Interfacing Smalltalk with the Real World

Contents

ESUG Summer School 1995

Interfacing Smalltalk
with the Real World

Ian Piumarta

INRIA Rocquencourt

About the speaker

Ian Piumarta discovered Smalltalk in 1987, shortly before completing a B.Sc. in computer science at the University of Manchester (UK). He later received a Ph.D. from the same university for developing novel code generation techniques, demonstrated practically in the compiler for his own Smalltalk implementation. His last few years at the University were spent as a research associate involved with the design and implementation of large knowledge-based Smalltalk systems for use in primary healthcare.

Ian has been involved frequently in the teaching of Smalltalk at all levels, mainly on industrial courses offered by the PEVE unit at Manchester.

In January 1995, after periods spent working for Harlequin in the UK and at IRCAM in Paris, he became a member of the SOR project at INRIA (the French National Institute for Research in Computing and Control) working on distributed object systems.

Although currently suffering from an overdose of the C++ "language" at work, he remains passionate about Smalltalk.

Ian can be contacted at:

INRIA Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France.

tel: +33 1 39 63 52 87
email: `ian.piumarta@inria.fr`

Overview

Three major topics covered:

- Smalltalk intrinsics: processes, sockets, snapshots.
- user-defined primitives (UDP): primitives, data, errors and failures; advanced topics.
- PPS Object Kit for C Programming: linking and calling C functions, managing C data, calling Smalltalk; packaging considerations.

Format:

- In-depth coverage of the topic.
- Extended example.

Overview

Part 1 — Smalltalk intrinsics. The Smalltalk system provides intrinsic facilities itself for creating and interacting with Unix processes, and for simple client/server programming using Unix sockets.

Part 2 — user-defined primitives (UDP). The UDP interface provides functions for invoking primitives written in C, accessing Smalltalk objects from within C, and for converting between Smalltalk and C representations.

Part 3 — the Object Kit for C Programming (CPOK). The CPOK can parse C header files to produce "stub" methods automatically. When called, these methods dynamically link any required C libraries if necessary. C data can be created and accessed from within Smalltalk, and "callbacks" can be made from C into Smalltalk.

Format of Each Section

Each section is in two halves.

- A brief overview of the topic, followed by an in-depth coverage of the parts that are practically useful. A lot of information will be covered in a relatively short time. A thorough understanding is not necessary of all the details, simply a grasp of the essential points and a feeling for the possibilities.
- An extended example to reinforce the theory covered in the first half of the section. This example will show most, if not all, of the details covered. Bringing together the theory in one example should provide a much better feel for how it all works.

The material covered is specific to ParcPlace ObjectWorks\Smalltalk release 4.1 running in a Unix environment.

Part 1 — Using Intrinsic Features

- Socket-based IPC
- The Unix Environment
- Running a Process
- Communicating with the Process
- Synchronisation and Process Status
- Simple Example: Running a C Program
- UnixProcess: Class Examples
- A Complete Example — A Smalltalk Server for C Clients

INRIA ROCQUENCOURT

Part 1 — Using Intrinsic Features

Smalltalk is supplied with several auxiliary code modules which are used for creating and interacting with Unix-level processes.

- `ExtIPC.st` provides the framework for communicating with external processes, and a mechanism for fetching the values of the Unix environment variables.
- `UnixIPC.st` adds the Unix-specific code for communicating with external processes. It provides facilities for process creation, reading results produced by a process, and communication with the process via its `stdin` and `stdout`.
- `UnixSock.st` contains classes representing Unix sockets providing Smalltalk programs with the opportunity to act as (for example) servers for, or clients of, external processes — without requiring the user to write any communication code in C.

Unix IPC

Sockets:

- universal IPC mechanism
- end-point for communication
- simplex or duplex
- point-to-point or one-shot
- socket ≈ file descriptor
  - ▷ `read(socket, ...)`
  - ▷ `write(socket, ...)`

Internals:

- host address
- port number
- file name

INRIA ROCQUENCOURT

Unix IPC

We will begin by jumping in at the deep end. A bottom-up approach to describing the facilities available in Smalltalk for communicating with the real world seems preferable to an attempt at describing it all in a "politically-correct" top-down manner. A feel for all the "cogs and wheels" at the outset will make understanding the layers built on top of them much easier, and will remove the need to gloss over any low-level details that might creep into more abstract facilities.

Before talking Smalltalk it is necessary to review briefly how Unix processes communicate with one another.

The universal communication abstraction used in Unix for inter-process communication (IPC) is the *socket*. A socket is an end-point for communication, which can be simplex or duplex. Sockets are associated with an address consisting of two parts, a host address and a port number. The host address identifies the machine to which the socket "belongs", and the port number is usually related to a specific service or session.

A socket looks a little like an open file, and is attached to a file descriptor. Many of the unbuffered input/output operations that apply to files can also be applied to sockets: `read()` and `write()` being the two most common.

The main differences between a socket and a file is that data written to a socket is sent "somewhere else" (another process) and lost forever to the sender, data can only be read from a socket if it has been sent from another process, and data can arrive at a socket for subsequent reading at any time. None of these properties (normally) apply to regular files.

SocketAddress
    domainCodeFromName: #afUnix

UDSocketAddress name: '/tmp/.X11-unix/X0'

Unix Domain Addresses

A socket is always associated with an address (even if the address represents "no address in particular"). There are many "families" of addresses in use, but the two most common are "unix" and "inet". An address in the unix family is said to be in the unix domain, and similarly an address in the inet family is in the inet domain.

A unix domain address is a file name. The socket is represented by an object that appears in the local filesystem with the specified name. All processes communicating using a unix domain socket must be running on the same host, which limits their usefulness. However since communication between the processes avoids the network software and hardware, communication is more efficient than with sockets in the inet domain.

For example, The X Window system uses a Unix domain socket called `/tmp/.X11-unix/X`*displayNumber* for clients that run on the same host as the server. (The use of the Unix domain for this socket is where the display name `unix:0` comes from.)

The address family of a socket must be specified when it is created. The class SocketAddress has a method that returns a domain code based on a symbolic name. The code for a Unix domain address can be found with

```
SocketAddress domainCodeFromName: #afUnix
```

A socket address in the Unix domain is created by sending the class UDSocketAddress the message name:, indicating the name of the "file" to which the socket will be attached.

The name of an existing Unix domain socket address can be found by sending it the message name.

Unix domain sockets do not clean up after themselves. The entry in the filesystem that they leave behind must be removed manually by the application which created the socket.

SocketAddress

domainCodeFromName: #afInet

IPSocketAddress

hostAddress: (IPSocketAddress thisHost)

hostName: 'localhost'

port: 6000

Inet Domain Addresses

An inet domain address has two parts: a host address (in the usual IP format) identifying the machine on which the remote process exists, and a port number representing the place where a remote process must “dock” in order to find the socket.

For example, the X Window System uses an Inet domain socket for clients that run on a different host to the server. The address of this socket is:

`host:` *displayServerHostAddress*
`port:` *6000 + displayNumber*

The code for a Unix domain address can be found with

```
SocketAddress domainCodeFromName: #afInet
```

A socket address in the Inet domain is created by sending the class IPSocketAddress a message indicating the host address or name and (optionally) the port number. The message thisHostAnyPort is useful for creating a socket associated with any free port, for clients that do not care which port they communicate on.

The host name, host address and port number of an existing Inet domain socket address can be found by sending it hostName, hostAddress and port respectively.

UnixSocketAccessor sockDgram.

Socket Types

Sockets come in two (sometimes more) types. Each type is associated with a particular communications protocol used to transport data between the communicating processes. The two main protocols in use by user-level applications are the User Datagram Protocol (UDP) and the Transmission Control Protocol (TCP).

UDP sockets are used for sending one-off, one-way messages to a known destination, and for receiving one-off, one-way messages from one or many sources. UDP sockets are essentially stateless—the destination address must be specified for every message sent via a UDP socket, and the source address can be different for each message received on a UDP socket. (The exception to this is that a receiving socket can be set up to receive messages from a given host only.) The overheads associated with sending a message on a UDP socket are high, since message routing and other operations must be performed for every message that is sent.

UDP messages are often called datagrams, and can be compared with (unreliable) telegrammes in the real world.

TCP sockets implement two-way byte streams. A destination address is connected with the socket when it is created. Before the socket can be used it must be connected to a corresponding remote TCP socket. The resulting communication channel acts like a piece of wire connecting the two processes along which information can be sent. Once this connection is made, the relationship between the two sockets cannot be altered until one of them is destroyed. The connection overhead is higher than with UDP sockets but is only paid once, when the socket is created. Subsequent messages sent via a TCP socket already “know” where they’re going, so the cost of communication is much lower than with a UDP socket which must figure out to where each individual message is to be delivered.

TCP connections are often called virtual circuits, and can be compared with telephone calls in the real world.

A code for the type must be supplied when a socket is created. The classUnixSocketAccessor provides the method sockDgram returning the code for UDP sockets, and sockStream for TCP sockets.

TCP Sessions and Port Numbers

We apparently have a contradiction here. If services are advertised on well-known port numbers (some of which are listed in the file `/etc/services`) agreed in advance, how can more than one TCP session ever be used with a service such as `telnet`? Such services plainly do not place a limit of 1 on the number of sessions that can be active at the same time.

The solution is that TCP connections are made to a socket associated with a well-known port, but the subsequent session uses a new socket associated with a system-supplied "random" port number. Two sockets are therefore involved: the well-known service socket to which connections are made, and a system-supplied session socket on which communication is performed.

Binding

Binding an address to a socket:

aTCPsocket bindTo: anIPSocketAddress

Creating a suitable address:

UDSocketAddress name: aString

IPSocketAddress hostName: aString

IPSocketAddress
hostName: aString
port: aSmallInteger

IPSocketAddress thisHostAnyPort

recvFrom: address...

Binding an Address to a Socket

Before a "server" socket can be used to receive data (either datagrams or a connection requests from a remote client) it must be bound to an address.

The socket is created with a particular address family. The kind of address bound to the socket must agree with the socket's address family. For sockets in the afInet family, the address must be an instance of IPSocketAddress. For sockets in the asUnix family, it must be an instance of UDSocketAddress.

The socket is bound to an address by sending it the message bindTo: with the address as the argument. Once a TCP socket is bound to an address it can be sent messages such as listenFor: and accept to accept connection requests from remote processes. Once a UDP socket is bound, it can be sent messages such as receiveFrom: to accept incoming datagrams. These messages are described in more detail below.

Socket addresses can be generated in a variety of ways.

- UDSocketAddress name: aString creates Unix-domain socket address corresponding to the given filesystem entry.
- IPSocketAddress hostName: aString returns an Inet-domain socket address for the named host with a "don't care" value for the port.
- IPSocketAddress hostName: aString port: aSmallInt returns an Inet-domain socket address for the named host with the named port.
- IPSocketAddress thisHostAnyPort returns an Inet-domain socket address for the current host with a "don't care" value for the port.
- Finally, the recvFrom: address... family of messages (applicable to UDP sockets and described below) will fill in the supplied address argument with the actual address of the sender of the received datagram.

A socket will respond with its bound address when sent the message getName.

Note that bindTo: associates a local address with the socket, in contrast to connectTo: (described below) which associates a remote address with a socket.

Listening and Accepting

```
boundTCPsocket listenFor: queueSize.

newSocket := boundTCPsocket accept.

(newSocket := boundTCPsocket
                        acceptNonBlock) isNil
    ifTrue: [ ...connection... ]
    ifFalse: [ ...no connection... ]
```

INRIA ROCQUENCOURT

Listening and Accepting TCP Connections

A bound TCP socket can listen for and accept incoming connection requests.

Sending the socket the message listenFor: causes the socket to become ready to accept incoming connection requests. The argument is a SmallInteger specifying the maximum size of the backlog of connect requests that will be queued on the socket during the processing of any given connection request. Once this limit is exceeded further connection requests will be refused. A value of 1 or 2 is fair, if connection requests are dealt with almost immediately by the server. More might be required if the server only occasionally polls the socket for pending connections, with 5 being a reasonable upper limit in almost any situation.

The socket can now be sent the message accept which will cause it to wait for (if necessary) and accept the next incoming connection request. The result of this message is a new socket connected to the remote socket from which the connection request originated. Once this message returns, the original socket is free to accept more incoming connections.

If there are no pending connection requests, the accept message will block the sender until one arrives. A similar message acceptNonBlock is provided which will return nil if there are no connection requests pending on the socket, without blocking the sender.

The newly created TCP socket can be used to communicate with the remote process that requested the connection, as described in the next slide.

Connecting

```
unboundTCPsocket
    connectTo: anIPSocketAddress.

| buffer |
buffer := ByteArray new: 4096.

connectedTCPsocket
    readInto: buffer;
    readInto: buffer startingAt: 2048 for: 2048;

    writeAll: buffer;
    writeFrom: buffer;

    writeFrom: buffer startingAt: 2048 for: 2048;

    getPeer.
```

Connecting to a Remote Socket

An unbound TCP "client" socket can be used to send a connection request to a remote socket by sending it the message `connectTo:` with the address of the remote socket as the argument.

As described in the previous slide, when the remote socket accepts the connection request a new (remote) socket is created to handle the communication. When the `connectTo:` message returns, the local socket can be used to communicate with the (newly created) remote socket using the `read` and `write` messages that apply to regular files (these are inherited from the class `IOAccessor`, which is a common superclass of both `UnixSocketAccessor` and the various flavours of `FileAccessor`):

- `readInto: buffer` reads data from the socket into the `buffer`. The number of bytes actually read is returned.
- `readInto:buffer startingAt: index for: count` is similar except that the maximum number of bytes to transfer, and a starting offset into the buffer, can be specified.
- `writeAll: buffer` writes the entire contents of `buffer` to the socket, returning the number of bytes actually transferred. This number will always be the same as the size of the buffer unless an error occurs during the transfer.
- `writeFrom:` and `writeFrom:startingAt:for:` do the same for writing as `readFrom:` and `readFrom:startingAt:for:` do for reading.

In all of the above messages, the `buffer` should be a `ByteArray`.

A connected socket will respond with the address of the remote socket when sent the message `getPeer`.

Note that `connectTo:` associates a remote address with the socket, in contrast to `bindTo:` (described above) which associates a local address with a socket.

The messages for sending and receiving datagrams described in the next slide can also be used on a connected TCP socket, although the only good reason for doing this is to be able to specify message options (described later) during the read or write.

Datagrams

```
unboundUDPsocket
    writeWait;
    sendTo: address buffer: buffer;
    sendTo: address buffer: buffer
            start: offset for: count;

    readWait;
    receiveFrom: address buffer: buffer;
    receiveFrom: address buffer: buffer
            start: offset for: count.
```

Sending and Receiving Datagrams

An unbound UDP socket can be used to send and receive datagrams. The destination address for an outgoing datagram is specified every time data is sent. An unbound UDP socket can therefore communicate with an arbitrary remote socket each time data is sent.

- sendTo: address buffer: buffer sends data from the buffer to the UDP socket with the given address.
- sendTo:address buffer: buffer start: offset for: count is similar, but allows a byte count and offset into the buffer to be specified.
- receiveFrom: address buffer: buffer receives a datagram from the socket and stores it in buffer. The size of the datagram is returned. The address does *not* specify the address of the sender; it is a "result parameter" which will be filled in with the address of the socket which sent the datagram. An unbound UDP socket will accept datagrams from any sender.
- receiveFrom: address buffer: buffer start: offset for: count is similar, but allows a maximum size of datagram and a starting offset into the buffer to be specified. (If the datagram is larger than count bytes, the datagram is truncated and the remaining data are lost.)

In all of the above messages, the family of the address must agree with that specified for the socket when it was created.

It is very important to note that the socket must be in a state where it is ready to send or receive data before any of the above messages are sent to it. Before sending a socket any of the read messages, it should be sent a readWait message which will block the caller until the socket has data available for reading. Similarly a writeWait should be sent prior to any write message.

An address cannot be bound to a UDP socket from Smalltalk. (This can be done from Unix, in a user-defined primitive for example, but is not advised. The utility of a bound UDP socket is negligible compared to the confusion that doing this might cause to the virtual machine.)

Creating a Socket

UnixSocketAccessor

family: domainCode *(afInet or afUnix)*

type: typeCode *(dgram or stream)*;

newTCP;

newUDP;

newTCPclientToHost: hostName port: portNo;

newTCPserverAtPort: portNo;

newUDPserverAtPort: portNo.

Creating a Socket

Finally we can come to the topic which might seem like the place where we should have started. Unfortunately, without some prior explanation of address families, socket addresses, socket types and binding, it is difficult to explain the various methods which create sockets.

The class UnixSocketAccessor provides several methods for creating bound or unbound sockets, using UDP or TCP protocols, in the Unix or Inet domain.

- family:type: creates an *unbound* socket with the given address family and type (the codes for these are provided by the classes SocketAddress and UnixSocketAccessor as described earlier).

The remainder of the socket creation messages provide "canned" sockets (by analogy with canned food which comes pre-prepared, pre-cooked, and often predigested too).

- newTCP and newUDP return a *bound* socket in the Inet domain with type TCP or UDP (respectively). The socket is bound to the localhost with a randomly allocated port number. Since the port number is unknown, these sockets are most useful for clients since the port number is irrelevant to the process that initiates a connection.
- newTCPclientToHost:port: returns an Inet domain socket of type TCP *connected* to the given host (name) and port number. Communication on the socket can commence immediately.
- newTCPserverAtPort: and newUDPserverAtPort: return Inet domain sockets of type TCP or UDP (respectively) *bound* to the local host at the given port number. A "server loop" can use such a socket immediately by sending it listenFor: message and then looping to accept connections (for TCP sockets), or looping to receive incoming datagrams.

Note that the only way to create a Unix domain socket is to use the family:type: message.

Synchronisation

```
anySocket
    bytesForRead;

    isActive;

    readWait;
    writeWait;
    ioWait;

    shutdown: how;

    close.
```

INRIA ROCQUENCOURT

Synchronisation

- bytesForRead returns the number of bytes of data waiting to be read from the socket. This message can be sent to both TCP and UDP sockets. Note that the number returned for a TCP socket is the actual number of bytes that the application can read from the socket. The number returned for a UDP socket is more than the number bytes the application can read since the size of the message header is included in the count. (The size of this header, which the application never sees, is usually 16 bytes.)
- isActive returns true if the socket still exists.
- readWait waits until data is available for reading on the socket, writeWait waits until it is safe to write to the socket, and ioWait waits until one or other of these operations can be performed. (Sending bytesForRead to the socket is the easiest way to determine if ioWait returned because input is available.)
- shutdown: how tells a TCP socket that no more input, no more output, or no more of either will be performed. The argument is an integer:
  - ▷ 0 means that no more reads will be performed on the socket. A write at the remote end of the connection will fail, indicating that the connection has been shutdown in that direction.
  - ▷ 1 means that no more writes will be performed on the socket. Subsequent reads at the remote and will return EOF, indicating that the socket has been shut down in that direction.
  - ▷ 2 shuts the socket down in both directions.

  This message is important when a remote process is looking for an error condition in order to detect that the connection should be broken. It is also more dramatic (and faster) than a close, since pending data is discarded rather than flushed.
- close flushes any data not yet sent on the socket and then destroys it. Operating system and VM resources that were associated with the socket are freed. A socket should always be closed, even if it has already been shutdown, in order to free these resources.

Summary of the Story So Far

A socket is created in a particular address family (Inet domain or Unix domain), and with a particular type (UDP or TCP).

For UDP sockets:

- The socket is bound to an address in the corresponding family using bindTo:.
- Data is sent on the socket using sendTo:..., specifying the address of the recipient socket.
- Data is received from the socket using receiveFrom:..., with the address of the sender being placed in the first argument.

For TCP "client" sockets:

- A connection request is sent to a remote TCP socket using connectTo: with the address of the remote socket as the argument.
- Once connected, data is read and written to the remote socket using the low-level file read/write messages readInto:..., writeFrom:..., and writeAll:.

For TCP "server" sockets:

- A local address in the correct family is bound to the socket using bindTo:.
- The socket is told to start listening for connection requests by sending it listenFor: with the maximum backlog as the argument.
- Wait for and accept a connection using accept which returns a new, connected socket.
- Perform communication on the new, connected socket using readInto:..., writeFrom: or writeAll:

For all TCP sockets:

- Inform the peer that part or all of the connection is being terminated using shutdown:.

