Contents

Chapter 42

Porting Squeak

"Nothing will ever be attempted if all possible objections must first be overcome." The famous words of Samuel Johnson are particulary relevant to the task of porting Squeak. As we shall see in this chapter, it is a task where most of the objections need not be overcome; they can quite cheerfully be left for that proverbial rainy day...

42.1 Introduction

Squeak must be one of the most ubiquitous programming languages to date. In addition to the original version for Mac OS, Squeak has been ported to a wide variety of very different platforms: most major flavors of Unix, MacOS-X, several variations of Windows and Win/CE, OS/2, several "bare hardware" systems, and so on.

The impressive list of ports has been possible because of the way Squeak cleanly separates the task of interpreting Smalltalk from the task of communicating with its host platform. The interpreter is actually a Smalltalk program within the image (in class Interpreter) which is translated into an equivalent C program that can subsequently be compiled on any system that has an ANSI C compiler. This program makes only one assumption: that pointers and integers are 32 bits wide.[1] Communication with the host platform is performed through a collection of a hundred or so "support functions", which

[1] This assumption may change in the future as 64-bit systems become more widespread. Existing 64-bit systems sometimes provide compiler options to limit pointers and integers to 32-bits, making them "Squeak-friendly".

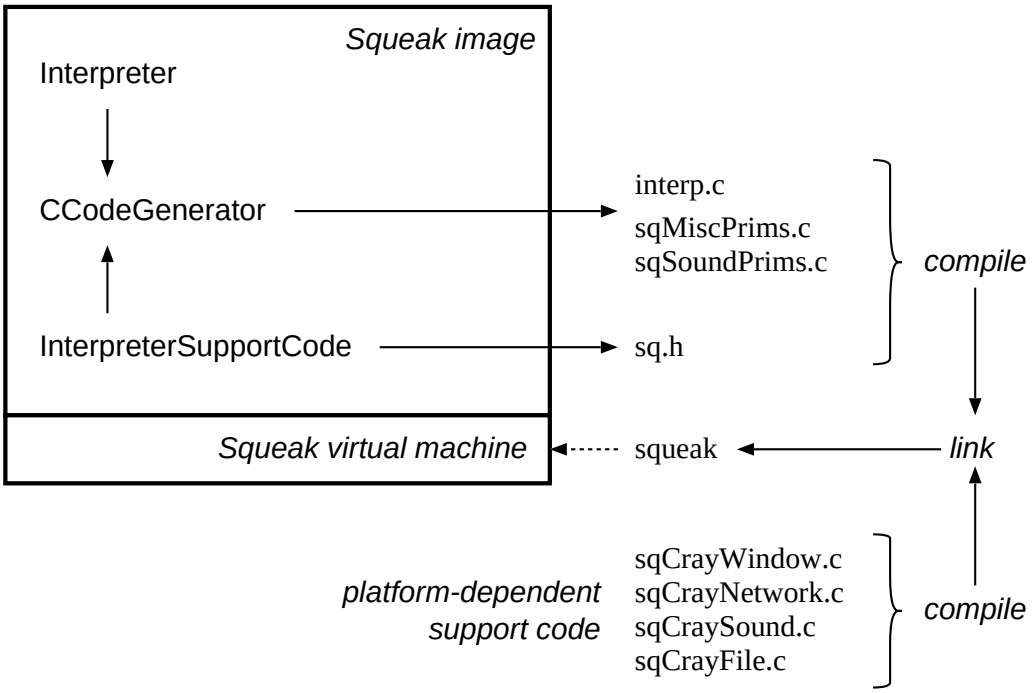

Figure 42.1: The majority of the Squeak virtual machine is generated automatically from an executable specification written in Smalltalk. The class Interpreter is a fully-functional implementation of the Squeak interpreter written in a subset of Smalltalk. The CCodeGenerator converts this Smalltalk program into an equivalent C program. The class InterpreterSupportCode adds many primitives (also written in Smalltalk and converted automatically to C) to the generated code, as well as some hand-written header files that are stored as String constants in the image (for convenience). The generated source and header files are compiled and then linked with platform-dependent support code (written by hand) to create the Squeak virtual machine.

---

perform platform-dependent tasks such as file input/output, updating the screen, and reading keyboard and mouse input. The generated interpreter code and platform support functions are compiled and then linked together to create the final virtual machine. Figure 42.1 illustrates this division of labor.

Looking at the amount and complexity of support code that comes with Squeak, it may seem that porting it to a new platform is a daunting task. This is not necessarily the case. A couple of points in particular make the

task much easier than it might appear.

First is the optional nature of many of the "advanced" features of Squeak (such as support for CD-quality stereo sound recording and output, MIDI, network connectivity, and so on). These features are associated with *primitive methods* in the Squeak image. Each of these primitives calls an associated `C` function in the support code, which normally implements some "external" action (such as opening a network connection) before returning. However, it is perfectly acceptable for these primitives to "fail" instead of implementing the external actions expected of them. An initial port can therefore avoid an immense amount of complexity by implementing tiny "stubs" in the associated support functions that simply "fail" the primitive from which they were called and then return, without performing any additional work at all. Squeak might be a less exciting place in which to play as a result, but at least it will be "up and running" far sooner because of the optional nature of its advanced features.

Second is the inclusion of the tiny file `sqMacMinimal.c` along with the regular Squeak source code. This file is a kind of "skeleton": it contains a complete set of declarations for both the mandatory and optional support routines, and the simplest possible definitions for the mandatory support that yield a running virtual machine on the Macintosh. (The optional functionality is defined too, but trivially—indicating its absence to the Squeak interpreter.) It is a very good starting point for porting Squeak to a new platform. Generating the Macintosh source files from within the image will also generate a copy of this file (see Section 42.3.2).

42.2 About this chapter

This chapter begins by explaining the structure of the Squeak virtual machine, and the process of putting it together based on the various pieces. These pieces are either provided, generated automatically based on Smalltalk code, or written by hand for each supported platform. The mechanism for extracting the provided and generated code from the image is described in Section 42.3.

Section 42.4 gives an overview of the steps involved in porting Squeak, and some tips on how to make the process as painless as possible. Following the instructions in this section will yield a minimal, but working, virtual machine for Squeak on a new platform.

Next come some important details about how the generated code interacts with the support code (Section 42.5), and how the compilation environment is set up for both generated and support code (Section 42.6). This last section also describes the additions that must be made to the compilation environment in order to support a port to a new platform.

The next few sections (42.7 through 42.14) deal mainly with the fine details of each of the mandatory "subsystems" in Squeak, and tell the full story behind the outline given earlier (in Section 42.4). These sections describe functionality that is required for Squeak to work properly, and will therefore be a major source of information during an initial porting effort.

The remaining sections (42.15 through 42.22) tell the tales of the various optional subsystems such as networking and sound. Once an initial port to a new platform is running reliably, they describe how to extend Squeak's capabilities in important and interesting ways.

Finally Section 42.23 offers some closing remarks, including why the significant rate of change in Squeak does not present additional difficulty to the task of porting, before closing the book on the Squeak Porting Story.

This chapter assumes that the reader is already familiar with Smalltalk, and preferably with Squeak. The text refers to several standard Squeak classes without explaining either their purpose or the details of their implementation, except where such explanation is required to understand how they interact with the support code.

Only two typographic conventions are used. Quantities from the `C` universe (variable and function identifiers, constants, and so on) appear in `fixed-width` font. Quantities from the Squeak universe (expressions and class or method names) appear in sans serif font.

42.3 Source code

Squeak tries to generate as much of its own implementation as possible automatically. Such code is referred to in this chapter as *generated code*. The code that depends on the host platform, and which is written by hand when porting Squeak to a new platform, is referred to as *support code*.

The generated code can be extracted from any running Squeak system. The next section explains how.

Writing the support code for a new platform is a more difficult task. To make things simpler, the support code is divided into several *subsystems*,

each of which deals with a particular aspect of Squeak's connectivity with the "outside world". They include user interaction (screen, keyboard and mouse), networking, sound, serial and MIDI ports, and so on. Some of these subsystems are mandatory: they *must* be implemented (or "mostly implemented") to obtain a working Squeak system. The other subsystems are optional; they represent the parts that can be left for that rainy day.

42.3.1 Generating the Squeak source files

The very first task when starting a new port is to generate the platform-independent source files, which will be compiled and then linked with the hand-written support code. These files include the Squeak interpreter itself, automatically-generated implementations of various primitive functions, and a few hand-written header files that are stored in image as String constants.

The Squeak interpreter is traditionally called `interp.c`. It is generated automatically by translating a complete, functional implementation written in Smalltalk into an equivalent `C` program. (See the class Interpreter for details.) This is done by evaluating the expression

```
Interpreter translate: 'interp.c' doInlining: true
```

in a Squeak Workspace. (See the method of the same name in Interpreter class for further details.) This takes a couple of minutes, and writes the generated code to the named file in the current working directory.

The automatically-generated primitives are created in the same manner as the Squeak interpreter itself—from equivalent Smalltalk implementations that are translated into `C`. The hand-written header files are stored in the image as constants, just for completeness. Both sets of files can written to the current working directory by evaluating the expression

```
InterpreterSupportCode writeSupportFiles
```

in a Squeak Workspace. (See the method of the same name in InterpreterSupportCode class for further information.)

42.3.2 Generating the Macintosh support files

The Squeak image also contains a complete copy of the support code for the Macintosh, divided into several source files, each of which corresponds to one

of the subsystems described in this chapter. Some of these files contain very good documentation, and offer much more information (as comments) than could reasonably be included here.

Porting these susbsystems to a new platform is therefore best accomplished by copying and then modifying the Macintosh version; unless one of the other ports of Squeak already offers support that is similar to the target platform, of course. Evaluating the expression

```
InterpreterSupportCode writeMacSourceFiles
```

in a Squeak Workspace will write these files to the current working directory.

If no file system is available for writing out copies of the hand-written files stored in the image, their contents can be browsed by looking in the 'source files' protocol of InterpreterSupportCode class.

42.4 Getting started

This section gives some overall advice about how to go about porting Squeak to a new platform. It begins by outlining which support functions are essential, and in what order they might be implemented to obtain a (partially) working Squeak system in the shortest amount of time.

It is very useful to have a running Squeak system available when porting to a new platform.[2] Apart from being able to look inside the image to see how the generated code interacts with the support code, it also affords the possibility of making a "customized" image for use during testing. (For example, such an image might disable the logging of errors to a file until the file system is fully operational in the new port.)

42.4.1 Roadmap to porting Squeak

Some Smalltalk programmers like write their code in a "demand-driven" style, implementing missing functionality when the Debugger pops open in

[2]This is by no means essential. The very first port of Squeak (to Unix) was made without access to a Macintosh, which at the time was the only platform on which Squeak ran. Apart from ten minutes spent on a Macintosh right at the start of the porting process (to generate the `interp.c` file and the Macintosh support code for reference purposes), the port was performed entirely "offline" up to the moment when Unix Squeak could load and run an image for itself.

response to doesNotUnderstand:. This style of development also makes perfect sense for porting Squeak to a new platform.

