Contents

Directory Structure and Compilation Instructions:

InstallDir - the installation directory of the software suite
(to which the compiled code should be copied)

HackPackageSource - should be compiled and zipped into Hack.jar and
coppied to InstallDir/bin/lib

HackGUIPackageSource - should be compiled and zipped into HackGUI.jar and
coppied to InstallDir/bin/lib

CompilersPackageSource - should be compiled and zipped into Compilers.jar and
coppied to InstallDir/bin/lib

SimulatorsPackageSource - should be compiled and zipped into Simulators.jar and
coppied to InstallDir/bin/lib

SimulatorsGUIPackageSource - should be compiled and zipped into
SimulatorsGUI.jar and coppied to
InstallDir/bin/lib

BuiltInChipsSource - should be compiled and copied to InstallDir/builtInChips

BuiltInVMCodeSource - should be compiled and copied to
InstallDir/builtInVMCode

MainClassesSource - should be compiled and copied to InstallDir/bin/classes

The Chip API for Implementation of Chips in Java

The Nand2Tetris Software Suite allows the implementation in Java of new chips for
use with the Hardware Simulator via the Chip API which is henceforth described.
This feature allows both the ability to achieve greater simulation speed and the ability to provide students with working chips without disclosing their implementation.

It is strongly advised, before implementing a chip, to browse through the implementations of chips provided with the Nand2Tetris Software Suite. The HDL
files of these chips are located under the InstallDir/builtInChips directory and Java sources for these chips - under the BuiltInChipsSource directory.

The input and output pins for each chip are specified by an HDL file for the chip which should bare the name of the chip followed by a .hdl suffix and be located in the InstallDir/builtInChips directory.
The HDL should be formed like a normal HDL file (see the Hardware Simulator Tutorial supplied at www.nand2tetris.org) and specify the chip name and all the input and
output pins in the usual manner. Instead of the list of gates which usually comprise the implementation, the line:
BUILTIN ChipName;
should appear (where ChipName should be substituted for the name of the chip).

In addition to the HDL file for the chip, each chip should be implemented in a
seprate class named the same as the chip. The compiled implementation should reside in the builtInChips package somewhere in the CLASSPATH (for example the compiled counterparts of the aforementioned chip provided with the Nand2Tetris
Software Suite are located under the InstallDir/builtInChips directory).

The chip class should extend (either directly or indirectly) the Hack.Gates.BuiltInGate class and may overwrite any of the following three methods (which by default do nothing):
void clockUp() - called when the clock goes up (useful for clocked chips)
void clockDown() - called when the clock goes down (useful for clocked chips)
void reCompute() - called whenever any of the input pins changes
(useful for combinatorial chips)
Required initialization code may be placed in a constructor accepting no arguments and any number of data members may be defined.

The code for the chip may access the input and outpu pins via the data members
inputPins and outputPins, respectively.

The value of a b-bit input pin/bus which was declared n-th (starting from zero)
in the chip HDL file can be accessed by evaluating the b least significant bits
of the value returned by calling inputPins[n].get().

The value of a b-bit output pin/bus which was declared n-th (starting from zero)

in the chip HDL file can be updated to v (a value in which all but the v least
significant bits are zero) by calling outputPins[n].set(v).

The Nand2Tetris Software Suite Hardware Simulator Chip Java API also provides support
for implementing a GUI visualization of the chip (similar to the one
implemented by the provided ALU, RAM*, ROM32K, ARegister and DRegister
chips).
Implementing chips should extend the Hack.Gates.BuilInGateWithGUI class. See
the implementations of this class (located in
SimulatorsPackageSource/Hack/Games/BuiltInGateWithGUI.java) and of the
aforementioned gui-powered chips for examples and information regarding
additional methods which should be implemented by gui-powered chips.

The VMCode API for Implementation of VM Functions/Classes in Java

The Nand2Tetris Software Suite allows the implementation in Java of new VM functions
for use by the VM Emulator via the VMCode API which is henceforth described.
This feature allows both the ability to achieve much greater simulation speed
and the ability to allow VM programs the access to features which aren't
otherwise available or feasible on the Hack platform (such features may
include e.g. time & date queries, random number generation using an RNG daemon
and performing complex calculation using 3rd-party closed libraries).

Whenever the VM Emulator encounters a call to a VM function with a prefix for
which the current program doesn't contain a *.vm implementation (e.g. a call
to "Screen.drawPixel" when the current program doesn't contain a file called
Screen.vm), the VM Emulator will invoke the Java implementation for that VM
function, if such an implementation exists. This priority mechanism is similar
to the priority mechanism of the Chip API in the Hardware Simulator.
Since this mechanism is not discussed in the book, a dialog will pop up to
confirm the usage of Java implementation of VM functions upon loading of a
program in which such usage is required.