For all sockets:

- Close the socket when communication is over, using close.

A Small Example

```
| socket buffer nRead |
"Return the date and time, according to Unix."

socket := UnixSocketAccessor
              newTCPclientToHost: 'localhost'
              port: (IPSocketAddress
                 servicePortByName: 'daytime').
buffer := ByteArray new: 64.
[nRead := socket
              readWait;
              readInto: buffer.
] valueNowOrOnUnwindDo: [socket close].
^(buffer copyFrom: 1 to: nRead) asByteString
```

A Small Example: Connecting to a Unix Server

A small example is probably in order at this point, to illustrate some of the theory in practice.

The example shown in the slide is a very simple TCP client which connects to the Unix “daytime” service. This service accepts TCP connections, to which it writes the current date and time, and then closes the connection.

The service also provides a UDP service on the same port (there is no conflict since the address family is considered part of the socket address, so TCP and UDP services on the same port are actually at different addresses). In response to any datagram received on the port, the service sends a datagram back to the sender’s address containing the current date and time.

The TCP version is shown here, but the UDP version should be easy to deduce.

First a new TCP socket is created, connected to “localhost” at the port offering the daytime service. IPSocketAddress provide several naming methods, but the only one of general interest converts the name of a service into its port number.

A buffer is created to accept the incoming data. 64 bytes is plenty.

A readWait is performed, to allow the data from the service time to arrive at the socket. The data is then readInto: the buffer, with the actual size of the data saved for later.

The socket is closed, regardless of success or failure.

Ignoring the possibility of failure (which would be indicated by nRead still being set to nil), the reply in the buffer is converted into a String and returned.

Socket Options

```
aSocket
    getOptionsLevel: l name: n;
    setOptionsLevel: l name: n value: v.

UnixSocketAccessor
    solSocket;

    soLinger;
    soDontlinger;
    soKeepalive;
    soReuseaddr;
    soUseloopback.
```

Socket Options

Several useful options can be specified either on a socket-by-socket, or a message-by-message, basis. Socket options apply to the socket itself, and affect the way it is managed by the operating system. One or more socket options can be set by sending the socket the message setOptionsLevel:name:value:, where

- level is the "protocol level" at which the option is to be set. Usually the only level of interest to applications is the socket level. The code for this level is provided by the class UnixSocketAccessor in response to the solSocket message.
- name is the name of the option that is to be set. The following selection of messages corresponding the useful socket option names return the name of the option when sent to UnixSocketAccessor:
  - soLinger tells the socket to wait for pending data to be transmitted when it is closed.
  - soDontlinger the opposite of soLinger, provided for convenience only.
  - soKeepalive tells a connected socket to periodically "ping" the other end of the connection to make sure the connection stays "warm". This allows for early detection of lost connections.
  - soReuseaddr tells the binding mechanism to relax the rules governing the allocation of addresses. With this option set, a local address that appears to be already in use will be reallocated to the new socket.
  - soUseloopback tells the socket to avoid the network hardware, if possible, when talking to a socket on the same host.
- value is 1 to turn the option on, and 0 to turn it off.

The current value of an option can be read with getOptionsLevel:name:.

Message Options

aSocket
    sendTo: address buffer: buffer
    start: offset for: count
    flags: flags;

    receiveFrom: address buffer: buffer
    start: offset for: count
    flags: flags.

UnixSocketAccessor
    msgOob;
    msgPeek.

Message Options

Message options apply to individual messages (packets of data exchanged between sockets), and affect the way the message is delivered.

A message option is specified when sending or receiving a message using the messages:

```
socket sendTo: address buffer: buffer start: offset for: count flags: flags.
socket receiveFrom: address buffer: buffer start: offset for: count flags: flags.
```

where flags is constructed by "or"ing the results of sending one or more of the following messages to UnixSocketAccessor:

- msgOob means send or receive "out of band" data on the socket, rather than the normal "in-band" data. Out-of-band data can be read before in-band data that arrived before it on a socket. It is often used to signal urgent conditions in which the in-band data may need to be discarded. For example, the rlogin program uses out-of-band data to inform the remote host that the terminate interrupt key has been pressed. The amount of out-of-band data supported depends on the protocol being used and the local implementation of that protocol. It can be as little as one byte.
- msgPeek means read the data without removing it from the socket. A subsequent read will see the same data again.

These messages can be used on both connected TCP sockets and unconnected UDP sockets.

Streams

```
anExternalConnection := ExternalConnection
        inputAccessor: inputSocket
        outputAccessor: outputSocket.
```

```
anExternalConnection
    readStream;
    writeStream;
    readAppendStream.
```


Streams and Sockets

The methods for reading and writing sockets presented so far are somewhat clumsy, being little removed from their raw Unix counterparts. The fact that both sockets and files share a common ancestry in IOAccessor suggests that it should be possible to create Streams over sockets in the same way that it is possible to create Streams over files. This is in fact the case.

The class ExternalConnection provides the interface between Smalltalk streaming behaviour and external I/O objects represented by kinds of IOAccessor, or a subclass thereof. An ExternalConnection can be created on a socket, or on a pair of sockets:

```
ExternalConnection
    inputAccessor: inputSocket
    outputAccessor: outputSocket.
```

inputSocket and outputSocket can (and frequently will be) the same socket.

Armed with an ExternalConnection connected to a socket, we can create an ExternalStream by sending it

- readStream to make an input stream,
- writeStream to make an output stream, or
- readAppendStream to make a stream supporting both input and output.

ExternalConnections will appear several more times later on, when they are used to represent connections to Unix processes spawned from within Smalltalk.

Accessing the Unix Environment

CEnvironment, a Subclass of Dictionary, manages collections of environment variables.

The “current” environment is accessed using:

- **CEnvironment getenv: ’name’**
- **CEnvironment userEnvironment**

Converting a CEnvironment into Unix-format `'NAME=VALUE'` strings:

- **aCEnvironment asStrings**

Instances created with

- **CEnvironment fromStrings: aCollection**

INRIA ROCQUENCOURT

Accessing the Unix Environment

`ExtIPC.st` provides the class CEnvironment (a kind of Dictionary) that maps the name of an environment variable onto its value. This dictionary is re-initialised automatically each time the snapshot is run.

CEnvironment responds to the following accessing messages:

- getenv: key where key is the name of an environment variable (a String), returns the value (another String) of the given variable. If the variable is not defined, it returns an empty string.
- userEnvironment returns an instance of CEnvironment whose keys are the names of the environment variables in effect when the virtual machine was started. Both keys and values are Strings.

In addition to the usual Dictionary protocol, instances of CEnvironment respond to the message asStrings by returning an Array of Strings in the “Unix” format `NAME=VALUE`.

Instances of CEnvironment are created using the usual Dictionary protocol, or using the message fromSrings: aCollection which creates an environment from a collection of strings in the Unix format `NAME=VALUE`.

Creating or modifying instances of CEnvironment is necessary if a process is to be spawned with a modified set of environment variables.

Running a Process

Instances of UnixProcess represent spawned processes.

Instances created with the name of the program to execute.

Optional arguments:

- an Array of arguments to pass;
- a CEnvironment;
- an Array of UnixIOAccessor used for redirecting input and output.

Running a Process

`UnixIPC.st` provides the class UnixProcess, supporting the creation of, and interaction with, Unix processes.

Instances are created with the message

```
forkJob: name
arguments: args
environment: env
descriptors: fds
```

(or one of the three messages taking a subset of these arguments).

name is a String naming a program which must be found in the current `PATH`. This argument is always specified.

env is the CEnvironment to pass to the process. If no environment (or nil) is specified, the default environment is passed to the spawned process.

args is an Array of Strings which are passed to the program as arguments. The spawned process will find these in `argv` as if they had been typed on a command line.

fds is a three-element Array of UnixIOAccessors. (A UnixIOAccessor represents an object which Unix would normally attach to a file descriptor: files, sockets, pipes, etc.) These will be attached to the `stdin`, `stdout` and `stderr` of the process. If no array (or nil) is specified, the process will inherit the current values from the VM.

These messages are most appropriate for commands that return no useful output, and which require no input:

```
| process |
process := UnixProcess
    forkJob: '/usr/local/bin/gzip'
    arguments: #('-9' '/home/ikp/humungouslyBigFile').
```

Necessary Evils 1: `fork()`, `exec()`, and `pipe()`

New Unix processes are typically created in two stages.

First a process that wishes to create another process makes a verbatim copy of itself. This copy, or “child”, process is running the same program, at the same point, as the parent process. It shares all the static and heap data of the parent, including access to all open files, but has its own stack (a copy of the parent’s stack).

To create the copy, the parent calls `fork()`. After this call, two identical processes exist which tell themselves apart by inspecting their return value from `fork()` (the child gets `0`, whereas the parent gets the process identifier of the child).

The child then normally calls `exec()` to replace itself with another program. `exec()` replaces the program and the data area of the process with those for another program. Open file descriptors are not stored in the process data space, and are not affected by the call to `exec()`.

This last point is vitally important — it means that the child process which exists after both `fork()` and `exec()` have finished, will share all the open files of the original parent process. This fact is the basis for the interprocess communication facilities described below.

A *pipe* is a communication channel that appears to a process as a pair of open file descriptors.

A process creates a pipe using the `pipe()` system call, which returns two descriptors attached to either end of the new pipe. One of these descriptors is open for writing, and the other for reading. Data written on the writable descriptor can be read later from the readable descriptor. The data is buffered between the two descriptors, so that a write need not wait for the corresponding read to occur.

The distinction between a pipe and a connected socket is so slim that we will not discuss pipes in any more detail. (Pipes are actually implemented as a pair of back-to-back sockets, with a sponge in between.) For the purposes of what follows, assume they are the same thing.

Necessary Evils 2: Creating the IPC Channel

A parent process can combine the facilities on the previous slide to communicate with a spawned process through a pair of pipes.

First the parent creates two pipes. One pipe will be used to send data to the child process, and the other to receive data from the child.

The parent then `fork()`s to create a copy of itself. The child inherits the pipes created by the parent. In the parent, the read end of the output pipe is closed and the write end of the input pipe is closed—since these descriptors are only of interest to the child. The child closes the other ends of the pipes which are only of interest to the parent. We now have the situation shown in the lower half of the slide.

If appropriate, the child process can close its `stdin` and `stdout` and then re-open them using `dup()` to connect them to the appropriate ends of the pipes. The child can then finally call `exec()` to replace itself with the desired program. (This is not shown in the diagram—just imagine `p2` replaced by `p2′` with `fd1` and `fd2` replaced by `stdin` and `stdout`.)

Data written by the parent on the first pipe now appears as the standard input in the child, and written to standard output by the child appears on the other pipe in the parent.

The fine details in the text of these last two slides are not really important. Understanding the pictures is all that is really required.

Communicating with the Child

Creating an ExternalConnection

```
| connection process |
. . .
connection := UnixProcess
        pipeConnectionFor: '/bin/ls'
        arguments: #(-lF)
        setProcessDescriptor:
                [ :p | process := p ].
. . .
```

which can be sent:

- readStream
- writeStream
- readAppendStream

Communicating with the Child

Fortunately for everyone the details of creating the pair of pipes, connecting them to a spawned process, closing the redundant endpoints, and reconnecting the child pipes as `stdin`, `stdout` and `stderr` are already provided by the method:

```
pipeConnectionFor: jobName
arguments: args
setProcessDescriptor: pdBlock
```

where

- jobName (String) is the name of the program to execute;
- args is an Array of the argument Strings to pass to the program; and
- pdBlock is a Block taking one argument, which will be invoked from within the method with the instance of UnixProcess of the newly spawned process as the argument (required for synchronisation operations).

The value returned by pipeConnectionFor:... is an instance of ExternalConnection representing the parent's ends of the two pipes connected to `stdin` and `stdout` of the child process.

As was explained earlier, the most useful trait of an ExternalConnection is that it can can act as the data source and/or sink for a Stream. ExternalConnection will return a Stream-like object when sent one of the messages:

- readStream — returns a ReadStream that guarantees to return data, by waiting for the process to generate some output if there is none already available for reading.
- writeStream — returns a WriteStream that is used to write data to the standard input of the child process, and which will only write data to the "end" of the stream.
- readAppendStream — returns an object that behaves like both of the above.

Status and Synchronisation

- ExternalConnection synchronisation:
  - closeOutput
- UnixProcess status enquiries:
  - isActive
  - isCrashed
  - isExited
  - isOK
- UnixProcess synchronisation:
  - wait
  - kill: signalNumber

Status and Synchronisation

Unix processes often exit when they detect an `EOF` condition on their input. The way to create this condition is to send the message outputClose to the ExternalConnection returned by the pipeConnectionFor:... message. It is impossible to do this from a Stream created using the connection. (In the other direction, `EOF` on the standard output of the spawned process can be detected by sending atEndOfData to the associated ExternalStream.)

UnixProcesses can be queried for their current status. They return true or false in response to the following messages:

- isActive — the process is still running normally.
- isCrashed — the process has exited with a return value less than zero (a Unix convention indicating that the process exited due to an error).
- isExited — the process is no longer running.
- isOK — the process is still running normally, and is still accessible from within Smalltalk.

Instances of UnixProcess provide two methods for synchronisation.

- wait blocks the Smalltalk parent process until the child process has exited.
- kill: signalNumber sends a signal to the child process.

A Smalltalk program can send wait to a child process to allow the child to finish before (for example) closing down any associated streams. (Note that it is the parent process that waits — the child process continues to run normally.)

The meaning of signalNumber is operating-system specific. Under (most) Unix systems the valid signal numbers and their meanings are defined in the C header file `/usr/include/signal.h`. The more useful signals are:

- 3 (`SIGQUIT`) terminates the child process gracefully.
- 9 (`SIGKILL`) terminates the child process mercilessly.
- 17 (`SIGSTOP`) suspends the child process.
- 19 (`SIGCONT`) resumes a child that was previously stopped with `SIGSTOP`.

Simple Example

```
| stream proc reply connection |
connection := UnixProcess
            pipeConnectionFor: '/bin/ls'
            arguments: #('-a' '-l')
            setProcessDescriptor:
                          [:pd | proc := pd].
stream := connection readStream.
[reply := iostream contents.
proc wait]
      valueNowOrOnUnwindDo:
                          [stream close].
^reply
```


Simple Example: Running a C Program

First we spawn a process running the command `/bin/ls -a -l`:

```
| cmd proc reply connection |
connection := UnixProcess
          pipeConnectionFor: '/bin/ls'
          arguments: #('-a' '-l')
          setProcessDescriptor: [:pd | proc := pd].
```

We must keep the ExternalConnection returned from the pipeConnectionFor... message in order to retrieve the output generated by the process. We must also save the process descriptor (the instance of UnixProcess itself) so that we can discover when the process has terminated.

In order to read the output from the process, we turn the ExternalConnection into a readStream:

```
cmd := connection readStream.
```

Next we read the output generated by the `ls` command (the read message will block until the data is available) and wait for the process to die. We also ensure that the stream connected with the process is closed, regardless of whether the command executes successfully or is aborted due to an error:

```
[reply := cmd contents.
proc wait]
    valueNowOrOnUnwindDo: [cmd close].
```

Finally we can return the output generated by the command:

```
^reply
```

UnixProcess: Class Examples

UnixProcess class examples:

- **cshBullet: commandString**
- **cshOne: commandString**

These rely on the property of `csh` that it will exit after reading and executing one command. For more general commands:

```
connection := UnixProcess
    pipeConnectionFor: programName
    arguments: argumentArray
    setProcessDescriptor: [:pd | proc := pd].
```

When all input has been sent to the process, generate `EOF` with:

```
connection closeOutput.
```

UnixProcess: Class Examples

For some applications, it will suffice to simply run a process and possibly gather the output it generates. The class examples in UnixProcess include two methods for executing a program or script under `csh`. The command name and arguments are passed in a single string, to be parsed by the spawned `csh`:

```
UnixProcess cshBullet: 'gs /tmp/output.ps'.
UnixProcess cshOne: '/bin/find $HOME -name ''*.st'' -print'.
```

cshBullet: is appropriate for processes that do not generate any useful output, whereas cshOne: returns the output from the process as a String.

These methods are good starting points for experimentation and useful behaviour can be obtained by modifying them slightly. For example, from cshOne:

```
UnixProcess≫cshOne: string
    connection := self pipeConnectionFor: 'csh'
                        arguments: #('-ft')
                        setProcessDescriptor: [:pd | proc := pd].
    . . .
    [cmd nextPutAll: string; cr; commit.
    . . .]
```

could be modified to run any given program:

```
UnixProcess≫execute: programName arguments: argArray input: aString
    connection := self pipeConnectionFor: programName
                        arguments: argArray
                        setProcessDescriptor: [:pd | proc := pd].
    . . .
    [cmd nextPutAll: aString; commit.
    connection closeOutput.
    . . .]
```

(The connection closeOutput creates the required `EOF` condition on the `stdin` of the spawned process.)

Example 1

Filter = `stdin` → modify → `stdout`, e.g:

> `cat`, `head`, `tail`, `more`, `nl`, `sort`,
> `uniq`, `tr`, `sed`, `awk`, `spell`
> `[tng]roff`, `/lib/cpp`, ...

Tremendous utility when used in pipelines:

```
cat file | tr '[ ]' '[\012]' |
        sort | uniq -c
```

- Smalltalk execution server, using sockets to communicate;
- simple C client connects to server, starts command, connects `stdin` and `stdout`;
- one-line shell script hides client program and Smalltalk command.

Example 1: A Smalltalk Execution Server

A very useful facility which could be provided using a client-server model, based entirely on the intrinsic features of Smalltalk, would be to allow Smalltalk expressions to be evaluated from a C process in a context where the standard input and output of the Unix process can be read and written. Such a facility would make it possible to start Smalltalk processes (in an already running image) as filters from a Unix shell prompt, or even to specify a pipeline (a sequence of Unix commands connected "head to tail" by their standard input and output) in which some of the commands are Smalltalk processes.

For example, in the command

```
unix$ cat inFile | tokenise | sort -r | capitalise | tee outFile
```

the `cat`, `sort` and `tee` would be the normal Unix commands, whereas `tokenise` and `capitalise` might be implemented in Smalltalk.

The model we will use is as follows:

- A Smalltalk server process, running at a slightly higher priority than user-level processes, waits for connection requests. When a client connects, a Smalltalk expression is read from the client, an ExternalStream is created connected to read from the `stdin` and write to the `stdout` of the client, and then the command is executed in a context where the stream can be named.
- A trivial C program is written to perform the client operations. It connects to the Smalltalk server, passes the command-line arguments to the Smalltalk server as the expression to evaluate, and then ships data between the the connection and standard input/output until the connection is closed by Smalltalk.
- The behaviour described in the example above can then be achieved with a one-line shell script that `exec`s the client program with a suitable Smalltalk expression as its arguments.

The C Client

- create socket connection to Smalltalk
- send command line
- copy `stdin` to, `stdout` from, socket
- close socket and exit when `EOF` on socket

```
to_from_duplex(source, sink, sock) {
  FD_SET(source, &openfds);
  FD_SET(sock, &openfds);
  while (FD_ISSET(sock, &openfds)) {
    readfds= openfds;
    select(NOFILE, &readfds, 0, 0);
    if (FD_ISSET(0, &readfds) &&
        ship(source, sock))
      FD_CLR(source, &openfds),
      shutdown(sock, 1); /* EOF on stdin */
    if (FD_ISSET(sock, &readfds) &&
        ship(sock, sink))
      FD_CLR(sock, &openfds),
      shutdown(sock, 2); /* socket closed */
  }}
```

The C Client

A complete listing of the C client is given in Appendix B.

The `main()` routine performs four tasks:

- It creates a TCP connection to the Smalltalk server using the function `tcp_connect`.
- It ensures the arguments (passed in `argv[]`) are terminated with a null pointer, and then calls `sendstrings()` which flattens the argument vector into a single string, terminates it with a newline, and sends it to the Smalltalk server.
- It copies data from `stdin` to the input side of the connection, and copies data from the output side of the connection to `stout`. This is done by the function `to_from_duplex()` which returns when an `EOF` is detected on the output side of the Smalltalk connection (indicating that Smalltalk has shut down its end of the connection).
- It closes the connection and exits with a “success” code.