A copy of the file `sqMacMinimal.c` (extracted from the image, as explained in Section 42.3.2) already contains trivial implementations of the optional functions, that will fail "gracefully" when the image tries to use an unimplemented feature. All that this file is missing are implementations of the "essential" functions.

With just a few minutes' work, trivial implementations of the essential functions can be added to `sqMacMinimal.c`. They can simply print a message (indicating the name of the function) before exiting.

Once this is done, the next step is to try to compile and link the Squeak virtual machine. At the time of writing, the following files are required:

- `interp.c` — automatically-generated bytecode interpreter;
- `sqMiscPrims.c` — automatically-generated primitives;
- `sqNamedPrims.c` — as generated by the image;
- `sqNamedPrims.h` — modified manually so as to declare no primitives at all (see Section 42.20);
- `sqFooWindow.c` — the modified copy of `sqMacMinimal.c`, lacking all of the essential support for platform "Foo", but destined eventually to become the "main" program in the "Foo" port of Squeak;
- `sqConfig.h` — a (slightly) modified copy of the original, with an additional section that recognizes the host platform (see Section 42.6);
- `sqPlatformSpecific.h` — a (possibly modified) copy of the original, with any modifications to the compilation environment that might be required (see Section 42.6);
- ...plus any necessary `C` libraries.

The interpreter code relies only on the following functions from libraries:

- math: `exp()`, `log()`, `atan()`, `sin()`, `sqrt()`, `ldexp()`, `frexp()`, `modf()`;

- standard input/output: `getchar()`, `putchar()`, `printf()`;[3]
- others: `memcpy()`, `strlen()`, `clock()`.

Given the above, it should be possible to compile and link a Squeak virtual machine. Running this VM should cause an almost immediate exit, after printing the name of the first "essential" support function that is missing. Porting then becomes an iterative process:

- implement the missing "essential" function that caused the exit;
- compile, link, run;
- repeat.

If this methodology is followed, the order in which the various essential functions are implemented should be approximately as follows. (Don't worry if some of the comments below seem to make little sense right now. When the time comes to implement each particular function, the comments will start to make perfect sense.)

Reading an image file

`sqImageFileRead()` and friends (Section 42.6.2).

Squeak can do nothing without an image file to run, and so the very first thing to get working is the image loading code.

The virtual machine keeps track of the full path name of the Squeak image file and the path to the directory containing the virtual machine. In a minimal implementation, the VM path can be the empty string and the image name hardwired to `squeak.image`. It is assumed that the image file, the changes file and the system sources file are all in the the same directory, and that this directory is the default working directory for any file operations.

The rest of the file system implementation can be left until later.

[3] On platforms that have no terminal input/output, even the standard I/O functions could be disabled. They are used only to report fatal errors from within the generated code. (Hopefully there will still be *some* way for the code to indicate which function caused an exit during the initial "demand driven" development.)

Displaying bits on screen

`ioShowDisplay()` and friends (Section 42.7).

Once the image loading code is working it's time to let Squeak display something. The critical function here is `ioShowDisplay()`, and it's likely that Squeak is now reaching the "stub" that was left in place of this function.

Once this function is working, the latest port of Squeak should actually be displaying something meaningful on the screen.[4] The most likely thing that Squeak will display is a messages saying that it can't find the changes file. (Which is already not bad, for two relatively simple steps!)

Now is the time to implement a few boring (but critical) user-interface related functions.

For graphical output: `ioShowDisplay()` is already done, but `ioScreenSize()` is also very usful (and easy to implement). Things like `ioHasDisplayDepth()` can be hard-wired to `1` (after making sure that `squeak.image` is using a depth with which `ioShowDisplay()` can cope.)

The following functions can be made no-ops at this stage of the port:

`ioProcessEvents()`,
`ioSetCursor()`, and
`ioSetCursorWithMask()`.

Handling time

`ioMSecs()` and friends (Section 42.11).

Squeak relies heavily on knowing how to tell the time, and in particular on knowing how much time has elapsed since a given event. This step is critical for many ports: without the timer it is impossible to proceed, since so much of the user interface code relies on it.

The following functions are essential for timekeeping: `ioMSecs()`, and `ioMicroMSecs()`.

The function `ioSeconds()` can initially be hard-wired to always return 0, with the effect that the current date and time will be wrong. On the other hand, the user interface never needs to know the "wall clock" time, and so this will not prevent significant further progress.

[4] Keep a bottle of champagne handy for this moment: the feeling of achievement that comes with it should not be underestimated.

Reading the keyboard and mouse

`ioGetKeystroke()` and friends (Section 42.8).

The following are essential for interacting with Squeak:

`ioGetKeystroke()`,
`ioGetButtonState()`, and
`ioPeekKeystroke()`.

The function `ioMousePoint()` is also needed to track the pointer position.

If the harware is normally polled to retrieve mouse/keyboard input then `ioProcessEvents()` can probably be made a no-op. Otherwise it might be necessary to implement it, in order to help with the conversion of mouse or keyboard events to a form that Squeak can poll (see Section 42.8.1).

The following functions can be made no-ops at this stage of the port:

`ioSetCursor()`,
`ioSetCursorWithMask()`, and
`ioBeep()`.

And the rest...

The new port of Squeak is now basically working!

It should be possible to interact with menus, browse the class hierarchy, open a Workspace and evaluate expressions in it, and even run some simple benchmarks. Reaching this point is a second good excuse for celebrating.[5]

Now that Squeak can interact, a potentially useful thing to do is to open the "preferences" menu and disable the logging of errors to a file in the current directory—just until the port has a working, writable file system. (Otherwise the first error of any kind will put the virtual machine into an infinite "fatal error" recursion.)

[5]As an indication of how quickly it is possible to arrive at this point, the first port of Squeak (to Unix/X11) was begun late on a Friday evening. After about two days of hacking, sometime during Sunday afternoon, Unix Squeak could be used to browse the class hierarchy and evaluate Smalltalk expressions in a Workspace. Later that same day it successfully saved a renamed `.image` file (copying the `.changes` file in the process) and then started up again using the newly-saved image. It was only a matter of a few days more before the first "release" of Unix Squeak was publicly available.

Note that it is possible to delay implementing the file system for quite a long time, and even then it can initially be made read-only.[6]

One *very* useful thing to get working early in the porting process is the interrupt key (or button). This is especially handy for ports to machines that have no keyboard (such as PDAs), to "escape" from prompts asking for keyboard input.

The rest of the porting process is just a matter of prioritizing which things to implement first. The functions that were mentioned above but left as "no-ops" would be a good starting point, followed by individual "subsystems" as described in the remainder of this chapter.

The preceding discussion assumes that the port is being made "in a vacuum". On the other hand, it is possible that an existing port offers a significant, reusable code base on which to build the new port.

42.4.2 Stealing code from other ports

By far the easiest way to get started with a new port is to take an existing port and modify it for the new host. In some cases a significant amount of work can be avoided by doing this. A good example is the code for updating the physical display.

The Unix/X11 port (at least) contains code to convert 8-, 16- and 32-bit deep internal Display formats into 8-, 16-, 24- or 32-bit deep physical screen data, with or without byte order reversal. It also contains a reusable skeleton for reconciling an entirely event-driven graphical, keyboard and mouse I/O model with Squeak's polled model. The network subsystem should also work with little or no modification on any host that has a BSD-compatible socket library.

Obviously, other ports might offer a more sensible "starting point", depending on the target platform. Certain subsystems (such as sound) are so dependent on the host that they will probably have to be rewritten from scratch in most cases.

An intermediate case concerns hosts that have significant characteristics in common with an an existing port, but which have sufficient differences

[6]Ports to "bare hardware" might also take advantage of the fact that most Flash and CompactFlash cards can be formatted as a FAT-16 MS-DOS file system, which has a published and very simple specification. Any necessary image, changes and sources files can be transferred easily to these cards on a workstation equipped with an appropriate adaptor.

to warrant an independent existence. In such cases a serious disadvantage of "forking" a new port is that two sets of essentially identical code must be maintained. Ideally the code for the new host would be integrated with the existing port, although this also has a disadvantage: it can only be accomplished with the complete cooperation of the maintainer of the existing port (which might have to be reorganized to isolate the incompatible functionality). This situation is probably rare, but examples of both approaches already exist.[7]

42.5 Squeak's C conventions

This section describes several conventions that are used universally by generated code, and to which support code must adhere in order to work correctly.

42.5.1 Squeak does not believe in pointers

Squeak's implementation treats almost everything as an `int`. In particular, pointers that are passed between Squeak and the support code always have type `int`. It is up to the support code to perform any casting that might be required.

42.5.2 C strings vs. Squeak strings

Squeak and C store strings in fundamentally different ways. In C a string is always terminated with a "null" character (ASCII value 0). Squeak, on the other hand, stores the length of the string and dispenses with any kind of terminating character. This difference is important whenever string data is transferred between Squeak and C. In most cases such transfers of string data follow the same pattern.

To export a string from Squeak to C, the support function is called with two arguments: a pointer to the string data and the number of bytes in the string. The support function simply copies the given number of bytes (using `memcpy()` for example) from the given address into its own memory. (If

[7] The Unix version of Squeak was the basis for a significant portion of the (entirely distinct) OS/2 port. The majority of the Unix code is also reused without modification in the Mac OS X version of Squeak. (Mac OS X is essentially BSD Unix, but with a graphics server that is incompatible with X11.)

allocating memory dynamically then it should always add one to the length indicated by Squeak, and append a terminating "null" to the copy.)

Importing a string is slightly more complex. In general Squeak will call two support routines. The first should return the length of the string to be imported from C into Squeak. (This allows Squeak to allocate a new String of the appropriate size, or to grow a buffer if necessary, or whatever.) The second routine is called with a pointer to the destination, and the actual number of bytes that Squeak expects to be copied. (`strncpy()` is the safest way to actually transfer the bytes into Squeak's memory, to avoid any possibility of trying to copy too many bytes from the point of view of both Squeak and C.)

The clipboard handling rountines (Section 42.9) illustrate perfectly the way Squeak and C exchange string data.

42.5.3 Interacting with Semaphores

Several support routines are required to report asynchronous events to the Squeak interpreter. One example is the networking subsystem, where the completion of a `write()` operation or the availablility of data for a `read()` operation must be communicated to the interpreter. This is accomplished by signalling a Semaphore.

Semaphores are identified (to the support code) by an integer index. Signalling a Semaphore is accomplished by calling the (generated) function

```
signalSemaphoreWithIndex(int semaIndex)
```

passing the appropriate Semaphore index as the argument.

42.5.4 Primitive success and failure

Some of the support functions are associated directly with a primitive method in the Squeak image. Such functions must indicate whether the primitive operation succeeds or fails. Two generated functions are provided to do this. The first is

```
void primitiveFail(void)
```

which is called to "fail" the primitive. (It does not transfer control back to the Squeak interpreter: the support code must ensure that a `return` is executed at the appropriate moment after failing a primitive.)

The second function is

```
int success(int successFlag)
```

which can be called several times from within a primitive support function. The argument should be either `true` or `false`. This function "composes" successive values of `successFlag`; that is, if a primitive support routine calls this function with `false` as the argument then Squeak will consider the primitive operation to have failed, regardless of how many times (or when) the support function calls it with the argument `true`.