It is strongly advised, before implementing a new VM function, to browse
through the implementation of the Jack OS supplied in the Nand2Tetris
Software Suite.
The Java sources for this implementation are located under the
BuiltInVMCodeSource directory.

It should be noted that since the VMCode API allows the implementation of
functions in the VM level and not in the Jack level, there is no concept of
constructors or methods in this API but only a concept of functions. Functions
may conceptually be class methods and therefore receive a this-style pointer
as the first argument (e.g., as the String.* functions do) or may conceptually
be class constructors which return a this-style pointer which they allocated
and initialized (e.g. as the String.new function does) but may conceptually
be mere functions or static methods (e.g., as the Output.* functions are).

Each VM function is implemented by a single Java static method. Since all VM data is 16-bit quantities and Java shorts are 16-bit quantities, all of the
arguments to the static method should be Java shorts. For convenience, the static method may return void (a value of 0 will be returned to the calling function), boolean (false will be converted to 0, true to 0xffff), char (will be cast to short and returned) or short.

As with normal .vm files (see the VM Emulator Tutorial at www.nand2tetris.org), all
VM functions which start with the same prefix (e.g., all VM functions which implement methods, constructors and functions of the same Jack class) are implemented together. In the case of normal .vm files, all such functions are implemented in a single file baring the prefix (e.g., the class name) as its name and therefore all such function share the same static segment. In the case of Java-implemented VM functions, all such functions are implemented by static methods of a single class baring the prefix as its name and therefore may share data using static variables of the class.

The compiled class should extend (either directly or indirectly) the Hack.VMEmulator.BuiltInVMClass class and should reside in the builtInVMCode package somewhere in the CLASSPATH (for example the compiled classes implementing the Jack OS supplied in the Nand2Tetris Software Suite are located
under the InstallDir/builtInVMCode directory).

A Java static method which implements a VM function may communiate with the Virual Machine using any of the following static methods which it inherits from
Hack.VMEmulator.BuiltInVMClass:

`short readMemory(int address)` - returns the value stored in the VM memory at the given address (the address argument is an int and not a short for convenience but may only be in the one of the ranges HEAP_START_ADDRESS - HEAP_END_ADDRESS or SCREEN_START_ADDRESS - SCREEN_END_ADDRESS (these are provided as static final constants of the Hack.VMEmulator.BuiltInVMClass).

`void writeMemory(int address, int value)` - changes the value stored in the VM memory at the given address (the value is cast to short and the address must be legal - see above). If data flow animation is on, the change in the VM memory will be animated.

`short callFunction(String functionName, short[] params)` - Calls the named VM function (which may be either be implemented in a normal .vm files or in Java using the VMCode API) with the given parameters. The return from the function is returned. To allow for maximum modularity, this function should be used for all calls to VM functions not implemented by the current class, to ensure that the implementation

currently used by the VM Emulator (which may or may not be implemented in Java) is called.

short callFunction(String functionName, short param1, ...) - For convenience, versions of the callFunction method are supplied for calling VM functions accepting 0-4 arguments without the need to allocate an array of parameters.

void infiniteLoop(String message) - Used to halt the VM program (this is the function called by the supplied Java implementation of the Sys.halt function of the Jack OS). If the optional message is provided (non-null) then a pop-up window is opened to display it. * Important: A Java static method implementing a VM function SHOULD NOT enter a blocking infinite loop *

A function calling any of the aforementioned static methods must be declared to throw Hack.VMEmulator.TerminateVMProgramThrowable. An instance of this Throwable class will be thrown by any of the aforementioned static methods if the user decides to restart the VM program (e.g. via the << button). This throwable may be caught by the calling method but must be rethrown.

For convenience, the following constants are provided by the Hack.VMEmulator.BuiltInVMClass for use by the classes which extend it:

```
short SCREEN_START_ADDRESS
short SCREEN_END_ADDRESS
int SCREEN_WIDTH
int SCREEN_HEIGHT
short HEAP_START_ADDRESS
short HEAP_END_ADDRESS
short KEYBOARD_ADDRESS
short NEWLINE_KEY
short BACKSPACE_KEY
```

Finally, it should be noted that the Java language does not allow the declaration of a method called "new" (such as the String.new and Array.new constructors from the Jack OS). The implementation of such VM functions using the VMCode API is achieved by implementing a static Java method called "NEW" (all capitals) - the VM Emulator will call the "NEW" function whenever it receives a request for calling the "new" function.