The only point worth noting is that when the `to_from_duplex()` function detects and `EOF` on `stdin` it performs a `shutdown()` on the *input side only* of the connection. This signals to Smalltalk that it should not expect any more input from the connection. (From Smalltalk’s point of view, the ExternalConnection will return false in response to isActive after the `shutdown()` has been performed by the client, which is required if any `ExternalStream` created from it to answer true to atEndOfData. Since the implementation of contents—and similar messages for reading data from an ExternalStream—call atEndOfData, if the `shutdown()` were not performed then Smalltalk would be waiting forever for more data to arrive.)

The remaining details of the client program are straightforward, are explained adequately in the Unix online manuals, and are consequently not explained here.

Server Anatomy

ExampleServer class is the server itself, executing in its own process.

- server loop
- server process control
- snapshot handling

ExampleServer instances are individual client requests, executing in dedicated processes.

- stream connection to "`stdio`"
- error trapping mechanisms
- client code

INRIA ROCQUENCOURT

Server Anatomy

The server itself is represented by the class ExampleServer. This class implements the server loop which executes in a server process, and implements several management operations related to creating, suspending, resuming, and destroying processes.

Instances of ExampleServer represent individual client requests, each of which executes in its own process.

Each client request must execute in a context where the input and output from the socket are available. To make this possible, instances of ExampleServer have an instance variable stdio which hold a Stream created from the client's socket. To perform the client's request the server instantiates itself, connects the stream to stdio, and then asks the instance to spawn a process executing the client code. The client code itself is a method on the instance side of ExampleServer.

The client methods will be executed in an environment in which errors are benign to the rest of the Smalltalk image—execution errors, syntax errors, and so on are trapped and reported to the client rather than disrupting the work of whoever is working at the image.

The ExampleServer class also ensures that the server process is recreated correctly whenever the image is saved and then restarted at a later date.

In summary, on the class side of ExampleServer we have:

- the server loop, executing in the server process
- server and client process control
- snapshot handling

and on the instance side:

- `stdio` Stream
- error handling code
- client methods

The following slides present the more important methods in the implementation. Many trivial details are omitted. A complete listing of the server can be found in Appendix A.

Class Initialisation

ExampleServer class≫initialize

```
ServerProcess notNil ifTrue: [self killServer].
ClientProcesses isNil
    ifTrue: [ClientProcesses := IdentitySet new]
    ifFalse: [self killClientProcesses].
(ObjectMemory dependents includes: self)
    ifFalse: [ObjectMemory addDependent: self]
```

Class Initialisation

We will start the tour of the Smalltalk server in the class initialisation method.

Two class variables are necessary. ServerProcess will contain the process running the server loop. This is necessary so that the server can be started, suspended, resumed and stopped as necessary. An invariant associated with this variable is that it contains nil when there is no server running, and the server process when the server is running. If the class is being re-initialised there may already be a server running; if so, we kill it first (setting ServerProcess to nil as a side effect).

ClientProcesses is an IdentitySet containing the active client processes. If we are re-initialising the class we need to kill all existing client processes.

Finally, the ExampleServer class registers itself as a dependent of ObjectMemory. In doing this, it ensures that it will receive notification of various memory-related events, including return from snapshot, using the change mechanism.

Process Control 1

ExampleServer≫startServer

```
ServerProcess isNil ifFalse:
  [self error: 'a server process already exists'].
ServerProcess :=
    [self serverLoop] newProcess.
ServerProcess priority:
    Processor userInterruptPriority.
ServerProcess resume.
```

ExampleServer≫killServer

```
ServerProcess isNil ifTrue:
  [self error: 'there is no server process to kill'].
ServerProcess terminate.
ServerProcess := nil.
```

Process Control 1: The Server

Careful management of all processes is essential. Apart from any other considerations, when developing an application that makes use of processes it is almost inevitable that early prototypes will contain bugs that can cause the application to lose contact with a process. It can be difficult to locate and destroy such processes unless steps are taken to manage them correctly.

Before starting a new server process it is necessary to check that a server process does not already exist. Duplicating the server process could cause severe problems if two server loops were trying to listen on the same socket.

If no server is currently running, a new process is created to launch the server loop.

The priority of the process is set slightly higher than that of normal user processes. This ensures that it will run immediately when a connection request arrives, briefly interrupting the normal user processes. (The server process will spend most of its time waiting on a Semaphore associated with the server's socket.)

New processes are created in a suspended state, so the last thing we need to do is resume the server process in order to launch the server loop.

Killing the server process is straightforward. The killServer method is careful to preserve the invariant on the ServerProcess variable by setting it to nil when no server process is running.

A further two methods, suspendServer and resumeServer, complete the set. These are similar to killServer, with the obvious differences.

Process Control 2

```
ExampleServer class≫killClientProcesses
    ClientProcesses do: [ :process |
        process terminate.
        ClientProcesses remove: process]
```

Process Control 2: The Clients

The same considerations that applied to the server process also apply to each of the client processes that will execute the incoming requests.

The set of active clients is kept in ClientProcesses. A similar invariant is maintained here too: when a client process is created it is immediately added to this set. When killing client processes, each client is removed from this at the instant the process is terminated.

Dealing with Snapshots

ExampleServer class≫
update: aspect with: parameter from: sender

```
(sender == ObjectMemory and:
 [aspect == #returnFromSnapshot])
     ifTrue: [self returnFromSnapshot]
```

ExampleServer class≫returnFromSnapshot

```
self killClientProcesses.
ServerProcess isNil ifFalse:
    [self killServer; startServer]
```

Dealing with Snapshots

Snapshots pose a problem. Sockets, and all other kinds of open descriptor that do not take special action, will not survive a snapshot. ("Snapshot" implies here that the virtual machine stops and is restarted at a later date; just saving the image will obviously not affect sockets or other kinds of open descriptor.)

It might be possible to add a general mechanism to UnixSocketAccessor such that sockets survive snapshots, but it would involve a disproportionate amount of effort. Sockets also differ from files in that they have a lot of associated transient state, and also in that closing one end of a socket connection usually causes the other end to close down.

For these reasons it is better to simply kill both the server process and all client processes when returning from a snapshot. The server because it is very difficult to recreate the exact state of the original socket without significant modifications to the UnixSocketAccessor implementation, and the clients because the far ends of their connected sockets will have long since been closed.

The class ExampleServer registered itself as a dependent of ObjectMemory during class initialisation. On returning from a snapshot, ObjecMemory sends its dependents a changed: message with the aspect #returnFromSnapshot. Checking for this aspect, and that the sender was indeed ObjectMemory, indicates to ExampleServer that it should terminate or recreate all of the processes that own a socket.

The Server Loop

```
| socket |
socket := UnixSocketAccessor
        family: (SocketAddress
                domainCodeFromName: #afInet)
        type: UnixSocketAccessor sockStream.
[       socket
            setOptionsLevel: UnixSocketAccessor solSocket
                name: UnixSocketAccessor soReuseaddr
                value: 1;
            bindTo: (IPSocketAddress
                hostAddress: IPSocketAddress thisHost
                port: self serverPort)
] valueOnUnwindDo:
        [socket close.
        self error: 'failed to create server socket'].

[       socket listenFor: self maxConnections.
        [true] whileTrue: "the loop..."
                [self dispatchClientOn: socket accept]
] valueNowOrOnUnwindDo: [socket close]
```

The Server Loop

First we create a socket on which the server will listen for connections. This socket is not immediately usable, since it has not been associated with an address. Sockets are not created with particular addresses since different address families can require different components to specify an address. We could have created the socket using (for example) newTCPserverAtPort: and immediately entered the server loop, but that would not have permitted the setting of any options on the socket.

Before associating the address with the socket, we choose to set the socket option soReuseaddr which indicates that the socket should not complain if we try to bind an address to it which is already being used. Since the Unix data structures associated with sockets are only "garbage collected" periodically, this avoids an annoying wait when restarting the server after a crash.

Next we bind the address to the socket, using thisHost as the IP address and self serverPort as the port number. The port number is chosen more or less arbitrarily, and is accessed via a method to help make the server easier to configure.

Either setting the option on, or binding the address to, the socket could cause a failure. These operations are therefore wrapped in a block whose unwind (*only*) action is to close the socket (since sockets are represented as open files, they are a finite resource to each process) and report an error.

Connections are then enabled on the socket using listenFor:, with 1 pending connection request allowed.

The server loop is infinite, repeatedly accepting connections on the server socket. Again, since sockets are a finite resource, it is important to close the server socket should any problem arise from within the server loop.

The client requests is dealt with by the server's method dispatchClientOn: which takes the newly-connected client socket as its argument.

Client Dispatch

```
ExampleServer class≫
dispatchClientOn: clientSocket
    | stdio command client process |
    stdio := clientSocket readAppendStream.
    command := stdio through: Character cr.
    client := self newClientStdio: stdio.
    ClientProcesses add: (process :=
        [
            [   self evaluateSafely: command
                    for: client.
                stdio flush; close.
            ] valueNowOrOnUnwindDo:
                [ClientProcesses remove: process.
                clientSocket close]
        ] newProcess).
    process priority: Processor userSchedulingPriority.
    process resume.
```

Client Dispatch

The server loop accepts incoming connection requests, therefore causing a client socket to be created for subsequent communication, and then calls dispatchClientOn: aSocket in order to create the process and context in which the client request will be executed.

As noted earlier, any ExternalConnection can be used to create a Stream to make input/output more conducive to Smalltalk paradigms. Hence the first thing that we do is create a stream based on the client socket—this stream is the Smalltalk equivalent of `stdin` and `stdout` for the client process.

Our protocol requires that the Smalltalk expression to be executed is sent to the server as the first line of data passed from the client. Using the newly created stream, the command is read using through:.

While they are active, clients are represented as instances of ExampleServer. The message newClientStdio: creates the instance for this client, and sets the instance variable stdio from the argument—being visible in the client code on the Smalltalk side, this stream provides easy exchange data with the C side of the client.

Having created the ExampleServer instance to represent the client, we create a process in which to execute the client's code. This process will perform four actions (when it is eventually resumed from its initial suspended state):

- The Smalltalk expression sent from the C side of the client is evaluated safely by calling evaluateSafely:for:. The first argument is a String containing the expression to evaluate, and the second is an object providing the context for the evaluation—in this case, the instance of ExampleServer representing the client.
- When the expression finishes evaluation the stdio stream is flushed and closed, ensuring that all data written to it is sent to the C side of the client.
- Regardless of any errors that might have occurred, the client process is removed from the server's ClientProcesses set and the associated socket is closed.

Finally, the newly created process is given the regular userSchedulingPriority and then resumed in order to launch the client.

Safe Evaluation

```
ExampleServer class≫
evaluateSafely: command for: client
    | unique |
    unique := 'unique'.
    ^Smalltalk errorSignal
        handle: [ :ex || message |
            message := ex originator == unique
                ifTrue: [
                    '*** syntax error']
                ifFalse: [
                    '*** ', ex errorString, Character cr asSymbol,
                    '*** for receiver: ', ex originator printString].
            client stdio nextPutAll: message; cr; flush.
            ex returnWith: '']
        do: [Compiler new
                evaluate: 'self ' , command
                in: nil
                to: client
                notifying: unique
                ifFail: nil]
```

Safe Evaluation

The evaluation of the expression received from the C side of the client is made by sending an evaluate:... message to the Compiler. This would normally cause notifiers to appear in the case of syntax or runtime errors.

Some small measures can be taken to lessen the risk of such events troubling the "regular" user of the Smalltalk image hosting the server, and at the same time providing useful feedback to the C side of the client about error conditions.

The heart of the evaluateSafely:for: method is a call to the Compiler. In addition to the expression and object providing the context, we pass a unique object as the notifying: argument (this object can be any unique object not implementing any of the CompilerErrorHandler messages). All this happens inside a handler for Smalltalk errorSignal, which will catch every possible exception. Two things can go wrong during this call:

- A runtime error can occur during the evaluation of the expression. In this case an error will be signaled, and caught by our handler. The receiver is unlikely to be our unique object, so the exception object is queried for the error message and the receiver of the message that caused the exception. This information is sent back to the C side of the client via the stdio stream.
- The compilation of the expression can fail, because of a syntax error. In this case, the object specified in the notification argument will be informed of the error. Our unique object happens to be a String, which does not implement any of the messages that a CompilerErrorHandler would understand, causing a 'doesNotUnderstand' exception to be raised, triggering our exception handler with an exception that has our unique object as the originator. By comparing the originator against our unique object we can therefore tell syntax errors from other runtime errors without having to go to the trouble of implementing a specialised class of CompilerErrorHandler. (Although it must be admitted that implementing a compiler error handler would be the correct way to do this, and would also allow meaningful syntax error messages to be sent back to the client.)

An Example Client Function

ExampleServer≫capitalise: aStream

aStream nextPutAll: aStream contents asUppercase

called with:

```
stclient capitalise: stdio
```

Alternatively, the client function can operate directly on stdio:

ExampleServer≫capitalise

stdio nextPutAll: stdio contents asUppercase

called with simply:

```
stclient capitalise
```

Which you prefer is a matter of taste.

An Example Client Function

On the Smalltalk side, client functions can be written in one of two styles.

They can be written so as to take an argument, aStream, which they use to read and write input and output. Methods using this style can be used (or tested) from within Smalltalk as easily as they can be used via the server. The disadvantage is that the C side of the client must have explicit knowledge of the server implementation, specifically the name of the ExampleServer instance variable holding the input/output stream.

The alternative is to put this knowledge in the Smalltalk client methods, making the choice of name transparent to the C side. The disadvantage is that the methods are no longer useful to other Smalltalk code, and are more difficult to test.

The final step in the creation of our "filter" is to create a one-line shell script that hides the C side of the Smalltalk client. In our capitalisation example, a shell script called `capitalise` which contained the single command

```
#!/bin/sh
exec stclient capitalise: stdio
```

can be used to start a Smalltalk process in a Unix pipeline as if it were a normal Unix filter program.

One final comment can be made. The evaluateSafely:for: method appends the string 'self ' to every expression it evaluates. This ensures that clients can only execute the methods directly implemented by instances of ExampleServer. By removing this string the server becomes completely general purpose, accepting any Smalltalk expression. For example, the following would be possible:

```
stclient stdio nextPutAll: 100 factorial printString
```

or even, for the more adventurous:

```
stclient Browser browseAllCallsOn: '#evaluateSafely:for:'
```

Summary

- IPC using sockets.
- Socket types and address families.
- Socket communication and synchronisation.
- The Unix environment.
- Unix process creation, IPC, synchronisation.
- Executing programs from Smalltalk, gathering results.
- Dealing with snapshots.
- Typical server anatomy, server/client process management.
- Basics of C clients.
- Error containment.

Summary of Part 1

This section introduced most of the practically useful features of the built-in Smalltalk facilities for controlling Unix processes and socket-based communication:

- How IPC works using sockets.
- Socket concepts: UDP and TCP types of socket, Unix and Inet address families.
- Communicating with sockets using read/write, synchronisation requirements.
- Using streams over sockets.
- Accessing the C environment variables, and creating sets of environment variables to pass to spawned processes.
- The fundamentals of Unix process creation, and how IPC works to a spawned process. Synchronisation with the spawned process, interrogating its state, and detecting termination.
- Creating a method which executes a Unix process and returns any data written to `stdout`.
- Accessing `stdin` and `stdout` as a Stream. Creating and detecting EOF conditions on input and output streams.
- Using the changed mechanism to take appropriate action after a snapshot.
- Creating a server socket-based server in Smalltalk. Managing server and client processes. Detecting the termination of a client Unix process, and informing a client Unix process of the termination of a service.
- A general overview of the C side of a socket-based client program. Mechanisms for synchronisation with the server.
- A couple of random, ad-hoc mechanisms for dealing with errors inside the Smalltalk side of a client process.

Part 2 — UDP

- Adding Primitives to the VM
- Corresponding Primitive Methods
- Returning from a Primitive
- Data Types
- Conversion Between Smalltalk and C
- Type Checking
- Allocating Objects
- Accessing Object State
- Error Handling
- Long-Lived Objects
- Asynchronous Events
- Unsafe Primitives

INRIA ROCQUENCOURT

Part 2 — User Defined Primitives

User defined primitives (UDPs) provide a means for software written in C (or C++) to be made available to Smalltalk programs. As the name suggests, a routine written in C appears to Smalltalk as a primitive. The behaviour is the same as for "normal" primitives — in particular, the primitive has the opportunity to fail and let Smalltalk code take over.

General Principles

- C code compiled with `userprim.h` into a `.o` file.
- New VM created by linking `.o` file with `st80.o`.
- Primitive methods defined in Smalltalk which invoke corresponding C routines.

INRIA ROCQUENCOURT

General Principles

In general, 3 steps are required to make a user-defined primitive available to Smalltalk code:

- The C code is compiled (into a ".o" file) using the `userprim.h` header file, which provides UDP support routines and definitions for:
  - registering the C routines with the VM;
  - sharing data between Smalltalk and C;
  - creating new Smalltalk objects;
  - error and event handling.
- The ".o" file is linked with the Smalltalk virtual machine, which is supplied in a `st80.o` format. The resulting virtual machine is identical to the normal Smalltalk virtual machine, except that it also contains the code implementing the new user-defined primitives. It is also possible to customize the "banner" string, to identify the version of the virtual machine at startup.
- Within Smalltalk, new primitive methods must be defined which are connected to the corresponding C routines. When these primitive methods are called, the user-supplied C code is invoked. Failcase Smalltalk code can be supplied, which is run if the primitive should fail for any reason.

Initialisation

```
char *versionString=
  "Smalltalk 4.1 with FFT UDPs";

char *resourceClassName=
  "St80";

extern char *environ[];

main(argc, argv)
char **argv;
{
  /* ...early initialisation... */

  /* ...options processing... */

  st80Main(argc, argv, (char *)0);
  /* NOTREACHED */
}
```

Initialisation and the User Environment

Two variables *must* be defined in the user-supplied code. `versionString` points to a banner string which will be printed by the virtual machine when it starts, identifying the VM version and the primitives that have been added. `resourceClassName` points to the string which will be used by the virtual machine when looking up X Window System resources. This can be changed so that different resources can be made to apply to different customised versions of the virtual machine.

The user supplied C code implementing the primitives is responsible for providing a `main()` routine (this is not defined in the `st80.o` file itself). The VM itself is started by calling the function `st80Main()`, passing a possibly modified `argc` and `argv`. (The third argument, `(char *)0`, appears to be unused—it is most likely an anachronism, corresponding to `envp`, left over from a time when the VM provided the `main()` routine itself.)

This allows the user-supplied code to perform any early initialisation that might be needed to support the primitives, before the virtual machine takes control. (Initialisation that can be delayed until the function `UPinstall()`, described below, should probably not be performed in `main()`.) Customised options processing can be performed, modifying `argc` and `argv` as appropriate to remove application-specific options, before starting the virtual machine.

The environment can be modified if required. The `st80Main` routine ignores its third argument and reads the environment from the external variable `char *environ[]` instead. Changes made to this variable, or modifications to the contents of the structure to which it points, will effect the user environment information returned by the class CEnvironment.

Adding Primitives

```
char *UPinstall()
{

  /* ...late initialisation ... */

  UPaddPrimitive(10001,
                 primFFTfunc,
                 7);
  /* and so on...
     ...for each primitive */

  return "id string";
}
```

INRIA ROCQUENCOURT

Adding Primitives to the Virtual Machine

The user-supplied C code must declare the existence of the new primitives to the virtual machine. When the VM starts (after the user-supplied `main()` routine has called `st80Main()`) it calls the user-supplied routine `UPinstall()` which installs each new primitive in the VM.

The `UPinstall()` procedure calls

```
UPaddPrimitive(primNumber, primFunction, argumentCount);
```

for each new primitive that it wants to register with the virtual machine. The arguments are as follows:

- `primNumber` is the number which will appear in the corresponding Smalltalk primitive method definition. By convention, user-defined primitives have numbers in the range 10000 to 19999 — implying a limit of 10000 user-defined primitives (which should be sufficient for most applications).
- `primFunction` is the C function to execute as the body of this primitive; it must have the type `void(*primFunc)()`.
- `argumentCount` specifies the number of arguments that should be passed to the `primFunc` function. This number does *not* include the receiver.