The following sections will indicate when a support function is associated directly with a Squeak primitive. Such functions should "fail" (as described above) whenever they cannot complete an operation successfully.

42.6 Compilation environment: `sq.h`

The generated code does not exist in a vacuum—it requires some kind of compilation environment to give it access to a few basic system services on the host platform. Porting Squeak therefore also involves defining an appropriate compilation environment for the generated code. This must be done before (or during) the initial attempt to compile and link the first "entirely unimplemented virtual machine" (as described in Section 42.4).

Each of the automatically-generated, platform-independent source files begins by including `sq.h`, which establishes a compilation environment for the source file. This header file is also typically included by the (hand-written) platform-dependent source files, since it declares many useful function prototypes—including those for all the functions that should be present in the support code. (Including it routinely in every source file therefore helps to detect errors due to incorrect function signatures.) The overall structure of `sq.h` is shown in Figure 42.2.

Generated code makes use of several ANSI and/or POSIX routines that should be available on most platforms that have a Standard `C` compiler. Since the names of the necessary header files have been standardized, `sq.h` assumes that they are available and `#include`s the following directly: `math.h`, `stdio.h`, `stdlib.h`, `string.h` and `time.h`.

Unfortunately, the generated code also uses facilities that may or may not be present—or that might be present in different forms depending on the platform. To cope with this, `sq.h` `#include`s two files which will certainly require modification for a new platform.

ANSI/POSIX headers

```
   #include <math.h>                          these are assumed to exist on all platforms
   #include <stdio.h>
   #include <stdlib.h>
   #include <string.h>
   #include <time.h>

▷  #include "sqConfig.h"                      identifies the host platform
```

defaults for platform-specific definitions

```
   #define true  1                            symbolic constants used by generated code
   #define false 0
   #define null  0

   #define EXPORT(type)...                    return type declaration for functions in dynamic
                                              libraries
   #define sqImageFile...                     ANSI/POSIX file types and functions for ac-
   #define sqImageFileOpen...                 cessing file streams
      ...

   #define sqAllocateMemory...                interface to memory allocation routines
   #define reserveExtraCHeapBytes...

   #define storeFloatAtfrom...                accessing 64-bit IEEE doubles
   #define fetchFloatAtinto...

   int ioMSecs(void);                         prototypes for fetching millisecond time
   int ioLowResMSecs(void);
   int ioMicroMSecs(void);

   #define ioMSecs()...                       defaults based on ANSI clock() function
   #define ioLowResMSecs()...
   #define ioMicroMSecs()...

▷  #include "sqPlatformSpecific.h"            redefines zero or more of the above defaults for
                                              a particular host platform
```

prototypes for all support functions

```
   /* file i/o */                             a list of prototypes which declares the set of
   /* directories */                          support functions that must be provided by the
      ...                                     platform-dependent support code for each plat-
                                              form on which Squeak runs
```

variables imported from generated code

```
   extern const char *interpreterVersion;     the version string of interp.c
```

Figure 42.2: The structure of the file `sq.h`. The two header files marked with '▷' must be modified when porting to a new platform. The file `sqConfig.h` can define symbols to change the way the two Float access macros are defined. The file `sqPlatformSpecific.h` should redefine any macros that did not receive suitable defaults in `sq.h`. See the text for further details.

The first of these is `sqConfig.h`, which is responsible for identifying the host platform. A new port will have to add a corresponding section to `sqConfig.h` (cut-and-paste from an existing section with minimal modifications will probably do the trick). The new section must define at least the symbol `SQ_CONFIG_DONE`, to indicate that the platform has been recognized.[8]

`sq.h` goes on to define various "sensible" defaults for things that the generated code will need, before including the second of these files: `sqPlatformSpecific.h`. As its name suggests, this file is responsible for providing the generated code with access to basic facilities that differ between platforms. On ANSI/POSIX platforms it will have almost nothing to do. On more exotic platforms it will have to "undo" some of the assumptions made in `sq.h`. It does this by selectively redefining the macros previously set up by `sq.h`, in a section of code compiled conditionally according to the host platform (as detected previously in `sqConfig.h`). If any system header files (other than those already included by `sq.h`) are required by the host, then `sqPlatformSpecific.h` is the place in which to `#include` them.

The macros that `sqPlatformSpecific.h` should consider redefining are concerned mainly with declaring functions for dynamically-loaded libraries, file access, memory allocation, and keeping track of elapsed time. They are described in the following four sections. Their default "reasonable" ANSI/POSIX definitions (provided by `sq.h`) are shown in Table 42.1.

42.6.1 Declaring functions for dynamic libraries

Some of the generated code is intended to be "pluggable"—compiled separately from the main virtual machine as a dynamically-loadable library, to be read into the virtual machine "on-demand" at runtime when first needed. Unfortunately, some compilers and hosts require special declarations for functions that are to be exported from a dynamic library. The macro `EXPORT(type)` is therefore used to declare the return type of any function that might be placed in a dynamic library; for example:

```
EXPORT(int) someDynamicallyLoadedFunction(void) { ... }
```

`sqPlatformSpecific.h` can redefine this macro to provide any additional declaration keywords that might be needed (not forgetting, of course, to

[8] Some platforms do not have a single, unique predefined preprocessor symbol to aid with their identification. Any disambiguation should be done in `sqConfig.h` and a unique, unambiguous identifying symbol `#defined` for use later on in `sqPlatformSpecific.h`.

symbol/macro	*default (ANSI/POSIX) definition*
`EXPORT(type)`	`type`
`sqImageFile`	`FILE *`
`sqImageFileOpen(name, mode)`	`fopen(name, mode)`
`sqImageFileRead(ptr, sz, count, f)`	`fread(ptr, sz, count, f)`
`sqImageFileWrite(ptr, sz, count, f)`	`fwrite(ptr, sz, count, f)`
`sqImageFilePosition(f)`	`ftell(f)`
`sqImageFileSeek(f, pos)`	`fseek(f, pos, SEEK_SET)`
`sqImageFileClose(f)`	`fclose(f)`
`reserveExtraCHeapBytes(size, extra)`	`size`
`sqAllocateMemory(min, desired)`	`malloc(desired)`
`ioMSecs()`	`((1000 * clock()) / CLOCKS_PER_SEC)`
`ioLowResMSecs()`	`((1000 * clock()) / CLOCKS_PER_SEC)`
`ioMicroMSecs()`	`((1000 * clock()) / CLOCKS_PER_SEC)`

Table 42.1: Default ANSI/POSIX values for the symbols and macros that `sqPlatformSpecific.h` might want to consider redefining.

include the return type of the function).

42.6.2 Reading and writing the image file

Most of the code needed for loading and saving images is generated automatically. This code assumes an ANSI-like interface to the filesystem. The symbol `sqImageFile` should be defined as the type of a "file handle" on the host platform.

`sqImageFileOpen()` is passed the `name` (a `C`-style null-terminated string) and `mode` (also a `C` string) of a file, and should return its handle (of type `sqImageFile`, or `null` if the file does not exist). The `mode` is as specified by ANSI; either `"rb"` (read binary bytes) or `"wb"` (create or truncate and then write binary bytes).[9]

The reading/writing macros are passed a pointer to an area of memory (`ptr`), a file handle (`f`, obtained from `sqImageFileOpen()`) and the number of bytes to transfer expressed as `count` "elements" of size `sz`. These macros should simply transfer `count * sz` "uninterpreted" bytes. (Any "endian" conversions that might be necessary are handled automatically elsewhere).

[9] "Binary bytes" implies that no CR/LF line-end conversion should be attempted when reading/writing the image file.

Finally, `sqImageFilePosition()` and `sqImageFileSeek()` are responsible for retrieving and setting the "file pointer", i.e. the offset (from the beginning of the file) at which the next read/write operation should commence. (The generated code tacitly assumes that read/write operations will increment the file pointer by the number of bytes read or written.)

42.6.3 Allocating memory

The generated code needs to know approximately how much space to reserve for the virtual machine's data. This space includes the Smalltalk "heap" (the in-memory copy of the image file) and any additional data space that might be required by dynamically-loaded libraries. The macro `reserveExtraCHeapBytes(size, extra)` is used to calculate how much data space should be reserved when the VM starts up. The first argument is the number of bytes required for the image; the second is an estimate of how much additional data space might be needed by dynamic libraries. This macro should return the total amount of data memory that the VM should "reserve" statically. If the host knows how to dynamically allocate more data memory as libraries are loaded (or if it doesn't support dynamic libraries at all) then the correct result is simply the size of the image file (the default definition).

The macro `sqAllocateMemory(min, desired)` is used to allocate the memory for the image (and possibly for the dynamic libraries). The first argument is the minimum acceptable size of memory (measured in bytes), and the second is the "ideal" size. This memory allocation macro should return `null` if it cannot allocate at least `min` bytes of memory.

42.6.4 Keeping track of elapsed time

`sq.h` also declares three functions that return the elapsed time (relative to any convenient point of reference):

```
int ioMSecs(void)
int ioLowResMSecs(void)
int ioMicroMSecs(void)
```

(The reason for having three functions to read the time will be explained later, in Section 42.11). `sq.h` then immediately "hides" these declarations with three macro definitions of the same names that use the ANSI `clock()` function to calculate the time.

sqPlatformSpecific.h should seriously consider redefining these macros (or simply undefining them so that real functions can be called in the support code) to improve the accuracy of timing within Squeak. The problem is that clock() usually measures elapsed CPU time rather than elapsed wall-clock time. Consequently, whenever Squeak goes to sleep (which it does whenever it runs out of interesting things to do) time will effectively stop passing if the host adheres to the ANSI definition of clock().

The remainder of sq.h provides a full set of prototypes for the functions that should be implemented by the platform support code. Since these declarations should never depend on the host platform (and should therefore be identical across all platforms) they appear after the inclusion of sqPlatformSpecific.h.

42.6.5 Reading and writing Floats

Two macros are defined by sq.h to copy 64-bit, double-precision, IEEE floating-point values between C doubles and the data portion of a Float object. By default these macros "do the right thing" on big-endian architectures, where the 64 bits of a double are stored most significant byte first in memory: the data is transferred in the obvious way, by dereferencing a pointer to double.

Two complications might arise with this. The first is that Squeak aligns all data on 32-bit boundaries, *including* the 64 bits of data in a Float. If the host imposes 64-bit alignment on doubles then the symbol DOUBLE_WORD_ALIGNMENT can be defined in sqConfig.h to force these macros to use two 32-bit transfers to move the data. The second complication arises on little-endian machines, where the least significant word is stored first. For these hosts, sqConfig.h should define DOUBLE_WORD_ORDER to cause the float macros to swap the two 32-bit halves of a double while copying it.

If some other scheme is necessary to copy doubles between C variables and Squeak's object memory then sqPlatformSpecific.h will have to redefine the macros

```
storeFloatAtfrom(i, floatVarName)
fetchFloatAtinto(i, floatVarName)
```

where i is an int *expression* giving the address of the data in a Float object, and floatVarName is the name of a *variable* of type double (whose address can be taken using the & operator to effect the transfer).

The two 32-bit halves of a Float's data are automatically byte-swapped to match the local host when the image is loaded. The Float macros therefore need not take byte order into account.

