawk ...]
Romizer processed 585 classes and generated these files:
  D:\w\work\zcvs\Squawk3G\squawk.sym
  D:\w\work\zcvs\Squawk3G\squawk.suite
  D:\w\work\zcvs\Squawk3G\slowvm\src\vm\rom.h
Compiling slowvm\src\vm\squawk.c ...
squawk.c
```

3, Configuring the environment for use of the embedded VM

The Squawk system currently starts up an embedded J2SE VM (via the JNI Invocation API) to implement its I/O system. For this embedded VM to start properly, an environment variable needs to be set telling Squawk where the dynamic or shared libraries implementing the J2SE VM are. Running the following command will tell you how to set this variable for your system:

```
D:\w\work\zcvs\Squawk3G> java -jar build.jar jvmenv
os=Windows XP
java.home=C:\j2sdk1.4.2_01
java.vm.version=1.4.2_01-b06
C compiler=msc

To configure the environment for Squawk, try the following command:

    set JVMDLL=C:\j2sdk1.4.2_01\jre\bin\server\jvm.dll
```

or on Solaris:

```
[pn13]dsimon:/home/dsimon/work/Squawk3G>java -jar build.jar jvmenv
os=SunOS
java.home=/home/dsimon/j2sdk1.4.2_04
java.vm.version=1.4.2_04-b05
C compiler=cc

To configure the environment for Squawk, try the following command under bash:

    export LD_LIBRARY_PATH=/home/dsimon/j2sdk1.4.2_04/jre/lib/sparc:$LD_LIBRARY_PATH

or in csh/tcsh

    setenv LD_LIBRARY_PATH /home/dsimon/j2sdk1.4.2_04/jre/lib/sparc:$LD_LIBRARY_PATH
```

3, Running squawk

Now that the Squawk executable has been built and the relevant environment variable has been set, you can run squawk without any arguments to get a usage message:

```
D:\w\work\zcvs\Squawk3G> squawk
Missing class name
Usage: squawk [-options] class [args...]

where options include:
    -cp:<directories and jar/zip files separated by ':' (Unix) or ';' (Windows)>
                            paths where classes, suites and sources can be found

    -suite:<name>           suite name (without ".suite") to load
    -resourcepath:<path>    set the resource path
    -verbose                report when a class is loaded
    -veryverbose            report when a class is initialized or looked up plus

                            various other output
    -egc                    enable excessive garbage collection
    -nogc                   disable application calls to Runtime.gc()
    -positivetck            indicate execution of a positive TCK test
    -negativetck            indicate execution of a negative TCK test
    -stats                  display execution statistics before exiting
    -X                      display help on native VM options
    -h                      display this help message

** VM stopped exit code = 0 **
```

Specifying the **–X** argument displays a usage message for the arguments that apply to the native (i.e. C code) part of the VM:

```
D:\w\work\zcvs\Squawk3G> squawk -X
    -Xmx<size>     set Squawk RAM size (default=8192Kb)
    -Xmxnvm<size>  set Squawk NVM size (default=8192Kb)
    -Xboot:<file>  load bootstrap suite from file (default=squawk.suite)
    -Xdsym:<file>  dump dynamic symbols to file (default=squawk_dynamic.sym)
    -Xtgc[:<n>]    set GC trace flags (default=1) where 'n' is the sum of:
                     1: minimal trace info of mem config and GC events
                     2: trace allocations
                     4: detailed trace of garbage collector
                     8: detailed trace of object graph copying
    -Xtgca:<n>     start GC tracing at the 'n'th collection (default=0)
```

The options above are the standard ones for a ‘production’ build. By omitting the **`-prod`** option, the system will be built with its internal instruction tracing system enabled which may be of use when debugging the VM. This enables a few extra native VM options as shown below:

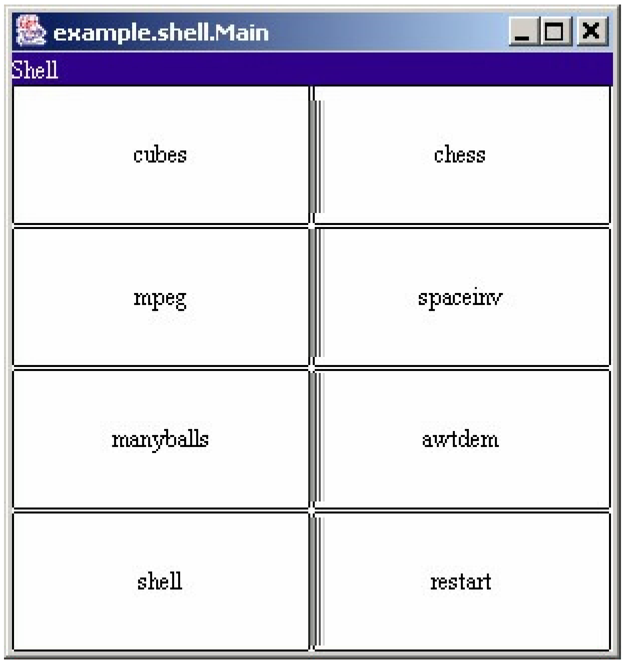
```
D:\w\work\zcvs\Squawk3G> squawk -X
    -Xmx<size>     set Squawk RAM size (default=8192Kb)
    -Xmxnvm<size>  set Squawk NVM size (default=8192Kb)
    -Xboot:<file>  load bootstrap suite from file (default=squawk.suite)
    -Xdsym:<file>  dump dynamic symbols to file (default=squawk_dynamic.sym)
    -Xtgc[:<n>]    set GC trace flags (default=1) where 'n' is the sum of:
                     1: minimal trace info of mem config and GC events
                     2: trace allocations
                     4: detailed trace of garbage collector
                     8: detailed trace of object graph copying
    -Xtgca:<n>     start GC tracing at the 'n'th collection (default=0)
    -Xts<n>        start tracing after 'n' instructions
    -Xte<n>        stop tracing after 'n' instructions
    -Xtr<n>        trace 200000 instructions after 'n' instructions
    -Xterr         trace to standard error output stream
    -Xst<n>        dump a stack trace every 'n' instructions
```

5, Running the demo

The demo is run with the following command:

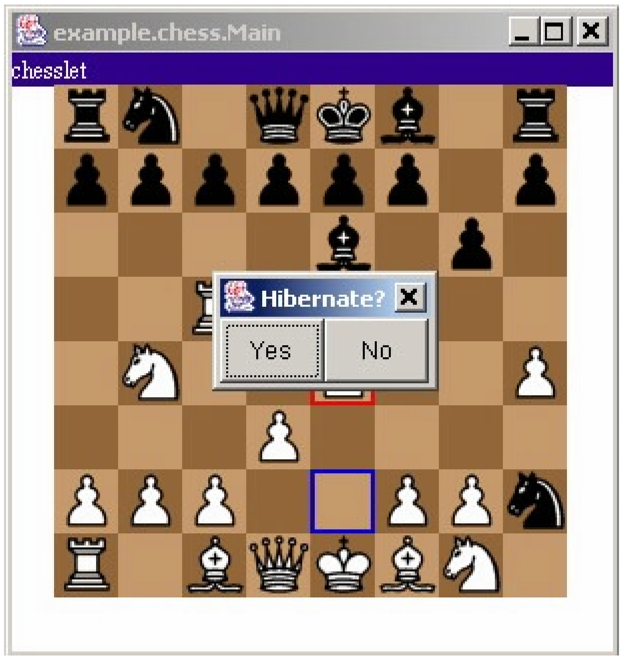
```
D:\w\work\zcvs\Squawk3G>squawk -verbose -cp:samples/j2meclasses example.shell.Main
[Loaded example.shell.Main]
[Loaded example.shell.Shell]
[Loaded example.shell.Shell$MainPanel]
[Loaded example.shell.Shell$1]
[Loaded example.shell.Shell$2]
```

You should see similar output as above on the console as some classes are loaded and then the launcher window will appear:



One of the more interesting features of the demo is that the running programs can be saved and restored. This is a very new feature for which there are a few known problems that we are currently working on. However it is fairly robust and can be demonstrated.

The general way in which applications are saved is to right-click the mouse over the application. A small pop-up window will appear asking for confirmation of the hibernation operation:

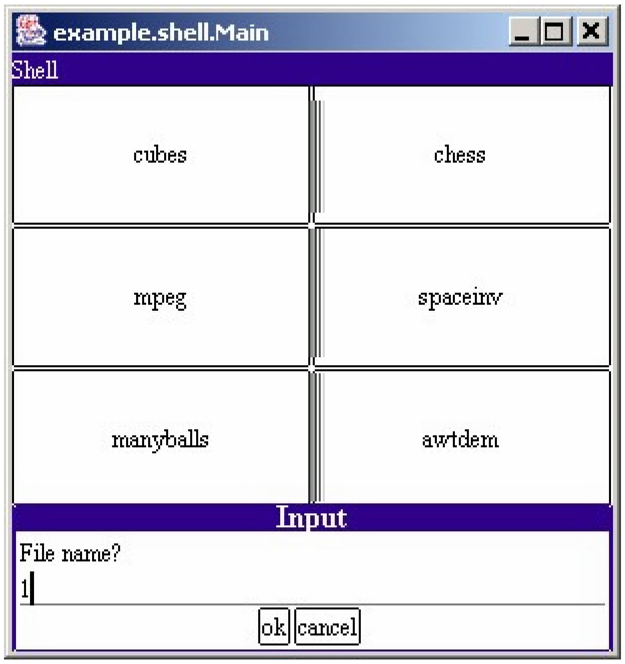


After pressing **"Yes"**, a message is printed on the console informing you of the identifier of the saved isolate:

```
Serializing channel to 2.cio
example.chess.Main finished
Saved isolate to file://2.isolate
```

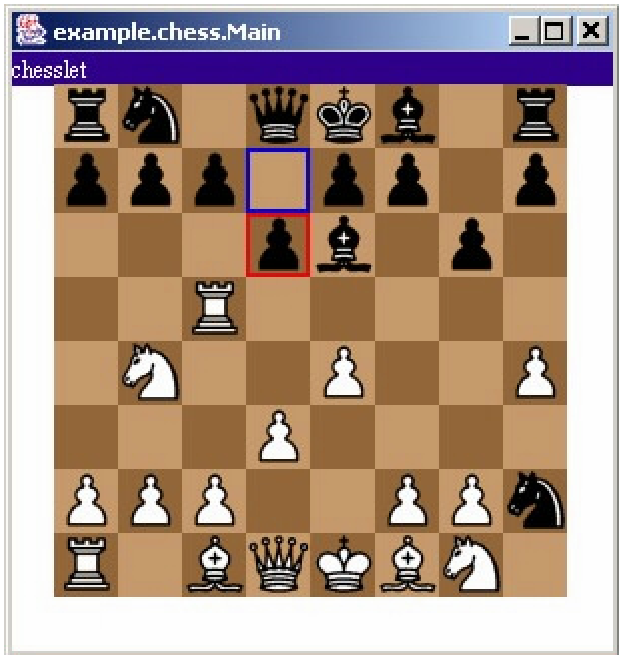
You will also note two files are created based on this identifier, one with a "**`.isolate`**" suffix and the other with a "**`.cio`**" suffix.

To un-hibernate an isolate select "restart" from the launcher and then type in the number of the isolate to restart. Just enter the number (without a suffix) and click on "ok":



File name?
1

Once the isolate has un-hibernated, it should be executing exactly where it left off:


example.chess.Main
chesslet