Calls to `UPaddPrimitive()` can be made in any order, and the primitive numbers need not be contiguous. The number of arguments declared for the primitive in the call to `UPaddPrimitive` must agree with the number actually passed to the corresponding Smalltalk primitive method. The VM will fail the primitive before it even begins if these two argument counts do not agree. It is the programmer's responsibility to ensure that the argument count in the C primitive function accepts the correct number of arguments—there is no way that the UDP interface can perform such a check automatically.

The `UPinstall()` function is expected to return a string identifying the user primitives. This appears to be ignored by the virtual machine, and is probably also an anachronism left over from the days when the `versionString` variable was not provided by user code.

C Primitive Definitions

```
void primFunc(rcv, arg1, arg2, ...)
upHandle rcv, arg1, arg2, ...;
{
  /* implementation of primitive */
}
```

C Primitive Definitions

The C part of each UDP is defined as a "function returning `void`". This function accepts one or more arguments, all of which are of type `upHandle`—a type defined in the header file `userprim.h` representing (the handle of) an object pointer.

The receiver of the primitive is always the first argument. Any additional arguments passed to the Smalltalk primitive method are passed after the receiver. As noted earlier, if the number of arguments declared in the call to `UPaddPrimitive()` and the number actually passed to the corresponding Smalltalk primitive method do not agree, the primitive will fail immediately.

Object pointers are not passed directly into C code since there are situations in which an object may be moved, due to garbage collection activity for example, during the call to the primitive. Instead an opaque handle is passed which is guaranteed to refer to the same object during the call to the primitive, regardless of how the virtual machine might rearrange the object memory. Handles cannot be converted into C pointers, and are only valid during a single UDP call. They most certainly can not be stored between primitive calls.

The maximum number of arguments that can be passed to a user primitive is 32.

A Simple Example

```
#include "userprim.h"
/* versionString... */

void primAdd(rcv, arg)
upHandle rcv, arg;
{
  upHandle result;
  /* ... result= rcv+arg ... */
  UPreturnHandle(result);
}

char *UPinstall()
{
  UPaddPrimitive(10001,
                 primAdd,
                 1);
  return "addition primitive";
}

/* main()... */
```

INRIA ROCQUENCOURT

A Simple Example

Putting all the above together, a simple example would look something like this:

```
#include "userprim.h"

char *versionString= "St80 4.1 with addition";
char *resourceClassName= "St80";

void primitiveAddFunction(receiver, argument)
upHandle receiver, argument;
{
    upHandle resultHandle;
    /* ... code to add receiver and argument ... */
    UPreturnHandle(resultHandle);
}

char *UPinstall()
{
    UPaddPrimitive(10001, primitiveAddFunction, 1);
    return "primitive addition";
}

main(argc, argv)
char **argv;
{
  st80Main(argc, argv, (char *)0);
}
```

This code is complied and linked with the virtual machine. (On Unix systems there is a `makefile` provided in the distribution to do this with a minimum of fuss.)

The resulting VM can be used to execute a Smalltalk image in the normal way.

Smalltalk Primitive Definition

SmallInteger≫myAdd: anotherInteger

```
"Add two SmallIntegers in a rather
roundabout way..."

<primitive: 10001>
"failcase code: what went wrong?"
^(anotherInteger isKindOf: Number)
  ifTrue:
    [self + anotherInteger]
  ifFalse:
    [self error: 'whatever this is, I can''t add it!']

<primitive: 10001 errorCode: errVal>
```

Smalltalk Primitive Definition

For each new primitive, a Smalltalk method must be defined through which the primitive is invoked. The syntax for this method is exactly the same as that for Smalltalk's "built-in" primitives.

Continuing with the example in the previous slide, we would define a new primitive method in class SmallInteger accepting one argument. The primitive number declared in this method agrees with the primitive number used to install the corresponding C function. When the Smalltalk method is called, the C function is invoked.

If the C code for the UDP decides that it cannot handle the argument that it was passed (the receiver will presumably always be a SmallInteger, but no such assumption can be made for the argument) then it can fail (by calling `UPfail()`) and let the Smalltalk failcase code try to recover.

For example, suppose that the class of anotherInteger is actually LargePositiveInteger. The C function discovers that the argument is not a SmallInteger, and fails the primitive. The Smalltalk failcase code will be invoked, and it can try to discover why the primitive failed. In this example, even though the primitive declines to handle the addition, the argument is a kind of Number—so the Smalltalk failcase code can complete the addition on behalf of the primitive.

Although rather contrived, this example illustrates a general approach that cam be applied to any user-defined primitive.

Calling a primitive that has not been installed will cause the primitive to fail.

One undocumented feature should be mentioned. If a Smalltalk primitive definition has a second part containing errorCode: followed by a variable name, that variable will be made a temporary of the failcase code. In the case of primitive failure it will be initialised with the error code returned by the primitive before the failcase code is entered. *Since this is an undocumented feature it may, of course, vanish in future releases!*

Returning

Successful return, with a value:

```
void UPreturnHandle(upHandle);
void UPreturnNil();
void UPreturnTrue();
void UPreturnFalse();
```

Failure return, with an error code:

```
void UPfail(upInt);
```

Returning from a User Defined Primitive

A UDP can return in one of two ways. It can either return normally, providing a value as the result of the Smalltalk primitive invocation, or the primitive can fail.

A primitive succeeds by calling the routine `UPreturnHandle()` whose argument is the handle of the value to return. This routine never returns to the caller (the C function), but rather it returns back to the Smalltalk method that invoked the primitive.

Alternatively, the C routine can return the Smalltalk value nil by calling `UPreturnNil()`. Similarly named functions are provided for returning for true and false.

A primitive fails by calling the routine `UPfail()` whose argument is an integer describing the reason for failure. The failcase code in the corresponding Smalltalk primitive method definition can retrieve this code and use it to determine why the primitive failed.

The Smalltalk failcase code invokes primitive number 9999 (which has to be defined manually in any class that needs it) in order to retrieve the last failure code returned from a user primitive. Failure code 0 must never be used, since this indicates "no error".

Failure codes have symbolic names, defined in `up_defs.h` (which is included automatically by `userprim.h`), all of which begin with `UPE`. Particular error codes will be introduced below, as and when necessary.

Note that all of these routines return to Smalltalk, and not to the calling C routine.

Data Types

C typedef name	typical definition	Smalltalk equivalent
upHandle	*opaque*	*oop*
upBool	int (0 or 1)	true or false
upByte	unsigned char	*byte-type*
upChar	char	Character
upInt	int	SmallInteger
upFloat	float	Float
upDouble	double	Double

Data Types

The UDP interface defines seven data types for use in user-defined primitives.

- `upHandle` is an arbitrary object reference. The type is opaque, and the C code should not try to interpret the value of an `upHandle` which can even change depending on the flags used during the compilation of user primitives.
- `upBool` is the C type corresponding to one of the Smalltalk objects true or false, almost always the integer `1` or `0`. (This type differs from `upInt` by convention only.)
- `upByte` is whatever C type corresponds to the contents of a ByteArray. This type is used only for operations involving ByteArrays.
- `upChar` is the C type corresponding to a Smalltalk Character.
- `upInt` is the C type corresponding to a Smalltalk `Integer`. However, not all Integers can be represented as an `upInt`. Normally every SmallInteger value can be represented as an `int`.
- `upFloat` is the C type corresponding to a Smalltalk single-precision Float, and `upDouble` to a Smalltalk `float`.

These seven defined types should always be used, even when it seems obvious what their equivalent C type might be, since architectural differences will often affect their definitions. For example, on 64-bit architectures, and depending on local C compiler conventions, the type `upInt` may not be defined as an `int` at all.

Conversions 1

```
upFloat UPSTtoCfloat(upHandle);
upHandle UPCtoSTfloat(upFloat);
```

Similarly for:

- integers
- doubles
- booleans
- characters

```
upHandle UPnil();
```

```
upHandle UPCtoSTbool((upBool)0);
upHandle UPCtoSTbool((upBool)1);
```

Conversion Routines 1: Single Values

Many conversion routines are provided for copying data from Smalltalk to C, and from C to Smalltalk. Conversion routines that translate from Smalltalk to C have names beginning with `UPSTtoC`, and routines that translate from C to Smalltalk begin with `UPCtoST`.

When converting a single data value from Smalltalk to C, the equivalent C value is returned directly by the conversion routine.

When converting a single C value to Smalltalk, a new Smalltalk object of an appropriate class is instantiated, and a handle to it returned.

Similar routines are provided for converting integers, doubles, booleans, and characters. A routine is also provided that returns a handle to the Smalltalk value nil.

The values true and false are easily generated by converting the `upBool` values `1` and `0` respectively.

Conversions 2

```
upInt UPcopySTtoCfloatArray(
          upHandle stFloatArray,
          upFloat *cFloatArray,
          upInt copyCount,
          upInt startingIndex)

upInt UPcopyCtoSTfloatArray(
          upHandle stFloatArray,
          upFloat *cFloatArray,
          upInt copyCount,
          upInt startingIndex)
```

Also:

- String
- ByteArray
- Array of Integer

INRIA ROCQUENCOURT

Conversion Routines 2: Arrays

When converting an array of values in either direction, the conversion routine requires both arrays to exists already. The arguments to these routines are: the Smalltalk and C arrays to be used, the maximum number of elements to copy, and the starting index for the copy operation. The result of these routines is always an integer giving the actual number of elements copied.

For example, the routine `UPcopySTtoCfloatArray()` copies `smalltalkFloatArray` to `cFloatArray` until `copyCount` floats have been copied, or the `smalltalkArray` has been exhausted. The routine `UPcopyCtoSTfloatArray()` performs the copy in the other direction.

Similar routines are provided for copying arrays of characters (strings), bytes, and integers.

Note that the order of the arguments is *unchanged*, regardless of the direction of the copy operation. Also, in either direction, the copy operation can be terminated either by `copyCount` bytes being copied, or by the Smalltalk array being exhausted — so it is often necessary to check the return value of the function since it does not necessarily equal either the `copyCount` or the size of the Smalltalk array.

In both directions the `startingIndex` is relative to the Smalltalk array, and is consequently 1-based.

Type Checking

```
upBool UPisInteger(upHandle obj)
upBool UPmustBeInteger(
                     upHandle obj)
```

- `Character`, `String`
- `Integer`, `Float`, `Double`
- `Array` of `Integer`, `Float`, `Byte`

Also:

- `ByteLike` (contains no oops)
- `Immediate` (oop encodes value)

```
upBool UPisKindOf(upHandle obj,
                  upHandle class)

upHandle UPclass(upHandle instance)
```

Type Checking

For each supported type, two routines are provided which check that the type of the given argument matches the expected type. The family of routines whose name begins with `UPis` returns `TRUE` or `FALSE` (`1` or `0`) if the argument is of the expected type. For example:

```
upBool UPisInteger(upHandle aUpHandle)
```

The family of routines whose name begins with `UPmustBe` will fail the primitive if the type of the argument is not the expected type, setting the failure code to `UPEwrongClass`. For example:

```
upBool UPisMustBeInteger(upHandle aUpHandle)
```

Several pseudo-types can also be checked with the `UPis` and `UPmustBe` routines. These types are:

- `ByteLike` — the argument contains only bytes (no object pointers);
- `Immediate` — the argument is an immediate object (one whose value cannot be changed without also changing the identity of the object).

Finally, the equivalent of the Smalltalk isKindOf: message is provided:

```
upBool UPisKindOf(upHandle object, upHandle class)
```

which returns `TRUE` or `FALSE` appropriately. The class of an object can be found by calling the function:

```
upHandle UPclass(upHandle instance)
```

Object Allocation

```
upHandle UPallocString(
            upChar charVal,
            long numElements)
upHandle UPallocByteArray(
            upByte byteVal,
            long numElements)
upHandle UPallocArray(
            upHandle initVal,
            long numElements)

upHandle UPallocFsObject(
            upHandle class)

upHandle UPallocVsObject(
            upHandle class,
            upInt size)
```

INRIA ROCQUENCOURT

Object Allocation

The C to Smalltalk conversion routines provide a means to allocate various Smalltalk objects from within a user primitive. Empty objects of type String, ByteArray and `Array` can be allocated explicitly by calling

```
upHandle UPallocString(
           upChar charVal, long numElements)
upHandle UPallocByteArray(
           upByte byteVal, long numElements)
upHandle UPallocArray(
           upHandle initVal, long numElements)
```

which allocate an object of size `numElements`, having every element initialised with the given value.

Smalltalk objects of an arbitrary class having only fixed fields can be allocated from within a user primitive by calling

```
upHandle UPallocFsObject(upHandle class)
```

which returns the new object with all fields initialised to nil. To allocate an object with variable fields, call

```
upHandle UPallocVsObject(upHandle class,
                         upInt size)
```

which returns an object with all fixed fields initialised to nil, all variable pointer fields initialised to nil, and all variable non-pointer fields initialised to 0.

If any of the above allocation routines cannot allocate the object for any reason, the user primitive fails with code `UPEallocationFailed`.

Indexed Access 1

```
upHandle UPbasicAt(
             upHandle array,
             upInt index);
```

- `upInt UPbyteAt()`
- `upFloat UPfloatAt()`

```
void UPbasicAtPut(
        upHandle array,
        upInt index,
        upHandle value);
```

- `UPbyteAtPut()`
- `UPfloatAtPut()`

Indexed Access 1: at: and at:put:

Several routines are available for reading indexed fields within an object. For Array objects, the element at `index` (1-based) in the given `array` is returned by the following routines:

```
upHandle UPbasicAt(upHandle array, upInt index);
upInt UPbyteAt(upHandle array, upInt index);
upFloat UPfloatAt(upHandle array, upInt index);
```

The first returns an arbitrary object, the second and third are for arrays known to contain only bytes or only floating-point numbers.

Corresponding routines are provided for writing the indexed fields:

```
void UPbasicAtPut(upHandle array, upInt index,
                  upHandle value);
```

(similarly for `UPbyteAtPut` and `UPfloatAtPut`).

Indexed Access 2

```
upHandle UPinstVarAt(
            upHandle obj,
            upInt index)
void UPinstVarAtPut(
            upHandle obj,
            upInt index,
            upHandle value)

upInt UPinstVarSize(upHandle obj)
upInt UPindexVarSize(upHandle obj)

upInt UPclassType(upHandle class)
```

#defines:

- UPnotAClass
- UPfixedSizeClass
- UPvariableSizeClass

Indexed Access 2: Classes and Instance Variables

Access to instance variables, corresponding to the Smalltalk messages instVarAt: and instVarAtPut:, is provided by the routines

```
upHandle UPinstVarAt(upHandle instance, upInt index)
void UPinstVarAtPut(upHandle instance, upInt index,
                      upHandle value)
```

The size of the fixed fields (named instance variables) and the size of the variable (indexable) fields can be found using the routines

```
upInt UPinstVarSize(upHandle instance)
upInt UPindexVarSize(upHandle instance)
```

respectively. Finally, an integer indicating whether an object has indexable fields is returned by the routine

```
upInt UPclassType(upHandle class)
```

(0 is returned if `class` is not a class).

Error Handling

```
#include "setjmp.h"
jmp_buf context;

void eHandler(upInt code) {
  longjmp(context, 1);
}

aPrimFunc(...) {
  if (!setjmp(context)) {
    /* primitive implementation */
  }
  else {
    /* return from handler */
  }
}

UPinstall(...) {
  UPinstallErrorHandler(eHandler);
}
```

Error Handling

Many of the support routines described above can fail, for a variety of reasons. A facility is provided to register an error handler which intercepts these failures, and which can perform arbitrary actions instead of simply failing the primitive. These actions might include cleanup operations before failing the primitive, ignoring the failure and returning successfully with some result, or transferring control to some earlier context using `setjmp`/`longjmp`.

The error handler is installed during user-primitive initialisation by calling

```
void UPinstalErrorHandler(void(*errorHandler)(upInt))
```

The error handler must accept one argument (the error code) and should return void.

The Registry

```
upInt slot= UPallocRegistrySlot();

upHandle UPregisteredHandleAt(
           upInt slot);

void UPregisteredHandleAtPut(
       upInt slot, upHandle obj);
```

Predefined indices for the single instances:

- nil, true, false

and the classes:

- SmallInteger, Character
- Array, ByteArray
- Float, Double
- *etc...*

The Registry of Long-Lived Objects

References to long-lived objects can be registered with the virtual machine. These references are persistent across calls to user-primitives, and are useful for preserving data that is created and managed by user primitives.

The registry is a collection of slots. The routine

```
upInt slot= UPallocRegistrySlot()
```

returns the number of an unused registry slot. One a slot has been allocated, it can be read and written in a similar way to a normal Array using the routines

```
upHandle UPregisteredHandleAt(upInt slot)
void UPregisteredHandleAtPut(upInt slot, upHandle object)
```

Once a slot is allocated, it cannot be deallocated. Registry slots are therefore a scarce resource that should be used sparingly, and only when absolutely necessary. They are most appropriate for Smalltalk objects that must be recreated after a snapshot (such as sockets), or objects that must be signaled during an asynchronous event (such as a semaphore). It is almost always more appropriate to allocate an instance or class instance variable for your application, and to use the `instVarAt/Put` mechanism instead.

The registry is used by the VM for internal purposes, to keep references to "well-known" objects and classes. Direct access to these objects is supported through a set of indices defined in `userprim.h`. These include:

- `nilOopX`, `trueOopX`, `falseOopX`
- `smallIntegerClassX`, `characterClassX`
- `arrayClassX`, `byteArrayClassX`
- `floatClassX`, `doubleClassX`,

Finally, any object stored in the registry is protected from garbage collection. If the object stored in a particular slot is no longer needed, it is a good idea to store nil in the slot. (Storing nil in a predefined slot is probably not a good idea.)

Asynchronous Events 1

- Install a poll handler using `UPinstallPollHandler()`.
- Enable a signal handler that calls `UPpostInterrupt()`, and then returns.
- VM calls poll handler in response.
- Poll handler signals a Semaphore stored in the registry.
- Smalltalk code waiting on this semaphore is resumed.
- The Smalltalk code takes appropriate action, possibly invoking a user primitive to handle the event.

Asynchronous Events 1: Theory of Operation

In Unix, asynchronous events are reported to a process by sending it a signal. User primitives can register handlers for most of the defined signals, but communicating the asynchronous event to Smalltalk is complicated by the possibility that the virtual machine could be performing operations that make the view of the object memory inconsistent as seen by the user signal handler.

One solution is to register a signal handler that simply sets a flag, and have Smalltalk periodically check that flag. This "polling" is however both inconvenient and inefficient.

The UDP interface supports a better solution, as follows.

- A Semaphore is created in Smalltalk. This semaphore is the synchronisation object on which a Smalltalk process can detect the delivery of a Unix signal.
- A user primitive is called, passing the semaphore as the receiver or an argument. This user primitive stores the semaphore in a registry slot, installs a "poll handler" (which will signal this semaphore) with `UPinstallPollHandler`, and then enables the catching of the signal(s) of interest by setting their handlers to a routine that calls `UPpostInterrupt()`.
- When one of the interesting Unix signals is delivered, the user-supplied signal handler calls `UPpostInterrupt()` and exits immediately.
- In response, the VM waits for a convenient moment and then calls the poll handler. (The delay between `UPpostInterrupt()` and the calling of the poll handler will be much smaller than any user-written polling loop could achieve.)
- The poll handler retrieves the semaphore stored in the registry and signals it. This causes any Smalltalk process waiting on the semaphore to be resumed.

Asynchronous Events 2

```
void primSetInterrupts(rcv)
upHandle rcv;
{
  upHandle semClass=
    UPregisteredHandleAt(
      semaphoreClassX);
  if (!UPisKindOf(rcv, semClass))
    UPfail(UPEwrongClass);
  if (!semSlot)
    semSlot= UPallocRegistrySlot();
  UPregisteredHandleAtPut(semSlot,
                          rcv);
  UPinstallPollHandler(pollHandler);
  oldHandler= signal(SIGUSR1,
                     sigHandler);
  UPreturnNil();
}
```

INRIA ROCQUENCOURT

Asynchronous Events 2: Preparing for a Signal

In this example, a pair of primitives are installed in (a subclass of) Semaphore which enable and disable notification of Unix signal delivery.