42.7 Graphical output

Just like the very first releases of Smalltalk-80 in the mid-1980s, Squeak performs all of its graphical output directly to an object inside the image. This object (called "Display") includes a Bitmap representing what the user should see. The task of rendering Squeak's graphical display is therefore relatively easy. Rather than having to implement a host of different graphical drawing operations directly, the support code simply copies bits out of the Display object and onto the screen at the appropriate moments. The precise nature of "the screen" depends on the platform, and might be a window on Mac OS or Unix (which uses the X Window System), or directly to a memory-mapped framebuffer device, or even to a tiny LCD panel—which is the case for at least one port of Squeak to "bare hardware".

The "appropriate moments" are determined entirely within Smalltalk, and ultimately result in the calling of the support function `ioShowDisplay()`. This function performs the required copying of bits onto the physical display, according to a "damage rectangle" that is supplied as an argument to the function.

42.7.1 Updating the display

The support function

```
int ioShowDisplay
  (int dispBitsIndex, int width, int height, int depth,
   int affectedL, int affectedR, int affectedT, int affectedB)
```

is responsible for updating the physical screen, based on a Bitmap representing the display within Squeak.

The first four arguments provide information about the Bitmap data to be transferred to the physical display:

- `dispBitsIndex` is the address of the first byte of the data portion of a Bitmap object in Squeak's memory. This address corresponds to the first pixel (top-left corner) of the display.

- `width` is the width of the bitmap's data. The "pitch" of the bitmap (the number of pixels in each scanline) is always rounded up so that each scanline is word-aligned (i.e. it is a multiple of 4 bytes wide).

- `height` is the total number of scanlines in the bitmap data.

- `depth` is the number of bits in a pixel. Currently depths of 1-, 2-, 4-, 8-, 16- and 32-bits per pixel are supported.

The final four arguments `affectedL`, `affectedR`, `affectedT`, and `affectedB` specify a "damage rectangle". They correspond to the left, right, top and bottom limits (respectively) of the portion of the display that should be updated.[10]

Bitmaps are word objects, and their byte order is swapped automatically if necessary (by generated code) when the image is loaded. No special action is needed if the host's byte order matches the physical display's byte order.

42.7.2 Display depths and the colormap

In 32-bits per pixel depth, Squeak really uses 24-bit pixels. The blue component is in the least significant 8 bits of each pixel, followed by 8 bits of green and then 8 bits of red. The most significant 8 bits are unused.

In 16-bits per pixel depth, Squeak really uses 15-bit pixels. The blue component is in the least significant 5 bits of each pixel, followed by 5 bits of green and 5 bits of red. The most significant bit is unused.

In 1- through 8-bits per pixel depths, Squeak uses the colormap shown in Table 42.2.

42.7.3 Other display functions

```
int ioScreenSize(void)
```

should return the current size of the screen, with the width in the most significant 16 bits and the height in the least significant 16 bits.

[10] An initial implementation of `ioShowDisplay()` could ignore the damage rectangle and simply update the entire screen area according to the bitmap. Although slow, this would avoid any possibility of the graphical output appearing to be broken due to misinterpretation of the damage rectangle (yielding an "update area" of zero size) when it is otherwise working perfectly.

pixel	*color*
0	white (or transparent if bpp > 1)
1	black
2	white (opaque)
3	50% gray
4	red
5	green
6	blue
7	cyan
8	yellow
9	magenta
10	1/8 gray
11	2/8 gray
12	3/8 gray
13	5/8 gray
14	6/8 gray
15	7/8 gray
16–39	1/32–31/32 gray (omitting $n/8$)
36r + 6b + g + 40	$6 \times 6 \times 6$ color cube

Table 42.2: Squeak's color map for 1-, 2-, 4- and 8-bit depths. The *pixel* column refers to the pixel values stored in the Display Bitmap. The *color* column specifies the corresponding colors to be rendered on the physical display. For display depths of less than 8 bits only an initial portion of this table will apply. For example, for depth 2 only the first four lines are relevant (pixel values `0` through `3`). For depth 8, the maximum pixel value (according to the table) corresponds to the final entry with $r = 5$, $g = 5$ and $b = 5$; i.e. $36 \times 5 + 6 \times 5 + 5 + 40$, which is (rather fortunately) equal to 255. For depths of greater than 8 bits Squeak does not use a color map; the bits in the pixel specify the red, green and blue intensities of the color directly. See the text for further details.

```
int ioHasDisplayDepth(int depth)
```

should return `1` if the host supports a Squeak Display of the given `depth`. (This function is used to avoid passing an unsupported depth to `ioShowDisplay()`.)

```
int ioSetFullScreen(int fullScreenFlag)
```

is used to turn "full screen" display on and off. If `fullScreenFlag` is `1` then the function should save the current screen size before resizing the display to occupy the entire screen, removing any window decorations if they are present. (The intention is that Squeak "take over" the entire physical display area.) If `fullScreenFlag` is `0` then the function should restore the physical screen (and any window decorations that might be present by default) to its saved original size.

```
int ioSetDisplayMode(int width, int height, int depth,
                     int fullscreenFlag)
```

is called before Squeak tries to change its Display characteristics. The arguments have the usual meanings. This function should return `1` to accept the new Display parameters, or `0` to reject them.

```
int ioForceDisplayUpdate(void)
```

is called from generated code whenever Squeak wants to be certain that its internal Display and the physical display are "synchronized". If the display is "local" (a framebuffer connected directly to the host) then nothing special need be done. If the display is "remote" (a network window system, for example) then this function should not return until it is certain that any pending display operations (initiated from `ioShowDisplay()`) have been completed.

```
int ioSetCursor(int cursorBits, int offsetX, int offsetY)
```

`cursorBits` is the address of a cursor bitmap. The bitmap is 16 bits wide and 16 bits high. The 16 bits of each "scanline" appear in the most significant 16 bits of a 32-bit word (the least significant 16 bits are unused). (Successive "scanlines" are therefore in the most significant halves of consecutive 32-bit words starting at `cursorBits`.) The host's cursor should be changed to reflect the bitmap, with a `1` in `cursorBitmap` being a black pixel in the cursor, and a `0` being transparent (the background shows through the cursor). The "hot spot" of the cursor is given by the second and third argument, which are measured from the top-left of the cursor (0, 0) and then *negated*.[11]

[11] A hotspot in the top-left corner of the cursor is at offset $(0, 0)$. A hotspot in the bottom-right corner is at offset $(-15, -15)$. (This, and similar, weirdness comes from Squeak's origins as a Macintosh application.)

```
int ioSetCursorWithMask(int cursorBits, int cursorMask,
        int offsetX, int offsetY)
```

is similar to `ioSetCursor()` except that `cursorMask` points to a bitmap (in the same format as `cursorBits`) specifying where the `0` pixels in `cursorBits` should be opaque. Wherever `cursorMask` contains a `1` and `cursorBits` contains a `0`, the cursor should have an opaque white pixel (obscuring the background) instead of the normal transparent pixel.

42.8 Mouse and keyboard input

The interpreter reads keyboard and mouse information with the help of four support functions. The simplest of these is

```
int ioMousePoint(void)
```

which should return an `int` representing the current position of the mouse pointer. The top 16 bits contain the x coordinate and the bottom 16 bits the y coordinate. The origin is the top-left corner of the window (or screen, if Squeak is using a raw framebuffer), with x increasing towards the right and y towards the bottom of the window.

The remaining three functions read keyboard input and the state of the "modifier" keys.[12]

```
int ioGetKeystroke(void)
```

reads (and returns) the next character in the keyboard input buffer, removing it from the buffer in the process. The result is a 12-bit integer, in which the least significant 8 bits contain the ASCII value of the character and the next four bits contain the "modifier" keys that were pressed at the time the keystroke was recorded. The bit assignments are shown in Table 42.3. A non-destructive read must also be provided, by the function

```
int ioPeekKeystroke(void)
```

[12]On the Macintosh these are "control", "shift", "option" and "command". On other platforms there are often "meta" and/or "alt" keys that can take the place of either "option" or "command". Other combinations, such as "shift"+"control" can be used if necessary to emulate "command" and/or "option"; the support code should implement whatever mapping seems appropriate or most natural for users accustomed to the platform.

bit	*meaning*
11	command
10	option
9	control
8	shift
0–7	ASCII code

Table 42.3: Value returned by `ioGetKeystroke()` and `ioPeekKeystroke()`. The low 8 bits contain the ASCII code. The next four bits are set to `1` if the corresponding modifier key was pressed when the keystroke was recorded.

bit	*meaning*
6	command
5	option
3	control
3	shift
2	left mouse button
1	middle mouse button
0	right mouse button

Table 42.4: Value returned by `ioGetButtonState()`. The low 3 bits indicate which mouse buttons are pressed. The next fours bits are set to `1` if the corresponding modifier key was pressed when the mouse button state was recorded. On systems having a single-button mouse, it should be treated as the left button. The left button should also obey the modifier keys, with "control" transforming it into the middle button and "meta" (or equivalent) transforming it into the right button.

The keyboard handling code should also check for a key code (ASCII character plus the modifier bits) equal to the contents of the variable `interruptKeycode` (declared and defined by generated code). If this key combination (usually "command" plus ".") is pressed then the support code should set the variable `interruptPending` to `true`, and `interruptCheckCounter` to `0` (both variables are declared in generated code). This will cause Squeak to abort its current activity, returning control to the user interface.

The mouse buttons are read by the function

```
int ioGetButtonState(void)
```

whose result is a 7-bit integer containing three mouse button flags and the four modifier key bits. The bit assignments are shown in Table 42.4.

42.8.1 Reconciling polling with event-driven input

Unlike most window systems and graphical toolkits (which tend to be event-driven), Squeak "polls" for incoming data from the keyboard, mouse and other sources. This polling normally occurs whenever the Smalltalk user interface reads the mouse or keyboard state.

Even on systems such as X (which buffer incoming events on behalf of the application) there is a conflict of interests. For example, Squeak expects to be able to read the current position of the mouse at any moment, regardless of how many keyboard events might be waiting in the buffer. This means that the support code must "service" events as soon as possible after they arrive (to keep the mouse position up to date, and to check for the "interrupt" key in a timely fashion), while providing some mechanism for "saving up" keyboard events to be delivered at some later time, when Squeak decides to poll for them.

To help reconcile polling with a possibly (or even probably) event-driven platform, the interpreter calls the support function `ioProcessEvents()` before reading the mouse or keyboard state during interactive operation (and approximately two times per second when running a CPU-bound activity, to give the support code chance to set the `interruptPending` flag if necessary).

`ioProcessEvents()` typically has four responsibilities, as follows:

- tracking the current position of the mouse based on any "motion" events that might have arrived;
- reading and recording any "keypress" and "buttonpress" events that might have arrived;
- recording the current state of the "modifier" keys along with button and keypress events; and
- setting the `interruptPending` flag to `true` if the `interruptKeycode` combination has been pressed.

```
int ioProcessEvents(void)
{
  while (/* input event available */)
    {
      event = /* next event */;
      switch (event.type)
        {
          case /* mouse motion */:
            mousePosition.x = event.x;
            mousePosition.y = event.y;
            break;
          case /* keypress */:
            recordKeystroke(event.keycode);    /* the character itself */
            recordModifiers(event.modifiers);  /* shift, control, alt */
            break;
          case /* window expose */:
            fullDisplayUpdate();
            break;
        }
    }
  return 0;
}
```

Figure 42.3: Typical definition of `ioProcessEvents()`.

Depending on the precise details of the platform, `ioProcessEvents()` might also be a good place in which to check for other sources of input/output activity (network and sound, for example).

Figure 42.3 shows a "skeleton" for a typical implementation of `ioProcessEvents()`.

If such a scheme is used to match events with Squeak's polling then the check for the `interruptKeycode` (described in the previous section) should be performed in the event handler, to ensure that user interrupts are caught at the earliest possible moment.[13]

[13]Every platform should try hard to decouple the test for the `interruptKeycode` from the reading of the keyboard via `ioGetKeystroke()`. If Squeak is stuck in an infinite loop, for example, then it is unlikely to ever call `ioGetKeystroke()` again—and Squeak would "freeze", with no possiblity of interruption.

42.8.2 Event-driven keyboard/mouse input

Starting with version 2.9 of Squeak there is experimental support for true event-driven input. If the image supports event-driven input then it will call the support function

```
int ioSetInputSemaphore(int inputSemaIndex)
```

once when starting up. The `inputSemaIndex` specifies the index of a Semaphore to be signalled (Section 42.5.3) whenever an input event becomes available. If this function is not called during startup then the support code should continue to provide "polled" input handling as described above, to remain compatible with older images.[14]

If the above function is called from generated code during startup then the support code should arrange for the input Semaphore to be signalled whenever an event arrives. This will cause the interpreter to call the support function

```
ioGetNextEvent(sqInputEvent *evt)
```

shortly afterwards. `evt` is a pointer to an `sqEvent` structure that should be filled in appropriately. The event structures (defined in `sq.h`) are shown in Figure 42.4.

The `type` field should be set to one of the following values (defined symbolically in `sq.h`):

`EventTypeMouse`	for mouse events
`EventTypeKeyboard`	for keyboard events

The `timeStamp` field should be the value of `ioMSecs()` at the time the event arrived.

For mouse events, `x` and `y` give the position of the mouse (relative to the top-left corner of the Squeak window). The `buttons` field details which button caused the event, according to the following constants defined in `sq.h`:[15]

[14] Backwards compatibility should not be a priority in an initial port of Squeak. The vast majority of Squeak users upgrade to the latest version of the system the instant it becomes available.

[15] The rather colorful names are traditional, and come from the colors of the mouse buttons found on the first machines on which Smalltalk ran in the 1970s.

```
typedef struct sqMouseEvent {
  int type;               /* EventType value */
  unsigned int timeStamp; /* time of arrival */
  int x;                  /* mouse X position */
  int y;                  /* mouse Y position */
  int buttons;            /* 'or'ed button bits */
  int modifiers;          /* 'or'ed modifier values */
  int reserved1;          /* reserved for future use */
  int reserved2;          /* reserved for future use */
} sqMouseEvent;

typedef struct sqKeyboardEvent {
  int type;               /* EventType value */
  unsigned int timeStamp; /* time of arrival */
  int charCode;           /* character code (see text) */
  int pressCode;          /* EventKey value */
  int modifiers;          /* 'or'ed modifier bits */
  int reserved1;          /* reserved for future use */
  int reserved2;          /* reserved for future use */
  int reserved3;          /* reserved for future use */
} sqKeyboardEvent;
```

Figure 42.4: Squeak mouse and keyboard event structures. Note that the common field `modifiers` is *not* in the same location in the two structures.

`RedButtonBit`	the left mouse button
`BlueButtonBit`	the middle mouse button
`YellowButtonBit`	the right mouse button

For keyboard events, the `charCode` field contains the character code of the key that was pressed.[16]

The `pressCode` field identifies the physical action that is being reported for the key, according to the following symbolic constants defined in `sq.h`:

[16]The details of event-driven input are still being debated at the time of writing. No final decision has been made about the way the keyboard characters should be encoded, although the tendancy seems to be towards using the 16-bit `keysyms` defined by the X Consortium for use in the X Window System. Conversion to these from an 8-bit ASCII code is trivial (using a lookup table), and avoids discarding potentially interesting information encoded in the `keysyms` on X-based (and similar) systems.

`EventKeyDown`	the key was pressed
`EventKeyUp`	the key was released

The final `modifiers` field in both mouse and keyboard events reflects the state of the "shift", "control", "alt" and any other kinds of "meta" key that might be present on the keyboard. If a given modifier key is down when an event arrives, the corresponding bit should be set in the event reported to Squeak. As before, from `sq.h`:

`ShiftKeyBit`	obvious
`CtrlKeyBit`	obvious
`CommandKeyBit`	"alt" or "meta" key
`OptionKeyBit`	"ctrl" + "command" if no distinct key available

42.9 The clipboard

Squeak's editing facilities include the usual "cut", "copy" and "paste" operations. In addition to working with text inside the image, they can be used to exhange data with other applications. To this effect, Squeak expects the support code to maintain a "clipboard" holding the text associated with these operations.

The clipboard is the destination for "cut" and "copy" operations. When one of these operations is performed by the user, the interpeter calls the support function

```
int clipboardWriteFromAt(int count, int base, int offset)
```

which should copy `count` bytes of text from the address `base + offset` into some suitably-allocated external storage.[17] The text is *not* terminated with a "NUL" character. The return value of this function is ignored.

The clipboard is the source for the "paste" operation. The interpreter first calls the support function

```
int clipboardSize(void)
```

which should return the number of bytes of text currently stored in the clipboard. The function

[17] On platforms that have the standard `C` library, such storage could by allocated by calling `malloc()`, for example.

```
int clipboardReadIntoAt(int count, int base, int offset)
```

is then called to transfer `count` bytes of data from the clipboard to the address `base + offset`. This function must not store more than `count` bytes, should not attempt to terminate the stored text with a "NUL" character, and should return the number of bytes actually transferred.

The addresses `base + offset` actually points into the middle of a Smalltalk String, and so any text read or written by these functions should use the Smalltalk line-end convention: a single "CR" character, ASCII value 13.

If the local platform supports copy-and-paste between applications then the clipboard is the place where such exchange of data will take place. If the platform's line-end convention is not the same as Smalltalk's then the support code will have to take care of any required conversion when exporting or importing the clipboard to or from other applications.

42.10 Files and directories

All of Squeak's file primitives are implemented by generated code, which assumes the existence of the ANSI `stdio` functions. Operations on directories are more complicated, and a certain amount of support code is necessary. Porting to a new platform will require the following support functions to be implemented. Unless otherwise indicated, these functions should return `1` to indicate success and `0` to indicate failure.

```
int dir_Delimitor(void)
```

should return the ASCII value of the character used to delimit directories in a pathname.[18]

[18]The absence of this function in the very first port of Squeak caused a certain amount of "entertainment". At the time, the image "remembered" the full paths to its `.changes` and `.sources` files. Since these were originally on a Macintosh file system, the directory delimiter in these paths was a colon ':'. It was necessary to make symbolic links (with ridiculously long names) to these files before Squeak would start up correctly. The next step was to change the delimiter to be correct for Unix (a slash '/'), which had the unfortunate side-effect that Squeak began looking for these files in directories that simply did not exist on a Unix system. A painful series of symbolic links (starting at the root of the filesystem) was needed before Squeak could successfully find the files—at which point the image could be saved from within Squeak, causing it to "remember" a much more "reasonable" set of paths to these files. More than four years after the initial port

```
int dir_Create(char *pathString, int pathStringLength)
```

is called to create a new directory.

```
int dir_Delete(char *pathString, int pathStringLength)
```

is called to delete a directory.

```
int dir_Lookup(char *pathString, int pathStringLength,
        int index,
        char *name, int *nameLength,
        int *creationDate, int *modificationDate,
        int *isDirectory, int *sizeIfFile)
```

is called to read information about a file in a directory. The first three arguments are inputs, specifying the path to the directory to be searched and the index of the file within the directory (starting at 1). The remaining arguments are pointers to variables in which the routine should store information about the entry. The creation and modification dates should be in seconds relative to the Squeak epoch (see Section 42.11). This function sould return a success code as follows:

- `0` to indicate success (an entry was found in the directory at the given index);
- `1` to indicate that the index was past the end of the directory;
- `2` to indicate a problem with the `pathString` (for example illegal syntax or a path to some filesystem object that is not a directory).

Finally,

```
int dir_SetMacFileTypeAndCreator(char *filename,
        int filenameSize, char *fType, char *fCreator)
```

is intended for Mac OS only, can be ignored, and should simply return `1`.

---

of Squeak to Unix, the machine that was used *still* had bizarre symbolic links lurking in obscure, seldom-visited corners of the filesystem. (Removing them had simply been forgotten in the excitement of having a working Squeak system to play with!)

42.11 Time

The interpreter needs to recover two kinds of time from the support code. The first is "absolute" time, used for calculating the current date and "wall-clock" time. The second is "relative" time, used for measuring intervals between events.

Absolute time is the responsibility of the function

```
int ioSeconds(void)
```

which should answer the number of seconds that have elapsed since the Squeak "epoch"—midnight on the 1st of January 1901. If the host platform has a different "epoch" then a conversion will be necessary. For example, many systems use 1 January 1970 as their epoch; such systems would have to add 2,177,452,800 seconds (the number of seconds in 17 leap and 52 non-leap years) to the current time.

Three other functions are responsible for "relative" time. It doesn't matter what "epoch" they use (provided that the point of reference doesn't change during a single run of the virtual machine), but greater resolution is required—preferably to the nearest millisecond.

The function

```
int ioMSecs(void)
```

should return the number of milliseconds that have elapsed since some suitable reference time. (For example, the number of milliseconds since the virtual machine started running, or the number of milliseconds since the machine was booted.) The interpreter uses this clock for timing purposes, for example to determine when Delays should expire and for generating MIDI events. Although millisecond resolution is not *required*, the better its resolution the more accurate these timing activities will be. This clock represents a compromise between efficiency and accuracy.

The interpreter can get by with a much lower resolution clock for some activities, particularly when calling `ioMSecs()` is relatively expensive. For these purposes it calls

```
int ioLowResMSecs(void)
```

which *must* be fast, even at the expense of accuracy. A resolution as low as a few tenths of a second is acceptable.

Lastly, the function

```
int ioMicroMSecs(void)
```

is called only for profiling purposes. (The slightly peculiar name is meant to suggest that this function could be based on a microsecond clock, even though the answer that it provides is in milliseconds.) It should return the highest resolution of millisecond time available, regardless of how expensive it might be to obtain.

42.12 Image name

The support code is responsible for recovering the pathnames of the virtual machine executable and image files during initialization. The generated code uses the following functions and variables to access this information:

```
int imageNameSize(void)
int vmPathSize(void)
```

should return the length (excluding any terminating nulls) of the absolute paths to the image file and VM executable, respectively.

```
int imageNameGetLength(int sqImageNameIndex, int length)
int vmPathGetLength(int sqVMPathIndex, int length)
```

should copy the name of the image file or virtual machine executable into memory at the address given by their first argument (remember that there are no pointers in Squeak, only `int`s) which should not exceed `length` characters.

```
int imageNamePutLength(int sqImageNameIndex, int length)
```

is called to inform the support code that the name of the image has changed (before saving it with a new name, for example). The support code should update any data that depend on the name of the image, including

```
char imageName[]
```

which should contain the (null terminated) name of the image. (Some generated code refers explicitly to this array.)