A user primitive is created whose receiver is a kind of Semaphore. After checking that the class of the receiver is indeed Semaphore, the semaphore is stored in a registry slot.

A `pollHandler` is installed by calling `UPinstallPollHandler()`, which will be called by the virtual machine when it is told that a signal has been delivered. This poll handler must be registered before there is any possibility of the `UPpostInterrupt()` routine being called.

Finally, the primitive calls `signal()` to arrange for the signal handler `sigHandler` to be called when the `SIGUSR1` signal is delivered.

The old signal handler is remembered in `oldHandler` and will be restored when the notification of signal delivery is disabled. (This is no more than good programming practice.)

Asynchronous Events 3

```
void sigHandler() {
  UPpostInterrupt();
}

void pollHandler() {
  UPsignalSemaphore(
    UPregisteredHandleAt(semSlot));
}

void primClrInterrupts(rcv)
upHandle rcv;
{
  UPregisteredHandleAtPut(semSlot,
                          UPnil());
  signal(SIGUSR1, oldHandler);
  UPreturnNil();
}
```

Asynchronous Events 3: Handling the Signal

The signal handler proper performs one action: to call the UDP interface routine `UPpostInterrupt()`. This informs the VM that a Unix signal has been delivered. This is the *only* UDP routine that can be called asynchronously.

Shortly thereafter the VM calls `pollHandler()` after creating a state in which any handles stored in the registry are accessible, and in which a Semaphore can be safely signaled. These are the *only* UDP routines that the poll handler is permitted to call.

The `pollHandle` retrieves the semaphore from the registry and signals it, then returns immediately. Smalltalk code waiting on the semaphore can perform the necessary handling action, or can invoke a user primitive to handle the signal. The full range of UDP facilities can be used from within this user primitive.

When the signal is no longer of interest to the application, a second primitive is invoked on the semaphore in order to disable the notification. This primitive deletes the reference to the Semaphore in the registry, and restores the handler for the signal to its original value.

Asynchronous Events 4

SignalWatchingClass≫signalCatchingLoop

```
| semaphore |
semaphore := UnixSignalSemaphore new.
semaphore primSetInterrupts.
[self signalIsStillInteresting]
    whileTrue:
        [semaphore wait.
         self handleTheSignal].
semaphore primClrInterrupts.
```

Signals not to be caught:

- SIGIO, SIGPOLL

Signals to be neither caught nor sent:

- SIGALRM, SIGVTALRM, SIGCHLD

INRIA ROCQUENCOURT

Asynchronous Events 4: The Smalltalk Side

We will assume a subclass of Semaphore called UnixSignalSemaphore in which the two user primitives described above have been installed.

The semaphore is created. It is sent a message associated with the primSet-Interrupts() primitive. This places the semaphore in the registry, and sets up the poll and signal handlers.

When the signal of interest is delivered, the poll handler is triggered and the semaphore signaled. The process waiting on the semaphore proceeds and performs the appropriate handling action.

Sending the signal a message associated with the primClrInterrupt primitive disables the delivery of the signal and removes the semaphore from the registry.

User primitives should not try to install handlers for SIGIO, SIGPOLL, SIGALRM, SIGVTALRM, or SIGCHLD. Nor should user primitives make any calls that generate the last three of these signals. Doing so is likely to break the VM. (Note that the restriction on SIGCHLD implies it is impossible to manage correctly processes spawned by user primitives.)

Unsafe Primitives

Default mode: “safe”.

“Unsafe” mode enabled with compiler flag `-DUNSAFE`.

Many VM checks on user code disabled.

Improvement in performance, increased risk of failure.

Errors trapped in “safe” mode can cause VM crashes or subtle VI corruption in “unsafe” mode.

All or nothing: cannot mix “safe” and “unsafe” primitives.

INRIA ROCQUENCOURT

Unsafe Primitives

The UDP interface normally performs numerous checks on application code. Argument types are checked in all calls UDP routines, all array accesses are bounds checked, classes are checked during conversions, and so on. These checks are obviously detrimental to performance, and in certain extreme circumstances are better switched off.

Most checking is turned off by compiling the user primitive code with the option `-DUNSAFE`. It is not possible to mix safe and unsafe code, so if `-DUNSAFE` is specified for any of the C files compiled for a particular virtual machine, then it must be specified for all of them.

Bugs in an unsafe primitive are not likely to cause the primitive to fail. They are much more likely to either crash the virtual machine, or to corrupt the image silently without crashing. A corrupted virtual image may remain undetected for a considerable amount of time, and saving the image will make the corruption permanent. The extent and severity of the damage that is possible when using unsafe primitives cannot be over stressed.

Example 2

```
typedef struct {
  char *dptr;
  int dsize;
} datum;

DBM *dbm_open(char *file,
              flags, mode);
void dbm_close(DBM *db);

datum dbm_fetch(DBM *db,
                datum key);
int dbm_store(DBM *db, datum key,
              datum value, flags);
int dbm_delete(DBM *db, datum key);

datum dbm_firstkey(DBM *db);
datum dbm_nextkey(DBM *db);
```

Example 2: DBM Database Connection

Unix provides a simple database facility called `dbm`. A `dbm` database is a dictionary using hashed keys to manage potentially billions of key/value pairs with very efficient access. This mechanism is used by (for example) `sendmail` to store mail aliases, and by the X Window System to map colour names onto RGB intensities.

The `dbm` interface is simple:

- the type `DBM *` represents a reference to a database;
- the type `datum` represents a key or a value and has two parts: a pointer to the data and the size (in bytes) of that data;
- `DBM *dbm_open(char *dbname, flags, mode)` opens, and returns a reference to, the database named `dbname`;
- `datum dbm_fetch(DBM *db, datum key)` returns the value associated with the given `key`;
- `int dbm_store(DBM *db, datum key, datum val)` stores the `value` at the given `key`;
- `void dbm_close(DBM *db)` closes the connection to the database.

Enumeration is supported with the functions

- `datum dbm_firstkey(DBM *db)` returns the first key in the database, and
- `datum dbm_nextkey(DBM *db, datum key)` returns the key after the given `key` in the database.

The following example demonstrates how to manipulate DBM databases from Smalltalk using user-defined primitives, and how to use the snapshot notification and finalisation mechanisms to manage such connections.

INRIA ROCQUENCOURT

Design Outline

Instances of DbmConnection

- represent database connections
- contain a "cookie", a C pointer to the `DBM` structure

Class DbmConnection

- manages all open connections
- destroys connections after snapshot
- automatically closes unreferenced connections

INRIA ROCQUENCOURT

Design Outline

An open connection to a `dbm` database is represented as an instances of DbmConnection. Instances implement the `dbm` operations through user-defined primitives.

Each instance contains a "cookie" representing the `DBM` structure returned by `dbm_open()`. (A "cookie" is an uninterpretable value that means nothing to the application, but is useful to the underlying implementation. In this example the cookie in a particular instance of DbmConnection is the address of the `DBM` structure associated with its open connection.)

The class DbmConnection performs two management rôles.

In a similar fashion to the ExampleServer in part 1, it registers itself as a dependent of ObjectMemory in order to receive notification of snapshots. Since `DBM` structures will not survive snapshots, it makes sense to destroy all lingering open connections on return from a snapshot. (Recall that "return from a snapshot" implies a closing down and restarting of the VM, and not merely saving the image.)

The class DbmConnection also places each instance that is created into a WeakDictionary. This means that instances have the chance to perform finalisation actions. In this example, the finalisation action is to close the connection. This is useful since each open connection consumes two file descriptors (which are a finite resource), and over 5 kilobytes of buffer space allocated by `malloc()`.

The User Primitives 1

```
#define UPEdbmCouldNotOpen   (-32)
#define UPEdbmKeyNotFound    (-33)
#define UPEdbmCouldNotStore  (-34)

#define MAXLEN 256

char *UPinstall() {
  UPaddPrimitive(
        10001, dbmOpen, 1);
  /* ... */
  UPaddPrimitive(
        10007, dbmNextKey, 1);
  return "dbm primitives";
}

main(argc, char **argv) {
  return st80Main(
    argc, argv, (char **)0);
}
```

INRIA ROCQUENCOURT

The User Primitives 1: Initialisation

Following the habitual gamut of `#include` directives (omitted here to help save the forests) are several `#defines` for new error codes. These will be used to indicate to Smalltalk where the problem lies, should a primitive fail due to some error with the `dbm` routines.

The symbol `MAXLEN` is set to the maximum size of datum that the primitives will handle. (This is used to truncate data when necessary, to fit into statically allocated data structures.)

The `main()` and `UPinstall()` routines are obvious.

The User Primitives 2

```
void dbmOpen(rcv, name)
upHandle rcv, name;
{
  DBM *db;
  upChar cname[MAXLEN];
  upInt size;
  upHandle instance;
  UPmustBeString(name);
  size= UPcopySTtoCstring(
      name, cname, MAXLEN, 1);
  cname[size]= '\0';
  if ((db= dbm_open(cname, O_RDWR
             | O_CREAT, 0644)) < 0)
    UPfail(UPEdbmCouldNotOpen);
  /* instantiate DbmConnection */
  instance= UPallocFsObject(recv);
  UPinstVarAtPut(instance, 1,
      UPCtoSTint((int)db));
  UPreturnHandle(instance);
}
```

INRIA ROCQUENCOURT

The User Primitives 2: Connecting to a Database

The primitive `dbmOpen` is installed on the class side of DbmConnection and accepts one argument, the name of the database. After checking that the name is a String, the primitive copies the String from the Smalltalk to a C representation. The C version is then manually null-terminated.

Next `dbm_open()` is called to open the database. The flags passed to it indicate that the database should be opened read/write, should not be truncated if it already exists, and should be created with a mode of `0644` if it does not already exist.

If the database can be opened successfully, the receiver (which is class DbmConnection) is instantiated by calling `UPallocFsObject()`, which is appropriate since DbmConnections have no variable fields. (Not that the receiver is never checked, so installing this primitive in another class, or on the instance side of any class, would have at best unpredictable results.)

The new instance has one instance variable (the cookie) which is set to the address of the `DBM` structure returned by `dbm_open()`, after being suitably massaged into a Smalltalk integer.

The primtive succeeds by returning the newly created instance of DbmConnection.

The User Primitives 3

```
void dbmFetch(recv, key)
upHandle recv, key;
{
  DBM *db;
  datum dbmkey, dbmvalue;
  upInt size;

  db= getDbm(recv);
  dbmkey= stringToDatum(key);
  if ((dbmvalue=
       dbm_fetch(db, dbmkey)).dptr)
    UPreturnHandle(
      datumToString(dbmvalue));
  UPfail(UPEdbmKeyNotFound);
}
```

The User Primitives 3: Database Operations

The `dbmFetch` primitive will be used to illustrate how the primitives are written.

The primitive expects the receiver to be an instance of DbmConnection. The primitive expects the programmer to ensure that the primitive is installed only on the instance side of DbmConnection which will guarantee the correct class of receiver. Hence no check is made of the receiver's class.

The primitive expects one argument, a String key to be used to fetch a value from the database. Before the fetch operation can be performed this key must be converted from a Smalltalk String into a `datum`. Since this conversion will be performed several times, it is delegated to the support routine `stringToDatum()`. No check is made on the class of the argument since this is done within the `stringToDatum()` routine itself in order to reduce duplication of code. If the conversion cannot be performed, the primitive is failed from within the `stringToDatum()` routine.

Armed with a `datum` corresponding to the key string, the `dbm_fetch()` operation is performed. The result is always a `datum`, with the "key not found" condition causing the result `datum` to have a null `dptr` field.

If the `dptr` field of the result is not null, the fetched `datum` is converted to a Smalltalk String returned as the result of the primitive with `UPreturnHandle()`. The conversion from a `datum` to a String is performed several times, and so it is delegated to another support routine `datumToString()`.

If the key was not found, the primitive fails with the application-specific error code `UPEdbmKeyNotFound`.

The other primitives `dbmStore`, `dbmDelete`, `dbmFirstkey`, `dbmNextkey`, and `dbmClose` follow the same general principles. A complete listing can be found in the appendices for those who wish to read the whole story.

The User Primitives 4

```
static
datum stringToDatum(stString)
upHandle stString;
{
  upInt size;
  static upChar cString[MAXLEN];
  datum dtm;

  UPmustBeString(stString);
  size= UPcopySTtoCstring(
          stString, cString,
          MAXLEN-1, 1);
  cString[size]= '\0';
  dtm.dptr= cString;
  dtm.dsize= strlen(cString);
  return dtm;
}
```

The User Primitives 4: Support Routines 1

Most of the primitives need to convert a Smalltalk String into a `datum` which can be used as the argument to one of the `DBM` library routines. The support routine `stringToDatum` performs this conversion.

The routine first checks that the argument is indeed a String. `UPmustBeString()` will fail the primitive with error code `UPEwrongClass` if the check fails.

The String is copied into the array `cString` using `UPcopySTtoCstring()`. The maximum number of characters that will be copied is specified as `MAXLEN-1`, ensuring that overlong keys do not fall off the end of the fixed-size `cString` array. The copy operation starts at index `1` relative to the Smalltalk String. `UPcopySTtoCstring()` returns the number of characters copied. This will be either the size of the String, or `MAXLEN-1` if the String contains more than `MAXLEN-1` characters.

The copy operation does not terminate the C string, so this is done manually. (This null termination is not strictly required since the final `'\0'` is not included in the size of the data passed to the `dbm` routines. However it is good practice to properly terminate the string since it might be necessary to print it, during debugging for example, or perform other string operations on it.)

The `datum dtm` can now be filled in. The `dptr` field is set to the address of the `cString` containing the copied String, and the `dsize` field initialised with the length of this string (excluding the null terminator).

Since this function returns a `datum` rather than a pointer to one, it is not necessary to explicitly copy the structure before returning it. On the other hand, the `dptr` field is initialised from a statically-allocate C array and so must be duplicated when necessary before a subsequent call to `stringToDatum()`. For example:

```
aDatum.dptr= strdup(aDatum.dptr);
...
free(aDatum.dptr);
```

This technique is used in the `dbmStore()` routine, shown in the full listing.

The User Primitives 5

```
static
upHandle datumToString(dtm)
datum dtm;
{
  upHandle string=
    UPallocString(' ', dtm.dsize);
  UPcopyCtoSTstring(
    string, dtm.dptr, dtm.dsize, 1);
  return string;
}
```

The User Primitives 5: Support Routines 2

The companion to `stringToDatum()` is `datumToString()` which creates a Smalltalk String given a `datum`.

This routine first instantiates a new Smalltalk String of the necessary size, filling the new instance with spaces.

`UPcopyCtoSTstring()` is called to copy the C string pointed to by `dptr` into the newly created Smalltalk String. The copy transfers exactly the number of characters in the C string, starting at index `1` relative to the Smalltalk string. Note that the arguments of both `UPcopySTtoCstring()` and `UPcopyCtoSTstring()` appear in the same order, even the the direction of the copy is different.

The value returned by `UPcopyCtoSTstring()` need not be checked since the size of the Smalltalk String and the number of characters to be copied are known to be the same.

The newly created Smalltalk String is returned as the result.

Class DbmConnection

DbmConnection class≫open: aDatabase

```
| connection |
connection := self primOpen: aDatabase.
OpenConnections at: connection cookie
        put: connection.
^connection
```

DbmConnection class≫primOpen: aDatabase

```
<primitive: 10001>
(aDatabase isKindOf: String)
  ifTrue:
    [OSErrorHolder nonexistentSignal raise]
  ifFalse
    [self error: 'name must be a String']
```

Class DbmConnection

A connection to a DBM database is opened by sending the class DbmConnection the message open: with the name of the database as the argument.

The open: method calls the private method primOpen: in order to create an instance of DbmConnection representing the open connection. If the dbmOpen primitive fails, primOpen: determines the cause of the failure by inspecting the class of the argument. If the argument is not a String then an error is signaled. If the argument is a String then the database does not exist, and the appropriate error signal is raised.

Back in the open: method, the newly created connection is registered in the OpenConnections dictionary. This is a WeakDictionary which will send a finalize message to any its values when they become unreachable. The obvious key for the newly created connection is the cookie representing the DBM structure in the C primitive implementation.

The newly created instance is returned.

DbmConnection Methods

DbmConnection Methods

DbmConnection≫fetch: key

```
<primitive: 10003>
(key isKindOf: String)
    ifTrue:
        [Object notFoundSignal raise]
    ifFalse:
        [self error: 'key must be a String']
```

DbmConnection≫close

```
OpenConnections removeKey: self cookie.
^self primClose
```

The instance side of DbmConnection has methods corresponding to the available operations: fetch:, store:value:, firstKey, nextKey:, and close. Since many of these methods are similar, we will look only at the first and last in detail.

The fetch: method returns a value given a key. It invokes the `dbmFetch` primitive directly, to return a Smalltalk String corresponding to the given key.

Should the primitive fail, the method attempts to discover what might have gone wrong. Either the given key was not a String, or the key was not found in the database. The appropriate error is signaled for each of these possibilities.

The close method invokes the corresponding `dbmClose`, but first removes the instance from the OpenConnections dictionary. (Finalisation is no longer required for the connection being closed.)

Finalisation

```
DbmConnection class≫open: aDatabase
    | connection |
    connection := self primOpen: aDatabase.
    OpenConnections
        at: connection cookie
        put: connection.
    ^connection

DbmConnection≫close
    OpenConnections removeKey: self cookie.
    ^self primClose

DbmConnection≫finalize
    self primClose
```

Finalisation

The four methods open:, close, finalize and primClose cooperate to provide finalisation for database connections. A database connection that is not closed explicitly by an application will be closed implicitly by the finalisation mechanism.

In the method open: the new instance of DbmConnection is added to OpenConnections, a WeakDictionary keyed by the "cookie" of the connection and whose values are the connections themselves.

WeakDictionary make shallow copies of its values. When one of its values is finalised, the clone receives a finalize message. Instances of DbmConnection respond to finalize by closing themselves. (They are removed from the OpenConnections dictionary by the dictionary itself, just before the finalize message is sent.)

Snapshots

DbmConnection class≫initialize

```
(ObjectMemory dependents includes: self)
    ifFalse:
        [ObjectMemory addDependent: self].
. . .
```

DbmConnection≫
update: aspect with: parameter from: sender

```
(sender == ObjectMemory and:
        [aspect == #returnFromSnapshot])
    ifTrue: [self returnFromSnapshot]
```

DbmConnection≫returnFromSnapshot

```
OpenConnections isNil ifFalse:
    [OpenConnections keys do:
        [ :key |
        OpenConnections removeKey: key]]
```

Snapshots

Database connections cannot persist across snapshots since they depend on a pair of open file descriptors attached to the database files, and an area of memory allocated with `malloc()` containing buffers and state information. When the virtual machine exits, these resources are lost. When the snapshot is resumed later, it is necessary to close all database connections that were open when the snapshot was made.

The class DbmConnection registers itself as a dependent of ObjectMemory and receives snapshot notifications in the same manner as the ExampleServer class did. On returning from a snapshot, DbmConnection is sent a changed: message. If DbmConnection establishes that this message came from ObjectMemory, and has an aspect indicating a return from a snapshot, it sends itself a returnFromSnapshot message.

The returnFromSnapshot message removes each key from the OpenConnections dictionary. It is not necessary to close these connection gracefully, since the external resources that they once represented no longer exist.

Summary

- Writing and adding UDPs.
- Primitive return, failure.
- Data conversion: objects, arrays.
- Type checking, class membership.
- Allocating Smalltalk objects.
- Indexed and instance variable access.
- Determining class type, instance and indexed var size.
- Error handlers.
- The registry.
- Handling asynchronous events.
- Using unsafe primitives.
- Snapshot and finalisation mechanisms.

INRIA ROCQUENCOURT

Summary of Part 2

- How to write primitives in C and add them to the virtual machine by calling `upAddPrimitive()` from within `UPinstall()`.
- Returning a value from a primitive using `UPreturnHandle()` and similar functions, and failing a primitive with `UPfail()`.
- Converting between C and Smalltalk data representation using `UPSTtoC()` and `UPCtoST()` for single objects, and `UPcopySTtoC...()` and `UPcopyCtoST...()` for arrays of objects.
- Type checking using `UPis...()` and `UPmustBe...()`, and testing for class membership using `UPisKindOf()`. Obtaining the class of an object with `UPclass()`.
- Allocating new Smalltalk objects of a specific class with `UPalloc...()`, and objects of an arbitrary class with `UPallocFsObject()` and `UPallocVsObject()`.
- Accessing indexed objects and instance variables using `UPbasicAt()`, `UPbasicAtPut()`, `UPinstVarAt()` and `UPinstVarAtPut()`.
- Determining the type of the fields of a class with `UPclassType()`. Finding the number of instance variables and indexed fields using `UPinstVarSize()` and `UPindexVarSize()`.
- Installing and using an error handler, with `UPinstallErrorHandler()`.
- Registering the handles of long-lived objects that will survive between primitive calls using `UPallocRegistrySlot()`, `UPregistrySlotAt()`, and `UPregistrySlotAtPut()`.
- Handling asynchronous events with `UPinstallPollHandler()` and `UPpostInterrupt()`.
- Using unsafe primitives.
- Using the snapshot notification and finalisation mechanisms to manage objects dependent on external state.

Part 3 — CPOK

- Interface Classes
- Declaring C Entities
- Calling C Functions
- Interface Inheritance
- Static Linking
- Failure Code
- The Heap
- Manipulating C Data
- Calling Smalltalk from C
- Deployment Considerations
- Two Examples

Part 3 — CPOK: the ObjectKit for C Programming

The Objectkit\Smalltalk C Programming (CPOK for short) provides mechanisms to call C functions and manipulate C data from Smalltalk. The C functions are normally placed in dynamically loaded libraries, linked into a running Smalltalk on demand. Stub methods can generated automatically, as can methods that build Smalltalk structures representing C data types and values.

The CPOK is delivered as a single file-in which provides:

- extensions to the Smalltalk compiler such that it will recognise C declarations appearing at the start of a method, in a similar manner to the way primitive declarations are made;
- a C declaration compiler which understands enough C to parse and understand most C header files;
- facilities for automatically constructing C data types and values using the framework in `ExtLib.st`

A Simple Function

Header file “mycmp.h”:

```
int mycmp(char *s1, char *s2);
```

Code file “mycmp.c”:

```
int mycmp(s1, s2)
register unsigned char *s1, *s2;
{
  while(*s1 == *s2++)
    if(*s1++ == '\0')
      return(0);
  return(*s1 - *--s2);
}
```

Compiled with:

```
cc -pic -c mycmp.c
ld -o mycmp.so -Bdynamic \
     -assert pure-text mycmp.o
```

A Simple Function

The easiest way to begin an explanation of the CPOK is with a trivial example.

Consider a library that provides a function `mycmp()`, a simple string comparison routine not entirely unlike the standard library routine `strcmp()`.

To access this code from Smalltalk using the CPOK we first compile it into position independent code, using the `-pic` flag of the compiler.

```
cc -pic -c mycmp.c
```

We then create a shared object file which can be linked into an already running executable at any address.

```
ld -Bdynamic -assert pure-text -o mycmp.so mycmp.o
```

The linker option `-Bdynamic` causes certain relocation information to be preserved, and creates additional structures in the output file needed to perform run-time linking. The `-assert pure-text` option is a safeguard: the linker will fail with an error message if it detects any references to absolute addresses in the object code.

The resulting shared object (“`.so`”) file can be used as a dynamically-loaded library.

In order to allow the CPOK to generate stubs for the functions in the libary, a header file declaring the `mycmp` function is also created. Even if the local compiler does not support ANSI prototypes (as in the example shown) header files for consumption by the CPOK must declare formal parameters in functions using ANSI prototypes if the argument checking facilities of the CPOK are to be used, or if user-defined data types are to be passed between Smalltalk and C functions.

Creating an Interface Class

ExternalInterface subclass: #TestInterface

includeFiles: 'mycmp.h'
includeDirectories: ''
libraryFiles: 'mycmp.so'
libraryDirectories: ''
generateMethods: '*'
beVirtual: false

instanceVariableNames: 'str result'
classVariableNames: ''
poolDictionaries: ''
category: 'External-Comparing'

Creating an Interface Class

One or more C libraries are associated with an interface class. The interface class provides access to the library functions, external variables, and the data types and constants which are required to use the functions and variables.

Interface classes are created using a class template similar to that used to create other kinds of classes. Interface classes are always subclasses of ExternalInterface, and their creation template starts with six fields that are not present for other kinds of class. (Instance and class variables can be declared at the same time, as usual.)

- includeFiles: takes a string containing the names of C header files which will be scanned for type declarations, macro and variable definitions, and function templates from which types, constants and method stubs can be generated automatically.
- includeDirectories: lists the directories that will be searched, left to right, for the include files.
- libraryFiles: specifies the names of the dynamic libraries ("`.so` files") which will be searched for the functions and/or variables used by the interface class.
- libraryDirectories: lists the directories that will be searched for the libraries.
- generateMethods: contains a list of function names for which stub methods will be (re)generated automatically. If this argument is an empty string then no methods will be (re)generated. If the list contains a single space then only existing methods will be regenerated. An asterisk acts as a wildcard character.
- beVirtual: controls the search order for inherited library files, as explained later. It can be true or false.

Each of the above methods is implemented as a separate message to which an interface class will respond in the expected manner. For example, an interface class using version `1.0` of a library can be upgraded to version `1.1` by sending it the single message:

```
InterfaceClass libraryFiles: 'libwhatever.so.1.1'
```

C Function Declarations

mycmp: arg1 with: arg2

<C: int mycmp(char *arg1, char *arg2)>

mycmp: arg1 with: arg2

<C: int mycmp(char *, char *)>

compare: arg1 to: arg2

<C: int mycmp(char *, char *)>

reverseCompare: arg1 to: arg2

<C: int mycmp(char *arg2, char *arg1)>

compareTo: arg

<C: result= int mycmp(char *str, char *arg)>

C Function Declarations

For each C function that is to be called from Smalltalk, a stub method must be defined using a notation similar to that used for primitives. However, in place of the

<primitive: *primNumber*>

declaration, for a primitive method, the stub method has a

<C: *functionPrototype*>

declaration instead.

Variable names can be included or omitted in the function prototype, but for each prototype this must be done consistently: either all the variables names are omitted or they are all included.

If the variable names are omitted, the arguments passed to the function are the arguments received by the stub methodm, in the same order. If variable names are included they must refer to variables accessible in the scope of the stub method body (in other words they must be the arguments to the stub method, instance variables of the receiver, or class variables in the receiver's class).

As with ANSI C, an ellipsis can be used to represent a variable number of arguments. In such cases all the arguments must be passed in an Array.

An instance variable of the receiver can be assigned with the result of the C function call by using the usual C assignment notation in the function declaration.

Automatically generated stub methods will have a selector made from the corresponding C function name, with zero or more 'with:'s tacked on the end to accommodate any second or subsequent parameters. However, there is no reason why the Smalltalk method name and the C function name need to be the same. Automatically generated names can be changed if desired to be more Smalltalk-like, and stub methods entered manually via the browser need not conform to the scheme used by the automatic stub generator.

Calling C Functions

Using the stub method:

```
AnInterface new
    someMethod: anArg
    with: anotherArg;
    varArgsMethod: argArray.
```

Using the ExternalProcedure in the pool dictionary:

```
someMethod
    callWith: anArg
    with: anotherArg.
```

Other calling messages:

```
call
callWith:[with:[with:]]
call: argArray
```

Calling C Functions

C functions are called when the associated stub method is invoked from Smalltalk. If the function has not been called previously, the library files defined in the interface class are searched and the corresponding library linked (if necessary) before calling the function.

If the C function was declared without a prototype (i.e. an empty parameter list) or if the formal parameters ended with an ellipsis (indicating a variable number of arguments), then the stub method accepts a single argument which is an array containing the actual arguments.

All C functions (and typedefs and macros) are placed in a pool dictionary associated with the interface class. Methods compiled in a context where the pool dictionary is visible will be able to access these entities by name, without sending a message to the interface. A family of call messages can be sent directly to the named C functions to effect a call. All the actual arguments can be passed in an array using the message call:, which is required if the function was declared without a prototype or if it accepts a variable number of arguments.

(For typedefs, the object associated with the name in the pool dictionary is the type itself. For macros, it is the value of the macro.)

Interface Inheritance

Interface classes can be formed into a hierarchy. Subclasses inherit the includeDirectories, libraryDirectories and libraryFiles from their parent (but they do not inherit the includeFiless).

A subclass inherits the stub methods of its parent (just like any other subclass in Smalltalk). When an inherited stub method is invoked, there are two possible places in which to search for the library containing the corresponding C function:

- in the class which implements the stub method (which is the default), or
- in the class of the receiver (which will be a subclass of the class which implements the inherited stub method).

In the latter case, the parent class is called a "virtual" parent. A class can be made virtual (or not) by sending it the message beVirtual: with true or false as the argument.

(One way to visualise this mechanisms is to imagine that the stub method in the virtual parent performs an additional "sends of *the C function* to 'self'" causing the search for the C function to begin again, in the libraries associated with the class of the receiver.)

This mechanism allows a subclass to substitute alternate implementations of a C library without having to alter the library used in the superclass, and without having to provide duplicate stub methods for the new implementation in the subclass.

The default is for a parent to be non-virtual, thus protecting its implementation from subclasses that might wish to try to tamper with it from below.

Types, Macros and Variables

size_t

<C: typedef int size_t>

```
#define bufferSize 4096
```

bufferSize

<C: #define bufferSize 4096

>

errno

<C: int errno>

errno: value

<C: int errno>

Types, Macros and Variables

Each user-defined type used in variable or function declarations must be defined using a C: typedef expression. (This method is used during creation, accessing, and typechecking of a datum of the given type.)

The standard C data types, described later, are handled automatically without requiring any definitions of their own.

Note that no error checking is performed by the compiler on C type declarations. If a type name is mis-spelled, or is not declared at all, the only (rather unhelpful) diagnostic that will be generated is a notifier saying

message not understood: #type

when an argument of that type is accessed, converted, or type checked.

The method name and the type name need not agree, although the name supplied in the typedef expression is used to identify the type in the interface's pool dictionary. Having two names (one in the pool dictionary, and the other being the name of the message returning the type when sent to an instance of the interface) could well prove to be confusing.

The CPOK C parser is not as robust as that found in most production C compilers, and it is not difficult to confuse it with a byzantine typedef expression. Such expressions can always be split into several simpler parts, with which the CPOK parser can cope.

Similar declarations are required for macros which name constants. (Only macros with no arguments and which expand to a constant value are permitted when generating macro definitions automatically.)

Read access to C variables is provided by declaring a stub method whose C declaration part simply declares the variable. Write access uses the same C declaration, but with an argument to the stub method.

Static Linking

```
void UPInitLinkRegistry()
{

    /* for functions: */
    UPRegisterSymbolAndHandle(
        "name",
        name);

    /* for variables: */
    UPRegisterSymbolAndHandle(
        "name",
        &name);
}
```

Static Linking

Libraries can be statically linked into a customised VM (in a manner similar to that of adding user primitive code to the VM) and then accessed using CPOK facilities.

The procedure for doing this is almost identical to that described earlier for user-defined primitives. A routine called `UPInitLinkRegistry()` must be written and included with the customised VM. This routine should call `UPRegisterSymbolAndHandle(name, addr)` for each external function or variable that will be accessed. For example:

```
UPInitLinkRegistry()
{
    UPRegisterSymbolAndHandle("strcmp", strcmp);
    UPRegisterSymbolAndHandle("errno", &errno);
}
```

The first parameter is the name that CPOK will recognise for the entity in the C declaration part of a stub method. The second parameter is the actual name of the function or variable that will be accessed. By convention these two names are the same.

Obviously, the libraries which define the functions that will be statically linked must be specified when the VM is linked.

It is almost always possible to reuse existing user primitive code with the CPOK, by supplying suitable stub method declarations (manually, or generated automatically from a header file). The CPOK recognises the type `upHandle` as being equivalent to `_oop`.

Just as with user-defined primitives, code running under the CPOK can be compiled in safe or unsafe mode. Exactly the same precautions should be employed as with UDPs when compiling code `UNSAFE`. Additionally, it is very important to make sure that the glue code (the `UPInitLinkRegistry()` function) is compiled with the same safety level as the code that it installs.

Failure Code

```
ExampleInterface≫someFunction: argument
    <C: void someFunction(int)>
    "what went wrong?"
    argument isMemberOf: SmallInteger
        ifTrue: [exception parameter == ...]
        ifFalse: [...]

ExampleInterface≫errorSignals
    ^SignalCollection
        with: Object nonIntegerIndexSignal
        with: Object subscriptOutOfBoundsSignal
```

Failure Code

In a similar manner to primitive declarations, C function declarations can be followed with a Smalltalk method body specifying the recovery action to be performed in case of failure.

Within the body of the failcase code, the (invisibly declared) variable exception is available, which will contain the exception which caused the failure. If the failure was signalled from within the primitive itself (using `UPfail()`) then the error code will be the parameter to this exception.

An interface can implement the method errorSignals which controls which signals are caught and passed to the failure code.

The value returned can be a single error signal, or a SignalCollection containing the set of signals to pass to the failure code. The default, implemented in class ExternalInterface, returns Object errorSignal which causes all errors to be passed to the failure code.

The Heap

```
pointerToThing := aThing copyToHeap.
pointerToThing := thingType malloc.
pointerToThings := thingType malloc: 5.

aPointerToSomething free.

aString := aStringPointer
               copyCStringFromHeap.

pointerToThing := aThing gcCopyToHeap.
pointerToThing := thingType gcMalloc.
pointerToThings := thingType gcMalloc: 5.
```

The Heap

The Smalltalk memory manager frequently moves objects. Any object can move between calls to C functions, or during a callback or message send originated within a C function. For this reason it is possible to place objects on the C heap, where they will not be tampered with by the Smalltalk memory manager. Any pointers to data that will be retained across functions calls, or during a callback or message send originated from within a C function, *must* be placed on the heap.

Most Smalltalk objects that have equivalents in C (strings, arrays, numbers, and data of user-define C composite types) can be copied to the heap by sending the object the message copyToHeap. The result is a CPointer object that refers to the copy on the heap. This copy will not be reclaimed automatically, and when it is no longer required the storage must be explicitly released by sending the pointer the message free.

Uninitialised heap storage for an object of a given class can be reserved by sending the class the message malloc. Space for a contiguous array of objects of a given type can be created by sending the type the message malloc:. As with the copyToHeap message, the pointer returned by the malloc message must be explicitly freed when no longer needed.

Objects on the heap are manipulated using a set of C-like operators, described below. They cannot be copied back to Smalltalk objects, except in the case of strings. The message copyCStringFromHeap copies a null-terminated `char *` object from the heap into a newly allocated Smalltalk String, omitting the terminator character.

The messages gcCopyToHeap, gcMalloc and gcMalloc: are similar to their counterparts without the gc prefix, except that the CPointer that they return is registered for finalisation. When no more references to the pointer exist, the storage that the pointer refers to is freed automatically. Pointers created with one of the three gc messages should *never* be freed explicitly.

Needless to say, data held on the C heap will not be preserved across snapshots.

C Data Types

```
CIntegerType int.
CIntegerType unsignedLong.

CLimitedPrecisionRealType float.
CLimitedPrecisionRealType longDouble.

CArrayType constLongPointer
    numElements: 42;
    numElements.

aType
    beConstant;
    cast: anObject.
```


C Data Types

A type object is required when creating new data on the heap with one of the malloc messages. The typedef declarations generated for user-defined types provide one source of these type objects. For the standard data types there are several classes whose instanced represent a type.

Several of these classes, in particular the classes representing scalar values, generate several different types. For example, CIntegerType can generate signed or unsigned characters, shorts, integers, and longs. CLimitedPrecisionRealType can generate both single and double precision floating point types.

Other types, such as CPointerType, can be instantiated with various kinds of referent type. Instances of CArrayType can be queried for the number of elements, and so on.

Given an instance of a type, it can be made into a `const` version of itself by sending it the beConstant message. Similarly, sending the message cast: with an object as the argument casts the object into the type represented by the receiver.

There are more type classes, instance creation messages, and operations for modifying types, but these simply extend the lists just given. Browsing in and under the CType class for a few minutes will reveal the entire story—trying to present an exhaustive list in this commentary would be futile.

Manipulating C Data

`*pc`	pc contents
`*pc= c`	pc contents: c
`px + n`	px + n (*also:* -, +=, -=, ++, –)
`px[i]`	px at: i (*also:* at:put:)
`x.member`	x memberAt: i
`px->member`	px memberAt: i
`(type)x`	type cast: x
`sizeof(x)`	x sizeof

Manipulating C Data

The Smalltalk representations for C data implement messages that mimic the operations available in C for accessing and manipulating data. The operations can manipulate objects in Smalltalk's memory or objects on the C heap.

`x= *px`	x := px contents
`*px= x`	px contents: x
`*px1= *px2`	px1 contentsFrom: px2
`p + offset`	p + offset
`p1 - p2`	p1 - p2
`p += offset`	p += offset
`p -= offset`	p -= offset
`++p`	p increment *or* p += 1
`--p`	p decrement *or* p -= 1
`x= ax[index]`	x := ax at: index
`ax[index]= x`	ax at: index put: x
`px= &ax[index]`	px := ax refAt: index
`x= ps->memberx`	x := ps memberAt: #memberx
`x= ps.memberx`	x := ps memberAt: #memberx
`ps->memberx= x`	x := ps memberAt: #memberx put: x
`ps.memberx= x`	x := ps memberAt: #memberx put: x
`px= &ps->memberx`	px := ps refMemberAt: #memberx
`x= (xType)y`	x := xType cast: y
`px= malloc(sizeof(xType))`	px := xType malloc
`px= malloc(sizeof(xType) * n)`	px := xType malloc: n
`free(p)`	p free
`sizeof(x)`	x sizeof

Callbacks

ExampleInterface≫pfiType

<C: typedef int (*pfiType)(void)>

ExampleInterface≫doSomething: arg

<C: void doSomething(pfiType arg)>

```
| ec if |
if := ExampleInterface new.
ec := ExternalCallback
    do: [(DialogView confirm: 'continue?')
            ifTrue: [1] ifFalse: [0]]
    ofType: if pfiType.
if doSomething: ec
```

```
void doSomething(int (*cbk)()) {
    cbk();
}
```


Callbacks

It is often useful to be able to invoke Smalltalk code from within a C function. Two mechanisms are available to do this: callbacks, and message sends originated from within a C function.

Callbacks are Smalltalk "functions" passed to C code, which can be invoked later from within C to perform the associated Smalltalk action. I callback has two parts: a block specifying the actions to perform, and the type which will be used when presenting the block to the recipient C function. In order for C to be able to invoke the callback object, its type must be "pointer to function returning...":

```
| callback |
callback := ExternalCallback
        do: aBlockToExecuteFromC
        ofType: apparentTypeOfTheBlockInC
```

Associated with each instance of ExternalCallback is an object responsible for handling the transition from C into Smalltalk. This object must survive for the duration of the callback. However, callbacks are collected like any other Smalltalk object when the last reference to them is deleted. It is therefore important to ensure that a reference to a callback is kept for the duration of the C function call in which it is used (or longer, if the C function stores its reference to the callback).

Callbacks do not survive snapshots.

Sending Messages

```
_oop UPSendMessage0(
      _oop rcv, _oop sel,
      _oop *keep, _oop fail);

_oop UPSendMessageN(
      _oop rcv, _oop sel,
      _oop arg1, _oop arg2,
            ..., _oop argN,
      _oop *keep, _oop fail);

_oop UPSendMessageMany(
      _oop rcv, _oop sel,
      _oop argArray,
      _oop *keep, _oop fail);

_oop UPCSendMessage0(
      _oop rcv, char *sel,
      _oop *keep, _oop fail);
```

INRIA ROCQUENCOURT

Sending Messages from C

An alternative to the callback mechanism, and much less intrusive on existing code that is being modified to work with Smalltalk, is a facility for sending a message to a Smalltalk object from within C.