42.13 Miscellany

```
int ioBeep(void)
```

should ring the keyboard bell. (Since any keyboard manufactured more recently than 1980 will probably not be equipped with a bell, it is acceptable that this function make some appropriate noise emanate from the computer's loudspeaker instead.)

```
int ioExit(void)
```

is called to terminate execution gracefully. This function should never return, and (apart from exiting) should perform no action other than releasing any resources that might have been allocated or reserved by the support code during initialization.

```
int ioFormPrint(int bitsAddr,
    int width, int height, int depth,
    double hScale, double vScale, int landscapeFlag)
```

is called to save an area of a Squeak Bitmap to a file in whatever the host might consider to be a useful format. Formats used on existing platforms include PostScript and PPM (Portable PixMap, a universal format for bitmapped images that can be converted easily into many tens of other popular formats). `bitsAddr` specifies the address of the first pixel in memory, `depth` the number of bits per pixel, `height` the number of scalines in the bitmap, and `width` the number of pixels in each scanline.[19] The final three arguments are obvious.

```
int ioRelinquishProcessorForMicroseconds(int microSecs)
```

is called from the generated code whenever Squeak runs out of interesting things to do. This function should "sleep" for the indicated number of `microSecs`. If any of the support code uses polling to check for input/output (network, serial port, and so on) then an "intelligent" implementation of this function would sleep while waiting for input to arrive (or output to complete), with a suitable timeout to ensure that Squeak wakes up again after no more

[19] Remember that the "pitch" of a scanline is always a multiple of 4 bytes, which means that some correction for the start of successive scanlines might be required if `width * depth` is not a multiple of 32.

than the given number of `microSecs` have elapsed. (If the host is dedicated to Squeak then a "stupid" implementation is also possible: the function can return immediately without sleeping. This will cause Squeak to "hog" the CPU, but on a "dedicated" host this is presumably not a problem.) This function should return the approximate number of microseconds that were spent sleeping, or `microSecs` if this information is not available.

42.14 Initialization and the function `main()`

The support code is responsible for providing the function `main()` (or whatever function is used for the "standard" entry point of a program on the host). `main()` is responsible for performing the following actions:

- parsing any command line arguments passed to the VM;
- determining the path to the image file either from the command line, from an environment variable, or from some other source (if the VM was started by a graphical manipulation for example);
- initializing any input/ouput subsystems that are supported (including the physical display and any colormaps that might be needed);
- loading the image file into memory;
- starting the Squeak interpreter to "run" the image.

These actions are described in more detail, and in the above order, below.

Parsing the command line arguments is only relevant on hosts that support a command-line interface. After parsing the arguments, the absolute paths to the image and VM executable files must be available via the functions described in Section 42.12, and the command-line arguments themselves must be available as system attributes. (Section 42.15 describes system attributes in detail.)

Three kinds of arguments should be distinguished:

- options meant specifically for the VM itself;
- the name of the image to run;
- options meant specifically for Squeak applications.

The exact format of the command line will depend on the host's conventions, but the above distinction should be respected and the VM should reject unknown VM options, if at all possible. The approach used on Unix-based systems, for example, is to enforce the following order on the command line arguments:

- options intended for the VM, distinguished from the image name by having a '-' prefix;
- the name of the image to run (which lacks the option prefix);
- "uninterpreted" arguments intended for Squeak applications.

(The VM saves all of these arguments for retrieval using negative system attributes, but saves only the arguments following the image name for retrieval as attributes 2 to 999.)

Initializing the input/output support code depends almost entirely on the platform, and the required actions must be inferred from the support code itself. The only platform-independent part of this initialization is related to the colormap that Squeak uses for 8-bit deep displays. This colormap is described in detail in Section 42.7.2.

Loading the image file into memory is accomplished by calling the generated function

```
readImageFromFileHeapSize(sqImageFile file, int heapSize)
```

where `file` is a handle on the (already opened) image file (of type `sqImageFile` as explained in Section 42.6), and `heapSize` is the amount of memory requested by the user (possibly from a command line option or environment variable). The return value of this function should be ignored.

A suitable default for `heapSize` should be provided. On a dedicated host this might be the total size of physical memory; otherwise 20 megabytes is certainly enough for all but the most demanding of Squeak images.

Finally, the `main()` function should call the automatically-generated function `interpret()`. This function is the entry point into Squeak's interpreter, and never returns to its caller (it's an infinite loop). All further interaction with the support code is made by "callbacks" from the generated interpreter code to the support functions described in this chapter.

id	*meaning of attribute*
-1...-N	the "raw" command line arguments that were supplied when starting the VM
0	the name of the VM executable
1	the name of the image file
2...M	the "cooked" command line arguments that were supplied when starting the VM
1001	the type of the operating system
1002	the name of the operating system
1003	the architecture of the host CPU
1004	the VM's version string

Table 42.5: Squeak's system attribute identifiers and their corresponding meanings.

42.15 System attributes

Squeak applications are sometimes interested in knowing about the host on which they are running. The support code provides this information through "system attributes", which are strings describing various characteristics of the host platform.[20]

Each attribute is identified by an integer. Generated code uses the usual two-function mechanism to retrieve this information from the support code (as described in Section 42.5.2).

```
int attributeSize(int id)
```

should return the number of characters in the string representing the attribute with the given `id`enditifer.

```
int getAttributeIntoLength(int id, int address, int length)
```

is called subsequently to transfer the string into Squeak's heap at the given `address`. The support code can assume that the `id` will not change between the generated code calling the first and second of these functions.

[20]The functions described in this section are connected directly to the primitive method SystemDictionary>>getSystemAttribute:.

Table 42.5 lists the currently assigned identifiers for system attributes, several of which merit further explanation.

The "raw" command line arguments are exactly as they appeared on the command line when the VM was invoked. They include both arguments intended for the VM and arguments intended to be recovered by Squeak applications. The latter will probably be more interested in the "cooked" command line arguments, which are uniquely those that the VM did not recognise as valid switches or the name of an image file.

The operating system *type* describes the "class" of operating system running on the host, while the *name* gives the particular OS within that class. (For example GNU/Linux returns `"unix"` for the type and `"linux-gnu"` for the name, whereas BSD returns `"unix"` and `"bsd"` respectively.) The processor architecture is a string such as `"68k"` (Motorola 68000 series), `"x86"` (Intel i386 and compatible), `"ppc"` (microprocessors based on the Motorola/IBM Power architecture), and so on.[21]

Finally, the interpreter version string should be taken from the variable

```
char *interpreterVersion
```

which is declared in, and defined automatically by, the generated code.

42.16 Support subsystems

A significant part of the support code is concerned with input/output subsystems. Any given subsystem `foo` implements at least two support functions: `fooInit()` is called to initialize it, and `fooShutdown()` is called to release any resources that it uses. The arguments to these two functions, and any additional support functions that might be necessary, depend on the subsystem itself.

The Macintosh versions of several subsystems are very well documented, and contain much more information than can (or should) be included here.

[21] Two possible ways to help determine the correct values of the OS attributes may exist on a given platform. The first is the "UTS" information for the host which is sometimes available via the command '`uname`'; the OS name should be the same as the UTS "system" and the architecture the same as the UTS "machine". Another possibility exists on hosts that use the GNU compiler. The output of '`gcc -v`' includes the canonical name of the host in the form *`cpu-vendor-os`* (with possibly a fourth component, which should be considered part of the *os*); the first and third components of this canonical host name correspond to Squeak's architecture and OS name attributes.

subsystem	*Macintosh source file*
asynchronous file i/o	`sqMacAsyncFilePrims.c`
file directories	`sqMacDirectory.c`
joystick	`sqMacJoystickAndTablet.c`
graphics tablet	`sqMacJoystickAndTablet.c`
MIDI port	`sqMacSerialAndMIDIPort.c`

Table 42.6: Optional subsystems that are well-documented in the Macintosh support code. The comments in each of these files are more than sufficient to modify the code for a new platform.

Table 42.6 lists these subsystems and the names of the corresponding Macintosh source files. They will not be described further here; instead the Macintosh files should be copied and then modified for the new host, according to the copious comments therein. To omit any given subsystem "foo" it is sufficient to "fail" the associated initialization primitive from within the function `fooInit()`. Section 42.3.2 describes how to extract the corresponding source files from the Squeak image.

The following sections describe only those optional subsystems that are difficult to implement, or that have poor documentation in the corresponding Macintosh source file: networking, sound, and serial port support.

42.17 Networking

Networking often proves to be one of the trickiest subsystems to implement, mainly because it inherits some peculiar conventions from the Macintosh origins of Squeak. For example, Squeak assumes that performing an `accept()` on a "listening" socket causes the socket itself to be connected to the peer—regardless of the capabilities of the socket implemention on the host. (On the vast majority of platforms the semantics are those of BSD Unix: the "accepted" socket creates a new connected socket, leaving the original socket listening for new connections. On such hosts we are obliged to destroy the original listening socket and create a new one, since that is the model adopted in Mac OS.)

The networking support can be divided into two independent services: socket-based communication and host name lookup (using the DNS).

42.17.1 Network initialization and shutdown

Generated code calls the support function

```
int sqNetworkInit(int resolverSemaIndex)
```

to initialize the networking subsystem. It should perform any platform-specific initialization and then store the `resolverSemaIndex` in a variable for use by the name lookup routines (which are described in Section 42.17.7). It should also compute (and remember somewhere) a unique integer that will be used to identify a network "session" (the period between initializing and shutting down the network subsystem). One possibility is to use the current millisecond time. This "session ID" is intended to help detect any attempt to use a "stale" Socket which was saved in the image and subsequently reloaded into a newly-launched Squeak. This function is a primitive, and should fail if the network cannot be initialized.

The corresponding shutdown function

```
int sqNetworkShutdown(void)
```

should release any resources that were allocated during network initialization.

42.17.2 Socket creation and management

When Squeak creates a Socket it calls a primitive method, associated with the support function

```
void sqSocketCreateNetTypeSocketTypeRecvBytesSendBytes\
     SemaIDReadSemaIDWriteSemaID
  (SocketPtr sptr, int netType, int socketType,
   int recvBufSize, int sendBufSize,
   int semaIndex, int readSemaIndex, int writeSemaIndex)
```

(Note that the identifier has been split simply because it is too long to fit the width of the page.) The `sptr` argument is a pointer to a structure (defined in `sq.h`) containing the following fields that must be initialized directly by the support code:

`int sessionID`	as computed during network initialization
`int socketType`	`0` streams, `1` for datagrams
`void *privateSocketPtr`	pointer to the associated `privateSocket` structure

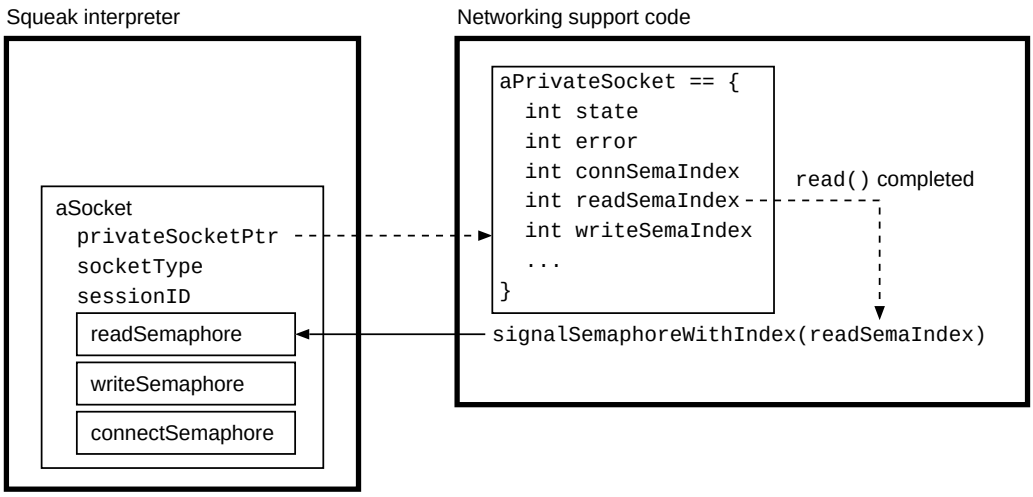

Figure 42.5: The relationship between a Socket object belonging to Squeak, and the corresonding `privateSocket` structure belonging to the support code. The three Semaphores are used to signal the completion of operations on the socket: read and write operations signal the Semaphores corresponding to `readSemaIndex` and `writeSemaIndex`, respectively. Completion of other operations (connecting, accepting, and so on) signal the Semaphore corresponding to `connSemaIndex`. See the text for descriptions of the other fields.

(Squeak will subsequently identify a Socket to the networking support by its associated `SocketPtr` pointer. The support code will have to dereference the `privateSocketPtr` field in this structure to retrieve the address of the `privateSocket` structure associated with the `C` socket.)

The `privateSocket` structure is defined by the support code, and can contain any information that might be required to manage a socket on the host. (The information in this structure is private to the support code, as implied by the name.) This structure should be allocated by the support code (using `malloc()`, for example) when a Socket is created, and then deallocated (using `free()`, for example) when the Socket is destroyed. Figure 42.5 illustrates the relationship between the Socket object in the image and the associated `privateSocket` structure maintained by the support code.

Most implementations will probably want to define at least the following fields in the `privateSocket` structure:

`int connSemaIndex`	"connect" completion Semaphore
`int readSemaIndex`	read completion Semaphore
`int writeSemaIndex`	write completion Semaphore
`int state`	the "connection status" of the socket
`int error`	the error code associated with the last operation performed on the socket

Whatever kind of "handle" the host uses to identify a socket should (of course) also be stored in this structure.[22]

The `netType` parameter is intended to specify alternate network protocols or interfaces, but is currently always `0`. Nevertheless, the support code should check this parameter (interpreting `0` as meaning "default") and fail the primitive if the type is non-zero (indicating that the support code is out of date with respect to the Socket facilities provided in the image). The `socketType` is either `0` for stream-based sockets (e.g. TCP), or `1` for datagram-based sockets (e.g. UDP).

The two buffer size arguments are used to tune the performance of the network code to a particular application. They specify (in bytes) the ideal size of buffer that should be associated with the socket. (These arguments can be ignored if the host does not support changing a socket's buffer sizes.)

The final three arguments specify the indices of Semaphores that are to be signalled (see Section 42.5.3) whenever a connection-, read-, or write-related operation is completed for the Socket.

The "connection status" of a socket is read by generated code via the function

```
int sqSocketConnectionStatus(SocketPtr s)
```

which should return one of the following values:

`0`	unconnected (the initial state)
`1`	waiting for a connection to complete
`2`	connected
`3`	closed (by the peer)
`4`	closed (by the local host)

Similarly, generated code uses the function

[22] The name `privateSocket` is an example only: the implementation can call this structure by any name it likes, since generated code never references it directly.

```
int  sqSocketError(SocketPtr s)
```

after the failure of a network operation to retrieve a code identifying the problem. The error codes are currently not interpreted by Squeak (since they depend intimately on the host). However, with future expansion in mind, the support code should remember (and provide via this function) whatever error code was indicated by the host operating system.

The support code should also provide four functions to retrieve the local and remote host and port numbers associated with a connected socket, as follows:

```
int sqSocketLocalAddress(SocketPtr s)
int sqSocketLocalPort(SocketPtr s)
int sqSocketRemoteAddress(SocketPtr s)
int sqSocketRemotePort(SocketPtr s)
```

These functions should return the information in host (not network) byte order, or `0` for a socket that is valid but inappropriate (the remote address for an unconnected socket, for example), or `-1` if the `SocketPtr` is invalid (its `sessionID` is not correct).

Finally, when Squeak destroys a Socket it calls the support function

```
void sqSocketDestroy(SocketPtr s)
```

which should release any private resources (including the `privateSocket` structure) associated with `s`. This function is associated with a primitive method, and should therefore fail the primitive if a problem occurs.

Note that all of the networking support functions that receive a `SocketPtr` as an argument should perform a minimum of "sanity checking", which means at least verifying that the `sessionID` stored in the `SocketPtr` corresponds to the one computed during network initialization.

42.17.3 Connecting and disconnecting

"Client" and "server" socket connections are implemented by the support functions

```
void sqSocketConnectToPort(SocketPtr s, int addr, int port)
void sqSocketListenOnPort(SocketPtr s, int port)
```

which (as before) use host byte order for `addr` and `port`. These functions should also ensure that `signalSemaphoreWithIndex()` is called for the connection Semaphore associated with `s` to let Squeak know when a connecting

socket is connected or when an `accept()` has been performed on a listening socket. (It is entirely the responsibility of the support code to detect when a connection request arrives at a listening socket and to perform any subsequent call to `accept()` that might be required.) Since these functions are associated with primitives, they should fail if a problem occurs during connection.

As mentioned above, listening sockets do not have the usual semantics. After `accept()`ing a connection, Squeak expects to use the *same* `SocketPtr` to perform subsequent data transfer on the connected socket. On hosts that use BSD-style sockets this involves destroying the listening socket and reinitializing the `SocketPtr` and `privateSocket` structures to refer to the newly-connected socket.

Connection termination is implemented by the functions

```
void sqSocketCloseConnection(SocketPtr s)
void sqSocketAbortConnection(SocketPtr s)
```

which are associated with primitive methods. The first should fail if the associated socket is not connected; the second should fail only if the `SocketPtr` is invalid for the current session.

42.17.4 Sending and receiving data

Data transfer is implemented by two functions

```
int sqSocketReceiveDataBufCount
      (SocketPtr s, int buf, int bufSize)

int sqSocketSendDataBufCount
      (SocketPtr s, int buf, int bufSize)
```

in which `buf` is the address of the data to be transferred and `bufSize` is the size of the data measured in bytes. These functions should return the actual number of bytes transferred (which can be `0`, in the case of an error).

Generated code also requires two support functions that answer whether data transfer can take place.

```
int sqSocketReceiveDataAvailable(SocketPtr s)
```

should return `true` or `false` to indicate whether data is available for `s`; similarly

```
int sqSocketSendDone(SocketPtr s)
```

to indicate whether data can be written without blocking the caller. Both functions should return `-1` if the `SocketPtr` is not valid for the current session.

The support code should also ensure that the read or write Semaphore (as appropriate) associated with the socket is signalled, whenever an operation completes.[23] Figure 42.5 illustrates this interaction for the case of a "read" operation that has been completed.

42.17.5 Optional BSD-style connection semantics

A recent addition to Squeak supports sockets that implement BSD-style semantics, in which the connected socket does not replace the listening socket when a connection request is `accept()`ed. The function

```
void sqSocketListenOnPortBacklogSize
  (SocketPtr s, int port, int backlogSize)
```

is similar to `sqListenOnPort()`, but should succeed only if the host supports BSD-style sockets. The `backlogSize` indicates the number of pending connections that should be allowed on the listening socket. This function should ensure that the connection Semaphore associated with `s` is signalled when an `accept()` can be performed (but it should *not* perform the `accept()`). Squeak will subsequently call

```
void sqSocketAcceptFromRecvBytesSendBytes\
     SemaIDReadSemaIDWriteSemaID
  (SocketPtr s, SocketPtr serverSocket,
   int recvBufSize, int sendBufSize,
   int semaIndex, int readSemaIndex, int writeSemaIndex)
```

to perform the `accept()`, passing the original listening socket as `serverSocket` and a newly-created SocketPtr as `s`. This function should initialize `s` as for any other newly-created socket, including allocating a new `privateSocket` structure for it.

Both of these functions are primitives, and should fail if an error occurs.

[23] Note that these Semaphores should always be signalled when an operation completes, even if the operation completes immediately.

If the host does not support BSD-style semantics for listening sockets then it should fail these two primitives, in which case Squeak will revert to the (origina, Macintosh-style) behavior described previously.

42.17.6 Backwards compatibility

Prior to version 2.8 of Squeak, all socket-based input/output used a single Semaphore to communicate asynchronous events to the virtual machine. For compatibility with older images the support code should therefore implement simplified versions of the socket creation and accept functions:

```
void sqSocketAcceptFromRecvBytesSendBytesSemaID
    (SocketPtr s, SocketPtr serverSocket,
     int recvBufSize, int sendBufSize, int semaIndex);

void sqSocketCreateNetTypeSocketTypeRecvBytesSendBytesSemaID
    (SocketPtr s, int netType, int socketType,
     int recvBufSize, int sendBufSize, int semaIndex)
```

These functions are trivial. They are identical to their "three-semaphore" equivalents, except that they take only a single `semaIndex` argument. They can simply call the three-semaphore versions, passing their arguments unmodified, and reusing their single `semaIndex` argument three times as the `semaIndex`, `readSemaIndex` and `writeSemaIndex` arguments.

42.17.7 Host name lookup

Squeak supports host name resolution via the DNS. The interface is a little larger than might be expected, to permit asynchronous lookup on hosts that support it.

Initialization is implicit in the network initialization described above. The support code need only store the `resolverSemaIndex` that was passed to `sqNetworkInit()`.

When Squeak wants to convert a host name string into a numeric address it calls the support function

```
void sqResolverStartNameLookup(char *hostName, int nameSize)
```

where `nameSize` is the length of the (Squeak) string in `hostName`. The support code should signal the `resolverSema` (saved during network initialization) when the lookup has completed. Squeak will then call

```
int sqResolverNameLookupResult(void)
```

to recover the result, which should be a numeric address in host byte order, or `-1` to indicate failure.

Reverse lookups (addresses to names) should also be provided by the support code. Squeak calls

```
void sqResolverStartAddrLookup(int address)
```

to begin the lookup, which should cause the `resolverSema` to be signalled when the lookup is finished. To retrieve the result, Squeak uses the usual pair of functions:

```
int sqResolverAddrLookupResultSize(void)
```

```
void sqResolverAddrLookupResult(char *nameForAddress,
                                int nameSize)
```

to recover the length of the result and then perform the actual transfer of bytes into a Squeak String.