The `UPSendMessage...()` family of functions take the following arguments:

- the receiver of the message, which should be a Smalltalk object (of type `_oop` or `upHandle`);
- the selector of the message to be sent, which can be supplied as a symbol (in which case Smalltalk must arrange to pass the message selector to the C function) or as a string (in which case the C code can even construct the selector dynamically);
- zero or more arguments, which must be Smalltalk objects;
- an array of objects which must be protected from the Smalltalk garbage collector during the callback; and
- the object to return as the result of the message send in the case of failure.

Objects referenced from anywhere in the C stack that are not protected from the GC by being placed in the `keep` array may be garbage collected during the callback.

Deployment Considerations

ExternalInterface
  bossIn: fileName;
  bossOut: fileName;
  bossOutHierarchy: fileName;
  bossOutSubclasses: fileName.

SpeechInterface
  libraryFiles: 'squark.so';
  libraryDirectories: '/usr/local/lib';
  unloadLibraries;
  sanityCheck.

INRIA ROCQUENCOURT

Deployment Considerations

What happens if you want to deliver an application which is based in part on C code connected to Smalltalk via the CPOK? Your clients cannot file-in the application unless they buy the CPOK.

The class ExternalInterface provides several methods for exchanging interface classes in binary for, using BOSS.

The method bossIn: takes a string naming a BOSS file as its argument, and reads in the class definitions found in that file. The companion method bossOut: writes an interface class definition onto a file.

```
SpeechInterface bossOut: 'speechInterface.b'.
"...time passes..."
ExternalInterface bossIn: 'speechInterface.b'.
```

The two related methods, bossOutHierarchy: and bossOutSubclasses:, do the obvious things.

There is no guarantee that a client has installed the libraries on which the interface classes depend in the same place as the developer. During the installation procedure, an interface class can be sent one or both of libraryFiles: and libraryDirectories: will cause the libraries to be unloaded and relinked.

An interface class can be told to unloadLibraries, but under SunOS this does not work due to a bug in Sun's dynamic library support. (This problem can be worked around during development by creating new versions of a library with a new name, and then sending the corresponding libraryFiles: message to the interface class.)

Finally, sending the message sanityCheck to an interface class causes it to verify that it can find all of the libraries on which it depends. Libraries that cannot be located are reported in the system transcript.

Example 3a

ExternalInterface subclass: #DBMInterface

includeFiles: 'ndbm.h '

includeDirectories: '/usr/include '

libraryFiles: 'libc.so.1.9 '

libraryDirectories: '/lib '

generateMethods: ''

beVirtual: false

instanceVariableNames: ''

classVariableNames: ''

poolDictionaries: 'DBMInterfaceDictionary '

category: 'ESUG-CPOK'

Example 3a: CPOK Version of DBM Connection

Two small examples will be used to illustrate the CPOK.

The first of these is a CPOK version of the connection to the DBM database which was introduced in the user-defined primitives section.

The interface class creation template specifies the header file `ndbm.h'` in the standard include directory `/usr/include`. The DBM functions are part of the standard C runtime library `libc.so.1.9` which can be found in the directory `/lib`.

The generateMethods template is specified as '*', which causes all typedef, macro and function definitions in the header file to be installed automatically in the interface class.

Generated Stubs

```
datum
    <C: typedef struct
            char * dptr;
            int dsize;
        datum>

DBM_INSERT
    <C: #define DBM_INSERT 0
    >

DBM_REPLACE
    <C: #define DBM_REPLACE 1
    >

dbm_open: argArray <C: DBM * dbm_open(...)>
```

Generated Stubs

Two typedefs are installed in the interface class. The first is the type of the DBM connection itself. (It's big and it's largely transparent to the use of the interface, so it's probably best not to reproduce it here.) The second is the `typedef` for `datum` containing the two fields that we saw earlier, `dptr` and `dsize`.

Amongst sever other "noise" definitions are definitions for `DBM_INSERT` and `DBM_REPLACE` which control the behaviour when adding an element with a key that already exists.

(It would be nice to include `fcntl.h` in the list of include files too, so as to get the definitions of `RDONLY`, `RDWR`, and other flags needed for the `dbm_open()` function, but unfortunately the CPOK seems to have trouble getting past the declaration of `fd_set` included in many versions of `fcntl.h`.)

Finally, since Sun's `ndbm.h` header file does not use function prototypes, all the stub methods declare C functions with arbitrary arguments. (This isn't really too much of a problem as long as extra care is taken to pass the correct arguments to the C functions.)

Accessing the Database

```
| cx key dict |
dict := Dictionary new.
cx := dbm_open callWith: '/etc/aliases'
            with: IOAccessor readOnly with: 0.
key := dbm_firstkey callWith: cx.
[(key memberAt: #dsize) isZero] whileFalse:
      [dict
            at: (self stringFrom: key)
            put: (self stringFrom:
                (dbm_fetch callWith: cx with: key)).
      key := dbm_nextkey callWith: cx with: key].
dbm_close callWith: cx.
^dict
```

Accessing the Database

The interface can now be used directly, as shown in the slide. The code must be compiled in a context where the DBMInterfaceDictionary is visible. Two caveats are worth mentioning:

- Since we could not include `fcntl.h` in the interface's includeFiles, we need to get the flags for the call to `dbm_open()` from elsewhere. Luckily the class IOAccessor provides the method readOnly which returns the desired value.
- There is a problem with the way that the CPOK handles strings. Strings used as keys and/or values used in a DBM database need not be null-terminated, since their size is given in the `dsize` field. In this case, the automatic conversion might not find the null terminator and will run off the end of the string. The solution is to write a small method that retrieves the string from a datum:

```
stringFrom: aDatum
    ^(aDatum memberAt: #dptr)
        copyFrom: 1
        to: ((aDatum memberAt: #dsize)
                min: (aDatum memberAt: #dptr) size)
```

Maybe it's a matter of taste, but to me this seems much less elegant than the equivalent code written using the UDP-based interface:

```
| dict dbm key |
dict := Dictionary new.
dbm := DbmConnection open: '/etc/aliases'.
key := dbm firstKey.
[key isNil] whileFalse:
    [dict at: key put: (dbm fetch: key).
    key := dbm nextKey].
^dict
```

Example 3b

ExternalInterface subclass: #SpeechInterface

includeFiles: 'libspeak.h '
includeDirectories: ''
libraryFiles: 'libspeak.so '
libraryDirectories: ''
generateMethods: '*'
beVirtual: false

instanceVariableNames: ''
classVariableNames: ''
poolDictionaries: ''
category: 'ESUG-CPOK'

INRIA ROCQUENCOURT

Example 3b: Making Smalltalk Talk

Since the previous example was so short, we'll have a second example (which is fun, even if it adds relatively little to the understanding of the CPOK).

This example will provide access from Smalltalk to a publicly available speech package for Sun workstations.[1]

The interface class creation template is so unremarkable that we won't remark on it.

[1] The "speak" package is available by anonymous FTP from `brain.vifp.monash.edu.au` in the file `/pub/src/util/speak.tar.gz`.

The Header File

```
int speak_load_samples(char *dir);
int speak_open(char *dev, int now);
void speak_volume(int fd,
                  double vol);

void speak_string(int fd,
                  char *str);

int speak_file(int fd,
               char *filename);

void speak_delay(int delay);
void speak_close(int fd);
```

The Header File

The `libspeak` library does not come with a header file. In order to create the stub methods automatically, a trivial header file needs to be writted describing the seven public functions in the library.

The generated stubs are as follows:

- speak_load_samples: causes the library to read a phoneme set from the named directory.
- speak_open:with: opens the audio device named in the first argument (usually `"/dev/audio"`). If the second argument is non-zero, the device is opened in non-blocking mode. Returns the descriptor of the open device.
- speak_volume:with: takes an open audio device descriptor as the first argument, and a floating point number between 0.0 and 1.0 as the second. The output volume is set accordingly.
- speak_string:with: takes an open audio device as the first argument, and a string to be spoken as the second.
- speak_file:with: as with speak_string: except that the second argument is the name of a text file, whose contents will be recited.
- speak_delay: introduces a delay in the speech stream (particularly useful when the device is opened in non-blocking mode). The integer second argument specifies the number of tenths of a second for which to pause. If zero is specified, the last pause time is re-used. The default is two tenths of a second.
- speak_close: takes an open device as the argument, and closes it.

Adding Glue

SpeechInterface class≫speak: words

```
"SpeechInterface speak: 'Smalltalk'"
| speaker device |
speaker := self new.
speaker speak_load_samples: 'phonemes'.
device := speaker
            speak_open: '/dev/audio'
            with: 0.
speaker speak_volume: device with: 1.0.
speaker speak_string: device with: words.
speaker speak_close: device.
```

Adding Glue Code for Convenience

The interface generated by the CPOK would be a little unpalatable if left in its raw state. A small glue method can make making smalltalk talk less painful.

An instance of the interface is created and stored in speaker. The interface is told to load the phonemes from some standard place (in this example we assume the current directory), the audio decide `/dev/audio` is opened in blocking mode, and is set to full volume. Finally the interface is told to speak the words, and is closed.

Given this glue code we can do some embarrassingly nerdy things, for example:

```
Object errorSignal
  handle: [ :ex |
    SpeechInterface speak: ex errorString.
    ex reject]
  do: [ whatever ]
```

One last point must be explained. The speak package uses the archive library `libaudio.a` provided by Sun. This library is not supplied as a shared object (dynamic library) file.

The easy solution is to statically link the library into the virtual machine, as explained earlier. The problem with this approach is that only those members of an archive that are actually referenced are extracted by the linker. Since the required routines are not referenced until the `libspeak.so` file is loaded at run time, nothing from `libaudio.a` ends up in the VM when it is linked statically.

We could determine manually which routines are used and then write a `UP-InitLinkRegistry()` routine, but this is tedious, potentially error-prone, and places unnecessary information in VM data structures. A "sledgehammer" alternative, appropriate for small archive libraries, is to extract all the members of the library and create a new `libaudio.o` file from them which can be linked with the VM (this must be done in a directory where there are no other `.o` files):

```
ar x /usr/demo/SOUND/libaudio.a
ld -r -o libaudio.o *.o
```

Summary

- Creating an Interface Class
- Declaring C Entities in the Interface
- Calling C Functions
- Statically Linked Libraries
- Implementing Failure Code
- Creating/Manipulating Heap Data
- Standard Data Types
- Calling Into Smalltalk
- Deployment Issues
- Small Examples

Summary of Part 3

- How to fill in an interface class definition template specifying included files, libraries, automatically generated methods, and the "virtual" flag for parent interfaces.
- How to declare C functions, variables, data types, and macros in the interface class.
- How to call a C function from Smalltalk, either by sending a message to an instance of the interface class or by using the object stored in the interface's pool dictionary.
- How to statically link a library, and then register the names of function within that library so that Smalltalk can call them.
- How to provide fail-case code for a C function declaration, possibly using the hidden exception variable.
- Creating and manipulating data on the C heap, and management of heap allocated data.
- How to create types representing standard C data types using the built-in type classes.
- How to call into Smalltalk from a inside C function.
- Some issues which need to be borne in mind when deploying an application that uses the CPOK.
- Two relatively small examples demonstrating many of the concepts described earlier.

Performance

- Common Operations
  - array access/copy
  - string access/copy
  - copy to heap
  - empty call
- CPOK
  - callback
  - message send
- `SAFE` and `UNSAFE`
- dynamic and static

INRIA ROCQUENCOURT

Performance of UDP and CPOK

The times reported for each operation are based on the sum of user and system times. The reported numbers of operations typically performed per second are based on the elapsed real time during the trial.

All measurements were performed on an Sun IPX running SunOS 4.1.3, with no other users or user processes present. The machine had sufficient real memory that paging activity was essentially zero during the trials.

All custom C code was compiled with the Sun C compiler, with optimisation `-O2`.

The CPOK tests were performed on both dynamically and statically linked libraries. No significant performance difference was apparent between the two.

Array Access

test	ms	freq
	SAFE	
fillArray	86.7	11.5
sumArray	99.6	9.8
fillByteArray	47.1	21.1
sumByteArray	40.9	24.0
	UNSAFE	
fillArray	56.2	17.7
sumArray	12.9	75.7
fillByteArray	11.6	83.0
sumByteArray	5.7	174.8

INRIA ROCQUENCOURT

Array Access

In all trials, the array had 20000 Integer elements and was allocated once. The code for the C routines is as follows:

```
upHandle fillArray(self, array)
upHandle self, array;
{
  register int size= 0;
  register int index;

  size= UPindexVarSize(array);

  for (index= 1; index <= size; index++)
    UPintegerAtPut(array, index, index);

  UPreturnHandle(UPCtoSTint(42));
}

upHandle sumArray(self, array)
upHandle self, array;
{
  register int result= 0;
  register int size= 0;
  register int index;

  size= UPindexVarSize(array);

  for (index= 1; index <= size; index++)
    result+= UPintegerAt(array, index);

  UPreturnHandle(UPCtoSTint(result));
}
```

```
| array |
array := Array new: 20000 withAll: 42.
. . .
self measure: [self fillArray: array].
self measure: [self sumArray: array].
```

Replacing occurrences of `Array` with `ByteArray` yields the other two routines.

Array Copying

	ms	*freq*
	SAFE	
UPcopySTtoCarray: 1	0.13	7590.9
10	0.13	7449.2
100	0.19	5267.3
1000	0.68	1441.3
10000	5.8	171.1
	UNSAFE	
UPcopySTtoCarray: 1	0.12	7950.5
10	0.13	7680.5
100	0.17	5711.0
1000	0.69	1452.2
10000	5.86	169.8

Array Copying

A single area of memory was used as the destination for each copy operation and each source array was allocated once, all before the first trial was run.

```
#define MAXSIZE 100000
static int *cArray= 0;

upHandle initUdp()
{
  if (!cArray) {
    cArray= (int *)malloc(MAXSIZE*sizeof(int));
    if (!cArray) UPfail(-99);
  }
  UPreturnNil();
}

upHandle copyArray(self, stArray)
upHandle self, stArray;
{
  int size= UPindexVarSize(stArray);

  if (size > MAXSIZE) UPfail(-98);
  UPcopySTtoCintArray(stArray, cArray, size, 1);
  UPreturnHandle(UPCtoSTint(size));
}
```

```
PrimitiveInterface≫copyArrayUdp: anArray
    <primitive: 10104>

| a1 a10 a100 a1000 a10000 |
a1 := Array new: 1 withAll: 42.
...
a10000 := Array new: 10000 withAll: 42.

self measure: [self copyArray: a1].
...
self measure: [self copyArray: a10000].
```

String Copying

	ms	*freq*
	SAFE	
UPcopySTtoCstring: 1	0.14	7334.7
10	0.14	7297.1
100	0.14	7102.0
1000	0.21	4759.7
10000	1.05	950.9
	UNSAFE	
UPcopySTtoCstring: 1	0.13	7643.5
10	0.12	7702.4
100	0.14	6975.7
1000	0.20	4886.4
10000	1.07	930.9

String Copying

The area of memory allocated for the array trials was reused by the string trials.

```
upHandle copyString(self, stString)
upHandle self, stString;
{
  int size= UPindexVarSize(stString);

  if (size > MAXSIZE*sizeof(int)) UPfail(-98);
  UPcopySTtoCstring(stString, (char *)cArray, size, 1);
  UPreturnHandle(UPCtoSTint(size));
}
```

```
PrimitiveInterface≫copyStringUdp: aString
    <primitive: 10104>

| s1 s10 s100 s1000 s10000 |
s1 := String new: 1 withAll: $*.
. . .
s10000 := String new: 10000 withAll: $*.

self measure: [self copyString: s1].
. . .
self measure: [self copyString: s10000].
```

Copy to Heap

test	ms	freq
	SAFE	
copyToHeap 1 byte	1.12	887.7
copyToHeap 10 bytes	3.99	249.4
copyToHeap 100	32.7	30.4
copyToHeap 1000	318.6	3.1
copyToHeap 10000	3181.6	0.31
	UNSAFE	
copyToHeap 1 byte	1.03	967.0
copyToHeap 10 bytes	3.42	290.1
copyToHeap 100	27.27	36.5
copyToHeap 1000	266.6	3.7
copyToHeap 10000	2656.5	0.4

Copy to Heap

This trial simply sent the copyToHeap message to a String. The strings were allocated once, before the first trial was run, and each was filled with an asterisk.

```
| s1 s10 s100 s1000 s10000 |
s1 := String new: 1 withAll: $*.
. . .
s10000 := String new: 10000 withAll: $*.

self measure: [s1 copyToHeap free].
. . .
self measure: [s10000 copyToHeap free].
```

Simple Call

	ms	*freq*
UDP SAFE	0.1108	8982.5
UDP UNSAFE	0.1140	8741.8
CPOK SAFE	0.2913	3343.8
CPOK UNSAFE	0.3084	3223.7

INRIA ROCQUENCOURT

Simple Call

The C routines simply returned immediately.

```
upHandle emptyCallUdp(self)
upHandle self;
{
    UPreturnNil();
}

void emptyCallCpok()
{
    return;
}
```

PerformanceInterface≫primEmptyCallUdp
<primitive: 10201>

PerformanceInterface≫primEmptyCallCpok
<C: void emptyCallCpok(void)>

self measure: [self emptyCallUdp].
self measure: [self emptyCallCpok].