Generated code will call the support routine

```
int sqResolverLocalAddress(void)
```

if it decides to abort a lookup operation before it has completed.

The support code should also provide three trivial functions:

```
int sqResolverLocalAddress(void)
```

should return the address of the local host;

```
int sqResolverError(void)
```

should return the operating system error code for the last operation in the case of failure (this value is currently not interpreted by Squeak, but should be correct to allow for future expansion); and finally

```
int sqResolverStatus(void)
```

should return one of the following values to indicate the current status of the resolver subsystem:

- `0` the resolver is uninitialized (`sqNetInit()` not yet called)
- `1` the last lookup was successful
- `2` a lookup is currently in progress
- `3` the last lookup failed

42.18 Sound

Squeak supports the generation and playback of CD-quality stereo audio.[24] The sound subsystem contains, as always, the usual initialization and shutdown functions.

```
int soundInit(void);
int soundShutdown(void);
```

Sound output is initiated by calling the function

```
int snd_Start
  (int frameCount, int samplesPerSec,
   int stereo, int semaIndex)
```

where `samplesPerSec` is the number of 16-bit samples to be played per second, `stereo` is `true` for stereo or `false` for mono, `semaIndex` refers to a Semaphore that should be signalled when sound input/output completes (see Section 42.5.3), and `frameSize` indicates the amount of buffer space that should be allocated for sound output. The size of output buffer (in bytes) that should be allocated is twice the `frameCount` for mono (two bytes per sample) or four times `frameCount` for stereo (two bytes per channel per sample). This function should return `true` if initialization is successful, `false` if not.

The function

```
int snd_AvailableSpace(void)
```

sould return the amount of space available in the ouput buffer, measured in bytes (not frames).

Three functions are used to insert sound into the output buffer.

```
int snd_PlaySilence(void);
```

is used to fill the output buffer with silence. It should return the number of bytes of space remaining in the output buffer.

[24] An upper limit on sound quality is imposed by the amount of processor power available. Recent machines have no trouble achieving CD quality.

```
int snd_PlaySamplesFromAtLength
  (int frameCount, int arrayIndex, int startIndex)
```

is called to insert `frameCount` samples into the output buffer, from memory at the address `arrayIndex + (startIndex * 2)` (mono) or `arrayIndex + (startIndex * 4)` (stereo). The sound should begin playing immediately if possible. This function should return the amount of available space remaining in the output buffer (measured in bytes).

```
int snd_InsertSamplesFromLeadTime
  (int frameCount, int srcBufPtr, int samplesOfLeadTime)
```

is called to insert `frameCount` samples from `srcBufPtr` into the output buffer, with the specified number of samples of lead time (delay) before the sound beings to play. Again, this function should return the amount of remaining available space in the output buffer. Finally,

```
int snd_Stop(void)
```

is called to abort sound output. It should take appropriate measures to stop sound output as soon as possible.

Sound input is handled via four support functions.

```
int snd_SetRecordLevel(int level)
```

is called to set the input gain to a value between `0` (minimum gain) and `1000` (maximum gain).

```
int snd_StartRecording
  (int desiredSamplesPerSec, int stereo, int semaIndex)
```

is called to initiate recording, with arguments analogous to those for sound output. The actual input sampling rate should be returned by the function

```
double snd_GetRecordingSampleRate(void)
```

Data transfer from the input buffer to Squeak's memory is the responsibility of

```
int snd_RecordSamplesIntoAtLength
  (int buf, int startSliceIndex, int bufferSize)
```

where `buf` is the destination address, `bufferSize` is measured in bytes, and `startSliceIndex` is the sample offset in `buf` from which data should be

written. Since this offset is measured samples it should be scaled by 2 (mono) or 4 (stereo) to arrive at a byte offset. The routine should take care not to write past the end of `buf` (remembering that `bufferSize` is measured in bytes, not samples). The return value is the number of samples (not bytes) that were actually transferred. Finally,

```
int snd_StopRecording(void)
```

is called to disable recording. The return value is ignored.

42.19 Serial port

As with the other subsystems, serial port support begins with the two functions

```
int serialPortInit(void)
int serialPortShutdown(void)
```

for initialization and subsequent releasing of resources. The first of these is a primitive and should therefore fail if no serial ports are supported.

Serial ports are "opened" via the support function

```
int serialPortOpen
  (int portNum,
   int baudRate, int stopBitsType,
   int parityType, int dataBits,
   int inFlowCtrl, int outFlowCtrl,
   int xOnChar, int xOffChar)
```

The possible values of these parameters are shown in Table 42.7. When a serial port is no longer needed, the genreated code calls

```
int serialPortClose(int portNum)
```

to release any resources owned by the specified port.

Data transfer is effected by two support functions

```
int serialPortReadInto(int portNum, int count, int bufferPtr)
int serialPortWriteFrom(int portNum, int count, int bufferPtr)
```

where `bufferPtr` is the address of the data source/destination, and `count` is the number of bytes to be transferred. These functions should return the number of bytes actually read/written, and immediately (even if no data can be transferred).

`portNum`	the port number, `0` or `1`
`baudRate`	requested port speed
`stopBitsType`	`0` means 1.5 stop bits
	`1` means 1 stop bit
	`2` means 2 stop bits
`parityType`	`0` means no parity
	`1` means odd parity
	`2` means even parity
`dataBits`	5–8
`inFlowCtrl`	`true` to use h/w flow control
`outFlowCtrl`	`true` to use h/w flow control
`xOnChar`	ASCII value of XON character, or `0`
`xOffChar`	ASCII value of XOFF character, or `0`

Table 42.7: Parameters passed to `serialPortOpen()`.

42.20 Plugin modules

Many primitives are "hardwired" into interpreter, and identified by a numeric index. This arrangement has several drawbacks, including possibly limiting the number of primitives that can be provided[25] and requiring the virtual machine to be recompiled whenever primitives are modified or added.

To circumvent these limitations, Squeak provides a mechanism for assigning names to primitives whose definitions are loaded at runtime from external, dynamically-loaded, shared libraries (sometimes called "DLLs"). From within Squeak these functions appear as "named primitives", and the dynamic libraries in which they are defined are called "modules" (or "plugins"). For this mechanism to work, the support code must provide functions for finding, loading, and resolving names in dynamically-loaded libraries.

```
int ioLoadModule(char *pluginName)
```

is called by the generated code to load the dynamic library with the given `pluginName`. This name does not make any assumptions about the host. If there is a standard prefix or suffix for dynamic libraries then the support code

[25]The primitive "dispatch" mechanism is translated into a `C switch` statement in the generated code, and some compilers place a limit on the number of `case` labels that can appear within a `switch`.

must add it to the `pluginName`. Also, if there are several standard places in which to search for the library then the support code must implement the search explicitly (the `pluginName` is *never* a pathname). This function should answer a unique non-zero integer "handle" that will be used to identify the plugin to the two other plugin support functions. If no library corresponding to the `pluginName` can be found then this function should return `0`.

```
int ioFindExternalFunctionIn(char *name, int moduleHandle)
```

should search the plugin module (dynamic library) having the given handle (obtained from a previous call to `ioLoadModule()`) for the function corresponding to `name`. `name` is an identifier for a `C` function, exactly as it appears in the plugin source code. If the host has any special conventions for symbols in binary files (for example, some binary formats prefix all symbols with an underscore '_') then the support code must take this into account. This function should return the address of the function corresponding to `name`, or `0` if the function is not present in the module.

```
int ioFreeModule(int moduleHandle)
```

is called when Squeak wants to "unload" a plugin module. This function should return `1`. If the host does not support the unloading of dynamic libraries, or if an error occurs, then it should return `0`.

For an initial port of Squeak, all three of these functions can be defined trivially to return `0`. They should *not* "fail" the primitive. (This detail is small, but *very* important.)

42.21 Profiling

Smalltalk (the SystemDictionary) contains four methods for collecting runtime profiling information. These are associated with four optional support functions. (Their return values are ignored.)

`int startProfiling(void)`	turns profiling on
`int stopProfiling(void)`	turns profiling off
`int clearProfile(void)`	should delete any stale profiling information (for example, clearing a buffer of sampled PC values to zero)
`int dumpProfile(void)`	should save the collected profiling information in a form appropriate for the host

Profiling is mainly of interest to the implementors of the Squeak interpreter, and should not be considered a priority in a new port.

42.22 "Headless" operation

Squeak provides some impressive "server" capabilities (for Web sites in particular). A Squeak-based server is not normally intended for interactive use, and the usual graphics/keyboard/mouse facilities are at best irrelevant (and at worst a security risk). "Headless" operation refers to running Squeak with these facilities disabled. Most of the current ports of Squeak support this mode of operation, either in response to a command-line option or by using a VM compiled with a special preprocessor symbol to conditionally omit these facilities in the support code.

If appropriate, any new port should try to implement a headless mode of operation. Doing so should require only the following changes in support code behavior:

- the warning beep is disabled. `ioBeep` should therefore return `0` without doing anything else;

- graphical output "succeeds" without actually transferring anything to a physical screen. The following functions should therefore do nothing (and return `0`):

  `ioShowDisplay()`,
  `ioForceDisplayUpdate()`,
  `ioSetFullScreen()`,
  `ioSetDisplayMode()`,
  `ioSetCursor()`, and
  `ioSetCursorWithMask()`.

- keyboard and mouse input is disabled. `ioGetKeystroke()` and `ioPeekKeystroke()` should return `-1` to indicate that there is nothing in the keyboard input buffer. `ioGetButtonState()` and `ioMousePoint()` should return `0` immediately;

- there is no screen, so there is no screen size. `ioScreenSize()` should return some harmless default value, such as `0x00400040` (64×64);

- `ioHasDisplayDepth()` should simply answer "yes" (return `1`) for all display depths;
- there are no keyboard/mouse input events. `ioProcessEvents()` can return `0` immediately (or possibly after performing any non-interactive polling that it might also be responsible for—network or serial port I/O, for example).

42.23 Conclusion

Squeak is a (rapidly) moving target. The user community is adding new features at a furious rate, and it is almost certain that Squeak will include new capabilities—and associated support code—by the time this book appears in print. This need not be a cause for alarm, for two reasons.

First, the fact that most new facilities are "optional" means that they do not affect the initial task of porting Squeak to a new platform; the information presented here should remain relevant (and sufficient) for a long time to come. Truly platform-dependent additions happen rarely, and are likely to be limited to very minor details such as provision of additional system attributes.

Second, Squeak's support for adding new primitive methods decouples the support code from many new "low-level" parts of the implementation. Writing new primitives in Smalltalk and then automatically generating the equivalent `C` is a routine activity for Squeak virtual machine hackers. Such generated primitives, which are necessarily platform-independent, are complemented nicely by "plugin modules" for dynamically adding primitives to a running system. These modules can include (and encapsulate) platform-specific details without affecting the "intrinsic" support code for a given platform at all.

Acknowledgements

I am grateful to Andreas Raab and John Maloney for their detailed comments on, and suggestions for improving, the first draft of this chapter.