Calling Smalltalk

	ms	*freq*
	callback	
SAFE	3.17	313.4
UNSAFE	3.22	308.8
	message send	
SAFE	1.03	968.2
UNSAFE	1.05	943.9

Calling Smalltalk

Both the callback block and the message send computed and returned the value of 21 *2. The time required to compute that expression was also tested, and found to be vanishingly small with respect to the times taken for both the callback and the message send.

```
int callbackCpok(cbk)
int (*cbk)();
{
  return cbk();
}

int messageSendCpok(rcv, sel, args)
_oop rcv, sel, args;
{
  extern int UPSendMessageMany();
  return UPSendMessageMany(
           rcv, sel, args, (_oop *)0, (_oop)0);
}
```

PerformanceInterface≫pfi
<C: typdef int (*pfi)()>

PerformanceInterface≫callbackCpok: cbk
<C: int callbackCpok(pfi cbk)>

PerformanceInterface≫messageSendCpokStatic: rcv perform: sel with: args
<C: _oop messageSendCpok(_oop rcv, _oop sel, _oop args)>

```
| ec |
ec := ExternalCallback do: [21*2] ofType: pfi.
. . .
self measure: [self callbackCpok: cbk].
measure: [self messageSendCpok: 21 perform: #'*' with: #(2)].
```

Appendix A: The C Client for a Smalltalk Server

```
/* stclient.c
 *
 * Connects to a local Smalltalk process on a well-known port, sends
 * the command-line arguments to Smalltalk as a string terminated with
 * a newline, and then loops copying stdin to the input side os the
 * Smalltalk socket and the  output side of the socket to stdout.  The
 * process exits when the socket is closed down from the Smalltalk side.
 *
 * ANSI-philes beware: this code is cheerfully traditional.
 */

#include <stdio.h>
#include <sysexits.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <sys/param.h>
#include <sys/stream.h>
#include <netdb.h>
#include <netinet/in.h>

#define SMALLTALK_HOST  "localhost"
#define SMALLTALK_PORT  1966

#define BUFFER_SIZE     4096

#define ERR(X) { perror(X); exit(EX_UNAVAILABLE); }
#define FAIL(X) { fprintf(stderr, "%s failed\n", (X)); exit(EX_UNAVAILABLE); }

/* open a TCP connection to the given host and port number
 */
static tcp_connect(name, port)
char *name;
{
  struct sockaddr_in addr;
  struct hostent *hp;
  int stream, true= 1;
  int retry= 10;

  while (1) { /* loop until connection established */
    /* setup the foreign socket address */
    addr.sin_family= AF_INET;
    addr.sin_port= htons(port);
    if (!(hp= gethostbyname(name))) ERR(name);
    bcopy(hp->h_addr, &addr.sin_addr, sizeof(struct in_addr));
    /* create a reusable socket for a TCP connection */
    if ((stream= socket(AF_INET, SOCK_STREAM, 0)) < 0) ERR("socket");
    if (setsockopt(stream, SOL_SOCKET, SO_REUSEADDR, &true, sizeof(true)) < 0)
      ERR("setsockopt");
    /* try to connect, returning connected socket if successful */
    if (connect(stream, &addr, sizeof(struct sockaddr_in)) >= 0)
      return stream; /* succeed */
    /* for end may be busy, so don't give up immediately if refused */
    if (errno == ECONNREFUSED && retry--) sleep(1);
    else {
      close(stream);
      return -1; /* fail */
    }
  }
}

/* send strings from "strvec" (terminated with a null pointer) to
 * "destfd", in a single message, separated by spaces, and terminated
 * by a newline.
 */
static sendstrings(destfd, strvec)
char **strvec;
{
  int len;
  char *obuf, *pchar, **pstr;

  /* allocate a buffer of sufficient size for strings */
  for (len= 0, pstr= strvec; *pstr; pstr++) len+= strlen(*pstr)+1;
  if (!(pchar= obuf= (char *)malloc(len))) FAIL("malloc");
  /* linearise the strings into the buffer, and terminate with newline */
  for (pstr= strvec; *pstr; pstr++) {
    pchar+= strlen(strcpy(pchar, *pstr));
    *pchar++= ' ';
  }
  *(pchar-1)= '\n';
  /* atomically write buffer to destination, being pessimistic about failure */
  if (len != write(destfd, obuf, len)) FAIL("atomic write of string vector");
  if(!free(obuf)) FAIL("free");
}

/* copy a chunk of data from "fromfd" to "tofd", returning -1 if the
 * "fromfd" has reached EOF.
 */
static ship(fromfd, tofd)
{
  static char iobuf[BUFFER_SIZE];
  int nread, nwritten;
  char *optr= iobuf;
  if ((nread= read(fromfd, iobuf, BUFFER_SIZE)) < 0) ERR("read");
  if (nread == 0) return -1;  /* fail: connection reset by peer */
```

```c
  /* loop, copying chunks of data, until input drained */
  while (nread > 0) {
    if ((nwritten= write(tofd, optr, nread)) < 0) ERR("write");
    if (nwritten) nread-= nwritten, optr+= nwritten;
    else {
      fprintf(stderr, "write returned zero -- why?\n");
      sleep(1);
    }
  }
  return 0;  /* succeed */
}

/* asynchronously connect a unidirectional "source" and "sink" to a
 * bidirectional socket, shutting down the input side of the socket
 * when "source" is exhausted, and returning when the socket is closed
 * down by the foreign host.
 */
static to_from_duplex(source, sink, sock)
{
  fd_set openfds;

  /* we're interested in input events on either source or socket */
  FD_ZERO(&openfds);
  FD_SET(source, &openfds);
  FD_SET(sock, &openfds);
  /* drain the socket to the sink, until the socket is disconnected */
  while (FD_ISSET(sock, &openfds)) {
    fd_set readfds;
    int nready;

    /* wait for data on either of the inputs, ignoring interrupted syscalls */
    readfds= openfds;
    while ((nready= select(NOFILE, &readfds, 0, 0)) < 0 && errno == EINTR);
    if (nready < 0) ERR("select");
    /* ship the available data in the appropriate direction(s) */
    if (FD_ISSET(0, &readfds) && ship(source, sock)) { /* EOF on source */
      FD_CLR(0, &openfds);
      shutdown(sock, 1); /* inform peer that input side is shut down */
    }
    if (FD_ISSET(sock, &readfds) && ship(sock, sink)) { /* connection reset */
      FD_CLR(sock, &openfds);
      shutdown(sock, 2); /* shut down both sides of socket */
    }
  }
}
```

```c
/* here we go...
 */
main(argc, argv)
char **argv;
{
  int sock;

  /* try to connect to the Smalltalk execution server */
  if ((sock= tcp_connect(SMALLTALK_HOST, SMALLTALK_PORT)) < 0) {
    fprintf(stderr, "failed to connect to host: %s\n", SMALLTALK_HOST);
    exit(EX_UNAVAILABLE);
  }
  /* send the command line arguments (i.e. the command to be evaluated) */
  argv[argc]= 0;
  sendstrings(sock, argv+1);
  /* connect stdin and stdout to the Smalltalk process */
  to_from_duplex(fileno(stdin), fileno(stdout), sock);
  /* communication finished: close socket and terminate successfully */
  if (close(sock)) ERR("close");
  return EX_OK;
}
```

Appendix B: The Smalltalk Server for C Clients

```
'From Objectworks\Smalltalk(R), Release 4.1 of 15 April 1992 on 2
August 1995 at 5:52:29 am'!

Object subclass: #ExampleServer
	instanceVariableNames: 'stdio '
	classVariableNames: 'ClientProcesses ServerProcess '
	poolDictionaries: ''
	category: 'ESUG-UnixServer'!

ExampleServer comment: 'Expression evaluation server for Unix.  See
the ESUG summer school ''95 proceedings for explanation.

This class, and all its methods, are Copyright (c) 1995 Ian Piumarta
(ian.piumarta@inria.fr).  Use of this class, and all its methods, is
hereby granted for any purpose provided that this paragraph appears in
any software, and its supporting documentation, that is in any way
based on, or otherwise derived from, this class or its methods.  '!


!ExampleServer methodsFor: 'initialize-release'!

setStdio: anExternalStream
	stdio := anExternalStream! !

!ExampleServer methodsFor: 'accessing'!

stdio
	^stdio! !

!ExampleServer methodsFor: 'services-capitalisation'!

capitalise
	stdio nextPutAll: stdio contents asUppercase!

capitalise: aStream
	aStream nextPutAll: aStream contents asUppercase! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

ExampleServer class
	instanceVariableNames: ''!


!ExampleServer class methodsFor: 'class initialization'!

initialize
	"Initialise the server. If the class has been previously initialised, we may
	need to kill an existing server and its clients."

	ServerProcess notNil ifTrue: [self killServer].
	ClientProcesses isNil
		ifTrue: [ClientProcesses := IdentitySet new]
		ifFalse: [self killClientProcesses].
	(ObjectMemory dependents includes: self)
		ifFalse: [ObjectMemory addDependent: self]! !

!ExampleServer class methodsFor: 'instance creation'!

newClientStdio: anExternalStream
	^super new setStdio: anExternalStream! !
```

```
!ExampleServer class methodsFor: 'server process control'!

killServer
	"Kill the server process."

	ServerProcess isNil ifTrue: [self error: 'there is no server process to kill'].
	ServerProcess terminate.
	ServerProcess := nil!

resumeServer
	"Resume the server process."

	ServerProcess isNil ifTrue: [self error: 'there is no server process to resume'].
	ServerProcess resume.!

startServer
	"Start a new server process."

	ServerProcess isNil ifFalse: [self error: 'a server process already exists'].
	ServerProcess := [self serverLoop] newProcess.
	ServerProcess priority: Processor userInterruptPriority.
	ServerProcess resume.!

suspendServer
	"Suspend the server process."

	ServerProcess isNil ifTrue: [self error: 'there is no server process to suspend'].
	ServerProcess suspend.! !

!ExampleServer class methodsFor: 'client process control'!

killClientProcesses
	"Terminate all the client processes."

	ClientProcesses do: [ :process |
		process terminate.
		ClientProcesses remove: process]! !

!ExampleServer class methodsFor: 'server loop'!

dispatchClientOn: clientSocket

	| stdio command client process |
	stdio := clientSocket readAppendStream.
	command := stdio through: Character cr.
	client := self newClientStdio: stdio.
	ClientProcesses add: (process :=
		[
			[	self evaluateSafely: command for: client.
				stdio flush; close.
			] valueNowOrOnUnwindDo:
				[ClientProcesses remove: process.
				clientSocket close]
		] newProcess).
	process priority: Processor userSchedulingPriority.
	process resume.!
```

```
serverLoop
	"Create a socket and listen for connections. When a connection
	arrives, create a stream connected to the client's stdin/out, read
	the command to be evaluated, and create a client process
	connected to the stdio stream to #perform: the command."

	| serverSocket |
	serverSocket := UnixSocketAccessor
				family: (SocketAddress domainCodeFromName: #afInet)
				type: UnixSocketAccessor sockStream.
	[	serverSocket
			setOptionsLevel: UnixSocketAccessor solSocket
				name: UnixSocketAccessor soReuseaddr
				value: 1;
			bindTo: (IPSocketAddress
						hostAddress: IPSocketAddress thisHost
						port: self serverPort)
	] valueOnUnwindDo:
		[serverSocket close.
		self error: 'failed to create server socket'].

	[	serverSocket listenFor: 1.
		[true] whileTrue: [self dispatchClientOn: serverSocket accept]
	] valueNowOrOnUnwindDo: [serverSocket close]! !

!ExampleServer class methodsFor: 'configuration constants'!

serverPort
	"Return the port number on which the server listens for connections"

	^1966! !

!ExampleServer class methodsFor: 'snapshot'!

returnFromSnapshot
	self killClientProcesses.
	ServerProcess isNil ifFalse: [self killServer; startServer].!

update: aspect with: parameter from: sender
	(sender == ObjectMemory and: [aspect == #returnFromSnapshot])
		ifTrue: [self returnFromSnapshot]! !

!ExampleServer class methodsFor: 'private'!

evaluateSafely: command for: client
	| unique |
	unique := 'unique'.
	^Smalltalk errorSignal
		handle: [ :ex || message |
			message := ex originator == unique
				ifTrue: ['*** syntax error']
				ifFalse: ['*** ' ,  ex errorString , Character cr asSymbol ,
						'*** for receiver: ' , ex originator printString].
			client stdio nextPutAll: message; cr; flush.
			ex returnWith: '']
		do: [Compiler new evaluate: command in: nil to: client notifying: unique ifFail: nil]! !


ExampleServer initialize!
```

Appendix C: C Primitive Definitions for DBM Database

```
/* dbm_prims.c --- primitives supporting class DbmConnection
 *
 * Copyright (c) 1995 Ian Piumarta (ian.piumarta@inria.fr).  Use of
 * this example code is hereby granted for any purpose provided that
 * this notice appears in any software, and its supporting
 * documentation, that is in any way based on, or otherwise derived
 * from, this example.
 */

#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <ndbm.h>
#include "userprim.h"

/* length to truncate and values */
#define MAXLEN  256

/* private failure codes */
#define UPEdbmCouldNotOpen (-32)
#define UPEdbmKeyNotFound (-33)
#define UPEdbmCouldNotStore (-34)

char *versionString= "Objectworks\\Smalltalk(R) Release 4.1 with DBM";
char *resourceClassName = "St80";

/* cpok nonsense */
void UPInitLinkRegistry() {}


/*** some useful stuff ***/


/* convert a smalltalk string into a dbm datum
 *
 * note: the dptr component of the result points to static storage
 * and must be duplicated before a subsequent call to this function.
 */
static datum stringToDatum(string)
upHandle string;
{
  upInt size;
  static upChar cString[MAXLEN];
  datum dtm;

  UPmustBeString(string);
  size= UPcopySTtoCstring(string, cString, MAXLEN, 1);
  cString[size]= '\0';
  dtm.dptr= cString;
  dtm.dsize= strlen(cString);
  return dtm;
}

/* convert a dbm datum into a freshly allocated smalltalk string
 */
static upHandle datumToString(dtm)
datum dtm;
{
  upHandle string= UPallocString(' ', dtm.dsize);
  UPcopyCtoSTstring(string, dtm.dptr, dtm.dsize, 1);
  return string;
}


/* convert an instance of DbmConnection into a DBM *
 */
static DBM *getDbm(dbm_conn)
upHandle dbm_conn;
{
  return (DBM *)UPSTtoCint(UPinstVarAt(dbm_conn, 1));
}


/*** primitives ***/


/* open the named dbm database, where:
 *
 *        recv = the class DbmConnection
 *        name = a String containing the database name
 *
 * returns an instance of DbmConnection with cookie set
 */
void dbmOpen(recv, name)
upHandle recv, name;
{
  DBM *db;
  upChar cname[MAXLEN];
  upInt size;
  upHandle instance;

  UPmustBeString(name);
  size= UPcopySTtoCstring(name, cname, MAXLEN, 1);
  cname[size]= '\0';
```

```
  if ((db= dbm_open(cname, O_RDWR | O_CREAT, 0644)) < 0)
    UPfail(UPEdbmCouldNotOpen);
  /* allocate an instance of DbmConnection */
  instance= UPallocFsObject(recv);
  /* set the cookie */
  UPinstVarAtPut(instance, 1, UPCtoSTint((int)db));
  UPreturnHandle(instance);
}

/* close the database associated with the receiver, where:
 *
 *        recv = an instance of DbmConnection
 *
 * returns nil
 */
void dbmClose(recv)
upHandle recv;
{
  DBM *db= getDbm(recv);
  dbm_close(db);
  UPinstVarAtPut(recv, 1, UPCtoSTint(0)); /* cookie= 0 => closed */
  UPreturnNil();
}


/* fetch a value from a database, where:
 *
 *        recv = an instance of DbmConnection
 *        key  = the Smalltalk String containing the key
 *
 * returns a String containing the value, or fails (key not found)
 */
void dbmFetch(recv, key)
upHandle recv, key;
{
  DBM *db;
  datum dbmkey, dbmvalue;
  upInt size;

  db= getDbm(recv);
  dbmkey= stringToDatum(key);
  if ((dbmvalue= dbm_fetch(db, dbmkey)).dptr)
    UPreturnHandle(datumToString(dbmvalue));
  UPfail(UPEdbmKeyNotFound);
}
```

```
/* store value at key, where:
 *
 *        recv = an instance of DbmConnection
 *        key = Smalltalk String containing the key
 *        value = Smalltalk String containing the value to store
 *
 * returns the value String
 */
void dbmStore(recv, key, value)
upHandle recv, key, value;
{
  DBM *db;
  datum ckey, cvalue;
  int status;

  db= getDbm(recv);
  ckey= stringToDatum(key);
  ckey.dptr= strdup(ckey.dptr); /* copy static storage */
  cvalue= stringToDatum(value);
  status= dbm_store(db, ckey, cvalue, DBM_REPLACE);
  free(ckey.dptr); /* release storage from strdup() */
  if (status < 0)
    UPfail(UPEdbmCouldNotStore);
  UPreturnHandle(value);
}


/* remove a key from the database, where:
 *
 *        recv = an instance of DbmConnection
 *        key = Smalltalk String containing the key to be removed
 *
 * returns self
 */
void dbmDelete(recv, key)
upHandle recv, key;
{
  DBM *db;
  datum ckey, cvalue;

  db= getDbm(recv);
  ckey= stringToDatum(key);
  if (dbm_delete(db, ckey) < 0)
    UPfail(UPEdbmKeyNotFound);
  UPreturnHandle(recv);
}
```

```
/* return the "first" key in the database, where:
 *
 *        recv = an instance of DbmConnection
 *
 * returns nil
 */
void dbmFirstKey(recv)
upHandle recv;
{
  DBM *db;
  datum ckey;
  upHandle string;

  db= getDbm(recv);
  if ((ckey= dbm_firstkey(db)).dptr) {
    string= datumToString(ckey);
    UPreturnHandle(string);
  }
  UPreturnNil();
}

/* return the key "after" the given key, where:
 *
 *        recv = an instance of DbmConnection
 *        key = the "predecessor" of the result
 *
 * returns a String containing the "next" key, or nil (no more keys)
 */
void dbmNextKey(recv, key)
upHandle recv, key;
{
  DBM *db;
  datum inkey, outkey;
  upHandle string;

  db= getDbm(recv);
  inkey= stringToDatum(key);
  if ((outkey= dbm_nextkey(db, inkey)).dptr) {
    string= datumToString(outkey);
    UPreturnHandle(string);
  }
  UPreturnNil();
}
```

```
/*** initialisation ***/


/* install the primitives
 */
char *UPinstall()
{
    UPaddPrimitive(10001, dbmOpen, 1);
    UPaddPrimitive(10002, dbmClose, 0);
    UPaddPrimitive(10003, dbmFetch, 1);
    UPaddPrimitive(10004, dbmStore, 2);
    UPaddPrimitive(10005, dbmDelete, 1);
    UPaddPrimitive(10006, dbmFirstKey, 0);
    UPaddPrimitive(10007, dbmNextKey, 1);
    return "dbm primitives";
}


/* main() starts the VM with no other actions...
 */
main(argc, argv)
char **argv;
{
  return st80Main(argc, argv, (char **)0);
}
```

Appendix D: Smalltalk DBM Interface Class

```
’From Objectworks\Smalltalk(R), Release 4.1 of 15 April 1992 on 2
August 1995 at 6:05:27 am’!

Object subclass: #DbmConnection
    instanceVariableNames: ’cookie ’
    classVariableNames: ’OpenConnections ’
    poolDictionaries: ”
    category: ’ESUG-DBM’!

DbmConnection comment: ’DBM Database Connection.  See the ESUG summer
school ”95 proceedings for explanation.

This class, and all its methods, are Copyright (c) 1995 Ian Piumarta
(ian.piumarta@inria.fr).  Use of this class, and all its methods, is
hereby granted for any purpose provided that this paragraph appears in
any software, and its supporting documentation, that is in any way
based on, or otherwise derived from, this class or its methods.  ’!


!DbmConnection methodsFor: ’closing’!

close
    OpenConnections removeKey: self cookie.
    ^self primClose! !

!DbmConnection methodsFor: ’accessing’!

fetch: key
    <primitive: 10003>
    (key isKindOf: String)
        ifTrue: [Object notFoundSignal raise]
        ifFalse: [self error: ’DbmConnections are keyed by Strings.’]!

store: key value: value
    <primitive: 10004>
    self error: ’store:value failed with code ’ , self failureCode printString! !

!DbmConnection methodsFor: ’removing’!

delete: key
    <primitive: 10005>
    Object notFoundSignal raise! !

!DbmConnection methodsFor: ’enumerating’!

do: aBlock
    | key |
    key := self firstKey.
    key notNil ifTrue: [aBlock value: key].
    [(key := self nextKey) isNil]
        whileFalse: [aBlock value: key].
    ^nil!

firstKey
    <primitive: 10006>
    self error: ’firstKey failed with code ’ , self failureCode printString!

nextKey
    <primitive: 10007>
    self error: ’nextKey failed with code ’ , self failureCode printString! !

!DbmConnection methodsFor: ’converting’!

asDictionary
```

```
	| dict |
	dict := Dictionary new.
	self do: [ :key | dict at: key put: (self fetch: key)].
	^dict! !

!DbmConnection methodsFor: 'finalization'!

finalize
	"my clone has expired."
	self primClose! !

!DbmConnection methodsFor: 'private'!

cookie
	^cookie!

failureCode
	<primitive: 9999>!

primClose
	<primitive: 10002>! !
"– – – – – – – – – – – – – – – – – – "!

DbmConnection class
	instanceVariableNames: ''!


!DbmConnection class methodsFor: 'class initialization'!

initialize
	(ObjectMemory dependents includes: self)
		ifFalse: [ObjectMemory addDependent: self].
	OpenConnections isNil
		ifTrue:	[OpenConnections := WeakDictionary new]! !

!DbmConnection class methodsFor: 'instance creation'!

open: aDatabase
	| connection |
	connection := self primOpen: aDatabase.
	OpenConnections at: connection cookie put: connection.
	^connection! !

!DbmConnection class methodsFor: 'snapshot'!

returnFromSnapshot
	OpenConnections isNil ifFalse:
		[OpenConnections keys do:
			[ :key | OpenConnections removeKey: key]]!

update: aspect with: parameter from: sender
	(sender == ObjectMemory and: [aspect == #returnFromSnapshot])
		ifTrue: [self returnFromSnapshot]! !

!DbmConnection class methodsFor: 'private'!

failureCode
	<primitive: 9999>!

primOpen: aDatabase
	<primitive: 10001>
	(aDatabase isKindOf: String)
		ifTrue: [OSErrorHolder nonexistentSignal raise]
		ifFalse: [self error: 'database name must be a String']! !

DbmConnection initialize!
```